

The L^AT_EX3 Sources

The L^AT_EX3 Project*

July 9, 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.

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

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	3
II	The <code>l3bootstrap</code> package: Bootstrap code	5
3	Using the <code>l3TeX</code> modules	5
III	The <code>l3names</code> package: Namespace for primitives	6
4	Setting up the <code>l3TeX</code> programming language	6
IV	The <code>l3basics</code> package: Basic definitions	7
5	No operation functions	7
6	Grouping material	8
7	Control sequences and functions	8
7.1	Defining functions	8
7.2	Defining new functions using primitive parameter text	9
7.3	Defining new functions using the signature	12
7.4	Copying control sequences	15
7.5	Deleting control sequences	16
7.6	Showing control sequences	16
7.7	Converting to and from control sequences	16
8	Using or removing tokens and arguments	18
8.1	Selecting tokens from delimited arguments	20
8.2	Decomposing control sequences	21

9	Predicates and conditionals	21
9.1	Tests on control sequences	23
9.2	Testing string equality	24
9.3	Engine-specific conditionals	24
9.4	Primitive conditionals	25
10	Internal kernel functions	27
V	The l3expan package: Argument expansion	27
11	Defining new variants	28
12	Methods for defining variants	28
13	Introducing the variants	29
14	Manipulating the first argument	30
15	Manipulating two arguments	32
16	Manipulating three arguments	33
17	Unbraced expansion	34
18	Preventing expansion	34
19	Internal functions and variables	36
VI	The l3prg package: Control structures	36
20	Defining a set of conditional functions	37
21	The boolean data type	40
22	Boolean expressions	41
23	Logical loops	43

24 Switching by case	44
25 Producing n copies	45
26 Detecting T _E X's mode	46
27 Internal programming functions	47
28 Experimental programmings functions	48
VII The l3quark package: Quarks	48
29 Defining quarks	49
30 Quark tests	50
31 Recursion	51
32 Internal quark functions	52
VIII The l3token package: Token manipulation	52
33 All possible tokens	53
34 Character tokens	54
35 Generic tokens	57
36 Converting tokens	58
37 Token conditionals	58
38 Peeking ahead at the next token	64
39 Decomposing a macro definition	67
40 Experimental token functions	68

IX The <code>l3int</code> package: Integers	69
41 Integer expressions	69
42 Creating and initialising integers	70
43 Setting and incrementing integers	71
44 Using integers	73
45 Integer expression conditionals	73
46 Integer expression loops	74
47 Formatting integers	75
48 Converting from other formats to integers	78
49 Viewing integers	79
50 Constant integers	79
51 Scratch integers	80
52 Internal functions	80
 X The <code>l3skip</code> package: Dimensions and skips	 82
53 Creating and initialising <code>dim</code> variables	82
54 Setting <code>dim</code> variables	82
55 Utilities for dimension calculations	84
56 Dimension expression conditionals	85
57 Dimension expression loops	85
58 Using <code>dim</code> expressions and variables	87

59 Viewing dim variables	87
60 Constant dimensions	88
61 Scratch dimensions	88
62 Creating and initialising skip variables	88
63 Setting skip variables	89
64 Skip expression conditionals	90
65 Using skip expressions and variables	90
66 Viewing skip variables	91
67 Constant skips	91
68 Scratch skips	91
69 Creating and initialising muskip variables	92
70 Setting muskip variables	92
71 Using muskip expressions and variables	93
72 Inserting skips into the output	94
73 Viewing muskip variables	94
74 Internal functions	94
75 Experimental skip functions	95
 XI The l3tl package: Token lists	 95
76 Creating and initialising token list variables	96
77 Adding data to token list variables	97

78 Modifying token list variables	99
79 Reassigning token list category codes	100
80 Reassigning token list character codes	101
81 Token list conditionals	101
82 Mapping to token lists	103
83 Using token lists	105
84 Working with the content of token lists	105
85 The first token from a token list	107
86 Viewing token lists	108
87 Constant token lists	108
88 Scratch token lists	109
89 Experimental token list functions	109
XII The l3seq package: Sequences and stacks	110
90 Creating and initialising sequences	110
91 Appending data to sequences	112
92 Recovering items from sequences	113
93 Modifying sequences	114
94 Sequence conditionals	115
95 Mapping to sequences	116
96 Sequences as stacks	118

97 Viewing sequences	119
98 Experimental sequence functions	119
99 Internal sequence functions	122
 XIII The l3clist package: Comma separated lists	 123
100 Creating and initialising comma lists	123
101 Appending items to comma lists	125
102 Comma lists as stacks	126
103 Using comma lists	128
104 Modifying comma lists	128
105 Comma list conditionals	129
106 Mapping to comma lists	130
107 Comma lists as stacks	132
108 Viewing comma lists	133
109 Experimental comma list functions	133
 XIV The l3prop package: Property lists	 134
110 Creating and initialising property lists	135
111 Adding entries to property lists	136
112 Recovering values from property lists	137
113 Modifying property lists	138

114	Property list conditionals	138
115	Mapping to property lists	139
116	Viewing property lists	140
117	Experimental property list functions	141
118	Internal property list functions	142
XV	The <code>l3box</code> package: Boxes	142
119	Creating and initialising boxes	143
120	Using boxes	144
121	Measuring and setting box dimensions	145
122	Box conditionals	146
123	The last box inserted	147
124	Constant boxes	147
125	Scratch boxes	147
126	Viewing box contents	147
127	Horizontal mode boxes	148
128	Vertical mode boxes	150
129	Primitive box conditionals	152
XVI	The <code>l3io</code> package: Input–output operations	153
130	Opening and closing streams	154

131	Writing to files	154
132	Wrapping lines in output	156
133	Reading from files	156
134	Internal input–output functions	158
XVII	The <code>libmsg</code> package: Messages	158
135	Creating new messages	159
136	Contextual information for messages	160
137	Issuing messages	161
138	Redirecting messages	163
139	Low-level message functions	164
140	Kernel-specific functions	165
XVIII	The <code>libkeyval</code> package: Key–value parsing	167
141	Parsing key–value lists	167
XIX	The <code>libkeys</code> package: Key–value interfaces	168
142	Creating keys	170
143	Sub-dividing keys	174
144	Multiple choice keys	175
145	Setting keys	176
146	Utility functions for keys	176

XX	The <code>l3file</code> package: File operations	177
147	File operation functions	177
148	Internal file functions	178
XXI	The <code>l3fp</code> package: Floating-point operations	179
149	Floating-point variables	180
150	Conversion of floating point values to other formats	182
151	Rounding floating point values	183
152	Floating-point conditionals	183
153	Unary floating-point operations	184
154	Floating-point arithmetic	185
155	Floating-point power operations	186
156	Exponential and logarithm functions	187
157	Trigonometric functions	187
158	Constant floating point values	188
159	Notes on the floating point unit	189
XXII	The <code>l3luatex</code> package: LuaTeX-specific functions	189
160	Breaking out to Lua	189
161	Category code tables	190
XXIII	Implementation	191

162	Bootstrap code	192
162.1	Format-specific code	192
162.2	Package-specific code	193
162.3	Dealing with package-mode meta-data	194
162.4	The <code>\pdfstrcmp</code> primitive in \TeX	198
162.5	Engine requirements	198
162.6	The \LaTeX 3 code environment	199
163	names implementation	201
164	basics implementation	211
164.1	Renaming some \TeX primitives (again)	211
164.2	Defining functions	213
164.3	Selecting tokens	214
164.4	Gobbling tokens from input	216
164.5	Conditional processing and definitions	216
164.6	Dissecting a control sequence	221
164.7	Exist or free	223
164.8	Defining and checking (new) functions	225
164.9	More new definitions	228
164.10	Copying definitions	230
164.11	Undefining functions	230
164.12	Defining functions from a given number of arguments	231
164.13	Using the signature to define functions	233
164.14	Checking control sequence equality	235
164.15	Diagnostic wrapper functions	235
164.16	Engine specific definitions	236
164.17	Doing nothing functions	237
164.18	String comparisons	237
164.19	Deprecated functions	237

1653	expan implementation	238
165.1	General expansion	238
165.2	Hand-tuned definitions	242
165.3	Definitions with the automated technique	245
165.4	Last-unbraced versions	246
165.5	Preventing expansion	247
165.6	Defining function variants	248
165.7	Variants which cannot be created earlier	251
1663	prg implementation	251
166.1	Defining a set of conditional functions	251
166.2	The boolean data type	251
166.3	Boolean expressions	253
166.4	Logical loops	259
166.5	Switching by case	260
166.6	Producing n copies	262
166.7	Detecting \TeX 's mode	265
166.8	Internal programming functions	266
166.9	Experimental programmings functions	267
166.10	Deprecated functions	270
1673	quark implementation	270
1683	token implementation	273
168.1	Character tokens	274
168.2	Generic tokens	276
168.3	Token conditionals	277
168.4	Peeking ahead at the next token	286
168.5	Decomposing a macro definition	291
168.6	Experimental token functions	292
168.7	Deprecated functions	292

169	int implementation	295
169.1	Integer expressions	295
169.2	Creating and initialising integers	297
169.3	Setting and incrementing integers	298
169.4	Using integers	299
169.5	Integer expression conditionals	300
169.6	Integer expression loops	303
169.7	Formatting integers	304
169.8	Converting from other formats to integers	309
169.9	Viewing integer	313
169.10	Constant integers	313
169.11	Scratch integers	314
169.12	Registers for earlier modules	314
169.13	Deprecated functions	314
170	skip implementation	315
170.1	Length primitives renamed	316
170.2	Creating and initialising dim variables	316
170.3	Setting dim variables	316
170.4	Utilities for dimension calculations	318
170.5	Dimension expression conditionals	318
170.6	Dimension expression loops	320
170.7	Using dim expressions and variables	321
170.8	Viewing dim variables	321
170.9	Constant dimensions	322
170.10	Scratch dimensions	322
170.11	Creating and initialising skip variables	322
170.12	Setting skip variables	323
170.13	Skip expression conditionals	323
170.14	Using skip expressions and variables	324
170.15	Inserting skips into the output	324
170.16	Viewing skip variables	325

170.1	Constant skips	325
170.1	Scratch skips	325
170.1	Creating and initialising muskip variables	325
170.2	Setting muskip variables	326
170.2	Using muskip expressions and variables	327
170.2	Viewing muskip variables	327
170.2	Experimental skip functions	327
171	3tl implementation	328
171.1	Functions	328
171.2	Adding to token list variables	329
171.3	Reassigning token list category codes	331
171.4	Reassigning token list character codes	333
171.5	Modifying token list variables	333
171.6	Token list conditionals	335
171.7	Mapping to token lists	339
171.8	Using token lists	340
171.9	Working with the contents of token lists	341
171.10	The first token from a token list	343
171.1	Viewing token lists	344
171.1	Constant token lists	344
171.1	Scratch token lists	345
171.1	Experimental functions	345
171.1	Deprecated functions	347
172	3seq implementation	348
172.1	Allocation and initialisation	349
172.2	Appending data to either end	350
172.3	Modifying sequences	350
172.4	Sequence conditionals	352
172.5	Recovering data from sequences	353
172.6	Mapping to sequences	355

172.7Sequence stacks	358
172.8Viewing sequences	358
172.9Experimental functions	359
172.10Deprecated interfaces	363
173clist implementation	364
173.1Allocation and initialisation	364
173.2Appending items to comma lists	366
173.3Comma lists as stacks	366
173.4Using comma lists	367
173.5Modifying comma lists	368
173.6Comma list conditionals	369
173.7Mapping to comma lists	371
174Viewing comma lists	373
174.1Experimental functions	373
174.2Deprecated interfaces	375
1753prop implementation	376
175.1Allocation and initialisation	376
175.2Accessing data in property lists	377
175.3Property list conditionals	381
175.4Mapping to property lists	382
175.5Viewing property lists	383
175.6Experimental functions	384
175.7Deprecated interfaces	386
1763box implementation	387
176.1Creating and initialising boxes	387
176.2Measuring and setting box dimensions	388
176.3Using boxes	389
176.4Box conditionals	389
176.5The last box inserted	390

176.6	Constant boxes	390
176.7	Scratch boxes	390
176.8	Viewing box contents	391
176.9	Horizontal mode boxes	391
176.10	Vertical mode boxes	392
177	io implementation	394
177.1	Primitives	394
177.2	Variables and constants	394
177.3	Stream management	395
177.4	Deferred writing	401
177.5	Immediate writing	401
177.6	Hard-wrapping lines based on length	402
177.7	Special characters for writing	405
177.8	Reading input	406
177.9	Deprecated functions	407
178	msg implementation	407
179	Creating messages	408
179.1	Messages: support functions and text	408
179.2	Showing messages: low level mechanism	409
179.3	Displaying messages	412
179.4	Kernel-specific functions	417
179.5	Deprecated functions	422
180	keyval implementation	423
180.1	Deprecated functions	426
181	keys Implementation	427
181.1	Constants and variables	427
181.2	The key defining mechanism	428
181.3	Turning properties into actions	430

181.4	Creating key properties	433
181.5	Setting keys	436
181.6	Utilities	438
181.7	Messages	439
182	file implementation	440
183	fp Implementation	444
183.1	Constants	444
183.2	Variables	445
183.3	Parsing numbers	448
183.4	Internal utilities	451
183.5	Operations for fp variables	453
183.6	Transferring to other types	458
183.7	Rounding numbers	464
183.8	Unary functions	467
183.9	Basic arithmetic	469
183.10	Arithmetic for internal use	478
183.11	Trigonometric functions	485
183.12	Exponent and logarithm functions	498
183.13	Tests for special values	520
183.14	Floating-point conditionals	520
183.15	Messages	526
184	luatex implementation	527
184.1	Category code tables	528
	Index	532

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.

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.

1.1 Terminological inexactitude

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

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in \TeX ’s stomach” (if you are familiar with the \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_p: *</code>
<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 \LaTeX 2_ϵ or plain \TeX . In these cases, the text will include an extra “ **\TeX hackers note**” section:

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

The normal description text.

\TeX hackers note: Detail for the experienced \TeX or \LaTeX 2_ϵ programmer. In this case, it would point out that this function is the \TeX primitive `\string`.

Part II

The l3bootstrap package

Bootstrap code

3 Using the \LaTeX 3 modules

The modules documented in `source3` are designed to be used on top of \LaTeX 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 \LaTeX 3 format, but work in this area is incomplete and not included in this documentation at present.

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

`\ExplSyntaxOn`
`\ExplSyntaxOff` `\ExplSyntaxOn` *$\langle code \rangle$* `\ExplSyntaxOff`

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

`\ExplSyntaxNamesOn`
`\ExplSyntaxNamesOff` `\ExplSyntaxNamesOn` *$\langle code \rangle$* `\ExplSyntaxNamesOff`

The `\ExplSyntaxOn` function switches to a category code régime in which the colon (`:`) and underscore (`_`) are treated as “letters”, thus allowing access to the names of code

functions and variables. In contrast to `\ExplSyntaxOn`, using `\ExplSyntaxNamesOn` does not cause spaces to be ignored. The `\ExplSyntaxNamesOff` reverts to the document category code régime.

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

These functions act broadly in the same way as the $\text{\LaTeX 2}_{\epsilon}$ kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as $\text{\LaTeX 2}_{\epsilon}$ provides in turning on `\makeatletter` within package and class code.)

<code>\GetIdInfo</code>	<code>\RequirePackage{l3names}</code>
	<code>\GetIdInfo \$Id: <SVN info field> \$ {<description>}</code>

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

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or alike are loaded with usual $\text{\LaTeX 2}_{\epsilon}$ category codes and the \LaTeX 3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

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

Part III

The l3names package Namespace for primitives

4 Setting up the \LaTeX 3 programming language

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

- defines new names for all \TeX primitives;

- switches to the category code regime for programming;
- provides support settings for building the code as a T_EX format.

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

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

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

`\pdftex_...` Introduced by pdfT_EX;

`\xetex_...` Introduced by X_YT_EX;

`\luatex_...` Introduced by LuaT_EX.

Part IV

The l3basics package

Basic definitions

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

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

5 No operation functions

`\prg_do_nothing: *` `\prg_do_nothing:`

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

`\scan_stop:` `\scan_stop:`

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

6 Grouping material

<code>\group_begin:</code>	<code>\group_begin:</code>
<code>\group_end:</code>	<code>\group_end:</code>

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

<code>\group_insert_after:N</code>	<code>\group_insert_after:N</code> <i><token></i>
------------------------------------	---

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

7 Control sequences and functions

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

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

7.1 Defining functions

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

7.2 Defining new functions using primitive parameter text

<code>\cs_new:Npn</code>
<code>\cs_new:cpn</code>
<code>\cs_new:Npx</code>
<code>\cs_new:cpx</code>

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

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

<code>\cs_new_nopar:Npn</code>
<code>\cs_new_nopar:cpn</code>
<code>\cs_new_nopar:Npx</code>
<code>\cs_new_nopar:cpx</code>

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

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

<code>\cs_new_protected:Npn</code>
<code>\cs_new_protected:cpn</code>
<code>\cs_new_protected:Npx</code>
<code>\cs_new_protected:cpx</code>

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

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

<code>\cs_new_protected_nopar:Npn</code>
<code>\cs_new_protected_nopar:cpn</code>
<code>\cs_new_protected_nopar:Npx</code>
<code>\cs_new_protected_nopar:cpx</code>

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

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

result if the $\langle function \rangle$ is already defined.

$\backslash cs_set:Npn$ $\backslash cs_set:cpn$ $\backslash cs_set:Npx$ $\backslash cs_set:cpx$	$\backslash cs_set:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	---

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

$\backslash cs_set_nopar:Npn$ $\backslash cs_set_nopar:cpn$ $\backslash cs_set_nopar:Npx$ $\backslash cs_set_nopar:cpx$	$\backslash cs_set_nopar:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	--

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

$\backslash cs_set_protected:Npn$ $\backslash cs_set_protected:cpn$ $\backslash cs_set_protected:Npx$ $\backslash cs_set_protected:cpx$	$\backslash cs_set_protected:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	--

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

$\backslash cs_set_protected_nopar:Npn$ $\backslash cs_set_protected_nopar:cpn$ $\backslash cs_set_protected_nopar:Npx$ $\backslash cs_set_protected_nopar:cpx$	$\backslash cs_set_protected_nopar:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	---

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

$\langle function \rangle$ will not expand within an x-type argument.

$\backslash cs_gset:Npn$ $\backslash cs_gset:cpn$ $\backslash cs_gset:Npx$ $\backslash cs_gset:cpx$	$\backslash cs_gset:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	--

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

$\backslash cs_gset_nopar:Npn$ $\backslash cs_gset_nopar:cpn$ $\backslash cs_gset_nopar:Npx$ $\backslash cs_gset_nopar:cpx$	$\backslash cs_gset_nopar:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	---

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

$\backslash cs_gset_protected:Npn$ $\backslash cs_gset_protected:cpn$ $\backslash cs_gset_protected:Npx$ $\backslash cs_gset_protected:cpx$	$\backslash cs_gset_protected:Npn \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$
--	---

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

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

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

7.3 Defining new functions using the signature

<code>\cs_new:Nn</code>
<code>\cs_new:cn</code>
<code>\cs_new:Nx</code>
<code>\cs_new:cX</code>

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

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

<code>\cs_new_nopar:Nn</code>
<code>\cs_new_nopar:cn</code>
<code>\cs_new_nopar:Nx</code>
<code>\cs_new_nopar:cX</code>

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

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

<code>\cs_new_protected:Nn</code>
<code>\cs_new_protected:cn</code>
<code>\cs_new_protected:Nx</code>
<code>\cs_new_protected:cX</code>

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

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

<code>\cs_new_protected_nopar:Nn</code>
<code>\cs_new_protected_nopar:cn</code>
<code>\cs_new_protected_nopar:Nx</code>
<code>\cs_new_protected_nopar:cX</code>

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

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

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

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

`\cs_set:Nn $\langle function \rangle$ { $\langle code \rangle$ }`

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

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

`\cs_set_nopar:Nn $\langle function \rangle$ { $\langle code \rangle$ }`

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

<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 $\langle function \rangle$ { $\langle code \rangle$ }`

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

<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>

`\cs_set_protected_nopar:Nn $\langle function \rangle$ { $\langle code \rangle$ }`

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

used the $\langle parameters \rangle$ absorbed cannot contain $\backslash par$ tokens. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to $\langle function \rangle$ is restricted to the current T_EX group level.

$\backslash cs_gset:Nn$
$\backslash cs_gset:cn$
$\backslash cs_gset:Nx$
$\backslash cs_gset:cx$

 $\backslash cs_gset:Nn \langle function \rangle \{ \langle code \rangle \}$

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

$\backslash cs_gset_nopar:Nn$
$\backslash cs_gset_nopar:cn$
$\backslash cs_gset_nopar:Nx$
$\backslash cs_gset_nopar:cx$

 $\backslash cs_gset_nopar:Nn \langle function \rangle \{ \langle code \rangle \}$

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

$\backslash cs_gset_protected:Nn$
$\backslash cs_gset_protected:cn$
$\backslash cs_gset_protected:Nx$
$\backslash cs_gset_protected:cx$

 $\backslash cs_gset_protected:Nn \langle function \rangle \{ \langle code \rangle \}$

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

$\backslash cs_gset_protected_nopar:Nn$
$\backslash cs_gset_protected_nopar:cn$
$\backslash cs_gset_protected_nopar:Nx$
$\backslash cs_gset_protected_nopar:cx$

 $\backslash cs_gset_protected_nopar:Nn \langle function \rangle \{ \langle code \rangle \}$

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

is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to $\langle function \rangle$ is global.

<code>\cs_generate_from_arg_count:NNnn</code> <code>\cs_generate_from_arg_count:cNnn</code>	<code>\cs_generate_from_arg_count:NNnn</code> $\langle function \rangle$ $\langle creator \rangle$ $\langle number \rangle$ $\langle code \rangle$
--	---

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

7.4 Copying control sequences

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

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

<code>\cs_new_eq:NN</code> <code>\cs_new_eq:Nc</code> <code>\cs_new_eq:cN</code> <code>\cs_new_eq:cc</code>	<code>\cs_new_eq:NN</code> $\langle cs\ 1 \rangle$ $\langle cs\ 2 \rangle$
--	--

Creates $\langle control\ sequence\ 1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence\ 2 \rangle$ at the point where `\cs_new_eq:NN` is executed. The two control sequences may subsequently be altered without affecting the copy. The assignment of a meaning to $\langle control\ sequence\ 1 \rangle$ is global.

<code>\cs_set_eq:NN</code> <code>\cs_set_eq:Nc</code> <code>\cs_set_eq:cN</code> <code>\cs_set_eq:cc</code>	<code>\cs_set_eq:NN</code> $\langle cs\ 1 \rangle$ $\langle cs\ 2 \rangle$
--	--

Sets $\langle control\ sequence\ 1 \rangle$ to have the same meaning as $\langle control\ sequence\ 2 \rangle$ at the point where `\cs_set_eq:NN` is executed. The two control sequences may subsequently be altered without affecting the copy. The assignment of a meaning to $\langle control\ sequence\ 1 \rangle$ is restricted to the current \TeX group level.

<code>\cs_gset_eq:NN</code> <code>\cs_gset_eq:Nc</code> <code>\cs_gset_eq:cN</code> <code>\cs_gset_eq:cc</code>	<code>\cs_gset_eq:NN</code> $\langle cs\ 1 \rangle$ $\langle cs\ 2 \rangle$
--	---

Globally sets $\langle control\ sequence\ 1 \rangle$ to have the same meaning as $\langle control\ sequence\ 2 \rangle$ at the

point where `\cs_gset_eq:NN` is executed. The two control sequences may subsequently be altered without affecting the copy. The assignment of a meaning to $\langle control\ sequence\ 1 \rangle$ is *not* restricted to the current $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ group level: the assignment is global.

7.5 Deleting control sequences

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

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

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

7.6 Showing control sequences

<code>\cs_meaning:N</code>	<code>\star</code>	<code>\cs_meaning:N</code>	$\langle control\ sequence \rangle$
<code>\cs_meaning:c</code>	<code>\star</code>		

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

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

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

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

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$ hackers note: This is the $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ primitive `\show`.

7.7 Converting to and from control sequences

<code>\use:c</code>	<code>\star</code>	<code>\use:c</code>	$\{ \langle control\ sequence\ name \rangle \}$
---------------------	--------------------	---------------------	---

Converts the given $\langle control\ sequence\ name \rangle$ into a single control sequence token. This process requires two expansions. The content for $\langle control\ sequence\ name \rangle$ may be literal material or from other expandable functions. The $\langle control\ sequence\ name \rangle$ must, when fully expanded, consist of character tokens which are not active: typically, they will be of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these. As an example, both

```
\use:c { a b c }
```

and

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

would be equivalent to

```
\abc
```

after two expansions of `\use:c`.

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

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

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

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

and

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

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

TeXhackers note: These are the TeX primitives `\csname` and `\endcsname`.

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

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

Converts the given *<control sequence>* into a series of characters with category code 12 (other), except spaces, of category code 10. The sequence will *not* include the current escape token, *cf.* `\token_to_str:N`. Full expansion of this function requires a variable number of expansion steps (either 3 or 4), and so an **f**- or **x**-type expansion will be required to convert the *<control sequence>* to a sequence of characters in the input stream.

8 Using or removing tokens and arguments

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

<code>\use:n</code>	★	<code>\use:n</code>	{⟨ <i>group</i> ₁ ⟩}
<code>\use:nn</code>	★	<code>\use:nn</code>	{⟨ <i>group</i> ₁ ⟩} {⟨ <i>group</i> ₂ ⟩}
<code>\use:nnn</code>	★	<code>\use:nnn</code>	{⟨ <i>group</i> ₁ ⟩} {⟨ <i>group</i> ₂ ⟩} {⟨ <i>group</i> ₃ ⟩}
<code>\use:nnnn</code>	★	<code>\use:nnn</code>	{⟨ <i>group</i> ₁ ⟩} {⟨ <i>group</i> ₂ ⟩} {⟨ <i>group</i> ₃ ⟩} {⟨ <i>group</i> ₄ ⟩}

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

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

will result in the input stream containing

```
abc { def }
```

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

<code>\use_i:nn</code>	★	<code>\use_i:nn</code>	{⟨ <i>group</i> ₁ ⟩} {⟨ <i>group</i> ₂ ⟩}
<code>\use_ii:nn</code>	★		

These functions will absorb two groups and leave only the first or the second in the input stream. The braces surrounding the arguments will be removed as part of this process. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<code>\use_i:nnn</code>	★	<code>\use_i:nnn</code>	{⟨ <i>group</i> ₁ ⟩} {⟨ <i>group</i> ₂ ⟩} {⟨ <i>group</i> ₃ ⟩}
<code>\use_ii:nnn</code>	★		
<code>\use_iii:nnn</code>	★		

These functions will absorb three groups and leave only of these in the input stream. The braces surrounding the arguments will be removed as part of this process. The

category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

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

`\use_i:nnnn {⟨group1⟩} {⟨group2⟩} {⟨group3⟩} {⟨group4⟩}`

These functions will absorb four groups and leave only of these in the input stream. The braces surrounding the arguments will be removed as part of this process. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

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

`\use_i_ii:nnn {⟨group1⟩} {⟨group2⟩} {⟨group3⟩}`

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

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

will result in the input stream containing

```
abc { def }
```

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

<code>\use_none:n</code>	★
<code>\use_none:nn</code>	★
<code>\use_none:nnn</code>	★
<code>\use_none:nnnn</code>	★
<code>\use_none:nnnnn</code>	★
<code>\use_none:nnnnnn</code>	★
<code>\use_none:nnnnnnn</code>	★
<code>\use_none:nnnnnnnn</code>	★
<code>\use_none:nnnnnnnnn</code>	★

`\use_none:n {⟨group1⟩}`

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

<code>\use:x</code>

`\use:x {⟨expandable tokens⟩}`

Fully expands the *⟨expandable tokens⟩* and inserts the result into the input stream at the current location.

8.1 Selecting tokens from delimited arguments

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

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

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

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

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

`\use_i_delimit_by_q_nil:nw {<inserted tokens>} <balanced text> \q_nil`

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

<code>\use_i_after_fi:nw</code>	★
<code>\use_i_after_else:nw</code>	★
<code>\use_i_after_or:nw</code>	★
<code>\use_i_after_orelse:nw</code>	★

`{<inserted tokens>} \fi:`
`{<inserted tokens>} \else:`
`<balanced text> \fi:`
`{<inserted tokens>} \or:`
`<balanced text> \fi:`
`{<inserted tokens>} \or: or \else:`
`<balanced text> \fi:`

Absorb the *<balanced text>*, if appropriate, delimited by the function name given. The *<inserted tokens>* are then placed in the input stream after the delimiter. Thus for example

```
\use_i_after_fi:nw { some tokens } \fi:
```

will leave

```
\fi: some tokens
```

in the input stream for further processing. See the discussion of the primitive \TeX conditionals for more detail on `\else:`, `\fi:` and `\or:`.

8.2 Decomposing control sequences

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

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

`\cs_get_function_name:N *` `\cs_get_function_name:NN` $\langle function \rangle$

Splits the $\langle function \rangle$ into the name (*i.e.* the part before the colon) and the signature (*i.e.* after the colon). The $\langle name \rangle$ is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

`[EXP]\cs_get_function_signature:N` `\cs_get_function_signature:NN` $\langle function \rangle$

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

`E` `XP]_split_function:NN` `\cs_split_function:NN` $\langle function \rangle$ $\langle processor \rangle$

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

`\cs_to_str:N *` `\cs_to_str:N` $\{\langle control sequence \rangle\}$

Converts the given $\langle control sequence \rangle$ into a series of characters with category code 12 (other), except spaces, of category code 10. The sequence will *not* include the current escape token, *cf.* `\token_to_str:N`. Full expansion of this function requires a variable number of expansion steps (either 3 or 4), and so an **f**- or **x**-type expansion will be required to convert the $\langle control sequence \rangle$ to a sequence of characters in the input stream.

9 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
  \langle true code \rangle
else:
  \langle false code \rangle
\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
} {\langle true code \rangle} {\langle false code \rangle}
```


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 $\text{\LaTeX 2}_{\epsilon}$. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

<code>\c_true_bool</code> <code>\c_false_bool</code>	Constants that represent <code>true</code> and <code>false</code> , respectively. Used to implement predicates.
---	---

9.1 Tests on control sequences

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

Compares the definition of two $\langle control\ sequence \rangle$ and is logically `true` if the two are the same. 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>\cs_if_exist_p:N *</code> <code>\cs_if_exist:NTF *</code> <code>\cs_if_exist_p:c *</code> <code>\cs_if_exist:cTF *</code>	<code>\cs_if_exist_p:N</code> $\langle control\ sequence \rangle$ <code>\cs_if_exist:NTF</code> $\langle control\ sequence \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$
--	--

Tests whether the $\langle control\ sequence \rangle$ is currently defined (whether as a function or another control sequence type). Any valid definition of $\langle control\ sequence \rangle$ will evaluate as `true`. 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>\cs_if_free_p:N *</code> <code>\cs_if_free:NTF *</code> <code>\cs_if_free_p:c *</code> <code>\cs_if_free:cTF *</code>	<code>\cs_if_free_p:N</code> $\langle control\ sequence \rangle$ <code>\cs_if_free:NTF</code> $\langle control\ sequence \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$
--	--

Tests whether the $\langle control\ sequence \rangle$ is currently free to be defined. This test will be

²If defined using the interface provided.

false if the *control sequence* currently exists (as defined by `\cs_if_exist: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. The logical truth of the test is left in the input stream by the predicate version.

9.2 Testing string equality

<code>\str_if_eq_p:nn *</code>	
<code>\str_if_eq:nnTF *</code>	
<code>\str_if_eq_p:Vn *</code>	
<code>\str_if_eq:VnTF *</code>	
<code>\str_if_eq_p:on *</code>	
<code>\str_if_eq:onTF *</code>	
<code>\str_if_eq_p:no *</code>	
<code>\str_if_eq:noTF *</code>	
<code>\str_if_eq_p:nV *</code>	
<code>\str_if_eq:nVTF *</code>	
<code>\str_if_eq_p:VV *</code>	
<code>\str_if_eq:VVTF *</code>	
<code>\str_if_eq_p:xx *</code>	<code>\str_if_eq_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq:xxTF *</code>	<code>\str_if_eq:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</code>

Compares the two *token lists* on a character by character basis, and is **true** if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_p:xx { abc } { \tl_to_str:n { abc } }
```

is logically **true**. 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. All versions of these functions are fully expandable (including those involving an **x**-type expansion).

9.3 Engine-specific conditionals

<code>\luatex_if_engine:TF *</code>	<code>\luatex_if_luatex:TF {<true code>} {<false code>}</code>
-------------------------------------	--

Detects if the document is being compiled using Lua_T_EX. 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.

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

Detects if the document is being compiled using pdf \TeX . 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.

<code>\xetex_if_engine:TF</code> *

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

Detects if the document is being compiled using X \LaTeX . 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.

<code>\c_luatex_is_engine_bool</code>
<code>\c_pdftex_is_engine_bool</code>
<code>\c_xetex_is_engine_bool</code>

Boolean versions of the engine conditionals, for use in predicate tests.

9.4 Primitive conditionals

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

`\if_true: $\langle true code \rangle$ \else: $\langle false code \rangle$ \fi:
\if_false: $\langle true code \rangle$ \else: $\langle false code \rangle$ \fi:
\reverse_if:N $\langle primitive conditional \rangle$`

`\if_true:` always executes $\langle true code \rangle$, while `\if_false:` always executes $\langle false code \rangle$. `\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.

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

<code>\if_meaning:w</code> *

`\if_meaning:w $\langle arg_1 \rangle$ $\langle arg_2 \rangle$ $\langle true code \rangle$ \else: $\langle false code \rangle$ \fi:`

`\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> \star <code>\if_charcode:w</code> \star <code>\if_catcode:w</code> \star	<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_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> \star	<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> \star	<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> \star <code>\if_cs_exist:w</code> \star	<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> $\langle tokens \rangle$ <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> \star <code>\if_mode_vertical:</code> \star <code>\if_mode_math:</code> \star <code>\if_mode_inner:</code> \star	<code>\if_mode_horizontal:</code> $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
---	--

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

10 Internal kernel functions

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

`\chk_if_exist_cs:N <cs>`

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

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

`\chk_if_free_cs:N <cs>`

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

<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 inside \mathbf{x} -type expansions.

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` \mathfrak{C} .

TeXhackers note: These prefixes are the primitives `\global`, `\long`, and `\protected`.

Part V

The l3expan package

Argument expansion

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

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

11 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` was not 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.

12 Methods for defining variants

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

This function is used to define argument-specifier variants of the *<parent control sequence>* for L^AT_EX3 code-level macros. The *<parent control sequence>* is first separated into the *<base name>* and *<original argument specifier>*. The comma-separated list of *<variant argument specifiers>* is then used to define variants of the *<original argument specifier>* where these are not already defined. For each *<variant>* given, a function is created which

will expand its arguments as detailed and pass them to the *parent control sequence*. So for example

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

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

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

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

13 Introducing the variants

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

- Arguments that might need expansion should come first in the list of arguments to make processing faster.
- Arguments that should consist of single tokens should come first.
- Arguments that need full expansion (*i.e.*, are denoted with `x`) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type will not work correctly in arguments that are itself subject to `x` expansion.
- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing which is not fast. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, with possible trailing `N` or `n`, which are not expanded.

The `V` type returns the value of a register, which can be one of `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in `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 `\tl var`. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straightforward approach is

```
\tl_set:No \l_tmpb_tl {\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_tmpb_tl` and not `\cs_set_eq:NwN \aaa = \blurb` as we probably wanted. Using `\tl_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\tl_set:Nf \l_tmpb_tl {\cs_set_eq:Nc \aaa { b \l_tmpa_tl b } }
```

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

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

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

14 Manipulating the first argument

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

`\exp_args:No *`

`\exp_args:No <function> {\<tokens>} {\<tokens_2>} ...`

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

`\exp_args:Nc *`
`\exp_args:cc *`

`\exp_args:Nc <function> {\<tokens>} {\<tokens_2>} ...`

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). The

result is inserted into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

The `:cc` variant constructs the $\langle function \rangle$ name in the same manner as described for the $\langle tokens \rangle$.

`\exp_args:Nv` \star `\exp_args:Nv` $\langle function \rangle$ $\langle variable \rangle$ $\{\langle tokens_2 \rangle\}$...

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

`\exp_args:Nv` \star `\exp_args:Nv` $\langle function \rangle$ $\{\langle tokens \rangle\}$ $\{\langle tokens_2 \rangle\}$...

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

`\exp_args:Nf` \star `\exp_args:Nf` $\langle function \rangle$ $\{\langle tokens \rangle\}$ $\{\langle tokens_2 \rangle\}$...

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token or space is found, and the result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

`\exp_args:Nx` `\exp_args:Nx` $\langle function \rangle$ $\{\langle tokens \rangle\}$ $\{\langle tokens_2 \rangle\}$...

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$ second. The result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

15 Manipulating two arguments

<code>\exp_args:NNo</code>	<code>*</code>
<code>\exp_args:NNc</code>	<code>*</code>
<code>\exp_args:NNv</code>	<code>*</code>
<code>\exp_args:NNV</code>	<code>*</code>
<code>\exp_args:NNf</code>	<code>*</code>
<code>\exp_args:Nco</code>	<code>*</code>
<code>\exp_args:Ncf</code>	<code>*</code>
<code>\exp_args:Ncc</code>	<code>*</code>
<code>\exp_args:NVV</code>	<code>*</code>

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

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

<code>\exp_args:Nno</code>	<code>*</code>
<code>\exp_args:NnV</code>	<code>*</code>
<code>\exp_args:Nnf</code>	<code>*</code>
<code>\exp_args:Noo</code>	<code>*</code>
<code>\exp_args:Noc</code>	<code>*</code>
<code>\exp_args:Nff</code>	<code>*</code>
<code>\exp_args:Nfo</code>	<code>*</code>
<code>\exp_args:Nnc</code>	<code>*</code>

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

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

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

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

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

16 Manipulating three arguments

<code>\exp_args:NNNo *</code> <code>\exp_args:NNNV *</code> <code>\exp_args:Nccc *</code> <code>\exp_args:NcNc *</code> <code>\exp_args:NcNo *</code> <code>\exp_args:Ncco *</code>	<code>\exp_args:NNNo</code> $\langle token1 \rangle$ $\langle token2 \rangle$ $\langle token3 \rangle$ $\{ \langle tokens \rangle \}$
--	--

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

<code>\exp_args:NNoo *</code> <code>\exp_args:NNno *</code> <code>\exp_args:Nnno *</code> <code>\exp_args:Nnnc *</code> <code>\exp_args:Nooo *</code>	<code>\exp_args:NNNo</code> $\langle token1 \rangle$ $\langle token2 \rangle$ $\langle token3 \rangle$ $\{ \langle tokens \rangle \}$
---	--

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

<code>\exp_args:NNnx</code> <code>\exp_args: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>	<code>\exp_args:NNnx</code> $\langle token1 \rangle$ $\langle token2 \rangle$ $\langle tokens1 \rangle$ $\{ \langle tokens_2 \rangle \}$
---	---

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

17 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:NNNo</code>
<code>\exp_last_unbraced:Nfo</code>
<code>\exp_last_unbraced:NNNV</code>
<code>\exp_last_unbraced:NNNo</code>

`\exp_last_unbraced:Nno <token> <tokens1> <tokens2>`

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the the results in the input stream, with the last argument not surrounded by the usual braces. Of these, `\exp_last_unbraced:Nfo` needs special (slower) processing.

<code>\exp_last_two_unbraced:Noo *</code>

`\exp_last_two_unbraced:Noo <token> <tokens1> {<tokens2>}`

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

<code>\exp_after:wN *</code>

`\exp_after:wN <token1> <token2>`

Carries out a single expansion of `<token2>` prior to expansion of `<token1>`. If `<token2>` is a `TEX` primitive, it will be executed rather than expanded, while if `<token2>` has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that `<token1>` may be *any* single token, including group-opening and -closing tokens (`{` or `}`" assuming normal `TEX` category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

T_EXhackers note: This is the `TEX` primitive `\expandafter` renamed.

18 Preventing expansion

<code>\exp_not:N</code>

`\exp_not:N <token>`

Prevents expansion of the `<token>` in a context where it would otherwise be expanded, for example an `x`-type argument.

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

`\exp_not:c` `\exp_not:c {⟨tokens⟩}`

Expands the `⟨tokens⟩` until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.

`\exp_not:n` `\exp_not:n {⟨tokens⟩}`

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

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

`\exp_not:V` `\exp_not:V ⟨variable⟩`

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

`\exp_not:v` `\exp_not:v {⟨tokens⟩}`

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

`\exp_not:o` `\exp_not:o {⟨tokens⟩}`

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

`\exp_not:f *` `\exp_not:f ⟨tokens⟩`

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

`\exp_stop_f: *` `\function:f ⟨tokens⟩ \exp_stop_f: ⟨more tokens⟩`

This function terminates an `f`-type expansion. Thus if a function `\function:f` starts an `f`-type expansion and all of `⟨tokens⟩` are expandable `\exp_stop:f` will terminate the expansion of tokens even if `⟨more tokens⟩` are also expandable. The function itself is an implicit space token. Inside an `x`-type expansion, it will retain its form, but when typeset it produces the underlying space (`_`).

19 Internal functions and variables

`\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</code> $\langle variable \rangle$
--	--

These functions evaluates a $\langle variable \rangle$ as part of a `V` or `v` expansion (respectively), preceded by `\c_zero` which stops the expansion of a previous `\tex_romannumeral:D`. A $\langle variable \rangle$ might exist as one of two things: a parameter-less non-long, non-protected macro or a built-in T_EX register such as `\count`.

<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. These names do *not* conform to the general L^AT_EX3 approach as this makes them more readily visible in the log and so forth.

<code>\cs_generate_internal_variant:n</code>	<code>\cs_generate_internal_variant:n</code> $\langle arg spec \rangle$
--	---

Tests if the function `\exp_args:N` $\langle arg spec \rangle$ exists, and defines it if it does not. The $\langle arg spec \rangle$ should be a series of one or more of the letters `N`, `c`, `n`, `o`, `V`, `v`, `f` and `x`.

Part VI

The l3prg package

Control structures

Conditional processing in L^AT_EX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead

in the input stream. After processing the input, a *state* is returned. The typical states returned are $\langle true \rangle$ 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

20 Defining a set of conditional functions

<code>\prg_new_conditional:Npnn</code>	<code>\prg_set_conditional:Npnn \langle name \rangle:\langle arg spec \rangle</code>
<code>\prg_set_conditional:Npnn</code>	<code>\langle parameters \rangle \{ \langle conditions \rangle \} \{ \langle code \rangle \}</code>
<code>\prg_new_conditional:Nnn</code>	<code>\prg_set_conditional:Nnn \langle name \rangle:\langle arg spec \rangle</code>
<code>\prg_set_conditional:Nnn</code>	<code>\{ \langle conditions \rangle \} \{ \langle code \rangle \}</code>

These functions creates a family of conditionals using the same $\{ \langle code \rangle \}$ to perform the test created. The **new** version will check for existing definitions (*cf.* `\cs_new:Npn`) whereas the **set** version will not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of $\langle conditions \rangle$, which should be one or more of **p**, **T**, **F** and **TF**. The conditionals are then defined in the obvious way as:

- $\langle name \rangle_p:\langle arg spec \rangle$, a predicate function which will supply either a logical **true** or logical **false**. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome.
- $\langle name \rangle:\langle arg spec \rangle T$, a function with one more argument than the original $\langle arg spec \rangle$ demands. The $\langle true branch \rangle$ code in this additional argument will be left on the input stream only if the test is **true**.
- $\langle name \rangle:\langle arg spec \rangle F$, a function with one more argument than the original $\langle arg spec \rangle$ demands. The $\langle false branch \rangle$ code in this additional argument will be left on the input stream only if the test is **false**.
- $\langle name \rangle:\langle arg spec \rangle TF$, a function with two more argument than the original $\langle arg spec \rangle$ demands. The $\langle true branch \rangle$ code in the first additional argument will be left on the input stream if the test is **true**, while the $\langle false branch \rangle$ code in the second argument will be left on the input stream if the test is **false**.

The *code* of the test may use *parameters* as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the *argument specification* but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (cf. `\cs_new:Nn`, etc.). Within the *code*, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test. If *code* is expandable then `\prg_set_conditional:Npnn` will generate a family of conditionals which are also expandable. All of the functions are created globally.

An example can easily clarify matters here:

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

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF`, `\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.

<pre>\prg_new_protected_conditional:Npnn \prg_set_protected_conditional:Npnn \prg_new_protected_conditional:Nnn \prg_set_protected_conditional:Nnn</pre>	<pre>\prg_set_protected_conditional:Npnn \<name>:\<arg spec> \<parameters> \<conditions> {\<code>} \prg_set_protected_conditional:Nnn \<name>:\<arg spec> \<conditions> {\<code>}</pre>
--	---

These functions creates a family of conditionals using the same `{\code}` to perform the test created. The **new** version will check for existing definitions (cf. `\cs_new:Npn`) whereas the **set** version will not (cf. `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of *conditions*, which should be one or more of `T`, `F` and `TF`. The conditionals are then defined in the obvious way as:

- `\<name>:\<arg spec>T`, a function with one more argument than the original *arg spec* demands. The *true branch* code in this additional argument will be left on the input stream only if the test is **true**.
- `\<name>:\<arg spec>F`, a function with one more argument than the original *arg spec* demands. The *false branch* code in this additional argument will be left on the input stream only if the test is **false**.

- $\backslash\langle name \rangle:\langle arg\ spec \rangle TF$, a function with two more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in the first additional argument will be left on the input stream if the test is **true**, while the $\langle false\ branch \rangle$ code in the second argument will be left on the input stream if the test is **false**.

The $\langle code \rangle$ of the test may use $\langle parameters \rangle$ as specified by the second argument to $\backslash prg_set_conditional:Npn$: this should match the $\langle argument\ specification \rangle$ but this is not enforced. The Nnn versions infer the number of arguments from the argument specification given (cf. $\backslash cs_new:Nn$, etc.). Within the $\langle code \rangle$, the functions $\backslash prg_return_true:$ and $\backslash prg_return_false:$ are used to indicate the logical outcomes of the test. $\backslash prg_set_protected_conditional:Npn$ will generate a family of protected conditional functions, and so $\langle code \rangle$ does not need to be expandable. All of the functions are created globally.

$\backslash prg_new_eq_conditional:NN$ $\backslash prg_set_eq_conditional:NN$	$\backslash prg_new_eq_conditional:NN$ $\backslash\langle name1 \rangle:\langle arg\ spec1 \rangle\ \backslash\langle name2 \rangle:\langle arg\ spec2 \rangle$
--	---

These will set the definitions of the functions

- $\backslash\langle name1 \rangle_p:\langle arg\ spec1 \rangle$
- $\backslash\langle name1 \rangle:\langle arg\ spec1 \rangle T$
- $\backslash\langle name1 \rangle:\langle arg\ spec1 \rangle F$
- $\backslash\langle name1 \rangle:\langle arg\ spec1 \rangle TF$

equal to those for

- $\backslash\langle name2 \rangle_p:\langle arg\ spec2 \rangle$
- $\backslash\langle name2 \rangle:\langle arg\ spec2 \rangle T$
- $\backslash\langle name2 \rangle:\langle arg\ spec2 \rangle F$
- $\backslash\langle name2 \rangle:\langle arg\ spec2 \rangle TF$

In most cases, the two $\langle arg\ specs \rangle$ will be identical, although this is not enforced. In the case of the **new** function, a check is made for any existing definitions for $\langle name1 \rangle$. The functions are set globally.

$\backslash prg_return_true: \star$ $\backslash prg_return_false: \star$	$\backslash prg_return_true:$ $\backslash prg_return_false:$
---	---

These functions define the logical state at the end of a conditional. As such, they should appear within the code for a conditional statement generated by $\backslash prg_set_conditional:Npnn$, etc.

21 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 <boolean></code>
--	--

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

<code>\bool_set_false:N</code> <code>\bool_set_false:c</code>	<code>\bool_set_false:N <boolean></code>
--	--

Sets `<boolean>` logically **false** within the current `TeX` group.

<code>\bool_gset_false:N</code> <code>\bool_gset_false:c</code>	<code>\bool_sget_false:N <boolean></code>
--	---

Sets `<boolean>` logically **false** globally.

<code>\bool_set_true:N</code> <code>\bool_set_true:c</code>	<code>\bool_set_true:N <boolean></code>
--	---

Sets `<boolean>` logically **true** within the current `TeX` group.

<code>\bool_gset_true:N</code> <code>\bool_gset_true:c</code>	<code>\bool_gset_true:N <boolean></code>
--	--

Sets `<boolean>` logically **true** globally.

<code>\bool_set_eq:NN</code> <code>\bool_set_eq:cN</code> <code>\bool_set_eq:Nc</code> <code>\bool_set_eq:cc</code>	<code>\bool_set_eq:NN <boolean1> <boolean2></code>
--	--

Sets the content of $\langle boolean1 \rangle$ equal to that of $\langle boolean2 \rangle$. This assignment is restricted to the current \TeX group level.

$\backslash\text{bool_gset_eq:N}$
$\backslash\text{bool_gset_eq:cN}$
$\backslash\text{bool_gset_eq:Nc}$
$\backslash\text{bool_gset_eq:cc}$

 $\backslash\text{bool_gset_eq:N} \langle boolean1 \rangle \langle boolean2 \rangle$

Sets the content of $\langle boolean1 \rangle$ equal to that of $\langle boolean2 \rangle$. This assignment is global and so is not limited by the current \TeX group level.

$\backslash\text{bool_set:Nn}$
$\backslash\text{bool_set:cn}$

 $\backslash\text{bool_set:Nn} \langle boolean \rangle \{ \langle boolean \rangle \}$

Evaluates the $\langle boolean \text{ expression} \rangle$ as described for $\backslash\text{bool_if:n(TF)}$, and sets the $\langle boolean \rangle$ variable to the logical truth of this evaluation. This assignment is local.

$\backslash\text{bool_gset:Nn}$
$\backslash\text{bool_gset:cn}$

 $\backslash\text{bool_gset:Nn} \langle boolean \rangle \{ \langle boolean \rangle \}$

Evaluates the $\langle boolean \text{ expression} \rangle$ as described for $\backslash\text{bool_if:n(TF)}$, and sets the $\langle boolean \rangle$ variable to the logical truth of this evaluation. This assignment is global.

$\backslash\text{bool_if_p:N} \star$
$\backslash\text{bool_if:N\textit{TF}} \star$
$\backslash\text{bool_if_p:c} \star$
$\backslash\text{bool_if:c\textit{TF}} \star$

 $\backslash\text{bool_if_p:N} \{ \langle boolean \rangle \}$
 $\backslash\text{bool_if:N\textit{TF}} \{ \langle boolean \rangle \} \{ \langle true \text{ code} \rangle \} \{ \langle false \text{ code} \rangle \}$

Tests the current truth of $\langle boolean \rangle$, and continues expansion based on this result. 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.

$\backslash\text{l_tmpa_bool}$

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any \LaTeX -defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

$\backslash\text{g_tmpa_bool}$

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any \LaTeX -defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

22 Boolean expressions

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

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators $\&\&$, $||$ and $!$. In addition to this, parentheses can be used to isolate sub-expressions. For example,

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

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

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

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

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

will be **true** and will not evaluate $\int_compare_p:nNn \{ 1 \} = \{ \error \}$. The logical Not applies to the next single predicate or group. As shown above, this means that any predicates requiring an argument have to be given within parentheses. 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.

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

Function version of `!(<boolean expression>)` within a boolean expression.

\bool_xor_p:nn *	\bool_xor_p:nn {<boolean ₁ >} {<boolean ₁ >}
------------------	--

Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operator.

23 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_until_do:Nn *	\bool_until_do:Nn {<boolean>} {<code>}
\bool_until_do:cn *	

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

\bool_while_do:Nn *	\bool_while_do:Nn {<boolean>} {<code>}
\bool_while_do:cn *	

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

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

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

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

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

24 Switching by case

For cases where a number of cases need to be considered a family of case-selecting functions are available.

```

\prg_case_int:nnn
  {\test integer expression}
  {
    {\langle intexpr case1 \rangle} {\langle code case1 \rangle}
    {\langle intexpr case2 \rangle} {\langle code case2 \rangle}
    ...
    {\langle intexpr casen \rangle} {\langle code casen \rangle}
  }
\prg_case_int:nnn * {\langle else case \rangle}

```

This function evaluates the $\langle test\ integer\ expression \rangle$ and compares this in turn to each of the $\langle integer\ expression\ cases \rangle$. If the two are equal then the associated $\langle code \rangle$ is left in the input stream. If none of the tests are **true** then the **else** code will be left in the input stream. For example

```

\prg_case_int:nnn
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }  { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }

```

will leave “Medium” in the input stream.

```

\prg_case_dim:nnn
  {\test dimension expression}
  {
    {\langle dimeexpr case1 \rangle} {\langle code case1 \rangle}
    {\langle dimeexpr case2 \rangle} {\langle code case2 \rangle}
    ...
    {\langle dimeexpr casen \rangle} {\langle code casen \rangle}
  }
\prg_case_dim:nnn * {\langle else case \rangle}

```

This function evaluates the $\langle test\ dimension\ expression \rangle$ and compares this in turn to each of the $\langle dimension\ expression\ cases \rangle$. If the two are equal then the associated $\langle code \rangle$ is left in the input stream. If none of the tests are **true** then the **else** code will be left in

the input stream.

	<code>\prg_case_str:nnn</code>	<code>{⟨test string⟩}</code>
	<code>{</code>	
		<code>{⟨string case₁⟩} {⟨code case₁⟩}</code>
		<code>{⟨string case₂⟩} {⟨code case₂⟩}</code>
		<code>...</code>
<code>\prg_case_str:nnn *</code>		<code>{⟨string case_n⟩} {⟨code case_n⟩}</code>
<code>\prg_case_str:onn *</code>	<code>}</code>	
<code>\prg_case_str:xxn *</code>	<code>{⟨else case⟩}</code>	

This function compares the *⟨test string⟩* in turn with each of the *⟨string cases⟩*. If the two are equal (as described for `\str_if_eq:nnTF` then the associated *⟨code⟩* is left in the input stream. If none of the tests are **true** then the **else** code will be left in the input stream. The **xx** variant is fully expandable, in the same way as the underlying `\str_if_eq:xxTF` test.

	<code>\prg_case_tl:Nnn</code>	<code>⟨test token list variable⟩</code>
	<code>{</code>	
		<code>⟨token list variable case₁⟩ {⟨code case₁⟩}</code>
		<code>⟨token list variable case₂⟩ {⟨code case₂⟩}</code>
		<code>...</code>
<code>\prg_case_tl:Nnn *</code>		<code>⟨token list variable case_n⟩ {⟨code case_n⟩}</code>
<code>\prg_case_tl:cnn *</code>	<code>}</code>	
	<code>{⟨else case⟩}</code>	

This function compares the *⟨test token list variable⟩* in turn with each of the *⟨token list variable cases⟩*. If the two are equal (as described for `\tl_if_eq:nnTF` then the associated *⟨code⟩* is left in the input stream. If none of the tests are **true** then the **else** code will be left in the input stream.

25 Producing *n* copies

<code>\prg_replicate:nn *</code>	<code>\prg_replicate:nn {⟨integer expression⟩} {⟨tokens⟩}</code>
----------------------------------	--

Evaluates the *⟨integer expression⟩* (which should be zero or positive) and creates the resulting number of copies of the *⟨tokens⟩*. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

<code>\prg_stepwise_function:nnnN *</code>	<code>\prg_stepwise_function:nnnN {⟨initial value⟩} {⟨step⟩}</code>
	<code>{⟨final value⟩} {⟨function⟩}</code>

This function first evaluates the *⟨initial value⟩*, *⟨step⟩* and *⟨final value⟩*, all of which should be integer expressions. The *⟨function⟩* is then placed in front of each *⟨value⟩* from

the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). Thus $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set_nopar:Npn \my_func:n #1 { I~saw~#1 \\\ }
\prg_stepwise_function:nnnN { 1 } { 5 } { 1 } \my_func:n
```

would print

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

<code>\prg_stepwise_inline:nnnn</code>	<code>\prg_stepwise_inline:nnnn {$\langle initial\ value \rangle$} {$\langle step \rangle$} {$\langle final\ value \rangle$} {$\langle code \rangle$}</code>
--	--

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

<code>\prg_stepwise_variable:nnnn</code>	<code>\prg_stepwise_inline:nnnn {$\langle initial\ value \rangle$} {$\langle step \rangle$} {$\langle final\ value \rangle$} $\langle tl\ var \rangle$ {$\langle code \rangle$}</code>
--	---

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

26 Detecting T_EX's mode

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

Detects if T_EX is currently in horizontal mode. 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>\mode_if_inner_p: *</code> <code>\mode_if_inner:TF *</code>	<code>\mode_if_inner_p:</code> <code>\mode_if_inner:TF {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code>
--	---

Detects if T_EX is currently in inner mode. The branching versions then leave either $\langle true$

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>\mode_if_math:TF ★</code>	<code>\mode_if_math:TF {⟨true code⟩} {⟨false code⟩}</code>
---------------------------------	--

Detects if T_EX is currently in maths mode. 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.

<code>\mode_if_vertical_p: ★</code>	<code>\mode_if_vertical_p:</code>
<code>\mode_if_vertical:TF ★</code>	<code>\mode_if_vertical:TF {⟨true code⟩} {⟨false code⟩}</code>

Detects if T_EX is currently in vertical mode. 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.

27 Internal programming functions

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

These functions are used to enclose material in a T_EX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the & token inside `\tex_halign:D`. This is necessary to allow grabbing of tokens for testing purposes, as T_EX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

<code>\scan_align_safe_stop:</code>	<code>\scan_align_safe_stop:</code>
-------------------------------------	-------------------------------------

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

<code>\prg_variable_get_scope:N ★</code>	<code>\prg_variable_get_scope:N ⟨variable⟩</code>
--	---

Returns the scope (g for global, blank otherwise) for the $\langle variable \rangle$.

<code>\prg_variable_get_type:N *</code>	<code>\prg_variable_get_type:N $\langle variable \rangle$</code>
---	---

Returns the type of $\langle variable \rangle$ (tl, int, etc.)

28 Experimental programmings functions

<code>\prg_quicksort:n</code>	<code>\prg_quicksort:n { $\{\langle item_1 \rangle\}$ $\{\langle item_2 \rangle\}$... $\{\langle item_n \rangle\}$ }</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 {$\langle element \rangle$}</code> <code>\prg_quicksort_compare:nnTF {$\langle element_1 \rangle$} {$\langle element_2 \rangle$}</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 }
```

Part VII

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_`.

29 Defining quarks

`\quark_new:N` `\quark_new:N <quark>`

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

`\q_stop` Used as a marker for delimited arguments, such as

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

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

`\q_nil` Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to `\q_stop`, which is only ever used as a delimiter).

`\q_no_value` A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as `\prop_get:NnN` if there is no data to return.

30 Quark tests

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

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

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

\quark_if_nil_p:n *	\quark_if_nil_p:n { <i><token list></i> }
\quark_if_nil:nTF *	
\quark_if_nil_p:o *	
\quark_if_nil:oTF *	
\quark_if_nil_p:V *	
\quark_if_nil:VTF *	
	\quark_if_nil:nTF { <i><token list></i> } { <i><true code></i> } { <i><false code></i> }

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

\quark_if_no_value_p:N *	\quark_if_no_value_p:N <i><token></i>
\quark_if_no_value:NTF *	
\quark_if_no_value_p:c *	
\quark_if_no_value:cTF *	
	\quark_if_no_value:NTF <i><token></i> { <i><true code></i> } { <i><false code></i> }

Tests if the *<token>* is equal to \q_no_value. 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.

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

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

31 Recursion

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

`\q_recursion_tail` This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop` This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N` `\quark_if_recursion_tail_stop:N {\token}`

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

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o` `\quark_if_recursion_tail_stop:n {\tokens}`

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

`\quark_if_recursion_tail_stop_do:Nn` `\quark_if_recursion_tail_stop_do:nn {\token} {\insertion}`

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

`\quark_if_recursion_tail_stop_do:nn`
`\quark_if_recursion_tail_stop_do:on` `\quark_if_recursion_tail_stop_do:nn {\tokens}`
`\quark_if_recursion_tail_stop_do:on {\insertion}`

Tests if $\langle tokens \rangle$ consists of the single token `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

32 Internal quark functions

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

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

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

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

Part VIII

The l3token package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in 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.

33 All possible tokens

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

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three for the same internal operation of \TeX , namely the primitive testing the next two characters for equality of their character code. They behave identically in many situations. However, \TeX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below will take everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

It is easier to start by considering the possibilities for what a token looks like, in other words, how \TeX sees it from the point of view of delimited arguments. Two cases: the token is either a control sequence or a single character.

Control sequence	Character
Control word: an escape character followed by some letters.	

The token can be a control sequence, in other words, an escape character followed by some letters, or by a single non-letter character. Examples of this include `\begin`, `\;`, `\emph...`. The other case There are two cases. either the token is a single character, in which case it is associated with a category code:

34 Character tokens

<pre> \char_set_catcode_escape:N \char_set_catcode_group_begin:N \char_set_catcode_group_end:N \char_set_catcode_math_toggle:N \char_set_catcode_alignment:N \char_set_catcode_end_line:N \char_set_catcode_parameter:N \char_set_catcode_math_superscript:N \char_set_catcode_math_subscript:N \char_set_catcode_ignore:N \char_set_catcode_space:N \char_set_catcode_letter:N \char_set_catcode_other:N \char_set_catcode_active:N \char_set_catcode_comment:N \char_set_catcode_invalid:N </pre>	<pre> \char_make_letter:N <character> </pre>
---	--

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

```
\char_set_catcode_other:N \%
```

The assignment is local.

<pre> \char_set_catcode_escape:n \char_set_catcode_group_begin:n \char_set_catcode_group_end:n \char_set_catcode_math_toggle:n \char_set_catcode_alignment:n \char_set_catcode_end_line:n \char_set_catcode_parameter:n \char_set_catcode_math_superscript:n \char_set_catcode_math_subscript:n \char_set_catcode_ignore:n \char_set_catcode_space:n \char_set_catcode_letter:n \char_set_catcode_other:n \char_set_catcode_active:n \char_set_catcode_comment:n \char_set_catcode_invalid:n </pre>	<pre> \char_make_letter:n {{integer expression}} </pre>
---	---

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

```
\char_set_catcode:nn \char_set_catcode:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}
```

These functions set the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. The first $\langle integer expression \rangle$ is the character code and the second is the category code to apply. The setting applies within the current T_EX group. In general, the symbolic functions $\backslash\text{char_make_}\langle type \rangle$ should be preferred, but there are cases where these lower-level functions may be useful.

```
\char_value_catcode:n * \char_value_catcode:n {\langle integer expression \rangle}
```

Expands to the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

```
\char_show_value_catcode:n \char_show_value_catcode:n {\langle integer expression \rangle}
```

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

```
\char_set_lccode:nn \char_set_lccode:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}
```

This function set up the behaviour of $\langle character \rangle$ when found inside $\backslash\text{tl_to_lowercase:n}$, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the T_EX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:

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

The setting applies within the current T_EX group.

```
\char_value_lccode:n * \char_value_lccode:n {\langle integer expression \rangle}
```

Expands to the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

```
\char_show_value_lccode:n \char_show_value_lccode:n {\langle integer expression \rangle}
```

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

`\char_set_uccode:nn` `\char_set_uccode:nn { $\langle integer_1 \rangle$ } { $\langle integer_2 \rangle$ }`

This function set up the behaviour of $\langle character \rangle$ when found inside `\tl_to_uppercase:n`, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the T_EX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:

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

The setting applies within the current T_EX group.

`\char_value_uccode:n *` `\char_value_uccode:n { $\langle integer expression \rangle$ }`

Expands to the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

`\char_show_value_uccode:n` `\char_show_value_uccode:n { $\langle integer expression \rangle$ }`

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

`\char_set_mathcode:nn` `\char_set_mathcode:nn { $\langle integer_1 \rangle$ } { $\langle integer_2 \rangle$ }`

This function sets up the math code of $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T_EX group.

`\char_value_mathcode:n *` `\char_value_mathcode:n { $\langle integer expression \rangle$ }`

Expands to the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

`\char_show_value_mathcode:n` `\char_show_value_mathcode:n { $\langle integer expression \rangle$ }`

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

`\char_set_sfcode:nn` `\char_set_sfcode:nn { $\langle integer_1 \rangle$ } { $\langle integer_2 \rangle$ }`

This function sets up the space factor for the $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T_EX group.

`\char_value_sfcode:n *` `\char_value_sfcode:n { $\langle integer expression \rangle$ }`

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

`\char_show_value_sfcode:n` `\char_show_value_sfcode:n { $\langle integer expression \rangle$ }`

Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

35 Generic tokens

`\token_new:Nn` `\token_new:Nn $\langle token_1 \rangle$ { $\langle token_2 \rangle$ }`

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

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

`\c_catcode_letter_token`
`\c_catcode_other_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be

used other than for category code tests.

<code>\c_catcode_active_tl</code>	A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.
-----------------------------------	--

36 Converting tokens

<code>\token_to_meaning:N *</code>	<code>\token_to_meaning:N <token></code>
------------------------------------	--

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

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

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

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

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

37 Token conditionals

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

Tests if $\langle token \rangle$ has the category code of a begin group token (`{` when normal \TeX category codes are in force). 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. Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

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

Tests if $\langle token \rangle$ has the category code of an end group token ($\}$ when normal \TeX category codes are in force). 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. Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

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

Tests if $\langle token \rangle$ has the category code of a math shift token ($\$$ when normal \TeX category codes are in force). 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>\token_if_alignment_p:N *</code>	<code>\token_if_alignment_p:N <token></code>
<code>\token_if_alignment:NTF *</code>	<code>\token_if_alignment:NTF <token> {\true code}</code>
	<code>{\false code}</code>

Tests if $\langle token \rangle$ has the category code of an alignment token ($\&$ when normal \TeX category codes are in force). 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>\token_if_parameter_p:N *</code>	<code>\token_if_parameter_p:N <token></code>
<code>\token_if_parameter:NTF *</code>	<code>\token_if_parameter:NTF <token> {\true code}</code>
	<code>{\false code}</code>

Tests if $\langle token \rangle$ has the category code of a macro parameter token ($\#$ when normal \TeX category codes are in force). 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>\token_if_math_superscript_p:N *</code>	<code>\token_if_math_superscript_p:N <token></code>
<code>\token_if_math_superscript:NTF *</code>	<code>\token_if_math_superscript:NTF <token> {\true code}</code>
	<code>{\false code}</code>

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

$\backslash token_if_math_subscript_p:N \star$ $\backslash token_if_math_subscript:N\text{\textit{TF}} \star$	$\backslash token_if_math_subscript_p:N \langle token \rangle$ $\backslash token_if_math_subscript:N\text{\textit{TF}} \langle token \rangle \{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$
---	---

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

$\backslash token_if_space_p:N \star$ $\backslash token_if_space:N\text{\textit{TF}} \star$	$\backslash token_if_space_p:N \langle token \rangle$ $\backslash token_if_space:N\text{\textit{TF}} \langle token \rangle \{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$
---	---

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

$\backslash token_if_letter_p:N \star$ $\backslash token_if_letter:N\text{\textit{TF}} \star$	$\backslash token_if_letter_p:N \langle token \rangle$ $\backslash token_if_letter:N\text{\textit{TF}} \langle token \rangle \{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$
---	---

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

$\backslash token_if_other_p:N \star$ $\backslash token_if_other:N\text{\textit{TF}} \star$	$\backslash token_if_other_p:N \langle token \rangle$ $\backslash token_if_other:N\text{\textit{TF}} \langle token \rangle \{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$
---	---

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

$\backslash token_if_active_p:N \star$ $\backslash token_if_active:N\text{\textit{TF}} \star$	$\backslash token_if_active_p:N \langle token \rangle$ $\backslash token_if_active:N\text{\textit{TF}} \langle token \rangle \{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$
---	---

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

$\backslash token_if_eq_catcode_p:NN \star$ $\backslash token_if_eq_catcode:NN\textit{TF} \star$	$\backslash token_if_eq_catcode_p:NN \langle token1 \rangle \langle token2 \rangle$ $\backslash token_if_eq_catcode:NN\textit{TF} \langle token1 \rangle \langle token2 \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	--

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

$\backslash token_if_eq_charcode_p:NN \star$ $\backslash token_if_eq_charcode:NN\textit{TF} \star$	$\backslash token_if_eq_charcode_p:NN \langle token1 \rangle \langle token2 \rangle$ $\backslash token_if_eq_charcode:NN\textit{TF} \langle token1 \rangle \langle token2 \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	--

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

$\backslash token_if_eq_meaning_p:NN \star$ $\backslash token_if_eq_meaning:NN\textit{TF} \star$	$\backslash token_if_eq_meaning_p:NN \langle token1 \rangle \langle token2 \rangle$ $\backslash token_if_eq_meaning:NN\textit{TF} \langle token1 \rangle \langle token2 \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	--

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

$\backslash token_if_macro_p:N \star$ $\backslash token_if_macro:N\textit{TF} \star$	$\backslash token_if_macro_p:N \langle token \rangle$ $\backslash token_if_macro:N\textit{TF} \langle token \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}$
--	---

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

$\backslash token_if_cs_p:N \star$ $\backslash token_if_cs:N\textit{TF} \star$	$\backslash token_if_cs_p:N \langle token \rangle$ $\backslash token_if_cs:N\textit{TF} \langle token \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}$
--	---

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

$\backslash\text{token_if_expandable_p:N} \star$ $\backslash\text{token_if_expandable:N}\underline{TF} \star$	$\backslash\text{token_if_expandable_p:N} \langle token \rangle$ $\backslash\text{token_if_expandable:N}\underline{TF} \langle token \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

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

$\backslash\text{token_if_long_macro_p:N} \star$ $\backslash\text{token_if_long_macro:N}\underline{TF} \star$	$\backslash\text{token_if_long_macro_p:N} \langle token \rangle$ $\backslash\text{token_if_long_macro:N}\underline{TF} \langle token \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

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

$\backslash\text{token_if_protected_macro_p:N} \star$ $\backslash\text{token_if_protected_macro:N}\underline{TF} \star$	$\backslash\text{token_if_protected_macro_p:N} \langle token \rangle$ $\backslash\text{token_if_protected_macro:N}\underline{TF} \langle token \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

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

$\backslash\text{token_if_protected_long_macro_p:N} \star$ $\backslash\text{token_if_protected_long_macro:N}\underline{TF} \star$	$\backslash\text{token_if_protected_long_macro_p:N} \langle token \rangle$ $\backslash\text{token_if_protected_long_macro:N}\underline{TF} \langle token \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

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

$\backslash\text{token_if_chardef_p:N} \star$ $\backslash\text{token_if_chardef:N}\underline{TF} \star$	$\backslash\text{token_if_chardef_p:N} \langle token \rangle$ $\backslash\text{token_if_chardef:N}\underline{TF} \langle token \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

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

$\backslash\text{token_if_mathchardef_p:N} \star$ $\backslash\text{token_if_mathchardef:N}\underline{TF} \star$	$\backslash\text{token_if_mathchardef_p:N} \langle\text{token}\rangle$ $\backslash\text{token_if_mathchardef:N}\underline{TF} \langle\text{token}\rangle$ $\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$
---	---

Tests if the $\langle\text{token}\rangle$ is defined to be a mathchardef. The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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.

$\backslash\text{token_if_dim_register_p:N} \star$ $\backslash\text{token_if_dim_register:N}\underline{TF} \star$	$\backslash\text{token_if_dim_register_p:N} \langle\text{token}\rangle$ $\backslash\text{token_if_dim_register:N}\underline{TF} \langle\text{token}\rangle$ $\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$
---	---

Tests if the $\langle\text{token}\rangle$ is defined to be a dimension register. The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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.

$\backslash\text{token_if_int_register_p:N} \star$ $\backslash\text{token_if_int_register:N}\underline{TF} \star$	$\backslash\text{token_if_int_register_p:N} \langle\text{token}\rangle$ $\backslash\text{token_if_int_register:N}\underline{TF} \langle\text{token}\rangle$ $\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$
---	---

Tests if the $\langle\text{token}\rangle$ is defined to be an integer register. The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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.

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

Tests if the $\langle\text{token}\rangle$ is defined to be a skip register. The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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.

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

Tests if the $\langle\text{token}\rangle$ is defined to be a toks register (not used by L^AT_EX3). The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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>\token_if_primitive_p:N *</code>	<code>\token_if_primitive_p:N <token></code>
<code>\token_if_primitive:NTF *</code>	<code>\token_if_primitive:NTF <token></code> <code>{<true code>} {<false code>}</code>

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

38 Peeking ahead at the next token

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

<code>\peek_after:Nw</code>	<code>\peek_after:Nw <function> <token></code>
-----------------------------	--

Locally sets the test variable `\l_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ will remain in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\peek_gafter:Nw</code>	<code>\peek_gafter:Nw <function> <token></code>
------------------------------	---

Globally sets the test variable `\g_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ will remain in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\l_peek_token</code>	Token set by <code>\peek_after:Nw</code> and available for testing as described above.
----------------------------	--

<code>\g_peek_token</code>	Token set by <code>\peek_gafter:Nw</code> and available for testing as described above.
----------------------------	---

<code>\peek_catcode:NTF</code>	<code>\peek_catcode:NTF <test token> {<true code>} {<false code>}</code>
--------------------------------	--

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test$

token) (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *token* will be left in the input stream after the *true code* or *false code* (as appropriate to the result of the test).

<code>\peek_catcode_ignore_spaces:NTF</code>	<code>\peek_catcode_ignore_spaces:NNTF</code> <i>test token</i> <code>{true code} {false code}</code>
--	--

Tests if the next *token* in the input stream has the same category code as the *test token* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are ignored by the test and the *token* will be left in the input stream after the *true code* or *false code* (as appropriate to the result of the test).

<code>\peek_catcode_remove:NNTF</code>	<code>\peek_catcode_remove:NNTF</code> <i>test token</i> <code>{true code} {false code}</code>
--	---

Tests if the next *token* in the input stream has the same category code as the *test token* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *token* will be removed from the input stream if the test is true. The function will then place either the *true code* or *false code* in the input stream (as appropriate to the result of the test).

<code>\peek_catcode_remove_ignore_spaces:NNTF</code>	<code>\peek_catcode_remove_ignore_spaces:NNTF</code> <i>test token</i> <code>{true code} {false code}</code>
--	---

Tests if the next *token* in the input stream has the same category code as the *test token* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are ignored by the test and the *token* will be removed from the input stream if the test is true. The function will then place either the *true code* or *false code* in the input stream (as appropriate to the result of the test).

<code>\peek_charcode:NNTF</code>	<code>\peek_charcode:NNTF</code> <i>test token</i> <code>{true code} {false code}</code>
----------------------------------	---

Tests if the next *token* in the input stream has the same character code as the *test token* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *token* will be left in the input stream after the *true code* or *false code* (as appropriate to the result of the test).

<code>\peek_charcode_ignore_spaces:NNTF</code>	<code>\peek_charcode_ignore_spaces:NNTF</code> <i>test token</i> <code>{true code} {false code}</code>
--	---

Tests if the next *token* in the input stream has the same character code as the *test token* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are ignored by the

test and the $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

<code>\peek_charcode_remove:NTF</code>	<code>\peek_charcode_remove:NTF</code> $\langle test token \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	---

Tests if the next $\langle token \rangle$ in the input stream has the same character code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true code \rangle$ or $\langle false code \rangle$ in the input stream (as appropriate to the result of the test).

<code>\peek_charcode_remove_ignore_spaces:NTF</code>	<code>\peek_charcode_remove_ignore_spaces:NTF</code> $\langle test token \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	---

Tests if the next $\langle token \rangle$ in the input stream has the same character code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are ignored by the test and the $\langle token \rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true code \rangle$ or $\langle false code \rangle$ in the input stream (as appropriate to the result of the test).

<code>\peek_meaning:NTF</code>	<code>\peek_meaning:NTF</code> $\langle test token \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--------------------------------	--

Tests if the next $\langle token \rangle$ in the input stream has the same meaning as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

<code>\peek_meaning_ignore_spaces:NTF</code>	<code>\peek_meaning_ignore_spaces:NTF</code> $\langle test token \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	---

Tests if the next $\langle token \rangle$ in the input stream has the same meaning as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are ignored by the test and the $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

<code>\peek_meaning_remove:NTF</code>	<code>\peek_meaning_remove:NTF</code> $\langle test token \rangle$ $\{\langle true code \rangle\} \{\langle false code \rangle\}$
---------------------------------------	--

Tests if the next $\langle token \rangle$ in the input stream has the same meaning as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ will be removed from the input stream if the test is true. The function

will then place either the $\langle true\ code\rangle$ or $\langle false\ code\rangle$ in the input stream (as appropriate to the result of the test).

<code>\peek_meaning_remove_ignore_spaces:NTF</code>	<code>\peek_meaning_remove_ignore_spaces:NTF</code> $\langle test\ token\rangle$ $\{\langle true\ code\rangle\}\ \{\langle false\ code\rangle\}$
---	---

Tests if the next $\langle token\rangle$ in the input stream has the same meaning as the $\langle test\ token\rangle$ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are ignored by the test and the $\langle token\rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true\ code\rangle$ or $\langle false\ code\rangle$ in the input stream (as appropriate to the result of the test).

39 Decomposing a macro definition

These functions decompose TeX macros into their constituent parts: if the $\langle token\rangle$ passed is not a macro then no decomposition can occur. In the later case, all three functions leave `\scan_stop:` in the input stream.

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

If the $\langle token\rangle$ is a macro, this function will leave the primitive TeX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1 y #2 }
```

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

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

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

If the $\langle token\rangle$ is a macro, this function will leave the replacement text in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1~y #2 }
```

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

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

If the $\langle token \rangle$ is a macro, this function will leave the T_EX prefixes applicable in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1~y #2 }
```

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

40 Experimental token functions

```
\char_active_set:Npn  
\char_active_set:Npx \char_active_set:Npn \langle char \rangle \langle parameters \rangle {\langle code \rangle}
```

Makes $\langle char \rangle$ an active character to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ (`#1`, `#2`, *etc.*) will be replaced by those absorbed This definition is local to the current T_EX group.

```
\char_active_gset:Npn  
\char_active_gset:Npx \char_active_gset:Npn \langle char \rangle \langle parameters \rangle {\langle code \rangle}
```

Makes $\langle char \rangle$ an active character to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ (`#1`, `#2`, *etc.*) will be replaced by those absorbed This definition is global.

```
\char_active_set_eq:NN \char_active_set_eq:NN \langle char \rangle \langle function \rangle
```

Makes $\langle char \rangle$ an active character equivalent in meaning to the $\langle function \rangle$ (which may itself be an active character). This definition is local to the current T_EX group.

```
\char_active_gset_eq:NN \char_active_gset_eq:NN \langle char \rangle \langle function \rangle
```

Makes $\langle char \rangle$ an active character equivalent in meaning to the $\langle function \rangle$ (which may itself be an active character). This definition is global.

Part IX

The l3int package

Integers

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

41 Integer expressions

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

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

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

and

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

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

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

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

`\int_div_round:nn *` `\int_div_round:nn {⟨integer1⟩} {⟨integer2⟩}`

Evaluates the two *⟨integer expressions⟩* as described earlier, then calculates the result of

dividing the first value by the second, round any remainder. Note that this is identical to using `/` directly in an *integer expression*. The result is left in the input stream as a *integer denotation* after two expansions.

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

`\int_div_truncate:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}`

Evaluates the two *integer expressions* 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 *integer denotation* after two expansions.

<code>\int_max:nn *</code>
<code>\int_min:nn *</code>

`\int_max:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}`
`\int_min:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}`

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

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

`\int_mod:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}`

Evaluates the two *integer expressions* 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 *integer denotation* after two expansions.

42 Creating and initialising integers

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

`\int_new:N \langle integer \rangle`

Creates a new *integer* or raises an error if the name is already taken. The declaration is global. The *integer* will initially be equal to 0.

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

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

Creates a new constant *integer* or raises an error if the name is already taken. The

value of the $\langle integer \rangle$ will be set globally to the $\langle integer expression \rangle$.

<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_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.

43 Setting and incrementing integers

<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 <integer>`

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 <integer>`

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 <integer>`

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 <integer>`

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

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

`\int_set:Nn <integer> {<integer expression>}`

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 <integer> {<integer expression>}`

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 <integer> {<integer expression>}`

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 <integer> {<integer expression>}`

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

44 Using integers

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

`\int_use:N <integer>`

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

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

45 Integer expression conditionals

<code>\int_compare_p:nNn *</code>
<code>\int_compare:nNnTF *</code>

```

\int_compare_p:nNn
  {<expr1>} <relation> {<expr2>}
\int_compare:nNnTF
  {<expr1>} <relation> {<expr2>}
  {<true code>} {<false code>}

```

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

Equal	=
Greater than	>
Less than	<

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>\int_compare:nTF *</code>

```

\int_compare_p:n
  { <expr1> <relation> <expr2> }
\int_compare:nTF
  { <expr1> <relation> <expr2> }
  {<true code>} {<false code>}

```

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

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

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.

$\backslash\text{int_if_even_p:n} \star$	
$\backslash\text{int_if_even:nTF} \star$	
$\backslash\text{int_if_odd_p:n} \star$	$\backslash\text{int_if_odd_p:n} \{\langle integer\ expression \rangle\}$
$\backslash\text{int_if_odd:nTF} \star$	$\backslash\text{int_if_odd:nTF} \{\langle integer\ expression \rangle\}$
	$\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

This function first evaluates the $\langle integer\ expression \rangle$ as described for $\backslash\text{int_eval:n}$. 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.

46 Integer expression loops

$\backslash\text{int_do_while:nNnn} \star$	$\backslash\text{int_do_while:nNnn}$
	$\{\langle intexpr_1 \rangle\} \langle relation \rangle \{\langle intexpr_2 \rangle\} \{\langle code \rangle\}$

Evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for $\backslash\text{int_compare:nNnTF}$, 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**.

$\backslash\text{int_do_until:nNnn} \star$	$\backslash\text{int_do_until:nNnn}$
	$\{\langle intexpr_1 \rangle\} \langle relation \rangle \{\langle intexpr_2 \rangle\} \{\langle code \rangle\}$

Evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for $\backslash\text{int_compare:nNnTF}$, 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**.

$\backslash\text{int_until_do:nNnn} \star$	$\backslash\text{int_until_do:nNnn}$
	$\{\langle intexpr_1 \rangle\} \langle relation \rangle \{\langle intexpr_2 \rangle\} \{\langle code \rangle\}$

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 $\backslash\text{int_compare:nNnTF}$. 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**.

$\backslash\text{int_while_do:nNnn} \star$	$\backslash\text{int_while_do:nNnn}$	$\{\langle intexpr_1 \rangle\} \langle relation \rangle$
	$\{\langle intexpr_2 \rangle\} \{\langle code \rangle\}$	

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for $\text{\int_compare:nNnTF}$. 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**.

$\text{\int_do_while:nn} \star$	$\text{\int_do_while:nNnn}$ $\{ \langle intexpr1 \rangle \langle relation \rangle \langle intexpr2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

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 \TeX the test will be repeated, and a loop will occur until the test is **false**.

$\text{\int_do_until:nn} \star$	\int_do_until:nn $\{ \langle intexpr1 \rangle \langle relation \rangle \langle intexpr2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

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 \TeX the test will be repeated, and a loop will occur until the test is **true**.

$\text{\int_until_do:nn} \star$	\int_until_do:nn $\{ \langle intexpr1 \rangle \langle relation \rangle \langle intexpr2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for \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 $\langle relation \rangle$ is **true**.

$\text{\int_while_do:nn} \star$	\int_while_do:nn $\{ \langle intexpr1 \rangle \langle relation \rangle \langle intexpr2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for \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 $\langle relation \rangle$ is **false**.

47 Formatting integers

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

$\text{\int_to_arabic:n} \star$	$\text{\int_to_arabic:n} \{ \langle integer\ expression \rangle \}$
-----------------------------------	---

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

$\text{\int_to_alph:n} \star$	$\text{\int_to_alph:n} \{ \langle integer\ expression \rangle \}$
$\text{\int_to_Alph:n} \star$	

Evaluates the $\langle integer\ expression \rangle$ and converts the result into a series of letters, which

are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order. Thus

```
\int_to_alph:n { 1 }
```

places **a** in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as **z** and

```
\int_to_alph:n { 27 }
```

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

	<code>\int_to_symbols:nnn</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\int_to_symbols:nnn *</code></div>	$\{ \langle integer\ expression \rangle \} \{ \langle total\ symbols \rangle \}$ $\langle value\ to\ symbol\ mapping \rangle$

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 }
  }
}

```

`\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_base:nn *` `\int_to_base:nn {<integer expression>} {<base>}`

Calculates the value of the *<integer expression>* and converts it into the appropriate representation in the *<base>*; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by the upper case letters from the English alphabet. The maximum *<base>* value is 36.

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

`\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 Roman 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.

48 Converting from other formats to integers

`\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 *<binary number>* into the integer (base 10) representation and leaves this in the input stream.

`\int_from_hexadecimal:n *` `\int_from_hexadecimal:n {<hexadecimal number>}`

Converts the *<hexadecimal number>* into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the *<hexadecimal number>* by upper or lower case letters.

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

Converts the *<octal number>* 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 *<roman numeral>* into the integer (base 10) representation and leaves this in the input stream. The *<roman numeral>* may be in upper or lower case; if the numeral is not valid then the resulting value will be -1 .

`\int_from_base:nn *` `\int_from_base:nn {<number>}`
`{<base>}`

Converts the *<number>* in *<base>* into the appropriate value in base 10. The *<number>* should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum *<base>* value is 36.

49 Viewing integers

`\int_show:N`
`\int_show:c` `\int_show:N` *<integer>*

Displays the value of the *<integer>* on the terminal.

50 Constant integers

```
\c_minus_one
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand
```

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

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

`\c_max_register_int` Maximum number of registers.

51 Scratch integers

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

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

<code>\g_tmpa_int</code>
<code>\g_tmpb_int</code>

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

52 Internal functions

<code>int_get_digits:n *</code>

`\int_get_digits:n <value>`

Parses the *<value>* to leave the absolute *<value>* in the input stream. This may therefore be used to remove multiple sign tokens from the *<value>* (which may be symbolic).

<code>int_get_sign:n *</code>

`\int_get_sign:n <value>`

Parses the *<value>* to leave a single sign token (either + or -) in the input stream. This may therefore be used to sanitise sign tokens from the *<value>* (which may be symbolic).

<code>int_to_letter:n *</code>

`\int_to_letter:n <integer value>`

For *<integer values>* from 0 to 9, leaves the *<value>* in the input stream unchanged. For *<integer values>* from 10 to 35, leaves the appropriate upper case letter (from the standard English alphabet) in the input stream: for example, 10 is converted to A, 11 to B, *etc.*

<code>\int_to_roman:w *</code>

`\int_to_roman:w <integer>`
<space> or <non-expandable token>

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

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

	<code>\if_num:w</code>	<code><integer1></code>	<code><relation></code>	<code><integer2></code>	
				<code><true code></code>	
	<code>\else:</code>				
	<code><false code></code>				
	<code>\fi:</code>				

Compare two integers using `<relation>`, 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><integer></code>	<code><case0></code>
	<code>\or:</code>	<code><case1></code>	
	<code>\or:</code>	<code>...</code>	
	<code>\else:</code>	<code><default></code>	
	<code>\fi:</code>		

Selects a case to execute based on the value of the `<integer>`. The first case (`<case0>`) is executed if `<integer>` is 0, the second (`<case1>`) if the `<integer>` is 1, *etc.* The `<integer>` 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><integer></code>
	<code>\int_value:w</code>	<code><tokens></code> <code><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>

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

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

	<code>\if_int_odd:w</code>	<code><tokens></code>	<code><optional space></code>
		<code><true code></code>	
	<code>\else:</code>		
	<code><true code></code>		
	<code>\fi:</code>		

Expands `<tokens>` until a non-numeric token or a space is found, and tests whether the resulting `<integer>` is odd. If so, `<true code>` is executed. The `\else:` branch is optional.

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

Part X

The l3skip package

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

53 Creating and initialising dim variables

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

`\dim_new:N <dimension>`

Creates a new `<dimension>` or raises an error if the name is already taken. The declaration is global. The `<dimension>` will initially be equal to 0pt.

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

`\dim_zero:N <dimension>`

Sets `<dimension>` to 0pt within the scope of the current T_EX group.

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

`\dim_gzero:N <dimension>`

Sets `<dimension>` to 0pt globally, *i.e.* not restricted by the current T_EX group level.

54 Setting dim variables

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

`\dim_add:Nn <dimension> {<dimension expression>}`

Adds the result of the `<dimension expression>` to the current content of the `<dimension>`.

This assignment is local.

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

`\dim_gadd:Nn <dimension> {(dimension expression)}`

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

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

`\dim_set:Nn <dimension> {(dimension expression)}`

Sets $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, which must evaluate to a length with units. This assignment is restricted to the current \TeX group.

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

`\dim_gset:Nn <dimension> {(dimension expression)}`

Sets $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm. This assignment is global and is not limited to the current \TeX group level.

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

`\dim_set_eq:NN <dimension1> <dimension2>`

Sets the content of $\langle dimension1 \rangle$ equal to that of $\langle dimension2 \rangle$. This assignment is restricted to the current \TeX group level.

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

`\dim_gset_eq:NN <dimension1> <dimension2>`

Sets the content of $\langle dimension1 \rangle$ equal to that of $\langle dimension2 \rangle$. This assignment is global and so is not limited by the current \TeX group level.

<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_sub:Nn</code>
<code>\dim_sub:cn</code>

`\dim_sub:Nn <dimension> {<dimension expression>}`

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

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

`\dim_gsub:Nn <dimension> {<dimension expression>}`

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

55 Utilities for dimension calculations

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

`\dim_ratio:nn {<dimexpr1>} {<dimexpr2>}`

Parses the two $\langle dimension expressions \rangle$ and converts the ratio of the two to a form suitable for use inside a $\langle dimension expression \rangle$. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ration expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

will display 327680/655360 on the terminal.

56 Dimension expression conditionals

	<code>\dim_compare_p:nNn</code>
	<code>{\langle dimexpr_1 \rangle} \langle relation \rangle {\langle dimexpr_2 \rangle}</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\dim_compare_p:nNn *</code> </div>	<code>\dim_compare:nNnTF</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\dim_compare:nNnTF *</code> </div>	<code>{\langle dimexpr_1 \rangle} \langle relation \rangle {\langle dimexpr_2 \rangle}</code> <code>{\langle true code \rangle} {\langle false code \rangle}</code>

This function first evaluates each of the $\langle dimension expressions \rangle$ as described for `\dim_eval:n`. The two results are then compared using the $\langle relation \rangle$:

Equal	=
Greater than	>
Less than	<

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>\dim_compare_p:n</code>
	<code>{ \langle dimexpr1 \rangle \langle relation \rangle \langle dimexpr2 \rangle }</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\dim_compare_p:n *</code> </div>	<code>\dim_compare:nTF</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\dim_compare:nTF *</code> </div>	<code>{ \langle dimexpr1 \rangle \langle relation \rangle \langle dimexpr2 \rangle }</code> <code>{\langle true code \rangle} {\langle false code \rangle}</code>

This function first evaluates each of the $\langle dimension expressions \rangle$ as described for `\dim_eval:n`. The two results are then compared using the $\langle relation \rangle$:

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

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.

57 Dimension expression loops

	<code>\dim_do_while:nNnn</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\dim_do_while:nNnn *</code> </div>	<code>{\langle dimexpr_1 \rangle} \langle relation \rangle {\langle dimexpr_2 \rangle} {\langle code \rangle}</code>

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

<code>\dim_do_until:nNnn *</code>	<code>\dim_do_until:nNnn</code> <code>{<dimexpr₁> <relation> {<dimexpr₂>} {<code>}}</code>
-----------------------------------	---

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

<code>\dim_until_do:nNnn *</code>	<code>\dim_until_do:nNnn</code> <code>{<dimexpr₁> <relation> {<dimexpr₂>} {<code>}}</code>
-----------------------------------	---

Places the *<code>* in the input stream for T_EX to process, and then evaluates the relationship between the two *<dimension expressions>* as described for `\dim_compare:nNnTF`. If the test is **false** then the *<code>* will be inserted into the input stream again and a loop will occur until the *<relation>* is **true**.

<code>\dim_while_do:nNnn *</code>	<code>\dim_while_do:nNnn</code> <code>{<dimexpr₁> <relation> {<dimexpr₂>} {<code>}}</code>
-----------------------------------	---

Places the *<code>* in the input stream for T_EX to process, and then evaluates the relationship between the two *<dimension expressions>* as described for `\dim_compare:nNnTF`. If the test is **true** then the *<code>* will be inserted into the input stream again and a loop will occur until the *<relation>* is **false**.

<code>\dim_do_while:nn *</code>	<code>\dim_do_while:nn</code> <code>{ <dimexpr₁> <relation> <dimexpr₂> } {<code>}}</code>
---------------------------------	--

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

<code>\dim_do_until:nn *</code>	<code>\dim_do_until:nn</code> <code>{ <dimexpr₁> <relation> <dimexpr₂> } {<code>}}</code>
---------------------------------	--

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

<code>\dim_until_do:nn *</code>	<code>\dim_until_do:nn</code> <code>{ <dimexpr₁> <relation> <dimexpr₂> } {<code>}}</code>
---------------------------------	--

Places the *<code>* in the input stream for T_EX to process, and then evaluates the relationship between the two *<dimension expressions>* as described for `\dim_compare: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 $\langle relation \rangle$ is `true`.

<code>\dim_while_do:nn</code>	<code>\dim_while_do:nn</code>
-------------------------------	-------------------------------

`\dim_while_do:nn`

`{ \dimexpr1 \rangle \langle relation \rangle \dimexpr2 \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 $\langle dimension expressions \rangle$ as described for `\dim_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 $\langle relation \rangle$ is `false`.

58 Using dim expressions and variables

<code>\dim_eval:n</code>	<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>\dim_use:N</code>	<code>\dim_use:N</code>
-------------------------	-------------------------

`\dim_use:N`
`\dim_use:c`

`\dim_use:N \langle dimension \rangle`

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

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

59 Viewing dim variables

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

`\dim_show:N`
`\dim_show:c`

`\dim_show:N \langle dimension \rangle`

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

60 Constant dimensions

<code>\c_max_dim</code>

 The maximum value that can be stored as a dimension or skip (these are equivalent).

<code>\c_zero_dim</code>

 A zero length as a dimension or a skip (these are equivalent).

61 Scratch dimensions

<code>\l_tmpa_dim</code>
<code>\l_tmpb_dim</code>
<code>\l_tmpc_dim</code>

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

<code>\g_tmpa_dim</code>
<code>\g_tmpb_dim</code>

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

62 Creating and initialising skip variables

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

`\skip_new:N` *<skip>*
Creates a new *<skip>* or raises an error if the name is already taken. The declaration is global. The *<skip>* will initially be equal to 0pt.

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

`\skip_zero:N` *<skip>*
Sets *<skip>* to 0pt within the scope of the current T_EX group.

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

`\skip_gzero:N` *<skip>*
Sets *<skip>* to 0pt globally, *i.e.* not restricted by the current T_EX group level.

63 Setting skip variables

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

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

Adds the result of the *<skip expression>* to the current content of the *<skip>*. This assignment is local.

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

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

Adds the result of the *<skip expression>* to the current content of the *<skip>*. This assignment is global.

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

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

Sets *<skip>* to the value of *<skip expression>*, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm. This assignment is restricted to the current T_EX group.

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

`\skip_gset_eq:NN <skip1> <skip2>`

Sets the content of *<skip1>* equal to that of *<skip2>*. This assignment is global and so is not limited by the current T_EX group level.

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

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

Sets *<skip>* to the value of *<skip expression>*, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm. This assignment is global and is not limited to the current T_EX group level.

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

`\skip_set_eq:NN <skip1> <skip2>`

Sets the content of *<skip1>* equal to that of *<skip2>*. This assignment is restricted to the current T_EX group level.

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

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

Subtracts the result of the *<skip expression>* to the current content of the *<skip>*. This

assignment is local.

$\backslash\text{skip_gsub:Nn}$ $\backslash\text{skip_gsub:cn}$	$\backslash\text{skip_gsub:Nn}$ $\langle\text{skip}\rangle$ $\{\langle\text{skip expression}\rangle\}$
--	---

Subtracts the result of the $\langle\text{skip expression}\rangle$ to the current content of the $\langle\text{skip}\rangle$. This assignment is global.

64 Skip expression conditionals

$\backslash\text{skip_if_eq_p:n}$ \star $\backslash\text{skip_if_eq:nnTF}$ \star	$\backslash\text{skip_if_eq_p:nn}$ $\{\langle\text{skipexpr}_1\rangle\}$ $\{\langle\text{skipexpr}_2\rangle\}$ $\backslash\text{dim_compare:nTF}$ $\{\langle\text{skipexpr}_1\rangle\}$ $\{\langle\text{skipexpr}_2\rangle\}$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$
--	--

This function first evaluates each of the $\langle\text{skip expressions}\rangle$ as described for $\backslash\text{skip_eval:n}$. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true. The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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.

$\backslash\text{skip_if_infinite_glue_p:n}$ \star $\backslash\text{skip_if_infinite_glue:nTF}$ \star	$\backslash\text{skip_if_infinite_glue_p:n}$ $\{\langle\text{skipexpr}\rangle\}$ $\backslash\text{skip_if_infinite_glue:nTF}$ $\{\langle\text{skipexpr}\rangle\}$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$
---	---

Evaluates the $\langle\text{skip expression}\rangle$ as described for $\backslash\text{skip_eval:n}$, and then tests if this contains an infinite stretch or shrink component (or both). The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ 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

65 Using skip expressions and variables

$\backslash\text{skip_eval:n}$ \star	$\backslash\text{skip_eval:n}$ $\{\langle\text{skip expression}\rangle\}$
---	--

Evaluates the $\langle\text{skip expression}\rangle$, expanding any skips and token list variables within the $\langle\text{expression}\rangle$ to their content (without requiring $\backslash\text{skip_use:N}/\backslash\text{tl_use:N}$) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle\text{glue denotation}\rangle$ after two expansions. This will be expressed in points (**pt**), and

will require suitable termination if used in a T_EX-style assignment as it is *not* an *⟨internal glue⟩*.

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

`\skip_use:N ⟨skip⟩`

Recovers the content of a *⟨skip⟩* 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 *⟨dimension⟩* is required (such as in the argument of `\skip_eval:n`).

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

66 Viewing skip variables

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

`\skip_show:N ⟨skip⟩`

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

67 Constant skips

<code>\c_max_skip</code>

The maximum value that can be stored as a dimension or skip (these are equivalent).

<code>\c_zero_skip</code>

A zero length as a dimension or a skip (these are equivalent).

68 Scratch skips

<code>\l_tmpa_skip</code>
<code>\l_tmpb_skip</code>
<code>\l_tmpc_skip</code>

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

<code>\g_tmpa_skip</code>
<code>\g_tmpb_skip</code>

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

69 Creating and initialising muskip variables

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

`\muskip_new:N <muskip>`

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

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

`\skip_zero:N <muskip>`

Sets $\langle muskip \rangle$ to 0 mu within the scope of the current T_EX group.

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

`\muskip_gzero:N <muskip>`

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

70 Setting muskip variables

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

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

Adds the result of the $\langle muskip \text{ expression} \rangle$ to the current content of the $\langle muskip \rangle$. This assignment is local.

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

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

Adds the result of the $\langle muskip \text{ expression} \rangle$ to the current content of the $\langle muskip \rangle$. This assignment is global.

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

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

Sets $\langle muskip \rangle$ to the value of $\langle muskip \text{ expression} \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu. This assignment is restricted to the current T_EX group.

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

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

Sets $\langle muskip \rangle$ to the value of $\langle muskip \text{ expression} \rangle$, which must evaluate to a math length

with units and may include a rubber component (for example 1 mu plus 0.5 mu. This assignment is global and is not limited to the current T_EX group level.

<code>\muskip_set_eq:NN</code> <code>\muskip_set_eq:cN</code> <code>\muskip_set_eq:Nc</code> <code>\muskip_set_eq:cc</code>	<code>\muskip_set_eq:NN <muskip1> <muskip2></code>
--	--

Sets the content of $\langle muskip1 \rangle$ equal to that of $\langle muskip2 \rangle$. This assignment is restricted to the current T_EX group level.

<code>\muskip_gset_eq:NN</code> <code>\muskip_gset_eq:cN</code> <code>\muskip_gset_eq:Nc</code> <code>\muskip_gset_eq:cc</code>	<code>\muskip_gset_eq:NN <muskip1> <muskip2></code>
--	---

Sets the content of $\langle muskip1 \rangle$ equal to that of $\langle muskip2 \rangle$. This assignment is global and so is not limited by the current T_EX group level.

<code>\muskip_sub:Nn</code> <code>\muskip_sub:cn</code>	<code>\muskip_sub:Nn <muskip> {\muskip expression}</code>
--	---

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

<code>\muskip_gsub:Nn</code> <code>\muskip_gsub:cn</code>	<code>\muskip_gsub:Nn <muskip> {\muskip expression}</code>
--	--

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

71 Using muskip expressions and variables

<code>\muskip_eval:n *</code>	<code>\muskip_eval:n {\muskip expression}</code>
-------------------------------	--

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

<code>\muskip_use:N *</code> <code>\muskip_use:c *</code>	<code>\muskip_use:N <muskip></code>
--	---

Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error will

be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\muskip_eval:n`).

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

72 Inserting skips into the output

<code>\skip_horizontal:N</code>	
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:N $\langle skip \rangle$</code>
<code>\skip_horizontal:n</code>	<code>\skip_horizontal:n {\mathit{skipexpr}}</code>

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

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

<code>\skip_vertical:N</code>	
<code>\skip_vertical:c</code>	<code>\skip_vertical:N $\langle skip \rangle$</code>
<code>\skip_vertical:n</code>	<code>\skip_vertical:n {\mathit{skipexpr}}</code>

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

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

73 Viewing muskip variables

<code>\muskip_show:N</code>	
<code>\muskip_show:c</code>	<code>\muskip_show:N $\langle muskip \rangle$</code>

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

74 Internal functions

<code>\if_dim:w</code>	<code>\if_dim:w $\langle dimen1 \rangle$ $\langle relation \rangle$ $\langle dimen1 \rangle$</code>
	<code>$\langle true code \rangle$</code>
	<code>\else:</code>
	<code>$\langle false \rangle$</code>
<code>\if_dim:w</code>	<code>\fi:</code>

Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

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

<code>\dim_eval:w</code>	<code>*</code>
<code>\dim_eval_end:</code>	<code>*</code>

`\dim_eval:w <dimexpr> \dim_eval_end:`

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

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

75 Experimental skip functions

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

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

Checks if the $\langle skipexpr \rangle$ contains finite glue. If it does then it assigns $\langle dimen1 \rangle$ the stretch component and $\langle dimen2 \rangle$ the shrink component. If it contains infinite glue set $\langle dimen1 \rangle$ and $\langle dimen2 \rangle$ to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.

Part XI

The l3tl package

Token lists

L^AT_EX3 stores lists of token 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. Many of the functions for token lists and token list variables are very similar, and so are grouped together here.

76 Creating and initialising token list variables

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

`\tl_new:N <tl var>`

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

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

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

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

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

`\tl_clear:N <tl var>`

Clears all entries from the $\langle tl\ var \rangle$ within the scope of the current \TeX group.

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

`\tl_gclear:N <tl var>`

Clears all entries from the $\langle tl\ var \rangle$ globally.

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

`\tl_clear_new:N <tl var>`

If the $\langle tl\ var \rangle$ already exists, clears it within the scope of the current \TeX group. If the $\langle tl\ var \rangle$ is not defined, it will be created (using `\tl_new:N`). Thus the sequence is guaranteed to be available and clear within the current \TeX group. The $\langle tl\ var \rangle$ will exist globally, but the content outside of the current \TeX group is not specified.

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

`\tl_gclear_new:N <tl var>`

If the $\langle tl\ var \rangle$ already exists, clears it globally. If the $\langle tl\ var \rangle$ is not defined, it will be created (using `\tl_new:N`). Thus the sequence is guaranteed to be available and globally clear.

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

`\tl_set_eq:NN <tl var1> <tl var2>`

Sets the content of $\langle tl\ var1 \rangle$ equal to that of $\langle tl\ var2 \rangle$. This assignment is restricted to the current T_EX group level.

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

`\tl_gset_eq:NN $\langle tl\ var1 \rangle$ $\langle tl\ var2 \rangle$`

Sets the content of $\langle tl\ var1 \rangle$ equal to that of $\langle tl\ var2 \rangle$. This assignment is global and so is not limited by the current T_EX group level.

77 Adding data to token list variables

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

`\tl_set:Nn $\langle tl\ var \rangle$ $\{ \langle tokens \rangle \}$`

Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable. This assignment is restricted to the current T_EX group.

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

`\tl_gset:Nn $\langle tl\ var \rangle$ $\{ \langle tokens \rangle \}$`

Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable. This

assignment is global and is not limited to the current T_EX group level.

<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>
<code>\tl_put_left:cx</code>

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

Appends *<tokens>* to the left side of the current content of *<tl var>*. This modification is restricted to the current T_EX group level.

<code>\tl_gput_left:Nn</code>
<code>\tl_gput_left:NV</code>
<code>\tl_gput_left:No</code>
<code>\tl_gput_left:Nx</code>
<code>\tl_gput_left:cn</code>
<code>\tl_gput_left:cV</code>
<code>\tl_gput_left:co</code>
<code>\tl_gput_left:cx</code>

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

Globally appends *<tokens>* to the left side of the current content of *<tl var>*. This modification is not limited by T_EX grouping.

<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>
<code>\tl_put_right:cx</code>

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

Appends *<tokens>* to the right side of the current content of *<tl var>*. This modification is restricted to the current T_EX group level.

<code>\tl_gput_right:Nn</code>
<code>\tl_gput_right:NV</code>
<code>\tl_gput_right:No</code>
<code>\tl_gput_right:Nx</code>
<code>\tl_gput_right:cn</code>
<code>\tl_gput_right:cV</code>
<code>\tl_gput_right:co</code>
<code>\tl_gput_right:cx</code>

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

Globally appends $\langle tokens \rangle$ to the right side of the current content of $\langle tl var \rangle$. This modification is not limited by T_EX grouping.

78 Modifying token list variables

$\backslash\mathrm{tl_replace_once:Nnn}$ $\backslash\mathrm{tl_replace_once:cnn}$	$\backslash\mathrm{tl_replace_once:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	--

Replaces the first (leftmost) occurrence of $\langle old tokens \rangle$ in the $\langle tl var \rangle$ with $\langle new tokens \rangle$. $\langle Old tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (assuming normal T_EX category codes). The assignment is restricted to the current T_EX group.

$\backslash\mathrm{tl_greplace_once:Nnn}$ $\backslash\mathrm{tl_greplace_once:cnn}$	$\backslash\mathrm{tl_greplace_once:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	---

Replaces the first (leftmost) occurrence of $\langle old tokens \rangle$ in the $\langle tl var \rangle$ with $\langle new tokens \rangle$. $\langle Old tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (assuming normal T_EX category codes). The assignment is applied globally.

$\backslash\mathrm{tl_replace_all:Nnn}$ $\backslash\mathrm{tl_replace_all:cnn}$	$\backslash\mathrm{tl_replace_all:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	---

Replaces all occurrences of $\langle old tokens \rangle$ in the $\langle tl var \rangle$ with $\langle new tokens \rangle$. $\langle Old tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (assuming normal T_EX category codes). The assignment is restricted to the current T_EX group.

$\backslash\mathrm{tl_greplace_all:Nnn}$ $\backslash\mathrm{tl_greplace_all:cnn}$	$\backslash\mathrm{tl_greplace_all:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	--

Replaces all occurrences of $\langle old tokens \rangle$ in the $\langle tl var \rangle$ with $\langle new tokens \rangle$. $\langle Old tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (assuming normal T_EX category codes). The assignment is applied globally.

$\backslash\mathrm{tl_remove_once:Nn}$ $\backslash\mathrm{tl_remove_once:cn}$	$\backslash\mathrm{tl_remove_once:Nn} \langle tl var \rangle \{ \langle tokens \rangle \}$
--	--

Removes the first (leftmost) occurrence of $\langle tokens \rangle$ from the $\langle tl var \rangle$. $\langle Tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (assuming normal T_EX category codes). The assignment is restricted to the current T_EX group.

$\backslash\mathrm{tl_gremove_once:Nn}$ $\backslash\mathrm{tl_gremove_once:cn}$	$\backslash\mathrm{tl_gremove_once:Nn} \langle tl var \rangle \{ \langle tokens \rangle \}$
--	---

Removes the first (leftmost) occurrence of $\langle tokens \rangle$ from the $\langle tl var \rangle$. $\langle Tokens \rangle$ cannot

contain `{`, `}` or `#` (assuming normal T_EX category codes). The assignment is applied globally.

<code>\tl_remove_all:Nn</code> <code>\tl_remove_all:cn</code>	<code>\tl_remove_all:Nn <tl var> {<tokens>}</code>
--	--

Removes all occurrences of `<tokens>` from the `<tl var>`. `<Tokens>` cannot contain `{`, `}` or `#` (assuming normal T_EX category codes). The assignment is restricted to the current T_EX group.

<code>\tl_gremove_all:Nn</code> <code>\tl_gremove_all:cn</code>	<code>\tl_gremove_all:Nn <tl var> {<tokens>}</code>
--	---

Removes all occurrences of `<tokens>` from the `<tl var>`. `<Tokens>` cannot contain `{`, `}` or `#` (assuming normal T_EX category codes). The assignment is applied globally.

79 Reassigning token list category codes

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

Sets `<tl var>` to contain `<tokens>`, applying the category code régime specified in the `<setup>` before carrying out the assignment. This allows the `<tl var>` to contain material with category codes other than those that apply when `<tokens>` are absorbed. The assignment is local to the current T_EX group. See also `\tl_rescan:nn`.

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

Sets `<tl var>` to contain `<tokens>`, applying the category code régime specified in the `<setup>` before carrying out the assignment. This allows the `<tl var>` to contain material with category codes other than those that apply when `<tokens>` are absorbed. The assignment is global. See also `\tl_rescan:nn`.

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

Rescans `<tokens>` applying the category code régime specified in the `<setup>`, and leaves the resulting tokens in the input stream. See also `\tl_set_rescan:Nnn`.

80 Reassigning token list character codes

<code>\tl_to_lowercase:n</code>

`\tl_to_lowercase:n {\tokens}`

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

T_EXhackers note: This is the T_EX primitive `\lowercase` renamed. As a result, this function takes place on execution and not on expansion.

<code>\tl_to_uppercase:n</code>

`\tl_to_uppercase:n {\tokens}`

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

T_EXhackers note: This is the T_EX primitive `\uppercase` renamed.. As a result, this function takes place on execution and not on expansion.

81 Token list conditionals

<code>\tl_if_blank_p:n</code>	<code>*</code>
<code>\tl_if_blank:nTF</code>	<code>*</code>
<code>\tl_if_blank_p:V</code>	<code>*</code>
<code>\tl_if_blank:VTF</code>	<code>*</code>
<code>\tl_if_blank_p:o</code>	<code>*</code>
<code>\tl_if_blank:oTF</code>	<code>*</code>

`\tl_if_blank_p:n {\token list}`
`\tl_if_blank:nTF {\token list} {\true code} {\false code}`

Tests if the $\langle token list \rangle$ consists only of blank spaces. The test is **true** if $\langle token list \rangle$ is empty or consists entirely of explicit tokens of character code 32 and category code 10, and is **false** otherwise. 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>\tl_if_empty_p:N</code>	<code>*</code>
<code>\tl_if_empty:NTF</code>	<code>*</code>
<code>\tl_if_empty_p:c</code>	<code>*</code>
<code>\tl_if_empty:cTF</code>	<code>*</code>

`\tl_if_empty_p:N \tl var`
`\tl_if_empty:NTF \tl var {\true code} {\false code}`

Tests if the $\langle token list variable \rangle$ is entirely empty (*i.e.* contains no tokens at all). 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>\tl_if_empty_p:n *</code>	
<code>\tl_if_empty:nTF *</code>	
<code>\tl_if_empty_p:V *</code>	
<code>\tl_if_empty:VTF *</code>	
<code>\tl_if_empty_p:o *</code>	<code>\tl_if_empty_p:n {$\langle token\ list \rangle$}</code>
<code>\tl_if_empty:oTF *</code>	<code>\tl_if_empty:nTF {$\langle token\ list \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code>

Tests if the $\langle token\ list \rangle$ is entirely empty (*i.e.* contains no tokens at all). 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>\tl_if_eq_p:NN *</code>	
<code>\tl_if_eq:NNTF *</code>	
<code>\tl_if_eq_p:Nc *</code>	
<code>\tl_if_eq:NcTF *</code>	
<code>\tl_if_eq_p:cN *</code>	
<code>\tl_if_eq:cNTF *</code>	
<code>\tl_if_eq_p:cc *</code>	<code>\tl_if_eq_p:NN {$\langle tl\ var_1 \rangle$} {$\langle tl\ var_2 \rangle$}</code>
<code>\tl_if_eq:ccTF *</code>	<code>\tl_if_eq:NNTF {$\langle tl\ var_1 \rangle$} {$\langle tl\ var_2 \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code>

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

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq_p:NN \l_tmpa_tl \l_tmpb_tl
```

is logically **false**. 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>\tl_if_eq:nnTF</code>	<code>\tl_if_eq:nnTF $\langle token\ list1 \rangle$ {$\langle token\ list2 \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code>
-----------------------------	--

Tests if $\langle token\ list1 \rangle$ and $\langle token\ list2 \rangle$ are equal, 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.

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

Tests if the $\langle token\ list \rangle$ is found in the content of the $\langle token\ list\ variable \rangle$. The $\langle token$

list cannot contain the tokens `{`, `}` or `#` (assuming the usual \TeX category codes apply). Either the *true code* or *false code* is left in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

$\backslash\mathrm{tl_if_in:nnTF}$ $\backslash\mathrm{tl_if_in:VnTF}$ $\backslash\mathrm{tl_if_in:onTF}$ $\backslash\mathrm{tl_if_in:onTF}$	$\backslash\mathrm{tl_if_in:nnTF}$ $\langle token\ list1 \rangle$ $\{\langle token\ list2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
--	--

Tests if the *token list1* is found inside *token list2*. The *token list* cannot contain the tokens `{`, `}` or `#` (assuming the usual \TeX category codes apply). Either the *true code* or *false code* is left in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

$\backslash\mathrm{tl_if_single_p:N}$ \star $\backslash\mathrm{tl_if_single:NTF}$ \star $\backslash\mathrm{tl_if_single_p:c}$ \star $\backslash\mathrm{tl_if_single:cTF}$ \star	$\backslash\mathrm{tl_if_single_p:N}$ $\{\langle tl\ var \rangle\}$ $\backslash\mathrm{tl_if_single:NTF}$ $\{\langle tl\ var \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
--	--

Tests if the content of the *tl var* consists of a single token or token group. The test is **true** if *token list* contains exactly one token, if it consists entirely of explicit tokens of character code 32 and category code 10, or if it contains one braced token group (optionally preceded by spaces), and it is **false** otherwise. 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.

$\backslash\mathrm{tl_if_single_p:n}$ \star $\backslash\mathrm{tl_if_single:nTF}$ \star	$\backslash\mathrm{tl_if_single_p:n}$ $\{\langle token\ list \rangle\}$ $\backslash\mathrm{tl_if_single:nTF}$ $\{\langle token\ list \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
---	--

Tests if the *token list* consists of a single token or token group. The test is **true** if *token list* contains exactly one token, or if it consists entirely of explicit tokens of character code 32 and category code 10, or if it contains a braced token group (optionally preceded by blank spaces), and it is **false** otherwise. 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.

82 Mapping to token lists

$\backslash\mathrm{tl_map_function:NN}$ \star $\backslash\mathrm{tl_map_function:cN}$ \star	$\backslash\mathrm{tl_map_function:NN}$ $\langle tl\ var \rangle$ $\langle function \rangle$
--	--

Applies *function* to every *token group* stored in the *tl var*. The *function* will

receive one argument for each iteration. This may be a number of tokens if the $\langle token\ group \rangle$ was an item stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n -type arguments. See also `\tl_map_function:nN`.

<code>\tl_map_function:nN *</code>	<code>\tl_map_function:nN $\langle token\ list \rangle$ $\langle function \rangle$</code>
------------------------------------	---

Applies $\langle function \rangle$ to every $\langle token\ group \rangle$ in the $\langle token\ list \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle token\ group \rangle$ was an item stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n -type arguments. See also `\tl_map_function:NN`.

<code>\tl_map_inline:Nn</code>	<code>\tl_map_inline:Nn $\langle tl\ var \rangle$ $\{ \langle inline\ function \rangle \}$</code>
<code>\tl_map_inline:cn</code>	

Applies the $\langle inline\ function \rangle$ to every $\langle token\ group \rangle$ stored within the $\langle tl\ var \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle token\ group \rangle$ as $\#1$. One in line mapping can be nested inside another. See also `\tl_map_function:Nn`.

<code>\tl_map_inline:nn</code>	<code>\tl_map_inline:nn $\langle token\ list \rangle$ $\{ \langle inline\ function \rangle \}$</code>
--------------------------------	---

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

<code>\tl_map_variable:NNn</code>	<code>\tl_map_variable:NNn $\langle tl\ var \rangle$ $\langle variable \rangle$ $\{ \langle function \rangle \}$</code>
<code>\tl_map_variable:cNn</code>	

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

<code>\tl_map_variable:nNn</code>	<code>\tl_map_variable:nNn $\langle token\ list \rangle$ $\langle variable \rangle$ $\{ \langle function \rangle \}$</code>
-----------------------------------	--

Applies the $\langle function \rangle$ to every $\langle token\ group \rangle$ stored within the $\langle token\ list \rangle$. The $\langle function \rangle$ should consist of code which will receive the $\langle token\ group \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also `\tl_map-inline:nn`.

<code>\tl_map_break: *</code>	<code>\tl_map_break:</code>
-------------------------------	-----------------------------

Used to terminate a `\tl_map...` function before all entries in the $\langle token\ list\ variable \rangle$ have been processed. This will normally take place within a conditional statement, for example

```

\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \tl_map_break: }
  {
    % Do something useful
  }
}

```

Use outside of a `\tl_map...` scenario will lead low level TeX errors.

83 Using token lists

<code>\tl_to_str:N *</code> <code>\tl_to_str:c *</code>	<code>\tl_to_str:N <tl var></code>
--	--

Converts the content of the `<tl var>` into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This `<string>` is then left in the input stream.

<code>\tl_to_str:n *</code>	<code>\tl_to_str:n {<tokens>}</code>
-----------------------------	--

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

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

<code>\tl_use:N *</code> <code>\tl_use:c *</code>	<code>\tl_use:N <tl var></code>
--	---------------------------------------

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

84 Working with the content of token lists

<code>\tl_length:n *</code> <code>\tl_length:V *</code> <code>\tl_length:o *</code>	<code>\tl_length:n {<tokens>}</code>
---	--

Counts the number of token groups in `<tokens>` and leaves this information in the input

stream. Unbraced tokens count as one element as do each token group ($\{\dots\}$). This process will ignore any unprotected spaces within $\langle tokens \rangle$. See also `\tl_length:N`. This function requires three expansions, giving an $\langle integer denotation \rangle$.

<code>\tl_length:N *</code>	<code>\tl_length:N {\tl var}</code>
<code>\tl_length:c *</code>	

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

<code>\tl_reverse:n *</code>	<code>\tl_reverse:n {\tokens}</code>
<code>\tl_reverse:V *</code>	
<code>\tl_reverse:o *</code>	

Reverses the order of the $\langle tokens \rangle$, so that $\langle token1 \rangle \langle token2 \rangle \langle token3 \rangle \dots \langle token_n \rangle$ becomes $\langle token_n \rangle \dots \langle token3 \rangle \langle token2 \rangle \langle token1 \rangle$. This process will remove any unprotected space within the $\langle tokens \rangle$. Tokens are not reversed within braced token groups, which lose their outer set of braces. See also `\tl_reverse:N`.

<code>\tl_reverse:N</code>	<code>\tl_reverse:N {\tl var}</code>
<code>\tl_reverse:c</code>	

Reverses the order of the $\langle tokens \rangle$ stored in $\langle tl var \rangle$, so that $\langle token1 \rangle \langle token2 \rangle \langle token3 \rangle \dots \langle token_n \rangle$ becomes $\langle token_n \rangle \dots \langle token3 \rangle \langle token2 \rangle \langle token1 \rangle$. This process will remove any unprotected spaces within the $\langle token list variable \rangle$. Braced token groups are copied without reversing the order of tokens, and the outer set of braces is removed. The reversal is local to the current \TeX group. See also `\tl_reverse:n`.

<code>\tl_trim_spaces:n *</code>	<code>\tl_trim_spaces:n \langle token list \rangle</code>
----------------------------------	---

Removes any leading and trailing spaces from the $\langle token list \rangle$ and leaves the result in the input stream. This process requires two expansions.

<code>\tl_trim_spaces:N</code>	<code>\tl_trim_spaces:N \langle tl var \rangle</code>
<code>\tl_trim_spaces:c</code>	

Removes any leading and trailing spaces from the content of the $\langle tl var \rangle$ within the current \TeX group.

<code>\tl_gtrim_spaces:N</code>	<code>\tl_gtrim_spaces:N \langle tl var \rangle</code>
<code>\tl_gtrim_spaces:c</code>	

Removes any leading and trailing spaces from the content of the $\langle tl var \rangle$ globally.

85 The first token from a token list

Functions which deal with either only the very first token of a token list or everything except the first token.

<code>\tl_head:n *</code>
<code>\tl_head:V *</code>
<code>\tl_head:v *</code>
<code>\tl_head:f *</code>

`\tl_head:n {<tokens>}`

Leaves only the first *<token>* in *<tokens>* in the input stream, discarding the remainder.

<code>\tl_head:w *</code>

`\tl_head:w {<tokens>} \q_stop`

Leaves only the first *<token>* in *<tokens>* in the input stream, discarding the remainder. This function requires only a single expansion, and so is suitable for use inside an *o*-type expansion. In general `\tl_head:n` should be preferred.

<code>\tl_tail:n *</code>
<code>\tl_tail:V *</code>
<code>\tl_tail:v *</code>
<code>\tl_tail:f *</code>

`\tl_tail:n {<tokens>}`

Discards the first *<token>* of the *<tokens>* and leaves the remainder in the input stream.

<code>\tl_tail:w *</code>

`\tl_tail:w {<tokens>} \q_stop`

Discards the first *<token>* of the *<tokens>* and leaves the remainder in the input stream. This function requires only a single expansion, and so is suitable for use inside an *o*-type expansion. In general `\tl_tail:n` should be preferred.

<code>\tl_if_head_eq_catcode_p:nN *</code>
<code>\tl_if_head_eq_catcode:nNTF *</code>

`\tl_if_head_eq_catcode_p:n {<token list>} <test token>`
`\tl_if_head_eq_catcode:nTF {<token list>} <test token>`
`{<true code>} {<false code>}`

Tests if the first *<token>* in the *<token list>* has the same category code as the *<test token>*. 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>\tl_if_head_eq_charcode_p:nN *</code>
<code>\tl_if_head_eq_charcode:nNTF *</code>
<code>\tl_if_head_eq_charcode_p:fN *</code>
<code>\tl_if_head_eq_charcode:fNTF *</code>

`\tl_if_head_eq_charcode_p:n {<token list>} <test token>`
`\tl_if_head_eq_charcode:nTF {<token list>} <test token>`
`{<true code>} {<false code>}`

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same character code as the $\langle test token \rangle$. 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.

$\backslash tl_if_head_eq_meaning_p:nN \star$ $\backslash tl_if_head_eq_meaning:nNTF \star$

 $\backslash tl_if_head_eq_meaning_p:n \{ \langle token list \rangle \} \langle test token \rangle$
 $\backslash tl_if_head_eq_meaning:nTF \{ \langle token list \rangle \} \langle test token \rangle$
 $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same meaning as the $\langle test token \rangle$. 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.

86 Viewing token lists

$\backslash tl_show:N$ $\backslash tl_show:c$
--

 $\backslash tl_show:N \langle tl var \rangle$

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

T_EXhackers note: $\backslash tl_show:N$ is the T_EX primitive $\backslash show$.

$\backslash tl_show:n$

 $\backslash tl_show:n \langle token list \rangle$

Displays the $\langle token list \rangle$ on the terminal.

T_EXhackers note: $\backslash tl_show:n$ is the ε -T_EX primitive $\backslash showtokens$.

87 Constant token lists

$\backslash c_job_name_tl$

Constant that gets the “job name” assigned when T_EX starts.

T_EXhackers note: This is the new name for the primitive $\backslash jobname$. It is a constant that is set by T_EX and should not be overwritten by the package.

$\backslash c_empty_tl$

Constant that is always empty.

$\backslash c_space_tl$

A space token contained in a token list (compare this with $\backslash c_space_token$). For use where an explicit space is required.

88 Scratch token lists

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

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

<code>\g_tmpa_tl</code> <code>\g_tmpb_tl</code>
--

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

89 Experimental token list functions

<code>\tl_if_single_item_p:n *</code> <code>\tl_if_single_item:nTF *</code>	<code>\tl_if_single_item_p:n {⟨token list⟩}</code> <code>\tl_if_single_item:nTF {⟨token list⟩}</code> <code>{⟨true code⟩} {⟨false code⟩}</code>
--	---

Tests if the token list has exactly one item, i.e., is either a single normal token or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has length 1 according to `\tl_length: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. The logical truth of the test is left in the input stream by the predicate.

<code>\tl_if_head_begin_group_p:n *</code> <code>\tl_if_head_begin_group:nTF *</code>	<code>\tl_if_head_begin_group_p:n {⟨token list⟩}</code> <code>\tl_if_head_begin_group:nTF {⟨token list⟩}</code> <code>{⟨true code⟩} {⟨false code⟩}</code>
--	---

Tests if the first *⟨token⟩* in the *⟨token list⟩* has catcode 1, i.e., is a begin-group character. This conditional is mainly intended to be used in combination with `\tl_if_head_eq_space:n` to check if grabbing an undelimited argument from the token list is safe. The branching versions 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.

<code>\tl_if_head_eq_space_p:n *</code> <code>\tl_if_head_eq_space:nTF *</code>	<code>\tl_if_head_eq_space_p:n {⟨token list⟩}</code> <code>\tl_if_head_eq_space:nTF {⟨token list⟩}</code> <code>{⟨true code⟩} {⟨false code⟩}</code>
--	---

Tests if the first token of the token list is an explicit space, i.e., a token which matches `\space` both in character code and in category code. In the case of an implicit space token, such as `\c_space_token`, the test will return `\false`. This conditional is mainly intended to be used in combination with `\tl_if_head_begin_group:n` to test if grabbing an unlimited argument is safe (see `\tl_if_single_token:n` for an example). The branching versions 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.

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

<code>\tl_if_single_token_p:n *</code> <code>\tl_if_single_token:nTF *</code>	<code>\tl_if_single_token_p:n {\token list}</code> <code>\tl_if_single_token:nTF {\token list}</code> <code>{\true code} {\false code}</code>
--	---

Tests if the token list consists of exactly one token, i.e., is either a single space character or a single “normal” token. Token groups are not single tokens. The branching versions leaves 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.

Part XII

The l3seq package

Sequences and stacks

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

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

90 Creating and initialising sequences

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

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

$\backslash seq_clear:N$
$\backslash seq_clear:c$

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

Clears all items from the $\langle sequence \rangle$ within the scope of the current $\text{T}_{\text{E}}\text{X}$ group.

$\backslash seq_gclear:N$
$\backslash seq_gclear:c$

 $\backslash seq_gclear:N \langle sequence \rangle$

Clears all entries from the $\langle sequence \rangle$ globally.

$\backslash seq_clear_new:N$
$\backslash seq_clear_new:c$

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

If the $\langle sequence \rangle$ already exists, clears it within the scope of the current $\text{T}_{\text{E}}\text{X}$ group. If the $\langle sequence \rangle$ is not defined, it will be created (using $\backslash seq_new:N$). Thus the sequence is guaranteed to be available and clear within the current $\text{T}_{\text{E}}\text{X}$ group. The $\langle sequence \rangle$ will exist globally, but the content outside of the current $\text{T}_{\text{E}}\text{X}$ group is not specified.

$\backslash seq_gclear_new:N$
$\backslash seq_gclear_new:c$

 $\backslash seq_gclear_new:N \langle sequence \rangle$

If the $\langle sequence \rangle$ already exists, clears it globally. If the $\langle sequence \rangle$ is not defined, it will be created (using $\backslash seq_new:N$). Thus the sequence is guaranteed to be available and globally clear.

$\backslash seq_set_eq:NN$
$\backslash seq_set_eq:cN$
$\backslash seq_set_eq:Nc$
$\backslash seq_set_eq:cc$

 $\backslash seq_set_eq:NN \langle sequence1 \rangle \langle sequence2 \rangle$

Sets the content of $\langle sequence1 \rangle$ equal to that of $\langle sequence2 \rangle$. This assignment is restricted to the current $\text{T}_{\text{E}}\text{X}$ group level.

$\backslash seq_gset_eq:NN$
$\backslash seq_gset_eq:cN$
$\backslash seq_gset_eq:Nc$
$\backslash seq_gset_eq:cc$

 $\backslash seq_gset_eq:NN \langle sequence1 \rangle \langle sequence2 \rangle$

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

$\backslash seq_concat:NNN$
$\backslash seq_concat:ccc$

 $\backslash seq_concat:NNN \langle sequence1 \rangle \langle sequence2 \rangle \langle sequence3 \rangle$

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 $\langle sequence1 \rangle$.

$\backslash seq_gconcat:Nn$ $\backslash seq_gconcat:ccc$	$\backslash seq_gconcat:Nn \langle sequence1 \rangle \langle sequence2 \rangle \langle sequence3 \rangle$
---	--

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 $\langle sequence1 \rangle$.

91 Appending data to sequences

$\backslash seq_put_left:Nn$ $\backslash seq_put_left:NV$ $\backslash seq_put_left:Nv$ $\backslash seq_put_left:No$ $\backslash seq_put_left:Nx$ $\backslash seq_put_left:cn$ $\backslash seq_put_left:cV$ $\backslash seq_put_left:cv$ $\backslash seq_put_left:co$ $\backslash seq_put_left:cx$	$\backslash seq_put_left:Nn \langle sequence \rangle \{ \langle item \rangle \}$
--	--

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$. The assignment is restricted to the current \TeX group.

$\backslash seq_gput_left:Nn$ $\backslash seq_gput_left:NV$ $\backslash seq_gput_left:Nv$ $\backslash seq_gput_left:No$ $\backslash seq_gput_left:Nx$ $\backslash seq_gput_left:cn$ $\backslash seq_gput_left:cV$ $\backslash seq_gput_left:cv$ $\backslash seq_gput_left:co$ $\backslash seq_gput_left:cx$	$\backslash seq_gput_left:Nn \langle sequence \rangle \{ \langle item \rangle \}$
--	---

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 $\langle sequence \rangle$ { $\langle item \rangle$ }`

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 $\langle sequence \rangle$ { $\langle item \rangle$ }`

Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$. The assignment is global.

92 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>

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

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>

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

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:NN $\langle sequence \rangle$ $\langle token list variable \rangle$</code>
<code>\seq_pop_left:cN</code>	

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:NN $\langle sequence \rangle$ $\langle token list variable \rangle$</code>
<code>\seq_gpop_left:cN</code>	

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:NN $\langle sequence \rangle$ $\langle token list variable \rangle$</code>
<code>\seq_pop_right:cN</code>	

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:NN $\langle sequence \rangle$ $\langle token list variable \rangle$</code>
<code>\seq_gpop_right:cN</code>	

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.

93 Modifying sequences

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

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

Removes duplicate items from the $\langle sequence \rangle$, leaving the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is local to the current \TeX group.

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

<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 the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is applied globally.

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

<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 T_EX 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.

94 Sequence conditionals

<code>\seq_if_empty_p:N *</code> <code>\seq_if_empty:N\underline{TF} *</code> <code>\seq_if_empty_p:c *</code> <code>\seq_if_empty:c\underline{TF} *</code>	<code>\seq_if_empty_p:N <sequence></code> <code>\seq_if_empty:N\underline{TF} <sequence> {<true code>} {<false code>}</code>
--	--

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:NnTF</code>	
<code>\seq_if_in:NVTF</code>	
<code>\seq_if_in:NvTF</code>	
<code>\seq_if_in:NoTF</code>	
<code>\seq_if_in:NxTF</code>	
<code>\seq_if_in:cnTF</code>	
<code>\seq_if_in:cVTF</code>	
<code>\seq_if_in:cvTF</code>	
<code>\seq_if_in:coTF</code>	
<code>\seq_if_in:cxTF</code>	
	<code>\seq_if_in:NnTF <sequence> {<item>} {<true code>} {<false code>}</code>

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.

95 Mapping to sequences

<code>\seq_map_function:NN *</code>	
<code>\seq_map_function:cN *</code>	<code>\seq_map_function:NN <sequence> <function></code>

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 <sequence> {<inline function>}</code>

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

<code>\seq_map_variable:NNn</code>	
<code>\seq_map_variable:Ncn</code>	
<code>\seq_map_variable:cNn</code>	
<code>\seq_map_variable:ccn</code>	<code>\seq_map_variable:NNn <sequence> <tl var.> {<function using tl var.>}</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.

`\seq_map_break: *` `\seq_map_break:`

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

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

Use outside of a `\seq_map...` scenario will lead to low level T_EX errors.

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

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

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

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

Use outside of a `\seq_map...` scenario will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\seq_break_point:n` before the $\langle tokens \rangle$ are inserted into the input stream. This will depend on the design of the mapping function.

96 Sequences as stacks

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

<code>\seq_get:NN</code>

<code>\seq_get:cN</code>

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

Reads the top item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty an error will be raised.

<code>\seq_pop:NN</code>

<code>\seq_pop:cN</code>

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

Pops the top item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty an error will be raised.

<code>\seq_gpop:NN</code>

<code>\seq_gpop:cN</code>

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

Pops the top item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty 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 T_EX 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.

97 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.

98 Experimental sequence functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

<code>\seq_get_left:NNTF</code>
<code>\seq_get_left:cNTF</code>

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

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle sequence \rangle$ is non-empty, stores the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from a $\langle sequence \rangle$. The $\langle true code \rangle$ is then left in the input stream. The $\langle token list variable \rangle$ is assigned locally.

<code>\seq_get_right:NNTF</code>
<code>\seq_get_right:cNTF</code>

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

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle sequence \rangle$ is non-empty, stores the right-most item from

a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from a $\langle sequence \rangle$. The $\langle true code \rangle$ is then left in the input stream. The $\langle token list variable \rangle$ is assigned locally.

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

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle sequence \rangle$ is non-empty, pops the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, i.e. removes the item from a $\langle sequence \rangle$. The $\langle true code \rangle$ is then left in the input stream. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

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

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle sequence \rangle$ is non-empty, pops the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, i.e. removes the item from a $\langle sequence \rangle$. The $\langle true code \rangle$ is then left in the input stream. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

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

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle sequence \rangle$ is non-empty, pops the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, i.e. removes the item from a $\langle sequence \rangle$. The $\langle true code \rangle$ is then left in the input stream. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

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

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle sequence \rangle$ is non-empty, pops the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, i.e. removes the item from a $\langle sequence \rangle$. The $\langle true code \rangle$ is then left in the input stream. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

$\backslash seq_length: N \star$ $\backslash seq_length: c \star$	$\backslash seq_length: N \langle sequence \rangle$
--	--

Leaves the number of items in the $\langle sequence \rangle$ in the input stream as an $\langle integer \rangle$

denotation). The total number of items in a $\langle sequence \rangle$ will include those which are empty and duplicates, *i.e.* every item in a $\langle sequence \rangle$ is unique.

<code>\seq_item:Nn *</code>
<code>\seq_item:cn *</code>

`\seq_item:Nn $\langle sequence \rangle$ { $\langle integer expression \rangle$ }`

Indexing items in the $\langle sequence \rangle$ from 0 at the top (left), this function will evaluate the $\langle integer expression \rangle$ and leave the appropriate item from the sequence in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. When the $\langle integer expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by `\seq_length:N`) then the function will expand to nothing.

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

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

Places each $\langle item \rangle$ in the $\langle sequence \rangle$ in turn in the input stream. This occurs in an expandable fashion, and is implemented as a mapping. This means that the process may be prematurely terminated using `\seq_map_break:` or `\seq_map_break:n`. The $\langle items \rangle$ in the $\langle sequence \rangle$ will be used from left (top) to right (bottom).

<code>\seq_mapthread_function:NNN *</code>
<code>\seq_mapthread_function:NcN *</code>
<code>\seq_mapthread_function:cNN *</code>
<code>\seq_mapthread_function:ccN *</code>

`\seq_mapthread_function:NNN $\langle seq1 \rangle$ $\langle seq2 \rangle$ $\langle function \rangle$`

Applies $\langle function \rangle$ to every pair of items $\langle seq1-item \rangle$ – $\langle seq2-item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ will receive two *n*-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_from_clist:NN</code>
<code>\seq_set_from_clist:cN</code>
<code>\seq_set_from_clist:Nc</code>
<code>\seq_set_from_clist:cc</code>
<code>\seq_set_from_clist:Nn</code>
<code>\seq_set_from_clist:cn</code>

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

Sets the $\langle sequence \rangle$ within the current \TeX group to be equal to the content of the

$\langle comma-list \rangle$.

<code>\seq_gset_from_clist:NN</code> <code>\seq_gset_from_clist:cN</code> <code>\seq_gset_from_clist:Nc</code> <code>\seq_gset_from_clist:cc</code> <code>\seq_gset_from_clist:Nn</code> <code>\seq_gset_from_clist:cn</code>	<code>\seq_gset_from_clist:NN</code> $\langle sequence \rangle$ $\langle comma-list \rangle$
--	--

Sets the $\langle sequence \rangle$ globally to equal to the content of the $\langle comma-list \rangle$.

99 Internal sequence functions

<code>\seq_if_empty_err_break:N</code>	<code>\seq_if_empty_err_break:N</code> $\langle sequence \rangle$
--	---

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 $\langle sequence \rangle$.

<code>\seq_item:n *</code>	<code>\seq_item:n</code> $\langle item \rangle$
----------------------------	---

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.

<code>\seq_push_item_def:n</code> <code>\seq_push_item_def:x</code>	<code>\seq_push_item_def:n</code> $\{ \langle code \rangle \}$
--	--

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:.`

<code>\seq_pop_item_def:</code>	<code>\seq_pop_item_def:</code>
---------------------------------	---------------------------------

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`.

<code>\seq_break: *</code>	<code>\seq_break:</code>
----------------------------	--------------------------

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.

<code>\seq_break:n *</code>	<code>\seq_break:n {<i><tokens></i>}</code>
-----------------------------	---

Used to terminate sequence functions by gobbling all tokens up to `\seq_break_point:n`, then inserting the *<tokens>* 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.

<code>\seq_break_point:n *</code>	<code>\seq_break_point:n <i><tokens></i></code>
-----------------------------------	---

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 *<tokens>* 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`.

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.

100 Creating and initialising comma lists

<code>\clist_new:N</code>	<code>\clist_new:N <i><comma list></i></code>
<code>\clist_new:c</code>	

Creates a new *<comma list>* or raises an error if the name is already taken. The declaration

is global. The $\langle comma list \rangle$ will initially contain no items.

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

`\clist_clear:N \langle comma list \rangle`
 Clears all items from the $\langle comma list \rangle$ within the scope of the current T_EX group.

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

`\clist_gclear:N \langle comma list \rangle`
 Clears all entries from the $\langle comma list \rangle$ globally.

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

`\clist_clear_new:N \langle comma list \rangle`
 If the $\langle comma list \rangle$ already exists, clears it within the scope of the current T_EX group. If the $\langle comma list \rangle$ is not defined, it will be created (using `\clist_new:N`). Thus the comma list is guaranteed to be available and clear within the current T_EX group. The $\langle comma list \rangle$ will exist globally, but the content outside of the current T_EX group is not specified.

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

`\clist_gclear_new:N \langle comma list \rangle`
 If the $\langle comma list \rangle$ already exists, clears it globally. If the $\langle comma list \rangle$ is not defined, it will be created (using `\clist_new:N`). Thus the comma list is guaranteed to be available and globally clear.

<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 \langle comma list1 \rangle \langle comma list2 \rangle`
 Sets the content of $\langle comma list1 \rangle$ equal to that of $\langle comma list2 \rangle$. This assignment is restricted to the current T_EX 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 \langle comma list1 \rangle \langle comma list2 \rangle`
 Sets the content of $\langle comma list1 \rangle$ equal to that of $\langle comma list2 \rangle$. This assignment is global and so is not limited by the current T_EX group level.

<code>\clist_concat:NNN</code>
<code>\clist_concat:ccc</code>

`\clist_concat:NNN \langle comma list1 \rangle \langle comma list2 \rangle \langle comma list3 \rangle`
 Concatenates the content of $\langle comma list2 \rangle$ and $\langle comma list3 \rangle$ together and saves the

result in $\langle comma list1 \rangle$. The items in $\langle comma list2 \rangle$ 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 $\langle comma list1 \rangle$.

$\backslash\text{clist_gconcat:NNN}$	$\backslash\text{clist_gconcat:NNN}$ $\langle comma list1 \rangle$ $\langle comma list2 \rangle$
$\backslash\text{clist_gconcat:ccc}$	$\langle comma list3 \rangle$

Concatenates the content of $\langle comma list2 \rangle$ and $\langle comma list3 \rangle$ together and saves the result in $\langle comma list1 \rangle$. The items in $\langle comma list2 \rangle$ will be placed at the left side of the new comma list. This operation is global and will remove any existing content in $\langle comma list1 \rangle$.

101 Appending items to comma lists

$\backslash\text{clist_put_left:Nn}$	$\backslash\text{clist_put_left:Nn}$ $\langle comma list \rangle$ $\{ \langle item \rangle \}$
$\backslash\text{clist_put_left:NV}$	
$\backslash\text{clist_put_left:No}$	
$\backslash\text{clist_put_left:Nx}$	
$\backslash\text{clist_put_left:cn}$	
$\backslash\text{clist_put_left:cV}$	
$\backslash\text{clist_put_left:co}$	
$\backslash\text{clist_put_left:cx}$	

Appends the $\langle item \rangle$ to the left of the $\langle comma list \rangle$. The assignment is restricted to the

current T_EX group.

<code>\clist_gput_left:Nn</code>
<code>\clist_gput_left:NV</code>
<code>\clist_gput_left:No</code>
<code>\clist_gput_left:Nx</code>
<code>\clist_gput_left:cn</code>
<code>\clist_gput_left:cV</code>
<code>\clist_gput_left:co</code>
<code>\clist_gput_left:cx</code>

`\clist_gput_left:Nn <comma list> {<item>}`

Appends the *<item>* to the left of the *<comma list>*. The assignment is global.

<code>\clist_put_right:Nn</code>
<code>\clist_put_right:NV</code>
<code>\clist_put_right:No</code>
<code>\clist_put_right:Nx</code>
<code>\clist_put_right:cn</code>
<code>\clist_put_right:cV</code>
<code>\clist_put_right:co</code>
<code>\clist_put_right:cx</code>

`\clist_put_right:Nn <comma list> {<item>}`

Appends the *<item>* to the right of the *<comma list>*. The assignment is restricted to the current T_EX group.

<code>\clist_gput_right:Nn</code>
<code>\clist_gput_right:NV</code>
<code>\clist_gput_right:No</code>
<code>\clist_gput_right:Nx</code>
<code>\clist_gput_right:cn</code>
<code>\clist_gput_right:cV</code>
<code>\clist_gput_right:co</code>
<code>\clist_gput_right:cx</code>

`\clist_gput_right:Nn <comma list> {<item>}`

Appends the *<item>* to the right of the *<comma list>*. The assignment is global.

102 Comma lists as stacks

<code>\clist_get:NN</code>
<code>\clist_get:cN</code>

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

Stores the left-most item from a *<comma list>* in the *<token list variable>* without removing

it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally.

<code>\clist_get:NN</code>
<code>\clist_get:cN</code>

`\clist_get:NN` $\langle comma list \rangle$ $\langle token list variable \rangle$

Stores the right-most item from a $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally.

<code>\clist_pop:NN</code>
<code>\clist_pop:cN</code>

`\clist_pop:NN` $\langle comma list \rangle$ $\langle token list variable \rangle$

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

<code>\clist_gpop:NN</code>
<code>\clist_gpop:cN</code>

`\clist_gpop:NN` $\langle comma list \rangle$ $\langle token list variable \rangle$

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

<code>\clist_push:Nn</code>
<code>\clist_push:NV</code>
<code>\clist_push:No</code>
<code>\clist_push:Nx</code>
<code>\clist_push:cn</code>
<code>\clist_push:cV</code>
<code>\clist_push:co</code>
<code>\clist_push:cx</code>

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

Adds the $\{\langle item \rangle\}$ to the top of the $\langle comma list \rangle$. The assignment is restricted to the current $\text{T}_{\text{E}}\text{X}$ group.

<code>\clist_gpush:Nn</code>
<code>\clist_gpush:NV</code>
<code>\clist_gpush:No</code>
<code>\clist_gpush:Nx</code>
<code>\clist_gpush:cn</code>
<code>\clist_gpush:cV</code>
<code>\clist_gpush:co</code>
<code>\clist_gpush:cx</code>

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

Pushes the $\langle item \rangle$ onto the end of the top of the $\langle comma list \rangle$. The assignment is global.

103 Using comma lists

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

`\clist_use:N <comma list>`

Places the *<comma list>* directly into the input stream, thus treating it as a *<token list>*.

104 Modifying comma lists

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

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

`\clist_remove_duplicates:N <comma list>`

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

\TeX hackers note: This function iterates through every item in the *<comma list>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large comma lists.

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

`\clist_gremove_duplicates:N <comma list>`

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

\TeX hackers note: This function iterates through every item in the *<comma list>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large comma lists.

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

`\clist_remove_all:Nn <comma list> {<item>}`

Removes every occurrence of *<item>* from the *<comma list>*. The *<item>* comparison takes

place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is local to the current \TeX group.

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

Removes each occurrence of $\langle item \rangle$ from the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is applied globally.

<code>\clist_trim_spaces:N</code> <code>\clist_trim_spaces:c</code>	<code>\clist_trim_spaces:N <comma list></code>
--	--

Removes leading and trailing spaces from each $\langle item \rangle$ in the $\langle comma list \rangle$ within the current \TeX group. This space-removal process takes place as described for `\tl_trim_spaces:n`.

<code>\clist_gtrim_spaces:N</code> <code>\clist_gtrim_spaces:c</code>	<code>\clist_gtrim_spaces:N <comma list></code>
--	---

Removes leading and trailing spaces from each $\langle item \rangle$ in the $\langle comma list \rangle$ globally. This space-removal process takes place as described for `\tl_trim_spaces:n`.

<code>\clist_trim_spaces:n *</code>	<code>\clist_trim_spaces:N <comma list></code>
-------------------------------------	--

Removes leading and trailing spaces from each $\langle item \rangle$ in the $\langle comma list \rangle$, leaving the resulting modified list in the input stream.

105 Comma list conditionals

<code>\clist_if_empty_p:N *</code> <code>\clist_if_empty:NTF *</code> <code>\clist_if_empty_p:c *</code> <code>\clist_if_empty:cTF *</code>	<code>\clist_if_empty_p:N <comma list></code> <code>\clist_if_empty:NTF <comma list></code> <code>{\<true code>}\{\<false code>}</code>
--	---

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.

<pre> \clist_if_eq_p:NN * \clist_if_eq:NNTF * \clist_if_eq_p:Nc * \clist_if_eq:NcTF * \clist_if_eq_p:cN * \clist_if_eq:cNTF * \clist_if_eq_p:cc * \clist_if_eq:ccTF * </pre>	<pre> \clist_if_eq_p:NN {<clist₁>} {<clist₂>} \clist_if_eq:NNTF {<clist₁>} {<clist₂>} {<true code>} {<false code>} </pre>
--	---

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.

<pre> \clist_if_in:NnTF \clist_if_in:NVTF \clist_if_in:NoTF \clist_if_in:cnTF \clist_if_in:cVTF \clist_if_in:coTF \clist_if_in:nnTF \clist_if_in:nVTF \clist_if_in:noTF </pre>	<pre> \clist_if_in:NnTF <comma list> {<item>} {<true code>} {<false code>} </pre>
--	---

Tests if the *<item>* is present in the *<comma list>*. Either the *<true code>* or *<false code>* is left in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

106 Mapping to comma lists

<pre> \clist_map_function:NN * \clist_map_function:cN * \clist_map_function:nN * </pre>	<pre> \clist_map_function:NN <comma list> <function> </pre>
---	---

Applies *<function>* to every *<item>* stored in the *<comma list>*. The *<function>* will receive one argument for each iteration. The *<items>* are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`.

One mapping may be nested inside another.

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

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

<code>\clist_map_variable:NNn</code> <code>\clist_map_variable:cNn</code> <code>\clist_map_variable:nNn</code>	<code>\clist_map_variable:NNn <comma list></code> <code><tl var.> {<function using tl var.>}</code>
--	--

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

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

Used to terminate a `\clist_map...` function before all entries in the *<comma list>* have been processed. This will normally take place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map...` scenario will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\clist_break_point:n` before further items are taken from the input stream. This will depend on the design of the mapping function.

<code>\clist_map_break:n *</code>	<code>\clist_map_break:n {<tokens>}</code>
-----------------------------------	--

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

have been processed, inserting the $\langle tokens \rangle$ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map...` scenario will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\clist_break_point:n` before the $\langle tokens \rangle$ are inserted into the input stream. This will depend on the design of the mapping function.

107 Comma lists as stacks

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

<code>\clist_get:NN</code> <code>\clist_get:cN</code>	<code>\clist_get:NN</code> $\langle comma list \rangle$ $\langle token list variable \rangle$
--	---

Reads the top item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle comma list \rangle$ is empty an error will be raised.

<code>\clist_pop:NN</code> <code>\clist_pop:cN</code>	<code>\clist_pop:NN</code> $\langle comma list \rangle$ $\langle token list variable \rangle$
--	---

Pops the top item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle comma list \rangle$ is empty an error will be raised.

<code>\clist_gpop:NN</code> <code>\clist_gpop:cN</code>	<code>\clist_gpop:NN</code> $\langle comma list \rangle$ $\langle token list variable \rangle$
--	--

Pops the top item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$. The $\langle comma list \rangle$

is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle comma\ list \rangle$ is empty an error will be raised.

<code>\clist_push:Nn</code>
<code>\clist_push:NV</code>
<code>\clist_push:No</code>
<code>\clist_push:Nx</code>
<code>\clist_push:cn</code>
<code>\clist_push:cV</code>
<code>\clist_push:co</code>
<code>\clist_push:cx</code>

`\clist_push:Nn $\langle comma\ list \rangle$ $\{ \langle item \rangle \}$`

Adds the $\{ \langle item \rangle \}$ to the top of the $\langle comma\ list \rangle$. The assignment is restricted to the current \TeX group.

<code>\clist_gpush:Nn</code>
<code>\clist_gpush:NV</code>
<code>\clist_gpush:No</code>
<code>\clist_gpush:Nx</code>
<code>\clist_gpush:cn</code>
<code>\clist_gpush:cV</code>
<code>\clist_gpush:co</code>
<code>\clist_gpush:cx</code>

`\clist_gpush:Nn $\langle comma\ list \rangle$ $\{ \langle item \rangle \}$`

Pushes the $\langle item \rangle$ onto the end of the top of the $\langle comma\ list \rangle$. The assignment is global.

108 Viewing comma lists

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

`\clist_show:N $\langle comma\ list \rangle$`

Displays the entries in the $\langle comma\ list \rangle$ in the terminal.

109 Experimental comma list functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

<code>\clist_length:N *</code>
<code>\clist_length:c *</code>
<code>\clist_length:n *</code>

`\clist_length:N $\langle comma\ list \rangle$`

Leaves the number of items in the $\langle comma\ list \rangle$ in the input stream as an $\langle integer \rangle$

denotation). The total number of items in a *⟨comma list⟩* will include those which are empty and duplicates, *i.e.* every item in a *⟨comma list⟩* is unique.

<code>\clist_item:Nn *</code>
<code>\clist_item:cn *</code>
<code>\clist_item:nn *</code>

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

Indexing items in the *⟨comma list⟩* from 0 at the top (left), this function will evaluate the *⟨integer expression⟩* and leave the appropriate item from the comma list in the input stream. If the *⟨integer expression⟩* is negative, indexing occurs from the bottom (right) of the comma list. When the *⟨integer expression⟩* is larger than the number of items in the *⟨comma list⟩* (as calculated by `\clist_length:N`) then the function will expand to nothing.

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

`\clist_set_from_seq:NN <comma list> <sequence>`

Sets the *⟨comma list⟩* within the current \TeX group to be equal to the content of the *⟨sequence⟩*.

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

`\clist_gset_from_seq:NN <comma list> <sequence>`

Sets the *⟨comma list⟩* globally to equal to the content of the *⟨sequence⟩*.

Part XIV

The l3prop package

Property lists

$\text{\LaTeX}3$ implements a “property list” data type, which contain an unordered list of entries each of which consists of a *⟨key⟩* and an associated *⟨value⟩*. The *⟨key⟩* and *⟨value⟩* may both be any *⟨balanced text⟩*. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique *⟨key⟩*: if an entry is added to a property list which already contains the *⟨key⟩* then the new entry will overwrite the existing one. The *⟨keys⟩* are compared on a string basis, using the same method as `\str_if_eq:nn`.

110 Creating and initialising property lists

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

`\prop_new:N <property list>`

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

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

`\prop_clear:N <property list>`

Clears all entries from the *<property list>* within the scope of the current TeX group.

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

`\prop_gclear:N <property list>`

Clears all entries from the *<property list>* globally.

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

`\prop_clear_new:N <property list>`

If the *<property list>* already exists, clears it within the scope of the current TeX group. If the *<property list>* is not defined, it will be created (using `\prop_new:N`). Thus the property list is guaranteed to be available and clear within the current TeX group. The *<property list>* will exist globally, but the content outside of the current TeX group is not specified.

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

`\prop_gclear_new:N <property list>`

If the *<property list>* already exists, clears it globally. If the *<property list>* is not defined, it will be created (using `\prop_new:N`). Thus the property list is guaranteed to be available and globally clear.

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

`\prop_set_eq:NN <property list1> <property list2>`

Sets the content of *<property list1>* equal to that of *<property list2>*. This assignment is restricted to the current TeX group level.

<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_gset_eq:NN <property list1> <property list2>`

Sets the content of *<property list1>* equal to that of *<property list2>*. This assignment is global and so is not limited by the current TeX group level.

111 Adding entries to property lists

```
\prop_put:Nnn
\prop_put:NnV
\prop_put:Nno
\prop_put:Nnx
\prop_put:NVn
\prop_put:NVV
\prop_put:Non
\prop_put:Noo
\prop_put:cnn
\prop_put:cnV
\prop_put:cno
\prop_put:cnx
\prop_put:cVn
\prop_put:cVV
\prop_put:con
\prop_put:coo
```

```
\prop_put:Nnn <property list> {<key>} {<value>}
```

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*. The assignment is restricted to the current T_EX group.

```
\prop_gput:Nnn
\prop_gput:NnV
\prop_gput:Nno
\prop_gput:Nnx
\prop_gput:NVn
\prop_gput:NVV
\prop_gput:Non
\prop_gput:Noo
\prop_gput:cnn
\prop_gput:cnV
\prop_gput:cno
\prop_gput:cnx
\prop_gput:cVn
\prop_gput:cVV
\prop_gput:con
\prop_gput:coo
```

```
\prop_gput:Nnn <property list> {<key>} {<value>}
```

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If

the $\langle key \rangle$ is already present in the $\langle property list \rangle$, the existing entry is overwritten by the new $\langle value \rangle$. The assignment is global.

$\backslash prop_put_if_new:Nnn$ $\backslash prop_put_if_new:cnn$	$\backslash prop_put_if_new:Nnn \langle property list \rangle \{ \langle key \rangle \} \{ \langle value \rangle \}$
--	---

If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then no action is taken. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then a new entry is added. Both the $\langle key \rangle$ and $\langle value \rangle$ may contain any *balanced text*. The $\langle key \rangle$ is stored after processing with $\backslash tl_to_str:n$, meaning that category codes are ignored. The assignment is restricted to the current TeX group.

$\backslash prop_gput_if_new:Nnn$ $\backslash prop_gput_if_new:cnn$	$\backslash prop_gput_if_new:Nnn \langle property list \rangle \{ \langle key \rangle \} \{ \langle value \rangle \}$
--	--

If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then no action is taken. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then a new entry is added. Both the $\langle key \rangle$ and $\langle value \rangle$ may contain any *balanced text*. The $\langle key \rangle$ is stored after processing with $\backslash tl_to_str:n$, meaning that category codes are ignored. The assignment is global.

112 Recovering values from property lists

$\backslash prop_get:NnN$ $\backslash prop_get:NVN$ $\backslash prop_get:NoN$ $\backslash prop_get:cnN$ $\backslash prop_get:cVN$ $\backslash prop_get:coN$	$\backslash prop_get:NnN \langle property list \rangle \{ \langle key \rangle \} \langle tl var \rangle$
--	---

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list variable \rangle$ will contain the special marker $\backslash q_no_value$. The $\langle token list variable \rangle$ is set within the current TeX group.

$\backslash prop_pop:NnN$ $\backslash prop_pop:NoN$ $\backslash prop_pop:cnN$ $\backslash prop_pop:coN$	$\backslash prop_pop:NnN \langle property list \rangle \{ \langle key \rangle \} \langle tl var \rangle$
--	---

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list$

variable) will contain the special marker `\q_no_value`. The $\langle key \rangle$ and $\langle value \rangle$ are then deleted from the property list. Both assignments are local.

<code>\prop_gpop:NnN</code>
<code>\prop_gpop:NoN</code>
<code>\prop_gpop:cnN</code>
<code>\prop_gpop:coN</code>

`\prop_gpop:NnN $\langle property list \rangle$ $\{ \langle key \rangle \}$ $\langle tl var \rangle$`

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list variable \rangle$ will contain the special marker `\q_no_value`. The $\langle key \rangle$ and $\langle value \rangle$ are then deleted from the property list. The $\langle property list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local.

113 Modifying property lists

<code>\prop_del:Nn</code>
<code>\prop_del:NV</code>
<code>\prop_del:cn</code>
<code>\prop_del:cV</code>

`\prop_del:Nn $\langle property list \rangle$ $\{ \langle key \rangle \}$`

Deletes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$ which may be accessed. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it. The deletion is restricted to the current \TeX group.

<code>\prop_gdel:Nn</code>
<code>\prop_gdel:NV</code>
<code>\prop_gdel:cn</code>
<code>\prop_gdel:cV</code>

`\prop_gdel:Nn $\langle property list \rangle$ $\{ \langle key \rangle \}$`

Deletes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$ which may be accessed. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it. The deletion is not restricted to the current \TeX group: it is global.

114 Property list conditionals

<code>\prop_if_empty_p:N *</code>
<code>\prop_if_empty:N\underline{TF} *</code>
<code>\prop_if_empty_p:c *</code>
<code>\prop_if_empty:c\underline{TF} *</code>

`\prop_if_empty_p:N $\langle property list \rangle$`
`\prop_if_empty:N \underline{TF} $\langle property list \rangle$`
 `$\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

Tests if the $\langle property list \rangle$ is empty (containing no entries). 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>\prop_if_in_p:Nn *</code>	
<code>\prop_if_in:NnTF *</code>	
<code>\prop_if_in_p:NV *</code>	
<code>\prop_if_in:NVTF *</code>	
<code>\prop_if_in_p:No *</code>	
<code>\prop_if_in:NoTF *</code>	
<code>\prop_if_in_p:cn *</code>	
<code>\prop_if_in:cnTF *</code>	
<code>\prop_if_in_p:cV *</code>	
<code>\prop_if_in:cVTF *</code>	
<code>\prop_if_in_p:co *</code>	
<code>\prop_if_in:coTF *</code>	

$\backslash\text{prop_if_in:NnTF } \langle property list \rangle \{ \langle key \rangle \}$
 $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nnTF`. 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.

TeXhackers note: This function iterates through every key–value pair in the $\langle property list \rangle$ and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

115 Mapping to property lists

<code>\prop_map_function:NN *</code>
<code>\prop_map_function:cn *</code>

$\backslash\text{prop_map_function:NN } \langle property list \rangle \langle function \rangle$

Applies $\langle function \rangle$ to every $\langle entry \rangle$ stored in the $\langle property list \rangle$. The $\langle function \rangle$ will receive two argument for each iteration.: the $\langle key \rangle$ and associated $\langle value \rangle$. The order in which $\langle entries \rangle$ are returned is not defined and should not be relied upon.

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

$\backslash\text{prop_map_inline:Nn } \langle property list \rangle \{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every $\langle entry \rangle$ stored within the $\langle property list \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle key \rangle$ as #1 and the $\langle value \rangle$ as #2. The order in which $\langle entries \rangle$ are returned is not defined and should not be relied upon.

<code>\prop_map_break: *</code>

$\backslash\text{prop_map_break:}$

Used to terminate a `\prop_map_...` function before all entries in the $\langle property list \rangle$

have been processed. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map_...` scenario will lead low level \TeX errors.

`\prop_map_break:n *`

`\prop_map_break:n {<tokens>}`

Used to terminate a `\prop_map_...` function before all entries in the *<property list>* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map_...` scenario will lead low level \TeX errors.

116 Viewing property lists

`\prop_show:N`
`\prop_show:c`

`\prop_show:N <property list>`

Displays the entries in the *<property list>* in the terminal.

117 Experimental property list functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

`\prop_get:NnTF` `\prop_get:NnTF <property list> {<key>}
<token list variable> {<true code>} {<false code>}`

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, stores the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle property list \rangle$. The $\langle true code \rangle$ is then left in the input stream. The $\langle token list variable \rangle$ is assigned locally.

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

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle property list \rangle$. The $\langle true code \rangle$ is then left in the input stream. Both the $\langle property list \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

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

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream and leaves the $\langle token list variable \rangle$ unchanged. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle property list \rangle$. The $\langle true code \rangle$ is then left in the input stream. The $\langle property list \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

`\prop_map_tokens:Nn *` `\prop_map_tokens:Nn <property list> {<code>}`

Analogue of `\prop_map_function:Nn` which maps several tokens instead of a single function. Useful in particular when mapping through a property list while keeping track of a given key.

`\prop_get:Nn *` `\prop_get:Nn <property list> {<key>}`

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

T_EXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`.

118 Internal property list functions

`\q_prop` The internal token used to separate out property list entries, separating both the $\langle key \rangle$ from the $\langle value \rangle$ and also one entry from another.

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

`\prop_split:Nnn` `\prop_spilt:Nnn` $\langle property list \rangle$ $\{ \langle key \rangle \}$ $\{ \langle code \rangle \}$
 Splits the $\langle property list \rangle$ at the $\langle key \rangle$, giving three groups: the $\langle extract \rangle$ of $\langle property list \rangle$ before the $\langle key \rangle$, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract \rangle$ of the $\langle property list \rangle$ after the $\langle value \rangle$. The first $\langle extract \rangle$ retains the internal structure of a property list. The second is only missing the leading separator `\q_prop`. This ensures that the concatenation of the two $\langle extracts \rangle$ is a property list. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then the second group will contain the marker `\q_no_value` and the third is empty. Once the split has occurred, the $\langle code \rangle$ is inserted followed by the three groups: thus the $\langle code \rangle$ should properly absorb three arguments. The $\langle key \rangle$ comparison takes place as described for `\str_if_eq:nn`.

`\prop_split:NnTF` `\prop_spilt:NnTF` $\langle property list \rangle$ $\{ \langle key \rangle \}$
 $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$
 Splits the $\langle property list \rangle$ at the $\langle key \rangle$, giving three groups: the $\langle extract \rangle$ of $\langle property list \rangle$ before the $\langle key \rangle$, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract \rangle$ of the $\langle property list \rangle$ after the $\langle value \rangle$. The first $\langle extract \rangle$ retains the internal structure of a property list. The second is only missing the leading separator `\q_prop`. This ensures that the concatenation of the two $\langle extracts \rangle$ is a property list. If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then the $\langle true code \rangle$ is left in the input stream, followed by the three groups: thus the $\langle true code \rangle$ should properly absorb three arguments. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then the $\langle false code \rangle$ is left in the input stream, with no trailing material. The $\langle key \rangle$ comparison takes place as described for `\str_if_eq:nn`.

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_`.

119 Creating and initialising boxes

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

`\box_new:N <box>`

Creates a new $\langle box \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle box \rangle$ will initially be void.

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

`\box_clear:N <box>`

Clears the content of the $\langle box \rangle$ by setting the box equal to `\c_void_box` within the current $\text{T}_{\text{E}}\text{X}$ group level.

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

`\box_gclear:N <box>`

Clears the content of the $\langle box \rangle$ by setting the box equal to `\c_void_box` globally.

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

`\box_clear_new:N <box>`

If the $\langle box \rangle$ is not defined, globally creates it. If the $\langle box \rangle$ is defined, clears the content of the $\langle box \rangle$ by setting the box equal to `\c_void_box` within the current $\text{T}_{\text{E}}\text{X}$ group level.

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

`\box_gclear_new:N <box>`

If the $\langle box \rangle$ is not defined, globally creates it. If the $\langle box \rangle$ is defined, clears the content of the $\langle box \rangle$ by setting the box equal to `\c_void_box` globally.

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

`\box_set_eq:NN <box1> <box2>`

Sets the content of $\langle box1 \rangle$ equal to that of $\langle box2 \rangle$. This assignment is restricted to the

current \TeX group level.

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

`\box_gset_eq:NN <box1> <box2>`

Sets the content of $\langle box1 \rangle$ equal to that of $\langle box2 \rangle$ globally.

<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>

`\box_set_eq_clear:NN <box1> <box2>`

Sets the content of $\langle box1 \rangle$ within the current \TeX group equal to that of $\langle box2 \rangle$, then clears $\langle box2 \rangle$ globally.

<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>

`\box_gset_eq_clear:NN <box1> <box2>`

Sets the content of $\langle box1 \rangle$ equal to that of $\langle box2 \rangle$, then clears $\langle box2 \rangle$. These assignments are global.

120 Using boxes

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

`\box_use:N <box>`

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting.

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

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

`\box_use_clear:N <box>`

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting, then globally clears the content of the $\langle box \rangle$.

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

<code>\box_move_right:nn</code> <code>\box_move_left:nn</code>

`\box_move_right:nn {<dimexpr>} {<box function>}`

This function operates in vertical mode, and inserts the material specified by the $\langle box \text{ function} \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box \text{ function} \rangle$ should be a box operation such as `\box_use:N \<box>` or a “raw” box specification such as `\vbox:n { xyz }`.

<code>\box_move_up:nn</code> <code>\box_move_down:nn</code>
--

`\box_move_up:nn {<dimexpr>} {<box function>}`

This function operates in horizontal mode, and inserts the material specified by the $\langle box \text{ function} \rangle$ such that its reference point is displaced vertical by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box \text{ function} \rangle$ should be a box operation such as `\box_use:N \<box>` or a “raw” box specification such as `\vbox:n { xyz }`.

121 Measuring and setting box dimensions

<code>\box_dp:N</code> <code>\box_dp:c</code>
--

`\box_dp:N <box>`

Calculates the depth (below the baseline) of the $\langle box \rangle$ and leaves this in the input stream. The output of this function is suitable for use in a $\langle dimension \text{ expression} \rangle$ for calculations.

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

<code>\box_ht:N</code> <code>\box_ht:c</code>
--

`\box_ht:N <box>`

Calculates the height (above the baseline) of the $\langle box \rangle$ and leaves this in the input stream. The output of this function is suitable for use in a $\langle dimension \text{ expression} \rangle$ for calculations.

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

<code>\box_wd:N</code> <code>\box_wd:c</code>
--

`\box_wd:N <box>`

Calculates the width of the $\langle box \rangle$ and leaves this in the input stream. The output of this function is suitable for use in a $\langle dimension \text{ expression} \rangle$ for calculations.

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

<code>\box_set_dp:Nn</code> <code>\box_set_dp:cn</code>
--

`\box_set_dp:Nn <box> {<dimension expression>}`

Set the depth (below the baseline) of the `<box>` to the value of the `{<dimension expression>}`. This is a global assignment.

<code>\box_set_ht:Nn</code> <code>\box_set_ht:cn</code>
--

`\box_set_ht:Nn <box> {<dimension expression>}`

Set the height (above the baseline) of the `<box>` to the value of the `{<dimension expression>}`. This is a global assignment.

<code>\box_set_wd:Nn</code> <code>\box_set_wd:cn</code>
--

`\box_set_wd:Nn <box> {<dimension expression>}`

Set the width of the `<box>` to the value of the `{<dimension expression>}`. This is a global assignment.

122 Box conditionals

<code>\box_if_empty_p:N *</code> <code>\box_if_empty:NTF *</code> <code>\box_if_empty_p:c *</code> <code>\box_if_empty:cTF *</code>
--

`\box_if_empty_p:N <box>`
`\box_if_empty:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a empty (equal to `\c_empty_box`). 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>\box_if_horizontal_p:N *</code> <code>\box_if_horizontal:NTF *</code> <code>\box_if_horizontal_p:c *</code> <code>\box_if_horizontal:cTF *</code>
--

`\box_if_horizontal_p:N <box>`
`\box_if_horizontal:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a horizontal box. 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>\box_if_vertical_p:N *</code> <code>\box_if_vertical:NTF *</code> <code>\box_if_vertical_p:c *</code> <code>\box_if_vertical:cTF *</code>
--

`\box_if_vertical_p:N <box>`
`\box_if_vertical:NTF <box> {<true code>} {<false code>}`

Tests if $\langle box \rangle$ is a vertical box. 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.

123 The last box inserted

`\l_last_box` This is a box containing the last item added to the current partial list, except in the case of the main vertical list (main galley), in which case this box is always void. Notice that although this is not a constant, it is *not* setttable by the programmer but is instead varied by T_EX.

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

124 Constant boxes

`\c_empty_box` This is a permanently empty box, which is neither set as horizontal nor vertical.

125 Scratch boxes

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

126 Viewing box contents

`\box_show:N`
`\box_show:c` `\box_show:N` $\langle box \rangle$

Writes the contents of $\langle box \rangle$ to the log file.

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

127 Horizontal mode boxes

`\hbox:n` `\hbox:n {⟨contents⟩}`

Typesets the `⟨contents⟩` into a horizontal box of natural width and then includes this box in the current list for typesetting.

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

`\hbox_to_wd:nn` `\hbox_to_wd:nn {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the `⟨contents⟩` into a horizontal box of width `⟨dimexpr⟩` and then includes this box in the current list for typesetting.

`\hbox_to_zero:n` `\hbox_to_zero:n {⟨contents⟩}`

Typesets the `⟨contents⟩` into a horizontal box of zero width and then includes this box in the current list for typesetting.

`\hbox_set:Nn`
`\hbox_set:cn` `\hbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the `⟨contents⟩` at natural width and then stores the result inside the `⟨box⟩`. The assignment is local.

`\hbox_gset:Nn`
`\hbox_gset:cn` `\hbox_gset:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the `⟨contents⟩` at natural width and then stores the result inside the `⟨box⟩`. The assignment is global.

`\hbox_set_to_wd:Nnn`
`\hbox_set_to_wd:cnn` `\hbox_set_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the `⟨contents⟩` to the width given by the `⟨dimexpr⟩` and then stores the result inside the `⟨box⟩`. The assignment is local.

`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn` `\hbox_gset_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the `⟨contents⟩` to the width given by the `⟨dimexpr⟩` and then stores the result inside the `⟨box⟩`. The assignment is global.

`\hbox_overlap_right:n` `\hbox_overlap_right:n {⟨contents⟩}`

Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the right of the insertion point.

<code>\hbox_overlap_left:n</code>	<code>\hbox_overlap_left:n {$\langle contents \rangle$}</code>
-----------------------------------	---

Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the left of the insertion point.

<code>\hbox_set_inline_begin:N</code>	<code>\hbox_set_inline_begin:N $\langle box \rangle$ $\langle contents \rangle$</code>
<code>\hbox_set_inline_begin:c</code>	
<code>\hbox_set_inline_end:</code>	

`\hbox_set_inline_end:`

Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. The assignment is local. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

<code>\hbox_gset_inline_begin:N</code>	<code>\hbox_gset_inline_begin:N $\langle box \rangle$ $\langle contents \rangle$</code>
<code>\hbox_gset_inline_begin:c</code>	
<code>\hbox_gset_inline_end:</code>	

`\hbox_gset_inline_end:`

Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. The assignment is global. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

<code>\hbox_unpack:N</code>	<code>\hbox_unpack:N $\langle box \rangle$</code>
<code>\hbox_unpack:c</code>	

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

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

<code>\hbox_unpack_clear:N</code>	<code>\hbox_unpack_clear:N $\langle box \rangle$</code>
<code>\hbox_unpack_clear:c</code>	

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

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

128 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box will have no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the later will typically be non-zero.

<code>\vbox:n</code>

`\vbox:n {<contents>}`

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting.

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

<code>\vbox_top:n</code>

`\vbox_top:n {<contents>}`

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box will be equal to that of the *first* item added to the box.

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

<code>\vbox_to_ht:nn</code>

`\vbox_to_ht:n {<contents>} {<dimexpr>}`

Typesets the `<contents>` into a vertical box of height `<dimexpr>` and then includes this box in the current list for typesetting.

<code>\vbox_to_zero:n</code>

`\vbox_to_zero:n {<contents>}`

Typesets the `<contents>` into a vertical box of zero height and then includes this box in the current list for typesetting.

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

`\vbox_set:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. The assignment is local.

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

`\vbox_gset:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. The

assignment is global.

<code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code>	<code>\vbox_set_top:Nn <box> {<contents>}</code>
--	--

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box will be equal to that of the *first* item added to the box. The assignment is local.

<code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code>	<code>\vbox_gset_top:Nn <box> {<contents>}</code>
--	---

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box will be equal to that of the *first* item added to the box. The assignment is global.

<code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code>	<code>\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code>
--	---

Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. The assignment is local.

<code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code>	<code>\vbox_gset_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code>
--	--

Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. The assignment is global.

<code>\vbox_set_inline_begin:N</code> <code>\vbox_set_inline_begin:c</code> <code>\vbox_set_inline_end:</code>	<code>\vbox_set_inline_begin:N <box> <contents></code> <code>\vbox_set_inline_end:</code>
--	--

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The assignment is local. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

<code>\vbox_gset_inline_begin:N</code> <code>\vbox_gset_inline_begin:c</code> <code>\vbox_gset_inline_end:</code>	<code>\vbox_gset_inline_begin:N <box> <contents></code> <code>\vbox_gset_inline_end:</code>
---	--

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The assignment is global. In contrast to `\vbox_set:Nn` this function does not absorb

the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

<code>\vbox_set_split_to_ht:NNn</code>
--

`\vbox_set_split_to_ht:NNn $\langle box1 \rangle$ $\langle box2 \rangle$ { $\langle dimexpr \rangle$ }`

Sets $\langle box1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box2 \rangle$ (which must be a vertical box).

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

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

`\vbox_unpack:N $\langle box \rangle$`

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

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

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

`\vbox_unpack:N $\langle box \rangle$`

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

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

129 Primitive box conditionals

<code>\if_hbox:N</code>

`\if_hbox:N $\langle box \rangle$
 $\langle true code \rangle$
\else:
 $\langle false code \rangle$

<code>\if_hbox:N</code>

\fi:`

Tests if $\langle box \rangle$ is a horizontal box.

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

```

\if_vbox:N <box>
  <true code>
\else:
  <false code>
\if_vbox:N * \fi:

```

Tests is $\langle box \rangle$ is a vertical box.

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

```

\if_box_empty:N <box>
  <true code>
\else:
  <false code>
\if_box_empty:N * \fi:

```

Tests is $\langle box \rangle$ is an empty (void) box.

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

Part XVI

The l3io package

Input–output operations

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).

130 Opening and closing streams

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

`\ior_open:Nn <stream> {\<file name>}`

Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a `\ior_close:N` instruction is given or the file ends.

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

`\iow_open:Nn <stream> {\<file name>}`

Opens $\langle file\ name \rangle$ for writing using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a `\iow_close:N` instruction is given or the file ends. Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive).

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

`\ior_close:N <stream>`

Closes the $\langle stream \rangle$. Streams should always be closed when they are finished with as this ensures that they remain available to other programmer. The name of the $\langle stream \rangle$ will be freed at this stage, to ensure that any further attempts to read from it results in an error.

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

`\iow_close:N <stream>`

Closes the $\langle stream \rangle$. Streams should always be closed when they are finished with as this ensures that they remain available to other programmer. The name of the $\langle stream \rangle$ will be freed at this stage, to ensure that any further attempts to write to it results in an error.

<code>\ior_list_streams:</code>
<code>\iow_list_streams:</code>

`\ior_list_streams:`

`\iow_list_streams:`

Displays a list of the file names associated with each open stream: intended for tracking down problems.

131 Writing to files

<code>\iow_now:Nn</code>
<code>\iow_now:Nx</code>

`\iow_now:Nn <stream> {\<tokens>}`

This function writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

TeXhackers note: `\iow_now:Nx` is a protected macro which expands to the two TeX primitives `\immediate\write`.

<code>\iow_log:n</code>
<code>\iow_log:x</code>

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

This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

<code>\iow_term:n</code>
<code>\iow_term:x</code>

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

This function writes the given $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

<code>\iow_now_when_avail:Nn</code>
<code>\iow_now_when_avail:Nx</code>

`\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 `\iow_now:Nn`. If the $\langle stream \rangle$ is not open, the $\langle tokens \rangle$ are simply thrown away.

<code>\iow_shipout:Nn</code>
<code>\iow_shipout:Nx</code>

`\iow_shipout:Nn $\langle stream \rangle$ { $\langle tokens \rangle$ }`

This function writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The x-type variants expand the $\langle tokens \rangle$ at the point where the function is used but *not* when the resulting tokens are written to the $\langle stream \rangle$ (*cf.* `\iow_shipout_x:Nn`).

<code>\iow_shipout_x:Nn</code>
<code>\iow_shipout_x:Nx</code>

`\iow_shipout_x:Nn $\langle stream \rangle$ { $\langle tokens \rangle$ }`

This function writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

TeXhackers note: `\iow_shipout_x:Nn` is the TeX primitive `\write` renamed.

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

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

Inserts $\langle token \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_stream { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

`\iow_newline: *` `\iow_newline:`

Function to add a new line within the *<tokens>* written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

132 Wrapping lines in output

`\iow_wrap:xnnnN` `\iow_wrap:xnnnN {<text>} {<run-on text>} {<run-on length>} {<set up>} <function>`

This function will wrap the *<text>* to a fixed number of characters per line. At the start of each line which is wrapped, the *<run-on text>* will be inserted. The line length targeted will be the value of `\l_iow_line_length_int` minus the *<run-on length>*. The later value should be the number of characters in the *<run-on text>*. Additional functions may be added to the wrapping by using the *<set up>*, which is executed before the wrapping takes place. The result of the wrapping operation is passed as a braced argument to the *<function>*, which will typically be a wrapper around a writing operation. Within the *<text>*, `\\` may be used to force a new line and `\` may be used to represent a forced space (for example after a control sequence). Both the wrapping process and the subsequent write operation will perform x-type expansion. For this reason, material which is to be written “as is” should be given as the argument to `\token_to_str:N` or `\tl_to_str:n` (as appropriate) within the *<text>*. The output of `\iow_wrap:xnnnN` (*i.e.* the argument passed to the *<function>*) will consist of characters of category code 12 (other) and 10 (space) only. This means that the output will *not* expand further when written to a file.

`\l_iow_line_length_int` The maximum length of a line to be written by the `\iow_wrap:xnnnN` function. This value depends on the T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EXlive and MiK_TE_X systems.

133 Reading from files

`\ior_to:NN` `\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>* and stores the result locally in the *<token list>* variable.

If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ will be tokenized by T_EX according to the category codes in force when the function is used.

T_EXhackers note: The is protected macro which expands to the T_EX primitive `\read` along with the `to` keyword.

`\ior_gto:NN` `\ior_gto:NN $\langle stream \rangle$ $\langle token list variable \rangle$`

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input $\langle stream \rangle$ and stores the result globally in the $\langle token list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ will be tokenized by T_EX according to the category codes in force when the function is used.

T_EXhackers note: The is protected macro which expands to the T_EX primitives `\global \read` along with the `to` keyword.

`\ior_str_to:NN` `\ior_str_to:NN $\langle stream \rangle$ $\langle token list variable \rangle$`

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space).

T_EXhackers note: The is protected macro which expands to the ϵ -T_EX primitive `\readline` along with the `to` keyword.

`\ior_str_gto:NN` `\ior_str_gto:NN $\langle stream \rangle$ $\langle token list variable \rangle$`

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input $\langle stream \rangle$ and stores the result globally in the $\langle token list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space).

T_EXhackers note: The is protected macro which expands to the primitives `\global \readline` along with the `to` keyword.

`\ior_if_eof_p:N *` `\ior_if_eof_p:N $\langle stream \rangle$`
`\ior_if_eof:NTF *` `\ior_if_eof:NTF $\langle stream \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test will

also return a `true` value if the $\langle stream \rangle$ is not open or the $\langle file\ name \rangle$ associated with a $\langle stream \rangle$ does not exist at all. 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.

134 Internal input–output functions

```

\if_eof:w \langle stream \rangle
  \langle true code \rangle
\else:
  \langle false code \rangle
\if_eof:w ★ \fi:

```

Tests if the $\langle stream \rangle$ returns “end of file”, which is true for non-existent files. The `\else:` branch is optional.

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

```

\ior_raw_new:N
\ior_raw_new:c \ior_raw_new:N \langle stream \rangle

```

Directly allocates a new stream for reading, bypassing the stack system. This is to be used only when a new stream is required at a T_EX level, when a new stream is requested by the stack itself.

```

\iow_raw_new:N
\iow_raw_new:c \iow_raw_new:N \langle stream \rangle

```

Directly allocates a new stream for writing, bypassing the stack system. This is to be used only when a new stream is required at a T_EX level, when a new stream is requested by the stack itself.

Part XVII

The l3msg package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type 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.

135 Creating new messages

All messages have to be created before they can be used. All message setting is local, with the general assumption that messages will be managed as part of module set up outside of any \TeX grouping.

The text of messages will automatically be wrapped to the length available in the console. As a result, formatting is only needed where it will help to show meaning. In particular, `\` may be used to force a new line and `_` forces an explicit space.

<code>\msg_new:nnnn</code> <code>\msg_new:nnn</code>	<code>\msg_new:nnnn {<module>} {<message>} {<text>}</code> <code>{<more text>}</code>
---	--

Creates a *<message>* for a given *<module>*. The message will be defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used. Within the *<text>* and *<more text>* `\` can be used to start a new line. An error will be raised if the *<message>* already exists.

<code>\msg_set:nnnn</code> <code>\msg_set:nnn</code>	<code>\msg_set:nnnn {<module>} {<message>} {<text>}</code> <code>{<more text>}</code>
---	--

Sets up the text for a *<message>* for a given *<module>*. The message will be defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used. Within the *<text>* and *<more text>* `\` can be used to start a new line.

136 Contextual information for messages

`\msg_line_context: *` `\msg_line_context:`

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text `on line`.

`\msg_line_number: *` `\msg_line_number:`

Prints the current line number when a message is given.

`\c_msg_return_text_tl`

Standard text to indicate that the user should try pressing `<return>` to continue. The standard definition reads:

Try typing `<return>` to proceed.

If that doesn't work, type `X <return>` to quit.

`\c_msg_trouble_text_tl`

Standard text to indicate that the more errors are likely and that aborting the run is advised. The standard definition reads:

More errors will almost certainly follow:
the LaTeX run should be aborted.

`\msg_fatal_text:n *`

`\msg_fatal_text:n {<module>}`

Produces the standard text:

Fatal `<module>` error

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the `<module>` to be included.

`\msg_critical_text:n *`

`\msg_critical_text:n {<module>}`

Produces the standard text:

Critical `<module>` error

This function can be redefined to alter the language in which the message is give, using #1 as the name of the *⟨module⟩* to be included.

```
\msg_error_text:n ★ \msg_error_text:n {⟨module⟩}
```

Produces the standard text:

```
<module> error
```

This function can be redefined to alter the language in which the message is give, using #1 as the name of the *⟨module⟩* to be included.

```
\msg_warning_text:n ★ \msg_warning_text:n {⟨module⟩}
```

Produces the standard text:

```
<module> warning
```

This function can be redefined to alter the language in which the message is give, using #1 as the name of the *⟨module⟩* to be included.

```
\msg_info_text:n ★ \msg_info_text:n {⟨module⟩}
```

Produces the standard text:

```
<module> info
```

This function can be redefined to alter the language in which the message is give, using #1 as the name of the *⟨module⟩* to be included.

137 Issuing messages

Messages behave differently depending on the message class. A number of standard message classes are supplied, but more can be created.

When issuing messages, any arguments passed should use `\tl_to_str:n` or `\token_to_str:N` to prevent unwanted expansion of the material.

```
\msg_class_set:nn \msg_class_set:nn {⟨class⟩} {⟨code⟩}
```

Sets a *⟨class⟩* to output a message, using *⟨code⟩* to process the message text. The *⟨class⟩*

should be a text value, while the $\langle code \rangle$ may be any arbitrary material. The $\langle code \rangle$ will receive 6 arguments: the module name (#1), the message name (#2) and the four arguments taken by the message text (#3 to #6).

The kernel 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).

<pre> \msg_fatal:nnxxxx \msg_fatal:nnxxx \msg_fatal:nnxx \msg_fatal:nnx \msg_fatal:nn </pre>	<pre> \msg_fatal:nnxxxx {\langle module \rangle} {\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle} </pre>
--	--

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the T_EX run will halt.

<pre> \msg_critical:nnxxxx \msg_critical:nnxxx \msg_critical:nnxx \msg_critical:nnx \msg_critical:nn </pre>	<pre> \msg_critical:nnxxxx {\langle module \rangle} {\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle} </pre>
---	---

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing the message reading the current input file will stop. This may halt the T_EX run (if the current file is the main file) or may abort reading a sub-file.

<pre> \msg_error:nnxxxx \msg_error:nnxxx \msg_error:nnxx \msg_error:nnx \msg_error:nn </pre>	<pre> \msg_error:nnxxxx {\langle module \rangle} {\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle} </pre>
--	--

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue.

<pre> \msg_warning:nnxxxx \msg_warning:nnxxx \msg_warning:nnxx \msg_warning:nnx \msg_warning:nn </pre>	<pre> \msg_warning:nnxxxx {\langle module \rangle} {\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle} </pre>
--	--

Issues $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating

functions. The warning text will be added to the log file, but the T_EX run will not be interrupted.

<code>\msg_info:nnxxxx</code>	
<code>\msg_info:nnxxx</code>	
<code>\msg_info:nnxx</code>	
<code>\msg_info:nnx</code>	
<code>\msg_info:nn</code>	<code>\msg_info:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text will be added to the log file.

<code>\msg_log:nnxxxx</code>	
<code>\msg_log:nnxxx</code>	
<code>\msg_log:nnxx</code>	
<code>\msg_log:nnx</code>	
<code>\msg_log:nn</code>	<code>\msg_log:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text will be added to the log file: the output is briefer than `\msg_info:nnxxxx`.

<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	<code>\msg_none:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

138 Redirecting messages

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
-------------------------------------	---

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.

<code>\msg_redirect_module:nnn</code>	<code>\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}</code>
---------------------------------------	--

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

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

<code>\msg_redirect_name:nnn</code>	<code>\msg_redirect_name:nn {$\langle module\rangle$} {$\langle message\rangle$} {$\langle class\rangle$}</code>
-------------------------------------	---

Redirects a specific $\langle message\rangle$ from a specific $\langle module\rangle$ to act as a member of $\langle class\rangle$ of messages. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

139 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

<code>\msg_newline:</code>	*
<code>\msg_two_newlines:</code>	*

`\msg_newline:`

Forces a new line in a message. This is a low-level function, which will not include any additional printing information in the message: contrast with `\\` in messages. The `two` version adds two lines.

<code>\msg_interrupt:xxx</code>	<code>\msg_interrupt:xxx {$\langle first\ line\rangle$} {$\langle text\rangle$} {$\langle extra\ text\rangle$}</code>
---------------------------------	--

Interrupts the T_EX run, issuing a formatted message comprising $\langle first\ line\rangle$ and $\langle text\rangle$ laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!.....
```

where the $\langle text \rangle$ will be wrapped to fit within the current line length. The user may then request more information, at which stage the $\langle extra\ text \rangle$ will be shown in the terminal in the format

```
|,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,  
|  <extra text>  
|.....
```

where the $\langle extra\ text \rangle$ will be wrapped to fit within the current line length.

`\msg_log:x` `\msg_log:x { $\langle text \rangle$ }`
Writes to the log file with the $\langle text \rangle$ laid out in the format

```
.....  
.  <text>  
.....
```

where the $\langle text \rangle$ will be wrapped to fit within the current line length.

`\msg_term:x` `\msg_term:x { $\langle text \rangle$ }`
Writes to the terminal and log file with the $\langle text \rangle$ laid out in the format

```
*****  
*  <text>  
*****
```

where the $\langle text \rangle$ will be wrapped to fit within the current line length.

140 Kernel-specific functions

Messages from L^AT_EX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

`\msg_kernel_new:nnnn` `\msg_kernel_new:nnnn { $\langle module \rangle$ } { $\langle message \rangle$ } { $\langle text \rangle$ }`
`\msg_kernel_new:nnn` `{ $\langle more\ text \rangle$ }`

Creates a kernel $\langle message \rangle$ for a given $\langle module \rangle$. The message will be defined to first give $\langle text \rangle$ and then $\langle more\ text \rangle$ if the user requests it. If no $\langle more\ text \rangle$ is available then

a standard text is given instead. Within $\langle text \rangle$ and $\langle more\ text \rangle$ four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used. Within the $\langle text \rangle$ and $\langle more\ text \rangle$ $\backslash\backslash$ can be used to start a new line. An error will be raised if the $\langle message \rangle$ already exists.

$\backslash msg_kernel_set:nnnn$ $\backslash msg_kernel_set:nnn$	$\backslash msg_kernel_set:nnnn \{\langle module \rangle\} \{\langle message \rangle\} \{\langle text \rangle\}$ $\{\langle more\ text \rangle\}$
---	--

Sets up the text for a kernel $\langle message \rangle$ for a given $\langle module \rangle$. The message will be defined to first give $\langle text \rangle$ and then $\langle more\ text \rangle$ if the user requests it. If no $\langle more\ text \rangle$ is available then a standard text is given instead. Within $\langle text \rangle$ and $\langle more\ text \rangle$ four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used. Within the $\langle text \rangle$ and $\langle more\ text \rangle$ $\backslash\backslash$ can be used to start a new line.

$\backslash msg_kernel_fatal:nnxxxx$ $\backslash msg_kernel_fatal:nnxxx$ $\backslash msg_kernel_fatal:nnxx$ $\backslash msg_kernel_fatal:nnx$ $\backslash msg_kernel_fatal:nn$	$\backslash msg_kernel_fatal:nnxxxx \{\langle module \rangle\} \{\langle message \rangle\} \{\langle arg\ one \rangle\}$ $\{\langle arg\ two \rangle\} \{\langle arg\ three \rangle\} \{\langle arg\ four \rangle\}$
--	---

Issues kernel $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg\ one \rangle$ to $\langle arg\ four \rangle$ to the text-creating functions. After issuing a fatal error the T_EX run will halt. Cannot be redirected.

$\backslash msg_kernel_error:nnxxxx$ $\backslash msg_kernel_error:nnxxx$ $\backslash msg_kernel_error:nnxx$ $\backslash msg_kernel_error:nnx$ $\backslash msg_kernel_error:nn$	$\backslash msg_kernel_error:nnxxxx \{\langle module \rangle\} \{\langle message \rangle\} \{\langle arg\ one \rangle\}$ $\{\langle arg\ two \rangle\} \{\langle arg\ three \rangle\} \{\langle arg\ four \rangle\}$
--	---

Issues kernel $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg\ one \rangle$ to $\langle arg\ four \rangle$ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

$\backslash msg_kernel_warning:nnxxxx$ $\backslash msg_kernel_warning:nnxxx$ $\backslash msg_kernel_warning:nnxx$ $\backslash msg_kernel_warning:nnx$ $\backslash msg_kernel_warning:nn$	$\backslash msg_kernel_warning:nnxxxx \{\langle module \rangle\} \{\langle message \rangle\} \{\langle arg\ one \rangle\}$ $\{\langle arg\ two \rangle\} \{\langle arg\ three \rangle\} \{\langle arg\ four \rangle\}$
--	---

Issues kernel $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg\ one \rangle$ to $\langle arg\ four \rangle$ to the text-creating functions. The warning text will be added to the log file, but the T_EX run will

not be interrupted.

<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:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
--	--

Issues kernel $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file.

Part XVIII

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).

141 Parsing key–value lists

The low-level parsing system converts a $\langle key\text{--}value\ list \rangle$ into $\langle keys \rangle$ and associated $\langle values \rangle$. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This is not carried out by the low-level parser, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (*i.e.* two arguments), and a second function if required for keys given without arguments (*i.e.* a single argument).

The parser does not double # tokens or expand any input. The tokens = and , are corrected so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Values which are given in braces will have exactly one set removed, thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

<code>\keyval_parse:NNn</code>	<code>\keyval_parse:NNn <function1> <function2> {<key-value list>}</code>
--------------------------------	---

Parses the *<key-value list>* into a series of *<keys>* and associated *<values>*, or keys alone (if no *<value>* was given). *<function1>* should take one argument, while *<function2>* should absorb two arguments. After `\keyval_parse:NNn` has parsed the *<key-value list>*, *<function1>* will be used to process keys given with no value and *<function2>* will be used to process keys given with a value. The order of the *<keys>* in the *<key-value list>* will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

will be converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all).

Part XIX

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. 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}.
```

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { 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` will be used within a document function, for example

```
\DeclareDocumentCommand \SomePackageSetup { m }
{ \keys_set:nn { module } { #1 } }
\DeclareDocumentCommand \SomePackageMacro { o m }
{ \keys_set:nn { module } { #1 } }
```

```

{
  \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 143, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```

\tl_set:Nn \l_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`.

142 Creating keys

`\keys_define:nn` `\keys_define:nn {<module>} {<keyval list>}`

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name should be a text value, but there are no restrictions on the nature of the text. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The *<keyval list>* should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```

\keys_define:nn { mymodule }
{
  keyname .code:n = Some~code~using~#1,
  keyname .value_required:
}

```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require exactly one argument. This is indicated in the name of the property using an argument specification.

In the following discussion, each property is illustrated attached to an arbitrary $\langle key \rangle$, which when used may be supplied with a $\langle value \rangle$. All key *definitions* are local.

```
.bool_set:N  $\langle key \rangle$  .bool_set:N =  $\langle boolean \rangle$ 
```

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either **true** or **false**). If the variable does not exist, it will be created at the point that the key is set up. The $\langle boolean \rangle$ will be assigned locally.

```
.bool_gset:N  $\langle key \rangle$  .bool_gset:N =  $\langle boolean \rangle$ 
```

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either **true** or **false**). If the variable does not exist, it will be created at the point that the key is set up. The $\langle boolean \rangle$ will be assigned globally.

```
.choice:  $\langle key \rangle$  .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 144.

```
.choice_code:n  
.choice_code:x  $\langle key \rangle$  .choice_code:n =  $\langle code \rangle$ 
```

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` will expand to the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list given to `.generate_choices:n`. Choices are discussed in detail in section 144.

```
.code:n  
.code:x  $\langle key \rangle$  .code:n =  $\langle code \rangle$ 
```

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (**#1**), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The x-type variant will expand $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

```
.default:n  
.default:V  $\langle key \rangle$  .default:n =  $\langle default \rangle$ 
```

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 '
}

```

```


.dim_set:N
    .dim_set:c


⟨key⟩ .dim_set:N = ⟨dimension⟩

```

Defines $\langle key \rangle$ to set $\langle dimension \rangle$ to $\langle value \rangle$ (which must a dimension expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle dimension \rangle$ will be assigned locally.

```


.dim_gset:N
    .dim_gset:c


⟨key⟩ .dim_gset:N = ⟨dimension⟩

```

Defines $\langle key \rangle$ to set $\langle dimension \rangle$ to $\langle value \rangle$ (which must a dimension expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle dimension \rangle$ will be assigned globally.

```


.fp_set:N
    .fp_set:c


⟨key⟩ .fp_set:N = ⟨floating point⟩

```

Defines $\langle key \rangle$ to set $\langle floating\ point \rangle$ to $\langle value \rangle$ (which must a integer expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle integer \rangle$ will be assigned locally.

```


.fp_gset:N
    .fp_gset:c


⟨key⟩ .fp_gset:N = ⟨floating point⟩

```

Defines $\langle key \rangle$ to set $\langle floating-point \rangle$ to $\langle value \rangle$ (which must a integer expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle integer \rangle$ will be assigned globally.

```


.generate_choices:n


⟨key⟩ .generate_choices:n = {⟨list⟩}

```

This property will mark $\langle key \rangle$ as a multiple choice key, and will use the $\langle list \rangle$ to define the choices. The $\langle list \rangle$ should consist of a comma-separated list of choice names. Each choice will be set up to execute $\langle code \rangle$ as set using `.choice_code:n` (or `.choice_code:x`). Choices are discussed in detail in [section 144](#).

```


.int_set:N
    .int_set:c


⟨key⟩ .int_set:N = ⟨integer⟩

```

Defines $\langle key \rangle$ to set $\langle integer \rangle$ to $\langle value \rangle$ (which must a integer expression). If the variable

does not exist, it will be created at the point that the key is set up. The $\langle integer \rangle$ will be assigned locally.

<code>.int_gset:N</code>
<code>.int_gset:c</code>

 $\langle key \rangle .int_gset:N = \langle integer \rangle$

Defines $\langle key \rangle$ to set $\langle integer \rangle$ to $\langle value \rangle$ (which must a integer expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle integer \rangle$ will be assigned globally.

<code>.meta:n</code>
<code>.meta:x</code>

 $\langle key \rangle .meta:n = \{\langle keyval list \rangle\}$

Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ 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**).

<code>.skip_set:N</code>
<code>.skip_set:c</code>

 $\langle key \rangle .skip_set:N = \langle skip \rangle$

Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must a skip expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle skip \rangle$ will be assigned locally.

<code>.skip_gset:N</code>
<code>.skip_gset:c</code>

 $\langle key \rangle .skip_gset:N = \langle skip \rangle$

Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must a skip expression). If the variable does not exist, it will be created at the point that the key is set up. The $\langle skip \rangle$ will be assigned globally.

<code>.tl_set:N</code>
<code>.tl_set:c</code>

 $\langle key \rangle .tl_set:N = \langle token list variable \rangle$

Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$. If the variable does not exist, it will be created at the point that the key is set up. The $\langle token list variable \rangle$ will be assigned locally.

<code>.tl_gset:N</code>
<code>.tl_gset:c</code>

 $\langle key \rangle .tl_gset:N = \langle token list variable \rangle$

Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$. If the variable does not exist, it will be created at the point that the key is set up. The $\langle token list variable \rangle$ will be assigned globally.

<code>.tl_set_x:N</code>
<code>.tl_set_x:c</code>

 $\langle key \rangle .tl_set_x:N = \langle token list variable \rangle$

Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$, which will be subjected to an **x**-type

expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it will be created at the point that the key is set up. The *token list variable* will be assigned locally.

<code>.tl_gset_x:N</code> <code>.tl_gset_x:c</code>	$\langle key \rangle$ <code>.tl_gset_x:N =</code> <i>token list variable</i>
--	--

Defines $\langle key \rangle$ to set *token list variable* to $\langle value \rangle$, which will be subjected to an `x`-type expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it will be created at the point that the key is set up. The *token list variable* will be assigned globally.

<code>.value_forbidden:</code>	$\langle key \rangle$ <code>.value_forbidden:</code>
--------------------------------	--

Specifies that $\langle key \rangle$ cannot receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is given then an error will be issued.

<code>.value_required:</code>	$\langle key \rangle$ <code>.value_required:</code>
-------------------------------	---

Specifies that $\langle key \rangle$ must receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is not given then an error will be issued.

143 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.

144 Multiple choice keys

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 \c_space_tl
    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.

<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).

145 Setting keys

```

\keys_set:nn
\keys_set:nV
\keys_set:nv
\keys_set:no \keys_set:nn {<module>} {<keyval list>}

```

Parses the `<keyval list>`, and sets those keys which are defined for `<module>`. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this will be illustrated later. 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_key_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`.

`\l_keys_path_tl` When processing an unknown key, the path of the key used is available as `\l_keys_path_tl`. Note that this will have been processed using `\tl_to_str:n`.

146 Utility functions for keys

```

\keys_if_exist_p:nn * \keys_if_exist:nn <module> <key>
\keys_if_exist:nnTF * \keys_if_exist:nn <module> <key>
                    {<true code>} {<false code>}

```

Tests if the $\langle key \rangle$ exists for $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle$. 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.

`\keys_show:nn` `\keys_show:nn { $\langle module \rangle$ } { $\langle key \rangle$ }`

Shows the function which is used to actually implement a $\langle key \rangle$ for a $\langle module \rangle$.

Part XX

The l3file package

File operations

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 T_EX will attempt to locate files using both the operating system path and entries in the T_EX file database (most T_EX systems use such a database). Thus the “current path” for T_EX is somewhat broader than that for other programs.

147 File operation functions

`\g_file_current_name_tl` Contains the name of the current L^AT_EX 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 L^AT_EX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF` `\file_if_exist:nTF { $\langle file\ name \rangle$ } { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }`
Searches for $\langle file\ name \rangle$ using the current T_EX search path and the additional paths controlled by `\file_path_include:n`. 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.

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 {<file name>} <tl var>`

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found sets the `<tl var>` the fully-qualified name of the file, *i.e.* the path and file name. If the file is not found then the `<tl var>` will be empty.

T_EXhackers note: The `<file name>` 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 {<file name>}`

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L^AT_EX source. All files read are recorded for information and the file name stack is updated by this function.

T_EXhackers note: The `<file name>` 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 {<path>}`

Adds `<path>` 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 {<path>}`

Removes `<path>` 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.

148 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 $\text{\LaTeX} 2_{\epsilon}$, there is a need to save the search path so that `\input@path` can be used as appropriate.

Part XXI

The `l3fp` package

Floating-point operations

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.

149 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 T_EX 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 T_EX 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 T_EX

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.

150 Conversion of floating point values 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.

151 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>	<code>\fp_round_figures:Nn</code> <i><floating point variable></i> $\{\langle target \rangle\}$
--	---

Rounds the *<floating point variable>* to the *<target>* number of significant figures (an integer expression). The rounding is carried out locally.

<code>\fp_ground_figures:Nn</code> <code>\fp_ground_figures:cn</code>	<code>\fp_ground_figures:Nn</code> <i><floating point variable></i> $\{\langle target \rangle\}$
--	--

Rounds the *<floating point variable>* to the *<target>* number of significant figures (an integer expression). The rounding is carried out globally.

<code>\fp_round_places:Nn</code> <code>\fp_round_places:cn</code>	<code>\fp_round_places:Nn</code> <i><floating point variable></i> $\{\langle target \rangle\}$
--	--

Rounds the *<floating point variable>* to the *<target>* number of decimal places (an integer expression). The rounding is carried out locally.

<code>\fp_ground_places:Nn</code> <code>\fp_ground_places:cn</code>	<code>\fp_ground_places:Nn</code> <i><floating point variable></i> $\{\langle target \rangle\}$
--	---

Rounds the *<floating point variable>* to the *<target>* number of decimal places (an integer expression). The rounding is carried out globally.

152 Floating-point conditionals

<code>\fp_if_undefined_p:N *</code> <code>\fp_if_undefined:NTF *</code>	<code>\fp_if_undefined_p:N</code> <i><fixed-point></i> <code>\fp_if_undefined:NTF</code> <i><fixed-point></i> $\{\langle true code \rangle\} \{\langle false code \rangle\}$
--	--

Tests if *<floating point>* is undefined (*i.e.* equal to the special `\c_undefined_fp` variable). 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>\fp_if_zero:N *</code>	<code>\fp_if_zero_p:N</code> <i><fixed-point></i> <code>\fp_if_zero:NTF</code> <i><fixed-point></i> $\{\langle true code \rangle\} \{\langle false code \rangle\}$
------------------------------	---

Tests if *<floating point>* is equal to zero (*i.e.* equal to the special `\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.

`\fp_compare:nNnTF`

`\fp_compare:nNnTF`

$$\{ \langle floating\ point_1 \rangle \} \langle relation \rangle \{ \langle floating\ point_2 \rangle \}$$

$$\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$$

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 $\langle true\ code \rangle$ or $\langle false\ code \rangle$ 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.

`\fp_compare:nTF`

`\fp_compare:nTF`

$$\{ \langle floating\ point_1 \rangle \langle relation \rangle \langle floating\ point_2 \rangle \}$$

$$\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$$

This function compared the two $\langle floating\ point \rangle$ values, which may be stored as `fp` variables, using the $\langle relation \rangle$:

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

Either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ 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.

153 Unary floating-point operations

The unary operations alter the value stored within an `fp` variable.

`\fp_abs:N`
`\fp_abs:c`

`\fp_abs:N` $\langle floating\ point\ variable \rangle$

Converts the $\langle floating\ point\ variable \rangle$ to its absolute value, assigning the result within

the current \TeX group.

\fp_gabs:N
\fp_gabs:c

 \fp_gabs:N *\langle floating point variable \rangle*

Converts the *\langle floating point variable \rangle* to its absolute value, assigning the result globally.

\fp_neg:N
\fp_neg:c

 \fp_neg:N *\langle floating point variable \rangle*

Reverse the sign of the *\langle floating point variable \rangle* , assigning the result within the current \TeX group.

\fp_gneg:N
\fp_gneg:c

 \fp_gneg:N *\langle floating point variable \rangle*

Reverse the sign of the *\langle floating point variable \rangle* , assigning the result globally.

154 Floating-point arithmetic

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).

\fp_add:Nn
\fp_add:cn

 \fp_add:Nn *\langle floating point \rangle* $\{ \langle value \rangle \}$

Adds the *$\langle value \rangle$* to the *\langle floating point \rangle* , making the assignment within the current \TeX group level.

\fp_gadd:Nn
\fp_gadd:cn

 \fp_gadd:Nn *\langle floating point \rangle* $\{ \langle value \rangle \}$

Adds the *$\langle value \rangle$* to the *\langle floating point \rangle* , making the assignment globally.

\fp_sub:Nn
\fp_sub:cn

 \fp_sub:Nn *\langle floating point \rangle* $\{ \langle value \rangle \}$

Subtracts the *$\langle value \rangle$* from the *\langle floating point \rangle* , making the assignment within the current

TeX group level.

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

`\fp_gsub:Nn` *<floating point>* {*<value>*}

Subtracts the *<value>* from the *<floating point>*, making the assignment globally.

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

`\fp_mul:Nn` *<floating point>* {*<value>*}

Multiplies the *<floating point>* by the *<value>*, making the assignment within the current TeX group level.

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

`\fp_gmul:Nn` *<floating point>* {*<value>*}

Multiplies the *<floating point>* by the *<value>*, making the assignment globally.

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

`\fp_div:Nn` *<floating point>* {*<value>*}

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:cn</code>

`\fp_gdiv:Nn` *<floating point>* {*<value>*}

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.

155 Floating-point 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 `\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

$\langle value \rangle$ should be either a positive real number or a negative integer. If the $\langle floating point \rangle$ is positive, then the $\langle value \rangle$ may be any real value. Mathematically invalid operations such as 0^0 will give set the $\langle floating point \rangle$ to to `\c_undefined_fp`. The assignment is global.

156 Exponential and logarithm functions

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

`\fp_exp:Nn $\langle floating point \rangle$ { $\langle value \rangle$ }`

Calculates the exponential of the $\langle value \rangle$ and assigns this to the $\langle floating point \rangle$. The assignment is local.

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

`\fp_gexp:Nn $\langle floating point \rangle$ { $\langle value \rangle$ }`

Calculates the exponential of the $\langle value \rangle$ and assigns this to the $\langle floating point \rangle$. The assignment is global.

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

`\fp_ln:Nn $\langle floating point \rangle$ { $\langle value \rangle$ }`

Calculates the natural logarithm of the $\langle value \rangle$ and assigns this to the $\langle floating point \rangle$. The assignment is local.

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

`\fp_gln:Nn $\langle floating point \rangle$ { $\langle value \rangle$ }`

Calculates the natural logarithm of the $\langle value \rangle$ and assigns this to the $\langle floating point \rangle$. The assignment is global.

157 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 $\langle floating point \rangle$ { $\langle value \rangle$ }`

Assigns the sine of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is local.

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

`\fp_gsin:Nn $\langle floating point \rangle$ { $\langle value \rangle$ }`

Assigns the sine of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ 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 $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ 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 $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ 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 $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ 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 $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is global.

158 Constant floating point values

<code>\c_e_fp</code>

The value of the base of natural numbers, e .

<code>\c_one_fp</code>

A floating point variable with permanent value 1: used for speeding up some comparisons.

<code>\c_pi_fp</code>

The value of π .

<code>\c_undefined_fp</code>

A special marker floating point variable representing the result of an operation which does not give a defined result (such as division by 0).

<code>\c_zero_fp</code>

A permanently zero floating point variable.

159 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 XXII

The `l3luatex` package Lua \TeX -specific functions

160 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 X \TeX these will raise an error: use `\luatex_if_engine:T` to avoid this. Details of coding the Lua \TeX engine are detailed in the Lua \TeX manual.

<code>\lua_now:n *</code>
<code>\lua_now:x *</code>

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

The `<token list>` is first tokenized by \TeX , which will include converting line ends to spaces in the usual \TeX manner and which respects currently-applicable \TeX category

codes. The resulting $\langle \textit{Lua input} \rangle$ 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 $\langle \textit{Lua input} \rangle$ immediately, and in an expandable manner.

T_EXhackers note: `\lua_now:x` is the LuaT_EX primitive `\directlua` renamed.

<code>\lua_shipout:n</code>
<code>\lua_shipout:x</code>

`\lua_shipout:x { $\langle token list \rangle$ }`

The $\langle token list \rangle$ is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting $\langle \textit{Lua input} \rangle$ 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 $\langle \textit{Lua input} \rangle$ during the page-building routine: no T_EX expansion of the $\langle \textit{Lua input} \rangle$ will occur at this stage.

T_EXhackers note: At a T_EX level, the $\langle \textit{Lua input} \rangle$ is stored as a “whatsit”.

<code>\lua_shipout_x:n</code>
<code>\lua_shipout_x:x</code>

`\lua_shipout:n { $\langle token list \rangle$ }`

The $\langle token list \rangle$ is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting $\langle \textit{Lua input} \rangle$ 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 $\langle \textit{Lua input} \rangle$ during the page-building routine: the $\langle \textit{Lua input} \rangle$ 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).

T_EXhackers note: `\lua_shipout_x:n` is the LuaT_EX primitive `\latelua` named using the L^AT_EX3 scheme.

At a T_EX level, the $\langle \textit{Lua input} \rangle$ is stored as a “whatsit”.

161 Category code tables

As well as providing methods to break out into Lua, there are places where additional L^AT_EX3 functions are provided by the LuaT_EX engine. In particular, LuaT_EX 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 `\LuaTeX` engine.

`\cctab_new:N` `\cctab_new:N` *<category code table>*

Creates a new category code table, initially with the codes as used by `\InitEX`.

`\cctab_gset:Nn` `\cctab_gset:Nn` *<category code table>*
{<category code set up>}

Sets the *<category code table>* to apply the category codes which apply when the prevailing regime is modified by the *<category code set up>*. 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` *<category code table>*

Switches the category codes in force to those stored in the *<category code table>*. 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 *<category code table>* 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 `\LaTeX` 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 `\InitEX`.

`\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 XXIII

Implementation

162 Bootstrap code

```
1 <*initex | package>
```

162.1 Format-specific code

The very first thing to do is to bootstrap the \TeX system so that everything else will actually work. \TeX does not start with some pretty basic character codes set up.

```
2 <*initex>
3 \catcode '\{ = 1 \relax
4 \catcode '\} = 2 \relax
5 \catcode '\# = 6 \relax
6 \catcode '\^ = 7 \relax
7 </initex>
```

Tab characters should not show up in the code, but to be on the safe side.

```
8 <*initex>
9 \catcode '\^^I = 10 \relax
10 </initex>
```

For \LuaTeX 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.

```
11 <*initex>
12 \begingroup\expandafter\expandafter\expandafter\endgroup
13 \expandafter\ifx\csname directlua\endcsname\relax
14 \else
15   \directlua
16   {
17     tex.enableprimitives('',tex.extraprimitives ())
18     lua.bytecode[1] = function ()
19       function strcmp (A, B)
20         if A == B then
21           tex.write("0")
22         elseif A < B then
23           tex.write("-1")
24         else
25           tex.write("1")
26         end

```

```

27         end
28     end
29     lua.bytecode[1]()
30 }
31 \everyjob\expandafter
32 { \csname\detokenize{luatex_directlua:D}\endcsname{lua.bytecode[1]()}}
33 \long\edef\pdfstrcmp#1#2%
34 {%
35     \expandafter\noexpand\csname\detokenize{luatex_directlua:D}\endcsname
36     {%
37         strcmp%
38         (%
39             "\noexpand\luaescapestring{#1}",%
40             "\noexpand\luaescapestring{#2}"%
41         )%
42     }%
43 }
44 \fi
45 </initex>

```

162.2 Package-specific code

The package starts by identifying itself: the information itself is taken from the SVN Id string at the start of the source file.

```

46 <*package>
47 \ProvidesPackage{l3bootstrap}
48 [%
49     \ExplFileDate\space v\ExplFileVersion\space
50     L3 Experimental bootstrap code%
51 ]
52 </package>

```

For LuaTeX the functionality of the `\pdfstrcmp` primitive needs to be provided: the `pdftexcmds` package is used to do this if necessary. At present, there is also a need to deal with some low-level allocation stuff that could usefully be added to `lualatex.ini`. As it is currently not, load Heiko Oberdiek's `luatex` package instead.

```

53 <*package>
54 \def\@tempa%
55 {%
56     \def\@tempa{}%
57     \RequirePackage{luatex}%
58     \RequirePackage{pdftexcmds}%
59     \let\pdfstrcmp\pdf@strcmp
60 }
61 \begingroup\expandafter\expandafter\expandafter\endgroup
62 \expandafter\ifx\csname directlua\endcsname\relax
63 \else

```

```

64 \expandafter\@tempa
65 \fi
66 \</package>

```

\ExplSyntaxOff Experimental syntax switching is set up here for the package-loading process. These are
\ExplSyntaxOn redefined in expl3 for the package and in l3final for the format.

```

67 <*package>
68 \protected\edef\ExplSyntaxOff
69 {%
70   \catcode 9 = \the\catcode 9\relax
71   \catcode 32 = \the\catcode 32\relax
72   \catcode 34 = \the\catcode 34\relax
73   \catcode 38 = \the\catcode 38\relax
74   \catcode 58 = \the\catcode 58\relax
75   \catcode 94 = \the\catcode 94\relax
76   \catcode 95 = \the\catcode 95\relax
77   \catcode 124 = \the\catcode 124\relax
78   \catcode 126 = \the\catcode 126\relax
79   \endlinechar = \the\endlinechar\relax
80   \chardef\csname\detokenize{l_expl_status_bool}\endcsname = 0 \relax
81 }
82 \protected\edef\ExplSyntaxOn
83 {
84   \catcode 9 = 9 \relax
85   \catcode 32 = 9 \relax
86   \catcode 34 = 12 \relax
87   \catcode 58 = 11 \relax
88   \catcode 94 = 7 \relax
89   \catcode 95 = 11 \relax
90   \catcode 124 = 12 \relax
91   \catcode 126 = 10 \relax
92   \endlinechar = 32 \relax
93   \chardef\csname\detokenize{l_expl_status_bool}\endcsname = 1 \relax
94 }
95 \</package>

```

(End definition for \ExplSyntaxOff and \ExplSyntaxOn. These functions are documented on page 5.)

\l_expl_status_bool The status for experimental code syntax: this is off at present. This code is used by both the package and the format.

```

96 \expandafter\chardef\csname\detokenize{l_expl_status_bool}\endcsname = 0 \relax

```

(End definition for \l_expl_status_bool. This function is documented on page ??.)

162.3 Dealing with package-mode meta-data

\GetIdInfo Functions for collecting up meta-data from the SVN information used by the L^AT_EX3 Project.

```

\GetIdInfoFull
\GetIdInfoAuxI
\GetIdInfoAuxII
\GetIdInfoAuxIII
\GetIdInfoAuxCVS
\GetIdInfoAuxSVN

```

```

97  \begin{package}
98  \protected\def\GetIdInfo
99  {
100   \begin{group}
101   \catcode 32 = 10 \relax
102   \GetIdInfoAuxI
103  }
104  \protected\def\GetIdInfoAuxI$#1$#2%
105  {
106   \def\tempa{#1}%
107   \def\tempb{Id}%
108   \ifx\tempa\tempb
109   \def\tempa
110   {%
111   \endgroup
112   \def\ExplFileDate{0000/00/00}%
113   \def\ExplFileDescription{#2}%
114   \def\ExplFileName{[unknown~name]}%
115   \def\ExplFileVersion{000}%
116   }%
117   \else
118   \def\tempa
119   {%
120   \endgroup
121   \GetIdInfoAuxII$#1$#2}%
122   }%
123   \fi
124   \tempa
125  }
126  \protected\def\GetIdInfoAuxII$#1 #2.#3 #4 #5 #6 #7 #8$#9%
127  {%
128   \def\ExplFileName{#2}%
129   \def\ExplFileVersion{#4}%
130   \def\ExplFileDescription{#9}%
131   \GetIdInfoAuxIII#5\relax#3\relax#5\relax#6\relax
132  }
133  \protected\def\GetIdInfoAuxIII#1#2#3#4#5#6\relax
134  {%
135   \ifx#5/%
136   \expandafter\GetIdInfoAuxCVS
137   \else
138   \expandafter\GetIdInfoAuxSVN
139   \fi
140  }
141  \protected\def\GetIdInfoAuxCVS#1,v\relax#2\relax#3\relax
142  {\def\ExplFileDate{#2}}
143  \protected\def\GetIdInfoAuxSVN#1\relax#2-#3-#4\relax#5Z\relax
144  {\def\ExplFileDate{#2/#3/#4}}
145  \end{package}

```

(End definition for `\GetIdInfo`. This function is documented on page 6.)

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

For other packages and classes building on this one it is convenient not to need `\ExplSyntaxOn` each time.

```

146 <*package>
147 \protected\def\ProvidesExplPackage#1#2#3#4%
148 {%
149   \ProvidesPackage{#1}[#2 v#3 #4]%
150   \ExplSyntaxOn
151 }
152 \protected\def\ProvidesExplClass#1#2#3#4%
153 {%
154   \ProvidesClass{#1}[#2 v#3 #4]%
155   \ExplSyntaxOn
156 }
157 \protected\def\ProvidesExplFile#1#2#3#4%
158 {%
159   \ProvidesFile{#1}[#2 v#3 #4]%
160   \ExplSyntaxOn
161 }
162 </package>

```

(End definition for `\ProvidesExplPackage`, `\ProvidesExplClass`, and `\ProvidesExplFile`. These functions are documented on page 6.)

`\@pushfilename`
`\@popfilename`

The idea here is to use L^AT_EX 2_ε's `\@pushfilename` and `\@popfilename` to track the current syntax status. This can be achieved by saving the current status flag at each push to a stack, then recovering it at the pop stage and checking if the code environment should still be active.

```

163 <*package>
164 \edef\@pushfilename
165 {%
166   \edef\expandafter\noexpand
167   \csname\detokenize{l_expl_status_stack_tl}\endcsname
168   {%
169     \noexpand\ifodd\expandafter\noexpand
170     \csname\detokenize{l_expl_status_bool}\endcsname
171     1%
172   \noexpand\else
173     0%
174   \noexpand\fi
175   \expandafter\noexpand
176   \csname\detokenize{l_expl_status_stack_tl}\endcsname
177   }%
178   \ExplSyntaxOff
179   \unexpanded\expandafter{\@pushfilename}%
180 }
181 \edef\@popfilename

```

```

182 {%
183   \unexpanded\expandafter{\@popfilename}%
184   \noexpand\if a\expandafter\noexpand\csname
185     \detokenize{l_expl_status_stack_tl}\endcsname a%
186     \ExplSyntaxOff
187   \noexpand\else
188     \noexpand\expandafter
189     \expandafter\noexpand\csname
190       \detokenize{expl_status_pop:w}\endcsname
191       \expandafter\noexpand\csname
192         \detokenize{l_expl_status_stack_tl}\endcsname
193       \noexpand\@nil
194     \noexpand\fi
195   }
196 </package>

```

(End definition for \@pushfilename and \@popfilename. These functions are documented on page ??.)

`\l_expl_status_stack_tl` As `expl3` itself cannot be loaded with the code environment already active, at the end of the package `\ExplSyntaxOff` can safely be called.

```

197 <*package>
198 \@namedef{\detokenize{l_expl_status_stack_tl}}{0}
199 </package>

```

(End definition for `\l_expl_status_stack_tl`. This function is documented on page ??.)

`\expl_status_pop:w` The `pop` auxiliary function removes the first item from the stack, saves the rest of the stack and then does the test. As `\ExplSyntaxOff` is already defined as a protected macro, there is no need for `\noexpand` here.

```

200 <*package>
201 \expandafter\edef\csname\detokenize{expl_status_pop:w}\endcsname#1#2\@nil
202 {%
203   \def\expandafter\noexpand
204     \csname\detokenize{l_expl_status_stack_tl}\endcsname{#2}%
205     \noexpand\ifodd#1\space
206     \noexpand\expandafter\noexpand\ExplSyntaxOn
207   \noexpand\else
208     \noexpand\expandafter\ExplSyntaxOff
209   \noexpand\fi
210 }
211 </package>

```

(End definition for `\expl_status_pop:w`.)

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.

```

212 <*package>
213 \expandafter\protected\expandafter\def

```

```

214 \csname\detokenize{package_check_loaded_expl:}\endcsname
215 {%
216   \@ifpackageloaded{expl3}
217   {}
218   {%
219     \PackageError{expl3}
220     {Cannot load the expl3 modules separately}
221     {%
222       The expl3 modules cannot be loaded separately;\MessageBreak
223       please \string\usepackage\string{expl3\string} instead.
224     }%
225   }%
226 }
227 \</package>

```

162.4 The `\pdfstrcmp` primitive in XeTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do.

```

228 \begingroup\expandafter\expandafter\expandafter\endgroup
229 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
230 \let\pdfstrcmp\strcmp
231 \fi

```

162.5 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to $\varepsilon\text{-TeX}$. The former is therefore used as a test for a suitable engine.

```

232 \begingroup\expandafter\expandafter\expandafter\endgroup
233 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
234 \<*\package>
235 \PackageError{!3names}{Required primitive not found: \protect\pdfstrcmp}
236 {%
237   LaTeX3 requires the e-TeX primitives and
238   \string\pdfstrcmp.\MessageBreak
239   These are available in engine versions: \MessageBreak
240   - pdfTeX 1.30 \MessageBreak
241   - XeTeX 0.9994 \MessageBreak
242   - LuaTeX 0.60 \MessageBreak
243   or later. \MessageBreak
244   \MessageBreak
245   Loading of expl3 will abort!
246 }
247 \</package>
248 \<*\initex>
249 \newlinechar'\^^J\relax

```



```

250 \errhelp{%
251     LaTeX3 requires the e-TeX primitives and
252     \string\pdfstrcmp. ^^J
253     These are available in engine versions: ^^J
254     - pdfTeX 1.30 ^^J
255     - XeTeX 0.9994 ^^J
256     - LuaTeX 0.60 ^^J
257     or later. ^^J
258     For pdfTeX and XeTeX the '-etex' command-line switch is also
259     needed. ^^J
260     ^^J
261     Format building will abort!
262 }
263 </initex>
264 \expandafter\endinput
265 \fi

```

162.6 The L^AT_EX3 code environment

`\ExplSyntaxNamesOn` These can be set up early, as they are not used anywhere in the package or format itself.
`\ExplSyntaxNamesOff` Using an `\edef` here makes the definitions that bit clearer later.

```

266 \protected\edef\ExplSyntaxNamesOn
267 {%
268     \expandafter\noexpand
269     \csname\detokenize{char_set_catcode_letter:n}\endcsname{58}%
270     \expandafter\noexpand
271     \csname\detokenize{char_set_catcode_letter:n}\endcsname{95}%
272 }
273 \protected\edef\ExplSyntaxNamesOff
274 {%
275     \expandafter\noexpand
276     \csname\detokenize{char_set_catcode_math_subscript:n}\endcsname{58}%
277     \expandafter\noexpand
278     \csname\detokenize{char_set_catcode_other:n}\endcsname{95}%
279 }

```

(End definition for `\ExplSyntaxNamesOn` and `\ExplSyntaxNamesOff`. These functions are documented on page 5.)

The code environment is now set up for the format: the package deals with this using `\ProvidesExplPackage`.

```

280 <*initex>
281 \catcode 9 = 9 \relax
282 \catcode 32 = 9 \relax
283 \catcode 34 = 12 \relax
284 \catcode 58 = 11 \relax
285 \catcode 94 = 7 \relax
286 \catcode 95 = 11 \relax

```

```

287 \catcode 124 = 12 \relax
288 \catcode 126 = 10 \relax
289 \endlinechar = 32 \relax
290 </initex>

```

\ExplSyntaxOn The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time.

```

291 <*initex>
292 \protected \def \ExplSyntaxOn
293 {
294   \bool_if:NF \l_expl_status_bool
295   {
296     \cs_set_protected_nopar:Npx \ExplSyntaxOff
297     {
298       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
299       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
300       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
301       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
302       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
303       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
304       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
305       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
306       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
307       \tex_endlinechar:D =
308       \tex_the:D \tex_endlinechar:D \scan_stop:
309       \bool_set_false:N \l_expl_status_bool
310       \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
311     }
312   }
313   \char_set_catcode_ignore:n { 9 } % tab
314   \char_set_catcode_ignore:n { 32 } % space
315   \char_set_catcode_other:n { 34 } % double quote
316   \char_set_catcode_alignment:n { 38 } % ampersand
317   \char_set_catcode_letter:n { 58 } % colon
318   \char_set_catcode_math_superscript:n { 94 } % circumflex
319   \char_set_catcode_letter:n { 95 } % underscore
320   \char_set_catcode_other:n { 124 } % pipe
321   \char_set_catcode_space:n { 126 } % tilde
322   \tex_endlinechar:D = 32 \scan_stop:
323   \bool_set_true:N \l_expl_status_bool
324 }
325 \protected \def \ExplSyntaxOff { }
326 </initex>

```

(End definition for `\ExplSyntaxOn` and `\ExplSyntaxOff`. These functions are documented on page 5.)

`\l_expl_status_bool` A flag to show the current syntax status.

```

327 <*initex>

```

```

328 \chardef \l_expl_status_bool = 0 ~
329 </initex>

330 </initex | package>

```

163 l3names implementation

```

331 <*initex | package>
332 <*package>
333 \ProvidesExplPackage
334   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
335 </package>

```

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

`\tex_undefined:D` This function does not exist at all, but is the name used by the plain T_EX format for an undefined function. So it should be marked here as “taken”.

(End definition for \tex_undefined:D. This function is documented on page ??.)

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```

336 \let \tex_global:D \global
337 \let \tex_let:D \let

```

Everything is inside a (rather long) group, which keeps `\name_primitive:NN` trapped.

```

338 \begingroup

```

`\name_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```

339 \long \def \name_primitive:NN #1#2
340   {
341     \tex_global:D \tex_let:D #2 #1
342 <*initex>
343     \tex_global:D \tex_let:D #1 \tex_undefined:D
344 </initex>
345   }

```

(End definition for \name_primitive:NN.)

In the current incarnation of this package, all T_EX 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.

```

346 \name_primitive:NN \tex_space:D
347 \name_primitive:NN \tex_italiccor:D
348 \name_primitive:NN \tex_hyphen:D

```

Now all the other primitives.

349	\name_primitive:NN \let	\tex_let:D
350	\name_primitive:NN \def	\tex_def:D
351	\name_primitive:NN \edef	\tex_edef:D
352	\name_primitive:NN \gdef	\tex_gdef:D
353	\name_primitive:NN \xdef	\tex_xdef:D
354	\name_primitive:NN \chardef	\tex_chardef:D
355	\name_primitive:NN \countdef	\tex_countdef:D
356	\name_primitive:NN \dimendef	\tex_dimendef:D
357	\name_primitive:NN \skipdef	\tex_skipdef:D
358	\name_primitive:NN \muskipdef	\tex_muskipdef:D
359	\name_primitive:NN \mathchardef	\tex_mathchardef:D
360	\name_primitive:NN \toksdef	\tex_toksdef:D
361	\name_primitive:NN \futurelet	\tex_futurelet:D
362	\name_primitive:NN \advance	\tex_advance:D
363	\name_primitive:NN \divide	\tex_divide:D
364	\name_primitive:NN \multiply	\tex_multiply:D
365	\name_primitive:NN \font	\tex_font:D
366	\name_primitive:NN \fam	\tex_fam:D
367	\name_primitive:NN \global	\tex_global:D
368	\name_primitive:NN \long	\tex_long:D
369	\name_primitive:NN \outer	\tex_outer:D
370	\name_primitive:NN \setlanguage	\tex_setlanguage:D
371	\name_primitive:NN \globaldefs	\tex_globaldefs:D
372	\name_primitive:NN \afterassignment	\tex_afterassignment:D
373	\name_primitive:NN \aftergroup	\tex_aftergroup:D
374	\name_primitive:NN \expandafter	\tex_expandafter:D
375	\name_primitive:NN \noexpand	\tex_noexpand:D
376	\name_primitive:NN \begingroup	\tex_begingroup:D
377	\name_primitive:NN \endgroup	\tex_endgroup:D
378	\name_primitive:NN \halign	\tex_halign:D
379	\name_primitive:NN \valign	\tex_valign:D
380	\name_primitive:NN \cr	\tex_cr:D
381	\name_primitive:NN \crrcr	\tex_crrcr:D
382	\name_primitive:NN \noalign	\tex_noalign:D
383	\name_primitive:NN \omit	\tex_omit:D
384	\name_primitive:NN \span	\tex_span:D
385	\name_primitive:NN \tabskip	\tex_tabskip:D
386	\name_primitive:NN \everycr	\tex_everycr:D
387	\name_primitive:NN \if	\tex_if:D
388	\name_primitive:NN \ifcase	\tex_ifcase:D
389	\name_primitive:NN \ifcat	\tex_ifcat:D
390	\name_primitive:NN \ifnum	\tex_ifnum:D
391	\name_primitive:NN \ifodd	\tex_ifodd:D
392	\name_primitive:NN \ifdim	\tex_ifdim:D
393	\name_primitive:NN \ifeof	\tex_ifeof:D
394	\name_primitive:NN \ifhbox	\tex_ifhbox:D
395	\name_primitive:NN \ifvbox	\tex_ifvbox:D
396	\name_primitive:NN \ifvoid	\tex_ifvoid:D

397	\name_primitive:NN \ifx	\tex_ifx:D
398	\name_primitive:NN \iffalse	\tex_iffalse:D
399	\name_primitive:NN \iftrue	\tex_iftrue:D
400	\name_primitive:NN \ifhmode	\tex_ifhmode:D
401	\name_primitive:NN \ifmmode	\tex_ifmmode:D
402	\name_primitive:NN \ifvmode	\tex_ifvmode:D
403	\name_primitive:NN \ifinner	\tex_ifinner:D
404	\name_primitive:NN \else	\tex_else:D
405	\name_primitive:NN \fi	\tex_fi:D
406	\name_primitive:NN \or	\tex_or:D
407	\name_primitive:NN \immediate	\tex_immediate:D
408	\name_primitive:NN \closeout	\tex_closeout:D
409	\name_primitive:NN \openin	\tex_openin:D
410	\name_primitive:NN \openout	\tex_openout:D
411	\name_primitive:NN \read	\tex_read:D
412	\name_primitive:NN \write	\tex_write:D
413	\name_primitive:NN \closein	\tex_closein:D
414	\name_primitive:NN \newlinechar	\tex_newlinechar:D
415	\name_primitive:NN \input	\tex_input:D
416	\name_primitive:NN \endinput	\tex_endinput:D
417	\name_primitive:NN \inputlineno	\tex_inputlineno:D
418	\name_primitive:NN \errmessage	\tex_errmessage:D
419	\name_primitive:NN \message	\tex_message:D
420	\name_primitive:NN \show	\tex_show:D
421	\name_primitive:NN \showthe	\tex_showthe:D
422	\name_primitive:NN \showbox	\tex_showbox:D
423	\name_primitive:NN \showlists	\tex_showlists:D
424	\name_primitive:NN \errhelp	\tex_errhelp:D
425	\name_primitive:NN \errorcontextlines	\tex_errorcontextlines:D
426	\name_primitive:NN \tracingcommands	\tex_tracingcommands:D
427	\name_primitive:NN \tracinglostchars	\tex_tracinglostchars:D
428	\name_primitive:NN \tracingmacros	\tex_tracingmacros:D
429	\name_primitive:NN \tracingonline	\tex_tracingonline:D
430	\name_primitive:NN \tracingoutput	\tex_tracingoutput:D
431	\name_primitive:NN \tracingpages	\tex_tracingpages:D
432	\name_primitive:NN \tracingparagraphs	\tex_tracingparagraphs:D
433	\name_primitive:NN \tracingrestores	\tex_tracingrestores:D
434	\name_primitive:NN \tracingstats	\tex_tracingstats:D
435	\name_primitive:NN \pausing	\tex_pausing:D
436	\name_primitive:NN \showboxbreadth	\tex_showboxbreadth:D
437	\name_primitive:NN \showboxdepth	\tex_showboxdepth:D
438	\name_primitive:NN \batchmode	\tex_batchmode:D
439	\name_primitive:NN \errorstopmode	\tex_errorstopmode:D
440	\name_primitive:NN \nonstopmode	\tex_nonstopmode:D
441	\name_primitive:NN \scrollmode	\tex_scrollmode:D
442	\name_primitive:NN \end	\tex_end:D
443	\name_primitive:NN \csname	\tex_csname:D
444	\name_primitive:NN \endcsname	\tex_endcsname:D
445	\name_primitive:NN \ignorespaces	\tex_ignorespaces:D
446	\name_primitive:NN \relax	\tex_relax:D

447	\name_primitive:NN	\the	\tex_the:D
448	\name_primitive:NN	\mag	\tex_mag:D
449	\name_primitive:NN	\language	\tex_language:D
450	\name_primitive:NN	\mark	\tex_mark:D
451	\name_primitive:NN	\topmark	\tex_topmark:D
452	\name_primitive:NN	\firstmark	\tex_firstmark:D
453	\name_primitive:NN	\botmark	\tex_botmark:D
454	\name_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
455	\name_primitive:NN	\splitbotmark	\tex_splitbotmark:D
456	\name_primitive:NN	\fontname	\tex_fontname:D
457	\name_primitive:NN	\escapechar	\tex_escapechar:D
458	\name_primitive:NN	\endlinechar	\tex_endlinechar:D
459	\name_primitive:NN	\mathchoice	\tex_mathchoice:D
460	\name_primitive:NN	\delimiter	\tex_delimiter:D
461	\name_primitive:NN	\mathaccent	\tex_mathaccent:D
462	\name_primitive:NN	\mathchar	\tex_mathchar:D
463	\name_primitive:NN	\mskip	\tex_mskip:D
464	\name_primitive:NN	\radical	\tex_radical:D
465	\name_primitive:NN	\vcenter	\tex_vcenter:D
466	\name_primitive:NN	\mkern	\tex_mkern:D
467	\name_primitive:NN	\above	\tex_above:D
468	\name_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
469	\name_primitive:NN	\atop	\tex_atop:D
470	\name_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
471	\name_primitive:NN	\over	\tex_over:D
472	\name_primitive:NN	\overwithdelims	\tex_overwithdelims:D
473	\name_primitive:NN	\displaystyle	\tex_displaystyle:D
474	\name_primitive:NN	\textstyle	\tex_textstyle:D
475	\name_primitive:NN	\scriptstyle	\tex_scriptstyle:D
476	\name_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
477	\name_primitive:NN	\nonscript	\tex_nonscript:D
478	\name_primitive:NN	\eqno	\tex_eqno:D
479	\name_primitive:NN	\leqno	\tex_leqno:D
480	\name_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D
481	\name_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
482	\name_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
483	\name_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
484	\name_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
485	\name_primitive:NN	\displayindent	\tex_displayindent:D
486	\name_primitive:NN	\displaywidth	\tex_displaywidth:D
487	\name_primitive:NN	\everydisplay	\tex_everydisplay:D
488	\name_primitive:NN	\predisplaysize	\tex_predisplaysize:D
489	\name_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
490	\name_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
491	\name_primitive:NN	\mathbin	\tex_mathbin:D
492	\name_primitive:NN	\mathclose	\tex_mathclose:D
493	\name_primitive:NN	\mathinner	\tex_mathinner:D
494	\name_primitive:NN	\mathop	\tex_mathop:D
495	\name_primitive:NN	\displaylimits	\tex_displaylimits:D
496	\name_primitive:NN	\limits	\tex_limits:D

497	\name_primitive:NN \nolimits	\tex_nolimits:D
498	\name_primitive:NN \mathopen	\tex_mathopen:D
499	\name_primitive:NN \mathord	\tex_mathord:D
500	\name_primitive:NN \mathpunct	\tex_mathpunct:D
501	\name_primitive:NN \mathrel	\tex_mathrel:D
502	\name_primitive:NN \overline	\tex_overline:D
503	\name_primitive:NN \underline	\tex_underline:D
504	\name_primitive:NN \left	\tex_left:D
505	\name_primitive:NN \right	\tex_right:D
506	\name_primitive:NN \binoppenalty	\tex_binoppenalty:D
507	\name_primitive:NN \relpenalty	\tex_relpenalty:D
508	\name_primitive:NN \delimitershortfall	\tex_delimitershortfall:D
509	\name_primitive:NN \delimiterfactor	\tex_delimiterfactor:D
510	\name_primitive:NN \nulldelimiterspace	\tex_nulldelimiterspace:D
511	\name_primitive:NN \everymath	\tex_everymath:D
512	\name_primitive:NN \mathsurround	\tex_mathsurround:D
513	\name_primitive:NN \medmuskip	\tex_medmuskip:D
514	\name_primitive:NN \thinmuskip	\tex_thinmuskip:D
515	\name_primitive:NN \thickmuskip	\tex_thickmuskip:D
516	\name_primitive:NN \scriptspace	\tex_scriptspace:D
517	\name_primitive:NN \noboundary	\tex_noboundary:D
518	\name_primitive:NN \accent	\tex_accent:D
519	\name_primitive:NN \char	\tex_char:D
520	\name_primitive:NN \discretionary	\tex_discretionary:D
521	\name_primitive:NN \hfil	\tex_hfil:D
522	\name_primitive:NN \hfilneg	\tex_hfilneg:D
523	\name_primitive:NN \hfill	\tex_hfill:D
524	\name_primitive:NN \hskip	\tex_hskip:D
525	\name_primitive:NN \hss	\tex_hss:D
526	\name_primitive:NN \vfil	\tex_vfil:D
527	\name_primitive:NN \vfilneg	\tex_vfilneg:D
528	\name_primitive:NN \vfill	\tex_vfill:D
529	\name_primitive:NN \vskip	\tex_vskip:D
530	\name_primitive:NN \vss	\tex_vss:D
531	\name_primitive:NN \unskip	\tex_unskip:D
532	\name_primitive:NN \kern	\tex_kern:D
533	\name_primitive:NN \unkern	\tex_unkern:D
534	\name_primitive:NN \hrule	\tex_hrule:D
535	\name_primitive:NN \vrule	\tex_vrule:D
536	\name_primitive:NN \leaders	\tex_leaders:D
537	\name_primitive:NN \cleaders	\tex_cleaders:D
538	\name_primitive:NN \xleaders	\tex_xleaders:D
539	\name_primitive:NN \lastkern	\tex_lastkern:D
540	\name_primitive:NN \lastskip	\tex_lastskip:D
541	\name_primitive:NN \indent	\tex_indent:D
542	\name_primitive:NN \par	\tex_par:D
543	\name_primitive:NN \noindent	\tex_noindent:D
544	\name_primitive:NN \vadjust	\tex_vadjust:D
545	\name_primitive:NN \baselineskip	\tex_baselineskip:D
546	\name_primitive:NN \lineskip	\tex_lineskip:D

547	\name_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
548	\name_primitive:NN	\clubpenalty	\tex_clubpenalty:D
549	\name_primitive:NN	\widowpenalty	\tex_widowpenalty:D
550	\name_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
551	\name_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
552	\name_primitive:NN	\linepenalty	\tex_linepenalty:D
553	\name_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
554	\name_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
555	\name_primitive:NN	\adjdemerits	\tex_adjdemerits:D
556	\name_primitive:NN	\hangafter	\tex_hangafter:D
557	\name_primitive:NN	\hangindent	\tex_hangindent:D
558	\name_primitive:NN	\parshape	\tex_parshape:D
559	\name_primitive:NN	\hsize	\tex_hsize:D
560	\name_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
561	\name_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
562	\name_primitive:NN	\leftskip	\tex_leftskip:D
563	\name_primitive:NN	\rightskip	\tex_rightskip:D
564	\name_primitive:NN	\looseness	\tex_looseness:D
565	\name_primitive:NN	\parskip	\tex_parskip:D
566	\name_primitive:NN	\parindent	\tex_parindent:D
567	\name_primitive:NN	\uchyph	\tex_uchyph:D
568	\name_primitive:NN	\emergencystretch	\tex_emergencystretch:D
569	\name_primitive:NN	\pretolerance	\tex_pretolerance:D
570	\name_primitive:NN	\tolerance	\tex_tolerance:D
571	\name_primitive:NN	\spaceskip	\tex_spaceskip:D
572	\name_primitive:NN	\xspaceskip	\tex_xspaceskip:D
573	\name_primitive:NN	\parfillskip	\tex_parfillskip:D
574	\name_primitive:NN	\everypar	\tex_everypar:D
575	\name_primitive:NN	\prevgraf	\tex_prevgraf:D
576	\name_primitive:NN	\spacefactor	\tex_spacefactor:D
577	\name_primitive:NN	\shipout	\tex_shipout:D
578	\name_primitive:NN	\vsize	\tex_vsize:D
579	\name_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
580	\name_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
581	\name_primitive:NN	\topskip	\tex_topskip:D
582	\name_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
583	\name_primitive:NN	\maxdepth	\tex_maxdepth:D
584	\name_primitive:NN	\output	\tex_output:D
585	\name_primitive:NN	\deadcycles	\tex_deadcycles:D
586	\name_primitive:NN	\pagedepth	\tex_pagedepth:D
587	\name_primitive:NN	\pagestretch	\tex_pagestretch:D
588	\name_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
589	\name_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
590	\name_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
591	\name_primitive:NN	\pageshrink	\tex_pageshrink:D
592	\name_primitive:NN	\pagegoal	\tex_pagegoal:D
593	\name_primitive:NN	\pagetotal	\tex_pagetotal:D
594	\name_primitive:NN	\outputpenalty	\tex_outputpenalty:D
595	\name_primitive:NN	\hoffset	\tex_hoffset:D
596	\name_primitive:NN	\voffset	\tex_voffset:D

597	\name_primitive:NN \insert	\tex_insert:D
598	\name_primitive:NN \holdinginserts	\tex_holdinginserts:D
599	\name_primitive:NN \floatingpenalty	\tex_floatingpenalty:D
600	\name_primitive:NN \insertpenalties	\tex_insertpenalties:D
601	\name_primitive:NN \lower	\tex_lower:D
602	\name_primitive:NN \moveleft	\tex_moveleft:D
603	\name_primitive:NN \moveright	\tex_moveright:D
604	\name_primitive:NN \raise	\tex_raise:D
605	\name_primitive:NN \copy	\tex_copy:D
606	\name_primitive:NN \lastbox	\tex_lastbox:D
607	\name_primitive:NN \vsplit	\tex_vsplit:D
608	\name_primitive:NN \unhbox	\tex_unhbox:D
609	\name_primitive:NN \unhcopy	\tex_unhcopy:D
610	\name_primitive:NN \unvbox	\tex_unvbox:D
611	\name_primitive:NN \unvcopy	\tex_unvcopy:D
612	\name_primitive:NN \setbox	\tex_setbox:D
613	\name_primitive:NN \hbox	\tex_hbox:D
614	\name_primitive:NN \vbox	\tex_vbox:D
615	\name_primitive:NN \vtop	\tex_vtop:D
616	\name_primitive:NN \prevdepth	\tex_prevdepth:D
617	\name_primitive:NN \badness	\tex_badness:D
618	\name_primitive:NN \hbadness	\tex_hbadness:D
619	\name_primitive:NN \vbadness	\tex_vbadness:D
620	\name_primitive:NN \hfuzz	\tex_hfuzz:D
621	\name_primitive:NN \vfuzz	\tex_vfuzz:D
622	\name_primitive:NN \overfullrule	\tex_overfullrule:D
623	\name_primitive:NN \boxmaxdepth	\tex_boxmaxdepth:D
624	\name_primitive:NN \splitmaxdepth	\tex_splitmaxdepth:D
625	\name_primitive:NN \splittopskip	\tex_splittopskip:D
626	\name_primitive:NN \everyhbox	\tex_everyhbox:D
627	\name_primitive:NN \everyvbox	\tex_everyvbox:D
628	\name_primitive:NN \nullfont	\tex_nullfont:D
629	\name_primitive:NN \textfont	\tex_textfont:D
630	\name_primitive:NN \scriptfont	\tex_scriptfont:D
631	\name_primitive:NN \scriptscriptfont	\tex_scriptscriptfont:D
632	\name_primitive:NN \fontdimen	\tex_fontdimen:D
633	\name_primitive:NN \hyphenchar	\tex_hyphenchar:D
634	\name_primitive:NN \skewchar	\tex_skewchar:D
635	\name_primitive:NN \defaultthyphenchar	\tex_defaultthyphenchar:D
636	\name_primitive:NN \defaultskewchar	\tex_defaultskewchar:D
637	\name_primitive:NN \number	\tex_number:D
638	\name_primitive:NN \romannumeral	\tex_romannumeral:D
639	\name_primitive:NN \string	\tex_string:D
640	\name_primitive:NN \lowercase	\tex_lowercase:D
641	\name_primitive:NN \uppercase	\tex_uppercase:D
642	\name_primitive:NN \meaning	\tex_meaning:D
643	\name_primitive:NN \penalty	\tex_penalty:D
644	\name_primitive:NN \unpenalty	\tex_unpenalty:D
645	\name_primitive:NN \lastpenalty	\tex_lastpenalty:D
646	\name_primitive:NN \special	\tex_special:D

647	<code>\name_primitive:NN \dump</code>	<code>\tex_dump:D</code>
648	<code>\name_primitive:NN \patterns</code>	<code>\tex_patterns:D</code>
649	<code>\name_primitive:NN \hyphenation</code>	<code>\tex_hyphenation:D</code>
650	<code>\name_primitive:NN \time</code>	<code>\tex_time:D</code>
651	<code>\name_primitive:NN \day</code>	<code>\tex_day:D</code>
652	<code>\name_primitive:NN \month</code>	<code>\tex_month:D</code>
653	<code>\name_primitive:NN \year</code>	<code>\tex_year:D</code>
654	<code>\name_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
655	<code>\name_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
656	<code>\name_primitive:NN \count</code>	<code>\tex_count:D</code>
657	<code>\name_primitive:NN \dimen</code>	<code>\tex_dimen:D</code>
658	<code>\name_primitive:NN \skip</code>	<code>\tex_skip:D</code>
659	<code>\name_primitive:NN \toks</code>	<code>\tex_toks:D</code>
660	<code>\name_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
661	<code>\name_primitive:NN \box</code>	<code>\tex_box:D</code>
662	<code>\name_primitive:NN \wd</code>	<code>\tex_wd:D</code>
663	<code>\name_primitive:NN \ht</code>	<code>\tex_ht:D</code>
664	<code>\name_primitive:NN \dp</code>	<code>\tex_dp:D</code>
665	<code>\name_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
666	<code>\name_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
667	<code>\name_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
668	<code>\name_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
669	<code>\name_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
670	<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_`.

671	<code>\name_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
672	<code>\name_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
673	<code>\name_primitive:NN \unless</code>	<code>\etex_unless:D</code>
674	<code>\name_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
675	<code>\name_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
676	<code>\name_primitive:NN \marks</code>	<code>\etex_marks:D</code>
677	<code>\name_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
678	<code>\name_primitive:NN \firstmarks</code>	<code>\etex_firstmarks:D</code>
679	<code>\name_primitive:NN \botmarks</code>	<code>\etex_botmarks:D</code>
680	<code>\name_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
681	<code>\name_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
682	<code>\name_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
683	<code>\name_primitive:NN \detokenize</code>	<code>\etex_detokenize:D</code>
684	<code>\name_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
685	<code>\name_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
686	<code>\name_primitive:NN \readline</code>	<code>\etex_readline:D</code>
687	<code>\name_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
688	<code>\name_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
689	<code>\name_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
690	<code>\name_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
691	<code>\name_primitive:NN \currentiflevel</code>	<code>\etex_currentiflevel:D</code>
692	<code>\name_primitive:NN \currentifbranch</code>	<code>\etex_currentifbranch:D</code>

693	<code>\name_primitive:NN \currentifttype</code>	<code>\etex_currentifttype:D</code>
694	<code>\name_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
695	<code>\name_primitive:NN \currentgrouplevel</code>	<code>\etex_currentgrouplevel:D</code>
696	<code>\name_primitive:NN \currentgrouptype</code>	<code>\etex_currentgrouptype:D</code>
697	<code>\name_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
698	<code>\name_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
699	<code>\name_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
700	<code>\name_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
701	<code>\name_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
702	<code>\name_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>
703	<code>\name_primitive:NN \fontchardp</code>	<code>\etex_fontchardp:D</code>
704	<code>\name_primitive:NN \fontcharwd</code>	<code>\etex_fontcharwd:D</code>
705	<code>\name_primitive:NN \fontcharic</code>	<code>\etex_fontcharic:D</code>
706	<code>\name_primitive:NN \parshapeindent</code>	<code>\etex_parshapeindent:D</code>
707	<code>\name_primitive:NN \parshapelength</code>	<code>\etex_parshapelength:D</code>
708	<code>\name_primitive:NN \parshapedimen</code>	<code>\etex_parshapedimen:D</code>
709	<code>\name_primitive:NN \numexpr</code>	<code>\etex_numexpr:D</code>
710	<code>\name_primitive:NN \dimexpr</code>	<code>\etex_dimexpr:D</code>
711	<code>\name_primitive:NN \glueexpr</code>	<code>\etex_glueexpr:D</code>
712	<code>\name_primitive:NN \muexpr</code>	<code>\etex_muexpr:D</code>
713	<code>\name_primitive:NN \gluestretch</code>	<code>\etex_gluestretch:D</code>
714	<code>\name_primitive:NN \glueshrink</code>	<code>\etex_glueshrink:D</code>
715	<code>\name_primitive:NN \gluestretchorder</code>	<code>\etex_gluestretchorder:D</code>
716	<code>\name_primitive:NN \glueshrinkorder</code>	<code>\etex_glueshrinkorder:D</code>
717	<code>\name_primitive:NN \gluetomu</code>	<code>\etex_gluetomu:D</code>
718	<code>\name_primitive:NN \mutoglua</code>	<code>\etex_mutoglua:D</code>
719	<code>\name_primitive:NN \lastlinefit</code>	<code>\etex_lastlinefit:D</code>
720	<code>\name_primitive:NN \interlinepenalties</code>	<code>\etex_interlinepenalties:D</code>
721	<code>\name_primitive:NN \clubpenalties</code>	<code>\etex_clubpenalties:D</code>
722	<code>\name_primitive:NN \widowpenalties</code>	<code>\etex_widowpenalties:D</code>
723	<code>\name_primitive:NN \displaywidowpenalties</code>	<code>\etex_displaywidowpenalties:D</code>
724	<code>\name_primitive:NN \middle</code>	<code>\etex_middle:D</code>
725	<code>\name_primitive:NN \savinghyphcodes</code>	<code>\etex_savinghyphcodes:D</code>
726	<code>\name_primitive:NN \savingvdiscards</code>	<code>\etex_savingvdiscards:D</code>
727	<code>\name_primitive:NN \pagediscards</code>	<code>\etex_pagediscards:D</code>
728	<code>\name_primitive:NN \splitdiscards</code>	<code>\etex_splitdiscards:D</code>
729	<code>\name_primitive:NN \TeXstate</code>	<code>\etex_TeXstate:D</code>
730	<code>\name_primitive:NN \beginL</code>	<code>\etex_beginL:D</code>
731	<code>\name_primitive:NN \endL</code>	<code>\etex_endL:D</code>
732	<code>\name_primitive:NN \beginR</code>	<code>\etex_beginR:D</code>
733	<code>\name_primitive:NN \endR</code>	<code>\etex_endR:D</code>
734	<code>\name_primitive:NN \predisplaydirection</code>	<code>\etex_predisplaydirection:D</code>
735	<code>\name_primitive:NN \everyeof</code>	<code>\etex_everyeof:D</code>
736	<code>\name_primitive:NN \protected</code>	<code>\etex_protected:D</code>

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective. In the case of the pdfTeX primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of pdfTeX primitives that start `\pdf...` but are not related to PDF output.

These ones re

```

737 \name_primitive:NN \pdfcreationdate \pdftex_pdfcreationdate:D
738 \name_primitive:NN \pdfcolorstack \pdftex_pdfcolorstack:D
739 \name_primitive:NN \pdfcompresslevel \pdftex_pdfcompresslevel:D
740 \name_primitive:NN \pdfdecimaldigits \pdftex_pdfdecimaldigits:D
741 \name_primitive:NN \pdfhorigin \pdftex_pdfhorigin:D
742 \name_primitive:NN \pdfinfo \pdftex_pdfinfo:D
743 \name_primitive:NN \pdfliteral \pdftex_pdfliteral:D
744 \name_primitive:NN \pdfminorversion \pdftex_pdfminorversion:D
745 \name_primitive:NN \pdfobjcompresslevel \pdftex_pdfobjcompresslevel:D
746 \name_primitive:NN \pdfoutput \pdftex_pdfoutput:D
747 \name_primitive:NN \pdfrestore \pdftex_pdfrestore:D
748 \name_primitive:NN \pdfsave \pdftex_pdfsave:D
749 \name_primitive:NN \pdfsetmatrix \pdftex_pdfsetmatrix:D
750 \name_primitive:NN \pdfpkresolution \pdftex_pdfpkresolution:D
751 \name_primitive:NN \pdftexrevision \pdftex_pdftextrevision:D
752 \name_primitive:NN \pdfvorigin \pdftex_pdfvorigin:D

```

While these are not.

```

753 \name_primitive:NN \pdfstrcmp \pdftex_strcmp:D

```

X_qTeX-specific primitives.

```

754 \name_primitive:NN \XeTeXversion \xetex_XeTeXversion:D

```

Primitives from Lua_TEX.

```

755 \name_primitive:NN \catcodetable \luatex_catcodetable:D
756 \name_primitive:NN \directlua \luatex_directlua:D
757 \name_primitive:NN \initcatcodetable \luatex_initcatcodetable:D
758 \name_primitive:NN \latelua \luatex_latelua:D
759 \name_primitive:NN \luatexversion \luatex_luatexversion:D
760 \name_primitive:NN \savecatcodetable \luatex_savecatcodetable:D

```

The job is done: close the group (using the primitive renamed!).

```

761 \tex_endgroup:D

```

L_AT_EX 2_ε will have moved a few primitives, so these are sorted out.

```

762 <*package>
763 \tex_let:D \tex_end:D \@@end
764 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
765 \tex_let:D \tex_everymath:D \frozen@everymath
766 \tex_let:D \tex_hyphen:D \@@hyph
767 \tex_let:D \tex_input:D \@@input
768 \tex_let:D \tex_italic_correction:D \@@italiccorr
769 \tex_let:D \tex_underline:D \@@underline

```

That is also true for the `luatex` package for L^AT_EX 2_ε.

```

770 \tex_let:D \luatex_catcodetable:D \luatexcatcodetable
771 \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
772 \tex_let:D \luatex_latelua:D \luatexlatelua
773 \tex_let:D \luatex_savecatcodetable:D \luatexsavecatcodetable
774 \</package>

775 \</initex | package>

```

164 l3basics implementation

```

776 \<*initex | package>
777 \<*package>
778 \ProvidesExplPackage
779   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
780 \package_check_loaded_expl:
781 \</package>

```

164.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.³

`\cs_set_eq:NwN` A pretty basic requirement: `\let` one control sequence to another.

```

782 >
783 \tex_let:D \cs_set_eq:NwN \tex_let:D

```

(End definition for `\cs_set_eq:NwN`. This function is documented on page ??.)

Then some conditionals.

<code>\if_true:</code>	784 <code>\cs_set_eq:NwN \if_true:</code>	<code>\tex_iftrue:D</code>
<code>\if_false:</code>	785 <code>\cs_set_eq:NwN \if_false:</code>	<code>\tex_iffalse:D</code>
<code>\or:</code>	786 <code>\cs_set_eq:NwN \or:</code>	<code>\tex_or:D</code>
<code>\else:</code>	787 <code>\cs_set_eq:NwN \else:</code>	<code>\tex_else:D</code>
<code>\fi:</code>	788 <code>\cs_set_eq:NwN \fi:</code>	<code>\tex_fi:D</code>
<code>\reverse_if:N</code>	789 <code>\cs_set_eq:NwN \reverse_if:N</code>	<code>\etex_unless:D</code>
<code>\if:w</code>	790 <code>\cs_set_eq:NwN \if:w</code>	<code>\tex_if:D</code>
<code>\if_bool:N</code>	791 <code>\cs_set_eq:NwN \if_bool:N</code>	<code>\tex_ifodd:D</code>
<code>\if_predicate:w</code>	792 <code>\cs_set_eq:NwN \if_predicate:w</code>	<code>\tex_ifodd:D</code>
<code>\if_charcode:w</code>	793 <code>\cs_set_eq:NwN \if_charcode:w</code>	<code>\tex_if:D</code>
<code>\if_catcode:w</code>	794 <code>\cs_set_eq:NwN \if_catcode:w</code>	<code>\tex_ifcat:D</code>

³This 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.

(End definition for `\if_true:`. This function is documented on page 26.)

`\if_meaning:w`

```
795 \cs_set_eq:NwN \if_meaning:w      \tex_ifx:D
```

(End definition for `\if_meaning:w`. This function is documented on page 25.)

`\if_mode_math:`

TeX lets us detect some of its modes.

`\if_mode_horizontal:`

`\if_mode_vertical:`

`\if_mode_inner:`

```
796 \cs_set_eq:NwN \if_mode_math:      \tex_ifmmode:D
797 \cs_set_eq:NwN \if_mode_horizontal: \tex_ifhmode:D
798 \cs_set_eq:NwN \if_mode_vertical:   \tex_ifvmode:D
799 \cs_set_eq:NwN \if_mode_inner:      \tex_ifinner:D
```

(End definition for `\if_mode_math:`. This function is documented on page 26.)

`\if_cs_exist:N`

`\if_cs_exist:w`

```
800 \cs_set_eq:NwN \if_cs_exist:N      \etex_ifdefined:D
801 \cs_set_eq:NwN \if_cs_exist:w      \etex_ifcsname:D
```

(End definition for `\if_cs_exist:N`. This function is documented on page 26.)

`\exp_after:wN`

`\exp_not:N`

`\exp_not:n`

The three `\exp_` functions are used in the `l3expan` module where they are described.

```
802 \cs_set_eq:NwN \exp_after:wN      \tex_expandafter:D
803 \cs_set_eq:NwN \exp_not:N         \tex_noexpand:D
804 \cs_set_eq:NwN \exp_not:n         \tex_unexpanded:D
```

(End definition for `\exp_after:wN`. This function is documented on page 35.)

`\token_to_meaning:N`

`\token_to_str:N`

`\cs:w`

`\cs_end:`

`\cs_meaning:N`

`\cs_show:N`

```
805 \cs_set_eq:NwN \token_to_meaning:N \tex_meaning:D
806 \cs_set_eq:NwN \token_to_str:N     \tex_string:D
807 \cs_set_eq:NwN \cs:w               \tex_csname:D
808 \cs_set_eq:NwN \cs_end:             \tex_endcsname:D
809 \cs_set_eq:NwN \cs_meaning:N       \tex_meaning:D
810 \cs_set_eq:NwN \cs_show:N          \tex_show:D
```

(End definition for `\token_to_meaning:N`. This function is documented on page 16.)

`\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.

```
811 \cs_set_eq:NwN \scan_stop:         \tex_relax:D
812 \cs_set_eq:NwN \group_begin:       \tex_begingroup:D
813 \cs_set_eq:NwN \group_end:         \tex_endgroup:D
```

(End definition for `\scan_stop:`. This function is documented on page 8.)

`\if_int_compare:w`
`\int_to_roman:w`

```
814 \cs_set_eq:NwN \if_int_compare:w \tex_ifnum:D
815 \cs_set_eq:NwN \int_to_roman:w \tex_romannumeral:D
```

(End definition for `\if_int_compare:w`. This function is documented on page 80.)

`\group_insert_after:N`

```
816 \cs_set_eq:NwN \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 8.)

`\pref_global:D`
`\pref_long:D`
`\pref_protected:D`

```
817 \cs_set_eq:NwN \pref_global:D \tex_global:D
818 \cs_set_eq:NwN \pref_long:D \tex_long:D
819 \cs_set_eq:NwN \pref_protected:D \etex_protected:D
```

(End definition for `\pref_global:D`. This function is documented on page 27.)

`\exp_args:Nc` Discussed in l3expan, but needed much earlier.

```
820 \tex_long:D \tex_def:D \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 30.)

`\token_to_str:c` A small number of variants by hand.
`\cs_meaning:c`
`\cs_show:c`

```
821 \tex_def:D \cs_meaning:c { \exp_args:Nc \cs_meaning:N }
822 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
823 \tex_def:D \cs_show:c { \exp_args:Nc \cs_show:N }
```

(End definition for `\token_to_str:c`. This function is documented on page 16.)

164.2 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.

```
824 \cs_set_eq:NwN \cs_set_nopar:Npn \tex_def:D
825 \cs_set_eq:NwN \cs_set_nopar:Npx \tex_edef:D
826 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npn
827 { \pref_long:D \cs_set_nopar:Npn }
828 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npx
829 { \pref_long:D \cs_set_nopar:Npx }
830 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
831 { \pref_protected:D \cs_set_nopar:Npn }
```

```

832 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx
833 { \pref_protected:D \cs_set_nopar:Npx }
834 \cs_set_protected_nopar:Npn \cs_set_protected:Npn
835 { \pref_protected:D \pref_long:D \cs_set_nopar:Npn }
836 \cs_set_protected_nopar:Npn \cs_set_protected:Npx
837 { \pref_protected:D \pref_long:D \cs_set_nopar:Npx }

```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 10.)

```

\cs_gset_nopar:Npn Global versions of the above functions.
\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
838 \cs_set_eq:NwN \cs_gset_nopar:Npn \tex_gdef:D
839 \cs_set_eq:NwN \cs_gset_nopar:Npx \tex_xdef:D
840 \cs_set_protected_nopar:Npn \cs_gset:Npn
841 { \pref_long:D \cs_gset_nopar:Npn }
842 \cs_set_protected_nopar:Npn \cs_gset:Npx
843 { \pref_long:D \cs_gset_nopar:Npx }
844 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
845 { \pref_protected:D \cs_gset_nopar:Npn }
846 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
847 { \pref_protected:D \cs_gset_nopar:Npx }
848 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn
849 { \pref_protected:D \pref_long:D \cs_gset_nopar:Npn }
850 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx
851 { \pref_protected:D \pref_long:D \cs_gset_nopar:Npx }

```

(End definition for `\cs_gset_nopar:Npn`. This function is documented on page 11.)

164.3 Selecting tokens

\use:c This macro grabs its argument and returns a csname from it.

```

852 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End definition for `\use:c`. This function is documented on page 16.)

\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.

```

853 \cs_set_protected:Npn \use:x #1
854 {
855   \group_begin:
856   \cs_set:Npx \cs_tmp:w {#1}
857   \exp_after:wN
858   \group_end:
859   \cs_tmp:w
860 }
861 \cs_set:Npn \cs_tmp:w { }

```


`\use:n` These macro grabs its arguments and returns it back to the input (with outer braces removed).

`\use:nn`
`\use:nnn`
`\use:nnnn`

```
862 \cs_set:Npn \use:n #1 {#1}
863 \cs_set:Npn \use:nn #1#2 {#1#2}
864 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
865 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

`\use_i:nn` The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

`\use_ii:nn`

```
866 \cs_set:Npn \use_i:nn #1#2 {#1}
867 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

`\use_i:nnn` We also need something for picking up arguments from a longer list.

`\use_ii:nnn`
`\use_iii:nnn`
`\use_i_ii:nnn`
`\use_i:nnnn`
`\use_ii:nnnn`
`\use_iii:nnnn`
`\use_iv:nnnn`

```
868 \cs_set:Npn \use_i:nnn #1#2#3 {#1}
869 \cs_set:Npn \use_ii:nnn #1#2#3 {#2}
870 \cs_set:Npn \use_iii:nnn #1#2#3 {#3}
871 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
872 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}
873 \cs_set:Npn \use_ii:nnnn #1#2#3#4 {#2}
874 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
875 \cs_set:Npn \use_iv:nnnn #1#2#3#4 {#4}
```

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil` or `\q_stop`, respectively.

`\use_none_delimit_by_q_stop:w`
`\use_none_delimit_by_q_recursion_stop:w`

```
876 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
877 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
878 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

`\use_i_delimit_by_q_nil:nw` Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

`\use_i_delimit_by_q_stop:nw`
`\use_i_delimit_by_q_recursion_stop:nw`

```
879 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
880 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
881 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

`\use_i_after-fi:nw` Returns the first argument after ending the conditional.

`\use_i_after_else:nw`
`\use_i_after_or:nw`
`\use_i_after_orelse:nw`

```
882 \cs_set:Npn \use_i_after-fi:nw #1 \fi: { \fi: #1 }
883 \cs_set:Npn \use_i_after_else:nw #1 \else: #2 \fi: { \fi: #1 }
884 \cs_set:Npn \use_i_after_or:nw #1 \or: #2 \fi: { \fi: #1 }
885 \cs_set:Npn \use_i_after_orelse:nw #1#2#3 \fi: { \fi: #1 }
```

164.4 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.

```

886 \cs_set:Npn \use_none:n      #1          { }
887 \cs_set:Npn \use_none:nn    #1#2        { }
888 \cs_set:Npn \use_none:nnn   #1#2#3      { }
889 \cs_set:Npn \use_none:nnnn  #1#2#3#4    { }
890 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5  { }
891 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
892 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
893 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
894 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }

```

164.5 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:
\prg_return_false:

```

The idea here is that `\int_to_roman:w` will expand fully any `\else:` and the `\fi:` 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.

```

895 \cs_set_nopar:Npn \prg_return_true:
896 { \exp_after:wN \use_i:nn \int_to_roman:w }
897 \cs_set_nopar:Npn \prg_return_false:
898 { \exp_after:wN \use_ii:nn \int_to_roman:w}

```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code will work.

`\prg_set_conditional:Npnn`
`\prg_new_conditional:Npnn`
`\prg_set_protected_conditional:Npnn`
`\prg_new_protected_conditional:Npnn`

The user functions for the types using parameter text from the programmer. Call aux function to grab parameters, split the base function into name and signature and then use, *e.g.*, `\cs_set:Npn` to define it with.

```

899 \cs_set_protected:Npn \prg_set_conditional:Npnn #1
900 {
901   \prg_get_parm_aux:nw
902   {
903     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
904     \cs_set:Npn { parm }
905   }
906 }
907 \cs_set_protected:Npn \prg_new_conditional:Npnn #1
908 {
909   \prg_get_parm_aux:nw
910   {
911     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
912     \cs_new:Npn { parm }
913   }
914 }
915 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn #1
916 {
917   \prg_get_parm_aux:nw{
918     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
919     \cs_set_protected:Npn { parm }
920   }
921 }
922 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn #1
923 {
924   \prg_get_parm_aux:nw
925   {
926     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
927     \cs_new_protected:Npn { parm }
928   }
929 }
```

`\prg_set_conditional:Nnn`
`\prg_new_conditional:Nnn`
`\prg_set_protected_conditional:Nnn`
`\prg_new_protected_conditional:Nnn`

The user functions for the types automatically inserting the correct parameter text based 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.

```

930 \cs_set_protected:Npn \prg_set_conditional:Nnn #1
931 {
932   \exp_args:Nnf \prg_get_count_aux:nn
933   {
934     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
```

```

935     \cs_set:Npn { count }
936   }
937   { \cs_get_arg_count_from_signature:N #1 }
938 }
939 \cs_set_protected:Npn \prg_new_conditional:Nnn #1
940 {
941   \exp_args:Nnf \prg_get_count_aux:nn
942   {
943     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
944     \cs_new:Npn { count}
945   }
946   { \cs_get_arg_count_from_signature:N #1 }
947 }
948
949 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn #1{
950   \exp_args:Nnf \prg_get_count_aux:nn{
951     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
952     \cs_set_protected:Npn {count}
953   }{\cs_get_arg_count_from_signature:N #1}
954 }
955
956 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn #1
957 {
958   \exp_args:Nnf \prg_get_count_aux:nn
959   {
960     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
961     \cs_new_protected:Npn {count}
962   }
963   { \cs_get_arg_count_from_signature:N #1 }
964 }

```

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn

The obvious setting-equal functions.

```

965 \cs_set_protected:Npn \prg_set_eq_conditional:NNn #1#2#3
966 { \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3} }
967 \cs_set_protected:Npn \prg_new_eq_conditional:NNn #1#2#3
968 { \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3} }

```

\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.

```

969 \cs_set:Npn \prg_get_count_aux:nn #1#2 { #1 {#2} }
970 \cs_set:Npn \prg_get_parm_aux:nw #1#2# { #1 {#2} }

```

\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 eight is the replacement text which we will augment when defining the forms.

```

971 \cs_set_protected:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8
972 {
973   \prg_generate_conditional_aux:nnw {#5}
974   {
975     #4 {#1} {#2} {#6} {#8}
976   }
977   #7 , ? , \q_recursion_stop
978 }

```

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.

```

979 \cs_set_protected:Npn \prg_generate_conditional_aux:nnw #1#2#3 ,
980 {
981   \if:w ?#3
982     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
983   \fi:
984   \use:c { prg_generate_#3_form_#1:Nnnnn } #2
985   \prg_generate_conditional_aux:nnw {#1} {#2}
986 }

```

`\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.

```

987 \cs_set_protected:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5
988 {
989   \exp_args:Nc #1 { #2 _p: #3 } #4
990   {
991     #5 \c_zero
992     \c_true_bool \c_false_bool
993   }
994 }
995 \cs_set_protected:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5
996 {
997   \exp_args:Nc #1 { #2 : #3 T } #4
998   {
999     #5 \c_zero
1000     \use:n \use_none:n
1001   }
1002 }

```

```

1003 \cs_set_protected:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5
1004 {
1005   \exp_args:Nc #1 { #2 : #3 F } #4
1006   {
1007     #5 \c_zero
1008     { }
1009   }
1010 }
1011 \cs_set_protected:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5
1012 {
1013   \exp_args:Nc #1 { #2 : #3 TF } #4
1014   { #5 \c_zero }
1015 }

```

```

\prg_generate_p_form_count:Nnnnn
\prg_generate_TF_form_count:Nnnnn
\prg_generate_T_form_count:Nnnnn
\prg_generate_F_form_count:Nnnnn

```

The **count** form is similar, but of course requires a number rather than a primitive argument specification.

```

1016 \cs_set_protected:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5
1017 {
1018   \cs_generate_from_arg_count:cNnn { #2 _p: #3 } #1 {#4}
1019   {
1020     #5 \c_zero
1021     \c_true_bool \c_false_bool
1022   }
1023 }
1024 \cs_set_protected:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5
1025 {
1026   \cs_generate_from_arg_count:cNnn { #2 : #3 T } #1 {#4}
1027   {
1028     #5 \c_zero
1029     \use:n \use_none:n
1030   }
1031 }
1032 \cs_set_protected:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5
1033 {
1034   \cs_generate_from_arg_count:cNnn { #2 : #3 F } #1 {#4}
1035   {
1036     #5 \c_zero
1037     { }
1038   }
1039 }
1040 \cs_set_protected:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5
1041 {
1042   \cs_generate_from_arg_count:cNnn { #2 : #3 TF } #1 {#4}
1043   { #5 \c_zero }
1044 }

```

```

\prg_set_eq_conditional_aux:NNNn
\prg_set_eq_conditional_aux:NNNw

```

```

1045 \cs_set_protected:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4
1046 { \prg_set_eq_conditional_aux:NNNw #1#2#3#4 , ? , \q_recursion_stop }

```

Manual clist loop over argument #4.

```

1047 \cs_set_protected:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4 ,
1048 {
1049   \if:w ? #4 \scan_stop:
1050   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1051   \fi:
1052   #1
1053   { \exp_args:NNc \cs_split_function:NN #2 { prg_conditional_form_#4:nnn } }
1054   { \exp_args:NNc \cs_split_function:NN #3 { prg_conditional_form_#4:nnn } }
1055   \prg_set_eq_conditional_aux:NNNw #1 {#2} {#3}
1056 }

1057 \cs_set:Npn \prg_conditional_form_p:nnn #1#2#3 { #1 _p : #2 }
1058 \cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 { #1 : #2 TF }
1059 \cs_set:Npn \prg_conditional_form_T:nnn #1#2#3 { #1 : #2 T }
1060 \cs_set:Npn \prg_conditional_form_F:nnn #1#2#3 { #1 : #2 F }

```

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`

```

1061 \tex_chardef:D \c_true_bool = 1~
1062 \tex_chardef:D \c_false_bool = 0~

```

164.6 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 are 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 `\int_to_roman:w`, 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 `\int_to_roman:w`, and `\cs_to_str_aux:w` is expanded. This inserts a space, making the `\if:w` test `true`. The second `\int_to_roman:w` will then execute the `\token_to_str:N`, with the escape-character space being consumed by the `\int_to_roman:w`, 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 `\int_to_roman:w` but leaves no token for the `\if:w` test. This means that the `\int_to_roman:w` is executed before the test is finished. The result is that the `\fi:`, expanded before the `\tex_if:D` is finished, becomes `\scan_stop: \fi:`, and the `\scan_stop:` is then used in the `\if:w` 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.

```

1063 \cs_set_nopar:Npn \cs_to_str:N
1064 {
1065   \if:w \int_to_roman:w - '0 \token_to_str:N \ %
1066     \cs_to_str_aux:w
1067   \fi:
1068   \exp_after:wN \use_none:n \token_to_str:N
1069 }
1070 \cs_set_nopar:Npn \cs_to_str_aux:w #1 \use_none:n
1071 { ~ \int_to_roman:w - '0 \fi: }

```

`\cs_split_function:NN`
`\cs_split_function_aux:w`
`\cs_split_function_auxii:w`

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>` 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`.

```

1072 \group_begin:
1073   \tex_lccode:D '@ = '\: \scan_stop:
1074   \tex_catcode:D '@ = 12~
1075   \tex_lowercase:D
1076   {
1077     \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.

```

1078 \cs_set:Npn \cs_split_function:NN #1#2
1079 {
1080   \exp_after:wN \cs_split_function_aux:w
1081   \int_to_roman:w - '\q \cs_to_str:N #1 @ a \q_stop #2
1082 }

```

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.

```

1083 \cs_set:Npn \cs_split_function_aux:w #1 @ #2#3 \q_stop #4
1084 {
1085   \if_meaning:w a #2
1086   \exp_after:wN \use_i:nn
1087   \else:
1088     \exp_after:wN \use_ii:nn
1089   \fi:
1090   { #4 {#1} { } \c_false_bool }
1091   { \cs_split_function_auxii:w #2#3 \q_stop #4 {#1} }
1092 }
1093 \cs_set:Npn \cs_split_function_auxii:w #1 @a \q_stop #2#3
1094 { #2{#3}{#1}\c_true_bool }

```

End of lowercase

```

1095 }

```

`\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.

```

1096 \cs_set:Npn \cs_get_function_name:N #1
1097 { \cs_split_function:NN #1 \use_i:nnn }
1098 \cs_set:Npn \cs_get_function_signature:N #1
1099 { \cs_split_function:NN #1 \use_ii:nnn }

```

164.7 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.

`\cs_if_exist_p:N` Two versions for checking existence. For the N form we firstly check for `\scan_stop:` and
`\cs_if_exist_p:c` then if it is in the hash table. There is no problem when inputting something like `\else:`
`\cs_if_exist:NTF` or `\fi:` as TeX will only ever skip input in case the token tested against is `\scan_stop:`.
`\cs_if_exist:cTF`

```

1100 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1101 {
1102   \if_meaning:w #1 \scan_stop:
1103   \prg_return_false:
1104   \else:
1105     \if_cs_exist:N #1
1106     \prg_return_true:
1107     \else:
1108       \prg_return_false:
1109     \fi:
1110   \fi:
1111 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1112 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1113 {
1114   \if_cs_exist:w #1 \cs_end:
1115   \exp_after:wN \use_i:nn
1116   \else:
1117     \exp_after:wN \use_ii:nn
1118   \fi:
1119   {
1120     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1121     \prg_return_false:
1122     \else:
1123       \prg_return_true:
1124     \fi:
1125   }
1126   \prg_return_false:
1127 }

```

(End definition for `\use:x`. This function is documented on page 23.)

`\cs_if_free_p:N`
`\cs_if_free_p:c`
`\cs_if_free:NTF`
`\cs_if_free:cTF`

The logical reversal of the above.

```

1128 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1129 {
1130   \if_meaning:w #1 \scan_stop:
1131   \prg_return_true:
1132   \else:
1133     \if_cs_exist:N #1
1134     \prg_return_false:
1135     \else:
1136       \prg_return_true:
1137     \fi:

```

```

1138     \fi:
1139   }
1140   \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1141   {
1142     \if_cs_exist:w #1 \cs_end:
1143     \exp_after:wN \use_i:nn
1144     \else:
1145     \exp_after:wN \use_ii:nn
1146     \fi:
1147     {
1148       \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1149       \prg_return_true:
1150     }
1151     \else:
1152     \prg_return_false:
1153     \fi:
1154   }
1155   { \prg_return_true: }

```

(End definition for `\cs_if_free:N` and `\cs_if_free:c`. These functions are documented on page 23.)

164.8 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
`\c_six` with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is
`\c_seven` required but it can't be used until the allocation has been set up properly! The actual
`\c_twelve` 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`.

```

1156 <*package>
1157 \cs_set_eq:NwN \c_minus_one \m@ne
1158 </package>
1159 <*initex>
1160 \tex_countdef:D \c_minus_one = 10 ~
1161 \c_minus_one = -1 ~
1162 </initex>
1163 \tex_chardef:D \c_sixteen = 16~
1164 \tex_chardef:D \c_zero = 0~
1165 \tex_chardef:D \c_six = 6~
1166 \tex_chardef:D \c_seven = 7~
1167 \tex_chardef:D \c_twelve = 12~

```

(End definition for `\c_minus_one`, `\c_zero`, and `\c_sixteen`. These functions are documented on page 79.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`

```
1168 \tex_mathchardef:D \c_max_register_int = 32 767 \scan_stop:
```

(End definition for `\c_max_register_int`. This function is documented on page 79.)

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. These will be redefined later by `l3io`.

```
1169 \cs_set_protected_nopar:Npn \iow_log:x
1170 { \tex_immediate:D \tex_write:D \c_minus_one }
1171 \cs_set_protected_nopar:Npn \iow_term:x
1172 { \tex_immediate:D \tex_write:D \c_sixteen }
```

(End definition for `\iow_log:x`. This function is documented on page 155.)

`\msg_kernel_error:nxxx` If an internal error occurs before L^AT_EX3 has loaded `l3msg` then the code should issue a
`\msg_kernel_error:nxx` usable if terse error message and halt. This can only happen if a coding error is made by
`\msg_kernel_error:nn` the team, so this is a reasonable response.

```
1173 \cs_set_protected_nopar:Npn \msg_kernel_error:nxxx #1#2#3#4
1174 {
1175   \tex_errmessage:D
1176   {
1177     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1178     Argh,~internal~LaTeX3~error! ^^J ^^J
1179     Module ~ #1 , ~ message~name~"#2": ^^J
1180     Arguments~'#3'~and~'#4' ^^J ^^J
1181     This~is~one~for~The~LaTeX3~Project:~bailing~out
1182   }
1183   \tex_end:D
1184 }
1185 \cs_set_protected_nopar:Npn \msg_kernel_error:nxx #1#2#3
1186 { \msg_kernel_error:nxxx {#1} {#2} {#3} { } }
1187 \cs_set_protected_nopar:Npn \msg_kernel_error:nn #1#2
1188 { \msg_kernel_error:nxxx {#1} {#2} { } { } }
```

(End definition for `\msg_kernel_error:nxxx`. This function is documented on page 166.)

`\msg_line_context:` Another one from `l3msg` which will be altered later.

```
1189 \cs_set_nopar:Npn \msg_line_context:
1190 { on~line~\tex_the:D \tex_inputlineno:D }
```

(End definition for `\msg_line_context:`. This function is documented on page 160.)

`\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.

```
1191 \cs_set:Npn \cs_record_meaning:N #1 { }
```

(End definition for `\cs_record_meaning:N`. This function is documented on page ??.)

`\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!

```
1192 \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1
1193 {
1194   \cs_if_free:NF #1
1195   {
1196     \msg_kernel_error:nnxx { kernel } { command-already-defined }
1197     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1198   }
1199 }
1200 <*package>
1201 \tex_ifodd:D \@l@expl@log@functions@bool
1202 \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1
1203 {
1204   \cs_if_free:NF #1
1205   {
1206     \msg_kernel_error:nnxx { kernel } { command-already-defined }
1207     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1208   }
1209   \iow_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1210 }
1211 \fi:
1212 </package>
1213 \cs_set_protected_nopar:Npn \chk_if_free_cs:c
1214 { \exp_args:Nc \chk_if_free_cs:N }
```

(End definition for `\chk_if_free_cs:N` and `\chk_if_free_cs:c`. These functions are documented on page 27.)

`\chk_if_exist_cs:N` This function issues a warning message when the control sequence in its argument does
`\chk_if_exist_cs:c` not exist.

```
1215 \cs_set_protected_nopar:Npn \chk_if_exist_cs:N #1
1216 {
1217   \cs_if_exist:NF #1
1218   {
1219     \msg_kernel_error:nnxx { kernel } { command-not-defined }
1220     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1221   }
1222 }
```

```

1222 }
1223 \cs_set_protected_nopar:Npn \chk_if_exist_cs:c
1224 { \exp_args:Nc \chk_if_exist_cs:N }

```

(End definition for `\chk_if_exist_cs:N` and `\chk_if_exist_cs:c`. These functions are documented on page 27.)

164.9 More new definitions

Global versions of the above functions.

```

\cs_new_nopar:Npn
\cs_new_nopar:Npx
  \cs_new:Npn
  \cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
  \cs_new_protected:Npn
  \cs_new_protected:Npx

```

```

1225 \cs_set:Npn \cs_tmp:w #1#2
1226 {
1227   \cs_set_protected_nopar:Npn #1 ##1
1228   {
1229     \chk_if_free_cs:N ##1
1230     #2 ##1
1231   }
1232 }
1233 \cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
1234 \cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1235 \cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
1236 \cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
1237 \cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1238 \cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1239 \cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
1240 \cs_tmp:w \cs_new_protected:Npx       \cs_gset_protected:Npx

```

(End definition for `\cs_new_nopar:Npn`. This function is documented on page 9.)

```

\cs_set_nopar:cpn
\cs_set_nopar:cpx
\cs_gset_nopar:cpn
\cs_gset_nopar:cpx
\cs_new_nopar:cpn
\cs_new_nopar:cpx

```

Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for `csname` argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn⟨string⟩⟨rep-text⟩` will turn `⟨string⟩` into a `csname` and then assign `⟨rep-text⟩` to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1241 \cs_set:Npn \cs_tmp:w #1#2
1242 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1243 \cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1244 \cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1245 \cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1246 \cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1247 \cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1248 \cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:cpn`. This function is documented on page 9.)

`\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
1249 \cs_tmp:w \cs_set:cpn \cs_set:Npn
1250 \cs_tmp:w \cs_set:cpx \cs_set:Npx
1251 \cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1252 \cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1253 \cs_tmp:w \cs_new:cpn \cs_new:Npn
1254 \cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:cpn`. This function is documented on page 9.)

`\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`

```

1255 \cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1256 \cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1257 \cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1258 \cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1259 \cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1260 \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 9.)

`\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`

```

1261 \cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1262 \cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1263 \cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1264 \cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1265 \cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1266 \cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for `\cs_set_protected:cpn`. This function is documented on page 9.)

`\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:
1267 \cs_new_nopar:cpn { use_0_parameter: } { }
1268 \cs_new_nopar:cpn { use_1_parameter: } { {##1} }
1269 \cs_new_nopar:cpn { use_2_parameter: } { {##1} {##2} }
1270 \cs_new_nopar:cpn { use_3_parameter: } { {##1} {##2} {##3} }
1271 \cs_new_nopar:cpn { use_4_parameter: } { {##1} {##2} {##3} {##4} }
1272 \cs_new_nopar:cpn { use_5_parameter: } { {##1} {##2} {##3} {##4} {##5} }
1273 \cs_new_nopar:cpn { use_6_parameter: } { {##1} {##2} {##3} {##4} {##5} {##6} }
1274 \cs_new_nopar:cpn { use_7_parameter: }
1275 { {##1} {##2} {##3} {##4} {##5} {##6} {##7} }
1276 \cs_new_nopar:cpn { use_8_parameter: }
1277 { {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} }
1278 \cs_new_nopar:cpn { use_9_parameter: }
1279 { {##1} {##2} {##3} {##4} {##5} {##6} {##7} {##8} {##9} }

```

(End definition for `\use_0_parameter:.`)

164.10 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_char{~}` 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”.

```
1280 \cs_new_protected:Npn \cs_set_eq:NN #1 { \cs_set_eq:NwN #1 =~ }
1281 \cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1282 \cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:Nnc \cs_set_eq:NN }
1283 \cs_new_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 15.)

`\cs_new_eq:NN`
`\cs_new_eq:cN`
`\cs_new_eq:Nc`
`\cs_new_eq:cc`

```
1284 \cs_new_protected:Npn \cs_new_eq:NN #1
1285 {
1286   \chk_if_free_cs:N #1
1287   \pref_global:D \cs_set_eq:NN #1
1288 }
1289 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1290 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:Nnc \cs_new_eq:NN }
1291 \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 15.)

`\cs_gset_eq:NN`
`\cs_gset_eq:cN`
`\cs_gset_eq:Nc`
`\cs_gset_eq:cc`

```
1292 \cs_new_protected_nopar:Npn \cs_gset_eq:NN { \pref_global:D \cs_set_eq:NN }
1293 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:Nnc \cs_gset_eq:NN }
1294 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1295 \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 15.)

164.11 Undefining functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some
`\cs_undefine:c` function that isn't in use any longer.

```
1296 \cs_new_protected_nopar:Npn \cs_undefine:N #1
```



```

1297 { \cs_gset_eq:NN #1 \c_undefined:D }
1298 \cs_new_protected_nopar:Npn \cs_undefine:c #1
1299 { \cs_gset_eq:cN {#1} \c_undefined:D }

```

(End definition for `\cs_undefine:N` and `\cs_undefine:c`. These functions are documented on page 16.)

164.12 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.

```

1300 \cs_new:Npn \cs_get_arg_count_from_signature:N #1
1301 { \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN }
1302 \cs_new:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3
1303 {
1304   \if_predicate:w #3
1305     \exp_after:wN \use_i:nn
1306   \else:
1307     \exp_after:wN \use_ii:nn
1308   \fi:
1309   {
1310     \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
1311     \use_none:nnnnnnnn #2 9876543210 \q_stop
1312   }
1313   { -1 }
1314 }
1315 \cs_new:Npn \cs_get_arg_count_from_signature_auxii:w #1#2 \q_stop {#1}

```

A variant form we need right away.

```

1316 \cs_new_nopar:Npn \cs_get_arg_count_from_signature:c
1317 { \exp_args:Nc \cs_get_arg_count_from_signature:N }

```

(End definition for `\cs_get_arg_count_from_signature:N`. This function is documented on page 21.)

```

\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

```

1318 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1319 {
1320   \if_case:w \int_eval:w #3 \int_eval_end:
1321     \use_i_after_orelse:nw {#2#1}
1322   \or:
1323     \use_i_after_orelse:nw {#2#1 ##1}
1324   \or:
1325     \use_i_after_orelse:nw {#2#1 ##1##2}
1326   \or:
1327     \use_i_after_orelse:nw {#2#1 ##1##2##3}
1328   \or:
1329     \use_i_after_orelse:nw {#2#1 ##1##2##3##4}
1330   \or:
1331     \use_i_after_orelse:nw {#2#1 ##1##2##3##4##5}
1332   \or:
1333     \use_i_after_orelse:nw {#2#1 ##1##2##3##4##5##6}
1334   \or:
1335     \use_i_after_orelse:nw {#2#1 ##1##2##3##4##5##6##7}
1336   \or:
1337     \use_i_after_orelse:nw {#2#1 ##1##2##3##4##5##6##7##8}
1338   \or:
1339     \use_i_after_orelse:nw {#2#1 ##1##2##3##4##5##6##7##8##9}
1340   \else:
1341     \use_i_after_fi:nw
1342     {
1343       \cs_generate_from_arg_count_error_msg:Nn #1 {#3}
1344       \use_none:n
1345     }
1346   \fi:
1347   {#4}
1348 }

```

A variant form we need right away.

```

1349 \cs_new_nopar:Npn \cs_generate_from_arg_count:cNnn
1350 { \exp_args:Nc \cs_generate_from_arg_count:NNnn }

```

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.

```

1351 \cs_new:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2
1352 {
1353   \msg_kernel_error:nxx { kernel } { bad-number-of-arguments }
1354   { \token_to_str:N #1 } { \int_eval:n {#2} }
1355 }

```

(End definition for `\cs_generate_from_arg_count:NNnn`.)

164.13 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

```

```

\cs_set_protected:Npn \cs_set:Nn #1#2
{
  \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
  { \cs_get_arg_count_from_signature:N #1 } {#2}
}

```

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.

```

1356 \cs_set:Npn \cs_tmp:w #1#2#3
1357 {
1358   \cs_set_protected:cpx { cs_ #1 : #2 } ##1##2
1359   {
1360     \exp_not:N \cs_generate_from_arg_count:NNnn ##1
1361     \exp_after:wN \exp_not:N \cs:w cs_#1 : #3 \cs_end:
1362     { \exp_not:N \cs_get_arg_count_from_signature:N ##1 }{##2}
1363   }
1364 }

```

Then we define the 32 variants beginning with N.

```

1365 \cs_tmp:w { set } { Nn } { Npn }
1366 \cs_tmp:w { set } { Nx } { Npx }
1367 \cs_tmp:w { set_nopar } { Nn } { Npn }
1368 \cs_tmp:w { set_nopar } { Nx } { Npx }
1369 \cs_tmp:w { set_protected } { Nn } { Npn }
1370 \cs_tmp:w { set_protected } { Nx } { Npx }
1371 \cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1372 \cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1373 \cs_tmp:w { gset } { Nn } { Npn }
1374 \cs_tmp:w { gset } { Nx } { Npx }
1375 \cs_tmp:w { gset_nopar } { Nn } { Npn }
1376 \cs_tmp:w { gset_nopar } { Nx } { Npx }
1377 \cs_tmp:w { gset_protected } { Nn } { Npn }
1378 \cs_tmp:w { gset_protected } { Nx } { Npx }
1379 \cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1380 \cs_tmp:w { gset_protected_nopar } { Nx } { Npx }

```

(End definition for `\cs_set:Nn`. This function is documented on page 14.)

```

\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

```

```

1381 \cs_tmp:w { new } { Nn } { Npn }
1382 \cs_tmp:w { new } { Nx } { Npx }
1383 \cs_tmp:w { new_nopar } { Nn } { Npn }
1384 \cs_tmp:w { new_nopar } { Nx } { Npx }
1385 \cs_tmp:w { new_protected } { Nn } { Npn }
1386 \cs_tmp:w { new_protected } { Nx } { Npx }
1387 \cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1388 \cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for `\cs_new:Nn`. This function is documented on page 12.)

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}
}

```

```

1389 \cs_set:Npn \cs_tmp:w #1#2#3
1390 {
1391   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1392     \exp_not:N\cs_generate_from_arg_count:cNnn {##1}
1393     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1394     { \exp_not:N \cs_get_arg_count_from_signature:c {##1} } {##2}
1395   }
1396 }

```

The 32 c variants.

```

\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

```

```

1397 \cs_tmp:w { set } { cn } { Npn }
1398 \cs_tmp:w { set } { cx } { Npx }
1399 \cs_tmp:w { set_nopar } { cn } { Npn }
1400 \cs_tmp:w { set_nopar } { cx } { Npx }
1401 \cs_tmp:w { set_protected } { cn } { Npn }
1402 \cs_tmp:w { set_protected } { cx } { Npx }
1403 \cs_tmp:w { set_protected_nopar } { cn } { Npn }
1404 \cs_tmp:w { set_protected_nopar } { cx } { Npx }
1405 \cs_tmp:w { gset } { cn } { Npn }
1406 \cs_tmp:w { gset } { cx } { Npx }
1407 \cs_tmp:w { gset_nopar } { cn } { Npn }
1408 \cs_tmp:w { gset_nopar } { cx } { Npx }
1409 \cs_tmp:w { gset_protected } { cn } { Npn }
1410 \cs_tmp:w { gset_protected } { cx } { Npx }
1411 \cs_tmp:w { gset_protected_nopar } { cn } { Npn }
1412 \cs_tmp:w { gset_protected_nopar } { cx } { Npx }

```

(End definition for `\cs_set:cn`. This function is documented on page 14.)

```

\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
1413 \cs_tmp:w { new } { cn } { Npn }
1414 \cs_tmp:w { new } { cx } { Npx }
1415 \cs_tmp:w { new_nopar } { cn } { Npn }
1416 \cs_tmp:w { new_nopar } { cx } { Npx }
1417 \cs_tmp:w { new_protected } { cn } { Npn }
1418 \cs_tmp:w { new_protected } { cx } { Npx }
1419 \cs_tmp:w { new_protected_nopar } { cn } { Npn }
1420 \cs_tmp:w { new_protected_nopar } { cx } { Npx }

```

(End definition for `\cs_new:cn`. This function is documented on page 12.)

164.14 Checking control sequence equality

```

\cs_if_eq_p:NN Check if two control sequences are identical.
\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
1421 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
1422 {
1423   \if_meaning:w #1#2
1424   \prg_return_true: \else: \prg_return_false: \fi:
1425 }
1426 \cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
1427 \cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
1428 \cs_new_nopar:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
1429 \cs_new_nopar:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
1430 \cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
1431 \cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
1432 \cs_new_nopar:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
1433 \cs_new_nopar:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
1434 \cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
1435 \cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
1436 \cs_new_nopar:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
1437 \cs_new_nopar:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for `\cs_if_eq:NN` and others. These functions are documented on page ??.)

164.15 Diagnostic wrapper functions

```

\kernel_register_show:N
\kernel_register_show:c
1438 \cs_new_nopar:Npn \kernel_register_show:N #1
1439 {
1440   \cs_if_exist:NTF #1
1441   { \tex_showthe:D #1 }
1442   {
1443     \msg_kernel_error:nxx { kernel } { variable-not-defined }
1444     { \token_to_str:N #1 }
1445   }

```

```

1446 }
1447 \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 ??.)

164.16 Engine specific definitions

`\c_pdftex_is_engine_bool` `\c_luatex_is_engine_bool` `\c_xetex_is_engine_bool` 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. This can all be hard-coded for speed.

```

\etex_if_engine:TF
\luatex_if_engine:TF
\pdftex_if_engine:TF
1448 \cs_new_eq:NN \luatex_if_engine:T \use_none:n
1449 \cs_new_eq:NN \luatex_if_engine:F \use:n
1450 \cs_new_eq:NN \luatex_if_engine:TF \use_ii:nn
1451 \cs_new_eq:NN \pdftex_if_engine:T \use:n
1452 \cs_new_eq:NN \pdftex_if_engine:F \use_none:n
1453 \cs_new_eq:NN \pdftex_if_engine:TF \use_i:nn
1454 \cs_new_eq:NN \xetex_if_engine:T \use_none:n
1455 \cs_new_eq:NN \xetex_if_engine:F \use:n
1456 \cs_new_eq:NN \xetex_if_engine:TF \use_ii:nn
1457 \cs_new_eq:NN \c_luatex_is_engine_bool \c_false_bool
1458 \cs_new_eq:NN \c_pdftex_is_engine_bool \c_true_bool
1459 \cs_new_eq:NN \c_xetex_is_engine_bool \c_false_bool
1460 \cs_if_exist:NT \xetex_XeTeXversion:D
1461 {
1462   \cs_set_eq:NN \pdftex_if_engine:T \use_none:n
1463   \cs_set_eq:NN \pdftex_if_engine:F \use:n
1464   \cs_set_eq:NN \pdftex_if_engine:TF \use_ii:nn
1465   \cs_set_eq:NN \xetex_if_engine:T \use:n
1466   \cs_set_eq:NN \xetex_if_engine:F \use_none:n
1467   \cs_set_eq:NN \xetex_if_engine:TF \use_i:nn
1468   \cs_set_eq:NN \c_pdftex_is_engine_bool \c_false_bool
1469   \cs_set_eq:NN \c_xetex_is_engine_bool \c_true_bool
1470 }
1471 \cs_if_exist:NT \luatex_directlua:D
1472 {
1473   \cs_set_eq:NN \luatex_if_engine:T \use:n
1474   \cs_set_eq:NN \luatex_if_engine:F \use_none:n
1475   \cs_set_eq:NN \luatex_if_engine:TF \use_i:nn
1476   \cs_set_eq:NN \pdftex_if_engine:T \use_none:n
1477   \cs_set_eq:NN \pdftex_if_engine:F \use:n
1478   \cs_set_eq:NN \pdftex_if_engine:TF \use_ii:nn
1479   \cs_set_eq:NN \c_luatex_is_engine_bool \c_true_bool
1480   \cs_set_eq:NN \c_pdftex_is_engine_bool \c_false_bool
1481 }

```

(End definition for `\c_pdftex_is_engine_bool`, `\c_luatex_is_engine_bool`, and `\c_xetex_is_engine_bool`. These functions are documented on page 24.)

164.17 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```
1482 \cs_new_nopar:Npn \prg_do_nothing: { }
```

(End definition for `\prg_do_nothing:`. This function is documented on page 7.)

164.18 String comparisons

`\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. These should eventually move somewhere else.

```
\str_if_eq:nnTF
\str_if_eq:nnTF
\str_if_eq_p:xx
\str_if_eq:xxTF
1483 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
1484 {
1485   \if_int_compare:w \pdfTeX_strcmp:D { \exp_not:n {#1} } { \exp_not:n {#2} }
1486   = \c_zero
1487   \prg_return_true: \else: \prg_return_false: \fi:
1488 }
1489 \prg_new_conditional:Npnn \str_if_eq:xx #1#2 { p , T , F , TF }
1490 {
1491   \if_int_compare:w \pdfTeX_strcmp:D {#1} {#2} = \c_zero
1492   \prg_return_true: \else: \prg_return_false: \fi:
1493 }
```

(End definition for `\str_if_eq:nn`. These functions are documented on page 24.)

164.19 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```
1494 \cs_new_eq:NN \cs_gnew_nopar:Npn \cs_new_nopar:Npn
1495 \cs_new_eq:NN \cs_gnew:Npn \cs_new:Npn
1496 \cs_new_eq:NN \cs_gnew_protected_nopar:Npn \cs_new_protected_nopar:Npn
1497 \cs_new_eq:NN \cs_gnew_protected:Npn \cs_new_protected:Npn
1498 \cs_new_eq:NN \cs_gnew_nopar:Npx \cs_new_nopar:Npx
1499 \cs_new_eq:NN \cs_gnew:Npx \cs_new:Npx
1500 \cs_new_eq:NN \cs_gnew_protected_nopar:Npx \cs_new_protected_nopar:Npx
1501 \cs_new_eq:NN \cs_gnew_protected:Npx \cs_new_protected:Npx
1502 \cs_new_eq:NN \cs_gnew_nopar:cpn \cs_new_nopar:cpn
1503 \cs_new_eq:NN \cs_gnew:cpn \cs_new:cpn
1504 \cs_new_eq:NN \cs_gnew_protected_nopar:cpn \cs_new_protected_nopar:cpn
1505 \cs_new_eq:NN \cs_gnew_protected:cpn \cs_new_protected:cpn
1506 \cs_new_eq:NN \cs_gnew_nopar:cpx \cs_new_nopar:cpx
1507 \cs_new_eq:NN \cs_gnew:cpx \cs_new:cpx
1508 \cs_new_eq:NN \cs_gnew_protected_nopar:cpx \cs_new_protected_nopar:cpx
1509 \cs_new_eq:NN \cs_gnew_protected:cpx \cs_new_protected:cpx
```

```

1510 \cs_new_eq:NN \cs_gnew_eq:NN \cs_new_eq:NN
1511 \cs_new_eq:NN \cs_gnew_eq:cN \cs_new_eq:cN
1512 \cs_new_eq:NN \cs_gnew_eq:Nc \cs_new_eq:Nc
1513 \cs_new_eq:NN \cs_gnew_eq:cc \cs_new_eq:cc

1514 \cs_new_eq:NN \cs_gundefine:N \cs_undefine:N
1515 \cs_new_eq:NN \cs_gundefine:c \cs_undefine:c

1516 \cs_new_eq:NN \group_execute_after:N \group_insert_after:N

1517 </initex | package>

```

165 l3expan implementation

```

1518 <*initex | package>

```

We start by ensuring that the required packages are loaded.

```

1519 <*package>
1520 \ProvidesExplPackage
1521   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
1522 \package_check_loaded_expl:
1523 </package>

```

`\exp_after:wN` These are defined in l3basics.

`\exp_not:N` *(End definition for `\exp_after:wN`. This function is documented on page 35.)*
`\exp_not:n`

165.1 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.⁴)

The definition of expansion functions with this technique happens in section 165.3. In section 165.2 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.

```

1524 \cs_new_nopar:Npn \l_exp_tl { }

```

⁴However, 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.

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.

```
1525 \cs_new:Npn \exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
1526 \cs_new:Npn \exp_arg_next_nobrace:nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End definition for `\exp_arg_next:nnn`.)

\::: The end marker is just another name for the identity function.

```
1527 \cs_new:Npn \::: #1 {#1}
```

(End definition for `\:::`. This function is documented on page 36.)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
1528 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(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.

```
1529 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for `\::N`. This function is documented on page ??.)

\::c This function is used to skip an argument that is turned into as control sequence without expansion.

```
1530 \cs_new:Npn \::c #1 \::: #2#3
1531 { \exp_after:wN \exp_arg_next_nobrace:nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for `\::c`. This function is documented on page ??.)

\::o This function is used to expand an argument once.

```
1532 \cs_new:Npn \::o #1 \::: #2#3
1533 { \exp_after:wN \exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(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. 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 T_EX 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 T_EX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

1534 \cs_new:Npn \::f #1 \::: #2#3
1535 {
1536   \exp_after:wN \exp_arg_next:nnn
1537   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1538   {#1} {#2}
1539 }
1540 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for `\::f`. This function is documented on page 35.)

`\::x` This function is used to expand an argument fully.

```

1541 \cs_new_protected:Npn \::x #1 \::: #2#3
1542 {
1543   \cs_set_nopar:Npx \l_exp_tl { {#3} }
1544   \exp_after:wN \exp_arg_next:nnn \l_exp_tl {#1} {#2}
1545 }
```

(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` and `muskip`. The V version expects a single token whereas `v` like `c` creates a `cname` from its argument given in braces and then evaluates it as if it was a V. The primitive `\tex_romannumeral:D` sets off an expansion similar to an `f` type expansion, which we will terminate using `\c_zero`. The argument is returned in braces.

```

1546 \cs_new:Npn \::V #1 \::: #2#3
1547 {
1548   \exp_after:wN \exp_arg_next:nnn
1549   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
1550   {#1} {#2}
1551 }
1552 \cs_new:Npn \::v # 1\::: #2#3
1553 {
```

```

1554 \exp_after:wN \exp_arg_next:nnn
1555 \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#3} }
1556 {#1} {#2}
1557 }

```

(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 **\exp_eval_register:c** 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 `\scan_stop:.`

```

1558 \cs_new_nopar:Npn \exp_eval_register:N #1
1559 {
1560 \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:.` In that case we throw an error. We could let 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.

```

1561 \if_meaning:w \scan_stop: #1
1562 \exp_eval_error_msg:w
1563 \fi:

```

The next bit requires some explanation. The function must be initiated by the primitive `\tex_romannumeral:D` and we want to terminate this expansion chain by inserting the `\c_zero` integer constant. 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 \c_zero \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \c_zero #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

1564 \else:
1565 \exp_after:wN \use_i_ii:nnn
1566 \fi:
1567 \exp_after:wN \c_zero \tex_the:D #1
1568 }
1569 \cs_new_nopar:Npn \exp_eval_register:c #1
1570 { \exp_after:wN \exp_eval_register:N \cs:w #1 \cs_end: }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
\exp_eval_error_msg:w ...erroneous variable used!

1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
1571 \group_begin:%
1572 \tex_catcode:D'\!=11\scan_stop:%
1573 \tex_catcode:D'\ =11\scan_stop:%
1574 \cs_new:Npn\exp_eval_error_msg:w#1\tex_the:D#2{%
1575 \fi:\fi:\exp_after:wN\c_zero\erroneous variable used!}%
1576 \group_end:%
```

(End definition for `\exp_eval_register:N` and `\exp_eval_register:c`. These functions are documented on page 36.)

165.2 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.

`\exp_args:No` Those lovely runs of expansion!

`\exp_args:NNo`

`\exp_args:NNNo`

```
1577 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
1578 \cs_new:Npn \exp_args:NNo #1#2#3
1579 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
1580 \cs_new:Npn \exp_args:NNNo #1#2#3#4
1581 { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(End definition for `\exp_args:No`. This function is documented on page 33.)

`\exp_args:Nc` In l3basics

(End definition for `\exp_args:Nc`. This function is documented on page 30.)

`\exp_args:cc` Here are the functions that turn their argument into csnames but are expandable.

`\exp_args:NNc`

`\exp_args:Ncc`

`\exp_args:Nccc`

```
1582 \cs_new:Npn \exp_args:cc #1#2
1583 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
1584 \cs_new:Npn \exp_args:NNc #1#2#3
1585 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
1586 \cs_new:Npn \exp_args:Ncc #1#2#3
1587 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
1588 \cs_new:Npn \exp_args:Nccc #1#2#3#4
1589 {
1590 \exp_after:wN #1
```

```

1591     \cs:w #2 \exp_after:wN \cs_end:
1592     \cs:w #3 \exp_after:wN \cs_end:
1593     \cs:w #4 \cs_end:
1594 }

```

(End definition for `\exp_args:cc` and others. These functions are documented on page 33.)

`\exp_args:Nf`
`\exp_args:Nv`
`\exp_args:Nf`
`\exp_args:Nx`

```

1595 \cs_new:Npn \exp_args:Nf #1#2
1596 { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -'0 #2 } }
1597 \cs_new:Npn \exp_args:Nv #1#2
1598 {
1599     \exp_after:wN #1 \exp_after:wN
1600     { \tex_romannumeral:D \exp_eval_register:c {#2} }
1601 }
1602 \cs_new:Npn \exp_args:NV #1#2
1603 {
1604     \exp_after:wN #1 \exp_after:wN
1605     { \tex_romannumeral:D \exp_eval_register:N #2 }
1606 }

```

(End definition for `\exp_args:Nf` and others. These functions are documented on page 31.)

`\exp_args:NNV`
`\exp_args:NNv`
`\exp_args:NNf`
`\exp_args:NNV`
`\exp_args:Ncf`
`\exp_args:Nco`

Some more hand-tuned function with three arguments. If we force that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

1607 \cs_new:Npn \exp_args:NNf #1#2#3
1608 {
1609     \exp_after:wN #1
1610     \exp_after:wN #2
1611     \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1612 }
1613 \cs_new:Npn \exp_args:NNv #1#2#3
1614 {
1615     \exp_after:wN #1
1616     \exp_after:wN #2
1617     \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#3} }
1618 }
1619 \cs_new:Npn \exp_args:NNV #1#2#3
1620 {
1621     \exp_after:wN #1
1622     \exp_after:wN #2
1623     \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
1624 }
1625 \cs_new:Npn \exp_args:Nco #1#2#3
1626 {
1627     \exp_after:wN #1
1628     \cs:w #2 \exp_after:wN \cs_end:

```

```

1629     \exp_after:wN {#3}
1630   }
1631   \cs_new:Npn \exp_args:Ncf #1#2#3
1632   {
1633     \exp_after:wN #1
1634     \cs:w #2 \exp_after:wN \cs_end:
1635     \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1636   }
1637   \cs_new_nopar:Npn \exp_args:NVV #1#2#3
1638   {
1639     \exp_after:wN #1
1640     \exp_after:wN { \tex_romannumeral:D \exp_after:wN
1641       \exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
1642     \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
1643   }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 32.)

`\exp_args:Ncco` A few more that we can hand-tune.

```

\exp_args:NcNc
\exp_args:NcNo
\exp_args:NNNV
1644   \cs_new:Npn \exp_args:NNNV #1#2#3#4
1645   {
1646     \exp_after:wN #1
1647     \exp_after:wN #2
1648     \exp_after:wN #3
1649     \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #4 }
1650   }
1651   \cs_new:Npn \exp_args:NcNc #1#2#3#4
1652   {
1653     \exp_after:wN #1
1654     \cs:w #2 \exp_after:wN \cs_end:
1655     \exp_after:wN #3
1656     \cs:w #4 \cs_end:
1657   }
1658   \cs_new:Npn \exp_args:NcNo #1#2#3#4
1659   {
1660     \exp_after:wN #1
1661     \cs:w #2 \exp_after:wN \cs_end:
1662     \exp_after:wN #3
1663     \exp_after:wN {#4}
1664   }
1665   \cs_new:Npn \exp_args:Ncco #1#2#3#4
1666   {
1667     \exp_after:wN #1
1668     \cs:w #2 \exp_after:wN \cs_end:
1669     \cs:w #3 \exp_after:wN \cs_end:
1670     \exp_after:wN {#4}
1671   }

```

(End definition for `\exp_args:Ncco` and others. These functions are documented on page 33.)

165.3 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions are all not long: they don't actually take any arguments themselves.

`\exp_args:Nx`

```
1672 \cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }
```

(End definition for `\exp_args:Nx`. This function is documented on page 31.)

`\exp_args:NNx`

`\exp_args:Nnc`

`\exp_args:Ncx`

`\exp_args:Nfo`

`\exp_args:Nff`

`\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.

```
1673 \cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
1674 \cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
1675 \cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
1676 \cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
1677 \cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
1678 \cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
1679 \cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
1680 \cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
1681 \cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
1682 \cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
1683 \cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
1684 \cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
1685 \cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
1686 \cs_new_protected_nopar:Npn \exp_args:Nxx { \::x \::x \::: }
```

(End definition for `\exp_args:NNx` and others. These functions are documented on page 32.)

`\exp_args:Nccx`

`\exp_args:Ncnx`

`\exp_args:NNno`

`\exp_args:Nnno`

`\exp_args:Nnnx`

`\exp_args:Nnox`

`\exp_args:Nooo`

`\exp_args:Nnox`

`\exp_args:Nnnc`

`\exp_args:NNnx`

`\exp_args:NNoo`

`\exp_args:NNox`

```
1687 \cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
1688 \cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
1689 \cs_new_nopar:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
1690 \cs_new_nopar:Npn \exp_args:Nnno { \::n \::n \::o \::: }
1691 \cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
1692 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
1693 \cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
1694 \cs_new_protected_nopar:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
1695 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
1696 \cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
1697 \cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
1698 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::o \::o \::x \::: }
```

(End definition for `\exp_args:Nccx` and others. These functions are documented on page 33.)

165.4 Last-unbraced versions

`\exp_arg_last_unbraced:nn`
`\::f_unbraced`
`\::o_unbraced`
`\::V_unbraced`
`\::v_unbraced`

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

1699 \cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
1700 \cs_new:Npn \::f_unbraced \::: #1#2
1701 {
1702   \exp_after:wN \exp_arg_last_unbraced:nn
1703   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1704 }
1705 \cs_new:Npn \::o_unbraced \::: #1#2
1706 { \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
1707 \cs_new:Npn \::V_unbraced \::: #1#2
1708 {
1709   \exp_after:wN \exp_arg_last_unbraced:nn
1710   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #2 } {#1}
1711 }
1712 \cs_new:Npn \::v_unbraced \::: #1#2
1713 {
1714   \exp_after:wN \exp_arg_last_unbraced:nn
1715   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#2} } {#1}
1716 }

```

(End definition for `\exp_arg_last_unbraced:nn`.)

`\exp_last_unbraced:NV`
`\exp_last_unbraced:Nv`
`\exp_last_unbraced:Nf`
`\exp_last_unbraced:No`
`\exp_last_unbraced:NcV`
`\exp_last_unbraced:NNV`
`\exp_last_unbraced:NNo`
`\exp_last_unbraced:Noo`
`\exp_last_unbraced:Nfo`
`\exp_last_unbraced:NNNV`
`\exp_last_unbraced:NNNo`

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

1717 \cs_new:Npn \exp_last_unbraced:NV #1#2
1718 { \exp_after:wN #1 \tex_romannumeral:D \exp_eval_register:N #2 }
1719 \cs_new:Npn \exp_last_unbraced:Nv #1#2
1720 { \exp_after:wN #1 \tex_romannumeral:D \exp_eval_register:c {#2} }
1721 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
1722 \cs_new:Npn \exp_last_unbraced:Nf #1#2
1723 { \exp_after:wN #1 \tex_romannumeral:D -'0 #2 }
1724 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
1725 {
1726   \exp_after:wN #1
1727   \cs:w #2 \exp_after:wN \cs_end:
1728   \tex_romannumeral:D \exp_eval_register:N #3
1729 }
1730 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
1731 {
1732   \exp_after:wN #1
1733   \exp_after:wN #2
1734   \tex_romannumeral:D \exp_eval_register:N #3
1735 }
1736 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
1737 { \exp_after:wN #1 \exp_after:wN #2 #3 }

```



```

1738 \cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
1739 \cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
1740 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
1741 {
1742   \exp_after:wN #1
1743   \exp_after:wN #2
1744   \exp_after:wN #3
1745   \tex_romannumeral:D \exp_eval_register:N #4
1746 }
1747 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
1748 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }

```

(End definition for `\exp_last_unbraced:NV`. This function is documented on page 34.)

`\exp_last_two_unbraced:Noo` If #2 is a single token then this can be implemented as

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

1749 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
1750 { \exp_after:wN \exp_last_two_unbraced_aux:nnN \exp_after:wN {#3} {#2} #1 }
1751 \cs_new:Npn \exp_last_two_unbraced_aux:nnN #1#2#3
1752 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo`. This function is documented on page 34.)

165.5 Preventing expansion

```

\exp_not:o
\exp_not:f
\exp_not:V
\exp_not:v
1753 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
1754 \cs_new:Npn \exp_not:f #1
1755 { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -'0 #1 } }
1756 \cs_new:Npn \exp_not:V #1
1757 {
1758   \etex_unexpanded:D \exp_after:wN
1759   { \tex_romannumeral:D \exp_eval_register:N #1 }
1760 }
1761 \cs_new:Npn \exp_not:v #1
1762 {
1763   \etex_unexpanded:D \exp_after:wN
1764   { \tex_romannumeral:D \exp_eval_register:c {#1} }
1765 }

```

(End definition for `\exp_not:o`. This function is documented on page 35.)

`\exp_not:c` A helper function.

```
1766 \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 35.)

165.6 Defining function variants

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`
`\cs_generate_variant_aux:nnNNn` #2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`
`\cs_generate_variant_aux:Nnnw`

`\cs_generate_variant_aux:NNn`
`\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`.

```
1767 \cs_new_protected:Npn \cs_generate_variant:Nn #1
1768 {
1769   \chk_if_exist_cs:N #1
1770   \cs_split_function:NN #1 \cs_generate_variant_aux:nnNNn
1771   #1
1772 }
```

We discard the boolean #3 and then set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```
1773 \cs_new:Npn \cs_generate_variant_aux:nnNNn #1#2#3#4#5
1774 { \cs_generate_variant_aux:Nnnw #4 {#1}{#2} #5 , ? , \q_recursion_stop }
```

Next is the real work to be done. We now have 1: original function, 2: base name, 3: base signature, 4: 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. Then build the variant function once, to avoid repeating this relatively expensive operation. Then recurse.

```
1775 \cs_new:Npn \cs_generate_variant_aux:Nnnw #1#2#3#4 ,
1776 {
1777   \if:w ? #4
1778   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1779   \fi:
1780   \exp_args:NNc \cs_generate_variant_aux:NNn
1781   #1
```

```

1782     { #2 : #4 \use:c { use_none: \cs_generate_variant_aux:N #4 ? } #3 }
1783     {#4}
1784     \cs_generate_variant_aux:Nnnw #1 {#2} {#3}
1785 }

```

Check if the variant form has already been defined. If not, then define it and then additionally check if the `\exp_args:N` form needed is defined. Otherwise tell that it was already defined.

```

1786 \cs_new:Npn \cs_generate_variant_aux:NNn #1 #2 #3
1787 {
1788   \cs_if_free:NTF #2
1789   {
1790     \cs_generate_variant_aux:NNpx #1 #2
1791     { \exp_not:c { exp_args:N #3 } \exp_not:N #1 }
1792     \cs_generate_internal_variant:n {#3}
1793   }
1794   {
1795     \iow_log:x
1796     {
1797       Variant~\token_to_str:N #2~%
1798       already~defined;~ not~ changing~ it~on~line~%
1799       \tex_the:D \tex_inputlineno:D
1800     }
1801   }
1802 }

```

The small loop for defining the required number of ns. Break when seeing a ?.

```

1803 \cs_new:Npn \cs_generate_variant_aux:N #1
1804 {
1805   \if:w ? #1
1806   \exp_after:wN \use_none:nn
1807   \fi:
1808   n
1809   \cs_generate_variant_aux:N
1810 }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 28.)

`\cs_generate_variant_aux:NNpx`
`\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.

```

1811 \group_begin:
1812 \tex_lccode:D ‘Z = ‘\d \scan_stop:
1813 \tex_lccode:D ‘? = ‘\ \scan_stop:
1814 \tex_catcode:D ‘P = 12 \scan_stop:
1815 \tex_catcode:D ‘R = 12 \scan_stop:

```

```

1816 \tex_catcode:D '\0 = 12 \scan_stop:
1817 \tex_catcode:D '\T = 12 \scan_stop:
1818 \tex_catcode:D '\E = 12 \scan_stop:
1819 \tex_catcode:D '\C = 12 \scan_stop:
1820 \tex_catcode:D '\Z = 12 \scan_stop:
1821 \tex_lowercase:D
1822 {
1823   \group_end:
1824   \cs_new_nopar:Npn \cs_generate_variant_aux:NNpx #1
1825   {
1826     \exp_after:wN \cs_generate_variant_aux:w
1827     \token_to_meaning:N #1 ? PROTECTEZ \q_stop
1828   }
1829   \cs_new:Npn \cs_generate_variant_aux:w #1 ? PROTECTEZ #2 \q_stop
1830   {
1831     \if_catcode:w a \etex_detokenize:D \exp_after:wN {#1} a
1832     \exp_after:wN \cs_new_protected_nopar:Npx
1833     \else:
1834     \exp_after:wN \cs_new_nopar:Npx
1835     \fi:
1836   }
1837 }

```

(End definition for `\cs_generate_variant_aux:NNpx`.)

`\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`

`\cs_generate_internal_variant_aux:N`

```

1838 \cs_new_protected:Npn \cs_generate_internal_variant:n #1
1839 {
1840   \cs_if_free:cT { \exp_args:N #1 }
1841   {
1842     \cs_new:cpx { \exp_args:N #1 }
1843     { \cs_generate_internal_variant_aux:N #1 : }
1844   }
1845 }

```

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.

```

1846 \cs_new:Npn \cs_generate_internal_variant_aux:N #1
1847 {
1848   \exp_not:c { \:: #1 }
1849   \if_meaning:w #1 :
1850   \exp_after:wN \use_none:n
1851   \fi:
1852   \cs_generate_internal_variant_aux:N
1853 }

```

(End definition for `\cs_generate_internal_variant:n`. This function is documented on page 36.)

165.7 Variants which cannot be created earlier

These cannot come earlier as they need `\cs_generate_variant:Nn`.

```

\str_if_eq_p:Vn
\str_if_eq_p:on
\str_if_eq_p:nV
\str_if_eq_p:no
\str_if_eq_p:VV
\str_if_eq:VnTF
\str_if_eq:onTF
\str_if_eq:nVTF
\str_if_eq:noTF
\str_if_eq:VVTF

```

(End definition for `\str_if_eq:Vn` and others. These functions are documented on page 24.)

```
1862 </initex | package>
```

166 l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

```

1863 <*initex | package>

1864 <*package>
1865 \ProvidesExplPackage
1866   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
1867 \package_check_loaded_expl:
1868 </package>

```

166.1 Defining a set of conditional functions

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_set_conditional:Npnn` and others. These functions are documented on page 39.)

```

\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
  \prg_return_true:
  \prg_return_false:

```

166.2 The boolean data type

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```

1869 \cs_new_protected_nopar:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
1870 \cs_generate_variant:Nn \bool_new:N { c }

```

(End definition for `\bool_new:N` and `\bool_new:c`. These functions are documented on page 40.)

`\bool_set_true:N` Setting is already pretty easy.
`\bool_set_true:c`
`\bool_gset_true:N`
`\bool_gset_true:c`
`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

```

1871 \cs_new_protected_nopar:Npn \bool_set_true:N #1
1872 { \cs_set_eq:NN #1 \c_true_bool }
1873 \cs_new_protected_nopar:Npn \bool_set_false:N #1
1874 { \cs_set_eq:NN #1 \c_false_bool }
1875 \cs_new_protected_nopar:Npn \bool_gset_true:N #1
1876 { \cs_gset_eq:NN #1 \c_true_bool }
1877 \cs_new_protected_nopar:Npn \bool_gset_false:N #1
1878 { \cs_gset_eq:NN #1 \c_false_bool }
1879 \cs_generate_variant:Nn \bool_set_true:N { c }
1880 \cs_generate_variant:Nn \bool_set_false:N { c }
1881 \cs_generate_variant:Nn \bool_gset_true:N { c }
1882 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 40.)

`\bool_set_eq:NN` The usual copy code.
`\bool_set_eq:cN`
`\bool_set_eq:Nc`
`\bool_set_eq:cc`
`\bool_gset_eq:NN`
`\bool_gset_eq:cN`
`\bool_gset_eq:Nc`
`\bool_gset_eq:cc`

```

1883 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
1884 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
1885 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
1886 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
1887 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
1888 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
1889 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
1890 \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 41.)

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning
`\bool_set:cn` `\c_true_bool` or `\c_false_bool`.
`\bool_gset:Nn`
`\bool_gset:cn`

```

1891 \cs_new:Npn \bool_set:Nn #1#2
1892 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
1893 \cs_new:Npn \bool_gset:Nn #1#2
1894 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
1895 \cs_generate_variant:Nn \bool_set:Nn { c }
1896 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

`\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 \overline{TF}`
`\bool_if:c \overline{TF}`

```

1897 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
1898 {
1899   \if_bool:N #1
1900     \prg_return_true:
1901   \else:
1902     \prg_return_false:
1903   \fi:

```

```

1904     }
1905     \cs_generate_variant:Nn \bool_if_p:N { c }
1906     \cs_generate_variant:Nn \bool_if:NT { c }
1907     \cs_generate_variant:Nn \bool_if:NF { c }
1908     \cs_generate_variant:Nn \bool_if:NTF { c }

```

(End definition for `\bool_set:Nn` and `\bool_set:cn`. These functions are documented on page 41.)

`\l_tmpa_bool` A few booleans just if you need them.

`\g_tmpa_bool`

```

1909 \bool_new:N \l_tmpa_bool
1910 \bool_new:N \g_tmpa_bool

```

166.3 Boolean expressions

`\bool_if_p:n`

`\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`

`\bool_S_0:w`

`\bool_S_1:w`

`\bool_eval_skip_to_end:Nw`

`\bool_eval_skip_to_end_aux:Nw`

`\bool_eval_skip_to_end_aux_ii:Nw`

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.

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:

<true>And Current truth value is true, logical And seen, continue with `GetNext` to examine truth value of next boolean (sub-)expression.

<false>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 **<false>**.

<true>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 **<true>**.

<false>Or Current truth value is false, logical Or seen, continue with `GetNext` to examine truth value of next boolean (sub-)expression.

<true>Close Current truth value is true, Close seen, return **<true>**.

$\langle false \rangle$ 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 `\int_value:w: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 \TeX . 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.

```

1911 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
1912 {
1913   \if_predicate:w \bool_if_p:n {#1}
1914   \prg_return_true:
1915   \else:
1916     \prg_return_false:
1917   \fi:
1918 }
1919 \cs_new:Npn \bool_if_p:n #1
1920 {
1921   \group_align_safe_begin:
1922   \bool_get_next:N ( #1 ) S
1923 }
```

The GetNext operation. We make it a switch: If not a `!` or `(`, we assume it is a predicate.

```

1924 \cs_new:Npn \bool_get_next:N #1
1925 {
1926   \use:c
1927   {
1928     bool_
1929     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
1930     :w
1931   }
1932   #1
1933 }
```

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.

```

1934 \cs_new:Npn \bool_get_not_next:N #1
1935 {
```



```

1936     \use:c
1937     {
1938         bool_not_
1939         \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
1940         :w
1941     }
1942     #1
1943 }

```

We need these later on to nullify the unity operation !!.

```

1944 \cs_new:Npn \bool_get_next:NN #1#2 { \bool_get_next:N #2 }
1945 \cs_new:Npn \bool_get_not_next:NN #1#2 { \bool_get_not_next:N #2 }

```

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.

```

1946 \cs_new:cpn { bool_!:w } #1#2
1947 {
1948     \if_meaning:w ( #2
1949     \exp_after:wN \bool_Not:w
1950     \else:
1951     \if_meaning:w ! #2
1952     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_next:NN
1953     \else:
1954     \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
1955     \fi:
1956     \fi:
1957     #2
1958 }

```

Variant called when already inside a Not. Essentially the opposite of the above.

```

1959 \cs_new:cpn { bool_not_!:w } #1#2
1960 {
1961     \if_meaning:w ( #2
1962     \exp_after:wN \bool_not_Not:w
1963     \else:
1964     \if_meaning:w ! #2
1965     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_not_next:NN
1966     \else:
1967     \exp_after:wN \exp_after:wN \exp_after:wN \bool_not_Not:N
1968     \fi:
1969     \fi:
1970     #2
1971 }

```

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.

```

1972 \cs_new:Npn \bool_Not:w { \exp_after:wN \int_value:w \bool_get_not_next:N }
1973 \cs_new:Npn \bool_not_Not:w { \exp_after:wN \int_value:w \bool_get_next:N }

```

These occur when processing !<bool> and can be evaluated directly.

```

1974 \cs_new:Npn \bool_Not:N #1
1975 {
1976   \exp_after:wN \bool_p:w
1977   \if_meaning:w #1 \c_true_bool
1978   \c_false_bool
1979   \else:
1980   \c_true_bool
1981   \fi:
1982 }
1983 \cs_new:Npn \bool_not_Not:N #1
1984 {
1985   \exp_after:wN \bool_p:w
1986   \if_meaning:w #1 \c_true_bool
1987   \c_true_bool
1988   \else:
1989   \c_false_bool
1990   \fi:
1991 }

```

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 !(...).

```

1992 \cs_new:cpn { bool_( :w } #1
1993 { \exp_after:wN \bool_cleanup:N \int_value:w \bool_get_next:N }
1994 \cs_new:cpn { bool_not_( :w } #1
1995 { \exp_after:wN \bool_not_cleanup:N \int_value:w \bool_get_next:N }

```

Otherwise just evaluate the predicate and look for And, Or or Close afterwards.

```

1996 \cs_new:cpn { bool_p:w } { \exp_after:wN \bool_cleanup:N \int_value:w }
1997 \cs_new:cpn { bool_not_p:w } { \exp_after:wN \bool_not_cleanup:N \int_value:w }

```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```

1998 \cs_new:Npn \bool_cleanup:N #1
1999 {
2000   \exp_after:wN \bool_choose:NN \exp_after:wN #1
2001   \int_to_roman:w - '\q
2002 }
2003 \cs_new:Npn \bool_not_cleanup:N #1
2004 {
2005   \exp_after:wN \bool_not_choose:NN \exp_after:wN #1
2006   \int_to_roman:w - '\q
2007 }

```

Branching the six way switch. Reversals should be reasonably straightforward.

```
2008 \cs_new_nopar:Npn \bool_choose:NN #1#2 { \use:c { bool_ #2 _ #1 :w } }
2009 \cs_new_nopar:Npn \bool_not_choose:NN #1#2 { \use:c { bool_not_ #2 _ #1 :w } }
```

Continues scanning. Must remove the second & or |.

```
2010 \cs_new_nopar:cpn { bool_&_1:w } & { \bool_get_next:N }
2011 \cs_new_nopar:cpn { bool_|_0:w } | { \bool_get_next:N }
2012 \cs_new_nopar:cpn { bool_not_&_0:w } & { \bool_get_next:N }
2013 \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.

```
2014 \cs_new_nopar:cpn { bool_)_0:w } { \c_false_bool }
2015 \cs_new_nopar:cpn { bool_)_1:w } { \c_true_bool }
2016 \cs_new_nopar:cpn { bool_not_)_0:w } { \c_true_bool }
2017 \cs_new_nopar:cpn { bool_not_)_1:w } { \c_false_bool }
2018 \cs_new_nopar:cpn { bool_S_0:w } { \group_align_safe_end: \c_false_bool }
2019 \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.

```
2020 \cs_new_nopar:cpn { bool_&_0:w } & { \bool_eval_skip_to_end:Nw \c_false_bool }
2021 \cs_new_nopar:cpn { bool_|_1:w } | { \bool_eval_skip_to_end:Nw \c_true_bool }
2022 \cs_new_nopar:cpn { bool_not_&_1:w } &
2023 { \bool_eval_skip_to_end:Nw \c_false_bool }
2024 \cs_new_nopar:cpn { bool_not_|_0:w } |
2025 { \bool_eval_skip_to_end:Nw \c_true_bool }
```

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.

```
2026 %% (
2027 \cs_new:Npn \bool_eval_skip_to_end:Nw #1#2 )
2028 {
2029   \bool_eval_skip_to_end_aux:Nw #1#2 ( % )
2030   \q_no_value \q_stop
2031   {#2}
2032 }
```

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

```
2033 \cs_new:Npn \bool_eval_skip_to_end_aux:Nw #1#2 ( #3#4 \q_stop #5 % )
2034 {
2035   \quark_if_no_value:NTF #3
2036   {#1}
2037   { \bool_eval_skip_to_end_aux_ii:Nw #1 #5 }
2038 }
```

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!

```
2039 \cs_new:Npn \bool_eval_skip_to_end_aux_ii:Nw #1#2 ( #3 )
2040 { % (
2041   \bool_eval_skip_to_end:Nw #1#3 )
2042 }
```

\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.

```
2043 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2044 \cs_new:Npn \bool_xor_p:nn #1#2
2045 {
2046   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2047     \c_false_bool
2048     \c_true_bool
2049 }
```

166.4 Logical loops

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

\bool_while_do:cn

\bool_until_do:Nn

\bool_until_do:cn

```
2050 \cs_new:Npn \bool_while_do:Nn #1#2
2051 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2052 \cs_new:Npn \bool_until_do:Nn #1#2
2053 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2054 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2055 \cs_generate_variant:Nn \bool_until_do:Nn { c }
```

\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

```
2056 \cs_new:Npn \bool_do_while:Nn #1#2
2057 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2058 \cs_new:Npn \bool_do_until:Nn #1#2
2059 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2060 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2061 \cs_generate_variant:Nn \bool_do_until:Nn { c }
```

`\bool_while_do:nn` Loop functions with the test either before or after the first body expansion.

`\bool_do_while:nn`
`\bool_until_do:nn`
`\bool_do_until:nn`

```

2062 \cs_new:Npn \bool_while_do:nn #1#2
2063 {
2064   \bool_if:nT {#1}
2065   {
2066     #2
2067     \bool_while_do:nn {#1} {#2}
2068   }
2069 }
2070 \cs_new:Npn \bool_do_while:nn #1#2
2071 {
2072   #2
2073   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2074 }
2075 \cs_new:Npn \bool_until_do:nn #1#2
2076 {
2077   \bool_if:nF {#1}
2078   {
2079     #2
2080     \bool_until_do:nn {#1} {#2}
2081   }
2082 }
2083 \cs_new:Npn \bool_do_until:nn #1#2
2084 {
2085   #2
2086   \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
2087 }

```

166.5 Switching by case

A family of functions to select one case of a number: the same ideas are used for a number of different situations.

`\prg_case_end:nw` In all cases the end statement is the same. Here, `#1` will be the code needed, `#2` the other cases to throw away, including the “else” case.

```

2088 \cs_new_eq:NN \prg_case_end:nw \use_i_delimit_by_q_recursion_stop:nw

```

`\prg_case_int:nnn` For integer cases, the first task to fully expand the check condition. After that, a loop is started to compare each possible value and stop if the test is true. The tested value is put at the end to ensure that there is necessarily a match, which will fire the “else” pathway.

`\prg_case_int_aux:nnn`
`\prg_case_int_aux:nw`

```

2089 \cs_new:Npn \prg_case_int:nnn #1
2090 { \exp_args:Nf \prg_case_int_aux:nnn { \int_eval:n {#1} } }
2091 \cs_new:Npn \prg_case_int_aux:nnn #1 #2 #3
2092 { \prg_case_int_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }

```

```

2093 \cs_new:Npn \prg_case_int_aux:nw #1#2#3
2094 {
2095   \int_compare:nNnTF {#1} = {#2}
2096   { \prg_case_end:nw {#3} }
2097   { \prg_case_int_aux:nw {#1} }
2098 }

```

The dimension function is the same, just a change of calculation method.

\prg_case_dim:nnn
\prg_case_dim_aux:nnn
\prg_case_dim_aux:nw

```

2099 \cs_new:Npn \prg_case_dim:nnn #1
2100 { \exp_args:Nf \prg_case_dim_aux:nnn { \dim_eval:n {#1} } }
2101 \cs_new:Npn \prg_case_dim_aux:nnn #1 #2 #3
2102 { \prg_case_dim_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
2103 \cs_new:Npn \prg_case_dim_aux:nw #1#2#3
2104 {
2105   \dim_compare:nNnTF {#1} = {#2}
2106   { \prg_case_end:nw {#3} }
2107   { \prg_case_dim_aux:nw {#1} }
2108 }

```

No calculations for strings, otherwise no surprises.

\prg_case_str:nnn
\prg_case_str:onn
\prg_case_str:xxn
\prg_case_str_aux:nw
\prg_case_str_x_aux:nw

```

2109 \cs_new:Npn \prg_case_str:nnn #1#2#3
2110 { \prg_case_str_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
2111 \cs_new:Npn \prg_case_str_aux:nw #1#2#3
2112 {
2113   \str_if_eq:nnTF {#1} {#2}
2114   { \prg_case_end:nw {#3} }
2115   { \prg_case_str_aux:nw {#1} }
2116 }
2117 \cs_generate_variant:Nn \prg_case_str:nnn { o }
2118 \cs_new:Npn \prg_case_str:xxn #1#2#3
2119 { \prg_case_str_x_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
2120 \cs_new:Npn \prg_case_str_x_aux:nw #1#2#3
2121 {
2122   \str_if_eq:xxTF {#1} {#2}
2123   { \prg_case_end:nw {#3} }
2124   { \prg_case_str_aux:nw {#1} }
2125 }

```

Similar again, but this time with some variants.

\prg_case_tl:Nnn
\prg_case_tl:cn
\prg_case_tl_aux:Nw

```

2126 \cs_new:Npn \prg_case_tl:Nnn #1#2#3
2127 { \prg_case_tl_aux:Nw #1 #2 #1 {#3} \q_recursion_stop }
2128 \cs_new:Npn \prg_case_tl_aux:Nw #1#2#3
2129 {
2130   \tl_if_eq:NNTF #1 #2
2131   { \prg_case_end:nw {#3} }
2132   { \prg_case_tl_aux:Nw #1 }
2133 }
2134 \cs_generate_variant:Nn \prg_case_tl:Nnn { c }

```

166.6 Producing n copies

```
\prg_replicate:nn
```

This function uses a cascading cname technique by David Kastrup (who else :-)

```

\prg_replicate_aux:N
\prg_replicate_first_aux:N
\prg_replicate_
\prg_replicate_0:n
\prg_replicate_1:n
\prg_replicate_2:n
\prg_replicate_3:n
\prg_replicate_4:n
\prg_replicate_5:n
\prg_replicate_6:n
\prg_replicate_7:n
\prg_replicate_8:n
\prg_replicate_9:n
\prg_replicate_first_-:n
\prg_replicate_first_0:n
\prg_replicate_first_1:n
\prg_replicate_first_2:n
\prg_replicate_first_3:n
\prg_replicate_first_4:n
\prg_replicate_first_5:n
\prg_replicate_first_6:n
\prg_replicate_first_7:n
\prg_replicate_first_8:n
\prg_replicate_first_9:n

```

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 `cnames` which means that we start building several `cnames` so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with `25` then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\int_to_roman:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it 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, and if necessary, it is possible to write `\prg_replicate:nn{1000}{\prg_replicate:nn{1000}{\code}}`. An alternative approach is to create a string of m's with `\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.

```

2135 \cs_new_nopar:Npn \prg_replicate:nn #1
2136 {
2137   \int_to_roman:w
2138   \exp_after:wN \prg_replicate_first_aux:N
2139   \int_value:w \int_eval:w #1 \int_eval_end:
2140   \cs_end:
2141 }
2142 \cs_new_nopar:Npn \prg_replicate_aux:N #1
2143 { \cs:w prg_replicate_#1 :n \prg_replicate_aux:N }
2144 \cs_new_nopar:Npn \prg_replicate_first_aux:N #1
2145 { \cs:w prg_replicate_first_#1 :n \prg_replicate_aux:N }

```

Then comes all the functions that do the hard work of inserting all the copies.

```

2146 \cs_new_nopar:Npn \prg_replicate_ :n #1 { \cs_end: }
2147 \cs_new:cpn { prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
2148 \cs_new:cpn { prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
2149 \cs_new:cpn { prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
2150 \cs_new:cpn { prg_replicate_3:n } #1
2151   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
2152 \cs_new:cpn { prg_replicate_4:n } #1
2153   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
2154 \cs_new:cpn { prg_replicate_5:n } #1
2155   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }

```



```
2156 \cs_new:cpn { prg_replicate_6:n } #1  
2157   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }  
2158 \cs_new:cpn { prg_replicate_7:n } #1  
2159   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }  
2160 \cs_new:cpn { prg_replicate_8:n } #1  
2161   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }  
2162 \cs_new:cpn { prg_replicate_9:n } #1  
2163   { \cs_end: {#1#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
```

Users shouldn't ask for something to be replicated once or even not at all but...

```

2164 \cs_new:cpn { prg_replicate_first_~:~n } #1 { \c_zero \negative_replication }
2165 \cs_new:cpn { prg_replicate_first_0:n } #1 { \c_zero }
2166 \cs_new:cpn { prg_replicate_first_1:n } #1 { \c_zero #1 }
2167 \cs_new:cpn { prg_replicate_first_2:n } #1 { \c_zero #1#1 }
2168 \cs_new:cpn { prg_replicate_first_3:n } #1 { \c_zero #1#1#1 }
2169 \cs_new:cpn { prg_replicate_first_4:n } #1 { \c_zero #1#1#1#1 }
2170 \cs_new:cpn { prg_replicate_first_5:n } #1 { \c_zero #1#1#1#1#1 }
2171 \cs_new:cpn { prg_replicate_first_6:n } #1 { \c_zero #1#1#1#1#1#1 }
2172 \cs_new:cpn { prg_replicate_first_7:n } #1 { \c_zero #1#1#1#1#1#1#1 }
2173 \cs_new:cpn { prg_replicate_first_8:n } #1 { \c_zero #1#1#1#1#1#1#1#1 }
2174 \cs_new:cpn { prg_replicate_first_9:n } #1 { \c_zero #1#1#1#1#1#1#1#1#1 }

```

(End definition for `\bool_if:n`. These functions are documented on page 45.)

```
\prg_stepwise_function:nnnN
    \prg_stepwise_function_incr:nnnN
    \prg_stepwise_function_decr:nnnN
```

Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around.

```

2175 \cs_new:Npn \prg_stepwise_function:nnnN #1#2
2176 {
2177     \int_compare:nNnTF {#2} > { 0 }
2178     { \exp_args:Nf \prg_stepwise_function_incr:nnnN }
2179     { \exp_args:Nf \prg_stepwise_function_decr:nnnN }
2180     { \int_eval:n {#1} } {#2}
2181 }
2182 \cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4
2183 {
2184     \int_compare:nNnF {#1} > {#3}
2185     {
2186         #4 {#1}
2187         \exp_args:Nf \prg_stepwise_function_incr:nnnN
2188         { \int_eval:n { #1 + #2 } } {#2} {#3} #4
2189     }
2190 }
2191 \cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4
2192 {
2193     \int_compare:nNnF {#1} < {#3}
2194     {
2195         #4 {#1}
2196         \exp_args:Nf \prg_stepwise_function_decr:nnnN

```

```

2197         { \int_eval:n { #1 + #2 } } {#2} {#3} #4
2198     }
2199 }

```

(End definition for `\prg_stepwise_function:nnnN`. This function is documented on page 45.)

`\g_prg_stepwise_level_int` For nesting, the usual approach of using a counter.

```

2200 \int_new:N \g_prg_stepwise_level_int

```

(End definition for `\g_prg_stepwise_level_int`.)

`\prg_stepwise_inline:nnnn` The approach here is similar but with a global integer required to make the nesting safe (as seen in other in line functions).

```

\prg_stepwise_inline_incr:Nnnn
\prg_stepwise_inline_decr:Nnnn

```

```

2201 \cs_new_protected:Npn \prg_stepwise_inline:nnnn #1#2#3#4
2202 {
2203   \int_gincr:N \g_prg_stepwise_level_int
2204   \cs_gset_nopar:cpn
2205     { g_prg_stepwise_ \int_use:N \g_prg_stepwise_level_int :n }
2206     ##1 {#4}
2207   \int_compare:nNnTF {#2} > { 0 }
2208     { \exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
2209     { \exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
2210     { g_prg_stepwise_ \int_use:N \g_prg_stepwise_level_int :n }
2211     { \int_eval:n {#1} } {#2} {#3}
2212   \int_gdecr:N \g_prg_stepwise_level_int
2213 }
2214 \cs_new_protected:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4
2215 {
2216   \int_compare:nNnF {#2} > {#4}
2217   {
2218     #1 {#2}
2219     \exp_args:Nnf \prg_stepwise_inline_incr:Nnnn #1
2220       { \int_eval:n { #2 + #3 } } {#3} {#4}
2221   }
2222 }
2223 \cs_new_protected:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4
2224 {
2225   \int_compare:nNnF {#2} < {#4}
2226   {
2227     #1 {#2}
2228     \exp_args:Nnf \prg_stepwise_inline_decr:Nnnn #1
2229       { \int_eval:n { #2 + #3 } } {#3} {#4}
2230   }
2231 }

```

(End definition for `\prg_stepwise_inline:nnnn`. This function is documented on page 46.)

`\prg_stepwise_variable:nnnNn` A wrapper for the above.

```

2232 \cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2#3#4#5
2233 {
2234   \prg_stepwise_inline:nnnn {#1} {#2} {#3}
2235   {
2236     \tl_set:Nn #4 {##1}
2237     #5
2238   }
2239 }

```

(End definition for `\prg_stepwise_variable:nnnNn`. This function is documented on page ??.)

166.7 Detecting T_EX's mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as
`\mode_if_vertical:TF` 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.

```

2240 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2241 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:`. These functions are documented on page 47.)

`\mode_if_horizontal_p:` For testing horizontal mode.

`\mode_if_horizontal:TF`

```

2242 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2243 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:`. These functions are documented on page 46.)

`\mode_if_inner_p:` For testing inner mode.

`\mode_if_inner:TF`

```

2244 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2245 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:`. These functions are documented on page 46.)

`\mode_if_math_p:` For testing math mode: without `\` things go wrong in alignments.

`\mode_if_math:TF`

```

2246 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2247 {
2248   \scan_align_safe_stop:
2249   \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi:
2250 }

```

(End definition for `\mode_if_math:`. These functions are documented on page 47.)

166.8 Internal programming functions

`\group_align_safe_begin:` `\group_align_safe_end:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: 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*...

```
2251 \cs_new_nopar:Npn \group_align_safe_begin:
2252 { \if_false: { \fi: \if_int_compare:w ‘} = \c_zero \fi: }
2253 \cs_new_nopar:Npn \group_align_safe_end:
2254 { \if_int_compare:w ‘{ = \c_zero } \fi: }
```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:`. These functions are documented on page 47.)

`\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_ifmmode: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 letters⁵ 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 if an only if 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.

```
2255 \cs_new_nopar:Npn \scan_align_safe_stop:
2256 {
2257   \int_compare:nNnT \etex_currentgrouptype:D = \c_six
2258   {
2259     \int_compare:nNnF \etex_lastnodetype:D = \c_zero
2260     { \int_compare:nNnF \etex_lastnodetype:D = \c_seven { \scan_stop: } }
2261   }
2262 }
```

(End definition for `\scan_align_safe_stop:`. This function is documented on page 47.)

`\prg_variable_get_scope:N` `\prg_variable_get_scope_aux:w` `\prg_variable_get_type:N` `\prg_variable_get_type:w` 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.

⁵Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

```

2263 \group_begin:
2264 \tex_lccode:D '\& = '\g \scan_stop:
2265 \tex_catcode:D '\& = \c_twelve
2266 \tl_to_lowercase:n
2267 {
2268   \group_end:
2269   \cs_new_nopar:Npn \prg_variable_get_scope:N #1
2270   {
2271     \exp_last_unbraced:Nf \prg_variable_get_scope_aux:w
2272     { \cs_to_str:N #1 \exp_stop_f: \q_stop }
2273   }
2274   \cs_new_nopar:Npn \prg_variable_get_scope_aux:w #1#2 \q_stop
2275   { \token_if_eq_meaning:NNT & #1 { g } }
2276 }
2277 \group_begin:
2278 \tex_lccode:D '\& = '\_ \scan_stop:
2279 \tex_catcode:D '\& = \c_twelve
2280 \tl_to_lowercase:n
2281 {
2282   \group_end:
2283   \cs_new_nopar:Npn \prg_variable_get_type:N #1
2284   {
2285     \exp_after:wN \prg_variable_get_type_aux:w
2286     \token_to_str:N #1 & a \q_stop
2287   }
2288   \cs_new_nopar:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_stop
2289   {
2290     \token_if_eq_meaning:NNTF a #2
2291     {#1}
2292     { \prg_variable_get_type_aux:w #2#3 \q_stop }
2293   }
2294 }

```

(End definition for `\prg_variable_get_scope:N`. This function is documented on page 48.)

166.9 Experimental programmings functions

`\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.

```
2295 \cs_new_protected_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
2296   \cs_set:cpx{#1_quicksort:n}##1{
2297     \exp_not:c{#1_quicksort_start_partition:w} ##1
2298     \exp_not:n{#2\q_nil#3\q_stop}
2299   }
2300   \cs_set:cpx{#1_quicksort_braced:n}##1{
2301     \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
2302     \exp_not:N\q_nil\exp_not:N\q_stop
2303   }
2304   \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
2305     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2306     \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{-}{-}
2307   }
2308   \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
2309     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2310     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{-}{-}
2311   }
```

Now for doing the partitions.

```
2312 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2313   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2314   {
2315     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2316     \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2317     \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2318   }
2319   {##1}{##2}{##3}{##4}
2320 }
2321 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2322   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2323   {
2324     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2325     \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2326     \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2327   }
2328   {##1}{##2}{##3}{##4}
2329 }
2330 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2331   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2332   {
```

```

2333     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2334     \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2335     \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2336   }
2337   {##1}{##2}{##3}{##4}
2338 }
2339 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2340   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2341   {
2342     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2343     \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2344     \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2345   }
2346   {##1}{##2}{##3}{##4}
2347 }

```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```

2348 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2349   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}{##3}}
2350 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2351   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3}{##4}}
2352 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2353   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}{##2}{##3}}
2354 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2355   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##4}{##3}}
2356 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2357   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##4}{##3}}
2358 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2359   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3}{##4}}
2360 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2361   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}{##2}{##3}}
2362 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2363   \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.

```

2364 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2365   \exp_not:c{#1_quicksort_braced:n}{##2}
2366   \exp_not:c{#1_quicksort_function:n}{##1}
2367   \exp_not:c{#1_quicksort_braced:n}{##3}
2368 }
2369 }

```

(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.

```

2370 \prg_define_quicksort:nnn {prg}{-}{-}

```

(End definition for `\prg_quicksort:n`. This function is documented on page 48.)

`\prg_quicksort_function:n`
`\prg_quicksort_compare:nnTF`

```
2371 \cs_set:Npn \prg_quicksort_function:n {\ERROR}
2372 \cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}
```

(End definition for `\prg_quicksort_function:n`. This function is documented on page 48.)

166.10 Deprecated functions

These were depreciated on 2011-05-27 and will be removed entirely by 2011-08-31.

`\prg_new_map_functions:Nn`
`\prg_set_map_functions:Nn`

As we have restructured the structured variables, these are no longer needed.

```
2373 \cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 { \depreciated }
2374 \cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 { \depreciated }
```

(End definition for `\prg_new_map_functions:Nn`. This function is documented on page ??.)

```
2375 </initex | package>
```

167 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

```
2376 <*initex | package>

2377 <*package>
2378 \ProvidesExplPackage
2379   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2380 \package_check_loaded_expl:
2381 </package>
```

`\quark_new:N` Allocate a new quark.

```
2382 \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 49.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value
`\q_mark` and `\q_no_value` marks an empty argument.
`\q_no_value`
`\q_stop`

```
2383 \quark_new:N \q_nil
2384 \quark_new:N \q_mark
2385 \quark_new:N \q_no_value
2386 \quark_new:N \q_stop
```


`\q_recursion_tail` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
2387 \quark_new:N \q_recursion_tail
2388 \quark_new:N \q_recursion_stop
```

`\quark_if_recursion_tail_stop:N` When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
2389 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2390 {
2391   \if_meaning:w #1 \q_recursion_tail
2392   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2393   \fi:
2394 }
2395 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2
2396 {
2397   \if_meaning:w #1 \q_recursion_tail
2398   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2399   \else:
2400     \exp_after:wN \use_none:n
2401   \fi:
2402   {#2}
2403 }
```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 51.)

`\quark_if_recursion_tail_stop:n` 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 `\if_catcode:w`, 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`.

```
2404 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2405 {
2406   \if_catcode:w
2407     A
2408     \etex_detokenize:D \exp_after:wN
2409     {
2410       \quark_if_recursion_tail_aux:w #1 \q_recursion_stop
2411       \q_recursion_tail \q_recursion_stop \q_stop
2412     }
2413 }
```

```

2413     A
2414     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2415     \fi:
2416   }
2417   \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2
2418   {
2419     \if_catcode:w
2420       A
2421       \etex_detokenize:D \exp_after:wN
2422       {
2423         \quark_if_recursion_tail_aux:w #1 \q_recursion_stop
2424         \q_recursion_tail \q_recursion_stop \q_stop
2425       }
2426       A
2427       \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2428     \else:
2429       \exp_after:wN \use_none:n
2430     \fi:
2431     {#2}
2432   }
2433   \cs_new:Npn \quark_if_recursion_tail_aux:w
2434   #1 \q_recursion_tail #2 \q_recursion_stop #3 \q_stop
2435   { #1 #2 }
2436   \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2437   \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 51.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with
`\quark_if_nil:NTF` `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value_p:N.` wrongly given a string like `aabc` instead of a single token.⁶
`\quark_if_no_value_p:c`
`\quark_if_no_value:N.TF`
`\quark_if_no_value:cTF`

```

2438   \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
2439   {
2440     \if_meaning:w \q_nil #1
2441     \prg_return_true:
2442   \else:
2443     \prg_return_false:
2444   \fi:
2445   }
2446   \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2447   {
2448     \if_meaning:w \q_no_value #1
2449     \prg_return_true:
2450   \else:
2451     \prg_return_false:
2452   \fi:

```

⁶It may still loop in special circumstances however!

```

2453 }
2454 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2455 \cs_generate_variant:Nn \quark_if_no_value:NT { c }
2456 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2457 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for `\quark_if_nil:N`. These functions are documented on page 50.)

```

\quark_if_nil_p:n These are essentially \str_if_eq:nn tests but done directly.
\quark_if_nil_p:V
\quark_if_nil_p:o
\quark_if_nil:nTF
\quark_if_nil:VTF
\quark_if_nil:oTF
\quark_if_no_value_p:n
\quark_if_no_value:nTF
2458 \prg_new_conditional:Nnn \quark_if_nil:n { p, T, F, TF }
2459 {
2460   \if_int_compare:w \pdfTeX_strcmp:D
2461     { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero
2462     \prg_return_true:
2463   \else:
2464     \prg_return_false:
2465   \fi:
2466 }
2467 \prg_new_conditional:Nnn \quark_if_no_value:n { p, T, F, TF }
2468 {
2469   \if_int_compare:w \pdfTeX_strcmp:D
2470     { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero
2471   \prg_return_true:
2472   \else:
2473     \prg_return_false:
2474   \fi:
2475 }
2476 \cs_generate_variant:Nn \quark_if_nil_p:n { V, o }
2477 \cs_generate_variant:Nn \quark_if_nil:nTF { V, o }
2478 \cs_generate_variant:Nn \quark_if_nil:nT { V, o }
2479 \cs_generate_variant:Nn \quark_if_nil:nF { V, o }

```

(End definition for `\quark_if_nil:n`, `\quark_if_nil:V`, and `\quark_if_nil:o`. These functions are documented on page 50.)

```

2480 </initex | package>

```

168 l3token implementation

```

2481 <*initex | package>
2482 <*package>
2483 \ProvidesExplPackage
2484   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2485 \package_check_loaded_expl:
2486 </package>

```

168.1 Character tokens

Category code changes.

```
\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n
```

```
2487 \cs_new_protected_nopar:Npn \char_set_catcode:nn #1#2
2488 { \tex_catcode:D #1 = \int_eval:w #2 \int_eval_end: }
2489 \cs_new_nopar:Npn \char_value_catcode:n #1
2490 { \tex_the:D \tex_catcode:D \int_eval:w #1\int_eval_end: }
2491 \cs_new_nopar:Npn \char_show_value_catcode:n #1
2492 { \tex_showthe:D \tex_catcode:D \int_eval:w #1 \int_eval_end: }
```

(End definition for `\char_set_catcode:nn`. This function is documented on page 55.)

```
\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

```
2493 \cs_new_protected_nopar:Npn \char_set_catcode_escape:N #1
2494 { \char_set_catcode:nn { '#1 } \c_zero }
2495 \cs_new_protected_nopar:Npn \char_set_catcode_group_begin:N #1
2496 { \char_set_catcode:nn { '#1 } \c_one }
2497 \cs_new_protected_nopar:Npn \char_set_catcode_group_end:N #1
2498 { \char_set_catcode:nn { '#1 } \c_two }
2499 \cs_new_protected_nopar:Npn \char_set_catcode_math_toggle:N #1
2500 { \char_set_catcode:nn { '#1 } \c_three }
2501 \cs_new_protected_nopar:Npn \char_set_catcode_alignment:N #1
2502 { \char_set_catcode:nn { '#1 } \c_four }
2503 \cs_new_protected_nopar:Npn \char_set_catcode_end_line:N #1
2504 { \char_set_catcode:nn { '#1 } \c_five }
2505 \cs_new_protected_nopar:Npn \char_set_catcode_parameter:N #1
2506 { \char_set_catcode:nn { '#1 } \c_six }
2507 \cs_new_protected_nopar:Npn \char_set_catcode_math_superscript:N #1
2508 { \char_set_catcode:nn { '#1 } \c_seven }
2509 \cs_new_protected_nopar:Npn \char_set_catcode_math_subscript:N #1
2510 { \char_set_catcode:nn { '#1 } \c_eight }
2511 \cs_new_protected_nopar:Npn \char_set_catcode_ignore:N #1
2512 { \char_set_catcode:nn { '#1 } \c_nine }
2513 \cs_new_protected_nopar:Npn \char_set_catcode_space:N #1
2514 { \char_set_catcode:nn { '#1 } \c_ten }
2515 \cs_new_protected_nopar:Npn \char_set_catcode_letter:N #1
2516 { \char_set_catcode:nn { '#1 } \c_eleven }
2517 \cs_new_protected_nopar:Npn \char_set_catcode_other:N #1
2518 { \char_set_catcode:nn { '#1 } \c_twelve }
2519 \cs_new_protected_nopar:Npn \char_set_catcode_active:N #1
2520 { \char_set_catcode:nn { '#1 } \c_thirteen }
2521 \cs_new_protected_nopar:Npn \char_set_catcode_comment:N #1
2522 { \char_set_catcode:nn { '#1 } \c_fourteen }
2523 \cs_new_protected_nopar:Npn \char_set_catcode_invalid:N #1
2524 { \char_set_catcode:nn { '#1 } \c_fifteen }
```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 54.)

```

\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

2525 \cs_new_protected_nopar:Npn \char_set_catcode_escape:n #1
2526 { \char_set_catcode:nn {#1} \c_zero }
2527 \cs_new_protected_nopar:Npn \char_set_catcode_group_begin:n #1
2528 { \char_set_catcode:nn {#1} \c_one }
2529 \cs_new_protected_nopar:Npn \char_set_catcode_group_end:n #1
2530 { \char_set_catcode:nn {#1} \c_two }
2531 \cs_new_protected_nopar:Npn \char_set_catcode_math_toggle:n #1
2532 { \char_set_catcode:nn {#1} \c_three }
2533 \cs_new_protected_nopar:Npn \char_set_catcode_alignment:n #1
2534 { \char_set_catcode:nn {#1} \c_four }
2535 \cs_new_protected_nopar:Npn \char_set_catcode_end_line:n #1
2536 { \char_set_catcode:nn {#1} \c_five }
2537 \cs_new_protected_nopar:Npn \char_set_catcode_parameter:n #1
2538 { \char_set_catcode:nn {#1} \c_six }
2539 \cs_new_protected_nopar:Npn \char_set_catcode_math_superscript:n #1
2540 { \char_set_catcode:nn {#1} \c_seven }
2541 \cs_new_protected_nopar:Npn \char_set_catcode_math_subscript:n #1
2542 { \char_set_catcode:nn {#1} \c_eight }
2543 \cs_new_protected_nopar:Npn \char_set_catcode_ignore:n #1
2544 { \char_set_catcode:nn {#1} \c_nine }
2545 \cs_new_protected_nopar:Npn \char_set_catcode_space:n #1
2546 { \char_set_catcode:nn {#1} \c_ten }
2547 \cs_new_protected_nopar:Npn \char_set_catcode_letter:n #1
2548 { \char_set_catcode:nn {#1} \c_eleven }
2549 \cs_new_protected_nopar:Npn \char_set_catcode_other:n #1
2550 { \char_set_catcode:nn {#1} \c_twelve }
2551 \cs_new_protected_nopar:Npn \char_set_catcode_active:n #1
2552 { \char_set_catcode:nn {#1} \c_thirteen }
2553 \cs_new_protected_nopar:Npn \char_set_catcode_comment:n #1
2554 { \char_set_catcode:nn {#1} \c_fourteen }
2555 \cs_new_protected_nopar:Npn \char_set_catcode_invalid:n #1
2556 { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 54.)

```

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n

2557 \cs_new_protected_nopar:Npn \char_set_mathcode:nn #1#2
2558 { \tex_mathcode:D #1 = \int_eval:w #2 \int_eval_end: }
2559 \cs_new_nopar:Npn \char_value_mathcode:n #1
2560 { \tex_the:D \tex_mathcode:D \int_eval:w #1 \int_eval_end: }
2561 \cs_new_nopar:Npn \char_show_value_mathcode:n #1
2562 { \tex_showthe:D \tex_mathcode:D \int_eval:w #1 \int_eval_end: }
2563 \cs_new_protected_nopar:Npn \char_set_lccode:nn #1#2
2564 { \tex_lccode:D #1 = \int_eval:w #2 \int_eval_end: }
2565 \cs_new_nopar:Npn \char_value_lccode:n #1
2566 { \tex_the:D \tex_lccode:D \int_eval:w #1 \int_eval_end: }

```

Pretty repetitive, but necessary!

```

2567 \cs_new_nopar:Npn \char_show_value_lccode:n #1
2568 { \tex_showthe:D \tex_lccode:D \int_eval:w #1 \int_eval_end: }
2569 \cs_new_protected_nopar:Npn \char_set_uccode:nn #1#2
2570 { \tex_uccode:D #1 = \int_eval:w #2 \int_eval_end: }
2571 \cs_new_nopar:Npn \char_value_uccode:n #1
2572 { \tex_the:D \tex_uccode:D \int_eval:w #1 \int_eval_end: }
2573 \cs_new_nopar:Npn \char_show_value_uccode:n #1
2574 { \tex_showthe:D \tex_uccode:D \int_eval:w #1 \int_eval_end: }
2575 \cs_new_protected_nopar:Npn \char_set_sfcode:nn #1#2
2576 { \tex_sfcode:D #1 = \int_eval:w #2 \int_eval_end: }
2577 \cs_new_nopar:Npn \char_value_sfcode:n #1
2578 { \tex_the:D \tex_sfcode:D \int_eval:w #1 \int_eval_end: }
2579 \cs_new_nopar:Npn \char_show_value_sfcode:n #1
2580 { \tex_showthe:D \tex_sfcode:D \int_eval:w #1 \int_eval_end: }

```

(End definition for `\char_set_mathcode:nn`. This function is documented on page 57.)

168.2 Generic tokens

`\token_new:Nn` Creates a new token.

```

2581 \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 57.)

`\c_group_begin_token` `\c_group_end_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```

\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_catcode_letter_token
\c_catcode_other_token
2582 \cs_new_eq:NN \c_group_begin_token {
2583 \cs_new_eq:NN \c_group_end_token }
2584 \group_begin:
2585 \char_set_catcode_math_toggle:N \*
2586 \token_new:Nn \c_math_toggle_token { * }
2587 \char_set_catcode_alignment:N \*
2588 \token_new:Nn \c_alignment_token { * }
2589 \token_new:Nn \c_parameter_token { # }
2590 \token_new:Nn \c_math_superscript_token { ^ }
2591 \char_set_catcode_math_subscript:N \*
2592 \token_new:Nn \c_math_subscript_token { * }
2593 \token_new:Nn \c_space_token { ~ }
2594 \token_new:Nn \c_catcode_letter_token { a }
2595 \token_new:Nn \c_catcode_other_token { 1 }
2596 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 57.)

`\c_catcode_active_tl` Not an implicit token!

```

2597 \group_begin:
2598 \char_set_catcode_active:N \*
2599 \cs_new_nopar:Npn \c_catcode_active_tl { \exp_not:N * }
2600 \group_end:

```

168.3 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:NTF`

```
2601 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
2602 {
2603   \if_catcode:w \exp_not:N #1 \c_group_begin_token
2604   \prg_return_true: \else: \prg_return_false: \fi:
2605 }
```

(End definition for `\token_if_group_begin:N`. These functions are documented on page 58.)

`\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`

```
2606 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
2607 {
2608   \if_catcode:w \exp_not:N #1 \c_group_end_token
2609   \prg_return_true: \else: \prg_return_false: \fi:
2610 }
```

(End definition for `\token_if_group_end:N`. These functions are documented on page 59.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:NTF`

```
2611 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
2612 {
2613   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
2614   \prg_return_true: \else: \prg_return_false: \fi:
2615 }
```

(End definition for `\token_if_math_toggle:N`. These functions are documented on page 59.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token` for this.
`\token_if_alignment:NTF`

```
2616 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
2617 {
2618   \if_catcode:w \exp_not:N #1 \c_alignment_token
2619   \prg_return_true: \else: \prg_return_false: \fi:
2620 }
```

(End definition for `\token_if_alignment:N`. These functions are documented on page 59.)

`\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: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they will remain after the group.

```

2621 \group_begin:
2622 \cs_set_eq:NN \c_parameter_token \scan_stop:
2623 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
2624 {
2625   \if_catcode:w \exp_not:N #1 \c_parameter_token
2626     \prg_return_true: \else: \prg_return_false: \fi:
2627 }
2628 \group_end:

```

(End definition for `\token_if_parameter:N`. These functions are documented on page 59.)

`\token_if_math_superscript_p:N`
`\token_if_math_superscript:NTF`

Check if token is a math superscript token. We use the constant `\c_superscript_token` for this.

```

2629 \prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T , F , TF }
2630 {
2631   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
2632     \prg_return_true: \else: \prg_return_false: \fi:
2633 }

```

(End definition for `\token_if_math_superscript:N`. These functions are documented on page 59.)

`\token_if_math_subscript_p:N`
`\token_if_math_subscript:NTF`

Check if token is a math subscript token. We use the constant `\c_subscript_token` for this.

```

2634 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
2635 {
2636   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
2637     \prg_return_true: \else: \prg_return_false: \fi:
2638 }

```

(End definition for `\token_if_math_subscript:N`. These functions are documented on page 60.)

`\token_if_space_p:N`
`\token_if_space:NTF`

Check if token is a space token. We use the constant `\c_space_token` for this.

```

2639 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
2640 {
2641   \if_catcode:w \exp_not:N #1 \c_space_token
2642     \prg_return_true: \else: \prg_return_false: \fi:
2643 }

```

(End definition for `\token_if_space:N`. These functions are documented on page 60.)

`\token_if_letter_p:N`
`\token_if_letter:NTF`

Check if token is a letter token. We use the constant `\c_letter_token` for this.

```

2644 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
2645 {
2646   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
2647     \prg_return_true: \else: \prg_return_false: \fi:
2648 }

```


(End definition for `\token_if_letter:N`. These functions are documented on page 60.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_other_char_token` for this.
`\token_if_other:N \underline{TF}`

```

2649 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
2650 {
2651   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
2652   \prg_return_true: \else: \prg_return_false: \fi:
2653 }

```

(End definition for `\token_if_other:N`. These functions are documented on page 60.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_active_char_tl` for this. A technical point is that `\c_active_char_tl` is in fact a macro expanding to `\exp_not:N *`, where `*` is active.
`\token_if_active:N \underline{TF}`

```

2654 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
2655 {
2656   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
2657   \prg_return_true: \else: \prg_return_false: \fi:
2658 }

```

(End definition for `\token_if_active:N`. These functions are documented on page 60.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NN \underline{TF}`

```

2659 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
2660 {
2661   \if_meaning:w #1 #2
2662   \prg_return_true: \else: \prg_return_false: \fi:
2663 }

```

(End definition for `\token_if_eq_meaning:NN`. These functions are documented on page 61.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NN \underline{TF}`

```

2664 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
2665 {
2666   \if_catcode:w \exp_not:N #1 \exp_not:N #2
2667   \prg_return_true: \else: \prg_return_false: \fi:
2668 }

```

(End definition for `\token_if_eq_catcode:NN`. These functions are documented on page 61.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NN \underline{TF}`

```

2669 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
2670 {
2671   \if_charcode:w \exp_not:N #1 \exp_not:N #2
2672   \prg_return_true: \else: \prg_return_false: \fi:
2673 }

```

(End definition for `\token_if_eq_charcode:NN`. These functions are documented on page 61.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`. However, this can fail the five `\tex...\mark:D` primitives, whose meaning has the form `...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:.` We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`

Both `ma` and `:` must be of category code 12 (other), and we achieve using the standard lowercasing technique.

```

2674 \group_begin:
2675 \char_set_catcode_other:N \M
2676 \char_set_catcode_other:N \A
2677 \char_set_lccode:nn { '\; } { '\: }
2678 \char_set_lccode:nn { '\T } { '\T }
2679 \char_set_lccode:nn { '\F } { '\F }
2680 \tl_to_lowercase:n
2681 {
2682   \group_end:
2683   \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T , F , TF }
2684   {
2685     \exp_after:wN \token_if_macro_p_aux:w
2686     \token_to_meaning:N #1 MA; \q_stop
2687   }
2688   \cs_new_nopar:Npn \token_if_macro_p_aux:w #1 MA #2 ; #3 \q_stop
2689   {
2690     \if_int_compare:w \pdfTeX_strcmp:D { #2 } { cro } = \c_zero
2691       \prg_return_true:
2692     \else:
2693       \prg_return_false:
2694     \fi:
2695   }
2696 }

```

(End definition for `\token_if_macro:N`. These functions are documented on page 61.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

2697 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
2698 {
2699     \if_catcode:w \exp_not:N #1 \scan_stop:
2700     \prg_return_true: \else: \prg_return_false: \fi:
2701 }

```

(End definition for `\token_if_cs:N`. These functions are documented on page 61.)

`\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.

```

2702 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
2703 {
2704     \cs_if_exist:NTF #1
2705     {
2706         \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2707         \prg_return_false: \else: \prg_return_true: \fi:
2708     }
2709     { \prg_return_false: }
2710 }

```

(End definition for `\token_if_expandable:N`. These functions are documented on page 62.)

`\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_long_macro_p:N` these characters have catcode 12 so we must do some serious substitutions in the code
`\token_if_protected_macro_p:N` below...

```

2711 \group_begin:
2712 \char_set_lccode:nn { '\T } { '\T }
2713 \char_set_lccode:nn { '\F } { '\F }
2714 \char_set_lccode:nn { '\X } { '\n }
2715 \char_set_lccode:nn { '\Y } { '\t }
2716 \char_set_lccode:nn { '\Z } { '\d }
2717 \char_set_lccode:nn { '\? } { '\ ' }
2718 \tl_map_inline:nn { \X \Y \Z \M \C \H \A \R \O \U \S \K \I \P \L \G \P \E }
2719 { \char_set_catcode:nn { '#1 } \c_twelve }

```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```

2720 \tl_to_lowercase:n
2721 {
2722     \group_end:
2723     \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T , F , TF }

```

`\token_if_dim_register_p:N`
`\token_if_skip_register_p:N`
`\token_if_int_register_p:N`
`\token_if_toks_register_p:N`
`\token_if_chardef_p_aux:w`
`\token_if_mathchardef_p_aux:w`
`\token_if_int_register_p_aux:w`
`\token_if_skip_register_p_aux:w`
`\token_if_dim_register_p_aux:w`
`\token_if_toks_register_p_aux:w`
`\token_if_protected_macro_p_aux:w`
`\token_if_long_macro_p_aux:w`
`\token_if_protected_long_macro_p_aux:w`

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since T_EX thinks of such tokens as hexadecimal so it stores them as `\char"<hex number>` or `\mathchar"<hex number>`.

```

2724     {
2725         \exp_after:wN \token_if_chardef_aux:w
2726         \token_to_meaning:N #1 ?CHAR" \q_stop
2727     }
2728 \cs_new_nopar:Npn \token_if_chardef_aux:w #1 ?CHAR" #2 \q_stop
2729 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

2730 \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T , F , TF }
2731 {
2732     \exp_after:wN \token_if_mathchardef_aux:w
2733     \token_to_meaning:N #1 ?MAYHCHAR" \q_stop
2734 }
2735 \cs_new_nopar:Npn \token_if_mathchardef_aux:w #1 ?MAYHCHAR" #2 \q_stop
2736 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

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.

```

2737 \prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T , F , TF }
2738 {
2739     \if_meaning:w \tex_countdef:D #1
2740     \prg_return_false:
2741     \else:
2742         \exp_after:wN \token_if_int_register_aux:w
2743         \token_to_meaning:N #1 ?COUXY \q_stop
2744     \fi:
2745 }
2746 \cs_new_nopar:Npn \token_if_int_register_aux:w #1 ?COUXY #2 \q_stop
2747 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Skip registers are done the same way as the integer registers.

```

2748 \prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T , F , TF }
2749 {
2750     \if_meaning:w \tex_skipdef:D #1
2751     \prg_return_false:
2752     \else:
2753         \exp_after:wN \token_if_skip_register_aux:w
2754         \token_to_meaning:N #1 ?SKIP \q_stop
2755     \fi:
2756 }
2757 \cs_new_nopar:Npn \token_if_skip_register_aux:w #1 ?SKIP #2 \q_stop
2758 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Dim registers. No news here

```

2759 \prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T , F , TF }
2760 {
2761     \if_meaning:w \tex_dimendef:D #1
2762     \c_false_bool
2763     \else:

```

```

2764         \exp_after:wN \token_if_dim_register_aux:w
2765         \token_to_meaning:N #1 ?ZIMEX \q_stop
2766     \fi:
2767 }
2768 \cs_new_nopar:Npn \token_if_dim_register_aux:w #1 ?ZIMEX #2 \q_stop
2769 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Toks registers.

```

2770 \prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T , F , TF }
2771 {
2772     \if_meaning:w \tex_toksdef:D #1
2773     \prg_return_false:
2774 }
2775 \exp_after:wN \token_if_toks_register_aux:w
2776 \token_to_meaning:N #1 ?YOKS \q_stop
2777 \fi:
2778 }
2779 \cs_new_nopar:Npn \token_if_toks_register_aux:w #1 ?YOKS #2 \q_stop
2780 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Protected macros.

```

2781 \prg_new_conditional:Npnn \token_if_protected_macro:N #1
2782 { p , T , F , TF }
2783 {
2784     \exp_after:wN \token_if_protected_macro_aux:w
2785     \token_to_meaning:N #1 ?PROYECYEZ~MACRO \q_stop
2786 }
2787 \cs_new_nopar:Npn \token_if_protected_macro_aux:w
2788 #1 ?PROYECYEZ~MACRO #2 \q_stop
2789 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Long macros.

```

2790 \prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T , F , TF }
2791 {
2792     \exp_after:wN \token_if_long_macro_aux:w
2793     \token_to_meaning:N #1 ?LOXG~MACRO \q_stop
2794 }
2795 \cs_new_nopar:Npn \token_if_long_macro_aux:w #1 ?LOXG~MACRO #2 \q_stop
2796 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

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.

```

2797 \prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
2798 { p , T , F , TF }
2799 {
2800     \exp_after:wN \token_if_protected_long_macro_aux:w
2801     \token_to_meaning:N #1 ?PROYECYEZ?LOXG~MACRO \q_stop

```

```

2802     }
2803     \cs_new_nopar:Npn \token_if_protected_long_macro_aux:w
2804       #1 ?PROYECY EZ?LOXG~MACRO #2 \q_stop
2805       { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Finally the `\tl_to_lowercase:n` ends!

```

2806   }

```

(End definition for `\token_if_chardef:N` and others. These functions are documented on page 63.)

```

\token_if_primitive_p:N
\token_if_primitive:NTF
\token_if_primitive_aux:NNw
\token_if_primitive_aux_space:w
\token_if_primitive_aux_nullfont:N
\token_if_primitive_aux_loop:N
\token_if_primitive_auxii:Nw
\token_if_primitive_aux_undefined:N

```

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\c_undefined:D` is not a primitive, but its meaning is `undefined`, only letters; on the other hand, `\tex_space:D`, `\tex_italiccorr:D`, `\tex_hyphen:D`, `\tex_firstmark:D`, `\tex_topmark:D`, `\tex_botmark:D`, `\tex_splitfirstmark:D`, `\tex_splitbotmark:D`, and `\tex_nullfont:D` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be inexistent depending on `\tex_endlinechar:D`), and takes care of three of the exceptions: `\tex_space:D`, `\tex_italiccorr:D` and `\tex_hyphen:D`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\q_stop`.

The meaning of each one of the five `\tex_...mark:D` primitives has the form $\langle letters \rangle : \langle user material \rangle$. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\c_undefined:D`, which is not a primitive, and `\tex_nullfont:D`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\tex_nullfont:D`. Otherwise, we go through characters one by one, and stop at the first character less than `'A` (this is not quite a test for “only letters”, but is close enough to work in this context). If this first character is `:` then we have a primitive, or `\c_undefined:D`, and if it is `"` or a digit, then the token is not a primitive.

```

2807 \tex_chardef:D \c_token_A_int = 'A ~ %
2808 \group_begin:
2809 \char_set_catcode_other:N \;
2810 \char_set_lccode:nn { '\; } { '\: }
2811 \char_set_lccode:nn { '\T } { '\T }
2812 \char_set_lccode:nn { '\F } { '\F }
2813 \tl_to_lowercase:n {

```

```

2814 \group_end:
2815 \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
2816 {
2817   \token_if_macro:NTF #1
2818   \prg_return_false:
2819   {
2820     \exp_after:wN \token_if_primitive_aux:NNw
2821     \token_to_meaning:N #1 ; ; ; \q_stop #1
2822   }
2823 }
2824 \cs_new_nopar:Npn \token_if_primitive_aux:NNw #1#2 #3 ; #4 \q_stop
2825 {
2826   \tl_if_empty:oTF { \token_if_primitive_aux_space:w #3 ~ }
2827   { \token_if_primitive_aux_loop:N #3 ; \q_stop }
2828   { \token_if_primitive_aux_nullfont:N }
2829 }
2830 }
2831 \cs_new_nopar:Npn \token_if_primitive_aux_space:w #1 ~ { }
2832 \cs_new:Npn \token_if_primitive_aux_nullfont:N #1
2833 {
2834   \if_meaning:w \tex_nullfont:D #1
2835   \prg_return_true:
2836   \else:
2837   \prg_return_false:
2838   \fi:
2839 }
2840 \cs_new_nopar:Npn \token_if_primitive_aux_loop:N #1
2841 {
2842   \if_num:w '#1 < \c_token_A_int %
2843   \exp_after:wN \token_if_primitive_auxii:Nw
2844   \exp_after:wN #1
2845   \else:
2846   \exp_after:wN \token_if_primitive_aux_loop:N
2847   \fi:
2848 }
2849 \cs_new_nopar:Npn \token_if_primitive_auxii:Nw #1 #2 \q_stop
2850 {
2851   \if:w : #1
2852   \exp_after:wN \token_if_primitive_aux_undefined:N
2853   \else:
2854   \prg_return_false:
2855   \exp_after:wN \use_none:n
2856   \fi:
2857 }
2858 \cs_new:Npn \token_if_primitive_aux_undefined:N #1
2859 {
2860   \if_cs_exist:N #1
2861   \prg_return_true:
2862   \else:
2863   \prg_return_false:

```

```

2864     \fi:
2865 }

```

(End definition for `\token_if_primitive:N`. These functions are documented on page 64.)

168.4 Peeking ahead at the next token

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

`\l_peek_token` Storage tokens which are publicly documented: the token peeked.

`\g_peek_token`

```

2866 \cs_new_eq:NN \l_peek_token ?
2867 \cs_new_eq:NN \g_peek_token ?

```

`\l_peek_search_token` The token to search for as an implicit token: cf. `\l_peek_search_tl`.

```

2868 \cs_new_eq:NN \l_peek_search_token ?

```

`\l_peek_search_tl` The token to search for as an explicit token: cf. `\l_peek_search_token`.

```

2869 \cs_new_nopar:Npn \l_peek_search_tl { }

```

`\peek_true:w` Functions used by the branching and space-stripping code.

`\peek_true_aux:w`

`\peek_false:w`

`\peek_tmp:w`

```

2870 \cs_new_nopar:Npn \peek_true:w { }
2871 \cs_new_nopar:Npn \peek_true_aux:w { }
2872 \cs_new_nopar:Npn \peek_false:w { }
2873 \cs_new:Npn \peek_tmp:w { }

```

(End definition for `\peek_true:w` and others.)

`\peek_after:Nw` Simple wrappers for `\tex_futurelet:D`: no arguments absorbed here.

`\peek_after:Nw`

```

2874 \cs_new_protected_nopar:Npn \peek_after:Nw
2875 { \tex_futurelet:D \l_peek_token }
2876 \cs_new_protected_nopar:Npn \peek_gafter:Nw
2877 { \pref_global:D \tex_futurelet:D \g_peek_token }

```

(End definition for `\peek_after:Nw`. This function is documented on page 64.)

`\peek_true_remove:w` A function to remove the next token and then regain control.

```

2878 \cs_new_protected:Npn \peek_true_remove:w
2879 {
2880   \group_align_safe_end:
2881   \tex_afterassignment:D \peek_true_aux:w
2882   \cs_set_eq:NN \peek_tmp:w
2883 }

```

(End definition for \peek_true_remove:w.)

`\peek_token_generic:NNTF` The generic function stores the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself.

```

2884 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4
2885 {
2886   \cs_set_eq:NN \l_peek_search_token #2
2887   \tl_set:Nn \l_peek_search_tl {#2}
2888   \cs_set_nopar:Npx \peek_true:w
2889   {
2890     \exp_not:N \group_align_safe_end:
2891     \exp_not:n {#3}
2892   }
2893   \cs_set_nopar:Npx \peek_false:w
2894   {
2895     \exp_not:N \group_align_safe_end:
2896     \exp_not:n {#4}
2897   }
2898   \group_align_safe_begin:
2899   \peek_after:Nw #1
2900 }
2901 \cs_new_protected:Npn \peek_token_generic:NNT #1#2#3
2902 { \peek_token_generic:NNTF #1 #2 {#3} { } }
2903 \cs_new_protected:Npn \peek_token_generic:NNF #1#2#3
2904 { \peek_token_generic:NNTF #1 #2 { } {#3} }

```

(End definition for \peek_token_generic:NN. This function is documented on page ??.)

`\peek_token_remove_generic:NNTF` For token removal there needs to be a call to the auxiliary function which does the work.

```

2905 \cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4
2906 {
2907   \cs_set_eq:NN \l_peek_search_token #2
2908   \tl_set:Nn \l_peek_search_tl {#2}
2909   \cs_set_eq:NN \peek_true:w \peek_true_remove:w
2910   \cs_set_nopar:Npx \peek_true_aux:w { \exp_not:n {#3} }
2911   \cs_set_nopar:Npx \peek_false:w
2912   {
2913     \exp_not:N \group_align_safe_end:

```

```

2914         \exp_not:n {#4}
2915     }
2916     \group_align_safe_begin:
2917     \peek_after:Nw #1
2918 }
2919 \cs_new_protected:Npn \peek_token_remove_generic:NNT #1#2#3
2920 { \peek_token_remove_generic:NNTF #1 #2 {#3} { } }
2921 \cs_new_protected:Npn \peek_token_remove_generic:NNF #1#2#3
2922 { \peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `\peek_token_remove_generic:NN`. This function is documented on page ??.)

`\peek_execute_branches_catcode:` The category code and meaning tests are straight forward.
`\peek_execute_branches_meaning:`

```

2923 \cs_new_nopar:Npn \peek_execute_branches_catcode:
2924 {
2925     \if_catcode:w
2926     \exp_not:N \l_peek_token \exp_not:N \l_peek_search_token
2927     \exp_after:wN \peek_true:w
2928     \else:
2929     \exp_after:wN \peek_false:w
2930     \fi:
2931 }
2932 \cs_new_nopar:Npn \peek_execute_branches_meaning:
2933 {
2934     \if_meaning:w \l_peek_token \l_peek_search_token
2935     \exp_after:wN \peek_true:w
2936     \else:
2937     \exp_after:wN \peek_false:w
2938     \fi:
2939 }

```

(End definition for `\peek_execute_branches_catcode:` and `\peek_execute_branches_meaning:`. These functions are documented on page ??.)

`\peek_execute_branches_charcode:` First the character code test there is a need to worry about T_EX grabbing brace group
`\peek_execute_branches_charcode:NN` or skipping spaces. These are all tested for using a category code check before grabbing what must be a real single token and doing the comparison.

```

2940 \cs_new_nopar:Npn \peek_execute_branches_charcode:
2941 {
2942     \bool_if:nTF
2943     {
2944         \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token
2945         || \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token
2946         || \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
2947     }
2948     { \peek_false:w }
2949     {
2950         \exp_after:wN \peek_execute_branches_charcode_aux:NN

```

```

2951         \l_peek_search_tl
2952     }
2953 }
2954 \cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2
2955 {
2956     \if:w \exp_not:N #1 \exp_not:N #2
2957         \exp_after:wN \peek_true:w
2958     \else:
2959         \exp_after:wN \peek_false:w
2960     \fi:
2961     #2
2962 }

```

(End definition for `\peek_execute_branches_charcode:`. This function is documented on page ??.)

`\peek_ignore_spaces_execute_branches:` This function removes one token at a time with a mechanism that can be applied to
`\peek_ignore_spaces_execute_branches_aux:` things other than spaces.

```

2963 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches:
2964 {
2965     \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
2966     {
2967         \tex_afterassignment:D \peek_ignore_spaces_execute_branches_aux:
2968         \cs_set_eq:NN \peek_tmp:w
2969     }
2970     { \peek_execute_branches: }
2971 }
2972 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches_aux:
2973 { \peek_after:Nw \peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_ignore_spaces_execute_branches:`. This function is documented on page ??.)

`\peek_def:nnnn` The public functions themselves cannot be defined using `\prg_set_conditional:Npnn`
`\peek_def_aux:nnnnn` and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

```

2974 \group_begin:
2975 \cs_set_nopar:Npn \peek_def:nnnn #1#2#3#4
2976 {
2977     \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { TF }
2978     \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { T }
2979     \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { F }
2980 }
2981 \cs_set_nopar:Npn \peek_def_aux:nnnnn #1#2#3#4#5
2982 {
2983     \cs_gset_nopar:cpx { #1 #5 }
2984     {
2985         \tl_if_empty:nF {#2}
2986         { \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 } }
2987         \exp_not:c { #3 #5 }

```

```

2988         \exp_not:n {#4}
2989     }
2990 }

```

(End definition for `\peek_def:nnnn`.)

```

\peek_catcode:NTF
\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove:NTF
\peek_catcode_remove_ignore_spaces:NTF

```

With everything in place the definitions can take place. First for category codes.

```

2991 \peek_def:nnnn { peek_catcode:N }
2992 { }
2993 { peek_token_generic:NN }
2994 { \peek_execute_branches_catcode: }
2995 \peek_def:nnnn { peek_catcode_ignore_spaces:N }
2996 { \peek_execute_branches_catcode: }
2997 { peek_token_generic:NN }
2998 { \peek_ignore_spaces_execute_branches: }
2999 \peek_def:nnnn { peek_catcode_remove:N }
3000 { }
3001 { peek_token_remove_generic:NN }
3002 { \peek_execute_branches_catcode: }
3003 \peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
3004 { \peek_execute_branches_catcode: }
3005 { peek_token_remove_generic:NN }
3006 { \peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_catcode:N` and others. These functions are documented on page 65.)

```

\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF

```

Then for character codes.

```

3007 \peek_def:nnnn { peek_charcode:N }
3008 { }
3009 { peek_token_generic:NN }
3010 { \peek_execute_branches_charcode: }
3011 \peek_def:nnnn { peek_charcode_ignore_spaces:N }
3012 { \peek_execute_branches_charcode: }
3013 { peek_token_generic:NN }
3014 { \peek_ignore_spaces_execute_branches: }
3015 \peek_def:nnnn { peek_charcode_remove:N }
3016 { }
3017 { peek_token_remove_generic:NN }
3018 { \peek_execute_branches_charcode: }
3019 \peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3020 { \peek_execute_branches_charcode: }
3021 { peek_token_remove_generic:NN }
3022 { \peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_charcode:N` and others. These functions are documented on page 66.)

```

\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF

```

Finally for meaning, with the group closed to remove the temporary definition functions.

```

3023 \peek_def:nnnn { peek_meaning:N }
3024 { }
3025 { peek_token_generic:NN }
3026 { \peek_execute_branches_meaning: }
3027 \peek_def:nnnn { peek_meaning_ignore_spaces:N }
3028 { \peek_execute_branches_meaning: }
3029 { peek_token_generic:NN }
3030 { \peek_ignore_spaces_execute_branches: }
3031 \peek_def:nnnn { peek_meaning_remove:N }
3032 { }
3033 { peek_token_remove_generic:NN }
3034 { \peek_execute_branches_meaning: }
3035 \peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3036 { \peek_execute_branches_meaning: }
3037 { peek_token_remove_generic:NN }
3038 { \peek_ignore_spaces_execute_branches: }
3039 \group_end:

```

(End definition for `\peek_meaning:N` and others. These functions are documented on page 67.)

168.5 Decomposing a macro definition

```

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
\token_get_prefix_arg_replacement_aux:wN

```

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.

```

3040 \exp_args:Nno \use:nn
3041 { \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:wN #1 }
3042 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3043 { #4 {#1} {#2} {#3} }
3044 \cs_new:Npn \token_get_prefix_spec:N #1
3045 {
3046   \token_if_macro:NTF #1
3047   {
3048     \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
3049     \token_to_meaning:N #1 \q_stop \use_i:nnn
3050   }
3051   { \scan_stop: }
3052 }
3053 \cs_new:Npn \token_get_arg_spec:N #1
3054 {
3055   \token_if_macro:NTF #1
3056   {
3057     \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
3058     \token_to_meaning:N #1 \q_stop \use_ii:nnn
3059   }

```

```

3060     { \scan_stop: }
3061   }
3062   \cs_new:Npn \token_get_replacement_spec:N #1
3063   {
3064     \token_if_macro:NTF #1
3065     {
3066       \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
3067       \token_to_meaning:N #1 \q_stop \use_iii:nnn
3068     }
3069     { \scan_stop: }
3070   }

```

(End definition for `\token_get_prefix_spec:N`. This function is documented on page ??.)

168.6 Experimental token functions

```

\char_active_set:Npn
\char_active_set:Npx
\char_active_set:Npn
\char_active_set:Npx
\char_active_set_eq:NN
\char_active_gset_eq:NN
3071 \group_begin:
3072   \char_set_catcode_active:N ^^@
3073   \cs_set:Npn \char_tmp:NN #1#2
3074   {
3075     \cs_new:Npn #1 ##1
3076     {
3077       \char_set_catcode_active:n { '##1 }
3078       \group_begin:
3079       \char_set_lccode:nn { '\^^@ } { '##1 }
3080       \tl_to_lowercase:n { \group_end: #2 ^^@ }
3081     }
3082   }
3083   \char_tmp:NN \char_active_set:Npn   \cs_set:Npn
3084   \char_tmp:NN \char_active_set:Npx   \cs_set:Npx
3085   \char_tmp:NN \char_active_gset:Npn  \cs_gset:Npn
3086   \char_tmp:NN \char_active_gset:Npx  \cs_gset:Npx
3087   \char_tmp:NN \char_active_set_eq:NN  \cs_set_eq:NN
3088   \char_tmp:NN \char_active_gset_eq:NN \cs_gset_eq:NN
3089 \group_end:

```

(End definition for `\char_active_set:Npn` and `\char_active_set:Npx`. These functions are documented on page 68.)

168.7 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```

\char_set_catcode:w Primitives renamed.
\char_set_mathcode:w
\char_set_lccode:w 3090 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
\char_set_uccode:w
\char_set_sfcode:w

```

```

3091 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
3092 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
3093 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
3094 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D

```

(End definition for `\char_set_catcode:w`. This function is documented on page ??.)

```

\char_value_catcode:w More w functions we should not have.
\char_show_value_catcode:w
\char_value_mathcode:w 3095 \cs_new_nopar:Npn \char_value_catcode:w { \tex_the:D \char_set_catcode:w }
\char_show_value_mathcode:w 3096 \cs_new_nopar:Npn \char_show_value_catcode:w
\char_value_lccode:w 3097 { \tex_showthe:D \char_set_catcode:w }
\char_show_value_lccode:w 3098 \cs_new_nopar:Npn \char_value_mathcode:w { \tex_the:D \char_set_mathcode:w }
\char_value_uccode:w 3099 \cs_new_nopar:Npn \char_show_value_mathcode:w
\char_show_value_uccode:w 3100 { \tex_showthe:D \char_set_mathcode:w }
\char_value_sfcode:w 3101 \cs_new_nopar:Npn \char_value_lccode:w { \tex_the:D \char_set_lccode:w }
\char_show_value_sfcode:w 3102 \cs_new_nopar:Npn \char_show_value_lccode:w
3103 { \tex_showthe:D \char_set_lccode:w }
3104 \cs_new_nopar:Npn \char_value_uccode:w { \tex_the:D \char_set_uccode:w }
3105 \cs_new_nopar:Npn \char_show_value_uccode:w
3106 { \tex_showthe:D \char_set_uccode:w }
3107 \cs_new_nopar:Npn \char_value_sfcode:w { \tex_the:D \char_set_sfcode:w }
3108 \cs_new_nopar:Npn \char_show_value_sfcode:w
3109 { \tex_showthe:D \char_set_sfcode:w }

```

(End definition for `\char_value_catcode:w`. This function is documented on page ??.)

```

\peek_after:NN The second argument here must be w.
\peek_gafter:NN

```

```

3110 \cs_new_eq:NN \peek_after:NN \peek_after:Nw
3111 \cs_new_eq:NN \peek_gafter:NN \peek_gafter:Nw

```

(End definition for `\peek_after:NN`. This function is documented on page ??.)

Functions deprecated 2011-05-28 for removal by 2011-08-31.

```

\c_alignment_tab_token
\c_math_shift_token 3112 \cs_new_eq:NN \c_alignment_tab_token \c_alignment_token
\c_letter_token 3113 \cs_new_eq:NN \c_math_shift_token \c_math_toggle_token
\c_other_char_token 3114 \cs_new_eq:NN \c_letter_token \c_catcode_letter_token
3115 \cs_new_eq:NN \c_other_char_token \c_catcode_other_token

```

(End definition for `\c_alignment_tab_token`. This function is documented on page ??.)

```

\c_active_char_token An odd one: this was never a token!

```

```

3116 \cs_new_eq:NN \c_active_char_token \c_catcode_active_tl

```

(End definition for `\c_active_char_token`. This function is documented on page ??.)

\char_make_escape:N Two renames in one block!

\char_make_group_begin:N	3117 \cs_new_eq:NN \char_make_escape:N	\char_set_catcode_escape:N
\char_make_group_end:N	3118 \cs_new_eq:NN \char_make_begin_group:N	\char_set_catcode_group_begin:N
\char_make_math_toggle:N	3119 \cs_new_eq:NN \char_make_end_group:N	\char_set_catcode_group_end:N
\char_make_alignment:N	3120 \cs_new_eq:NN \char_make_math_shift:N	\char_set_catcode_math_toggle:N
\char_make_end_line:N	3121 \cs_new_eq:NN \char_make_alignment_tab:N	\char_set_catcode_alignment:N
\char_make_parameter:N	3122 \cs_new_eq:NN \char_make_end_line:N	\char_set_catcode_end_line:N
\char_make_math_superscript:N	3123 \cs_new_eq:NN \char_make_parameter:N	\char_set_catcode_parameter:N
\char_make_math_subscript:N	3124 \cs_new_eq:NN \char_make_math_superscript:N	
\char_make_ignore:N	3125 \char_set_catcode_math_superscript:N	
\char_make_space:N	3126 \cs_new_eq:NN \char_make_math_subscript:N	
\char_make_letter:N	3127 \char_set_catcode_math_subscript:N	
\char_make_other:N	3128 \cs_new_eq:NN \char_make_ignore:N	\char_set_catcode_ignore:N
\char_make_active:N	3129 \cs_new_eq:NN \char_make_space:N	\char_set_catcode_space:N
\char_make_comment:N	3130 \cs_new_eq:NN \char_make_letter:N	\char_set_catcode_letter:N
\char_make_invalid:N	3131 \cs_new_eq:NN \char_make_other:N	\char_set_catcode_other:N
\char_make_escape:n	3132 \cs_new_eq:NN \char_make_active:N	\char_set_catcode_active:N
\char_make_group_begin:n	3133 \cs_new_eq:NN \char_make_comment:N	\char_set_catcode_comment:N
\char_make_group_end:n	3134 \cs_new_eq:NN \char_make_invalid:N	\char_set_catcode_invalid:N
\char_make_math_toggle:n	3135 \cs_new_eq:NN \char_make_escape:n	\char_set_catcode_escape:n
\char_make_alignment:n	3136 \cs_new_eq:NN \char_make_begin_group:n	\char_set_catcode_group_begin:n
\char_make_end_line:n	3137 \cs_new_eq:NN \char_make_end_group:n	\char_set_catcode_group_end:n
\char_make_parameter:n	3138 \cs_new_eq:NN \char_make_math_shift:n	\char_set_catcode_math_toggle:n
\char_make_math_superscript:n	3139 \cs_new_eq:NN \char_make_alignment_tab:n	\char_set_catcode_alignment:n
\char_make_math_subscript:n	3140 \cs_new_eq:NN \char_make_end_line:n	\char_set_catcode_end_line:n
\char_make_ignore:n	3141 \cs_new_eq:NN \char_make_parameter:n	\char_set_catcode_parameter:n
\char_make_space:n	3142 \cs_new_eq:NN \char_make_math_superscript:n	
\char_make_letter:n	3143 \char_set_catcode_math_superscript:n	
\char_make_other:n	3144 \cs_new_eq:NN \char_make_math_subscript:n	
\char_make_active:n	3145 \char_set_catcode_math_subscript:n	
\char_make_comment:n	3146 \cs_new_eq:NN \char_make_ignore:n	\char_set_catcode_ignore:n
\char_make_invalid:n	3147 \cs_new_eq:NN \char_make_space:n	\char_set_catcode_space:n
	3148 \cs_new_eq:NN \char_make_letter:n	\char_set_catcode_letter:n
	3149 \cs_new_eq:NN \char_make_other:n	\char_set_catcode_other:n
	3150 \cs_new_eq:NN \char_make_active:n	\char_set_catcode_active:n
	3151 \cs_new_eq:NN \char_make_comment:n	\char_set_catcode_comment:n
	3152 \cs_new_eq:NN \char_make_invalid:n	\char_set_catcode_invalid:n

(End definition for \char_make_escape:N and others. These functions are documented on page ??.)

\token_if_alignment_tab_p:N	
\token_if_alignment_tab:N \overline{TF}	
\token_if_math_shift_p:N	3153 \cs_new_eq:NN \token_if_alignment_tab_p:N \token_if_alignment_p:N
\token_if_math_shift:N \overline{TF}	3154 \cs_new_eq:NN \token_if_alignment_tab:NT \token_if_alignment:NT
\token_if_other_char_p:N	3155 \cs_new_eq:NN \token_if_alignment_tab:NF \token_if_alignment:NF
\token_if_other_char:N \overline{TF}	3156 \cs_new_eq:NN \token_if_alignment_tab:N \overline{TF} \token_if_alignment:N \overline{TF}
\token_if_active_char_p:N	3157 \cs_new_eq:NN \token_if_math_shift_p:N \token_if_math_toggle_p:N
\token_if_active_char:N \overline{TF}	3158 \cs_new_eq:NN \token_if_math_shift:NT \token_if_math_toggle:NT
	3159 \cs_new_eq:NN \token_if_math_shift:NF \token_if_math_toggle:NF


```

3160 \cs_new_eq:NN \token_if_math_shift:NTF \token_if_math_toggle:NTF
3161 \cs_new_eq:NN \token_if_other_char_p:N \token_if_other_p:N
3162 \cs_new_eq:NN \token_if_other_char:NT \token_if_other:NT
3163 \cs_new_eq:NN \token_if_other_char:NF \token_if_other:NF
3164 \cs_new_eq:NN \token_if_other_char:NTF \token_if_other:NTF
3165 \cs_new_eq:NN \token_if_active_char_p:N \token_if_active_p:N
3166 \cs_new_eq:NN \token_if_active_char:NT \token_if_active:NT
3167 \cs_new_eq:NN \token_if_active_char:NF \token_if_active:NF
3168 \cs_new_eq:NN \token_if_active_char:NTF \token_if_active:NTF

```

(End definition for `\token_if_alignment_tab:N`. These functions are documented on page ??.)

```

3169 </initex | package>

```

169 l3int implementation

```

3170 <*initex | package>

```

The following test files are used for this code: `m3int001,m3int002,m3int03`.

```

3171 <*package>
3172 \ProvidesExplPackage
3173   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
3174 \package_check_loaded_expl:
3175 </package>

```

`\int_to_roman:w` Done in `l3basics`.

`\if_int_compare:w` (End definition for `\int_to_roman:w`. This function is documented on page 81.)

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

```

\int_eval:w
\int_eval_end:
\if_num:w
\if_int_odd:w
\if_case:w
3176 \cs_set_eq:NN \int_value:w \tex_number:D
3177 \cs_set_eq:NN \int_eval:w \etex_numexpr:D
3178 \cs_new_eq:NN \int_eval_end: \tex_relax:D
3179 \cs_new_eq:NN \if_num:w \tex_ifnum:D
3180 \cs_set_eq:NN \if_int_odd:w \tex_ifodd:D
3181 \cs_new_eq:NN \if_case:w \tex_ifcase:D

```

(End definition for `\int_value:w`. This function is documented on page 81.)

169.1 Integer expressions

`\int_eval:n` Wrapper for `\int_eval:w`. Can be used in an integer expression or directly in the input stream.

```

3182 \cs_new:Npn \int_eval:n #1
3183 { \int_value:w \int_eval:w #1 \int_eval_end: }

```

(End definition for `\int_eval:n`. This function is documented on page 69.)

`\int_max:nn` Functions for min, max, and absolute value.

`\int_min:nn`

`\int_abs:n`

```

3184 \cs_new:Npn \int_abs:n #1
3185 {
3186   \int_value:w
3187   \if_int_compare:w \int_eval:w #1 < \c_zero
3188   -
3189   \fi:
3190   \int_eval:w #1 \int_eval_end:
3191 }
3192 \cs_new:Npn \int_max:nn #1#2
3193 {
3194   \int_value:w \int_eval:w
3195   \if_int_compare:w
3196     \int_eval:w #1 > \int_eval:w #2 \int_eval_end:
3197     #1
3198   \else:
3199     #2
3200   \fi:
3201   \int_eval_end:
3202 }
3203 \cs_new:Npn \int_min:nn #1#2
3204 {
3205   \int_value:w \int_eval:w
3206   \if_int_compare:w
3207     \int_eval:w #1 < \int_eval:w #2 \int_eval_end:
3208     #1
3209   \else:
3210     #2
3211   \fi:
3212   \int_eval_end:
3213 }
```

(End definition for `\int_max:nn`. This function is documented on page 69.)

`\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. This version is thanks to Heiko Oberdiek: getting things right in all cases is
`\int_mod:nn` not so easy.

```

3214 \cs_new:Npn \int_div_truncate:nn #1#2
3215 {
3216   \int_value:w \int_eval:w
3217   \if_int_compare:w \int_eval:w #1 = \c_zero
3218   0
3219   \else:
3220     ( #1 % )
3221   \if_int_compare:w \int_eval:w #1 < \c_zero
3222     \if_int_compare:w \int_eval:w #2 < \c_zero
```

```

3223         - ( #2 + % )
3224     \else:
3225         + ( #2 - % )
3226     \fi:
3227 \else:
3228     \if_int_compare:w \int_eval:w #2 < \c_zero
3229         + ( #2 + % )
3230     \else:
3231         - ( #2 - % )
3232     \fi:
3233     \fi: % ( (
3234         1 ) / 2 )
3235     \fi:
3236     / ( #2 )
3237 \int_eval_end:
3238 }

```

For the sake of completeness:

```

3239 \cs_new:Npn \int_div_round:nn #1#2 { \int_eval:n { ( #1 ) / ( #2 ) } }

```

Finally there's the modulus operation.

```

3240 \cs_new:Npn \int_mod:nn #1#2
3241 {
3242     \int_value:w \int_eval:w
3243     #1 - \int_div_truncate:nn {#1} {#2} * ( #2 )
3244     \int_eval_end:
3245 }

```

(End definition for `\int_div_truncate:nn`. This function is documented on page 70.)

169.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package.
`\int_new:c`

```

3246 <*initex>
3247 \alloc_new:nnnN { int } { 11 } { \c_max_register_int } \tex_countdef:D
3248 </initex>
3249 <*package>
3250 \cs_new_protected_nopar:Npn \int_new:N #1
3251 {
3252     \chk_if_free_cs:N #1
3253     \newcount #1
3254 }
3255 </package>
3256 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N` and `\int_new:c`. These functions are documented on page 70.)

`\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.

```

3257 \cs_new_protected_nopar:Npn \int_const:Nn #1#2
3258 {
3259   \int_compare:nNnTF {#2} > \c_minus_one
3260   {
3261     \int_compare:nNnTF {#2} > \c_max_register_int
3262     {
3263       \int_new:N #1
3264       \int_gset:Nn #1 {#2}
3265     }
3266     {
3267       \chk_if_free_cs:N #1
3268       \pref_global:D \tex_mathchardef:D #1 =
3269       \int_eval:w #2 \int_eval_end:
3270     }
3271   }
3272   {
3273     \int_new:N #1
3274     \int_gset:Nn #1 {#2}
3275   }
3276 }
3277 \cs_generate_variant:Nn \int_const:Nn { c }

```

(End definition for `\int_const:Nn` and `\int_const:cn`. These functions are documented on page 70.)

`\int_zero:N` Functions that reset an *integer* register to zero.

```

\int_zero:c
\int_gzero:N
\int_gzero:c
3278 \cs_new_protected_nopar:Npn \int_zero:N #1 { #1 = \c_zero }
3279 \cs_new_protected_nopar:Npn \int_gzero:N #1 { \pref_global:D #1 = \c_zero }
3280 \cs_generate_variant:Nn \int_zero:N { c }
3281 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End definition for `\int_zero:N` and `\int_zero:c`. These functions are documented on page 71.)

`\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
3282 \cs_new_protected_nopar:Npn \int_set_eq:NN #1#2 { #1 = #2 }
3283 \cs_generate_variant:Nn \int_set_eq:NN { c }
3284 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
3285 \cs_new_protected_nopar:Npn \int_gset_eq:NN #1#2 { \pref_global:D #1 = #2 }
3286 \cs_generate_variant:Nn \int_gset_eq:NN { c }
3287 \cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }

```

(End definition for `\int_set_eq:NN` and others. These functions are documented on page 71.)

169.3 Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter ...

```

\int_add:cn
\int_gadd:Nn
\int_gadd:cn
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

```

```

3288 \cs_new_protected_nopar:Npn \int_add:Nn #1#2
3289 { \tex_advance:D #1 by \int_eval:w #2 \int_eval_end: }
3290 \cs_new_nopar:Npn \int_sub:Nn #1#2
3291 { \tex_advance:D #1 by - \int_eval:w #2 \int_eval_end: }
3292 \cs_new_protected_nopar:Npn \int_gadd:Nn
3293 { \pref_global:D \int_add:Nn }
3294 \cs_new_protected_nopar:Npn \int_gsub:Nn
3295 { \pref_global:D \int_sub:Nn }
3296 \cs_generate_variant:Nn \int_add:Nn { c }
3297 \cs_generate_variant:Nn \int_gadd:Nn { c }
3298 \cs_generate_variant:Nn \int_sub:Nn { c }
3299 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and `\int_add:cn`. These functions are documented on page 72.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\int_incr:c
\int_gincr:N
\int_gincr:c
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c
3300 \cs_new_protected_nopar:Npn \int_incr:N #1
3301 { \tex_advance:D #1 \c_one }
3302 \cs_new_protected_nopar:Npn \int_decr:N #1
3303 { \tex_advance:D #1 \c_minus_one }
3304 \cs_new_protected_nopar:Npn \int_gincr:N
3305 { \pref_global:D \int_incr:N }
3306 \cs_new_protected_nopar:Npn \int_gdecr:N
3307 { \pref_global:D \int_decr:N }
3308 \cs_generate_variant:Nn \int_incr:N { c }
3309 \cs_generate_variant:Nn \int_decr:N { c }
3310 \cs_generate_variant:Nn \int_gincr:N { c }
3311 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and `\int_incr:c`. These functions are documented on page 72.)

`\int_set:Nn` As integers are register-based TeX will issue an error if they are not defined. Thus there is no need for the checking code seen with token list variables.

```

\int_set:cn
\int_gset:Nn
\int_gset:cn
3312 \cs_new_protected_nopar:Npn \int_set:Nn #1#2
3313 { #1 ~ \int_eval:w #2\int_eval_end: }
3314 \cs_new_protected_nopar:Npn \int_gset:Nn { \pref_global:D \int_set:Nn }
3315 \cs_generate_variant:Nn \int_set:Nn { c }
3316 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_set:cn`. These functions are documented on page 72.)

169.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

\int_use:c
3317 \cs_new_eq:NN \int_use:N \tex_the:D
3318 \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 73.)

169.5 Integer expression conditionals

`\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_aux:nw
\int_compare_aux:Nw
  int_compare_=:w      \int_compare_p:n { 5 + \l_tmpa_int != 4 - \l_tmpb_int }
  int_compare_=:w
  int_compare_!=:w
  int_compare_<:w
  int_compare_>:w
  int_compare_<=:w
  int_compare_>=:w

```

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.

```

3319 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3320 { \exp_after:wN \int_compare_aux:nw \int_value:w \int_eval:w #1 \q_stop }

```

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. `\int_to_roman:w` 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 `\int_to_roman:w`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3321 \cs_new:Npn \int_compare_aux:nw #1#2 \q_stop
3322 {
3323   \exp_after:wN \int_compare_aux:Nw
3324   \int_to_roman:w
3325   \if:w #1 -
3326   \else:
3327     -
3328   \fi:
3329   #1#2 \q_mark #1#2 \q_stop
3330 }

```

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.

```

3331 \cs_new:Npn \int_compare_aux:Nw #1#2#3 \q_mark
3332 { \use:c { int_compare_ #1 \if_meaning:w = #2 = \fi: :w } }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3333 \cs_new:cpn { int_compare_=:w } #1 = #2 \q_stop
3334 {
3335   \if_int_compare:w #1 = \int_eval:w #2 \int_eval_end:

```

```

3336     \prg_return_true:
3337 \else:
3338     \prg_return_false:
3339 \fi:
3340 }

```

So is the one using == we just have to use == in the parameter text.

```

3341 \cs_new:cpn { int_compare_==:w } #1 == #2 \q_stop
3342 {
3343     \if_int_compare:w #1 = \int_eval:w #2 \int_eval_end:
3344     \prg_return_true:
3345 \else:
3346     \prg_return_false:
3347 \fi:
3348 }

```

Not equal is just about reversing the truth value.

```

3349 \cs_new:cpn { int_compare_!=:w } #1 != #2 \q_stop
3350 {
3351     \if_int_compare:w #1 = \int_eval:w #2 \int_eval_end:
3352     \prg_return_false:
3353 \else:
3354     \prg_return_true:
3355 \fi:
3356 }

```

Less than and greater than are also straight forward.

```

3357 \cs_new:cpn { int_compare_<:w } #1 < #2 \q_stop
3358 {
3359     \if_int_compare:w #1 < \int_eval:w #2 \int_eval_end:
3360     \prg_return_true:
3361 \else:
3362     \prg_return_false:
3363 \fi:
3364 }
3365 \cs_new:cpn { int_compare_>:w } #1 > #2 \q_stop
3366 {
3367     \if_int_compare:w #1 > \int_eval:w #2 \int_eval_end:
3368     \prg_return_true:
3369 \else:
3370     \prg_return_false:
3371 \fi:
3372 }

```

The less than or equal operation is just the opposite of the greater than operation. *Vice versa* for less than or equal.

```

3373 \cs_new:cpn { int_compare_<=:w } #1 <= #2 \q_stop

```

```

3374 {
3375   \if_int_compare:w #1 > \int_eval:w #2 \int_eval_end:
3376   \prg_return_false:
3377   \else:
3378     \prg_return_true:
3379   \fi:
3380 }
3381 \cs_new:cpn { int_compare_>=:w } #1 >= #2 \q_stop
3382 {
3383   \if_int_compare:w #1 < \int_eval:w #2 \int_eval_end:
3384   \prg_return_false:
3385   \else:
3386     \prg_return_true:
3387   \fi:
3388 }

```

(End definition for `\int_compare:n`. These functions are documented on page 73.)

`\int_compare_p:nNn` More efficient but less natural in typing.

`\int_compare:nNnTF`

```

3389 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF}
3390 {
3391   \if_int_compare:w \int_eval:w #1 #2 \int_eval:w #3 \int_eval_end:
3392   \prg_return_true:
3393   \else:
3394     \prg_return_false:
3395   \fi:
3396 }

```

(End definition for `\int_compare:nNn`. These functions are documented on page 73.)

`\int_if_odd_p:n` A predicate function.

`\int_if_odd:nTF`

`\int_if_even_p:n`

`\int_if_even:nTF`

```

3397 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
3398 {
3399   \if_int_odd:w \int_eval:w #1 \int_eval_end:
3400   \prg_return_true:
3401   \else:
3402     \prg_return_false:
3403   \fi:
3404 }
3405 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3406 {
3407   \if_int_odd:w \int_eval:w #1 \int_eval_end:
3408   \prg_return_false:
3409   \else:
3410     \prg_return_true:
3411   \fi:
3412 }

```

(End definition for `\int_if_odd:n`. These functions are documented on page 74.)

169.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

`\int_until_do:nn`
`\int_do_while:nn`
`\int_do_until:nn`

```

3413 \cs_new:Npn \int_while_do:nn #1#2
3414 {
3415   \int_compare:nT {#1}
3416   {
3417     #2
3418     \int_while_do:nn {#1} {#2}
3419   }
3420 }
3421 \cs_new:Npn \int_until_do:nn #1#2
3422 {
3423   \int_compare:nF {#1}
3424   {
3425     #2
3426     \int_until_do:nn {#1} {#2}
3427   }
3428 }
3429 \cs_new:Npn \int_do_while:nn #1#2
3430 {
3431   #2
3432   \int_compare:nT {#1}
3433   { \int_do_while:nn {#1} {#2} }
3434 }
3435 \cs_new:Npn \int_do_until:nn #1#2
3436 {
3437   #2
3438   \int_compare:nF {#1}
3439   { \int_do_until:nn {#1} {#2} }
3440 }

```

(End definition for `\int_while_do:nn`. This function is documented on page 75.)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```

3441 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3442 {
3443   \int_compare:nNnT {#1} #2 {#3}
3444   {
3445     #4
3446     \int_while_do:nNnn {#1} #2 {#3} {#4}
3447   }
3448 }
3449 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3450 {
3451   \int_compare:nNnF {#1} #2 {#3}
3452   {

```

```

3453     #4
3454     \int_until_do:nNnn {#1} #2 {#3} {#4}
3455   }
3456 }
3457 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3458 {
3459   #4
3460   \int_compare:nNnT {#1} #2 {#3}
3461   { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3462 }
3463 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3464 {
3465   #4
3466   \int_compare:nNnF {#1} #2 {#3}
3467   { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3468 }

```

(End definition for `\int_while_do:nNnn`. This function is documented on page 74.)

169.7 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

3469 \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 75.)

`\int_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".

```

3470 \cs_new_nopar:Npn \int_to_symbols:nnn #1#2#3
3471 {
3472   \int_compare:nNnTF {#1} > {#2}
3473   {
3474     \exp_args:Nf \int_to_symbols:nnn
3475     { \int_div_truncate:nn { #1 - 1 } {#2} } {#2} {#3}
3476     \exp_args:Nf \prg_case_int:nnn
3477     { \int_eval:n { 1 + \int_mod:nn { #1 - 1 } {#2} } }
3478     {#3} { }
3479   }
3480   { \exp_args:Nf \prg_case_int:nnn { \int_eval:n {#1} } {#3} { } }
3481 }

```

(End definition for `\int_to_symbols:nnn`. This function is documented on page 76.)

`\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.

```

3482 \cs_new:Npn \int_to_alph:n #1
3483 {
3484   \int_to_symbols:nnn {#1} { 26 }
3485   {
3486     { 1 } { a }
3487     { 2 } { b }
3488     { 3 } { c }
3489     { 4 } { d }
3490     { 5 } { e }
3491     { 6 } { f }
3492     { 7 } { g }
3493     { 8 } { h }
3494     { 9 } { i }
3495     { 10 } { j }
3496     { 11 } { k }
3497     { 12 } { l }
3498     { 13 } { m }
3499     { 14 } { n }
3500     { 15 } { o }
3501     { 16 } { p }
3502     { 17 } { q }
3503     { 18 } { r }
3504     { 19 } { s }
3505     { 20 } { t }
3506     { 21 } { u }
3507     { 22 } { v }
3508     { 23 } { w }
3509     { 24 } { x }
3510     { 25 } { y }
3511     { 26 } { z }
3512   }
3513 }
3514 \cs_new:Npn \int_to_Alph:n #1
3515 {
3516   \int_to_symbols:nnn {#1} { 26 }
3517   {
3518     { 1 } { A }
3519     { 2 } { B }
3520     { 3 } { C }
3521     { 4 } { D }
3522     { 5 } { E }
3523     { 6 } { F }
3524     { 7 } { G }
3525     { 8 } { H }
3526     { 9 } { I }

```

```

3527      { 10 } { J }
3528      { 11 } { K }
3529      { 12 } { L }
3530      { 13 } { M }
3531      { 14 } { N }
3532      { 15 } { O }
3533      { 16 } { P }
3534      { 17 } { Q }
3535      { 18 } { R }
3536      { 19 } { S }
3537      { 20 } { T }
3538      { 21 } { U }
3539      { 22 } { V }
3540      { 23 } { W }
3541      { 24 } { X }
3542      { 25 } { Y }
3543      { 26 } { Z }
3544    }
3545  }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 75.)

```

\int_to_base:nn
\int_to_base_aux:nnn
\int_to_letter:n

```

Converting from base ten (#1) to a second base (#2) starts with a simple sign check. As the input is base 10 TeX can then do the actual work with the sign itself.

```

3546 \cs_new:Npn \int_to_base:nn #1#2
3547 {
3548   \int_compare:nNnTF {#1} < \c_zero
3549   {
3550     -
3551     \exp_args:Nnf \int_to_base_aux:nnn
3552     { } { \int_eval:n { 0 - ( #1 ) } } {#2}
3553   }
3554   {
3555     \exp_args:Nnf \int_to_base_aux:nnn
3556     { } { \int_eval:n {#1} } {#2}
3557   }
3558 }

```

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.

```

3559 \cs_new:Npn \int_to_base_aux:nnn #1#2#3 {
3560   \int_compare:nNnTF {#2} < {#3}
3561   {

```

```

3562     \int_to_letter:n {#2}
3563     #1
3564   }
3565   {
3566     \exp_args:Nff \int_to_base_aux:nnn
3567     {
3568       \int_to_letter:n { \int_mod:nn {#2} {#3} }
3569       #1
3570     }
3571     { \int_div_truncate:nn {#2} {#3} }
3572     {#3}
3573   }
3574 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged.

```

3575 \cs_new:Npn \int_to_letter:n #1
3576 {
3577   \prg_case_int:nnn { #1 - 9 }
3578   {
3579     { 1 } { A }
3580     { 2 } { B }
3581     { 3 } { C }
3582     { 4 } { D }
3583     { 5 } { E }
3584     { 6 } { F }
3585     { 7 } { G }
3586     { 8 } { H }
3587     { 9 } { I }
3588     { 10 } { J }
3589     { 11 } { K }
3590     { 12 } { L }
3591     { 13 } { M }
3592     { 14 } { N }
3593     { 15 } { O }
3594     { 16 } { P }
3595     { 17 } { Q }
3596     { 18 } { R }
3597     { 19 } { S }
3598     { 20 } { T }
3599     { 21 } { U }
3600     { 22 } { V }
3601     { 23 } { W }
3602     { 24 } { X }
3603     { 25 } { Y }
3604     { 26 } { Z }
3605   }
3606   {#1}
3607 }

```

(End definition for `\int_to_base:nn`. This function is documented on page 80.)

`\int_to_binary:n`
`\int_to_hexadecimal:n`
`\int_to_octal:n`

Wrappers around the generic function.

```

3608 \cs_new:Npn \int_to_binary:n #1
3609   { \int_to_base:nn {#1} { 2 } }
3610 \cs_new:Npn \int_to_hexadecimal:n #1
3611   { \int_to_base:nn {#1} { 16 } }
3612 \cs_new:Npn \int_to_octal:n #1
3613   { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_binary:n`, `\int_to_hexadecimal:n`, and `\int_to_octal:n`. These functions are documented on page 77.)

`\int_to_roman:n`
`\int_to_Roman:n`
`\int_to_roman_aux:N`
`\int_to_roman_aux:N`
`\int_to_roman_i:w`
`\int_to_roman_v:w`
`\int_to_roman_x:w`
`\int_to_roman_l:w`
`\int_to_roman_c:w`
`\int_to_roman_d:w`
`\int_to_roman_m:w`
`\int_to_roman_Q:w`
`\int_to_Roman_i:w`
`\int_to_Roman_v:w`
`\int_to_Roman_x:w`
`\int_to_Roman_l:w`
`\int_to_Roman_c:w`
`\int_to_Roman_d:w`
`\int_to_Roman_m:w`
`\int_to_Roman_Q:w`

The `\int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop will be terminated by the conversion of the Q.

```

3614 \cs_new_nopar:Npn \int_to_roman:n #1
3615   {
3616     \exp_after:wN \int_to_roman_aux:N
3617     \int_to_roman:w \int_eval:n {#1} Q
3618   }
3619 \cs_new_nopar:Npn \int_to_roman_aux:N #1
3620   {
3621     \use:c { int_to_roman_ #1 :w }
3622     \int_to_roman_aux:N
3623   }
3624 \cs_new_nopar:Npn \int_to_Roman:n #1
3625   {
3626     \exp_after:wN \int_to_Roman_aux:N
3627     \int_to_roman:w \int_eval:n {#1} Q
3628   }
3629 \cs_new_nopar:Npn \int_to_Roman_aux:N #1
3630   {
3631     \use:c { int_to_Roman_ #1 :w }
3632     \int_to_Roman_aux:N
3633   }
3634 \cs_new_nopar:Npn \int_to_roman_i:w { i }
3635 \cs_new_nopar:Npn \int_to_roman_v:w { v }
3636 \cs_new_nopar:Npn \int_to_roman_x:w { x }
3637 \cs_new_nopar:Npn \int_to_roman_l:w { l }
3638 \cs_new_nopar:Npn \int_to_roman_c:w { c }
3639 \cs_new_nopar:Npn \int_to_roman_d:w { d }
3640 \cs_new_nopar:Npn \int_to_roman_m:w { m }
3641 \cs_new_nopar:Npn \int_to_roman_Q:w #1 { }
3642 \cs_new_nopar:Npn \int_to_Roman_i:w { I }
3643 \cs_new_nopar:Npn \int_to_Roman_v:w { V }
3644 \cs_new_nopar:Npn \int_to_Roman_x:w { X }
3645 \cs_new_nopar:Npn \int_to_Roman_l:w { L }
3646 \cs_new_nopar:Npn \int_to_Roman_c:w { C }

```

```

3647 \cs_new_nopar:Npn \int_to_Roman_d:w { D }
3648 \cs_new_nopar:Npn \int_to_Roman_m:w { M }
3649 \cs_new_nopar:Npn \int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and `\int_to_Roman:n`. These functions are documented on page 77.)

169.8 Converting from other formats to integers

`\int_get_sign:n` Finding a number and its sign requires dealing with an arbitrary list of + and - symbols.
`\int_get_digits:n` 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
`\int_get_sign_and_digits_aux:nNNN` function.
`\int_get_sign_and_digits_aux:oNNN`

```

3650 \cs_new:Npn \int_get_sign:n #1
3651 {
3652   \int_get_sign_and_digits_aux:nNNN {#1}
3653   \c_true_bool \c_true_bool \c_false_bool
3654 }
3655 \cs_new:Npn \int_get_digits:n #1
3656 {
3657   \int_get_sign_and_digits_aux:nNNN {#1}
3658   \c_true_bool \c_false_bool \c_true_bool
3659 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3660 \cs_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4
3661 {
3662   \tl_if_head_eq_charcode:fNTF {#1} -
3663   {
3664     \bool_if:NTF #2
3665     {
3666       \int_get_sign_and_digits_aux:oNNN
3667       { \use_none:n #1 } \c_false_bool #3#4
3668     }
3669     {
3670       \int_get_sign_and_digits_aux:oNNN
3671       { \use_none:n #1 } \c_true_bool #3#4
3672     }
3673   }
3674   {
3675     \tl_if_head_eq_charcode:fNTF {#1} +
3676     { \int_get_sign_and_digits_aux:oNNN { \use_none:n #1 } #2#3#4 }
3677     {
3678       \bool_if:NT #3 { \bool_if:NF #2 - }
3679       \bool_if:NT #4 {#1}
3680     }
3681   }

```

```

3681     }
3682   }
3683   \cs_generate_variant:Nn \int_get_sign_and_digits_aux:nNNN { o }

```

(End definition for `\int_get_sign:n`. This function is documented on page 80.)

```

\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.

```

3684   \cs_new:Npn \int_from_alph:n #1
3685   {
3686     \int_eval:n
3687     {
3688       \int_get_sign:n {#1}
3689       \exp_args:Nf \int_from_alph_aux:n { \int_get_digits:n {#1} }
3690     }
3691   }
3692   \cs_new:Npn \int_from_alph_aux:n #1
3693   { \int_from_alph_aux:nN { 0 } #1 \q_nil }
3694   \cs_new:Npn \int_from_alph_aux:nN #1#2
3695   {
3696     \quark_if_nil:NTF #2
3697     {#1}
3698     {
3699       \exp_args:Nf \int_from_alph_aux:nN
3700       { \int_eval:n { #1 * 26 + \int_from_alph_aux:N #2 } }
3701     }
3702   }
3703   \cs_new:Npn \int_from_alph_aux:N #1
3704   { \int_eval:n { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } } }

```

(End definition for `\int_from_alph:n`. This function is documented on page 78.)

```

\int_from_base:nn
\int_from_base_aux:nn
\int_from_base_aux:nnN
\int_from_base_aux:N

```

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.

```

3705   \cs_new:Npn \int_from_base:nn #1#2
3706   {
3707     \int_eval:n
3708     {
3709       \int_get_sign:n {#1}
3710       \exp_args:Nf \int_from_base_aux:nn
3711       { \int_get_digits:n {#1} } {#2}
3712     }
3713   }
3714   \cs_new:Npn \int_from_base_aux:nn #1#2
3715   { \int_from_base_aux:nnN { 0 } { #2 } #1 \q_nil }
3716   \cs_new:Npn \int_from_base_aux:nnN #1#2#3
3717   {

```



```

3718 \quark_if_nil:NTF #3
3719 {#1}
3720 {
3721 \exp_args:Nf \int_from_base_aux:nnN
3722 { \int_eval:n { #1 * #2 + \int_from_base_aux:N #3 } }
3723 {#2}
3724 }
3725 }

```

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.

```

3726 \cs_new:Npn \int_from_base_aux:N #1
3727 {
3728 \int_compare:nNnTF { '#1 } < { 58 }
3729 {#1}
3730 {
3731 \int_eval:n
3732 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3733 }
3734 }

```

(End definition for `\int_from_base:nn`. This function is documented on page 78.)

`\int_from_binary:n`
`\int_from_hexadecimal:n`
`\int_from_octal:n`

Wrappers around the generic function.

```

3735 \cs_new:Npn \int_from_binary:n #1
3736 { \int_from_base:nn {#1} \c_two }
3737 \cs_new:Npn \int_from_hexadecimal:n #1
3738 { \int_from_base:nn {#1} \c_sixteen }
3739 \cs_new:Npn \int_from_octal:n #1
3740 { \int_from_base:nn {#1} \c_eight }

```

(End definition for `\int_from_binary:n`, `\int_from_hexadecimal:n`, and `\int_from_octal:n`. These functions are documented on page 78.)

[aux] `\int_from_roman_i_int`, `\int_from_roman_v_int`, `\int_from_roman_x_int`,
`\int_from_roman_l_int`, `\int_from_roman_c_int`, `\int_from_roman_d_int`,
`\int_from_roman_m_int`, `\int_from_roman_I_int`, `\int_from_roman_V_int`, `\int_from_roman_X_int`,
`\int_from_roman_L_int`, `\int_from_roman_C_int`, `\int_from_roman_D_int`, `\int_from_roman_M_int` Constants used to convert from Roman numerals to integers.

```

3741 \int_const:cn { c_int_from_roman_i_int } { 1 }
3742 \int_const:cn { c_int_from_roman_v_int } { 5 }
3743 \int_const:cn { c_int_from_roman_x_int } { 10 }
3744 \int_const:cn { c_int_from_roman_l_int } { 50 }
3745 \int_const:cn { c_int_from_roman_c_int } { 100 }
3746 \int_const:cn { c_int_from_roman_d_int } { 500 }
3747 \int_const:cn { c_int_from_roman_m_int } { 1000 }

```

```

3748 \int_const:cn { c_int_from_roman_I_int } { 1 }
3749 \int_const:cn { c_int_from_roman_V_int } { 5 }
3750 \int_const:cn { c_int_from_roman_X_int } { 10 }
3751 \int_const:cn { c_int_from_roman_L_int } { 50 }
3752 \int_const:cn { c_int_from_roman_C_int } { 100 }
3753 \int_const:cn { c_int_from_roman_D_int } { 500 }
3754 \int_const:cn { c_int_from_roman_M_int } { 1000 }

```

```

\int_from_roman:n
\int_from_roman_aux:NN
\int_from_roman_end:w
\int_from_roman_clean_up:w

```

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 \TeX .

```

3755 \cs_new_nopar:Npn \int_from_roman:n #1
3756 {
3757   \tl_if_blank:nF {#1}
3758   {
3759     \exp_after:wN \int_from_roman_end:w
3760     \int_value:w \int_eval:w
3761     \int_from_roman_aux:NN #1 Q \q_stop
3762   }
3763 }
3764 \cs_new_nopar:Npn \int_from_roman_aux:NN #1#2
3765 {
3766   \str_if_eq:nnTF {#1} { Q }
3767   {#1#2}
3768   {
3769     \str_if_eq:nnTF {#2} { Q }
3770     {
3771       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3772       { \int_from_roman_clean_up:w }
3773       +
3774       \use:c { c_int_from_roman_ #1 _int }
3775       #2
3776     }
3777     {
3778       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3779       { \int_from_roman_clean_up:w }
3780       \cs_if_exist:cF { c_int_from_roman_ #2 _int }
3781       { \int_from_roman_clean_up:w }
3782       \int_compare:nNnTF
3783       { \use:c { c_int_from_roman_ #1 _int } }
3784       <
3785       { \use:c { c_int_from_roman_ #2 _int } }
3786       {
3787         + \use:c { c_int_from_roman_ #2 _int }
3788         - \use:c { c_int_from_roman_ #1 _int }
3789         \int_from_roman_aux:NN
3790       }
3791       {
3792         + \use:c { c_int_from_roman_ #1 _int }
3793         \int_from_roman_aux:NN #2

```

```

3794         }
3795     }
3796 }
3797 }
3798 \cs_new_nopar:Npn \int_from_roman_end:w #1 Q #2 \q_stop
3799 { \tl_if_empty:nTF {#2} {#1} {#2} }
3800 \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 78.)

169.9 Viewing integer

```

\int_show:N
\int_show:c
3801 \cs_new_eq:NN \int_show:N \kernel_register_show:N
3802 \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 79.)

169.10 Constant integers

`\c_minus_one` This is needed early, and so is in `l3basics`

`\c_zero` Again, one in `l3basics` for obvious reasons.

`\c_six` Once again, in `l3basics`.

`\c_seven`

`\c_twelve` Low-number values not previously defined.

`\c_one`

`\c_sixteen`

`\c_three`

`\c_four`

`\c_five`

`\c_eight`

`\c_nine`

`\c_ten`

`\c_eleven`

`\c_thirteen`

`\c_fourteen`

`\c_fifteen`

```

3803 \int_const:Nn \c_one      { 1 }
3804 \int_const:Nn \c_two      { 2 }
3805 \int_const:Nn \c_three    { 3 }
3806 \int_const:Nn \c_four     { 4 }
3807 \int_const:Nn \c_five     { 5 }
3808 \int_const:Nn \c_eight    { 8 }
3809 \int_const:Nn \c_nine     { 9 }
3810 \int_const:Nn \c_ten      { 10 }
3811 \int_const:Nn \c_eleven   { 11 }
3812 \int_const:Nn \c_thirteen { 13 }
3813 \int_const:Nn \c_fourteen { 14 }
3814 \int_const:Nn \c_fifteen { 15 }

```

`\c_thirty_two` One middling value.

```

3815 \int_const:Nn \c_thirty_two { 32 }

```

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

`\c_two_hundred_fifty_six`

```
3816 \int_const:Nn \c_two_hundred_fifty_five { 255 }
3817 \int_const:Nn \c_two_hundred_fifty_six { 256 }
```

`\c_one_hundred` Simple runs of powers of ten.

`\c_one_thousand`

`\c_ten_thousand`

```
3818 \int_const:Nn \c_one_hundred { 100 }
3819 \int_const:Nn \c_one_thousand { 1000 }
3820 \int_const:Nn \c_ten_thousand { 10000 }
```

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
3821 \int_const:Nn \c_max_int { 2 147 483 647 }
```

169.11 Scratch integers

`\l_tmpa_int` We provide four local and two global scratch counters, maybe we need more or less.

`\l_tmpb_int`

`\l_tmpc_int`

`\g_tmpa_int`

`\g_tmpb_int`

```
3822 \int_new:N \l_tmpa_int
3823 \int_new:N \l_tmpb_int
3824 \int_new:N \l_tmpc_int
3825 \int_new:N \g_tmpa_int
3826 \int_new:N \g_tmpb_int
```

169.12 Registers for earlier modules

Needed from other modules:

```
3827 \int_new:N \g_seq_nesting_depth_int
3828 \int_new:N \g_tl_inline_level_int
```

169.13 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\int_convert_from_base_ten:nn`

`\int_convert_to_symbols:nnn`

`\int_convert_to_base_ten:nn`

Some simple renames.

```
3829 \cs_new_eq:NN \int_convert_from_base_ten:nn \int_to_base:nn
3830 \cs_new_eq:NN \int_convert_to_symbols:nnn \int_to_symbols:nnn
3831 \cs_new_eq:NN \int_convert_to_base_ten:nn \int_from_base:nn
```

(End definition for `\int_convert_from_base_ten:nn`. This function is documented on page ??.)

```

\int_to_symbol:n This is rather too tied to LATEX 2ε.
\int_to_symbol_math:n
\int_to_symbol_text:n
3832 \cs_new_nopar:Npn \int_to_symbol:n
3833 {
3834   \mode_if_math:TF
3835     { \int_to_symbol_math:n }
3836     { \int_to_symbol_text:n }
3837 }
3838 \cs_new:Npn \int_to_symbol_math:n #1
3839 {
3840   \int_to_symbols:nnn {#1} { 9 }
3841   {
3842     { 1 } { * }
3843     { 2 } { \dagger }
3844     { 3 } { \ddagger }
3845     { 4 } { \mathsection }
3846     { 5 } { \mathparagraph }
3847     { 6 } { \l }
3848     { 7 } { ** }
3849     { 8 } { \dagger \dagger }
3850     { 9 } { \ddagger \ddagger }
3851   }
3852 }
3853 \cs_new:Npn \int_to_symbol_text:n #1
3854 {
3855   \int_to_symbols:nnn {#1} { 9 }
3856   {
3857     { 1 } { \textasteriskcentered }
3858     { 2 } { \textdagger }
3859     { 3 } { \textdaggerdbl }
3860     { 4 } { \textsection }
3861     { 5 } { \textparagraph }
3862     { 6 } { \textbardbl }
3863     { 7 } { \textasteriskcentered \textasteriskcentered }
3864     { 8 } { \textdagger \textdagger }
3865     { 9 } { \textdaggerdbl \textdaggerdbl }
3866   }
3867 }

```

(End definition for `\int_to_symbol:n`. This function is documented on page 78.)

```
3868 \</initex | package>
```

170 l3skip implementation

```

3869 \<*initex | package>
3870 \<*package>
3871 \ProvidesExplPackage
3872   { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }

```

```

3873 \package_check_loaded_expl:
3874 \</package>

```

170.1 Length primitives renamed

```

\if_dim:w Primitives renamed.
\dim_eval:w
\dim_eval_end:
3875 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
3876 \cs_new_eq:NN \dim_eval:w \etex_dimexpr:D
3877 \cs_new_eq:NN \dim_eval_end: \tex_relax:D

```

(End definition for `\if_dim:w`. This function is documented on page 95.)

170.2 Creating and initialising dim variables

```

\dim_new:N Allocating <dim> registers ...
\dim_new:c
3878 <*initex>
3879 \alloc_new:nnnN { dim } \c_zero \c_max_register_int \tex_dimendef:D
3880 </initex>
3881 <*package>
3882 \cs_new_protected_nopar:Npn \dim_new:N #1
3883 {
3884   \chk_if_free_cs:N #1
3885   \newdimen #1
3886 }
3887 </package>
3888 \cs_generate_variant:Nn \dim_new:N { c }

```

(End definition for `\dim_new:N` and `\dim_new:c`. These functions are documented on page 82.)

```

\dim_zero:N Reset the register to zero.
\dim_zero:c
\dim_gzero:N
\dim_gzero:c
3889 \cs_new_protected_nopar:Npn \dim_zero:N #1 { #1 \c_zero_dim }
3890 \cs_new_protected_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N }
3891 \cs_generate_variant:Nn \dim_zero:N { c }
3892 \cs_generate_variant:Nn \dim_gzero:N { c }

```

(End definition for `\dim_zero:N` and `\dim_zero:c`. These functions are documented on page 82.)

170.3 Setting dim variables

```

\dim_set:Nn Setting dimensions is easy enough.
\dim_set:cn
\dim_gset:Nn
\dim_gset:cn
3893 \cs_new_protected_nopar:Npn \dim_set:Nn #1#2
3894 { #1 ~ \dim_eval:w #2 \dim_eval_end: }
3895 \cs_new_protected_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn }
3896 \cs_generate_variant:Nn \dim_set:Nn { c }
3897 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_set:cn`. These functions are documented on page 83.)

```

\dim_set_eq:NN All straightforward.
\dim_set_eq:cN
\dim_set_eq:Nc
\dim_set_eq:cc
\dim_gset_eq:NN
\dim_gset_eq:cN
\dim_gset_eq:Nc
\dim_gset_eq:cc

```

(End definition for `\dim_set_eq:NN` and others. These functions are documented on page 83.)

```

\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
\dim_set_min:cn
\dim_gset_max:Nn
\dim_gset_max:cn
\dim_gset_min:Nn
\dim_gset_min:cn

```

```

3904 \cs_new_protected_nopar:Npn \dim_set_max:Nn #1#2
3905 { \dim_compare:nNnT {#1} < {#2} { \dim_set:Nn #1 {#2} } }
3906 \cs_new_protected_nopar:Npn \dim_gset_max:Nn #1#2
3907 { \dim_compare:nNnT {#1} < {#2} { \dim_gset:Nn #1 {#2} } }
3908 \cs_new_protected_nopar:Npn \dim_set_min:Nn #1#2
3909 { \dim_compare:nNnT {#1} > {#2} { \dim_set:Nn #1 {#2} } }
3910 \cs_new_protected_nopar:Npn \dim_gset_min:Nn #1#2
3911 { \dim_compare:nNnT {#1} > {#2} { \dim_gset:Nn #1 {#2} } }
3912 \cs_generate_variant:Nn \dim_set_max:Nn { c }
3913 \cs_generate_variant:Nn \dim_gset_max:Nn { c }
3914 \cs_generate_variant:Nn \dim_set_min:Nn { c }
3915 \cs_generate_variant:Nn \dim_gset_min:Nn { c }

```

(End definition for `\dim_set_max:Nn` and `\dim_set_max:cn`. These functions are documented on page 84.)

```

\dim_add:Nn Using by here deals with the (incorrect) case \dimen123.
\dim_add:cn

```

```

\dim_gadd:Nn
\dim_gadd:cn
\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

```

```

3916 \cs_new_protected_nopar:Npn \dim_add:Nn #1#2
3917 { \tex_advance:D #1 by \dim_eval:w #2 \dim_eval_end: }
3918 \cs_new_protected_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
3919 \cs_generate_variant:Nn \dim_add:Nn { c }
3920 \cs_generate_variant:Nn \dim_gadd:Nn { c }
3921 \cs_new_protected_nopar:Npn \dim_sub:Nn #1#2
3922 { \tex_advance:D #1 by - \dim_eval:w #2 \dim_eval_end: }
3923 \cs_new_protected_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn }
3924 \cs_generate_variant:Nn \dim_sub:Nn { c }
3925 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and `\dim_add:cn`. These functions are documented on page 84.)

170.4 Utilities for dimension calculations

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` will not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

3926 \cs_new_nopar:Npn \dim_ratio:nn #1#2
3927 { \dim_ratio_aux:n {#1} / \dim_ratio_aux:n {#2} }
3928 \cs_new_nopar:Npn \dim_ratio_aux:n #1
3929 { \exp_after:wN \int_value:w \dim_eval:w #1 \dim_eval_end: }

```

(End definition for `\dim_ratio:nn`. This function is documented on page 84.)

170.5 Dimension expression conditionals

`\dim_compare_p:nNn`
`\dim_compare:nNnTF`

```

3930 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
3931 {
3932   \if_dim:w \dim_eval:w #1 #2 \dim_eval:w #3 \dim_eval_end:
3933   \prg_return_true: \else: \prg_return_false: \fi:
3934 }

```

(End definition for `\dim_compare_p:nNn`. This function is documented on page 85.)

`\dim_compare_p:n`
`\dim_compare:nTF`
`\dim_compare_aux:wNN`
`\dim_compare_<:nw`
`\dim_compare_=:nw`
`\dim_compare_>:nw`
`\dim_compare_=:nw`
`\dim_compare_<=:nw`
`\dim_compare_!:=:nw`
`\dim_compare_>=:nw`

[This code plus comments are adapted 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 { 5mm + \l_tmpa_dim >= 4pt - \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_use:N \dim_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.

Eventually, we will convert the relation symbol to the appropriate version of `\if_dim:w`, and add `\dim_eval:w` after it. We optimize by placing the end-code already here: this avoids needless grabbing of arguments later.

```

3935 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
3936 {
3937   \exp_after:wN \dim_compare_aux:wNN \dim_use:N \dim_eval:w #1
3938   \dim_eval_end:
3939   \prg_return_true:

```



```

3940     \else:
3941         \prg_return_false:
3942     \fi:
3943 }

```

Contrarily to the case of integers, where we have to remove the result in order to access the relation, `\dim_use:N` nicely produces a result which ends in `pt`. We can thus use a delimited argument to find the relation. `\tl_to_str:n` is needed to convert `pt` to “other” characters.

The relation might be one character, `#2`, or two characters `#2#3`. 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 (unbalanced) test.

```

3944 \exp_args:Nno \use:nn
3945 { \cs_new:Npn \dim_compare_aux:wNN #1 }
3946 { \tl_to_str:n { pt } }
3947 #2 #3
3948 {
3949     \use:c
3950     {
3951         dim_compare_ #2
3952         \if_meaning:w = #3 = \fi:
3953         :nw
3954     }
3955     { #1 pt } #3
3956 }

```

Here, `\dim_eval:w` will begin the right hand side of a dimension comparison (with `\if_dim:w`), closed cleanly by the trailing tokens we put in the definition of `\dim_compare:n`.

The actual comparisons take as a first argument the left-hand side of the comparison (a length). In the case of normal comparisons, just place the relevant `\if_dim:w`, with a trailing `\dim_eval:w` to evaluate the right hand side. For extended comparisons, remove the trailing `=` that we left, before evaluating with `\dim_eval:w`. In both cases, the expansion of `\dim_eval:w` is stopped properly, and the conditional ended correctly by the tokens we put in the definition of `\dim_compare:n`.

Equal, less than and greater than are straightforward.

```

3957 \cs_new:cpn { dim_compare_<:nw } #1 { \if_dim:w #1 < \dim_eval:w }
3958 \cs_new:cpn { dim_compare_=:nw } #1 { \if_dim:w #1 = \dim_eval:w }
3959 \cs_new:cpn { dim_compare_>:nw } #1 { \if_dim:w #1 > \dim_eval:w }

```

For the extended syntax `==`, we remove `#2`, trailing `=` sign, and otherwise act as for `=`.

```

3960 \cs_new:cpn {dim_compare_==:nw} #1#2 { \if_dim:w #1 = \dim_eval:w }

```

Not equal, greater than or equal, less than or equal follow the same scheme as the extended equality syntax, with an additional `\reverse_if:N` to get the opposite of their “simple” analog.

```

3961 \cs_new:cpn {dim_compare_<=:nw} #1#2 {\reverse_if:N \if_dim:w #1 > \dim_eval:w}
3962 \cs_new:cpn {dim_compare_!:=:nw} #1#2 {\reverse_if:N \if_dim:w #1 = \dim_eval:w}
3963 \cs_new:cpn {dim_compare_>=:nw} #1#2 {\reverse_if:N \if_dim:w #1 < \dim_eval:w}

```

(End definition for `\dim_compare:n`. These functions are documented on page 85.)

170.6 Dimension expression loops

`\dim_while_do:nn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nn
\dim_do_while:nn
\dim_do_until:nn
3964 \cs_set:Npn \dim_while_do:nn #1#2
3965 {
3966   \dim_compare:nT {#1}
3967   {
3968     #2
3969     \dim_while_do:nn {#1} {#2}
3970   }
3971 }
3972 \cs_set:Npn \dim_until_do:nn #1#2
3973 {
3974   \dim_compare:nF {#1}
3975   {
3976     #2
3977     \dim_until_do:nn {#1} {#2}
3978   }
3979 }
3980 \cs_set:Npn \dim_do_while:nn #1#2
3981 {
3982   #2
3983   \dim_compare:nT {#1}
3984   { \dim_do_while:nn {#1} {#2} }
3985 }
3986 \cs_set:Npn \dim_do_until:nn #1#2
3987 {
3988   #2
3989   \dim_compare:nF {#1}
3990   { \dim_do_until:nn {#1} {#2} }
3991 }

```

(End definition for `\dim_while_do:nn`. This function is documented on page 86.)

`\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
3992 \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
3993 {
3994   \dim_compare:nNnT {#1} #2 {#3}
3995   {

```

```

3996         #4
3997         \dim_while_do:nNnn {#1} #2 {#3} {#4}
3998     }
3999 }
4000 \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
4001 {
4002     \dim_compare:nNnF {#1} #2 {#3}
4003     {
4004         #4
4005         \dim_until_do:nNnn {#1} #2 {#3} {#4}
4006     }
4007 }
4008 \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4009 {
4010     #4
4011     \dim_compare:nNnT {#1} #2 {#3}
4012     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4013 }
4014 \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4015 {
4016     #4
4017     \dim_compare:nNnF {#1} #2 {#3}
4018     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4019 }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 86.)

170.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4020 \cs_new_nopar:Npn \dim_eval:n #1
4021 { \dim_use:N \dim_eval:w #1 \dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 87.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

```

\dim_use:c
4022 \cs_new_eq:NN \dim_use:N \tex_the:D
4023 \cs_generate_variant:Nn \dim_use:N { c }

```

(End definition for `\dim_use:N` and `\dim_use:c`. These functions are documented on page 87.)

170.8 Viewing dim variables

`\dim_show:N` Diagnostics.

```

\dim_show:c
4024 \cs_new_eq:NN \dim_show:N \kernel_register_show:N
4025 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N` and `\dim_show:c`. These functions are documented on page 87.)

170.9 Constant dimensions

`\c_zero_dim` The source for these depends on whether we are in package mode.
`\c_max_dim`

```

4026 <*initex>
4027 \dim_new:N \c_zero_dim
4028 \dim_new:N \c_max_dim
4029 \dim_set:Nn \c_max_dim { 16383.99999 pt }
4030 </initex>
4031 <*package>
4032 \cs_new_eq:NN \c_zero_dim \z@
4033 \cs_new_eq:NN \c_max_dim \maxdimen
4034 </package>

```

170.10 Scratch dimensions

`\l_tmpa_dim` We provide three local and two global scratch registers, maybe we need more or less.
`\l_tmpb_dim`
`\l_tmpc_dim`
`\g_tmpa_dim`
`\g_tmpb_dim`

```

4035 \dim_new:N \l_tmpa_dim
4036 \dim_new:N \l_tmpb_dim
4037 \dim_new:N \l_tmpc_dim
4038 \dim_new:N \g_tmpa_dim
4039 \dim_new:N \g_tmpb_dim

```

170.11 Creating and initialising skip variables

`\skip_new:N` Allocation of a new internal registers.
`\skip_new:c`

```

4040 <*initex>
4041 \alloc_new:nnnN { skip } \c_zero \c_max_register_int \tex_skipdef:D
4042 </initex>
4043 <*package>
4044 \cs_new_protected_nopar:Npn \skip_new:N #1
4045 {
4046   \chk_if_free_cs:N #1
4047   \newskip #1
4048 }
4049 </package>
4050 \cs_generate_variant:Nn \skip_new:N { c }

```

(End definition for \skip_new:N and \skip_new:c. These functions are documented on page 88.)

`\skip_zero:N` Reset the register to zero.
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c`

```

4051 \cs_new_protected_nopar:Npn \skip_zero:N #1 { #1 \c_zero_skip }
4052 \cs_new_protected_nopar:Npn \skip_gzero:N { \pref_global:D \skip_zero:N }
4053 \cs_generate_variant:Nn \skip_zero:N { c }
4054 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End definition for \skip_zero:N and \skip_zero:c. These functions are documented on page 88.)

170.12 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn`

```

4055 \cs_new_protected_nopar:Npn \skip_set:Nn #1#2
4056 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
4057 \cs_new_protected_nopar:Npn \skip_gset:Nn { \pref_global:D \skip_set:Nn }
4058 \cs_generate_variant:Nn \skip_set:Nn { c }
4059 \cs_generate_variant:Nn \skip_gset:Nn { c }

```

(End definition for `\skip_set:Nn` and `\skip_set:cn`. These functions are documented on page 89.)

`\skip_set_eq:NN` All straightforward.
`\skip_set_eq:cn`
`\skip_set_eq:Nc`
`\skip_set_eq:cc`
`\skip_gset_eq:NN`
`\skip_gset_eq:cn`
`\skip_gset_eq:Nc`
`\skip_gset_eq:cc`

```

4060 \cs_new_protected_nopar:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
4061 \cs_generate_variant:Nn \skip_set_eq:NN { c }
4062 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
4063 \cs_new_protected_nopar:Npn \skip_gset_eq:NN #1#2 { \pref_global:D #1 = #2 }
4064 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
4065 \cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }

```

(End definition for `\skip_set_eq:NN` and others. These functions are documented on page 89.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.
`\skip_add:cn`
`\skip_gadd:Nn`
`\skip_gadd:cn`
`\skip_sub:Nn`
`\skip_sub:cn`
`\skip_gsub:Nn`
`\skip_gsub:cn`

```

4066 \cs_new_protected_nopar:Npn \skip_add:Nn #1#2
4067 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
4068 \cs_new_protected_nopar:Npn \skip_gadd:Nn { \pref_global:D \skip_add:Nn }
4069 \cs_generate_variant:Nn \skip_add:Nn { c }
4070 \cs_generate_variant:Nn \skip_gadd:Nn { c }
4071 \cs_new_protected_nopar:Npn \skip_sub:Nn #1#2
4072 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4073 \cs_new_protected_nopar:Npn \skip_gsub:Nn { \pref_global:D \skip_sub:Nn }
4074 \cs_generate_variant:Nn \skip_sub:Nn { c }
4075 \cs_generate_variant:Nn \skip_gsub:Nn { c }

```

(End definition for `\skip_add:Nn` and `\skip_add:cn`. These functions are documented on page 90.)

170.13 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```

4076 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4077 {
4078   \if_int_compare:w
4079     \pdfTeX_strcmp:D { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4080     = \c_zero
4081     \prg_return_true:

```

```

4082     \else:
4083         \prg_return_false:
4084     \fi:
4085 }

```

(End definition for `\skip_if_eq:nm`. These functions are documented on page 90.)

`\skip_if_infinite_glue_p:n`
`\skip_if_infinite_glue:nTF`

With ε -TeX we all of a sudden get access to a lot of information we should otherwise consider ourselves lucky to get. One is the stretch and shrink components of a skip register and the order of those components. `\csskip_if_infinite_glue:nTF` tests it directly by looking at the stretch and shrink order. If either of the predicate functions return $\langle true \rangle$, `\bool_if:nTF` will return $\langle true \rangle$ and the logic test will take the true branch.

```

4086 \prg_new_conditional:Npnn \skip_if_infinite_glue:n #1 { p , T , F , TF }
4087 {
4088     \bool_if:nTF
4089     {
4090         \int_compare_p:nNn { \etex_gluestretchorder:D #1 } > \c_zero ||
4091         \int_compare_p:nNn { \etex_glueshrinkorder:D #1 } > \c_zero
4092     }
4093     { \prg_return_true: }
4094     { \prg_return_false: }
4095 }

```

(End definition for `\skip_if_infinite_glue:n`. These functions are documented on page 90.)

170.14 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

4096 \cs_new_nopar:Npn \skip_eval:n #1
4097 { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }

```

(End definition for `\skip_eval:n`. This function is documented on page 90.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

`\skip_use:c`

```

4098 \cs_new_eq:NN \skip_use:N \tex_the:D
4099 \cs_generate_variant:Nn \skip_use:N { c }

```

(End definition for `\skip_use:N` and `\skip_use:c`. These functions are documented on page 91.)

170.15 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

`\skip_horizontal:c`

`\skip_horizontal:n`

`\skip_vertical:N`

`\skip_vertical:c`

`\skip_vertical:n`

```

4100 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
4101 \cs_new_nopar:Npn \skip_horizontal:n #1

```

```

4102 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
4103 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
4104 \cs_new_nopar:Npn \skip_vertical:n #1
4105 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4106 \cs_generate_variant:Nn \skip_horizontal:N { c }
4107 \cs_generate_variant:Nn \skip_vertical:N { c }

```

(End definition for `\skip_horizontal:N`, `\skip_horizontal:c`, and `\skip_horizontal:n`. These functions are documented on page 94.)

170.16 Viewing skip variables

`\skip_show:N` Diagnostics.
`\skip_show:c`

```

4108 \cs_new_eq:NN \skip_show:N \kernel_register_show:N
4109 \cs_generate_variant:Nn \skip_show:N { c }

```

(End definition for `\skip_show:N` and `\skip_show:c`. These functions are documented on page 91.)

170.17 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions
`\c_max_skip`

```

4110 \cs_new_eq:NN \c_zero_skip \c_zero_dim
4111 \cs_new_eq:NN \c_max_skip \c_max_dim

```

(End definition for `\c_zero_skip`. This function is documented on page 91.)

170.18 Scratch skips

`\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`

```

4112 \skip_new:N \l_tmpa_skip
4113 \skip_new:N \l_tmpb_skip
4114 \skip_new:N \l_tmpc_skip
4115 \skip_new:N \g_tmpa_skip
4116 \skip_new:N \g_tmpb_skip

```

170.19 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.
`\muskip_new:c`

```

4117 <*initex>
4118 \alloc_new:nnnN { muskip } \c_zero \c_max_register_int \tex_muskipdef:D
4119 </initex>
4120 <*package>
4121 \cs_new_protected_nopar:Npn \muskip_new:N #1

```

```

4122 {
4123   \chk_if_free_cs:N #1
4124   \newmuskip #1
4125 }
4126 \</package>
4127 \cs_generate_variant:Nn \muskip_new:N { c }

```

(End definition for `\muskip_new:N` and `\muskip_new:c`. These functions are documented on page 92.)

`\muskip_zero:N` Reset the register to zero.

```

\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c
4128 \cs_new_protected_nopar:Npn \muskip_zero:N #1
4129 { #1 \c_zero_muskip }
4130 \cs_new_protected_nopar:Npn \muskip_gzero:N { \pref_global:D \muskip_zero:N }
4131 \cs_generate_variant:Nn \muskip_zero:N { c }
4132 \cs_generate_variant:Nn \muskip_gzero:N { c }

```

(End definition for `\muskip_zero:N` and `\muskip_zero:c`. These functions are documented on page 92.)

170.20 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.

```

\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn
4133 \cs_new_protected_nopar:Npn \muskip_set:Nn #1#2
4134 { #1 ~ \etex_muexpr:D #2 \scan_stop: }
4135 \cs_new_protected_nopar:Npn \muskip_gset:Nn { \pref_global:D \muskip_set:Nn }
4136 \cs_generate_variant:Nn \muskip_set:Nn { c }
4137 \cs_generate_variant:Nn \muskip_gset:Nn { c }

```

(End definition for `\muskip_set:Nn` and `\muskip_set:cn`. These functions are documented on page 92.)

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cN
\muskip_set_eq:Nc
\muskip_set_eq:cc
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc
4138 \cs_new_protected_nopar:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
4139 \cs_generate_variant:Nn \muskip_set_eq:NN { c }
4140 \cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
4141 \cs_new_protected_nopar:Npn \muskip_gset_eq:NN #1#2 { \pref_global:D #1 = #2 }
4142 \cs_generate_variant:Nn \muskip_gset_eq:NN { c }
4143 \cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }

```

(End definition for `\muskip_set_eq:NN` and others. These functions are documented on page 93.)

`\muskip_add:Nn` Using by here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn
4144 \cs_new_protected_nopar:Npn \muskip_add:Nn #1#2
4145 { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
4146 \cs_new_protected_nopar:Npn \muskip_gadd:Nn { \pref_global:D \muskip_add:Nn }
4147 \cs_generate_variant:Nn \muskip_add:Nn { c }
4148 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
4149 \cs_new_protected_nopar:Npn \muskip_sub:Nn #1#2

```



```

4150 { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
4151 \cs_new_protected_nopar:Npn \muskip_gsub:Nn { \pref_global:D \muskip_sub:Nn }
4152 \cs_generate_variant:Nn \muskip_sub:Nn { c }
4153 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for `\muskip_add:Nn` and `\muskip_add:cn`. These functions are documented on page 93.)

170.21 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

4154 \cs_new_nopar:Npn \muskip_eval:n #1
4155 { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }

```

(End definition for `\muskip_eval:n`. This function is documented on page 93.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

`\muskip_use:c`

```

4156 \cs_new_eq:NN \muskip_use:N \tex_the:D
4157 \cs_generate_variant:Nn \muskip_use:N { c }

```

(End definition for `\muskip_use:N` and `\muskip_use:c`. These functions are documented on page 93.)

170.22 Viewing muskip variables

`\muskip_show:N` Diagnostics.

`\muskip_show:c`

```

4158 \cs_new_eq:NN \muskip_show:N \kernel_register_show:N
4159 \cs_generate_variant:Nn \muskip_show:N { c }

```

(End definition for `\muskip_show:N` and `\muskip_show:c`. These functions are documented on page 94.)

170.23 Experimental skip functions

`\skip_split_finite_else_action:nnNN` This macro is useful when performing error checking in certain circumstances. If the $\langle skip \rangle$ 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.

```

4160 \cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
4161 {
4162   \skip_if_infinite_glue:nTF {#1}
4163   {
4164     #3 = \c_zero_skip
4165     #4 = \c_zero_skip
4166     #2
4167   }
4168   {

```

```

4169      #3 = \etex_gluestretch:D #1 \scan_stop:
4170      #4 = \etex_glueshrink:D #1 \scan_stop:
4171    }
4172  }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 95.)

```

4173 \</initex | package>

```

171 l3tl implementation

```

4174 \<*initex | package>
4175 \<*package>
4176 \ProvidesExplPackage
4177   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4178 \package_check_loaded_expl:
4179 \</package>

```

A token list variable is a $\text{T}_{\text{E}}\text{X}$ macro that holds tokens. By using the ε - $\text{T}_{\text{E}}\text{X}$ primitive `\unexpanded` inside a $\text{T}_{\text{E}}\text{X}$ `\edef` it is possible to store any tokens, including `#`, in this way.

171.1 Functions

`\tl_new:N` Creating new token list variables is a case of checking for an existing definition and if free doing the definition.
`\tl_new:c`

```

4180 \cs_new_protected_nopar:Npn \tl_new:N #1
4181 {
4182   \chk_if_free_cs:N #1
4183   \cs_gset_eq:NN #1 \c_empty_tl
4184 }
4185 \cs_generate_variant:Nn \tl_new:N { c }

```

(End definition for `\tl_new:N` and `\tl_new:c`. These functions are documented on page 96.)

`\tl_const:Nn` Constants are also easy to generate.

```

\<
\<
\<
\<
4186 \cs_new_protected:Npn \tl_const:Nn #1#2
4187 {
4188   \chk_if_free_cs:N #1
4189   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4190 }
4191 \cs_new_protected:Npn \tl_const:Nx #1#2
4192 {
4193   \chk_if_free_cs:N #1
4194   \cs_gset_nopar:Npx #1 {#2}
4195 }
4196 \cs_generate_variant:Nn \tl_const:Nn { c }
4197 \cs_generate_variant:Nn \tl_const:Nx { c }

```

(End definition for `\tl_const:Nn` and others. These functions are documented on page 96.)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

`\tl_clear:c`

`\tl_gclear:N`

`\tl_gclear:c`

```

4198 \cs_new_protected_nopar:Npn \tl_clear:N #1
4199 { \tl_set_eq:NN #1 \c_empty_tl }
4200 \cs_new_protected_nopar:Npn \tl_gclear:N #1
4201 { \tl_gset_eq:NN #1 \c_empty_tl }
4202 \cs_generate_variant:Nn \tl_clear:N { c }
4203 \cs_generate_variant:Nn \tl_gclear:N { c }

```

(End definition for `\tl_clear:N` and `\tl_clear:c`. These functions are documented on page 96.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

`\tl_clear_new:c`

`\tl_gclear_new:N`

`\tl_gclear_new:c`

```

4204 \cs_new_protected_nopar:Npn \tl_clear_new:N #1
4205 { \cs_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4206 \cs_new_protected_nopar:Npn \tl_gclear_new:N #1
4207 { \cs_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4208 \cs_generate_variant:Nn \tl_clear_new:N { c }
4209 \cs_generate_variant:Nn \tl_gclear_new:N { c }

```

(End definition for `\tl_clear_new:N` and `\tl_clear_new:c`. These functions are documented on page 96.)

`\tl_set_eq:NN` For setting token list variables equal to each other.

`\tl_set_eq:Nc`

`\tl_set_eq:cN`

`\tl_set_eq:Nc`

`\tl_set_eq:cc`

`\tl_gset_eq:NN`

`\tl_gset_eq:Nc`

`\tl_gset_eq:cN`

`\tl_gset_eq:cc`

```

4210 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
4211 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
4212 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
4213 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
4214 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4215 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
4216 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
4217 \cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc

```

(End definition for `\tl_set_eq:NN` and others. These functions are documented on page 97.)

171.2 Adding to token list variables

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain # tokens, which makes the token list registers provided by T_EX more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot.

`\tl_set:NV`

`\tl_set:Nv`

`\tl_set:No`

`\tl_set:Nf`

`\tl_set:Nx`

`\tl_set:cn`

`\tl_set:Nv`

`\tl_set:Nv`

`\tl_set:co`

`\tl_set:cf`

`\tl_set:cx`

`\tl_gset:Nn`

`\tl_gset:NV`

`\tl_gset:Nv`

`\tl_gset:No`

```

4218 \cs_new_protected:Npn \tl_set:Nn #1#2
4219 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4220 \cs_new_protected:Npn \tl_set:No #1#2

```

```

4221 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4222 \cs_new_protected:Npn \tl_set:Nx #1#2
4223 { \cs_set_nopar:Npx #1 {#2} }
4224 \cs_new_protected:Npn \tl_gset:Nn #1#2
4225 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4226 \cs_new_protected:Npn \tl_gset:No #1#2
4227 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
4228 \cs_new_protected:Npn \tl_gset:Nx #1#2
4229 { \cs_gset_nopar:Npx #1 {#2} }
4230 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
4231 \cs_generate_variant:Nn \tl_set:Nx { c }
4232 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
4233 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
4234 \cs_generate_variant:Nn \tl_gset:Nx { c }
4235 \cs_generate_variant:Nn \tl_gset:Nn { c , co , cV , cv , cf }

```

(End definition for `\tl_set:Nn` and others. These functions are documented on page 97.)

`\tl_put_left:Nn` Adding to the left is done directly to gain a little performance.

```

\tl_put_left:Nn
\tl_put_left:NV
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:co
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn
\tl_gput_left:cV
\tl_gput_left:co
\tl_gput_left:cx
4236 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4237 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4238 \cs_new_protected:Npn \tl_put_left:NV #1#2
4239 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4240 \cs_new_protected:Npn \tl_put_left:No #1#2
4241 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4242 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4243 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4244 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4245 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4246 \cs_new_protected:Npn \tl_gput_left:NV #1#2
4247 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4248 \cs_new_protected:Npn \tl_gput_left:No #1#2
4249 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4250 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4251 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4252 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4253 \cs_generate_variant:Nn \tl_put_left:NV { c }
4254 \cs_generate_variant:Nn \tl_put_left:No { c }
4255 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4256 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4257 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4258 \cs_generate_variant:Nn \tl_gput_left:No { c }
4259 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and others. These functions are documented on page 98.)

`\tl_put_right:Nn` The same on the right.

```

\tl_put_right:Nn
\tl_put_right:NV
\tl_put_right:No
\tl_put_right:Nx
\tl_put_right:cn
\tl_put_right:cV
\tl_put_right:co
\tl_put_right:cx
\tl_gput_right:Nn
\tl_gput_right:NV
\tl_gput_right:No
\tl_gput_right:Nx
\tl_gput_right:cn
4260 \cs_new_protected:Npn \tl_put_right:Nn #1#2

```

```

4261 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
4262 \cs_new_protected:Npn \tl_put_right:NV #1#2
4263 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
4264 \cs_new_protected:Npn \tl_put_right:No #1#2
4265 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
4266 \cs_new_protected:Npn \tl_put_right:Nx #1#2
4267 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
4268 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
4269 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
4270 \cs_new_protected:Npn \tl_gput_right:NV #1#2
4271 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
4272 \cs_new_protected:Npn \tl_gput_right:No #1#2
4273 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
4274 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4275 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4276 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4277 \cs_generate_variant:Nn \tl_put_right:NV { c }
4278 \cs_generate_variant:Nn \tl_put_right:No { c }
4279 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4280 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4281 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4282 \cs_generate_variant:Nn \tl_gput_right:No { c }
4283 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and others. These functions are documented on page 98.)

171.3 Reassigning token list category codes

`\c_tl_rescan_marker_tl` The rescanning code needs a special token list containing the same character with two different category codes. This is set up here, while the detail is described below.

```

4284 \group_begin:
4285 \tex_lccode:D '\A = '\@ \scan_stop:
4286 \tex_lccode:D '\B = '\@ \scan_stop:
4287 \tex_catcode:D '\A = 8 \scan_stop:
4288 \tex_catcode:D '\B = 3 \scan_stop:
4289 \tex_lowercase:D
4290 {
4291   \group_end:
4292   \tl_const:Nn \c_tl_rescan_marker_tl { A B }
4293 }

```

`\l_tl_rescan_tl` A token list variable to actually store the material being processed.

```

4294 \tl_new:N \l_tl_rescan_tl

```

`\tl_set_rescan:Nnn` The idea here is to deal cleanly with the problem that `\tex_scantokens:D` treats the argument as a file, and without the correct settings a T_EX error occurs:

`\tl_set_rescan:Nno`

`\tl_set_rescan:cnn`

`\tl_set_rescan:cno`

`\tl_gset_rescan:Nnn`

`\tl_gset_rescan:Nno`

`\tl_gset_rescan:cnn`

`\tl_gset_rescan:cno`

`\tl_set_rescan_aux:Nnn`

`\tl_rescan_aux:w`

! File ended while scanning definition of ...

When expanding a token list this can be handled using `\exp_not:N` but this fails if the token list is not being expanded. So instead a delimited argument is used with an end marker which cannot appear within the token list which is scanned: two `@` symbols with different category codes. The rescanned token list cannot contain the end marker, because all `@` present in the token list are read with the same category code. The setting of `\tex_endlinechar:D` is needed to avoid introducing an extraneous space in the result.

```

4295 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
4296 { \tl_set_rescan_aux:NNnn \tl_set:Nn }
4297 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
4298 { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
4299 \cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4
4300 {
4301   \group_begin:
4302     \exp_args:No \etex_everyeof:D { \c_tl_rescan_marker_tl }
4303     \tex_endlinechar:D \c_minus_one
4304     #3
4305     \tl_clear:N \l_tl_rescan_tl
4306     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
4307     \exp_args:NNNo \group_end:
4308     #1 #2 \l_tl_rescan_tl
4309   }
4310 \cs_new_nopar:Npx \tl_rescan_aux:w
4311 {
4312   \cs_set_protected:Npn \exp_not:N \tl_rescan_aux:w ##1
4313     \c_tl_rescan_marker_tl
4314     { \tl_set:Nn \exp_not:N \l_tl_rescan_tl {##1} }
4315 }
4316 \tl_rescan_aux:w
4317 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno }
4318 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno }
4319 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno }
4320 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }

```

(End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 100.)

```

\tl_set_rescan:Nnx
\tl_set_rescan:cnx
\tl_gset_rescan:Nnx
\tl_gset_rescan:cnx
\tl_set_rescan_aux:NNnx

```

With x-type expansion the `\tex_everyeof:D` method does apply and the code is simple.

```

4321 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnx
4322 { \tl_set_rescan_aux:NNnx \tl_set:Nn }
4323 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnx
4324 { \tl_set_rescan_aux:NNnx \tl_gset:Nn }
4325 \cs_new_protected_nopar:Npn \tl_set_rescan_aux:NNnx #1#2#3#4
4326 {
4327   \group_begin:
4328     \etex_everyeof:D { \exp_not:N }
4329     \tex_endlinechar:D \c_minus_one

```

```

4330     #3
4331     \tl_set:Nx \l_tl_rescan_tl { \etex_scantokens:D {#4} }
4332     \exp_args:NNNo \group_end:
4333     #1 #2 \l_tl_rescan_tl
4334 }
4335 \cs_generate_variant:Nn \tl_set_rescan:Nnx { c }
4336 \cs_generate_variant:Nn \tl_gset_rescan:Nnx { c }

```

(End definition for `\tl_set_rescan:Nnx` and `\tl_set_rescan:cnx`. These functions are documented on page 100.)

`\tl_rescan:nn` The same idea is also applied to in line token lists.

```

4337 \cs_new_protected:Npn \tl_rescan:nn #1#2
4338 {
4339   \group_begin:
4340   \exp_args:No \etex_veryeof:D { \c_tl_rescan_marker_tl }
4341   \tex_endlinechar:D \c_minus_one
4342   #1
4343   \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
4344   \exp_args:No \group_end:
4345   \l_tl_rescan_tl
4346 }

```

(End definition for `\tl_rescan:nn`. This function is documented on page 100.)

171.4 Reassigning token list character codes

`\tl_to_lowercase:n` Just some names for a few primitives.

`\tl_to_uppercase:n`

```

4347 \cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
4348 \cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D

```

(End definition for `\tl_to_lowercase:n`. This function is documented on page 101.)

171.5 Modifying token list variables

`\l_tl_replace_tl` A scratch variable for doing token replacement.

```

4349 \tl_new:N \l_tl_replace_tl

```

`\tl_replace_once:Nnn` 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
`\tl_replace_once:cn` the last token is the marker there is nothing to do, otherwise replace the token and clean
`\tl_greplace_once:Nnn` up (hence the second use of `\tl_tmp:w`). To prevent losing braces or spaces there are
`\tl_greplace_once:cn` a couple of empty groups and the strange-looking `\use:n`. There is a `\q_nil` between
`\tl_replace_once_aux:NNnn` the original and the search input, to prevent cases where the end of the original and the
start of the search run together to give an erroneous test result.

```

4350 \cs_new_protected_nopar:Npn \tl_replace_once:Nnn
4351 { \tl_replace_once_aux:NNnn \tl_set_eq:NN }
4352 \cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
4353 { \tl_replace_once_aux:NNnn \tl_gset_eq:NN }
4354 \cs_new_protected:Npn \tl_replace_once_aux:NNnn #1#2#3#4
4355 {
4356   \cs_set_protected:Npx \tl_tmp:w ##1 #3 ##2 \q_stop
4357   {
4358     \exp_not:N \quark_if_no_value:nF {##2}
4359     {
4360       \tl_set:No \exp_not:N \l_tl_replace_tl { ##1 \exp_not:n{#4} }
4361       \exp_not:n
4362       {
4363         \cs_set_protected:Npn \tl_tmp:w ##1 \q_nil #3 \q_no_value
4364         { \tl_put_right:No \l_tl_replace_tl {##1} }
4365       }
4366       \exp_not:n { \tl_tmp:w \prg_do_nothing: } ##2
4367       \exp_not:n { #1 #2 \l_tl_replace_tl }
4368     }
4369   }
4370   \exp_after:wN \tl_tmp:w \exp_after:wN \prg_do_nothing:
4371   #2 \q_nil #3 \q_no_value \q_stop
4372 }
4373 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
4374 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }

```

(End definition for `\tl_replace_once:Nnn` and `\tl_replace_once:cnn`. These functions are documented on page 99.)

```

\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
\tl_replace_all_aux:NNnn

```

A similar approach here but with a loop built in.

```

4375 \cs_new_protected_nopar:Npn \tl_replace_all:Nnn
4376 { \tl_replace_all_aux:NNnn \tl_set_eq:NN }
4377 \cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
4378 { \tl_replace_all_aux:NNnn \tl_gset_eq:NN }
4379 \cs_new_protected:Npn \tl_replace_all_aux:NNnn #1#2#3#4
4380 {
4381   \tl_clear:N \l_tl_replace_tl
4382   \cs_set_protected:Npx \tl_tmp:w ##1 #3 ##2 \q_stop
4383   {
4384     \exp_not:N \quark_if_no_value:nTF {##2}
4385     {
4386       \exp_not:n
4387       {
4388         \cs_set_protected:Npn \tl_tmp:w ##1 \q_nil ##2 \q_stop
4389         { \tl_put_right:No \l_tl_replace_tl {##1} }
4390       }
4391       \exp_not:N \tl_tmp:w ##1 \exp_not:N \q_stop
4392     }
4393   }

```



```

4394         \exp_not:n { \tl_put_right:No \l_tl_replace_tl }
4395         { ##1 \exp_not:n{#4} }
4396         \exp_not:n { \tl_tmp:w \prg_do_nothing: } ##2 \exp_not:N \q_stop
4397     }
4398 }
4399 \exp_after:wN \tl_tmp:w \exp_after:wN \prg_do_nothing:
4400 #2 \q_nil #3 \q_no_value \q_stop
4401 #1 #2 \l_tl_replace_tl
4402 }
4403 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
4404 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and `\tl_replace_all:cnn`. These functions are documented on page 99.)

`\tl_remove_once:Nn` Removal is just a special case of replacement.

```

\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
4405 \cs_new_protected_nopar:Npn \tl_remove_once:Nn #1#2
4406 { \tl_replace_once:Nnn #1 {#2} { } }
4407 \cs_new_protected_nopar:Npn \tl_gremove_once:Nn #1#2
4408 { \tl_greplace_once:Nnn #1 {#2} { } }
4409 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
4410 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }

```

(End definition for `\tl_remove_once:Nn` and `\tl_remove_once:cn`. These functions are documented on page 99.)

`\tl_remove_all:Nn` Removal is just a special case of replacement.

```

\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
4411 \cs_new_protected_nopar:Npn \tl_remove_all:Nn #1#2
4412 { \tl_replace_all:Nnn #1 {#2} { } }
4413 \cs_new_protected_nopar:Npn \tl_gremove_all:Nn #1#2
4414 { \tl_greplace_all:Nnn #1 {#2} { } }
4415 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
4416 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

171.6 Token list conditionals

`\tl_if_blank_p:n` `\tl_if_blank_p:V` `\tl_if_blank_p:o` `\tl_if_blank:nTF` `\tl_if_blank:VTF` `\tl_if_blank:oTF` `\tl_if_blank_p_aux:NNw` TeX skips spaces when reading a non-delimited arguments. Thus, a *token list* is blank if and only if `\use_none:n <token list> ?` is empty. For performance reasons, we hard-code the emptiness test done in `\tl_if_empty:n(TF)`: convert to harmless characters with `\tl_to_str:n`, and then use `\if_meaning:w \q_nil ... \q_nil`. Note that converting to a string is done after reading the delimited argument for `\use_none:n`. The similar construction `\exp_after:wN \use_none:n \tl_to_str:n {<token list>} ?` would fail if the token list contains the control sequence `\`, while `\tex_escapechar:D` is a space or is unprintable.

```

4417 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
4418 { \tl_if_empty_return:o { \use_none:n #1 ? } }

```

```

4419 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
4420 \cs_generate_variant:Nn \tl_if_blank:nT { V }
4421 \cs_generate_variant:Nn \tl_if_blank:nF { V }
4422 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
4423 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
4424 \cs_generate_variant:Nn \tl_if_blank:nT { o }
4425 \cs_generate_variant:Nn \tl_if_blank:nF { o }
4426 \cs_generate_variant:Nn \tl_if_blank:nTF { o }

```

(End definition for `\tl_remove_all:Nn` and `\tl_remove_all:cn`. These functions are documented on page 101.)

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

`\tl_if_empty_p:c`
`\tl_if_empty:NTF`
`\tl_if_empty:cTF`

```

4427 \prg_set_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
4428 {
4429   \if_meaning:w #1 \c_empty_tl
4430   \prg_return_true:
4431   \else:
4432     \prg_return_false:
4433   \fi:
4434 }
4435 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
4436 \cs_generate_variant:Nn \tl_if_empty:NT { c }
4437 \cs_generate_variant:Nn \tl_if_empty:NF { c }
4438 \cs_generate_variant:Nn \tl_if_empty:NTF { c }

```

(End definition for `\tl_if_empty:N` and `\tl_if_empty:c`. These functions are documented on page 101.)

`\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.

`\tl_if_empty_p:V`
`\tl_if_empty:nTF`
`\tl_if_empty:VTF`

```

4439 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p,TF,T,F } {
4440   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4441   \prg_return_true:
4442   \else:
4443     \prg_return_false:
4444   \fi:
4445 }
4446 \cs_generate_variant:Nn \tl_if_empty_p:n { V }
4447 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
4448 \cs_generate_variant:Nn \tl_if_empty:nT { V }
4449 \cs_generate_variant:Nn \tl_if_empty:nF { V }

```

(End definition for `\tl_if_empty:n` and `\tl_if_empty:V`. These functions are documented on page 102.)

`\tl_if_empty_p:o` The auxiliary function `\tl_if_empty_return:o` is for use in conditionals on token lists, which mostly reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:n(TF)`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note that this works because `\tl_to_str:n` expands tokens that follow until reading a catcode 1 (begin-group) token.

```

4450 \cs_new:Npn \tl_if_empty_return:o #1 {
4451   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4452   \tl_to_str:n \exp_after:wN {#1} \q_nil
4453   \prg_return_true:
4454   \else:
4455   \prg_return_false:
4456   \fi:
4457 }
4458 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p,TF,T,F}
4459 { \tl_if_empty_return:o {#1} }
```

(End definition for `\tl_if_empty:o`. These functions are documented on page 102.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` if and only if the two token list variables are equal.

```

4460 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
4461 {
4462   \if_meaning:w #1 #2
4463   \prg_return_true:
4464   \else:
4465   \prg_return_false:
4466   \fi:
4467 }
4468 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
4469 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
4470 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
4471 \cs_generate_variant:Nn \tl_if_eq:NNF { Nc , c , cc }
```

(End definition for `\tl_if_eq:NN` and others. These functions are documented on page 102.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

4472 \l_tl_tmpa_tl
4473 \l_tl_tmpb_tl
4474 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
4475 {
4476   \group_begin:
4477   \tl_set:Nn \l_tl_tmpa_tl {#1}
4478   \tl_set:Nn \l_tl_tmpb_tl {#2}
4479   \if_meaning:w \l_tl_tmpa_tl \l_tl_tmpb_tl
4480   \group_end:
4481   \prg_return_true:
4482 }
```

```

4480     \else:
4481         \group_end:
4482         \prg_return_false:
4483     \fi:
4484 }
4485 \tl_new:N \l_tl_tmpa_tl
4486 \tl_new:N \l_tl_tmpb_tl

```

(End definition for `\tl_if_eq:nn`. This function is documented on page ??.)

`\tl_if_in:NnTF` See `\tl_if_in:nn(TF)` for further comments. Here we simply expand the token list
`\tl_if_in:cnTF` variable and pass it to `\tl_if_in:nn(TF)`.

```

4487 \cs_new_protected_nopar:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4488 \cs_new_protected_nopar:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4489 \cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4490 \cs_generate_variant:Nn \tl_if_in:NnT { c }
4491 \cs_generate_variant:Nn \tl_if_in:NnF { c }
4492 \cs_generate_variant:Nn \tl_if_in:NnTF { c }

```

(End definition for `\tl_if_in:Nn` and `\tl_if_in:cn`. These functions are documented on page 102.)

`\tl_if_in:nnTF` Once more, the test relies on `\tl_to_str:n` for robustness. The function `\tl_tmp:w`
`\tl_if_in:VnTF` removes tokens until the first occurrence of #2. If this does not appear in #1, then the
`\tl_if_in:onTF` final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the
`\tl_if_in:noTF` test is false. See `\tl_if_empty:n(TF)` for details on the emptiness test.

Special care is needed to treat correctly cases like `\tl_if_in:nnTF {a state}{states}`, where #1#2 contains #2 before the end. To cater for this case, we insert `{}``{}` between the two token lists. This marker may not appear in #2 because of T_EX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments.

```

4493 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
4494 {
4495     \cs_set:Npn \tl_tmp:w ##1 #2 { }
4496     \tl_if_empty:oTF { \tl_tmp:w #1 {} {} #2 }
4497     { \prg_return_false: } { \prg_return_true: }
4498 }
4499 \cs_generate_variant:Nn \tl_if_in:nnT { V }
4500 \cs_generate_variant:Nn \tl_if_in:nnF { V }
4501 \cs_generate_variant:Nn \tl_if_in:nnTF { V }
4502 \cs_generate_variant:Nn \tl_if_in:nnT { o }
4503 \cs_generate_variant:Nn \tl_if_in:nnF { o }
4504 \cs_generate_variant:Nn \tl_if_in:nnTF { o }
4505 \cs_generate_variant:Nn \tl_if_in:nnT { no }
4506 \cs_generate_variant:Nn \tl_if_in:nnF { no }
4507 \cs_generate_variant:Nn \tl_if_in:nnTF { no }

```

(End definition for `\tl_if_in:nn` and others. These functions are documented on page ??.)

`\tl_if_single_p:n` If the argument is a single token, or a single brace group preceeded by optional (explicit) spaces, or a non-zero number of spaces.

```

\tl_if_single_aux:n
4508 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4509 {
4510   \tl_if_blank:nTF {#1}
4511     { \tl_if_empty:nTF {#1} { \prg_return_false: } { \prg_return_true: } }
4512     { \tl_if_single_aux:n {#1} }
4513 }
4514 \prg_new_conditional:Npnn \tl_if_single:N #1 { p , T , F , TF }
4515 {
4516   \tl_if_blank:oTF #1
4517     { \tl_if_empty:NTF #1 { \prg_return_false: } { \prg_return_true: } }
4518     { \exp_args:No \tl_if_single_aux:n {#1} }
4519 }
4520 \cs_new:Npn \tl_if_single_aux:n #1
4521 { \tl_if_empty_return:o { \use_none:n #1 } }

```

(End definition for `\tl_if_single:n`. These functions are documented on page 103.)

171.7 Mapping to token lists

`\tl_map_function:nN` Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

```

\tl_map_function_aux:N
4522 \cs_new:Npn \tl_map_function:nN #1#2
4523 { \tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop }
4524 \cs_new_nopar:Npn \tl_map_function:NN #1#2
4525 {
4526   \exp_after:wN \tl_map_function_aux:Nn
4527   \exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop
4528 }
4529 \cs_new:Npn \tl_map_function_aux:Nn #1#2
4530 {
4531   \quark_if_recursion_tail_stop:n {#2}
4532   #1 {#2} \tl_map_function_aux:Nn #1
4533 }
4534 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for `\tl_map_function:nN`. This function is documented on page 103.)

`\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. (`\g_tl_inline_level_int` is defined in `l3int` for order-of-loading reasons.)

```

\tl_map_inline_aux:n
\g_tl_inline_level_int
4535 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4536 {

```

```

4537 \int_gincr:N \g_tl_inline_level_int
4538 \cs_gset:cpn { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4539 ##1 {#2}
4540 \exp_args:Nc \tl_map_function_aux:Nn
4541 { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4542 #1 \q_recursion_tail \q_recursion_stop
4543 \int_gdecr:N \g_tl_inline_level_int
4544 }
4545 \cs_new_protected:Npn \tl_map_inline:Nn #1#2
4546 {
4547 \int_gincr:N \g_tl_inline_level_int
4548 \cs_gset:cpn { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4549 ##1 {#2}
4550 \exp_last_unbraced:NcV \tl_map_function_aux:Nn
4551 { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4552 #1 \q_recursion_tail \q_recursion_stop
4553 \int_gdecr:N \g_tl_inline_level_int
4554 }
4555 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End definition for `\tl_map_inline:nn`. This function is documented on page ??.)

```

\tl_map_variable:nNn \tl_map_variable:nNn <token list> <temp> <action> assigns <temp> to each element and
\tl_map_variable:NNn executes <action>.
\tl_map_variable:cNn
\tl_map_variable_aux:NnN
4556 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
4557 { \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop }
4558 \cs_new_protected_nopar:Npn \tl_map_variable:NNn
4559 { \exp_args:No \tl_map_variable:nNn }
4560 \cs_new_protected:Npn \tl_map_variable_aux:Nnn #1#2#3
4561 {
4562 \tl_set:Nn #1 {#3}
4563 \quark_if_recursion_tail_stop:N #1
4564 #2 \tl_map_variable_aux:Nnn #1 {#2}
4565 }
4566 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`. This function is documented on page 104.)

`\tl_map_break:` The break statement.

```

4567 \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 104.)

171.8 Using token lists

`\tl_to_str:n` Another name for a primitive.

```

4568 \cs_new_eq:NN \tl_to_str:n \etex_detokenize:D

```

(End definition for `\tl_to_str:n`. This function is documented on page 105.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

`\tl_to_str:c`

```
4569 \cs_new_nopar:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
4570 \cs_generate_variant:Nn \tl_to_str:N { c }
```

(End definition for `\tl_to_str:N` and `\tl_to_str:c`. These functions are documented on page 105.)

`\tl_use:N` Token lists which are simply not defined will give a clear T_EX error here. No such luck

`\tl_use:c` for ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error

`\tl_error_message:` is forced.

```
4571 \cs_new_eq:NN \tl_use:N \prg_do_nothing:
4572 \cs_new_nopar:Npn \tl_use:c #1
4573 {
4574   \if_cs_exist:w #1 \cs_end:
4575   \cs:w #1 \exp_after:wN \cs_end:
4576   \else:
4577     \exp_after:wN \tl_error_message:
4578   \fi:
4579 }
4580 \group_begin:
4581 \tex_catcode:D ‘\! = 11 \scan_stop:
4582 \tex_catcode:D ‘\ = 11\scan_stop:%
4583 \cs_gset_nopar:Npn\tl_error_message:{\undefined variable name!}%
4584 \group_end:%
```

(End definition for `\tl_use:N` and `\tl_use:c`. These functions are documented on page 105.)

171.9 Working with the contents of token lists

`\tl_length:n` Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. `\tl_length_aux:n` grabs the element and replaces it by +1. The 0 to ensure it works on an empty list.

`\tl_length:V`

`\tl_length:o`

`\tl_length:N`

`\tl_length:c`

`\tl_length_aux:n`

```
4585 \cs_new:Npn \tl_length:n #1
4586 {
4587   \int_eval:n
4588   { 0 \tl_map_function:nN {#1} \tl_length_aux:n }
4589 }
4590 \cs_new_nopar:Npn \tl_length:N #1
4591 {
4592   \int_eval:n
4593   { 0 \tl_map_function:NN #1 \tl_length_aux:n }
4594 }
4595 \cs_new:Npn \tl_length_aux:n #1 { + 1 }
4596 \cs_generate_variant:Nn \tl_length:n { V , o }
4597 \cs_generate_variant:Nn \tl_length:N { c }
```

(End definition for `\tl_length:n`, `\tl_length:V`, and `\tl_length:o`. These functions are documented on page 106.)

`\tl_reverse:n` Reversal of a token list is done by taking one token at a time and putting it in front of
`\tl_reverse:V` the ones before it.
`\tl_reverse:o`

```
\tl_reverse_aux:nN 4598 \cs_new:Npn \tl_reverse:n #1
4599 { \tl_reverse_aux:nN { } #1 \q_recursion_tail \q_recursion_stop }
4600 \cs_new:Npn \tl_reverse_aux:nN #1#2
4601 {
4602   \quark_if_recursion_tail_stop_do:nn {#2} {#1}
4603   \tl_reverse_aux:nN { #2 #1 }
4604 }
4605 \cs_generate_variant:Nn \tl_reverse:n {V,o}
```

(End definition for `\tl_reverse:n`, `\tl_reverse:V`, and `\tl_reverse:o`. These functions are documented on page 106.)

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which in turn is removed by the `f`
`\tl_reverse:c` expansion which comes to a halt.

```
4606 \cs_new_protected_nopar:Npn \tl_reverse:N #1
4607 { \tl_set:Nf #1 { \tl_reverse:o { #1 \exp_stop_f: } } }
4608 \cs_generate_variant:Nn \tl_reverse:N { c }
```

(End definition for `\tl_reverse:N` and `\tl_reverse:c`. These functions are documented on page 106.)

`\tl_trim_spaces:n` Trimming spaces from around the input uses the idea of a sufficiently odd token to allow
`\tl_trim_spaces:N` for a delimited argument to do this. Here, the standard approach [a Q with category code
`\tl_trim_spaces:c` 3 (math toggle)] is used. The `\etex_unexpanded:D` here is used so that space trimming
`\tl_gtrim_spaces:N` will behave correctly within an x-type expansion.
`\tl_gtrim_spaces:c`

```
\tl_trim_spaces_exp:n 4609 \cs_new:Npn \tl_trim_spaces:n #1
\tl_trim_spaces_aux_i:w 4610 {
\tl_trim_spaces_aux_ii:w 4611   \etex_unexpanded:D \exp_after:wN \exp_after:wN \exp_after:wN
4612   { \tl_trim_spaces_exp:n {#1} }
4613 }
4614 \cs_new_protected:Npn \tl_trim_spaces:N #1
4615 { \tl_set:Nx #1 { \exp_after:wN \tl_trim_spaces:n \exp_after:wN {#1} } }
4616 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4617 { \tl_gset:Nx #1 { \exp_after:wN \tl_trim_spaces:n \exp_after:wN {#1} } }
4618 \group_begin:
4619   \tex_catcode:D '\Q = 3 \scan_stop:
4620   \cs_new:Npn \tl_trim_spaces_exp:n #1
4621   { \tex_romannumeral:D - '0 \tl_trim_spaces_aux_i:w \exp_not:N #1 Q ~ Q }
4622   \cs_new:Npn \tl_trim_spaces_aux_i:w #1 ~ Q { \tl_trim_spaces_aux_ii:w #1 Q }
4623   \cs_new:Npn \tl_trim_spaces_aux_ii:w #1 Q #2 {#1}
4624 \group_end:
4625 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4626 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
```

(End definition for `\tl_trim_spaces:n`. This function is documented on page 106.)

171.10 The first token from a token list

`\tl_head:n` These functions pick up either the head or the tail of a list.

```

\tl_head:V
\tl_head:v
\tl_head:f
\tl_head:w
\tl_tail:n
\tl_tail:V
\tl_tail:v
\tl_tail:f
\tl_tail:w

```

```

4627 \cs_new:Npn \tl_head:n #1 { \tl_head:w #1 \q_stop }
4628 \cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
4629 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4630 \cs_new:Npn \tl_tail:w #1#2 \q_stop {#2}
4631 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4632 \cs_generate_variant:Nn \tl_tail:n { V , v , f }

```

(End definition for `\tl_head:n` and others. These functions are documented on page 107.)

```

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF

```

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_charcode_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.

```

4633 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
4634 {
4635   \exp_after:wN \if_meaning:w \tl_head:w #1 \q_stop #2
4636   \prg_return_true:
4637   \else:
4638     \prg_return_false:
4639   \fi:
4640 }

```

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.

```

4641 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4642 {
4643   \exp_after:wN \if:
4644   \exp_after:wN \exp_not:N \tl_head:w #1 \q_stop \exp_not:N #2
4645   \prg_return_true:
4646   \else:
4647     \prg_return_false:
4648   \fi:
4649 }
4650 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
4651 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
4652 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
4653 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF { f }

```

And now catcodes:

```

4654 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1#2 { p , T , F , TF }

```

```

4655 {
4656   \exp_after:wN \if_catcode:w \exp_after:wN \exp_not:N
4657   \tl_head:w #1 \q_stop \exp_not:N #2
4658   \prg_return_true: \else: \prg_return_false: \fi:
4659 }

```

(End definition for `\tl_if_head_eq_meaning:nN`. These functions are documented on page 107.)

171.11 Viewing token lists

`\tl_show:N` Showing token list variables is done directly: at the moment do not worry if they are defined.
`\tl_show:c`

```

4660 \cs_new_protected:Npn \tl_show:N #1 { \cs_show:N #1 }
4661 \cs_generate_variant:Nn \tl_show:N { c }

```

(End definition for `\tl_show:N` and `\tl_show:c`. These functions are documented on page 108.)

`\tl_show:n` For literal token lists, life is easy.

```

4662 \cs_new_eq:NN \tl_show:n \etex_showtokens:D

```

(End definition for `\tl_show:n`. This function is documented on page 108.)

171.12 Constant token lists

`\c_job_name_tl` Inherited from the L^AT_EX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course.

```

4663 <*package>
4664 \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
4665 </package>
4666 <*initex>
4667 \tex_everyjob:D \exp_after:wN
4668 {
4669   \tex_the:D \tex_everyjob:D
4670   \luatex_if_engine:TF
4671   {
4672     \tl_if_in:onTF { \tex_jobname:D } { ~ }
4673     { \tl_const:Nx \c_job_name_tl { " \tex_jobname:D " } }
4674     { \tl_const:Nx \c_job_name_tl { \tex_jobname:D } }
4675   }
4676   { \tl_const:Nx \c_job_name_tl { \tex_jobname:D } }
4677 }
4678 </initex>

```

`\c_empty_tl` Never full.

```

4679 \tl_const:Nn \c_empty_tl { }

```

`\c_space_tl` A space as a token list (as opposed to as a character).

```
4680 \tl_const:Nn \c_space_tl { ~ }
```

171.13 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
4681 \tl_new:N \g_tmpa_tl
4682 \tl_new:N \g_tmpb_tl
```

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
4683 \tl_new:N \l_tmpa_tl
4684 \tl_new:N \l_tmpb_tl
```

171.14 Experimental functions

`\tl_if_single_item_p:n` A token list has exactly one item if it is either a single token surrounded by optional explicit spaces, or a single brace group surrounded by optional explicit spaces. The naive version of this test would do `\use_none:n #1`, and test if the result is empty. However, this will fail when the token list is empty. Furthermore, it does not allow optional trailing spaces.

```
4685 \cs_new:Npn \tl_if_single_item:nTF #1
4686 { \str_if_eq:onTF { \use_ii:nn #1 {?} ? } {??} }
4687 \cs_new:Npn \tl_if_single_item:nT #1
4688 { \str_if_eq:onT { \use_ii:nn #1 {?} ? } {??} }
4689 \cs_new:Npn \tl_if_single_item:nF #1
4690 { \str_if_eq:onF { \use_ii:nn #1 {?} ? } {??} }
4691 \cs_new:Npn \tl_if_single_item_p:n #1
4692 { \str_if_eq_p:on { \use_ii:nn #1 {?} ? } {??} }
```

(End definition for `\tl_if_single_item:n`. These functions are documented on page 109.)

`\tl_if_head_begin_group_p:n`
`\tl_if_head_begin_group:nTF`

```
4693 \prg_new_conditional:Npnn \tl_if_head_begin_group:n #1 { p , T , F , TF }
4694 {
4695   \exp_after:wN \use_none:n
4696   \exp_after:wN {
4697     \exp_after:wN {
4698       \token_to_str:N #1 .
4699     }

```

```

4700     \prg_return_true: \exp_after:wN \use_none:nn \token_to_str:N
4701   }
4702   \prg_return_false:
4703 }

```

(End definition for `\tl_if_head_begin_group:n`. These functions are documented on page 109.)

`\tl_if_head_eq_space_p:n` If the first token of the token list is an explicit space, i.e., a character token with character code 32 and category code 10, then this test will be `<true>`. It is `<false>` if the token list is empty, if the first token is an implicit space token, such as `\c_space_token`, or any token other than an explicit space.

`\tl_if_head_eq_space:nTF`

`\tl_if_head_eq_space_aux:w`

```

4704 \prg_new_conditional:Npnn \tl_if_head_eq_space:n #1 { p , T , F , TF }
4705 {
4706   \if_false: { \fi:
4707     \tl_if_head_eq_space_aux:w \prg_do_nothing: #1 ? ~ }
4708   }
4709 \cs_new:Npn \tl_if_head_eq_space_aux:w #1 ~ %
4710 {
4711   \tl_if_empty_return:o { #1 }
4712   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4713 }

```

(End definition for `\tl_if_head_eq_space:n`. These functions are documented on page 109.)

`\tl_if_single_token_p:n` If the token list starts with a space, then it is a single token if and only if its string representation is exactly one space character. Otherwise, we check if removing the first token group makes the token list empty. If it doesn't, the token list is not a single token. If it does, then the token list is either a single token or a single brace group. This is tested for using `\tl_if_head_begin_group:n`.

`\tl_if_single_token:nTF`

```

4714 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
4715 {
4716   \tl_if_head_eq_space:nTF {#1}
4717   {
4718     \str_if_eq:nnTF {#1} {~}
4719     \prg_return_true: \prg_return_false:
4720   }
4721   {
4722     \str_if_eq:onTF { \use_none:n #1 ? } {?}
4723     {
4724       \tl_if_head_begin_group:nTF {#1}
4725       \prg_return_false: \prg_return_true:
4726     }
4727     \prg_return_false:
4728   }
4729 }

```

(End definition for `\tl_if_single_token:n`. These functions are documented on page 110.)

171.15 Deprecated functions

`\tl_new:Nn` Use either `\tl_const:Nn` or `\tl_new:N`.

```
\tl_new:cn
\tl_new:Nx
4730 \cs_new_protected:Npn \tl_new:Nn #1#2
4731 {
4732     \tl_new:N #1
4733     \tl_gset:Nn #1 {#2}
4734 }
4735 \cs_generate_variant:Nn \tl_new:Nn { c }
4736 \cs_generate_variant:Nn \tl_new:Nn { Nx }
```

(End definition for `\tl_new:Nn`, `\tl_new:cn`, and `\tl_new:Nx`. These functions are documented on page ??.)

`\tl_gset:Nc` This was useful once, but nowadays does not make much sense.

```
\tl_set:Nc
4737 \cs_new_protected_nopar:Npn \tl_gset:Nc
4738 { \pref_global:D \tl_set:Nc }
4739 \cs_new_protected_nopar:Npn \tl_set:Nc #1#2
4740 { \tl_set:No #1 { \cs:w #2 \cs_end: } }
```

(End definition for `\tl_gset:Nc`. This function is documented on page ??.)

`\tl_replace_in:Nnn` These are renamed.

```
\tl_replace_in:cnn
\tl_greplace_in:Nnn
\tl_greplace_in:cnn
4741 \cs_new_eq:NN \tl_replace_in:Nnn \tl_replace_once:Nnn
4742 \cs_new_eq:NN \tl_replace_in:cnn \tl_replace_once:cnn
4743 \cs_new_eq:NN \tl_greplace_in:Nnn \tl_greplace_once:Nnn
4744 \cs_new_eq:NN \tl_greplace_in:cnn \tl_greplace_once:cnn
4745 \cs_new_eq:NN \tl_replace_all_in:Nnn \tl_replace_all:Nnn
4746 \cs_new_eq:NN \tl_replace_all_in:cnn \tl_replace_all:cnn
4747 \cs_new_eq:NN \tl_greplace_all_in:Nnn \tl_greplace_all:Nnn
4748 \cs_new_eq:NN \tl_greplace_all_in:cnn \tl_greplace_all:cnn
```

(End definition for `\tl_replace_in:Nnn` and `\tl_replace_in:cnn`. These functions are documented on page ??.)

`\tl_remove_in:Nn` Also renamed.

```
\tl_remove_in:cn
\tl_gremove_in:Nn
\tl_gremove_in:cn
4749 \cs_new_eq:NN \tl_remove_in:Nn \tl_remove_once:Nn
4750 \cs_new_eq:NN \tl_remove_in:cn \tl_remove_once:cn
4751 \cs_new_eq:NN \tl_gremove_in:Nn \tl_gremove_once:Nn
4752 \cs_new_eq:NN \tl_gremove_in:cn \tl_gremove_once:cn
4753 \cs_new_eq:NN \tl_remove_all_in:Nn \tl_remove_all:Nn
4754 \cs_new_eq:NN \tl_remove_all_in:cn \tl_remove_all:cn
4755 \cs_new_eq:NN \tl_gremove_all_in:Nn \tl_gremove_all:Nn
4756 \cs_new_eq:NN \tl_gremove_all_in:cn \tl_gremove_all:cn
```

(End definition for `\tl_remove_in:Nn` and `\tl_remove_in:cn`. These functions are documented on page ??.)

`\tl_elt_count:n` Another renaming job.

```

\tl_elt_count:V
\tl_elt_count:o 4757 \cs_new_eq:NN \tl_elt_count:n \tl_length:n
\tl_elt_count:N 4758 \cs_new_eq:NN \tl_elt_count:V \tl_length:V
\tl_elt_count:c 4759 \cs_new_eq:NN \tl_elt_count:o \tl_length:o
4760 \cs_new_eq:NN \tl_elt_count:N \tl_length:N
4761 \cs_new_eq:NN \tl_elt_count:c \tl_length:c

```

(End definition for `\tl_elt_count:n`, `\tl_elt_count:V`, and `\tl_elt_count:o`. These functions are documented on page ??.)

`\tl_head_i:n` Two renames, and a few that are rather too specialised.

```

\tl_head_i:w
\tl_head_iii:n 4762 \cs_new_eq:NN \tl_head_i:n \tl_head:n
\tl_head_iii:f 4763 \cs_new_eq:NN \tl_head_i:w \tl_head:w
\tl_head_iii:w 4764 \cs_new:Npn \tl_head_iii:n #1 { \tl_head_iii:w #1 \q_stop }
4765 \cs_generate_variant:Nn \tl_head_iii:n { f }
4766 \cs_new:Npn \tl_head_iii:w #1#2#3#4 \q_stop {#1#2#3}

```

(End definition for `\tl_head_i:n`. This function is documented on page ??.)

```
4767 </initex | package>
```

172 l3seq implementation

The following test files are used for this code: `m3seq002,m3seq003`.

```

4768 <*initex | package>

4769 <*package>
4770 \ProvidesExplPackage
4771   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4772 \package_check_loaded_expl:
4773 </package>

```

A sequence is a control sequence whose top-level expansion is of the form “`\seq_item:n {⟨item0⟩} ... \seq_item:n {⟨itemn-1⟩}`”. 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 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.

```

4774 \cs_new:Npn \seq_item:n
4775 {
4776   \seq_use_error:
4777   \use_none:n
4778 }

```

(End definition for `\seq_item:n`. This function is documented on page 122.)

`\l_seq_tmpa_tl` Scratch space for various internal uses.
`\l_seq_tmpb_tl`

4779 `\tl_new:N \l_seq_tmpa_tl`
4780 `\tl_new:N \l_seq_tmpb_tl`

172.1 Allocation and initialisation

`\seq_new:N` Internally, sequences are just token lists.
`\seq_new:c`

4781 `\cs_new_eq:NN \seq_new:N \tl_new:N`
4782 `\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 110.)

`\seq_clear:N` Clearing sequences is just the same as clearing token lists.
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c`

4783 `\cs_new_eq:NN \seq_clear:N \tl_clear:N`
4784 `\cs_new_eq:NN \seq_clear:c \tl_clear:c`
4785 `\cs_new_eq:NN \seq_gclear:N \tl_gclear:N`
4786 `\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 111.)

`\seq_clear_new:N` Once again a copy from the token list functions.
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

4787 `\cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N`
4788 `\cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c`
4789 `\cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N`
4790 `\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 111.)

`\seq_set_eq:NN` Once again, these are simple 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`

4791 `\cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN`
4792 `\cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc`
4793 `\cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN`
4794 `\cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc`
4795 `\cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN`
4796 `\cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc`
4797 `\cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN`
4798 `\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 111.)

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

Concatenating sequences is easy.

```

4799 \cs_new_protected_nopar:Npn \seq_concat:NNN #1#2#3
4800 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4801 \cs_new_protected_nopar:Npn \seq_gconcat:NNN #1#2#3
4802 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4803 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
4804 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_concat:ccc`. These functions are documented on page 112.)

172.2 Appending data to either end

`\seq_put_left:Nn`
`\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_gput_left:Nv`
`\seq_gput_left:Nv`
`\seq_gput_left:Nx`
`\seq_gput_left:No`
`\seq_gput_left:No`
`\seq_gput_left:cV`
`\seq_gput_left:cV`
`\seq_gput_left:cV`
`\seq_gput_left:co`
`\seq_gput_left:cx`
`\seq_gput_right:Nn`
`\seq_gput_right:Nv`
`\seq_gput_right:Nv`
`\seq_gput_right:No`
`\seq_gput_right:Nx`
`\seq_gput_right:cn`
`\seq_gput_right:cV`
`\seq_gput_right:cV`
`\seq_remove_duplicates:N`
`\seq_remove_duplicates:c`
`\seq_remove_duplicates:aux:NN`

The code here is just a wrapper for adding to token lists.

```

4805 \cs_new_protected:Npn \seq_put_left:Nn #1#2
4806 { \tl_put_left:Nn #1 { \seq_item:n {#2} } }
4807 \cs_new_protected:Npn \seq_put_right:Nn #1#2
4808 { \tl_put_right:Nn #1 { \seq_item:n {#2} } }
4809 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
4810 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
4811 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
4812 \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 113.)

The same for global addition.

```

4813 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
4814 { \tl_gput_left:Nn #1 { \seq_item:n {#2} } }
4815 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
4816 { \tl_gput_right:Nn #1 { \seq_item:n {#2} } }
4817 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
4818 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
4819 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
4820 \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 113.)

172.3 Modifying sequences

`\seq_remove_duplicates:N`
`\seq_remove_duplicates:c`
`\seq_remove_duplicates:aux:NN`

An internal sequence for the removal routines.

```

4821 \seq_new:N \l_seq_remove_seq

```

Removing duplicates means making a new list then copying it.

```

4822 \cs_new_protected:Npn \seq_remove_duplicates:N

```



```

4823 { \seq_remove_duplicates_aux:NN \seq_set_eq:NN }
4824 \cs_new_protected:Npn \seq_gremove_duplicates:N
4825 { \seq_remove_duplicates_aux:NN \seq_gset_eq:NN }
4826 \cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2
4827 {
4828   \seq_clear:N \l_seq_remove_seq
4829   \seq_map_inline:Nn #2
4830   {
4831     \seq_if_in:NnF \l_seq_remove_seq {##1}
4832     { \seq_put_right:Nn \l_seq_remove_seq {##1} }
4833   }
4834   #1 #2 \l_seq_remove_seq
4835 }
4836 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
4837 \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 115.)

```

\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.

```

4838 \cs_new_protected:Npn \seq_remove_all:Nn
4839 { \seq_remove_all_aux:NNn \tl_set:Nx }
4840 \cs_new_protected:Npn \seq_gremove_all:Nn
4841 { \seq_remove_all_aux:NNn \tl_gset:Nx }
4842 \cs_new_protected:Npn \seq_remove_all_aux:NNn #1#2#3
4843 {
4844   \seq_push_item_def:n
4845   {
4846     \str_if_eq:nnT {##1} {#3}
4847     {
4848       \if_false: { \fi: }
4849       \tl_set:Nn \l_seq_tmpb_tl {##1}
4850       #1 #2
4851       { \if_false: } \fi:
4852       \exp_not:o {#2}
4853       \tl_if_eq:NNT \l_seq_tmpa_tl \l_seq_tmpb_tl
4854       { \use_none:nn }
4855     }
4856     \exp_not:n { \seq_item:n {##1} }

```

```

4857     }
4858     \tl_set:Nn \l_seq_tmpa_tl {#3}
4859     #1 #2 {#2}
4860     \seq_pop_item_def:
4861   }
4862   \cs_generate_variant:Nn \seq_remove_all:Nn { c }
4863   \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 115.)

172.4 Sequence conditionals

`\seq_if_empty_p:N` Simple copies from the token list variable material.

```

\seq_if_empty_p:c
\seq_if_empty:NTF
\seq_if_empty:cTF
4864 \prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N
4865 { p , T , F , TF }
4866 \prg_new_eq_conditional:NNn \seq_if_empty:c \tl_if_empty:c
4867 { p , T , F , TF }

```

(End definition for `\seq_if_empty:N` and `\seq_if_empty:c`. These functions are documented on page 115.)

`\seq_if_in:NnTF` The approach here is to define `\seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\prg_return_true:` is inserted. On the other hand, if there is no match then the loop will break returning `\prg_return_false:`. In either case, `\seq_break_point:n` ensures that the group ends before the logical value is returned. Everything is inside a group so that `\seq_item:n` is preserved in nested situations.

```

\seq_if_in:NvTF
\seq_if_in:NoTF
\seq_if_in:NxTF
\seq_if_in:cnTF
\seq_if_in:cVTF
\seq_if_in:cvTF
\seq_if_in:coTF
\seq_if_in:cxTF
\seq_if_in_aux:
4868 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
4869 { T , F , TF }
4870 {
4871   \group_begin:
4872     \tl_set:Nn \l_seq_tmpa_tl {#2}
4873     \cs_set_protected:Npn \seq_item:n ##1
4874     {
4875       \tl_set:Nn \l_seq_tmpb_tl {##1}
4876       \if_meaning:w \l_seq_tmpa_tl \l_seq_tmpb_tl
4877         \exp_after:wN \seq_if_in_aux:
4878       \fi:
4879     }
4880     #1
4881     \seq_break:n { \prg_return_false: }
4882     \seq_break_point:n { \group_end: }
4883   }
4884   \cs_new_nopar:Npn \seq_if_in_aux: { \seq_break:n { \prg_return_true: } }
4885   \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
4886   \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }

```

```

4887 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
4888 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
4889 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }
4890 \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 116.)

172.5 Recovering data from sequences

`\seq_get_left:NN` `\seq_get_left:cN` Getting an item from the left of a sequence is pretty easy: just trim off the first item after removing the `\seq_item:n` at the start.

```

\seq_get_left_aux:NnwN
4891 \cs_new_protected_nopar:Npn \seq_get_left:NN #1#2
4892 {
4893   \seq_if_empty_err_break:N #1
4894   \exp_after:wN \seq_get_left_aux:NnwN #1 \q_stop #2
4895   \seq_break_point:n { }
4896 }
4897 \cs_new_protected:Npn \seq_get_left_aux:NnwN \seq_item:n #1#2 \q_stop #3
4898 { \tl_set:Nn #3 {#1} }
4899 \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 119.)

`\seq_pop_left:NN` `\seq_pop_left:cN` `\seq_gpop_left:NN` `\seq_gpop_left:cN` The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```

\seq_pop_left_aux:NNN
\seq_pop_left_aux:NnwNNN
4900 \cs_new_protected_nopar:Npn \seq_pop_left:NN
4901 { \seq_pop_left_aux:NNN \tl_set:Nn }
4902 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
4903 { \seq_pop_left_aux:NNN \tl_gset:Nn }
4904 \cs_new_protected_nopar:Npn \seq_pop_left_aux:NNN #1#2#3
4905 {
4906   \seq_if_empty_err_break:N #2
4907   \exp_after:wN \seq_pop_left_aux:NnwNNN #2 \q_stop #1#2#3
4908   \seq_break_point:n { }
4909 }
4910 \cs_new_protected:Npn \seq_pop_left_aux:NnwNNN \seq_item:n #1#2 \q_stop #3#4#5
4911 {
4912   #3 #4 {#2}
4913   \tl_set:Nn #5 {#1}
4914 }
4915 \cs_generate_variant:Nn \seq_pop_left:NN { c }
4916 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and `\seq_gpop_left:cN`. These functions are documented on page 120.)

`\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`.

```

4917 \cs_new_protected_nopar:Npn \seq_get_right:NN #1#2
4918 {
4919   \seq_if_empty_err_break:N #1
4920   \seq_get_right_aux:NN #1#2
4921   \seq_break_point:n { }
4922 }
4923 \cs_new_protected_nopar:Npn \seq_get_right_aux:NN #1#2
4924 {
4925   \seq_push_item_def:n { }
4926   \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
4927   \exp_after:wN \use_none:n #1
4928   { \tl_set:Nn #2 }
4929   { }
4930   {
4931     \seq_pop_item_def:
4932     \seq_break:
4933   }
4934 }
4935 \cs_new:Npn \seq_get_right_loop:nn #1#2
4936 {
4937   #2 {#1}
4938   \seq_get_right_loop:nn
4939 }
4940 \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 119.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{ \if_false:} \fi: ... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding” definition for `\seq_item:n`, the left-most $n - 1$ entries in a sequence of n items will be stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` 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.

```

4941 \cs_new_protected_nopar:Npn \seq_pop_right:NN
4942 { \seq_pop_right_aux:NNN \tl_set:Nx }
4943 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
4944 { \seq_pop_right_aux:NNN \tl_gset:Nx }

```

```

4945 \cs_new_protected_nopar:Npn \seq_pop_right_aux:NNN #1#2#3
4946 {
4947   \seq_if_empty_err_break:N #2
4948   \seq_pop_right_aux_ii:NNN #1 #2 #3
4949   \seq_break_point:n { }
4950 }
4951 \cs_new_protected_nopar:Npn \seq_pop_right_aux_ii:NNN #1#2#3
4952 {
4953   \seq_push_item_def:n { \exp_not:n { \seq_item:n {##1} } }
4954   #1 #2 { \if_false: } \fi:
4955   \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
4956   \exp_after:wN \use_none:n #2
4957   {
4958     \if_false: { \fi: }
4959     \tl_set:Nn #3
4960   }
4961   { }
4962   {
4963     \seq_pop_item_def:
4964     \seq_break:
4965   }
4966 }
4967 \cs_generate_variant:Nn \seq_pop_right:NN { c }
4968 \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 120.)

172.6 Mapping to sequences

`\seq_break:` 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.

```

4969 \cs_new:Npn \seq_break: #1 \seq_break_point:n #2 {#2}
4970 \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 123.)

`\seq_map_break:` Semantically-logical copies of the break functions for use inside mappings.
`\seq_map_break:n`

```

4971 \cs_new_eq:NN \seq_map_break: \seq_break:
4972 \cs_new_eq:NN \seq_map_break:n \seq_break:n

```

(End definition for `\seq_map_break:.` This function is documented on page 117.)

`\seq_break_point:n` Normally, the marker token will not be executed, but if it is then the end code is simply inserted.

```

4973 \cs_new_eq:NN \seq_break_point:n \use:n

```

(End definition for `\seq_break_point:n`. This function is documented on page 123.)

`\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.

```

4974 \cs_new_protected_nopar:Npn \seq_if_empty_err_break:N #1
4975 {
4976   \if_meaning:w #1 \c_empty_tl
4977     \msg_kernel_error:nxx { seq } { empty-sequence } { \token_to_str:N #1 }
4978     \exp_after:wN \seq_break:
4979   \fi:
4980 }

```

(End definition for `\seq_if_empty_err_break:N`. This function is documented on page 122.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering
`\seq_map_function:cN` the definition of `\seq_item:n`. This is done as by noting that every odd token in the
`\seq_map_function_aux:NNn` 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.

```

4981 \cs_new:Npn \seq_map_function:NN #1#2
4982 {
4983   \exp_after:wN \seq_map_function_aux:NNn \exp_after:wN #2 #1
4984   { ? \seq_map_break: } { }
4985   \seq_break_point:n { }
4986 }
4987 \cs_new:Npn \seq_map_function_aux:NNn #1#2#3
4988 {
4989   \use_none:n #2
4990   #1 {#3}
4991   \seq_map_function_aux:NNn #1
4992 }
4993 \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 116.)

`\g_seq_nesting_depth_int` A counter to keep track of nested functions: defined in `l3int`.

`\seq_push_item_def:n` The definition of `\seq_item:n` needs to be saved and restored at various points within
`\seq_push_item_def:x` the mapping and manipulation code. That is handled here: as always, this approach uses
`\seq_push_item_def_aux:` global assignments.

`\seq_pop_item_def:`

```

4994 \cs_new_protected:Npn \seq_push_item_def:n
4995 {
4996   \seq_push_item_def_aux:
4997   \cs_gset:Npn \seq_item:n ##1
4998 }

```

```

4999 \cs_new_protected:Npn \seq_push_item_def:x
5000 {
5001   \seq_push_item_def_aux:
5002   \cs_gset:Npx \seq_item:n ##1
5003 }
5004 \cs_new_protected:Npn \seq_push_item_def_aux:
5005 {
5006   \cs_gset_eq:cN { seq_item_ \int_use:N \g_seq_nesting_depth_int :n }
5007   \seq_item:n
5008   \int_gincr:N \g_seq_nesting_depth_int
5009 }
5010 \cs_new_protected_nopar:Npn \seq_pop_item_def:
5011 {
5012   \int_gdecr:N \g_seq_nesting_depth_int
5013   \cs_gset_eq:Nc \seq_item:n
5014   { seq_item_ \int_use:N \g_seq_nesting_depth_int :n }
5015 }

```

(End definition for `\seq_push_item_def:n` and `\seq_push_item_def:x`. These functions are documented on page 122.)

`\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`.

```

5016 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
5017 {
5018   \seq_push_item_def:n {#2}
5019   #1
5020   \seq_break_point:n { \seq_pop_item_def: }
5021 }
5022 \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 116.)

`\seq_map_variable:NNn` This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.
`\seq_map_variable:Ncn`
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

```

5023 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
5024 {
5025   \seq_push_item_def:x
5026   {
5027     \tl_set:Nn \exp_not:N #2 {##1}
5028     \exp_not:n {#3}
5029   }
5030   #1
5031   \seq_break_point:n { \seq_pop_item_def: }
5032 }
5033 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
5034 \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 116.)

172.7 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` 5035 `\cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn`

`\seq_push:No` 5036 `\cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv`

`\seq_push:Nx` 5037 `\cs_new_eq:NN \seq_push:No \seq_put_left:No`

`\seq_push:cn` 5038 `\cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx`

`\seq_push:cV` 5039 `\cs_new_eq:NN \seq_push:cn \seq_put_left:cn`

`\seq_push:cV` 5040 `\cs_new_eq:NN \seq_push:cV \seq_put_left:cV`

`\seq_push:co` 5041 `\cs_new_eq:NN \seq_push:cv \seq_put_left:cv`

`\seq_push:cx` 5042 `\cs_new_eq:NN \seq_push:co \seq_put_left:co`

`\seq_gpush:Nn` 5043 `\cs_new_eq:NN \seq_gpush:cx \seq_put_left:cx`

`\seq_gpush:NV` 5044 `\cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn`

`\seq_gpush:Nv` 5045 `\cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv`

`\seq_gpush:No` 5046 `\cs_new_eq:NN \seq_gpush:No \seq_gput_left:No`

`\seq_gpush:Nx` 5047 `\cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx`

`\seq_gpush:cn` 5048 `\cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn`

`\seq_gpush:cV` 5049 `\cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV`

`\seq_gpush:cv` 5050 `\cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv`

`\seq_gpush:co` 5051 `\cs_new_eq:NN \seq_gpush:co \seq_gput_left:co`

`\seq_gpush:cx` 5052 `\cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx`

5053 `\cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx`

5054

(End definition for `\seq_push:Nn` and others. These functions are documented on page 119.)

`\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` 5055 `\cs_new_eq:NN \seq_get:NN \seq_get_left:NN`

`\seq_gpop:NN` 5056 `\cs_new_eq:NN \seq_get:cN \seq_get_left:cN`

`\seq_gpop:cN` 5057 `\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN`

5058 `\cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN`

5059 `\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN`

5060 `\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 118.)

172.8 Viewing sequences

`\l_seq_show_tl` Used to store the material for display.

5061 `\tl_new:N \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”.

```

5062 \cs_new_protected_nopar:Npn \seq_show:N #1
5063 {
5064   \seq_if_empty:NTF #1
5065   {
5066     \iow_term:x { Sequence~\token_to_str:N #1 \c_space_tl is-empty }
5067     \tl_show:n { }
5068   }
5069   {
5070     \iow_term:x
5071     {
5072       Sequence~\token_to_str:N #1 \c_space_tl
5073       contains~the~items~(without~outer~braces):
5074     }
5075     \tl_set:Nx \l_seq_show_tl
5076     { \seq_map_function:NN #1 \seq_show_aux:n }
5077     \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
5078     { \exp_after:wN \seq_show_aux:w \l_seq_show_tl }
5079   }
5080 }
5081 \cs_new:Npn \seq_show_aux:n #1
5082 {
5083   \iow_newline: > \c_space_tl \c_space_tl
5084   \iow_char:N \{ \exp_not:n {#1} \iow_char:N \}
5085 }
5086 \cs_new:Npn \seq_show_aux:w #1 > ~ { }
5087 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page 119.)

172.9 Experimental functions

`\seq_if_empty_break_return_false:N` The name says it all: if the sequence is empty, returns logical `false`.

```

5088 \cs_new_nopar:Npn \seq_if_empty_break_return_false:N #1
5089 {
5090   \if_meaning:w #1 \c_empty_tl
5091   \prg_return_false:
5092   \exp_after:wN \seq_break:
5093   \fi:
5094 }

```

(End definition for `\seq_if_empty_break_return_false:N`.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results.
`\seq_get_left:cNTF`
`\seq_get_right:NNTF`
`\seq_get_right:cNTF`

```

5095 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1 #2 { T , F , TF }
5096 {
5097   \seq_if_empty_break_return_false:N #1
5098   \exp_after:wN \seq_get_left_aux:Nw #1 \q_stop #2
5099   \prg_return_true:
5100   \seq_break:
5101   \seq_break_point:n { }
5102 }
5103 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
5104 {
5105   \seq_if_empty_break_return_false:N #1
5106   \seq_get_right_aux:NN #1#2
5107   \prg_return_true: \seq_break:
5108   \seq_break_point:n { }
5109 }
5110 \cs_generate_variant:Nn \seq_get_left:NNT { c }
5111 \cs_generate_variant:Nn \seq_get_left:NNF { c }
5112 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
5113 \cs_generate_variant:Nn \seq_get_right:NNT { c }
5114 \cs_generate_variant:Nn \seq_get_right:NNF { c }
5115 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for `\seq_get_left:NN` and `\seq_get_left:cN`. These functions are documented on page 119.)

`\seq_pop_left:NNTF`
`\seq_pop_left:cNTF`
`\seq_gpop_left:NNTF`
`\seq_gpop_left:cNTF`
`\seq_pop_right:NNTF`
`\seq_pop_right:cNTF`
`\seq_gpop_right:NNTF`
`\seq_gpop_right:cNTF`

More or less the same for popping.

```

5116 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
5117 {
5118   \seq_if_empty_break_return_false:N #1
5119   \exp_after:wN \seq_pop_left_aux:NwNNN #1 \q_stop \tl_set:Nn #1#2
5120   \prg_return_true: \seq_break:
5121   \seq_break_point:n { }
5122 }
5123 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
5124 {
5125   \seq_if_empty_break_return_false:N #1
5126   \exp_after:wN \seq_pop_left_aux:NwNNN #1 \q_stop \tl_gset:Nn #1#2
5127   \prg_return_true: \seq_break:
5128   \seq_break_point:n { }
5129 }
5130 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
5131 {
5132   \seq_if_empty_break_return_false:N #1
5133   \seq_pop_right_aux_ii:NNN \tl_set:Nx #1 #2
5134   \prg_return_true: \seq_break:
5135   \seq_break_point:n { }
5136 }
5137 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
5138 {

```

```

5139 \seq_if_empty_break_return_false:N #1
5140 \seq_pop_right_aux_ii:NNN \tl_gset:Nx #1 #2
5141 \prg_return_true: \seq_break:
5142 \seq_break_point:n { }
5143 }
5144 \cs_generate_variant:Nn \seq_pop_left:NNT { c }
5145 \cs_generate_variant:Nn \seq_pop_left:NNF { c }
5146 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
5147 \cs_generate_variant:Nn \seq_gpop_left:NNT { c }
5148 \cs_generate_variant:Nn \seq_gpop_left:NNF { c }
5149 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
5150 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
5151 \cs_generate_variant:Nn \seq_pop_right:NNF { c }
5152 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
5153 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
5154 \cs_generate_variant:Nn \seq_gpop_right:NNF { c }
5155 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for `\seq_pop_left:MN` and `\seq_pop_left:cN`. These functions are documented on page 120.)

`\seq_length:N` Counting the items in a sequence is done using the same approach as for other length functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.
`\seq_length:c`
`\seq_length_aux:n`

```

5156 \cs_new:Npn \seq_length:N #1
5157 {
5158   \int_eval:n
5159   {
5160     0
5161     \seq_map_function:NN #1 \seq_length_aux:n
5162   }
5163 }
5164 \cs_new:Npn \seq_length_aux:n #1 { +1 }
5165 \cs_generate_variant:Nn \seq_length:N { c }

```

(End definition for `\seq_length:N` and `\seq_length:c`. These functions are documented on page 120.)

`\seq_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the stop code `{ ? \seq_break } { }` will be used by the auxiliary, terminating the loop and returning nothing at all.
`\seq_item:cn`
`\seq_item_aux:nnn`

```

5166 \cs_new_nopar:Npn \seq_item:Nn #1#2
5167 {
5168   \exp_last_unbraced:Nfo \seq_item_aux:nnn
5169   {
5170     \int_eval:n
5171     {
5172       \int_compare:nNnT {#2} < \c_zero
5173       { \seq_length:N #1 + }

```

```

5174         #2
5175     }
5176 }
5177 #1
5178 { ? \seq_break: }
5179 { }
5180 \seq_break_point:n { }
5181 }
5182 \cs_new_nopar:Npn \seq_item_aux:nnn #1#2#3
5183 {
5184     \use_none:n #2
5185     \int_compare:nNnTF {#1} = \c_zero
5186     { \seq_break:n {#3} }
5187     { \exp_args:Nf \seq_item_aux:nnn { #1 - 1 } }
5188 }
5189 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and `\seq_item:cn`. These functions are documented on page 121.)

`\seq_use:N` A simple short cut for a mapping.

`\seq_use:c`

```

5190 \cs_new_nopar:Npn \seq_use:N #1 { \seq_map_function:NN #1 \use:n }
5191 \cs_generate_variant:Nn \seq_use:N { c }

```

(End definition for `\seq_use:N` and `\seq_use:c`. These functions are documented on page 121.)

`\seq_mapthread_function:NNN`
`\seq_mapthread_function:NcN`
`\seq_mapthread_function:cNN`
`\seq_mapthread_function:ccN`

The idea here is to first expand both of the sequences, adding the usual `{ ? \seq_break: } { }` to the end of each on. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first token of #2 and #5, which for items in the sequences will both be `\seq_item:n`. The function to be mapped will then be applied to the two entries. When the code hits the end of one of the sequences, the break material will stop the entire loop and tidy up. This avoids needing to find the length of the two sequences, or worrying about which is longer.

`\seq_mapthread_function_aux:NN`
`\seq_mapthread_function_aux:Nnnwnn`

```

5192 \cs_new_nopar:Npn \seq_mapthread_function:NNN #1#2#3
5193 {
5194     \exp_after:wN \seq_mapthread_function_aux:NN
5195     \exp_after:wN #3
5196     \exp_after:wN #1
5197     #2
5198     { ? \seq_break: } { }
5199     \seq_break_point:n { }
5200 }
5201 \cs_new_nopar:Npn \seq_mapthread_function_aux:NN #1#2
5202 {
5203     \exp_after:wN \seq_mapthread_function_aux:Nnnwnn
5204     \exp_after:wN #1
5205     #2
5206     { ? \seq_break: } { }
5207     \q_stop

```

```

5208 }
5209 \cs_new:Npn \seq_mapthread_function_aux:Nnnwnn #1#2#3#4 \q_stop #5#6
5210 {
5211   \use_none:n #2
5212   \use_none:n #5
5213   #1 {#3} {#6}
5214   \seq_mapthread_function_aux:Nnnwnn #1 #4 \q_stop
5215 }
5216 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }
5217 \cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }

```

(End definition for `\seq_mapthread_function:NNN` and others. These functions are documented on page 121.)

```

\seq_set_from_clist:NN
\seq_set_from_clist:cN
\seq_set_from_clist:Nc
\seq_set_from_clist:cc
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:cN
\seq_gset_from_clist:Nc
\seq_gset_from_clist:cc
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn
\seq_wrap_item:n

```

Setting a sequence from a comma-separated list is done using a simple mapping.

```

5218 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
5219 {
5220   \tl_set:Nx #1
5221   { \clist_map_function:NN #2 \seq_wrap_item:n }
5222 }
5223 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
5224 {
5225   \tl_set:Nx #1
5226   { \clist_map_function:nN {#2} \seq_wrap_item:n }
5227 }
5228 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
5229 {
5230   \tl_gset:Nx #1
5231   { \clist_map_function:NN #2 \seq_wrap_item:n }
5232 }
5233 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
5234 {
5235   \tl_gset:Nx #1
5236   { \clist_map_function:nN {#2} \seq_wrap_item:n }
5237 }
5238 \cs_new:Npn \seq_wrap_item:n #1 { \exp_not:n { \seq_item:n {#1} } }
5239 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
5240 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
5241 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
5242 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
5243 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
5244 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c }

```

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 122.)

172.10 Deprecated 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`

```
5245 \cs_new_eq:NN \seq_top:NN \seq_get_left:NN
5246 \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`

```
5247 \cs_new_eq:NN \seq_display:N \seq_show:N
5248 \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 ??.)

```
5249 \</initex | package>
```

173 l3clist implementation

The following test files are used for this code: `m3clist002`.

```
5250 \<*initex | package>
```

```
5251 \<*package>
```

```
5252 \ProvidesExplPackage
```

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

```
5253 \package_check_loaded_expl:
```

```
5254 \</package>
```

`\l_clist_tmpa_tl` Scratch space for various internal uses.

`\l_clist_tmpb_tl`

```
5256 \tl_new:N \l_clist_tmpa_tl
```

```
5257 \tl_new:N \l_clist_tmpb_tl
```

173.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

`\clist_new:c`

```
5258 \cs_new_eq:NN \clist_new:N \tl_new:N
```

```
5259 \cs_new_eq:NN \clist_new:c \tl_new:c
```

(End definition for `\clist_new:N` and `\clist_new:c`. These functions are documented on page 123.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

`\clist_clear:c`

`\clist_gclear:N`

`\clist_gclear:c`

```
5260 \cs_new_eq:NN \clist_clear:N \tl_clear:N
```

```
5261 \cs_new_eq:NN \clist_clear:c \tl_clear:c
```

```
5262 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
```

```
5263 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(End definition for `\clist_clear:N` and `\clist_clear:c`. These functions are documented on page 124.)

```
\clist_clear_new:N Once again a copy from the token list functions.
\clist_clear_new:c
\clist_gclear_new:N 5264 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:c 5265 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
                    5266 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
                    5267 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(End definition for `\clist_clear_new:N` and `\clist_clear_new:c`. These functions are documented on page 124.)

```
\clist_set_eq:NN Once again, these are simple copies from the token list functions.
\clist_set_eq:cN
\clist_set_eq:Nc 5268 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:cc 5269 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_gset_eq:NN 5270 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:cN 5271 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:Nc 5272 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:cc 5273 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 5274 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\clist_gset_eq:cc 5275 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\clist_set_eq:NN` and others. These functions are documented on page 124.)

```
\clist_concat:NNN Concatenating sequences is not quite as easy as it seems, as there is the danger that #1
\clist_concat:ccc may be the same as either #2 or #3. Also, there needs to be the correct addition of a
\clist_gconcat:NNN comma to the output. So a little work to do.
\clist_gconcat:ccc
\clist_concat_aux:NNNN 5276 \cs_new_protected_nopar:Npn \clist_concat:NNN
                    5277 { \clist_concat_aux:NNNN \tl_set:Nx }
                    5278 \cs_new_protected_nopar:Npn \clist_gconcat:NNN
                    5279 { \clist_concat_aux:NNNN \tl_gset:Nx }
                    5280 \cs_new_protected_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4
                    5281 {
                    5282   #1 #2
                    5283   {
                    5284     \clist_if_empty:NTF #3
                    5285     { \exp_not:o #4 }
                    5286     {
                    5287       \exp_not:o #3
                    5288       \clist_if_empty:NF #4
                    5289       {
                    5290         ,
                    5291         \exp_not:o #4
                    5292       }
                    5293     }
                    5294   }
                    5295 }
                    5296 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
                    5297 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(End definition for `\clist_concat:NNN` and `\clist_concat:ccc`. These functions are documented on page 125.)

173.2 Appending items to comma lists

`\clist_put_left:Nn`
`\clist_put_left:NV`
`\clist_put_left:No`
`\clist_put_left:Nx`
`\clist_put_left:cn`
`\clist_put_left:cV`
`\clist_put_left:co`
`\clist_put_left:cx`
`\clist_gput_left:Nn`
`\clist_gput_left:NV`
`\clist_gput_left:No`
`\clist_gput_left:Nx`
`\clist_gput_left:cn`
`\clist_gput_left:cV`
`\clist_gput_left:co`
`\clist_gput_left:cx`

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```
5298 \cs_new_protected_nopar:Npn \clist_put_left:Nn
5299 { \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn { } , }
5300 \cs_new_protected_nopar:Npn \clist_gput_left:Nn
5301 { \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_left:Nn { } , }
5302 \cs_new_protected:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6
5303 {
5304   \clist_if_empty:NTF #5
5305     { #1 #5 {#6} }
5306     { \tl_if_empty:nF {#6} { #2 #5 { #3 #6 #4 } } }
5307 }
5308 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
5309 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
5310 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
5311 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
```

(End definition for `\clist_put_left:Nn` and others. These functions are documented on page 126.)

`\clist_put_right:Nn`
`\clist_put_right:NV`
`\clist_put_right:No`
`\clist_put_right:Nx`
`\clist_put_right:cn`
`\clist_put_right:cV`
`\clist_put_right:co`
`\clist_put_right:cx`
`\clist_gput_right:Nn`
`\clist_gput_right:NV`
`\clist_gput_right:No`
`\clist_gput_right:Nx`
`\clist_gput_right:cn`
`\clist_gput_right:cV`
`\clist_gput_right:co`
`\clist_gput_right:cx`
`\clist_get:Nn`
`\clist_get:NV`
`\clist_get:No`
`\clist_get:Nx`
`\clist_get:cn`
`\clist_get:cV`
`\clist_get:co`
`\clist_get:cx`

The same for the right side.

```
5312 \cs_new_protected:Npn \clist_put_right:Nn
5313 { \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn , { } }
5314 \cs_new_protected_nopar:Npn \clist_gput_right:Nn
5315 { \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , { } }
5316 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
5317 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
5318 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
5319 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
```

(End definition for `\clist_put_right:Nn` and others. These functions are documented on page 126.)

173.3 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

`\clist_get_aux:wN`

```
5320 \cs_new_protected_nopar:Npn \clist_get:NN #1#2
5321 { \exp_after:wN \clist_get_aux:wN #1 , \q_stop #2 }
5322 \cs_new_protected:Npn \clist_get_aux:wN #1 , #2 \q_stop #3
5323 { \tl_set:Nn #3 {#1} }
5324 \cs_generate_variant:Nn \clist_get:NN { c }
```


(End definition for `\clist_get:NN` and `\clist_get:cN`. These functions are documented on page 132.)

`\clist_pop:NN` The aim here is to get the popped item as #1 in the auxiliary, with #2 containing either
`\clist_pop:cN` the remainder of the list or `\q_nil` if there were insufficient items. That keeps the
`\clist_gpop:NN` number of auxiliary functions down.
`\clist_gpop:cN`

```

5325 \cs_new_protected_nopar:Npn \clist_pop:NN
5326   { \clist_pop_aux:NNN \tl_set:Nn }
5327 \cs_new_protected_nopar:Npn \clist_gpop:NN
5328   { \clist_pop_aux:NNN \tl_gset:Nn }
5329 \cs_new_protected_nopar:Npn \clist_pop_aux:NNN #1#2#3
5330   { \exp_after:wN \clist_pop_aux:wNNN #2 , \q_nil , \q_nil , \q_stop #1#2#3 }
5331 \cs_new_protected:Npn \clist_pop_aux:wNNN #1 , #2 , \q_nil , #3 \q_stop #4#5#6
5332   {
5333     \quark_if_nil:nTF {#2}
5334       { #4 #5 { } }
5335       { #4 #5 {#2} }
5336     \tl_set:Nn #6 {#1}
5337   }
5338 \cs_generate_variant:Nn \clist_pop:NN { c }
5339 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and `\clist_pop:cN`. These functions are documented on page 132.)

`\clist_push:Nn` Pushing to a sequence is the same as adding on the left.
`\clist_push:NV`
`\clist_push:No`
`\clist_push:Nx`
`\clist_push:cn`
`\clist_push:cV`
`\clist_push:co`
`\clist_push:cx`
`\clist_gpush:Nn`
`\clist_gpush:NV`
`\clist_gpush:No`
`\clist_gpush:Nx`
`\clist_gpush:cn`
`\clist_gpush:cV`
`\clist_gpush:co`
`\clist_gpush:cx`

```

5340 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
5341 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
5342 \cs_new_eq:NN \clist_push:No \clist_put_left:No
5343 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
5344 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
5345 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
5346 \cs_new_eq:NN \clist_push:co \clist_put_left:co
5347 \cs_new_eq:NN \clist_push:cx \clist_put_left:cx
5348 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
5349 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
5350 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
5351 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
5352 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
5353 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
5354 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
5355 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

```

(End definition for `\clist_push:Nn` and others. These functions are documented on page 133.)

173.4 Using comma lists

`\clist_use:N` The approach is the same as for `\tl_use:N`.
`\clist_use:c`

```

5356 \cs_new_eq:NN \clist_use:N \tl_use:N
5357 \cs_new_eq:NN \clist_use:c \tl_use:c

```

(End definition for `\clist_use:N` and `\clist_use:c`. These functions are documented on page 128.)

173.5 Modifying comma lists

`\l_clist_remove_clist` An internal comma list for the removal routines.

```
5358 \clist_new:N \l_clist_remove_clist
```

Removing duplicates means making a new list then copying it.

```
\clist_remove_duplicates:N
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
\clist_remove_duplicates_aux:NN
5359 \cs_new_protected:Npn \clist_remove_duplicates:N
5360 { \clist_remove_duplicates_aux:NN \clist_set_eq:NN }
5361 \cs_new_protected:Npn \clist_gremove_duplicates:N
5362 { \clist_remove_duplicates_aux:NN \clist_gset_eq:NN }
5363 \cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2
5364 {
5365   \clist_clear:N \l_clist_remove_clist
5366   \clist_map_inline:Nn #2
5367   {
5368     \clist_if_in:NnF \l_clist_remove_clist {##1}
5369     { \clist_put_right:Nn \l_clist_remove_clist {##1} }
5370   }
5371   #1 #2 \l_clist_remove_clist
5372 }
5373 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
5374 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(End definition for `\clist_remove_duplicates:N` and `\clist_remove_duplicates:c`. These functions are documented on page 128.)

`\clist_remove_all:Nn` Removing an item from a comma list is done without looping over the entire list, as the performance of that approach is very bad for long lists. Instead, a delimited function is needed. For this to work correctly, there is a need to add an additional comma at the start of the list, and to remove it again once the removal is complete. Of course, the list can end up empty, which is the reason for the test before copying back to the parent.

`\clist_remove_all:cn`

`\clist_gremove_all:Nn`

`\clist_gremove_all:cn`

`\clist_remove_all_aux:NNn`

`\clist_remove_all_aux:w`

```
5375 \cs_new_protected:Npn \clist_remove_all:Nn
5376 { \clist_remove_all_aux:NNn \clist_set_eq:NN }
5377 \cs_new_protected:Npn \clist_gremove_all:Nn
5378 { \clist_remove_all_aux:NNn \clist_gset_eq:NN }
5379 \cs_new_protected:Npn \clist_remove_all_aux:NNn #1#2#3
5380 {
5381   \clist_if_empty:NF #2
5382   {
5383     \clist_clear:N \l_clist_remove_clist
5384     \cs_set_protected:Npn \clist_remove_all_aux:w
5385     ##1 , #3 , ##2 \q_stop
5386     {
5387       \tl_put_right:Nn \l_clist_remove_clist {##1}
```

```

5388     \quark_if_no_value:nF {##2}
5389     { \clist_remove_all_aux:w , ##2 \q_stop }
5390   }
5391   \exp_after:wN \clist_remove_all_aux:w
5392   \exp_after:wN , #2 , #3 , \q_no_value \q_stop
5393   \tl_if_empty:NF \l_clist_remove_clist
5394   {
5395     \exp_after:wN \tl_set:No \exp_after:wN
5396     \l_clist_remove_clist \exp_after:wN
5397     { \exp_after:wN \use_none:n \l_clist_remove_clist }
5398   }
5399   #1 #2 \l_clist_remove_clist
5400 }
5401 }
5402 \cs_new_protected:Npn \clist_remove_all_aux:w { }
5403 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
5404 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and `\clist_remove_all:cn`. These functions are documented on page 129.)

```

\clist_trim_spaces:n
\clist_trim_spaces:N
\clist_trim_spaces:c
\clist_gtrim_spaces:N
\clist_gtrim_spaces:c
\clist_trim_spaces_aux_i:n
\clist_trim_spaces_aux_ii:n

```

Here, the basic plan is to use `\tl_trim_spaces:n` to do the work: the only issue is to make sure that the number of commas at the end of the process is correct.

```

5405 \cs_new:Npn \clist_trim_spaces:n #1
5406 {
5407   \exp_args:Nf \clist_trim_spaces_aux_i:n
5408   { \clist_map_function:nN {#1} \clist_trim_spaces_aux_ii:n }
5409 }
5410 \cs_new:Npn \clist_trim_spaces_aux_i:n #1 { \use_ii:nn #1 }
5411 \cs_new:Npn \clist_trim_spaces_aux_ii:n #1
5412 { , \tl_trim_spaces:n {#1} }
5413 \cs_new_protected:Npn \clist_trim_spaces:N #1
5414 { \tl_set:Nf #1 { \exp_args:No \clist_trim_spaces:n #1 } }
5415 \cs_new_protected:Npn \clist_gtrim_spaces:N #1
5416 { \tl_gset:Nf #1 { \exp_args:No \clist_trim_spaces:n #1 } }
5417 \cs_generate_variant:Nn \clist_trim_spaces:N { c }
5418 \cs_generate_variant:Nn \clist_gtrim_spaces:N { c }

```

(End definition for `\clist_trim_spaces:n`. This function is documented on page ??.)

173.6 Comma list conditionals

`\clist_tmp:w` A temporary function for comparison.

```

5419 \cs_new_protected:Npn \clist_tmp:w { }

```

(End definition for `\clist_tmp:w`.)

`\clist_if_empty_p:N` Simple copies from the token list variable material.

```

\clist_if_empty_p:c 5420 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }
\clist_if_empty:NTF 5421 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c { p , T , F , TF }
\clist_if_empty:cTF

```

(End definition for `\clist_if_empty:N` and `\clist_if_empty:c`. These functions are documented on page 129.)

`\clist_if_eq_p:NN` Simple copies from the token list variable material.

```

\clist_if_eq_p:Nc 5422 \prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
\clist_if_eq_p:cN 5423 \prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
\clist_if_eq_p:cc 5424 \prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
\clist_if_eq:NTF 5425 \prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc { p , T , F , TF }
\clist_if_eq:NcTF
\clist_if_eq:cNTF
\clist_if_eq:ccTF
\clist_if_in:NnTF
\clist_if_in:NVTF
\clist_if_in:NoTF
\clist_if_in:cnTF
\clist_if_in:cVTF
\clist_if_in:coTF

```

(End definition for `\clist_if_eq:NN` and others. These functions are documented on page 130.)

```

5426 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2
5427 { T , F , TF }
5428 {
5429   \cs_set_protected:Npn \clist_tmp:w ##1 , #2 , ##2##3 \q_stop
5430   {
5431     \if_meaning:w \q_no_value ##2
5432     \prg_return_false:
5433     \else:
5434       \prg_return_true:
5435     \fi:
5436   }
5437   \exp_last_unbraced:NNo \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
5438 }
5439 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2
5440 { T , F , TF }
5441 {
5442   \cs_set_protected:Npn \clist_tmp:w ##1 , #2 , ##2##3 \q_stop
5443   {
5444     \if_meaning:w \q_no_value ##2
5445     \prg_return_false:
5446     \else:
5447       \prg_return_true:
5448     \fi:
5449   }
5450   \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
5451 }
5452 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }
5453 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
5454 \cs_generate_variant:Nn \clist_if_in:NnF { NV , No }
5455 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
5456 \cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }
5457 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }

```

```

5458 \cs_generate_variant:Nn \clist_if_in:nnT {      nV , no }
5459 \cs_generate_variant:Nn \clist_if_in:nnF {      nV , no }
5460 \cs_generate_variant:Nn \clist_if_in:nnTF {      nV , no }

```

(End definition for `\clist_if_in:Nn` and others. These functions are documented on page 130.)

173.7 Mapping to comma lists

```

\clist_map_function:NN
\clist_map_function:cN
\clist_map_function:nN
\clist_map_function_aux:Nw

```

Mapping to comma lists is pretty simple, if not massively efficient.

```

5461 \cs_new_nopar:Npn \clist_map_function:NN #1#2
5462 {
5463   \clist_if_empty:NF #1
5464   {
5465     \exp_last_unbraced:NNo \clist_map_function_aux:Nw #2 #1
5466     , \q_recursion_tail , \q_recursion_stop
5467   }
5468 }
5469 \cs_new:Npn \clist_map_function:nN #1#2
5470 {
5471   \tl_if_empty:nF {#1}
5472   {
5473     \clist_map_function_aux:Nw #2 #1
5474     , \q_recursion_tail , \q_recursion_stop
5475   }
5476 }
5477 \cs_new:Npn \clist_map_function_aux:Nw #1#2 ,
5478 {
5479   \quark_if_recursion_tail_stop:n {#2}
5480   #1 {#2}
5481   \clist_map_function_aux:Nw #1
5482 }
5483 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN` and `\clist_map_function:cN`. These functions are documented on page 130.)

`\g_clist_map_inline_int` For the nesting of mappings.

```

5484 \int_new:N \g_clist_map_inline_int

```

```

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

```

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with \TeX ’s groups.

```

5485 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
5486 {
5487   \int_gincr:N \g_clist_map_inline_int
5488   \cs_gset:cpn { clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5489   ##1

```

```

5490     {#2}
5491     \exp_args:Nnc \clist_map_function:NN #1
5492     { \clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5493     \int_gdecr:N \g_clist_map_inline_int
5494   }
5495   \cs_new_protected:Npn \clist_map_inline:nn #1#2
5496   {
5497     \int_gincr:N \g_clist_map_inline_int
5498     \cs_gset:cpn { \clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5499     ##1
5500     {#2}
5501     \exp_args:Nnc \clist_map_function:nn {#1}
5502     { \clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5503     \int_gdecr:N \g_clist_map_inline_int
5504   }
5505   \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End definition for `\clist_map_inline:nn` and `\clist_map_inline:cn`. These functions are documented on page 131.)

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

This is just a dedicated version of the inline mapping.

```

5506   \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
5507   {
5508     \clist_map_inline:Nn #1
5509     {
5510       \tl_set:Nn #2 {##1}
5511       #3
5512     }
5513   }
5514   \cs_new_protected:Npn \clist_map_variable:nNn #1#2#3
5515   {
5516     \clist_map_inline:nn {#1}
5517     {
5518       \tl_set:Nn #2 {##1}
5519       #3
5520     }
5521   }
5522   \cs_generate_variant:Nn \clist_map_variable:NNn { c }

```

(End definition for `\clist_map_variable:NNn` and `\clist_map_variable:cNn`. These functions are documented on page 131.)

`\clist_map_break:`
`\clist_map_break:n`

Both are simple renaming.

```

5523   \cs_new_eq:NN \clist_map_break: \use_none_delimit_by_q_recursion_stop:w
5524   \cs_new_eq:NN \clist_map_break:n \use_i_delimit_by_q_recursion_stop:nw

```

(End definition for `\clist_map_break:`. This function is documented on page 131.)

174 Viewing comma lists

`\clist_show:N` The aim of the mapping here is to create a token list containing the formatted comma list. The very first item needs the new line and `>\` removing, which is achieved using a `w`-type auxiliary. To avoid a low-level TeX error if there is an empty comma list, a simple test is used to keep the output “clean”.

`\clist_show:c`

`\clist_show_aux:n`

`\clist_show_aux:w`

```

5525 \cs_new_protected_nopar:Npn \clist_show:N #1
5526 {
5527   \clist_if_empty:NTF #1
5528   {
5529     \iow_term:x { Comma-list~\token_to_str:N #1 \c_space_tl is~empty }
5530     \tl_show:n { }
5531   }
5532   {
5533     \iow_term:x
5534     {
5535       Comma-list~\token_to_str:N #1 \c_space_tl
5536       contains~the~items~(without~outer~braces):
5537     }
5538     \tl_set:Nx \l_clist_show_tl
5539     { \clist_map_function:NN #1 \clist_show_aux:n }
5540     \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
5541     { \exp_after:wN \clist_show_aux:w \l_clist_show_tl }
5542   }
5543 }
5544 \cs_new:Npn \clist_show_aux:n #1
5545 {
5546   \iow_newline: > \c_space_tl \c_space_tl
5547   \iow_char:N \{ \exp_not:n {#1} \iow_char:N \}
5548 }
5549 \cs_new:Npn \clist_show_aux:w #1 > ~ { }
5550 \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for `\clist_show:N` and `\clist_show:c`. These functions are documented on page 133.)

174.1 Experimental functions

`\clist_length:N` Counting the items in a comma list is done using the same approach as for other length functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

`\clist_length:c`

`\clist_length:n`

`\clist_length_aux:n`

```

5551 \cs_new:Npn \clist_length:N #1
5552 {
5553   \int_eval:n
5554   {
5555     0
5556     \clist_map_function:NN #1 \clist_length_aux:n

```

```

5557     }
5558   }
5559   \cs_new:Npn \clist_length:n #1
5560   {
5561     \int_eval:n
5562     {
5563       0
5564       \clist_map_function:nN {#1} \clist_length_aux:n
5565     }
5566   }
5567   \cs_new:Npn \clist_length_aux:n #1 { +1 }
5568   \cs_generate_variant:Nn \clist_length:N { c }

```

(End definition for `\clist_length:N` and `\clist_length:c`. These functions are documented on page 133.)

`\clist_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the
`\clist_item:cn` correct item. If the resulting offset is too large, then `\quark_if_recursion_stop:n` will
`\clist_item:nn` be true, terminating the loop and returning nothing at all.

`\clist_item_aux:nnn`

```

5569   \cs_set_nopar:Npn \clist_item:Nn #1#2
5570   { \exp_args:No \clist_item:nn #1 {#2} }
5571   \cs_set:Npn \clist_item:nn #1#2
5572   {
5573     \int_compare:nNnTF {#2} < \c_zero
5574     {
5575       \exp_args:Nf \clist_item_aux:nw
5576       { \int_eval:n { \clist_length:n {#1} + #2 } }
5577       #1 , \q_recursion_tail \q_recursion_stop
5578     }
5579     { \clist_item_aux:nw {#2} #1 , \q_recursion_tail \q_recursion_stop }
5580   }
5581   \cs_set:Npn \clist_item_aux:nw #1#2 , #3
5582   {
5583     \int_compare:nNnTF {#1} = \c_zero
5584     { \use_i_delimit_by_q_recursion_stop:nw {#2} }
5585     {
5586       \quark_if_recursion_tail_stop:n {#3}
5587       \exp_args:Nf \clist_item_aux:nw
5588       { \int_eval:n { #1 - 1 } }
5589       #3
5590     }
5591   }
5592   \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn` and `\clist_item:cn`. These functions are documented on page 134.)

`\clist_set_from_seq:NN` Setting a comma list from a comma-separated list is done using a simple mapping. We
`\clist_set_from_seq:cN` wrap each item with braces, `\exp_not:n`, and a comma. The first comma must be
`\clist_set_from_seq:Nc`
`\clist_set_from_seq:cc`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:cN`
`\clist_gset_from_seq:Nc`
`\clist_gset_from_seq:cc`

removed, except in the case of an empty comma-list.

```

5593 \cs_new_protected:Npn \clist_set_from_seq:NN #1#2
5594 {
5595   \seq_if_empty:NTF #2
5596   { \clist_clear:N #1 }
5597   {
5598     \seq_push_item_def:n { , \exp_not:n {{##1}} }
5599     \tl_set:Nx #1
5600     { \exp_after:wN \use_none:n \tex_romannumeral:D -'\0 #2 }
5601     \seq_pop_item_def:
5602   }
5603 }
5604 \cs_new_protected:Npn \clist_gset_from_seq:NN #1#2
5605 {
5606   \seq_if_empty:NTF #2
5607   { \clist_gclear:N #1 }
5608   {
5609     \seq_push_item_def:n { , \exp_not:n {{##1}} }
5610     \tl_gset:Nx #1
5611     { \exp_after:wN \use_none:n \tex_romannumeral:D -'\0 #2 }
5612     \seq_pop_item_def:
5613   }
5614 }
5615 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
5616 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
5617 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
5618 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }

```

(End definition for `\clist_set_from_seq:NN` and others. These functions are documented on page [134](#).)

174.2 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\clist_top:NN` These are old stack functions.

`\clist_top:cN`

```

5619 \cs_new_eq:NN \clist_top:NN \clist_get:NN
5620 \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 ??.)

`\clist_remove_element:Nn` An older name for `\clist_remove_all:Nn`.

`\clist_gremove_element:Nn`

```

5621 \cs_new_eq:NN \clist_remove_element:Nn \clist_remove_all:Nn
5622 \cs_new_eq:NN \clist_gremove_element:Nn \clist_gremove_all:Nn

```

(End definition for `\clist_remove_element:Nn`. This function is documented on page ??.)

`\clist_display:N` An older name for `\clist_show:N`.
`\clist_display:c`

```

5623 \cs_new_eq:NN \clist_display:N \clist_show:N
5624 \cs_new_eq:NN \clist_display:c \clist_show:c

(End definition for \clist_display:N and \clist_display:c. These functions are documented on page
??.)

5625 </initex | package>

```

175 l3prop implementation

The following test files are used for this code: *m3prop001*.

```

5626 <*initex | package>

5627 <*package>
5628 \ProvidesExplPackage
5629   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5630 \package_check_loaded_expl:
5631 </package>

```

A property list is a macro whose top-level expansion is for the form “`\q_prop <key0> \q_prop {<value0>} \q_prop ... \q_prop <keyn-1> \q_prop {<valuen-1>} \q_prop`”. The trailing `\q_prop` is always present for performance reasons: this means that empty property lists are not actually empty.

`\q_prop` A private quark is used as a marker between entries.

```

5632 \quark_new:N \q_prop

(End definition for \q_prop. This function is documented on page 142.)

```

`\c_empty_prop` An empty prop contains exactly one `\q_prop`.

```

5633 \tl_const:Nn \c_empty_prop { \q_prop }

```

175.1 Allocation and initialisation

`\prop_new:N` Internally, property lists are token lists, but an empty prop is not an empty tl, so we
`\prop_new:c` need to do things by hand.

```

5634 \cs_new_protected:Npn \prop_new:N #1 { \cs_new_eq:NN #1 \c_empty_prop }
5635 \cs_new_protected:Npn \prop_new:c #1 { \cs_new_eq:cN {#1} \c_empty_prop }

(End definition for \prop_new:N and \prop_new:c. These functions are documented on page 135.)

```

`\prop_clear:N` The same idea for clearing

`\prop_clear:c`

`\prop_gclear:N`

`\prop_gclear:c`

```
5636 \cs_new_protected:Npn \prop_clear:N #1 { \cs_set_eq:NN #1 \c_empty_prop }
5637 \cs_new_protected:Npn \prop_clear:c #1 { \cs_set_eq:cN {#1} \c_empty_prop }
5638 \cs_new_protected:Npn \prop_gclear:N #1 { \cs_gset_eq:NN #1 \c_empty_prop }
5639 \cs_new_protected:Npn \prop_gclear:c #1 { \cs_gset_eq:cN {#1} \c_empty_prop }
```

(End definition for `\prop_clear:N` and `\prop_clear:c`. These functions are documented on page 135.)

`\prop_clear_new:N`

`\prop_clear_new:c`

`\prop_gclear_new:N`

`\prop_gclear_new:c`

Once again a simple copy from the token list functions.

```
5640 \cs_new_protected:Npn \prop_clear_new:N #1
5641 { \cs_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
5642 \cs_generate_variant:Nn \prop_clear_new:N {c}
5643 \cs_new_protected:Npn \prop_gclear_new:N #1
5644 { \cs_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
5645 \cs_new_eq:NN \prop_gclear_new:c \prop_gclear:c
```

(End definition for `\prop_clear_new:N` and `\prop_clear_new:c`. These functions are documented on page 135.)

`\prop_set_eq:NN`

`\prop_set_eq:cN`

`\prop_set_eq:Nc`

`\prop_set_eq:cc`

`\prop_gset_eq:NN`

`\prop_gset_eq:cN`

`\prop_gset_eq:Nc`

`\prop_gset_eq:cc`

Once again, these are simply copies from the token list functions.

```
5646 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
5647 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
5648 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
5649 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
5650 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
5651 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
5652 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
5653 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\prop_set_eq:NN` and others. These functions are documented on page 135.)

175.2 Accessing data in property lists

`\prop_split:NnTF`

`\prop_split_aux:NnTF`

`\prop_split_aux:nnnn`

`\prop_split_aux:w`

This function is used by most of the module, and hence must be fast. The aim here is to split a property list at a given key into the part before the key–value pair, the value associated with the key and the part after the key–value pair. To do this, the key is first detokenized (to avoid repeatedly doing this), then a delimited function is constructed to match the key. It will match `\q_prop <detokenized key> \q_prop {<value>} <extra argument>`, effectively separating an *<extract1>* before the key in the property list and an *<extract2>* after the key.

If the key is present in the property list, then *<extra argument>* is simply `\q_prop`, and `\prop_split_aux:nnnn` will gobble this and the false branch (#4), leaving the correct code on the input stream. More precisely, it leaves the user code (true branch), followed by three groups, $\{\langle extract_1 \rangle\} \{\langle value \rangle\} \{\langle extract_2 \rangle\}$. In order for *<extract1>* *<extract2>* to

be a well-formed property list, $\langle extract1 \rangle$ has a leading and trailing $\backslash q_prop$, retaining exactly the structure of a property list, while $\langle extract2 \rangle$ omits the leading $\backslash q_prop$.

If the key is not there, then $\langle extra\ argument \rangle$ is $? \backslash use_ii:nn \{ \}$, and $\backslash prop_split_aux:nnnn ? \backslash use_i$ removes the three brace groups that just follow. Then $\backslash use_ii:nn$ removes the true branch, leaving the false branch, with no trailing material.

```

5654 \cs_set_protected:Npn \prop_split:NnTF #1#2
5655 { \exp_args:NNo \prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
5656 \cs_new_protected:Npn \prop_split_aux:NnTF #1#2
5657 {
5658   \cs_set_protected:Npn \prop_split_aux:w
5659     ##1 \q_prop #2 \q_prop ##2 ##3 ##4 \q_mark ##5 \q_stop
5660     { \prop_split_aux:nnnn ##3 { {##1 \q_prop } {##2} {##4} } }
5661   \exp_after:wN \prop_split_aux:w #1 \q_mark
5662     \q_prop #2 \q_prop { } { ? \use_ii:nn { } } \q_mark \q_stop
5663 }
5664 \cs_new:Npn \prop_split_aux:nnnn #1#2#3#4 { #3 #2 }
5665 \cs_new_protected:Npn \prop_split_aux:w { }

```

(End definition for $\backslash prop_split:NnTF$. This function is documented on page 142.)

$\backslash prop_split:Nnn$ The goal here is to provide a common interface for both true and false branches of $\backslash prop_split:NnTF$. In both cases, the code given by the user will be placed in front of three brace groups, $\{\langle extract1 \rangle\} \{\langle value \rangle\} \{\langle extract2 \rangle\}$. If the key was missing from the property list, then $\langle extract1 \rangle$ is the full property list, $\langle value \rangle$ is $\backslash q_no_value$, and $\langle extract2 \rangle$ is empty. Otherwise, $\langle extract1 \rangle$ is the part of the property list before the $\langle key \rangle$, and has the structure of a property list, $\langle value \rangle$ is the value corresponding to the $\langle key \rangle$, and $\langle extract2 \rangle$ (the part after the $\langle key \rangle$) is missing the leading $\backslash q_prop$.

```

5666 \cs_set_protected:Npn \prop_split:Nnn #1#2#3
5667 {
5668   \prop_split:NnTF #1 {#2}
5669   {#3}
5670   { \exp_args:Nno \use:n {#3} {#1} { \q_no_value } { } }
5671 }

```

(End definition for $\backslash prop_split:Nnn$. This function is documented on page 142.)

$\backslash prop_del:Nn$ Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

$\backslash prop_del:NV$
 $\backslash prop_del:cn$
 $\backslash prop_del:cV$
 $\backslash prop_gdel:Nn$
 $\backslash prop_gdel:NV$
 $\backslash prop_gdel:cn$
 $\backslash prop_gdel:cV$

```

5672 \cs_new_protected:Npn \prop_del:Nn #1#2
5673 { \prop_split:NnTF #1 {#2} { \prop_del_aux:NNnnn \tl_set:Nn #1 } { } }
5674 \cs_new_protected:Npn \prop_gdel:Nn #1#2
5675 { \prop_split:NnTF #1 {#2} { \prop_del_aux:NNnnn \tl_gset:Nn #1 } { } }
5676 \cs_new_protected:Npn \prop_del_aux:NNnnn #1#2#3#4#5
5677 { #1 #2 { #3 #5 } }
5678 \cs_generate_variant:Nn \prop_del:Nn { NV }
5679 \cs_generate_variant:Nn \prop_del:Nn { c , cV }

```

$\backslash prop_del_aux:NNnnn$

```

5680 \cs_generate_variant:Nn \prop_gdel:Nn { NV }
5681 \cs_generate_variant:Nn \prop_gdel:Nn { c , cV }

```

(End definition for `\prop_del:Nn` and others. These functions are documented on page 138.)

`\prop_get:NnN` Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

`\prop_get:NVN`

`\prop_get:NoN`

`\prop_get:cnN`

`\prop_get:cVN`

`\prop_get:NoN`

`\prop_get_aux:Nnnn`

```

5682 \cs_new_protected:Npn \prop_get:NnN #1#2#3
5683 {
5684   \prop_split:NnTF #1 {#2}
5685   { \prop_get_aux:Nnnn #3 }
5686   { \tl_set:Nn #3 { \q_no_value } }
5687 }
5688 \cs_new_protected:Npn \prop_get_aux:Nnnn #1#2#3#4
5689 { \tl_set:Nn #1 {#3} }
5690 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
5691 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

```

(End definition for `\prop_get:NnN` and others. These functions are documented on page 137.)

`\prop_pop:NnN` Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

`\prop_pop:NoN`

`\prop_pop:cnN`

`\prop_pop:coN`

`\prop_gpop:NnN`

`\prop_gpop:NoN`

`\prop_gpop:cnN`

`\prop_gpop:coN`

`\prop_pop_aux:NNNnnn`

```

5692 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
5693 {
5694   \prop_split:NnTF #1 {#2}
5695   { \prop_pop_aux:NNNnnn \tl_set:Nn #1 #3 }
5696   { \tl_set:Nn #3 { \q_no_value } }
5697 }
5698 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
5699 {
5700   \prop_split:NnTF #1 {#2}
5701   { \prop_pop_aux:NNNnnn \tl_gset:Nn #1 #3 }
5702   { \tl_set:Nn #3 { \q_no_value } }
5703 }
5704 \cs_new_protected:Npn \prop_pop_aux:NNNnnn #1#2#3#4#5#6
5705 {
5706   \tl_set:Nn #3 {#5}
5707   #1 #2 { #4 #6 }
5708 }
5709 \cs_generate_variant:Nn \prop_pop:NnN { No }
5710 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
5711 \cs_generate_variant:Nn \prop_gpop:NnN { No }
5712 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and others. These functions are documented on page 138.)

`\prop_put:Nnn` Putting a key–value pair in a property list starts by splitting to remove any existing value. The property list is then reconstructed with the two remaining parts #5 and #7 first, followed by the new or updated entry.

```

5713 \cs_new_protected:Npn \prop_put:Nnn { \prop_put_aux:NNnn \tl_set:Nx }
5714 \cs_new_protected:Npn \prop_gput:Nnn { \prop_put_aux:NNnn \tl_gset:Nx }
5715 \cs_new_protected:Npn \prop_put_aux:NNnn #1#2#3#4
5716 {
5717   \prop_split:Nnn #2 {#3} { \prop_put_aux:NNnnnn #1 #2 {#3} {#4} }
5718 }
5719 \cs_new_protected:Npn \prop_put_aux:NNnnnn #1#2#3#4#5#6#7
5720 {
5721   #1 #2
5722   {
5723     \exp_not:n { #5 #7 }
5724     \tl_to_str:n {#3} \exp_not:n { \q_prop {#4} \q_prop }
5725   }
5726 }
5727 \cs_generate_variant:Nn \prop_put:Nnn
5728 { NnV , Nno , Nnx , NV , NVV , No , Noo }
5729 \cs_generate_variant:Nn \prop_put:Nnn
5730 { c , cnV , cno , cnx , cV , cVV , co , coo }
5731 \cs_generate_variant:Nn \prop_gput:Nnn
5732 { NnV , Nno , Nnx , NV , NVV , No , Noo }
5733 \cs_generate_variant:Nn \prop_gput:Nnn
5734 { c , cnV , cno , cnx , cV , cVV , co , coo }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 136.)

`\prop_gput:Nnn` Adding conditionally also splits. If the key is already present, the three brace groups given by `\prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```

5735 \cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
5736 { \prop_put_if_new_aux:NNnn \tl_put_right:Nx }
5737 \cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
5738 { \prop_put_if_new_aux:NNnn \tl_gput_right:Nx }
5739 \cs_new_protected:Npn \prop_put_if_new_aux:NNnn #1#2#3#4
5740 {
5741   \prop_split:NnTF #2 {#3}
5742   { \use_none:nnn }
5743   {
5744     #1 #2
5745     { \tl_to_str:n {#3} \exp_not:n { \q_prop {#4} \q_prop } }
5746   }
5747 }
5748 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
5749 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn` and `\prop_gput_if_new:cnn`. These functions are documented on page 137.)

175.3 Property list conditionals

`\prop_if_empty_p:N`
`\prop_if_empty_p:c`
`\prop_if_empty:NTF`
`\prop_if_empty:cTF`

The test here uses `\c_empty_prop` as it is not really empty!

```
5750 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
5751 {
5752   \if_meaning:w #1 \c_empty_prop
5753   \prg_return_true:
5754   \else:
5755     \prg_return_false:
5756   \fi:
5757 }
5758 \cs_generate_variant:Nn \prop_if_empty_p:N {c}
5759 \cs_generate_variant:Nn \prop_if_empty:NTF {c}
5760 \cs_generate_variant:Nn \prop_if_empty:NT {c}
5761 \cs_generate_variant:Nn \prop_if_empty:NF {c}
```

(End definition for `\prop_if_empty:N` and `\prop_if_empty:c`. These functions are documented on page 138.)

`\prop_if_in_p:Nn`
`\prop_if_in_p:Nv`
`\prop_if_in_p:No`
`\prop_if_in_p:cn`
`\prop_if_in_p:cV`
`\prop_if_in_p:co`
`\prop_if_in:NnTF`
`\prop_if_in:NvTF`
`\prop_if_in:NoTF`
`\prop_if_in:cnTF`
`\prop_if_in:cVTF`
`\prop_if_in:coTF`
`\prop_if_in_aux:w`

Testing expandably if a key is in a property list requires to go through the key-value pairs one by one. This is rather slow, and a faster test would be

```
\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
  \prop_split:NnTF #1 {#2}
  {
    \prg_return_true:
    \use_none:nnn
  }
  { \prg_return_false: }
}
```

but `\prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq:nn`, which is expandable. The mapping is stopped using A, which cannot appear within a key of the property list, since keys are strings. Here, `\prop_map_function:NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```
5762 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
5763 {
5764   \exp_last_unbraced:Noo \prop_if_in_aux:nwn
5765   { \tl_to_str:n {#2} } #1
5766   A \q_prop { } \q_stop
5767 }
5768 \cs_new:Npn \prop_if_in_aux:nwn #1 \q_prop #2 \q_prop #3
5769 {
```

```

5770 \if_catcode:w A #2
5771 \prg_return_false:
5772 \exp_after:wN \use_none_delimit_by_q_stop:w
5773 \fi:
5774 \str_if_eq:nnT {#1} {#2}
5775 {
5776 \prg_return_true:
5777 \use_none_delimit_by_q_stop:w
5778 }
5779 \prop_if_in_aux:nwn {#1}
5780 }
5781 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
5782 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
5783 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
5784 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
5785 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
5786 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
5787 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
5788 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }

```

(End definition for `\prop_if_in:Nn` and others. These functions are documented on page 139.)

175.4 Mapping to property lists

`\prop_map_function:NN` The fastest way to do a recursion here is to use an `\if_catcode:w` test: the keys are strings, and thus cannot match the marker A (which has catcode “letter”).

`\prop_map_function:Nc`

`\prop_map_function:cN`

`\prop_map_function:cc`

`\prop_map_function_aux:Nwn`

```

5789 \cs_new_nopar:Npn \prop_map_function:NN #1#2
5790 {
5791 \exp_last_unbraced:NNo \prop_map_function_aux:Nwn #2
5792 #1 A \q_prop { } \q_recursion_stop
5793 }
5794 \cs_new:Npn \prop_map_function_aux:Nwn #1 \q_prop #2 \q_prop #3
5795 {
5796 \if_catcode:w A #2
5797 \exp_after:wN \prop_map_break:
5798 \fi:
5799 #1 {#2} {#3}
5800 \prop_map_function_aux:Nwn #1
5801 }
5802 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
5803 \cs_generate_variant:Nn \prop_map_function:NN { c , cc }

```

(End definition for `\prop_map_function:NN` and others. These functions are documented on page ??.)

`\g_prop_map_inline_int` A nesting counter for mapping.

```

5804 \int_new:N \g_prop_map_inline_int

```


`\prop_map_inline:Nn` Mapping in line requires a nesting level counter.

`\prop_map_inline:cn`

```

5805 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
5806 {
5807   \int_gincr:N \g_prop_map_inline_int
5808   \cs_gset:cpn { prop_map_inline_ \int_use:N \g_prop_map_inline_int :nn }
5809   ##1##2 {#2}
5810   \prop_map_function:Nc #1
5811   { prop_map_inline_ \int_use:N \g_prop_map_inline_int :nn }
5812   \int_gdecr:N \g_prop_map_inline_int
5813 }
5814 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn` and `\prop_map_inline:cn`. These functions are documented on page 139.)

`\prop_map_break:` Breaking the map function simply means removing everything up to the `\q_stop` marker.

```

5815 \cs_new_eq:NN \prop_map_break: \use_none_delimit_by_q_recursion_stop:w

```

(End definition for `\prop_map_break:.` This function is documented on page 139.)

`\prop_map_break:n` The same idea for using one set of tokens.

```

5816 \cs_new_eq:NN \prop_map_break:n \use_i_delimit_by_q_recursion_stop:nw

```

(End definition for `\prop_map_break:n`. This function is documented on page 140.)

175.5 Viewing property lists

`\l_prop_show_tl` Used to store the material for display.

```

5817 \tl_new:N \l_prop_show_tl

```

`\prop_show:N` The aim of the mapping here is to create a token list containing the formatted property list. The very first item needs the new line and `>_` removing, which is achieved using

`\prop_show:c`

`\prop_show_aux:n` a `w`-type auxiliary. To avoid a low-level T_EX error if there is an empty property list, a simple test is used to keep the output “clean”.

`\prop_show_aux:w`

```

5818 \cs_new_protected_nopar:Npn \prop_show:N #1
5819 {
5820   \prop_if_empty:NTF #1
5821   {
5822     \iow_term:x { Property-list~\token_to_str:N #1 \c_space_tl is~empty }
5823     \tl_show:n { }
5824   }
5825   {
5826     \iow_term:x
5827     {

```

```

5828         Property~list~\token_to_str:N #1 \c_space_tl
5829         contains~the~pairs~(without~outer~braces):
5830     }
5831     \tl_set:Nx \l_prop_show_tl
5832     { \prop_map_function:NN #1 \prop_show_aux:nn }
5833     \tl_show:n \exp_after:wN \exp_after:wN \exp_after:wN
5834     { \exp_after:wN \prop_show_aux:w \l_prop_show_tl }
5835 }
5836 }
5837 \cs_new:Npn \prop_show_aux:nn #1#2
5838 {
5839     \iow_newline: > \c_space_tl \c_space_tl
5840     \iow_char:N \{ #1 \iow_char:N \}
5841     \c_space_tl \c_space_tl => \c_space_tl \c_space_tl
5842     \iow_char:N \{ \exp_not:n {#2} \iow_char:N \}
5843 }
5844 \cs_new:Npn \prop_show_aux:w #1 > ~ { }
5845 \cs_generate_variant:Nn \prop_show:N { c }

```

(End definition for `\prop_show:N` and `\prop_show:c`. These functions are documented on page 140.)

175.6 Experimental functions

`\prop_get:NnTF`
`\prop_get_aux_true:Nnnn`

Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

5846 \prg_new_protected_conditional:Npnn \prop_get:Nn #1#2#3 { T , F , TF }
5847 {
5848     \prop_split:NnTF #1 {#2}
5849     { \prop_get_aux_true:Nnnn #3 }
5850     { \prg_return_false: }
5851 }
5852 \cs_new_protected:Npn \prop_get_aux_true:Nnnn #1#2#3#4
5853 {
5854     \tl_set:Nn #1 {#3}
5855     \prg_return_true:
5856 }

```

(End definition for `\prop_get:Nn`. This function is documented on page 141.)

`\prop_pop:NnTF`
`\prop_gpop:NnTF`
`\prop_pop_aux_true:NNNnnn`

Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

```

5857 \prg_new_protected_conditional:Npnn \prop_pop:Nn #1#2#3 { T,F,TF }
5858 {
5859     \prop_split:NnTF #1 {#2}
5860     { \prop_pop_aux_true:NNNnnn \tl_set:Nn #1 #3 }

```

```

5861     { \prg_return_false: }
5862   }
5863   \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 {T,F,TF}
5864   {
5865     \prop_split:NnTF #1 {#2}
5866     { \prop_pop_aux_true:NNNnnn \tl_gset:Nn #1 #3 }
5867     { \prg_return_false: }
5868   }
5869   \cs_new_protected:Npn \prop_pop_aux_true:NNNnnn #1#2#3#4#5#6
5870   {
5871     \tl_set:Nn #3 {#5}
5872     #1 #2 { #4 #6 }
5873     \prg_return_true:
5874   }

```

(End definition for `\prop_pop:NnN` and `\prop_gpop:NnN`. These functions are documented on page 141.)

`\prop_map_tokens:Nn`
`\prop_map_tokens_aux:nwn`

The mapping grabs one key–value pair at a time, and stops when reaching the marker key A, with catcode “letter”, which cannot appear in normal keys since those are strings. The odd construction `\use:n {#1}` allows #1 to contain any token.

```

5875   \cs_new:Npn \prop_map_tokens:Nn #1#2
5876   {
5877     \exp_last_unbraced:Nno \prop_map_tokens_aux:nwn {#2} #1
5878     A \q_prop { } \q_recursion_stop
5879   }
5880   \cs_new:Npn \prop_map_tokens_aux:nwn #1 \q_prop #2 \q_prop #3
5881   {
5882     \if_catcode:w A #2
5883     \exp_after:wN \prop_map_break:
5884     \fi:
5885     \use:n {#1} {#2} {#3}
5886     \prop_map_tokens_aux:nwn {#1}
5887   }

```

(End definition for `\prop_map_tokens:Nn`. This function is documented on page 141.)

`\prop_get:Nn`
`\prop_get_aux:nnn`

Getting expandably the value corresponding to a key in a property list is a simple instance of mapping some tokens. Map the function `\prop_get_aux:nnn` which takes as its three arguments the $\langle key \rangle$ that we are looking for, the current $\langle key \rangle$ and the current $\langle value \rangle$. If the $\langle keys \rangle$ match, the $\langle value \rangle$ is returned. If none of the keys match, this expands to nothing.

```

5888   \cs_new:Npn \prop_get:Nn #1 #2
5889   { \prop_map_tokens:Nn #1 { \prop_get_aux:nnn {#2} } }
5890   \cs_new:Npn \prop_get_aux:nnn #1 #2 #3
5891   { \str_if_eq:nnT {#1} {#2} { \prop_map_break:n {#3} } }

```

(End definition for `\prop_get:Nn`. This function is documented on page 141.)

175.7 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\prop_display:N` An older name for `\prop_show:N`.

`\prop_display:c`

5892 `\cs_new_eq:NN \prop_display:N \prop_show:N`

5893 `\cs_new_eq:NN \prop_display:c \prop_show:c`

(End definition for `\prop_display:N` and `\prop_display:c`. These functions are documented on page ??.)

`\prop_gget:NnN`

`\prop_gget:NVN`

`\prop_gget:cnN`

`\prop_gget:cVN`

`\prop_gget_aux:Nnnn`

Getting globally is no longer supported: this is a conceptual change, so the necessary code for the transition is provided directly.

5894 `\cs_new_protected:Npn \prop_gget:NnN #1#2#3`

5895 `{ \prop_split:Nnn #1 {#2} { \prop_gget_aux:Nnnn #3 } }`

5896 `\cs_new_protected:Npn \prop_gget_aux:Nnnn #1#2#3#4`

5897 `{ \tl_gset:Nn #1 {#3} }`

5898 `\cs_generate_variant:Nn \prop_gget:NnN { NV }`

5899 `\cs_generate_variant:Nn \prop_gget:NnN { c , cV }`

(End definition for `\prop_gget:NnN` and others. These functions are documented on page ??.)

`\prop_get_gdel:NnN`

This name seems very odd.

5900 `\cs_new_eq:NN \prop_get_gdel:NnN \prop_gpop:NnN`

(End definition for `\prop_get_gdel:NnN`. This function is documented on page ??.)

`\prop_if_in:ccTF`

A hang-over from an ancient implementation

5901 `\cs_generate_variant:Nn \prop_if_in:NnT { cc }`

5902 `\cs_generate_variant:Nn \prop_if_in:NnF { cc }`

5903 `\cs_generate_variant:Nn \prop_if_in:NnTF { cc }`

(End definition for `\prop_if_in:cc`. This function is documented on page ??.)

`\prop_gput:ccx`

Another one.

5904 `\cs_generate_variant:Nn \prop_gput:Nnn { ccx }`

(End definition for `\prop_gput:ccx`. This function is documented on page ??.)

`\prop_if_eq_p:NN`

`\prop_if_eq_p:Nc`

`\prop_if_eq_p:cN`

`\prop_if_eq_p:cc`

`\prop_if_eq:NNTF`

`\prop_if_eq:NcTF`

`\prop_if_eq:cNTF`

`\prop_if_eq:ccTF`

These ones do no even make sense!

5905 `\prg_new_eq_conditional:NNn \prop_if_eq:NN \tl_if_eq:NN { p , T , F , TF }`

5906 `\prg_new_eq_conditional:NNn \prop_if_eq:cN \tl_if_eq:cN { p , T , F , TF }`

5907 `\prg_new_eq_conditional:NNn \prop_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }`

5908 `\prg_new_eq_conditional:NNn \prop_if_eq:cc \tl_if_eq:cc { p , T , F , TF }`

(End definition for `\prop_if_eq:NN` and others. These functions are documented on page ??.)

5909 `\</initex | package>`

176 l3box implementation

```
5910 <*initex | package>
5911 <*package>
5912 \ProvidesExplPackage
5913   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5914 \package_check_loaded_expl:
5915 <*initex | package>
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

176.1 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

\box_new:N Defining a new $\langle box \rangle$ register: remember that box 255 is not generally available.

\box_new:c

```
5916 <*initex>
5917 \alloc_new:nnnN { box } \c_zero \c_max_register_int \tex_mathchardef:D
5918 \seq_put_right:Nn \g_box_allocation_seq {255}
5919 </initex>
5920 <*package>
5921 \cs_new_protected:Npn \box_new:N #1
5922 {
5923   \chk_if_free_cs:N #1
5924   \newbox #1
5925 }
5926 </package>
5927 \cs_generate_variant:Nn \box_new:N { c }
```

\box_clear:N Clear a $\langle box \rangle$ register.

\box_clear:c

\box_gclear:N

\box_gclear:c

```
5928 \cs_new_protected_nopar:Npn \box_clear:N #1
5929 { \box_set_eq:NN #1 \c_empty_box }
5930 \cs_new_protected_nopar:Npn \box_gclear:N #1
5931 { \box_gset_eq:NN #1 \c_empty_box }
5932 \cs_generate_variant:Nn \box_clear:N { c }
5933 \cs_generate_variant:Nn \box_gclear:N { c }
```

\box_clear_new:N Clear or new.

\box_clear_new:c

\box_gclear_new:N

\box_gclear_new:c

```
5934 \cs_new_protected_nopar:Npn \box_clear_new:N #1
5935 {
5936   \cs_if_exist:NTF #1
5937   { \box_set_eq:NN #1 \c_empty_box }
5938   { \box_new:N #1 }
5939 }
5940 \cs_new_protected_nopar:Npn \box_gclear_new:N #1
```

```

5941 {
5942   \cs_if_exist:NTF #1
5943   { \box_gset_eq:NN #1 \c_empty_box }
5944   { \box_new:N #1 }
5945 }
5946 \cs_generate_variant:Nn \box_clear_new:N { c }
5947 \cs_generate_variant:Nn \box_gclear_new:N { c }

```

`\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
\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc
5948 \cs_new_protected_nopar:Npn \box_set_eq:NN #1#2
5949 { \tex_setbox:D #1 \tex_copy:D #2 }
5950 \cs_new_protected_nopar:Npn \box_gset_eq:NN
5951 { \pref_global:D \box_set_eq:NN }
5952 \cs_generate_variant:Nn \box_set_eq:NN { cN , Nc , cc }
5953 \cs_generate_variant:Nn \box_gset_eq:NN { cN , Nc , cc }

```

`\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
\box_gset_eq_clear:NN
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc
5954 \cs_new_protected_nopar:Npn \box_set_eq_clear:NN #1#2
5955 { \tex_setbox:D #1 \tex_box:D #2 }
5956 \cs_new_protected_nopar:Npn \box_gset_eq_clear:NN
5957 { \pref_global:D \box_set_eq_clear:NN }
5958 \cs_generate_variant:Nn \box_set_eq_clear:NN { cN , Nc , cc }
5959 \cs_generate_variant:Nn \box_gset_eq_clear:NN { cN , Nc , cc }

```

176.2 Measuring and setting box dimensions

`\box_ht:N` Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

\box_ht:c
\box_dp:N
\box_dp:c
\box_wd:N
\box_wd:c
5960 \cs_new_eq:NN \box_ht:N \tex_ht:D
5961 \cs_new_eq:NN \box_dp:N \tex_dp:D
5962 \cs_new_eq:NN \box_wd:N \tex_wd:D
5963 \cs_generate_variant:Nn \box_ht:N { c }
5964 \cs_generate_variant:Nn \box_dp:N { c }
5965 \cs_generate_variant:Nn \box_wd:N { c }

```

`\box_set_ht:Nn` Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```

\box_set_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_set_wd:Nn
\box_set_wd:cn
5966 \cs_new_protected_nopar:Npn \box_set_dp:Nn #1#2
5967 { \box_dp:N #1 \dim_eval:w #2 \dim_eval_end: }
5968 \cs_new_protected_nopar:Npn \box_set_ht:Nn #1#2
5969 { \box_ht:N #1 \dim_eval:w #2 \dim_eval_end: }
5970 \cs_new_protected_nopar:Npn \box_set_wd:Nn #1#2
5971 { \box_wd:N #1 \dim_eval:w #2 \dim_eval_end: }
5972 \cs_generate_variant:Nn \box_set_ht:Nn { c }
5973 \cs_generate_variant:Nn \box_set_dp:Nn { c }
5974 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

176.3 Using boxes

`\box_use_clear:N` Using a $\langle box \rangle$. These are just TeX primitives with meaningful names.

```
\box_use_clear:c
\box_use:N
\box_use:c

5975 \cs_new_eq:NN \box_use_clear:N \tex_box:D
5976 \cs_new_eq:NN \box_use:N \tex_copy:D
5977 \cs_generate_variant:Nn \box_use_clear:N { c }
5978 \cs_generate_variant:Nn \box_use:N { c }
```

`\box_move_left:nn` Move box material in different directions.

```
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

5979 \cs_new_protected:Npn \box_move_left:nn #1#2
5980 { \tex_moveleft:D \dim_eval:w #1 \dim_eval_end: #2 }
5981 \cs_new_protected:Npn \box_move_right:nn #1#2
5982 { \tex_moveright:D \dim_eval:w #1 \dim_eval_end: #2 }
5983 \cs_new_protected:Npn \box_move_up:nn #1#2
5984 { \tex_raise:D \dim_eval:w #1 \dim_eval_end: #2 }
5985 \cs_new_protected:Npn \box_move_down:nn #1#2
5986 { \tex_lower:D \dim_eval:w #1 \dim_eval_end: #2 }
```

176.4 Box conditionals

`\if_hbox:N` The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```
\if_vbox:N
\if_box_empty:N

5987 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
5988 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
5989 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

```
\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:NTF
\box_if_horizontal:cTF
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:NTF
\box_if_vertical:cTF

5990 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
5991 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
5992 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
5993 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
5994 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
5995 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
5996 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
5997 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
5998 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
5999 \cs_generate_variant:Nn \box_if_vertical:NT { c }
6000 \cs_generate_variant:Nn \box_if_vertical:NF { c }
6001 \cs_generate_variant:Nn \box_if_vertical:NTF { c }
```

`\box_if_empty_p:N` Testing if a $\langle box \rangle$ is empty/void.

```
\box_if_empty_p:c
\box_if_empty:NTF
\box_if_empty:cTF

6002 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
6003 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6004 \cs_generate_variant:Nn \box_if_empty_p:N { c }
```

```

6005 \cs_generate_variant:Nn \box_if_empty:NT { c }
6006 \cs_generate_variant:Nn \box_if_empty:NF { c }
6007 \cs_generate_variant:Nn \box_if_empty:NTF { c }

```

(End definition for `\box_new:N` and `\box_new:c`. These functions are documented on page 146.)

176.5 The last box inserted

`\l_last_box` A different name for this read-only primitive.

```

6008 \cs_new_eq:NN \l_last_box \tex_lastbox:D

```

`\box_set_to_last:N` Set a box to the previous box.

`\box_set_to_last:c`

`\box_gset_to_last:N`

`\box_gset_to_last:c`

```

6009 \cs_new_protected_nopar:Npn \box_set_to_last:N #1
6010 { \tex_setbox:D #1 \l_last_box }
6011 \cs_new_protected_nopar:Npn \box_gset_to_last:N
6012 { \pref_global:D \box_set_to_last:N }
6013 \cs_generate_variant:Nn \box_set_to_last:N { c }
6014 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for `\box_set_to_last:N` and `\box_set_to_last:c`. These functions are documented on page ??.)

176.6 Constant boxes

`\c_empty_box`

```

6015 <*package>
6016 \cs_new_eq:NN \c_empty_box \voidb@x
6017 </package>
6018 <*initex>
6019 \box_new:N \c_empty_box
6020 </initex>

```

176.7 Scratch boxes

`\l_tmpa_box`

`\l_tmpb_box`

```

6021 <*package>
6022 \cs_new_eq:NN \l_tmpa_box \@tempboxa
6023 </package>
6024 <*initex>
6025 \box_new:N \l_tmpa_box
6026 </initex>
6027 \box_new:N \l_tmpb_box

```


176.8 Viewing box contents

`\box_show:N` Show the contents of a box and write it into the log file.

`\box_show:c`

```
6028 \cs_new_eq:NN \box_show:N \tex_showbox:D
6029 \cs_generate_variant:Nn \box_show:N { c }
```

(End definition for `\box_show:N` and `\box_show:c`. These functions are documented on page 147.)

176.9 Horizontal mode boxes

`\hbox:n` (The test suite for this command, and others in this file, is `m3box002.lvt`.)

Put a horizontal box directly into the input stream.

```
6030 \cs_new_protected_nopar:Npn \hbox:n { \tex_hbox:D \scan_stop: }
```

(End definition for `\hbox:n`. This function is documented on page 148.)

`\hbox_set:Nn`

`\hbox_set:cn`

`\hbox_gset:Nn`

`\hbox_gset:cn`

```
6031 \cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
6032 \cs_new_protected_nopar:Npn \hbox_gset:Nn { \pref_global:D \hbox_set:Nn }
6033 \cs_generate_variant:Nn \hbox_set:Nn { c }
6034 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(End definition for `\hbox_set:Nn` and `\hbox_set:cn`. These functions are documented on page 148.)

`\hbox_set_to_wd:Nnn`

`\hbox_set_to_wd:cnn`

`\hbox_gset_to_wd:Nnn`

`\hbox_gset_to_wd:cnn`

Storing material in a horizontal box with a specified width.

```
6035 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
6036 { \tex_setbox:D #1 \tex_hbox:D to \dim_eval:w #2 \dim_eval_end: {#3} }
6037 \cs_new_protected_nopar:Npn \hbox_gset_to_wd:Nnn
6038 { \pref_global:D \hbox_set_to_wd:Nnn }
6039 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
6040 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn {cnn}
```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_set_to_wd:cnn`. These functions are documented on page 148.)

`\hbox_set_inline_begin:N`

`\hbox_set_inline_begin:c`

`\hbox_gset_inline_begin:N`

`\hbox_gset_inline_begin:c`

`\hbox_set_inline_end:`

`\hbox_gset_inline_end:`

Storing material in a horizontal box. This type is useful in environment definitions.

```
6041 \cs_new_protected_nopar:Npn \hbox_set_inline_begin:N #1
6042 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
6043 \cs_new_protected_nopar:Npn \hbox_gset_inline_begin:N
6044 { \pref_global:D \hbox_set_inline_begin:N }
6045 \cs_generate_variant:Nn \hbox_set_inline_begin:N { c }
6046 \cs_generate_variant:Nn \hbox_gset_inline_begin:N { c }
6047 \cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token
6048 \cs_new_eq:NN \hbox_gset_inline_end: \c_group_end_token
```

(End definition for `\hbox_set_inline_begin:N` and `\hbox_set_inline_begin:c`. These functions are documented on page 149.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.
`\hbox_to_zero:n`

```
6049 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
6050 { \tex_hbox:D to \dim_eval:w #1 \dim_eval_end: {#2} }
6051 \cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_skip {#1} }
```

(End definition for `\hbox_to_wd:nn`. This function is documented on page 148.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out
`\hbox_overlap_right:n` on the other) directly into the input stream.

```
6052 \cs_new_protected:Npn \hbox_overlap_left:n #1
6053 { \hbox_to_zero:n { \tex_hss:D #1 } }
6054 \cs_new_protected:Npn \hbox_overlap_right:n #1
6055 { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. This function is documented on page 148.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.
`\hbox_unpack:c`
`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

```
6056 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
6057 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
6058 \cs_generate_variant:Nn \hbox_unpack:N { c }
6059 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_clear:N`. These functions are documented on page 149.)

176.10 Vertical mode boxes

`\vbox:n` The following test files are used for this code: `m3box003.lvt`.

`\vbox:n\vbox_top:n` The following test files are used for this code: `m3box003.lvt`.

Put a vertical box directly into the input stream.

```
6060 \cs_new_protected_nopar:Npn \vbox:n { \tex_vbox:D \scan_stop: }
6061 \cs_new_protected_nopar:Npn \vbox_top:n { \tex_vtop:D \scan_stop: }
```

(End definition for `\vbox:n`. This function is documented on page ??.)

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.
`\vbox_to_zero:n`
`\vbox_to_ht:nn`
`\vbox_to_zero:n`

```
6062 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
6063 { \tex_vbox:D to \dim_eval:w #1 \dim_eval_end: {#2} }
6064 \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 150.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.
`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

```

6065 \cs_new_protected:Npn \vbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_vbox:D {#2} }
6066 \cs_new_protected_nopar:Npn \vbox_gset:Nn { \pref_global:D \vbox_set:Nn }
6067 \cs_generate_variant:Nn \vbox_set:Nn { c }
6068 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End definition for `\vbox_set:Nn` and `\vbox_set:cn`. These functions are documented on page 150.)

`\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`

```

6069 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
6070 { \tex_setbox:D #1 \tex_vtop:D {#2} }
6071 \cs_new_protected_nopar:Npn \vbox_gset_top:Nn
6072 { \pref_global:D \vbox_set_top:Nn }
6073 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
6074 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }

```

(End definition for `\vbox_set_top:Nn` and `\vbox_set_top:cn`. These functions are documented on page 151.)

`\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`

```

6075 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
6076 { \tex_setbox:D #1 \tex_vbox:D \dim_eval:w #2 \dim_eval_end: {#3} }
6077 \cs_new_protected_nopar:Npn \vbox_gset_to_ht:Nnn
6078 { \pref_global:D \vbox_set_to_ht:Nnn }
6079 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
6080 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }

```

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_set_to_ht:cnn`. These functions are documented on page 151.)

`\vbox_set_inline_begin:N` Storing material in a vertical box. This type is useful in environment definitions.
`\vbox_set_inline_begin:c`
`\vbox_gset_inline_begin:N`
`\vbox_gset_inline_begin:c`
`\vbox_set_inline_end:`
`\vbox_gset_inline_end:`

```

6081 \cs_new_nopar:Npn \vbox_set_inline_begin:N #1
6082 { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
6083 \cs_new_protected_nopar:Npn \vbox_gset_inline_begin:N
6084 { \pref_global:D \vbox_set_inline_begin:N }
6085 \cs_generate_variant:Nn \vbox_set_inline_begin:N { c }
6086 \cs_generate_variant:Nn \vbox_gset_inline_begin:N { c }
6087 \cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token
6088 \cs_new_eq:NN \vbox_gset_inline_end: \c_group_end_token

```

(End definition for `\vbox_set_inline_begin:N` and `\vbox_set_inline_begin:c`. These functions are documented on page 151.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.
`\vbox_unpack:c`
`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

```

6089 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
6090 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
6091 \cs_generate_variant:Nn \vbox_unpack:N { c }
6092 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

```

(End definition for `\vbox_unpack:N` and `\vbox_unpack_clear:c`. These functions are documented on page 152.)

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

```

6093 \cs_new_protected_nopar:Npn \vbox_set_split_to_ht:NNn #1#2#3
6094 { \tex_setbox:D #1 \tex_vsplit:D #2 to \dim_eval:w #3 \dim_eval_end: }

```

(End definition for `\vbox_set_split_to_ht:NNn`. This function is documented on page 152.)

```

6095 </initex | package>

```

177 l3io implementation

```

6096 <*initex | package>
6097 <*package>
6098 \ProvidesExplPackage
6099 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
6100 \package_check_loaded_expl:
6101 </package>

```

177.1 Primitives

`\if_eof:w` The primitive conditional

```

6102 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End definition for `\if_eof:w`. This function is documented on page 158.)

177.2 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.

```

6103 \cs_new_eq:NN \c_iow_term_stream \c_sixteen
6104 \cs_new_eq:NN \c_ior_term_stream \c_sixteen
6105 \cs_new_eq:NN \c_iow_log_stream \c_minus_one
6106 \cs_new_eq:NN \c_ior_log_stream \c_minus_one

```

`\c_iow_streams_tl` The list of streams available, by number.
`\c_ior_streams_tl`

```

6107 \tl_const:Nn \c_iow_streams_tl
6108 {
6109   \c_zero
6110   \c_one
6111   \c_two
6112   \c_three
6113   \c_four
6114   \c_five
6115   \c_six
6116   \c_seven
6117   \c_eight
6118   \c_nine
6119   \c_ten
6120   \c_eleven
6121   \c_twelve
6122   \c_thirteen
6123   \c_fourteen
6124   \c_fifteen
6125 }
6126 \cs_new_eq:NN \c_ior_streams_tl \c_iow_streams_tl

```

`\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.

```

6127 \prop_new:N \g_iow_streams_prop
6128 \prop_new:N \g_ior_streams_prop
6129 <*package>
6130 \prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e-reserved }
6131 \prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e-reserved }
6132 \prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e-reserved }
6133 \prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e-reserved }
6134 </package>

```

`\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.

```

6135 \int_new:N \l_iow_stream_int
6136 \cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int

```

177.3 Stream management

`\ior_raw_new:N` The lowest level for stream management is actually creating raw T_EX streams. As these
`\ior_raw_new:c` are very limited (even with ε -T_EX), this should not be addressed directly.

`\iow_raw_new:N`
`\iow_raw_new:c` 6137 <*initex>

```

6138 \alloc_setup_type:nnn { ior } \c_zero \c_sixteen
6139 \cs_new_protected_nopar:Npn \ior_raw_new:N #1
6140 { \alloc_reg:NnNN g { ior } \tex_chardef:D #1 }
6141 \alloc_setup_type:nnn { iow } \c_zero \c_sixteen
6142 \cs_new_protected_nopar:Npn \iow_raw_new:N #1
6143 { \alloc_reg:NnNN g { iow } \tex_chardef:D #1 }
6144 \</initex>
6145 \*package>
6146 \cs_set_eq:NN \iow_raw_new:N \newwrite
6147 \cs_set_eq:NN \ior_raw_new:N \newread
6148 \</package>
6149 \cs_generate_variant:Nn \ior_raw_new:N { c }
6150 \cs_generate_variant:Nn \iow_raw_new:N { c }

```

(End definition for `\ior_raw_new:N` and `\iow_raw_new:c`. These functions are documented on page 158.)

`\ior_open:Nn` In both cases, opening a stream starts with a call to the closing function: this is safest.
`\ior_open:cn` There is then a loop through the allocation number list to find the first free stream
`\iow_open:Nn` number. When one is found the allocation can take place, the information can be stored
`\iow_open:cn` and finally the file can actually be opened.

```

6151 \cs_new_protected_nopar:Npn \ior_open:Nn #1#2
6152 {
6153   \ior_close:N #1
6154   \int_set:Nn \l_ior_stream_int \c_sixteen
6155   \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
6156   \int_compare:nNnTF \l_ior_stream_int = \c_sixteen
6157   { \msg_kernel_error:nn { ior } { streams-exhausted } }
6158   {
6159     \ior_stream_alloc:N #1
6160     \prop_gput:NVn \g_ior_streams_prop \l_ior_stream_int {#2}
6161     \tex_openin:D #1#2 \scan_stop:
6162   }
6163 }
6164 \cs_new_protected_nopar:Npn \iow_open:Nn #1#2
6165 {
6166   \iow_close:N #1
6167   \int_set:Nn \l_iow_stream_int \c_sixteen
6168   \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n
6169   \int_compare:nNnTF \l_iow_stream_int = \c_sixteen
6170   { \msg_kernel_error:nn { iow } { streams-exhausted } }
6171   {
6172     \iow_stream_alloc:N #1
6173     \prop_gput:NVn \g_iow_streams_prop \l_iow_stream_int {#2}
6174     \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
6175   }
6176 }
6177 \cs_generate_variant:Nn \ior_open:Nn { c }
6178 \cs_generate_variant:Nn \iow_open:Nn { c }

```

(End definition for `\ior_open:Nn` and `\ior_open:cn`. These functions are documented on page 154.)

`\ior_alloc_read:n` These functions are used to see if a particular stream is available. The property list
`\iow_alloc_write:n` contains file names for streams in use, so any unused ones are for the taking.

```

6179 \cs_new_protected_nopar:Npn \iow_alloc_write:n #1
6180 {
6181   \prop_if_in:NnF \g_iow_streams_prop {#1}
6182   {
6183     \int_set:Nn \l_iow_stream_int {#1}
6184     \tl_map_break:
6185   }
6186 }
6187 \cs_new_protected_nopar:Npn \ior_alloc_read:n #1
6188 {
6189   \prop_if_in:NnF \g_iow_streams_prop {#1}
6190   {
6191     \int_set:Nn \l_ior_stream_int {#1}
6192     \tl_map_break:
6193   }
6194 }

```

(End definition for `\ior_alloc_read: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 `LATEX 2ε` to get “back on track” with allocation.

```

6195 \cs_new_protected_nopar:Npn \iow_stream_alloc:N #1
6196 {
6197   \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
6198   { \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream } }
6199   {
6200     <*package>
6201     \iow_stream_alloc_aux:
6202     \int_compare:nNnT \l_iow_stream_int = \c_sixteen
6203     {
6204       \iow_raw_new:N \g_iow_tmp_stream
6205       \int_set:Nn \l_iow_stream_int { \g_iow_tmp_stream }
6206       \cs_gset_eq:cN
6207       { g_iow_ \int_use:N \l_iow_stream_int _stream }
6208       \g_iow_tmp_stream
6209     }
6210     </package>
6211     <*initex>
6212     \iow_raw_new:c { g_iow_ \int_use:N \l_iow_stream_int _stream }

```

```

6213 </initex>
6214 \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream }
6215 }
6216 }
6217 <*package>
6218 \cs_new_protected_nopar:Npn \iow_stream_alloc_aux:
6219 {
6220 \int_incr:N \l_iow_stream_int
6221 \int_compare:nNnT \l_iow_stream_int < \c_sixteen
6222 {
6223 \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
6224 {
6225 \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
6226 { \iow_stream_alloc_aux: }
6227 }
6228 { \iow_stream_alloc_aux: }
6229 }
6230 }
6231 </package>
6232 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1
6233 {
6234 \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
6235 { \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream } }
6236 {
6237 <*package>
6238 \ior_stream_alloc_aux:
6239 \int_compare:nNnT \l_ior_stream_int = \c_sixteen
6240 {
6241 \ior_raw_new:N \g_ior_tmp_stream
6242 \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }
6243 \cs_gset_eq:cN
6244 { g_ior_ \int_use:N \l_iow_stream_int _stream }
6245 \g_ior_tmp_stream
6246 }
6247 </package>
6248 <*initex>
6249 \ior_raw_new:c { g_ior_ \int_use:N \l_ior_stream_int _stream }
6250 </initex>
6251 \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream }
6252 }
6253 }
6254 <*package>
6255 \cs_new_protected_nopar:Npn \ior_stream_alloc_aux:
6256 {
6257 \int_incr:N \l_ior_stream_int
6258 \int_compare:nNnT \l_ior_stream_int < \c_sixteen
6259 {
6260 \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
6261 {
6262 \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int

```



```

6263         { \ior_stream_alloc_aux: }
6264     }
6265     { \ior_stream_alloc_aux: }
6266 }
6267 }
6268 \</package>

```

(End definition for \iow_stream_alloc:N.)

\iow_close:N Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

\iow_close:c

\iow_close:N

\iow_close:c

```

6269 \cs_new_protected_nopar:Npn \ior_close:N #1
6270 {
6271     \cs_if_exist:NT #1
6272     {
6273         \int_compare:nNnF #1 = \c_minus_one
6274         {
6275             \tex_closein:D #1
6276             \prop_gdel:NV \g_ior_streams_prop #1
6277             \cs_undefine:N #1
6278         }
6279     }
6280 }
6281 \cs_new_protected_nopar:Npn \iow_close:N #1
6282 {
6283     \cs_if_exist:NT #1
6284     {
6285         \int_compare:nNnF #1 = \c_minus_one
6286         {
6287             \tex_immediate:D \tex_closeout:D #1
6288             \prop_gdel:NV \g_iow_streams_prop #1
6289             \cs_undefine:N #1
6290         }
6291     }
6292 }
6293 \cs_generate_variant:Nn \ior_close:N { c }
6294 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for \iow_close:N and \iow_close:c. These functions are documented on page 154.)

\ior_list_streams: Show the property lists, but with some “pretty printing”.

\iow_list_streams:

```

\iow_show_aux:nn 6295 \cs_new_protected_nopar:Npn \ior_list_streams:
\ior_show_aux:nn 6296 {
6297     \prop_if_empty:NTF \g_ior_streams_prop
6298     {
6299         \iow_term:x { No~input~streams~are~open }
6300         \tl_show:n { }

```

```

6301     }
6302     {
6303         \iow_term:x { The~following~input~streams~are~in~use: }
6304         \tl_set:Nx \l_prop_show_tl
6305         { \prop_map_function:NN \g_iow_streams_prop \ior_show_aux:nn }
6306         \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
6307         { \exp_after:wN \prop_show_aux:w \l_prop_show_tl }
6308     }
6309 }
6310 \cs_new:Npn \ior_show_aux:nn #1#2
6311 {
6312     \iow_newline: > \c_space_tl \c_space_tl
6313     #1 \iow_char:N
6314     \c_space_tl \c_space_tl => \c_space_tl \c_space_tl
6315     \exp_not:n {#2}
6316 }
6317 \cs_new_protected_nopar:Npn \iow_list_streams:
6318 {
6319     \prop_if_empty:NTF \g_iow_streams_prop
6320     {
6321         \iow_term:x { No~output~streams~are~open }
6322         \tl_show:n { }
6323     }
6324     {
6325         \iow_term:x { The~following~output~streams~are~in~use: }
6326         \tl_set:Nx \l_prop_show_tl
6327         { \prop_map_function:NN \g_iow_streams_prop \iow_show_aux:nn }
6328         \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
6329         { \exp_after:wN \prop_show_aux:w \l_prop_show_tl }
6330     }
6331 }
6332 \cs_new_eq:NN \iow_show_aux:nn \ior_show_aux:nn

```

(End definition for `\ior_list_streams:`. This function is documented on page 154.)

Text for the error messages.

```

6333 \msg_kernel_new:nnnn { iow } { streams-exhausted }
6334 { Output~streams~exhausted }
6335 {
6336     TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
6337     All~16 are currently~in~use,~and~something~wanted~to~open
6338     another~one.
6339 }
6340 \msg_kernel_new:nnnn { ior } { streams-exhausted }
6341 { Input~streams~exhausted }
6342 {
6343     TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
6344     All~16 are currently~in~use,~and~something~wanted~to~open
6345     another~one.
6346 }

```

177.4 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive.
`\iow_shipout_x:Nx`

```
6347 \cs_new_eq:NN \iow_shipout_x:Nn \tex_write:D
6348 \cs_generate_variant:Nn \iow_shipout_x:Nn { Nx }
```

(End definition for `\iow_shipout_x:Nn` and `\iow_shipout_x:Nx`. These functions are documented on page 155.)

`\iow_shipout:Nn` With ϵ -TeX available deferred writing is easy.
`\iow_shipout:Nx`

```
6349 \cs_new_protected_nopar:Npn \iow_shipout:Nn #1#2
6350 { \iow_shipout_x:Nn #1 { \exp_not:n {#2} } }
6351 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }
```

(End definition for `\iow_shipout:Nn` and `\iow_shipout:Nx`. These functions are documented on page 155.)

177.5 Immediate writing

`\iow_now:Nx` An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```
6352 \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 154.)

`\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.

```
6353 \cs_new_protected_nopar:Npn \iow_now:Nn #1#2
6354 { \iow_now:Nx #1 { \exp_not:n {#2} } }
```

(End definition for `\iow_now:Nn`. This function is documented on page 154.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.

```
\iow_log:x
\iow_term:n
\iow_term:x
6355 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_iow_log_stream }
6356 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_iow_log_stream }
6357 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
6358 \cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_iow_term_stream }
```

(End definition for `\iow_log:n` and `\iow_log:x`. These functions are documented on page 155.)

`\iow_now_when_avail:Nn` For writing only if the stream requested is open at all.
`\iow_now_when_avail:Nx`

```
6359 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nn #1
6360 { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 } }
6361 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nx #1
6362 { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 } }
```

(End definition for `\iow_now_when_avail:Nn` and `\iow_now_when_avail:Nx`. These functions are documented on page 155.)

177.6 Hard-wrapping lines based on length

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_length_int` This is the “raw” length of a line which can be written to file. The standard value is the line length typically used by T_EXLive and MikT_EX.

```
6363 \int_new:N \l_iow_line_length_int
6364 \int_set:Nn \l_iow_line_length_int { 78 }
```

(End definition for \l_iow_line_length_int. This function is documented on page 156.)

`\l_iow_target_length_int` This stores the target line length: the full length minus any part for a leader at the start of each line.

```
6365 \int_new:N \l_iow_target_length_int
```

(End definition for \l_iow_target_length_int.)

`\l_iow_current_line_int` These store the number of characters in the line and word currently being constructed,
`\l_iow_current_word_int` respectively.

```
6366 \int_new:N \l_iow_current_line_int
6367 \int_new:N \l_iow_current_word_int
```

(End definition for \l_iow_current_line_int and \l_iow_current_word_int.)

`\l_iow_current_line_tl` These hold the current line of text and current word, respectively.
`\l_iow_current_word_tl`

```
6368 \tl_new:N \l_iow_current_line_tl
6369 \tl_new:N \l_iow_current_word_tl
```

(End definition for \l_iow_current_line_tl and \l_iow_current_word_tl.)

`\l_iow_wrap_tl` Used for the expansion step before detokenizing.

```
6370 \tl_new:N \l_iow_wrap_tl
```

(End definition for \l_iow_wrap_tl.)

`\l_iow_wrapped_tl` The output from wrapping text: fully expanded and with lines which are not overly long.

```
6371 \tl_new:N \l_iow_wrapped_tl
```

(End definition for \l_iow_wrapped_tl.)

`\q_iow_stop` A quark which will not appear elsewhere.

```
6372 \quark_new:N \q_iow_stop
```

(End definition for `\q_iow_stop`. This function is documented on page ??.)

`\l_iow_line_start_bool` Boolean to avoid adding a space at the beginning of lines.

```
6373 \bool_new:N \l_iow_line_start_bool
```

(End definition for `\l_iow_line_start_bool`. This function is documented on page ??.)

`\iow_wrap:xnnnN` The main wrapping function works as follows. The target number of characters in a line is calculated, before fully-expanding the input such that `\` and `_` are converted into the appropriate values. There is then a loop over each word in the input, which will do the actual wrapping. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The argument `#4` is available for additional set up steps for the output.

```
6374 \cs_new_protected:Npn \iow_wrap:xnnnN #1#2#3#4#5
6375 {
6376   \group_begin:
6377     \int_set:Nn \l_iow_target_length_int { \l_iow_line_length_int - ( #3 ) }
6378     \int_zero:N \l_iow_current_line_int
6379     \tl_clear:N \l_iow_current_line_tl
6380     \tl_clear:N \l_iow_wrap_tl
6381     \bool_set_true:N \l_iow_line_start_bool
6382     \cs_set:Npx \l { \c_space_tl \iow_newline: \c_space_tl }
6383     \cs_set_eq:NN \ \c_space_tl
6384     #4
6385     <*initex>
6386     \tl_set:Nx \l_iow_wrap_tl {#1}
6387     </initex>
6388     <*package>
6389     \protected@edef \l_iow_wrap_tl {#1}
6390     </package>
6391     \cs_set:Npn \l { \iow_newline: #2 }
6392     \use:x
6393     {
6394       \exp_not:N \iow_wrap_loop:w
6395       \tl_to_str:N \l_iow_wrap_tl \c_space_tl
6396       \exp_not:N \q_iow_stop \c_space_tl
6397     }
6398     \exp_args:NNo \group_end:
6399     #5 \l_iow_wrapped_tl
6400 }
```

The loop grabs one word in the input, and checks whether it is the end, or a forced new line, or a normal word.

```
6401 \cs_new_protected:Npn \iow_wrap_loop:w #1 ~ %
6402 {
6403   \tl_set:Nn \l_iow_current_word_tl {#1}
6404   \tl_if_eq:NNTF \l_iow_current_word_tl \iow_newline:
```

```

6405     { \iow_wrap_newline: }
6406     {
6407         \tl_if_eq:NNTF \l_iow_current_word_tl \q_iow_stop
6408         { \iow_wrap_end: }
6409         { \iow_wrap_word: }
6410     }
6411 }

```

For a normal word, update the line length, then test if the current word would fit in the current line, and call the appropriate function.

```

6412 \cs_new_protected_nopar:Npn \iow_wrap_word:
6413 {
6414     \int_set:Nn \l_iow_current_word_int
6415     { \str_length_skip_spaces:N \l_iow_current_word_tl }
6416     \int_add:Nn \l_iow_current_line_int { \l_iow_current_word_int }
6417     \int_compare:nNnTF \l_iow_current_line_int
6418         < \l_iow_target_length_int
6419     { \iow_wrap_word_fits: }
6420     { \iow_wrap_word_newline: }
6421     \iow_wrap_loop:w
6422 }

```

If the word fits in the current line, add it to the line, preceded by a space unless it is the first word of the line.

```

6423 \cs_new_protected_nopar:Npn \iow_wrap_word_fits:
6424 {
6425     \bool_if:NTF \l_iow_line_start_bool
6426     {
6427         \bool_set_false:N \l_iow_line_start_bool
6428         \tl_set_eq:NN \l_iow_current_line_tl \l_iow_current_word_tl
6429     }
6430     {
6431         \tl_put_right:Nx \l_iow_current_line_tl
6432         { ~ \l_iow_current_word_tl }
6433         \int_incr:N \l_iow_current_line_int
6434     }
6435 }

```

Otherwise, the current line is added to the result, with the run-on text. The current word (and its length) are then put in the new line.

```

6436 \cs_new_protected_nopar:Npn \iow_wrap_word_newline:
6437 {
6438     \tl_put_right:Nx \l_iow_wrapped_tl
6439     { \l_iow_current_line_tl \\ }
6440     \int_set_eq:NN \l_iow_current_line_int \l_iow_current_word_int
6441     \tl_set_eq:NN \l_iow_current_line_tl \l_iow_current_word_tl
6442 }

```

Forced newlines are almost identical to those caused by overflow, except that here the word is empty. And remember to continue the loop!

```

6443 \cs_new_protected_nopar:Npn \iow_wrap_newline:
6444 {
6445   \tl_put_right:Nx \l_iow_wrapped_tl
6446   { \l_iow_current_line_tl \ }
6447   \int_zero:N \l_iow_current_line_int
6448   \tl_clear:N \l_iow_current_line_tl
6449   \bool_set_true:N \l_iow_line_start_bool
6450   \iow_wrap_loop:w
6451 }

```

At the end, we simply save the last line (without the run-on text).

```

6452 \cs_new_protected_nopar:Npn \iow_wrap_end:
6453 {
6454   \tl_put_right:Nx \l_iow_wrapped_tl
6455   { \l_iow_current_line_tl }
6456 }

```

(End definition for `\iow_wrap:xxxxN`. This function is documented on page 156.)

```

\str_length_skip_spaces:N
\str_length_skip_spaces:n
\str_length_loop:NNNNNNNNN

```

The wrapping code requires to measure the number of character in each word. This could be done with `\tl_length:n`, but it is ten times faster (literally) to use the code below.

```

6457 \cs_new_nopar:Npn \str_length_skip_spaces:N
6458 { \exp_args:No \str_length_skip_spaces:n }
6459 \cs_new:Npn \str_length_skip_spaces:n #1
6460 {
6461   \int_value:w \int_eval:w
6462   \exp_after:wN \str_length_loop:NNNNNNNNN \tl_to_str:n {#1}
6463   {X8}{X7}{X6}{X5}{X4}{X3}{X2}{X1}{X0} \q_stop
6464   \int_eval_end:
6465 }
6466 \cs_new:Npn \str_length_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
6467 {
6468   \if_catcode:w X #9
6469   \exp_after:wN \use_none_delimit_by_q_stop:w
6470   \else:
6471     9 +
6472     \exp_after:wN \str_length_loop:NNNNNNNNN
6473   \fi:
6474 }

```

(End definition for `\str_length_skip_spaces:N`. This function is documented on page ??.)

177.7 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream

```
6475 \cs_new_nopar:Npn \iow_newline: { ^^J }
```

(End definition for `\iow_newline:`. This function is documented on page 156.)

`\iow_char:N` Function to write any escaped char to an output stream.

```
6476 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for `\iow_char:N`. This function is documented on page 155.)

177.8 Reading input

`\ior_if_eof_p:p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof_p:N TF` As the pool model means that closed streams are undefined control sequences, the test has two parts.

```
6477 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
6478 {
6479   \cs_if_exist:NTF #1
6480   {
6481     \if_eof:w #1
6482     \prg_return_true:
6483   \else:
6484     \prg_return_false:
6485   \fi:
6486   }
6487   { \prg_return_true: }
6488 }
```

(End definition for `\ior_if_eof_p:N`. These functions are documented on page 157.)

`\ior_to:NN` And here we read from files.

`\ior_gto:NN`

```
6489 \cs_new_protected_nopar:Npn \ior_to:NN #1#2
6490 { \tex_read:D #1 to #2 }
6491 \cs_new_protected_nopar:Npn \ior_gto:NN #1#2
6492 { \pref_global:D \tex_read:D #1 to #2 }
```

(End definition for `\ior_to:NN`. This function is documented on page 157.)

`\ior_str_to:NN` Reading as strings is also a primitive wrapper.

`\ior_str_gto:NN`

```
6493 \cs_new_protected_nopar:Npn \ior_str_to:NN #1#2
6494 { \etex_readline:D #1 to #2 }
6495 \cs_new_protected_nopar:Npn \ior_str_gto:NN #1#2
6496 { \pref_global:D \etex_readline:D #1 to #2 }
```

(End definition for `\ior_str_to:NN`. This function is documented on page 157.)

177.9 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\iow_now_buffer_safe:Nn` This is much more easily done using the wrapping system: there is an expansion there,
`\iow_now_buffer_safe:Nx` so a bit of a hack is needed.

```

6497 \cs_new_protected:Npn \iow_now_buffer_safe:Nn #1#2
6498 { \iow_wrap:xnnnN { \exp_not:n {#2} } { } \c_zero { } \iow_now:Nn #1 }
6499 \cs_new_protected:Npn \iow_now_buffer_safe:Nx #1#2
6500 { \iow_wrap:xnnnN {#2} { } \c_zero { } \iow_now:Nn #1 }

```

(End definition for `\iow_now_buffer_safe:Nn` and `\iow_now_buffer_safe:Nx`. These functions are documented on page ??.)

`\ior_new:N` As input–output operations are done using a stack, **new** operations seem out-of-place.
`\ior_new:c` They are therefore set up just to gobble the input.

```

\iow_new:N
\iow_new:c
6501 \cs_new_eq:NN \ior_new:N \use_none:n
6502 \cs_new_eq:NN \ior_new:c \use_none:n
6503 \cs_new_eq:NN \iow_new:N \use_none:n
6504 \cs_new_eq:NN \iow_new:c \use_none:n

```

(End definition for `\ior_new:N` and `\ior_new:c`. These functions are documented on page ??.)

`\ior_open_streams:` Slightly misleading names.

```

\iow_open_streams:
6505 \cs_new_eq:NN \ior_open_streams: \ior_list_streams:
6506 \cs_new_eq:NN \iow_open_streams: \iow_list_streams:

```

(End definition for `\ior_open_streams:`. This function is documented on page ??.)

```

6507 </initex | package>

```

178 l3msg implementation

```

6508 <*initex | package>
6509 <*package>
6510 \ProvidesExplPackage
6511 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
6512 \package_check_loaded_expl:
6513 </package>

```

`\l_msg_tmp_tl` A general scratch for the module.

```

6514 \tl_new:N \l_msg_tmp_tl

```

179 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

```
\c_msg_text_prefix_tl  
\c_msg_more_text_prefix_tl
```

Locations for the text of messages.

```
6515 \tl_const:Nn \c_msg_text_prefix_tl { msg~text~>~ }  
6516 \tl_const:Nn \c_msg_more_text_prefix_tl { msg~extra~text~>~ }
```

```
\msg_new:nnnn  
\msg_new:nnn  
\msg_set:nnnn  
\msg_set:nnn
```

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
6517 \cs_new_protected:Npn \msg_new:nnnn #1#2  
6518 {  
6519   \cs_if_exist:cT { \c_msg_text_prefix_tl #1 / #2 }  
6520   {  
6521     \msg_kernel_error:nn { msg } { message-already-defined }  
6522     {#1} {#2}  
6523   }  
6524   \msg_set:nnnn {#1} {#2}  
6525 }  
6526 \cs_new_protected:Npn \msg_new:nnn #1#2#3  
6527 { \msg_new:nnnn {#1} {#2} {#3} { } }  
6528 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4  
6529 {  
6530   \cs_set:cpn { \c_msg_text_prefix_tl #1 / #2 }  
6531   ##1##2##3##4 {#3}  
6532   \cs_set:cpn { \c_msg_more_text_prefix_tl #1 / #2 }  
6533   ##1##2##3##4 {#4}  
6534 }  
6535 \cs_new_protected:Npn \msg_set:nnn #1#2#3  
6536 { \msg_set:nnnn {#1} {#2} {#3} { } }
```

(End definition for `\msg_new:nnnn` and `\msg_new:nnn`. These functions are documented on page 159.)

179.1 Messages: support functions and text

```
\c_msg_coding_error_text_tl  
\c_msg_continue_text_tl  
\c_msg_critical_text_tl  
  \c_msg_fatal_text_tl  
  \c_msg_help_text_tl  
\c_msg_no_info_text_tl  
  \c_msg_on_line_tl  
  \c_msg_return_text_tl  
  \c_msg_trouble_text_tl
```

Simple pieces of text for messages.

```
6537 \tl_const:Nn \c_msg_coding_error_text_tl  
6538 {  
6539   This-is-a-coding-error.  
6540   \\ \\  
6541 }  
6542 \tl_const:Nn \c_msg_continue_text_tl  
6543 { Type~<return>~to~continue }
```

```

6544 \tl_const:Nn \c_msg_critical_text_tl
6545   { Reading~the~current~file~will~stop }
6546 \tl_const:Nn \c_msg_fatal_text_tl
6547   { This~is~a~fatal~error:~LaTeX~will~abort }
6548 \tl_const:Nn \c_msg_help_text_tl
6549   { For~immediate~help~type~H~<return> }
6550 \tl_const:Nn \c_msg_no_info_text_tl
6551   {
6552     LaTeX~does~not~know~anything~more~about~this~error,~sorry.
6553     \c_msg_return_text_tl
6554   }
6555 \tl_const:Nn \c_msg_on_line_text_tl { on~line }
6556 \tl_const:Nn \c_msg_return_text_tl
6557   {
6558     \\ \\
6559     Try~typing~<return>~to~proceed.
6560     \\
6561     If~that~doesn't~work,~type~X~<return>~to~quit.
6562   }
6563 \tl_const:Nn \c_msg_trouble_text_tl
6564   {
6565     \\ \\
6566     More~errors~will~almost~certainly~follow: \\
6567     the~LaTeX~run~should~be~aborted.
6568   }

```

\msg_newline: New lines are printed in the same way as for low-level file writing.
\msg_two_newlines:

```

6569 \cs_new_nopar:Npn \msg_newline: { ^^J }
6570 \cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }

```

(End definition for \msg_newline: and \msg_two_newlines:. These functions are documented on page 164.)

\msg_line_number: For writing the line number nicely.
\msg_line_context:

```

6571 \cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
6572 \cs_set_nopar:Npn \msg_line_context:
6573   {
6574     \c_msg_on_line_text_tl
6575     \c_space_tl
6576     \msg_line_number:
6577   }

```

(End definition for \msg_line_number:. This function is documented on page 160.)

179.2 Showing messages: low level mechanism

\c_msg_hide_tl aux]_msg_hide_tl<dots> 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”. Using two variables here means that later life is a little easier.

```

6578 \char_set_catcode_letter:N \.
6579 \tl_new:N
6580 \c_msg_hide_tl.....
6581 \tl_const:Nn \c_msg_hide_tl
6582 { \c_msg_hide_tl..... }
6583 \char_set_catcode_other:N \.

```

```

\msg_interrupt:xxx
\msg_interrupt_no_details:xx
\msg_interrupt_details:xxx
\msg_interrupt_text:n
\msg_interrupt_more_text:n
\msg_interrupt_aux:

```

The low-level interruption macro is rather opaque, unfortunately. The idea here is to create a message which hides all of T_EX’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.

```

6584 \cs_new_protected:Npn \msg_interrupt:xxx #1#2#3
6585 {
6586   \group_begin:
6587   \tl_if_empty:nTF {#3}
6588     { \msg_interrupt_no_details:xx {#1} {#2} }
6589     { \msg_interrupt_details:xxx {#1} {#2} {#3} }
6590   \msg_interrupt_aux:
6591   \group_end:
6592 }

6593 % Depending on the availability of more information there is a choice of
6594 % how to set up the further help. The extra help text has to be set
6595 % before the message itself can be issued. Everything is done using
6596 % \texttt{x}-type expansion as the new line markers are different for
6597 % the two type of text and need to be correctly set up.
6598 % \begin{macrocode}
6599 \cs_new_protected:Npn \msg_interrupt_no_details:xx #1#2
6600 {
6601   \iow_wrap:xnnnN
6602   { \\ \c_msg_no_info_text_tl }
6603   { /~ } { 2 } { } \msg_interrupt_more_text:n
6604   \iow_wrap:xnnnN { #1 \\ \\ #2 \\ \\ \c_msg_continue_text_tl }
6605   { ! ~ } { 2 } { } \msg_interrupt_text:n
6606 }
6607 \cs_new_protected:Npn \msg_interrupt_details:xxx #1#2#3
6608 {
6609   \iow_wrap:xnnnN
6610   { \\ #3 }
6611   { /~ } { 2 } { } \msg_interrupt_more_text:n
6612   \iow_wrap:xnnnN { #1 \\ \\ #2 \\ \\ \c_msg_help_text_tl }
6613   { ! ~ } { 2 } { } \msg_interrupt_text:n
6614 }
6615 \cs_new_protected:Npn \msg_interrupt_text:n #1
6616 { \tl_set:Nn \l_msg_text_tl {#1} }

```

```

6617 \cs_new_protected:Npn \msg_interrupt_more_text:n #1
6618 {
6619   \*initex
6620   \tl_set:Nx \l_msg_tmp_tl
6621 \*initex
6622 \*package
6623   \protected@edef \l_msg_tmp_tl
6624 \*package
6625 {
6626   |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
6627   #1
6628   \msg_newline:
6629   |.....
6630 }
6631 \tex_errhelp:D \exp_after:wN { \l_msg_tmp_tl }
6632 }

```

The business end of the process starts by producing some visual separation of the message from the main part of the log. It then adds the hiding text to the message to print. The error message needs to be printed with everything made “invisible”: this is where the strange business with & comes in: this is made into another !. There is also a closing brace that will show up in the output, which is turned into a blank space.

```

6633 \group_begin: % {
6634   \char_set_lccode:w '\} = '\ \scan_stop:
6635   \char_set_lccode:w '\& = '\! \scan_stop:
6636   \char_set_catcode_active:N \&
6637   \tl_to_lowercase:n
6638   {
6639     \group_end:
6640     \cs_new_protected:Npn \msg_interrupt_aux:
6641     {
6642       \iow_term:x
6643       {
6644         \iow_newline:
6645         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
6646         \iow_newline:
6647         !
6648       }
6649       \tl_put_right:No \l_msg_text_tl { \c_msg_hide_tl }
6650       \cs_set_protected_nopar:Npx &
6651       { \tex_errmessage:D { \exp_not:o { \l_msg_text_tl } } }
6652       &
6653     }
6654   }

```

(End definition for `\msg_interrupt:xxx`. This function is documented on page 164.)

`\msg_log:x` Printing to the log or terminal without a stop is rather easier. A bit of simple visual
`\msg_term:x` work sets things off nicely.

```

6655 \cs_new_protected:Npn \msg_log:x #1
6656 {
6657   \iow_log:x { ..... }
6658   \iow_wrap:xnnnN { . ~ #1 } { . ~ } { 2 } { }
6659   \iow_log:x
6660   \iow_log:x { ..... }
6661 }
6662 \cs_new_protected:Npn \msg_term:x #1
6663 {
6664   \iow_term:x { ***** }
6665   \iow_wrap:xnnnN { * ~ #1 } { * ~ } { 2 } { }
6666   \iow_term:x
6667   \iow_term:x { ***** }
6668 }

```

(End definition for `\msg_log:x`. This function is documented on page 165.)

179.3 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled.

```

6669 \int_set:Nn \tex_errorcontextlines:D { -1 }

```

```

\msg_fatal_text:n
\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n

```

A function for issuing messages: both the text and order could in principal vary.

```

6670 \cs_new_nopar:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
6671 \cs_new_nopar:Npn \msg_critical_text:n #1 { Critical~#1~error }
6672 \cs_new_nopar:Npn \msg_error_text:n #1 { #1~error }
6673 \cs_new_nopar:Npn \msg_warning_text:n #1 { #1~warning }
6674 \cs_new_nopar:Npn \msg_info_text:n #1 { #1~info }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 161.)

```

\msg_see_documentation_text:n

```

Contextual footer information.

```

6675 \cs_new_nopar:Npn \msg_see_documentation_text:n #1
6676 { \\ \\ See~the~#1~documentation~for~further~information. }

```

(End definition for `\msg_see_documentation_text:n`. This function is documented on page ??.)

```

\l_msg_redirect_classes_prop
\l_msg_redirect_names_prop

```

For filtering messages, a list of all messages and of those which have to be modified is required.

```

6677 \prop_new:N \l_msg_redirect_classes_prop
6678 \prop_new:N \l_msg_redirect_names_prop

```

```

\msg_class_set:nn

```

Setting up a message class does two tasks. Any existing redirection is cleared, and the various message functions are created to simply use the code stored for the message.

```

6679 \cs_new_protected_nopar:Npn \msg_class_set:nn #1#2
6680 {
6681   \prop_clear_new:c { l_msg_redirect_ #1 _prop }
6682   \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
6683   { \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6} }
6684   \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
6685   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
6686   \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
6687   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
6688   \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3
6689   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
6690   \cs_set_protected:cpx { msg_ #1 :nn } ##1##2
6691   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { } }
6692 }

```

(End definition for `\msg_class_set:nn`. This function is documented on page 161.)

`\msg_if_more_text_p:N` A test to see if any more text is available, using a permanently-empty text function.
`\msg_if_more_text_p:c`
`\msg_if_more_text:NTF`
`\msg_if_more_text:cTF`
`\msg_no_more_text:xxxx`

```

6693 \prg_set_conditional:Npnn \msg_if_more_text:N #1 { p , T , F , TF }
6694 {
6695   \cs_if_eq:NNTF #1 \msg_no_more_text:xxxx
6696   { \prg_return_false: }
6697   { \prg_return_true: }
6698 }
6699 \cs_new:Npn \msg_no_more_text:xxxx #1#2#3#4 { }
6700 \cs_generate_variant:Nn \msg_if_more_text_p:N { c }
6701 \cs_generate_variant:Nn \msg_if_more_text:NT { c }
6702 \cs_generate_variant:Nn \msg_if_more_text:NF { c }
6703 \cs_generate_variant:Nn \msg_if_more_text:NTF { c }

```

(End definition for `\msg_if_more_text:N` and `\msg_if_more_text:c`. These functions are documented on page ??.)

`\msg_fatal:nnxxxx` For fatal errors, after the error message TeX bails out.

```

6704 \msg_class_set:nn { fatal }
6705 {
6706   \msg_interrupt:xxx
6707   { \msg_fatal_text:n {#1} : ~ "#2" }
6708   {
6709     \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6710     \msg_see_documentation_text:n {#1}
6711   }
6712   { \c_msg_fatal_text_tl }
6713   \tex_end:D
6714 }

```

(End definition for `\msg_fatal:nnxxxx` and others. These functions are documented on page 162.)

`\msg_critical:nnxxxx` Not quite so bad: just end the current file.

```
\msg_critical:nnxxx
\msg_critical:nnxx
\msg_critical:nnx
\msg_critical:nn
\msg_critical:nn

6715 \msg_class_set:nn { critical }
6716 {
6717   \msg_interrupt:xxx
6718   { \msg_critical_text:n {#1} : ~ "#2" }
6719   {
6720     \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6721     \msg_see_documentation_text:n {#1}
6722   }
6723   { \c_msg_critical_text_tl }
6724   \tex_endinput:D
6725 }
```

(End definition for `\msg_critical:nnxxxx` and others. These functions are documented on page 162.)

`\msg_error:nnxxxx` For an error, the interrupt routine is called, then any recovery code is tried.

```
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn
\msg_error:nn

6726 \msg_class_set:nn { error }
6727 {
6728   \msg_if_more_text:cTF { \c_msg_more_text_prefix_tl #1 / #2 }
6729   {
6730     \msg_interrupt:xxx
6731     { \msg_error_text:n {#1} : ~ "#2" }
6732     {
6733       \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6734       \msg_see_documentation_text:n {#1}
6735     }
6736     { \use:c { \c_msg_more_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
6737   }
6738   {
6739     \msg_interrupt:xxx
6740     { \msg_error_text:n {#1} : ~ "#2" }
6741     {
6742       \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6743       \msg_see_documentation_text:n {#1}
6744     }
6745     { }
6746   }
6747 }
```

(End definition for `\msg_error:nnxxxx` and others. These functions are documented on page 162.)

`\msg_warning:nnxxxx` Warnings are printed to the terminal.

```
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn
\msg_warning:nn

6748 \msg_class_set:nn { warning }
6749 {
6750   \msg_term:x
6751   {
6752     \msg_warning_text:n {#1} : ~ "#2" \\ \\
```



```

6753         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6754     }
6755 }

```

(End definition for `\msg_warning:nnxxxx` and others. These functions are documented on page 162.)

```

\msg_info:nnxxxx Information only goes into the log.
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn
\msg_info:nn
6756 \msg_class_set:nn { info }
6757 {
6758     \msg_log:x
6759     {
6760         \msg_info_text:n {#1} : ~ "#2" \\ \\
6761         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6762     }
6763 }

```

(End definition for `\msg_info:nnxxxx` and others. These functions are documented on page 163.)

```

\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
\msg_log:nn
6764 \msg_class_set:nn { log }
6765 {
6766     \msg_log:x
6767     { \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
6768 }

```

(End definition for `\msg_log:nnxxxx` and others. These functions are documented on page 163.)

```

\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
\msg_none:nn
6769 \msg_class_set:nn { none } { }

```

(End definition for `\msg_none:nnxxxx` and others. These functions are documented on page 163.)

```

\l_msg_redirect_classes_seq Support variables needed for the redirection system.
\l_msg_class_tl
\l_msg_current_class_tl
\l_msg_current_module_tl
6770 \seq_new:N \l_msg_redirect_classes_seq
6771 \tl_new:N \l_msg_class_tl
6772 \tl_new:N \l_msg_current_class_tl
6773 \tl_new:N \l_msg_current_module_tl

```

```

\msg_use:nnnnxxxx The main message-using macro creates two auxiliary functions: one containing the code
\msg_use_aux:nnn for the message, and the second a loop function. There is then a hand-off to the system
\msg_use_aux:nn for checking if redirection is needed.
\msg_use_loop_check:nn
\msg_use_code:
\msg_use_loop:n
\msg_use_loop:o
6774 \cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8
6775 {
6776     \cs_set_protected_nopar:Npx \msg_use_code:
6777     {

```

```

6778     \seq_clear:N \exp_not:N \l_msg_redirect_classes_seq
6779     \exp_not:n {#2}
6780   }
6781   \cs_set_protected:Npx \msg_use_loop:n ##1
6782   {
6783     \seq_if_in:NnTF \exp_not:n \l_msg_redirect_classes_seq {#1}
6784     { \msg_kernel_error:nn { msg } { message-loop } {#1} }
6785     {
6786       \seq_put_right:Nn \exp_not:N \l_msg_redirect_classes_seq {#1}
6787       \exp_not:N \cs_if_exist:cTF { msg_ ##1 :nnxxxx }
6788       {
6789         \exp_not:N \use:c { msg_ ##1 :nnxxxx }
6790         \exp_not:n { {#3} {#4} {#5} {#6} {#7} {#8} }
6791       }
6792       {
6793         \msg_kernel_error:nnx { msg } { message-class-unknown } {##1}
6794       }
6795     }
6796   }
6797   \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 }
6798   { \msg_use_aux:nnn {#1} {#3} {#4} }
6799   { \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4} }
6800 }

```

The first auxiliary macro looks for a match by name: the most restrictive check.

```

6801 \cs_new_protected_nopar:Npn \msg_use_aux:nnn #1#2#3
6802 {
6803   \tl_set:Nn \l_msg_current_class_tl {#1}
6804   \tl_set:Nn \l_msg_current_module_tl {#2}
6805   \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / }
6806   { \msg_use_loop_check:nn { names } { // #2 / #3 / } }
6807   { \msg_use_aux:nn {#1} {#2} }
6808 }

```

The second function checks for general matches by module or for all modules.

```

6809 \cs_new_protected_nopar:Npn \msg_use_aux:nn #1#2
6810 {
6811   \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2}
6812   { \msg_use_loop_check:nn {#1} {#2} }
6813   {
6814     \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } { * }
6815     { \msg_use_loop_check:nn {#1} { * } }
6816     { \msg_use_code: }
6817   }
6818 }

```

When checking whether to loop, the same code is needed in a few places.

```

6819 \cs_new_protected:Npn \msg_use_loop_check:nn #1#2

```

```

6820 {
6821   \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
6822   \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl
6823   {
6824     { \msg_use_code: }
6825     { \msg_use_loop:o \l_msg_class_tl }
6826   }
6827 }
6828 \cs_new_protected_nopar:Npn \msg_use_code: { }
6829 \cs_new_protected:Npn \msg_use_loop:n #1 { }
6830 \cs_generate_variant:Nn \msg_use_loop:n { o }

```

(End definition for `\msg_use:nnnnxxxx`. This function is documented on page ??.)

`\msg_redirect_class:nn` Converts class one into class two.

```

6831 \cs_new_protected_nopar:Npn \msg_redirect_class:nn #1#2
6832 { \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#2} }

```

(End definition for `\msg_redirect_class:nn`. This function is documented on page 163.)

`\msg_redirect_module:nnn` For when all messages of a class should be altered for a given module.

```

6833 \cs_new_protected_nopar:Npn \msg_redirect_module:nnn #1#2#3
6834 { \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3} }

```

(End definition for `\msg_redirect_module:nnn`. This function is documented on page 163.)

`\msg_redirect_name:nnn` Named message will always use the given class.

```

6835 \cs_new_protected_nopar:Npn \msg_redirect_name:nnn #1#2#3
6836 { \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3} }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 164.)

179.4 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`

```

6837 \cs_new_protected_nopar:Npn \msg_kernel_new:nnnn #1#2
6838 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
6839 \cs_new_protected_nopar:Npn \msg_kernel_new:nnn #1#2
6840 { \msg_new:nnn { LaTeX } { #1 / #2 } }
6841 \cs_new_protected_nopar:Npn \msg_kernel_set:nnnn #1#2
6842 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
6843 \cs_new_protected_nopar:Npn \msg_kernel_set:nnn #1#2
6844 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `\msg_kernel_new:nnnn`. This function is documented on page 166.)

`\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
6845 \cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6
6846 {
6847   \msg_interrupt:xxx
6848   { \msg_fatal_text:n { LaTeX } : ~ "#1 / #2" }
6849   {
6850     \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
6851     {#3} {#4} {#5} {#6}
6852     \msg_see_documentation_text:n { LaTeX3 }
6853   }
6854   { \c_msg_fatal_text_tl }
6855   \tex_end:D
6856 }
6857 \cs_new_protected:Npn \msg_kernel_fatal:nnxxx #1#2#3#4#5
6858 { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
6859 \cs_new_protected:Npn \msg_kernel_fatal:nnxx #1#2#3#4
6860 { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} {#4} { } { } }
6861 \cs_new_protected:Npn \msg_kernel_fatal:nnx #1#2#3
6862 { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} { } { } { } }
6863 \cs_new_protected:Npn \msg_kernel_fatal:nn #1#2
6864 { \msg_kernel_fatal:nnxxxx {#1} {#2} { } { } { } { } }
```

(End definition for `\msg_kernel_fatal:nnxxxx`. This function is documented on page 166.)

`\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
6865 \cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6
6866 {
6867   \msg_if_more_text:cTF { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 }
6868   {
6869     \msg_interrupt:xxx
6870     { \msg_error_text:n { LaTeX } : ~ " #1 / #2 " }
6871     {
6872       \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
6873       {#3} {#4} {#5} {#6}
6874       \msg_see_documentation_text:n { LaTeX3 }
6875     }
6876     {
6877       \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 }
6878       {#3} {#4} {#5} {#6}
6879     }
6880   }
6881   {
6882     \msg_interrupt:xxx
6883     { \msg_error_text:n { LaTeX } : ~ " #1 / #2 " }
6884     {
6885       \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }

```

```

6886         {#3} {#4} {#5} {#6}
6887         \msg_see_documentation_text:n { LaTeX3 }
6888     }
6889     { }
6890 }
6891 }
6892 \cs_new_protected:Npn \msg_kernel_error:nnxxx #1#2#3#4#5
6893 { \msg_kernel_error:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
6894 \cs_set_protected:Npn \msg_kernel_error:nnxx #1#2#3#4
6895 { \msg_kernel_error:nnxxxx {#1} {#2} {#3} {#4} { } { } }
6896 \cs_set_protected:Npn \msg_kernel_error:nnx #1#2#3
6897 { \msg_kernel_error:nnxxxx {#1} {#2} {#3} { } { } { } }
6898 \cs_set_protected:Npn \msg_kernel_error:nn #1#2
6899 { \msg_kernel_error:nnxxxx {#1} {#2} { } { } { } { } }

```

(End definition for `\msg_kernel_error:nnxxxx`. This function is documented on page 166.)

`\msg_kernel_warning:nnxxxx` Kernel messages which can be redirected.

```

\msg_kernel_warning:nnxxxx
\msg_kernel_warning:nnxxxx
\msg_kernel_warning:nnxx
\msg_kernel_warning:nnx
\msg_kernel_warning:nn
\msg_kernel_info:nnxxxx
\msg_kernel_info:nnxxx
\msg_kernel_info:nnxx
\msg_kernel_info:nnx
\msg_kernel_info:nn

6900 \prop_new:N \l_msg_redirect_kernel_warning_prop
6901 \cs_new_protected:Npn \msg_kernel_warning:nnxxxx #1#2#3#4#5#6
6902 {
6903     \msg_use:nnnnxxxx { warning }
6904     {
6905         \msg_term:x
6906         {
6907             \msg_warning_text:n { LaTeX } : ~ " #1 / #2 " \\ \\
6908             \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
6909             {#3} {#4} {#5} {#6}
6910         }
6911     }
6912     { LaTeX } { #1 / #2 } {#3} {#4} {#5} {#6}
6913 }
6914 \cs_new_protected:Npn \msg_kernel_warning:nnxxx #1#2#3#4#5
6915 { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
6916 \cs_new_protected:Npn \msg_kernel_warning:nnxx #1#2#3#4
6917 { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} {#4} { } { } }
6918 \cs_new_protected:Npn \msg_kernel_warning:nnx #1#2#3
6919 { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} { } { } { } }
6920 \cs_new_protected:Npn \msg_kernel_warning:nn #1#2
6921 { \msg_kernel_warning:nnxxxx {#1} {#2} { } { } { } { } }
6922 \prop_new:N \l_msg_redirect_kernel_info_prop
6923 \cs_new_protected:Npn \msg_kernel_info:nnxxxx #1#2#3#4#5#6
6924 {
6925     \msg_use:nnnnxxxx { info }
6926     {
6927         \msg_log:x
6928         {
6929             \msg_info_text:n { LaTeX } : ~ " #1 / #2 " \\ \\
6930             \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }

```

```

6931         {#3} {#4} {#5} {#6}
6932     }
6933 }
6934 { LaTeX } { #1 / #2 } {#3} {#4} {#5} {#6}
6935 }
6936 \cs_new_protected:Npn \msg_kernel_info:nnxxx #1#2#3#4#5
6937 { \msg_kernel_info:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
6938 \cs_new_protected:Npn \msg_kernel_info:nnxx #1#2#3#4
6939 { \msg_kernel_info:nnxxxx {#1} {#2} {#3} {#4} { } { } }
6940 \cs_new_protected:Npn \msg_kernel_info:nnx #1#2#3
6941 { \msg_kernel_info:nnxxxx {#1} {#2} {#3} { } { } { } }
6942 \cs_new_protected:Npn \msg_kernel_info:nn #1#2
6943 { \msg_kernel_info:nnxxxx {#1} {#2} { } { } { } { } }

```

(End definition for `\msg_kernel_warning:nnxxxx`. This function is documented on page 167.)

Error messages needed to actually implement the message system itself.

```

6944 \msg_kernel_new:nnnn { msg } { message-already-defined }
6945 { Message~'#2'~for~module~'#1'~already~defined. }
6946 {
6947     \c_msg_coding_error_text_tl
6948     LaTeX~was~asked~to~define~a~new~message~called~'#2'
6949     by~the~module~'#1'~module:~\
6950     this~message~already~exists.
6951     \c_msg_return_text_tl
6952 }
6953 \msg_kernel_new:nnnn { msg } { message-unknown }
6954 { Unknown~message~'#2'~for~module~'#1'. }
6955 {
6956     \c_msg_coding_error_text_tl
6957     LaTeX~was~asked~to~display~a~message~called~'#2'~\
6958     by~the~module~'#1'~module:~this~message~does~not~exist.
6959     \c_msg_return_text_tl
6960 }
6961 \msg_kernel_new:nnnn { msg } { message-class-unknown }
6962 { Unknown~message~class~'#1'. }
6963 {
6964     LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':~\
6965     this~was~never~defined.
6966     \c_msg_return_text_tl
6967 }
6968 \msg_kernel_new:nnnn { msg } { redirect-loop }
6969 { Message~redirection~loop~for~message~class~'#1'. }
6970 {
6971     LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.~\
6972     The~original~message~here~has~been~lost.
6973     \c_msg_return_text_tl
6974 }

```

Messages for earlier kernel modules.

```

6975 \msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
6976 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
6977 {
6978   \c_msg_coding_error_text_tl
6979   LaTeX~has~been~asked~to~define~a~function~'#1'~with~
6980   #2~arguments. \\
6981   TeX~allows~between~0~and~9~arguments~for~a~single~function.
6982 }
6983 \msg_kernel_new:nnnn { kernel } { command-already-defined }
6984 { Control~sequence~#1~already~defined. }
6985 {
6986   \c_msg_coding_error_text_tl
6987   LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
6988   but~this~name~has~already~been~used~elsewhere. \\ \\
6989   The~current~meaning~is:\\
6990   \ \ #2
6991 }
6992 \msg_kernel_new:nnnn { kernel } { command-not-defined }
6993 { Control~sequence~#1~undefined. }
6994 {
6995   \c_msg_coding_error_text_tl
6996   LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
6997   been~defined~yet.
6998 }
6999 \msg_kernel_new:nnnn { kernel } { variable-not-defined }
7000 { Variable~#1~undefined. }
7001 {
7002   \c_msg_coding_error_text_tl
7003   LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
7004   been~defined~yet.
7005 }
7006 \msg_kernel_new:nnnn { seq } { empty-sequence }
7007 { Empty~sequence~#1. }
7008 {
7009   \c_msg_coding_error_text_tl
7010   LaTeX~has~been~asked~to~recover~an~entry~from~a~sequence~that~
7011   has~no~content:~that~cannot~happen!
7012 }

```

```

\msg_kernel_bug:x
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl

```

The L^AT_EX coding bug error gets re-visited here.

```

7013 \cs_set_protected:Npn \msg_kernel_bug:x #1
7014 {
7015   \msg_interrupt:xxx { \c_msg_kernel_bug_text_tl }
7016   {
7017     #1
7018     \msg_see_documentation_text:n { LaTeX3 }
7019   }
7020   { \c_msg_kernel_bug_more_text_tl }
7021 }

```

```

7022 \tl_const:Nn \c_msg_kernel_bug_text_tl
7023 { This~is~a~LaTeX~bug:~check~coding! }
7024 \tl_const:Nn \c_msg_kernel_bug_more_text_tl
7025 {
7026   There~is~a~coding~bug~somewhere~around~here. \\
7027   This~probably~needs~examining~by~an~expert.
7028   \c_msg_return_text_tl
7029 }

```

(End definition for `\msg_kernel_bug:x`. This function is documented on page ??.)

179.5 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\msg_class_new:nn` This is only ever used in a `set` fashion.

```

7030 \cs_new_eq:NN \msg_class_new:nn \msg_class_set:nn

```

(End definition for `\msg_class_new:nn`. This function is documented on page ??.)

`\msg_trace:nnxxxx` The performance here is never going to be good enough for tracing code, so let's be realistic.

```

\msg_trace:nnxxx
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nnx
\msg_trace:nn
7031 \cs_new_eq:NN \msg_trace:nnxxxx \msg_log:nnxxxx
7032 \cs_new_eq:NN \msg_trace:nnxxx \msg_log:nnxxx
7033 \cs_new_eq:NN \msg_trace:nnxx \msg_log:nnxx
7034 \cs_new_eq:NN \msg_trace:nnx \msg_log:nnx
7035 \cs_new_eq:NN \msg_trace:nn \msg_log:nn

```

(End definition for `\msg_trace:nnxxxx` and others. These functions are documented on page ??.)

`\msg_generic_new:nnn` These were all too low-level.

```

\msg_generic_new:nn
\msg_generic_set:nnn
\msg_generic_set:nn
\msg_direct_interrupt:xxxxx
\msg_direct_log:xx
\msg_direct_term:xx
7036 \cs_new_protected:Npn \msg_generic_new:nnn #1#2#3 { \deprecated }
7037 \cs_new_protected:Npn \msg_generic_new:nn #1#2 { \deprecated }
7038 \cs_new_protected:Npn \msg_generic_set:nnn #1#2#3 { \deprecated }
7039 \cs_new_protected:Npn \msg_generic_set:nn #1#2 { \deprecated }
7040 \cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 { \deprecated }
7041 \cs_new_protected:Npn \msg_direct_log:xx #1#2 { \deprecated }
7042 \cs_new_protected:Npn \msg_direct_term:xx #1#2 { \deprecated }

```

(End definition for `\msg_generic_new:nnn`. This function is documented on page ??.)

```

7043 \</initex | package>

```


180 l3keyval implementation

```

7044 <*initex | package>
7045 <*package>
7046 \ProvidesExplPackage
7047   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7048 \package_check_loaded_expl:
7049 </package>

```

`\g_keyval_level_int` For nesting purposes an integer is needed for the current level.

```

7050 \int_new:N \g_keyval_level_int

```

`\l_keyval_key_tl` The current key name and value.
`\l_keyval_value_tl`

```

7051 \tl_new:N \l_keyval_key_tl
7052 \tl_new:N \l_keyval_value_tl

```

`\l_keyval_sanitise_tl` Token list variables for dealing with awkward category codes in the input.
`\l_keyval_parse_tl`

```

7053 \tl_new:N \l_keyval_sanitise_tl
7054 \tl_new:N \l_keyval_parse_tl

```

`\keyval_parse:n` The parsing function first deals with the category codes for = and ,, so that there are no odd events. The input is then handed off to the element by element system.

```

7055 \group_begin:
7056   \char_set_catcode_active:n { '=' }
7057   \char_set_catcode_active:n { ',' }
7058   \char_set_lccode:nn { '\8' } { '=' }
7059   \char_set_lccode:nn { '\9' } { ',' }
7060   \tl_to_lowercase:n
7061   {
7062     \group_end:
7063     \cs_new_protected:Npn \keyval_parse:n #1
7064     {
7065       \group_begin:
7066         \tl_clear:N \l_keyval_sanitise_tl
7067         \tl_set:Nn \l_keyval_sanitise_tl {#1}
7068         \tl_replace_all_in:Nnn \l_keyval_sanitise_tl { = } { 8 }
7069         \tl_replace_all_in:Nnn \l_keyval_sanitise_tl { , } { 9 }
7070         \tl_clear:N \l_keyval_parse_tl
7071         \exp_after:wN \keyval_parse_elt:w \exp_after:wN
7072         \q_no_value \l_keyval_sanitise_tl 9 \q_nil 9
7073         \exp_after:wN \group_end:
7074         \l_keyval_parse_tl
7075       }
7076     }

```

(End definition for `\keyval_parse:n`. This function is documented on page ??.)

`\keyval_parse_elt:w` Each item to be parsed will have `\q_no_value` added to the front. Hence the blank test here can always be used to find a totally empty argument. If this is the case, the system loops round. If there is something to parse, there is a check for the `\q_nil` marker and if not a hand-off.

```

7077 \cs_new_protected:Npn \keyval_parse_elt:w #1 ,
7078 {
7079   \tl_if_blank:oTF { \use_none:n #1 }
7080   { \keyval_parse_elt:w \q_no_value }
7081   {
7082     \quark_if_nil:oF { \use_ii:nn #1 }
7083     {
7084       \keyval_split_key_value:w #1 = = \q_stop
7085       \keyval_parse_elt:w \q_no_value
7086     }
7087   }
7088 }

```

(End definition for `\keyval_parse_elt:w`. This function is documented on page ??.)

`\keyval_split_key_value:w` The key and value are handled separately. First the key is grabbed and saved as `\l_keyval_key_tl`. Then a check is need to see if there is a value at all: if not then the key name is simply added to the output. If there is a value then there is a check to ensure that there was only one `=` in the input (remembering some extra ones are around at the moment to prevent errors). All being well, there is an hand-off to find the value: the `\q_nil` is there to prevent loss of braces.

```

7089 \cs_new_protected:Npn \keyval_split_key_value:w #1 = #2 \q_stop
7090 {
7091   \keyval_split_key:w #1 \q_stop
7092   \str_if_eq:nnTF {#2} { = }
7093   {
7094     \tl_put_right:Nx \l_keyval_parse_tl
7095     {
7096       \exp_not:c { keyval_key_no_value_elt_ \int_use:N \g_keyval_level_int :n }
7097       { \exp_not:o \l_keyval_key_tl }
7098     }
7099   }
7100   {
7101     \keyval_split_key_value_aux:wTF #2 \q_no_value \q_stop
7102     { \keyval_split_value:w \q_nil #2 }
7103     { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
7104   }
7105 }
7106 \cs_new:Npn \keyval_split_key_value_aux:wTF #1 = #2#3 \q_stop
7107 { \tl_if_head_eq_meaning:nNTF {#3} \q_no_value }

```

(End definition for `\keyval_split_key_value:w`. This function is documented on page ??.)

`\keyval_split_key:w` The aim here is to remove spaces and also exactly one set of braces. The spaces are trimmed off from each end using a “funny” Q, which will never turn up in normal use.
`\keyval_remove_spaces:w`
`\keyval_split_key_aux:w`
`\keyval_remove_spaces_aux:w`

The idea is that the f-type expansion will stop if it finds an unexpandable token or a space, and will gobble the space. To avoid expanding anything else, the `\exp_not:N` works by ensuring that the first non-space token in the setting will stop the f-type expansion. The `\use_none:n` is needed to remove the leading quark, while the second setting of `\l_keyval_key_tl` removes exactly one set of braces.

```

7108 \group_begin:
7109   \char_set_catcode_math_toggle:n { '\Q }
7110   \cs_new_protected:Npn \keyval_split_key:w #1 \q_stop
7111     {
7112       \exp_args:NNf \tl_set:Nn \l_keyval_key_tl
7113       {
7114         \exp_after:wN \keyval_remove_spaces:w \exp_after:wN
7115         \exp_not:N \use_none:n #1 Q ~ Q
7116       }
7117       \tl_set:Nx \l_keyval_key_tl
7118       { \exp_after:wN \keyval_split_key_aux:w \l_keyval_key_tl \q_stop }
7119     }
7120   \cs_gset:Npn \keyval_split_key_aux:w #1 \q_stop { \exp_not:n {#1} }
7121   \cs_gset:Npn \keyval_remove_spaces:w #1 ~ Q { \keyval_remove_spaces_aux:w #1 Q }
7122   \cs_gset:Npn \keyval_remove_spaces_aux:w #1 Q #2 {#1}
7123 \group_end:

```

(End definition for `\keyval_split_key:w`. This function is documented on page ??.)

`\keyval_split_value:w` Here the value has to be separated from the equals signs and the leading `\q_nil` added in to keep the brace levels. First the processing function can be added to the output list. If there is no value, setting `\l_keyval_value_tl` with three groups removed will leave nothing at all, and so an empty group can be added to the parsed list. On the other hand, if the value is entirely contained within a set of braces then `\l_keyval_value_tl` will contain `\q_nil` only. In that case, strip off the leading quark using `\use_ii:nnn`, which also deals with any spaces.

```

7124 \cs_new_protected:Npn \keyval_split_value:w #1 = =
7125 {
7126   \tl_put_right:Nx \l_keyval_parse_tl
7127   {
7128     \exp_not:c { keyval_key_value_elt_ \int_use:N \g_keyval_level_int :nn }
7129     { \exp_not:o \l_keyval_key_tl }
7130   }
7131   \tl_set:Nx \l_keyval_value_tl { \exp_not:o { \use_none:nnn #1 \q_nil \q_nil } }
7132   \tl_if_empty:NTF \l_keyval_value_tl
7133   { \tl_put_right:Nn \l_keyval_parse_tl { { } } }
7134   {
7135     \quark_if_nil:NTF \l_keyval_value_tl
7136     {
7137       \tl_put_right:Nx \l_keyval_parse_tl
7138       { { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
7139     }
7140   }

```

```

7140         { \keyval_split_value_aux:w #1 \q_stop }
7141     }
7142 }

```

A similar idea to the key code: remove the spaces from each end and deal with one set of braces.

```

7143 \group_begin:
7144   \char_set_catcode_math_toggle:n { '\Q }
7145   \cs_new_protected:Npn \keyval_split_value_aux:w \q_nil #1 \q_stop
7146   {
7147     \exp_args:NNf \tl_set:Nn \l_keyval_value_tl
7148     { \keyval_remove_spaces:w \exp_not:N #1 Q ~ Q }
7149     \tl_put_right:Nx \l_keyval_parse_tl { { \exp_not:o \l_keyval_value_tl } }
7150   }
7151 \group_end:

```

(End definition for `\keyval_split_value:w`. This function is documented on page ??.)

`\keyval_parse:NNn` The outer parsing routine just sets up the processing functions and hands off.

```

7152 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
7153 {
7154   \int_gincr:N \g_keyval_level_int
7155   \cs_gset_eq:cN { keyval_key_no_value_elt_ \int_use:N \g_keyval_level_int :n } #1
7156   \cs_gset_eq:cN { keyval_key_value_elt_ \int_use:N \g_keyval_level_int :nn } #2
7157   \keyval_parse:n {#3}
7158   \int_gdecr:N \g_keyval_level_int
7159 }

```

(End definition for `\keyval_parse:NNn`. This function is documented on page 168.)

One message for the low level parsing system.

```

7160 \msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
7161 { Misplaced-equals-sign~in~key-value-input~\msg_line_number: }
7162 {
7163   LaTeX-is~attempting~to~parse~some~key-value-input~but~found~
7164   two~equals~signs~not~separated~by~a~comma.
7165 }

```

180.1 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\KV_process_space_removal_sanitize:NNn` There is just one function for this now.

`\KV_process_space_removal_no_sanitize:NNn`

`\KV_process_no_space_removal_no_sanitize:NNn`

```

7166 \cs_new_eq:NN \KV_process_space_removal_sanitize:NNn \keyval_parse:NNn
7167 \cs_new_eq:NN \KV_process_space_removal_no_sanitize:NNn \keyval_parse:NNn
7168 \cs_new_eq:NN \KV_process_no_space_removal_no_sanitize:NNn \keyval_parse:NNn

```

(End definition for `\KV_process_space_removal_sanitiz:NNn`. This function is documented on page ??.)

```
7169 </initex | package>
```

181 l3keys Implementation

```
7170 <*initex | package>
7171 <*package>
7172 \ProvidesExplPackage
7173   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7174 \package_check_loaded_expl:
7175 </package>
```

181.1 Constants and variables

<code>\c_keys_code_root_tl</code>	The prefixes for the code and variables of the keys themselves.
<code>\c_keys_vars_root_tl</code>	<pre>7176 \tl_const:Nn \c_keys_code_root_tl { key~code~>~ } 7177 \tl_const:Nn \c_keys_vars_root_tl { key~var~>~ }</pre>
<code>\c_keys_props_root_tl</code>	The prefix for storing properties.
	<pre>7178 \tl_const:Nn \c_keys_props_root_tl { key~prop~>~ }</pre>
<code>\c_keys_value_forbidden_tl</code>	Two marker token lists.
<code>\c_keys_value_required_tl</code>	<pre>7179 \tl_const:Nn \c_keys_value_forbidden_tl { forbidden } 7180 \tl_const:Nn \c_keys_value_required_tl { required }</pre>
<code>\l_keys_choice_int</code>	Publicly accessible data on which choice is being used when several are generated as a set.
<code>\l_keys_choices_tl</code>	<pre>7181 \int_new:N \l_keys_choice_int 7182 \tl_new:N \l_keys_choices_tl</pre>
<code>\l_keys_key_tl</code>	The name of a key itself: needed when setting keys.
	<pre>7183 \tl_new:N \l_keys_key_tl</pre>
<code>\l_keys_module_tl</code>	The module for an entire set of keys.
	<pre>7184 \tl_new:N \l_keys_module_tl</pre>
<code>\l_keys_no_value_bool</code>	A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.
	<pre>7185 \bool_new:N \l_keys_no_value_bool</pre>

`\l_keys_path_tl` The “path” of the current key is stored here: this is available to the programmer and so is public.

```
7186 \tl_new:N \l_keys_path_tl
```

`\l_keys_property_tl` The “property” begin set for a key at definition time is stored here.

```
7187 \tl_new:N \l_keys_property_tl
```

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

```
7188 \tl_new:N \l_keys_value_tl
```

181.2 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting.

`\keys_define_aux:nnn`
`\keys_define_aux:onn`

```
7189 \cs_new_protected:Npn \keys_define:nn
7190   { \keys_define_aux:onn \l_keys_module_tl }
7191 \cs_new_protected:Npn \keys_define_aux:nnn #1#2#3
7192   {
7193     \tl_set:Nx \l_keys_module_tl { \tl_to_str:n {#2} }
7194     \keyval_parse:NNn \keys_define_elt:n \keys_define_elt:nn {#3}
7195     \tl_set:Nn \l_keys_module_tl {#1}
7196   }
7197 \cs_generate_variant:Nn \keys_define_aux:nnn { o }
```

(End definition for `\keys_define:nn`. This function is documented on page 170.)

`\keys_define_elt:n` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

`\keys_define_elt:nn`
`\keys_define_elt_aux:nnn`

```
7198 \cs_new_protected_nopar:Npn \keys_define_elt:n #1
7199   {
7200     \bool_set_true:N \l_keys_no_value_bool
7201     \keys_define_elt_aux:nn {#1} { }
7202   }
7203 \cs_new_protected:Npn \keys_define_elt:nn #1#2
7204   {
7205     \bool_set_false:N \l_keys_no_value_bool
7206     \keys_define_elt_aux:nn {#1} {#2}
7207   }
7208 \cs_new_protected:Npn \keys_define_elt_aux:nn #1#2 {
7209   \keys_property_find:n {#1}
7210   \cs_if_exist:cTF { \c_keys_props_root_tl \l_keys_property_tl }
```

```

7211 { \keys_define_key:n {#2} }
7212 {
7213   \msg_kernel_error:nxxx { keys } { property-unknown }
7214   { \l_keys_property_tl } { \l_keys_path_tl }
7215 }
7216 }

```

(End definition for \keys_define_elt:n. This function is documented on page ??.)

\keys_property_find:n Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

```

7217 \cs_new_protected_nopar:Npn \keys_property_find:n #1
7218 {
7219   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
7220   \tl_if_in:nnTF {#1} { . }
7221   { \keys_property_find_aux:w #1 \q_stop }
7222   { \msg_kernel_error:nxx { keys } { key-no-property } {#1} }
7223 }
7224 \cs_new_protected_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop
7225 {
7226   \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
7227   \tl_if_in:nnTF {#2} { . }
7228   {
7229     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
7230     \keys_property_find_aux:w #2 \q_stop
7231   }
7232   { \tl_set:Nn \l_keys_property_tl { . #2 } }
7233 }

```

(End definition for \keys_property_find:n. This function is documented on page ??.)

\keys_define_key:n Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it is missing the earlier tests will have failed.

```

7234 \cs_new_protected:Npn \keys_define_key:n #1
7235 {
7236   \bool_if:NTF \l_keys_no_value_bool
7237   {
7238     \exp_after:wN \keys_define_key_aux:w
7239     \l_keys_property_tl \q_stop
7240     { \use:c { \c_keys_props_root_tl \l_keys_property_tl } }
7241     {
7242       \msg_kernel_error:nxxx { keys }
7243       { property-requires-value } { \l_keys_property_tl }
7244       { \l_keys_path_tl }
7245     }

```

```

7246     }
7247     { \use:c { \c_keys_props_root_tl \l_keys_property_tl } {#1} }
7248   }
7249   \cs_new_protected:Npn \keys_define_key_aux:w #1 : #2 \q_stop
7250     { \tl_if_empty:nTF {#2} }

```

(End definition for `\keys_define_key:n`. This function is documented on page ??.)

181.3 Turning properties into actions

`\keys_bool_set:NN` Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

7251 \cs_new_nopar:Npn \keys_bool_set:NN #1#2
7252 {
7253   \cs_if_exist:NF #1 { \bool_new:N #1 }
7254   \keys_choice_make:
7255   \keys_cmd_set:nx { \l_keys_path_tl / true }
7256     { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
7257   \keys_cmd_set:nx { \l_keys_path_tl / false }
7258     { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
7259   \keys_cmd_set:nn { \l_keys_path_tl / unknown }
7260     {
7261       \msg_kernel_error:nxx { keys } { boolean-values-only }
7262       { \l_keys_key_tl }
7263     }
7264   \keys_default_set:n { true }
7265 }

```

(End definition for `\keys_bool_set:NN`. This function is documented on page ??.)

`\keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key.

```

7266 \cs_new_protected_nopar:Npn \keys_choice_make:
7267 {
7268   \keys_cmd_set:nn { \l_keys_path_tl }
7269     { \keys_choice_find:n {##1} }
7270   \keys_cmd_set:nn { \l_keys_path_tl / unknown }
7271     {
7272       \msg_kernel_error:nxxx { keys } { choice-unknown }
7273       { \l_keys_path_tl } {##1}
7274     }
7275 }

```

(End definition for `\keys_choice_make:.` This function is documented on page ??.)

`\keys_choices_generate:n` Creating multiple-choices means setting up the “indicator” code, then applying whatever the user wanted.


```

7276 \cs_new_protected:Npn \keys_choices_generate:n #1
7277 {
7278   \cs_if_exist:cTF
7279   { \c_keys_vars_root_tl \l_keys_path_tl .choice-code }
7280   {
7281     \keys_choice_make:
7282     \int_zero:N \l_keys_choice_int
7283     \clist_map_function:nN {#1} \keys_choices_generate_aux:n
7284   }
7285   {
7286     \msg_kernel_error:nxx { keys }
7287     { generate-choices-before-code } { \l_keys_path_tl }
7288   }
7289 }
7290 \cs_new_protected_nopar:Npn \keys_choices_generate_aux:n #1
7291 {
7292   \keys_cmd_set:nx { \l_keys_path_tl / #1 }
7293   {
7294     \exp_not:n { \tl_set:Nn \l_keys_choice_tl } {#1}
7295     \exp_not:n { \int_set:Nn \l_keys_choice_int }
7296     { \int_use:N \l_keys_choice_int }
7297     \exp_not:v
7298     { \c_keys_vars_root_tl \l_keys_path_tl .choice-code }
7299   }
7300   \int_incr:N \l_keys_choice_int
7301 }

```

(End definition for \keys_choices_generate:n. This function is documented on page ??.)

\keys_choice_code_store:x The code for making multiple choices is stored in a token list.

```

7302 \cs_new_protected:Npn \keys_choice_code_store:x #1
7303 {
7304   \cs_if_exist:cF
7305   { \c_keys_vars_root_tl \l_keys_path_tl .choice-code }
7306   {
7307     \tl_new:c
7308     { \c_keys_vars_root_tl \l_keys_path_tl .choice-code }
7309   }
7310   \tl_set:cx { \c_keys_vars_root_tl \l_keys_path_tl .choice-code }
7311   {#1}
7312 }

```

(End definition for \keys_choice_code_store:x. This function is documented on page ??.)

\keys_cmd_set:nn Creating a new command means tidying up the properties and then making the internal function which actually does the work.
\keys_cmd_set:nx
\keys_cmd_set_aux:n

```

7313 \cs_new_protected:Npn \keys_cmd_set:nn #1#2
7314 {

```

```

7315     \keys_cmd_set_aux:n {#1}
7316     \cs_set:cpn { \c_keys_code_root_tl #1 } ##1 {#2}
7317   }
7318   \cs_new_protected:Npn \keys_cmd_set:nx #1#2
7319   {
7320     \keys_cmd_set_aux:n {#1}
7321     \cs_set:cpx { \c_keys_code_root_tl #1 } ##1 {#2}
7322   }
7323   \cs_new_protected_nopar:Npn \keys_cmd_set_aux:n #1
7324   {
7325     \tl_clear_new:c { \c_keys_vars_root_tl #1 .default }
7326     \tl_set:cn { \c_keys_vars_root_tl #1 .default } { \q_no_value }
7327     \tl_clear_new:c { \c_keys_vars_root_tl #1 .req }
7328   }

```

(End definition for `\keys_cmd_set:nn` and `\keys_cmd_set:nx`. These functions are documented on page ??.)

`\keys_default_set:n` Setting a default value is easy.
`\keys_default_set:V`

```

7329   \cs_new_protected:Npn \keys_default_set:n #1
7330   { \tl_set:cn { \c_keys_vars_root_tl \l_keys_path_tl .default } {#1} }
7331   \cs_generate_variant:Nn \keys_default_set:n { V }

```

(End definition for `\keys_default_set:n` and `\keys_default_set:V`. These functions are documented on page ??.)

`\keys_meta_make:n` To create a meta-key, simply set up to pass data through.
`\keys_meta_make:x`

```

7332   \cs_new_protected_nopar:Npn \keys_meta_make:n #1
7333   {
7334     \exp_args:NNo \keys_cmd_set:nn \l_keys_path_tl
7335     { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }
7336   }
7337   \cs_new_protected_nopar:Npn \keys_meta_make:x #1
7338   {
7339     \keys_cmd_set:nx { \l_keys_path_tl }
7340     { \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1} }
7341   }

```

(End definition for `\keys_meta_make:n` and `\keys_meta_make:x`. These functions are documented on page ??.)

`\keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```

7342   \cs_new_protected_nopar:Npn \keys_value_requirement:n #1
7343   {
7344     \tl_set_eq:cc
7345     { \c_keys_vars_root_tl \l_keys_path_tl .req }
7346     { c_keys_value_ #1 _tl }
7347   }

```

(End definition for `\keys_value_requirement:n`. This function is documented on page ??.)

`\keys_variable_set:NnNN` Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed. The three-argument version is set up so that the use of `{ }` as an N-type variable is only done once!

`\keys_variable_set:cnNN`

`\keys_variable_set:NnN`

`\keys_variable_set:cnN`

```

7348 \cs_new_protected_nopar:Npn \keys_variable_set:NnNN #1#2#3#4
7349 {
7350   \cs_if_exist:NF #1 { \use:c { #2 _new:N } #1 }
7351   \keys_cmd_set:nx { \l_keys_path_tl }
7352   { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1} }
7353 }
7354 \cs_new_protected_nopar:Npn \keys_variable_set:NnN #1#2#3
7355 { \keys_variable_set:NnNN #1 {#2} { } #3 }
7356 \cs_generate_variant:Nn \keys_variable_set:NnNN { c }
7357 \cs_generate_variant:Nn \keys_variable_set:NnN { c }

```

(End definition for `\keys_variable_set:NnNN` and `\keys_variable_set:cnNN`. These functions are documented on page ??.)

181.4 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

`.bool_set:N` One function for this.

`.bool_gset:N`

```

7358 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .bool_set:N } #1
7359 { \keys_bool_set:NN #1 { } }
7360 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .bool_gset:N } #1
7361 { \keys_bool_set:NN #1 g }

```

(End definition for `.bool_set:N`. This function is documented on page 171.)

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

7362 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .choice: }
7363 { \keys_choice_make: }

```

(End definition for `.choice:.` This function is documented on page 171.)

`.code:n` Creating code is simply a case of passing through to the underlying `set` function.

`.code:x`

```

7364 \cs_new_protected:cpn { \c_keys_props_root_tl .code:n } #1
7365 { \keys_cmd_set:nn { \l_keys_path_tl } {#1} }
7366 \cs_new_protected:cpn { \c_keys_props_root_tl .code:x } #1
7367 { \keys_cmd_set:nx { \l_keys_path_tl } {#1} }

```

(End definition for `.code:n` and `.code:x`. These functions are documented on page 171.)

.choice_code:n Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

.choice_code:x

```
7368 \cs_new_protected:cpn { \c_keys_props_root_tl .choice_code:n } #1
7369 { \keys_choice_code_store:x { \exp_not:n {#1} } }
7370 \cs_new_protected:cpn { \c_keys_props_root_tl .choice_code:x } #1
7371 { \keys_choice_code_store:x {#1} }
```

(End definition for `.choice_code:n` and `.choice_code:x`. These functions are documented on page 171.)

.default:n Expansion is left to the internal functions.

.default:V

```
7372 \cs_new_protected:cpn { \c_keys_props_root_tl .default:n } #1
7373 { \keys_default_set:n {#1} }
7374 \cs_new_protected:cpn { \c_keys_props_root_tl .default:V } #1
7375 { \keys_default_set:V #1 }
```

(End definition for `.default:n` and `.default:V`. These functions are documented on page 171.)

.dim_set:N Setting a variable is very easy: just pass the data along.

.dim_set:c

.dim_gset:N

.dim_gset:c

```
7376 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_set:N } #1
7377 { \keys_variable_set:NnN #1 { dim } n }
7378 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_set:c } #1
7379 { \keys_variable_set:cnN {#1} { dim } n }
7380 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_gset:N } #1
7381 { \keys_variable_set:NnNN #1 { dim } g n }
7382 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_gset:c } #1
7383 { \keys_variable_set:cnNN {#1} { dim } g n }
```

(End definition for `.dim_set:N` and `.dim_set:c`. These functions are documented on page 172.)

.fp_set:N Setting a variable is very easy: just pass the data along.

.fp_set:c

.fp_gset:N

.fp_gset:c

```
7384 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_set:N } #1
7385 { \keys_variable_set:NnN #1 { fp } n }
7386 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_set:c } #1
7387 { \keys_variable_set:cnN {#1} { fp } n }
7388 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_gset:N } #1
7389 { \keys_variable_set:NnNN #1 { fp } g n }
7390 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_gset:c } #1
7391 { \keys_variable_set:cnNN {#1} { fp } g n }
```

(End definition for `.fp_set:N` and `.fp_set:c`. These functions are documented on page 172.)

.generate_choices:n Making choices is easy.

```
7392 \cs_new_protected:cpn { \c_keys_props_root_tl .generate_choices:n } #1
7393 { \keys_choices_generate:n {#1} }
```

(End definition for `.generate_choices:n`. This function is documented on page 172.)

`.int_set:N` Setting a variable is very easy: just pass the data along.

`.int_set:c`

`.int_gset:N`

`.int_gset:c`

```

7394 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_set:N } #1
7395   { \keys_variable_set:NnN #1 { int } n }
7396 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_set:c } #1
7397   { \keys_variable_set:cnN {#1} { int } n }
7398 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_gset:N } #1
7399   { \keys_variable_set:NnNN #1 { int } g n }
7400 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_gset:c } #1
7401   { \keys_variable_set:cnNN {#1} { int } g n }

```

(End definition for `.int_set:N` and `.int_set:c`. These functions are documented on page 173.)

`.meta:n` Making a meta is handled internally.

`.meta:x`

```

7402 \cs_new_protected:cpn { \c_keys_props_root_tl .meta:n } #1
7403   { \keys_meta_make:n {#1} }
7404 \cs_new_protected:cpn { \c_keys_props_root_tl .meta:x } #1
7405   { \keys_meta_make:x {#1} }

```

(End definition for `.meta:n` and `.meta:x`. These functions are documented on page 173.)

`.skip_set:N` Setting a variable is very easy: just pass the data along.

`.skip_set:c`

`.skip_gset:N`

`.skip_gset:c`

```

7406 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_set:N } #1
7407   { \keys_variable_set:NnN #1 { skip } n }
7408 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_set:c } #1
7409   { \keys_variable_set:cnN {#1} { skip } n }
7410 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_gset:N } #1
7411   { \keys_variable_set:NnNN #1 { skip } g n }
7412 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_gset:c } #1
7413   { \keys_variable_set:cnNN {#1} { skip } g n }

```

(End definition for `.skip_set:N` and `.skip_set:c`. These functions are documented on page 173.)

`.tl_set:N` Setting a variable is very easy: just pass the data along.

`.tl_set:c`

`.tl_gset:N`

`.tl_gset:c`

`.tl_set_x:N`

`.tl_set_x:c`

`.tl_gset_x:N`

`.tl_gset_x:c`

```

7414 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set:N } #1
7415   { \keys_variable_set:NnN #1 { tl } n }
7416 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set:c } #1
7417   { \keys_variable_set:cnN {#1} { tl } n }
7418 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set_x:N } #1
7419   { \keys_variable_set:NnN #1 { tl } x }
7420 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set_x:c } #1
7421   { \keys_variable_set:cnN {#1} { tl } x }
7422 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset:N } #1
7423   { \keys_variable_set:NnNN #1 { tl } g n }
7424 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset:c } #1
7425   { \keys_variable_set:cnNN {#1} { tl } g n }
7426 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset_x:N } #1
7427   { \keys_variable_set:NnNN #1 { tl } g x }
7428 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset_x:c } #1
7429   { \keys_variable_set:cnNN {#1} { tl } g x }

```

(End definition for `.tl_set:N` and `.tl_set:c`. These functions are documented on page 174.)

`.value_forbidden:` These are very similar, so both call the same function.
`.value_required:`

```
7430 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .value_forbidden: }
7431 { \keys_value_requirement:n { forbidden } }
7432 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .value_required: }
7433 { \keys_value_requirement:n { required } }
```

(End definition for `.value_forbidden:`. This function is documented on page 174.)

181.5 Setting keys

`\keys_set:nn` A simple wrapper again.

```
\keys_set:nV
\keys_set:nv
\keys_set:no
\keys_set_aux:nnn
\keys_set_aux:onn
7434 \cs_new_protected:Npn \keys_set:nn
7435 { \keys_set_aux:onn { \l_keys_module_tl } }
7436 \cs_new_protected:Npn \keys_set_aux:nnn #1#2#3
7437 {
7438   \tl_set:Nn \l_keys_module_tl {#2}
7439   \keyval_parse:Nn \keys_set_elt:n \keys_set_elt:nn {#3}
7440   \tl_set:Nn \l_keys_module_tl {#1}
7441 }
7442 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
7443 \cs_generate_variant:Nn \keys_set_aux:nnn { o }
```

(End definition for `\keys_set:nn` and others. These functions are documented on page 176.)

`\keys_set_elt:n` A shared system once again. First, set the current path and add a default if needed.
`\keys_set_elt:nn` There are then checks to see if the a value is required or forbidden. If everything passes,
`\keys_set_elt_aux:nn` move on to execute the code.

```
7444 \cs_new_protected_nopar:Npn \keys_set_elt:n #1
7445 {
7446   \bool_set_true:N \l_keys_no_value_bool
7447   \keys_set_elt_aux:nn {#1} { }
7448 }
7449 \cs_new_protected:Npn \keys_set_elt:nn #1#2
7450 {
7451   \bool_set_false:N \l_keys_no_value_bool
7452   \keys_set_elt_aux:nn {#1} {#2}
7453 }
7454 \cs_new_protected:Npn \keys_set_elt_aux:nn #1#2
7455 {
7456   \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
7457   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
7458   \keys_value_or_default:n {#2}
7459   \bool_if:nTF
7460   {
```

```

7461     \keys_if_value_p:n { required } &&
7462     \l_keys_no_value_bool
7463   }
7464   {
7465     \msg_kernel_error:nnx { keys } { value-required }
7466     { \l_keys_path_tl }
7467   }
7468   {
7469     \bool_if:nTF
7470     {
7471       \keys_if_value_p:n { forbidden } &&
7472       ! \l_keys_no_value_bool
7473     }
7474     {
7475       \msg_kernel_error:nnxx { keys } { value-forbidden }
7476       { \l_keys_path_tl } { \l_keys_value_tl }
7477     }
7478     { \keys_execute: }
7479   }
7480 }

```

(End definition for `\keys_set_elt:n` and `\keys_set_elt:nn`. These functions are documented on page ??.)

`\keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```

7481 \cs_new_protected:Npn \keys_value_or_default:n #1
7482 {
7483   \tl_set:Nn \l_keys_value_tl {#1}
7484   \bool_if:NT \l_keys_no_value_bool
7485   {
7486     \quark_if_no_value:cF { \c_keys_vars_root_tl \l_keys_path_tl .default }
7487     {
7488       \tl_set_eq:Nc \l_keys_value_tl
7489       { \c_keys_vars_root_tl \l_keys_path_tl .default }
7490     }
7491   }
7492 }

```

(End definition for `\keys_value_or_default:n`. This function is documented on page ??.)

`\keys_if_value_p:n` To test if a value is required or forbidden. A simple check for the existence of the appropriate marker.

```

7493 \prg_new_conditional:Npnn \keys_if_value:n #1 { p }
7494 {
7495   \tl_if_eq:ccTF { c_keys_value_ #1 _tl }
7496   { \c_keys_vars_root_tl \l_keys_path_tl .req }
7497   { \prg_return_true: }
7498   { \prg_return_false: }
7499 }

```

(End definition for `\keys_if_value_p:n`. This function is documented on page ??.)

`\keys_execute:` Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain.
`\keys_execute_unknown:`
`\keys_execute:nn`

```

7500 \cs_new_nopar:Npn \keys_execute:
7501   { \keys_execute:nn { \l_keys_path_tl } { \keys_execute_unknown: } }
7502 \cs_new_nopar:Npn \keys_execute_unknown:
7503   {
7504     \keys_execute:nn { \l_keys_module_tl / unknown }
7505     {
7506       \msg_kernel_error:nxxx { keys } { key-unknown }
7507       { \l_keys_path_tl } { \l_keys_module_tl }
7508     }
7509   }
7510 \cs_new_nopar:Npn \keys_execute:nn #1#2
7511   {
7512     \cs_if_exist:cTF { \c_keys_code_root_tl #1 }
7513     {
7514       \exp_args:Nno \use:c { \c_keys_code_root_tl #1 }
7515       \l_keys_value_tl
7516     }
7517     {#2}
7518   }

```

(End definition for `\keys_execute:.` This function is documented on page ??.)

`\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.

```

7519 \cs_new_nopar:Npn \keys_choice_find:n #1
7520   {
7521     \keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
7522     { \keys_execute:nn { \l_keys_path_tl / unknown } { } }
7523   }

```

(End definition for `\keys_choice_find:n`. This function is documented on page ??.)

181.6 Utilities

`\keys_if_exist_p:nn` A utility for others to see if a key exists.
`\keys_if_exist:nnTF`

```

7524 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
7525   {
7526     \cs_if_exist:cTF { \c_keys_code_root_tl #1 / #2 }
7527     { \prg_return_true:}
7528     { \prg_return_false:}
7529   }

```


(End definition for `\keys_if_exist:nn`. These functions are documented on page 176.)

`\keys_show:nn` Showing a key is just a question of using the correct name.

```
7530 \cs_new_nopar:Npn \keys_show:nn #1#2
7531 { \cs_show:c { \c_keys_code_root_tl #1 / \tl_to_str:n {#2} } }
```

(End definition for `\keys_show:nn`. This function is documented on page 177.)

181.7 Messages

For when there is a need to complain.

```
7532 \msg_kernel_new:nnnn { keys } { boolean-values-only }
7533 { Key~'#1'~accepts~boolean~values~only. }
7534 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
7535 \msg_kernel_new:nnnn { keys } { choice-unknown }
7536 { Choice~'#2'~unknown~for~key~'#1'. }
7537 {
7538   The~key~'#1'~takes~a~limited~number~of~values.\\
7539   The~input~given,~'#2',~is~not~on~the~list~accepted.
7540 }
7541 \msg_kernel_new:nnnn { keys } { generate-choices-before-code }
7542 { No~code~available~to~generate~choices~for~key~'#1'. }
7543 {
7544   \c_msg_coding_error_text_tl
7545   Before~using~.generate_choices:n~the~code~should~be~defined~
7546   with~'.choice_code:n'~or~'.choice_code:x'.
7547 }
7548 \msg_kernel_new:nnnn { keys } { key-no-property }
7549 { No~property~given~in~definition~of~key~'#1'. }
7550 {
7551   \c_msg_coding_error_text_tl
7552   Inside~\keys_define:nn~each~key~name
7553   needs~a~property:~\\
7554   ~ ~ #1 .<property>~\\
7555   LaTeX~did~not~find~a~'. ' ~to~indicate~the~start~of~a~property.
7556 }
7557 \msg_kernel_new:nnnn { keys } { key-unknown }
7558 { The~key~'#1'~is~unknown~and~is~being~ignored. }
7559 {
7560   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
7561   Check~that~you~have~spelled~the~key~name~correctly.
7562 }
7563 \msg_kernel_new:nnnn { keys } { option-unknown }
7564 { Unknown~option~'#1'~for~package~#2. }
7565 {
7566   LaTeX~has~been~asked~to~set~an~option~called~'#1'~
7567   but~the~#2~package~has~not~created~an~option~with~this~name.
```

```

7568 }
7569 \msg_kernel_new:nnnn { keys } { property-requires-value }
7570 { The~property~'#1'~requires~a~value. }
7571 {
7572   \c_msg_coding_error_text_tl
7573   LaTeX~was~asked~to~set~property~'#2'~for~key~'#1'.\\
7574   No~value~was~given~for~the~property,~and~one~is~required.
7575 }
7576 \msg_kernel_new:nnnn { keys } { property-unknown }
7577 { The~key~property~'#1'~is~unknown. }
7578 {
7579   \c_msg_coding_error_text_tl
7580   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
7581   this~property~is~not~defined.
7582 }
7583 \msg_kernel_new:nnnn { keys } { value-forbidden }
7584 { The~key~'#1'~does~not~taken~a~value. }
7585 {
7586   The~key~'#1'~should~be~given~without~a~value.\\
7587   LaTeX~will~ignore~the~given~value~'#2'.
7588 }
7589 \msg_kernel_new:nnnn { keys } { value-required }
7590 { The~key~'#1'~requires~a~value. }
7591 {
7592   The~key~'#1'~must~have~a~value.\\
7593   No~value~was~present:~the~key~will~be~ignored.
7594 }
7595 </initex | package>

```

182 l3file implementation

The following test files are used for this code: *m3file001*.

```

7596 <*initex | package>

7597 <*package>
7598 \ProvidesExplPackage
7599   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7600 \package_check_loaded_expl:
7601 </package>

```

\g_file_current_name_tl The name of the current file should be available at all times.

```

7602 \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.

```

7603 <*initex>
7604 \tex_everyjob:D \exp_after:wN
7605 {
7606   \tex_the:D \tex_everyjob:D
7607   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
7608 }
7609 </initex>
7610 <*package>
7611 \tl_gset_eq:NN \g_file_current_name_tl \@currname
7612 </package>

```

\g_file_stack_seq The input list of files is stored as a sequence stack.

```

7613 \seq_new:N \g_file_stack_seq

```

\g_file_record_seq The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list.

```

7614 \seq_new:N \g_file_record_seq

```

The current file name should be included in the file list!

```

7615 <*initex>
7616 \tex_everyjob:D \exp_after:wN
7617 {
7618   \tex_the:D \tex_everyjob:D
7619   \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
7620 }
7621 </initex>

```

\l_file_name_tl Used to return the fully-qualified name of a file.

```

7622 \tl_new:N \l_file_name_tl

```

\l_file_search_path_seq The current search path.

```

7623 \seq_new:N \l_file_search_path_seq

```

\l_file_search_path_saved_seq The current search path has to be saved for package use.

```

7624 <*package>
7625 \seq_new:N \l_file_search_path_saved_seq
7626 </package>

```

\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
\g_file_test_stream report end-of-file. For files which are in the current directory, this is straight-forward.
\file_add_path_search:nN 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.

```

7627 \cs_new_protected_nopar:Npn \file_add_path:nN #1#2
7628 {
7629   \ior_open:Nn \g_file_test_stream {#1}
7630   \ior_if_eof:NTF \g_file_test_stream
7631   { \file_add_path_search:nN {#1} #2 }
7632   {
7633     \ior_close:N \g_file_test_stream
7634     \tl_set:Nx #2 {#1}
7635   }
7636 }
7637 \cs_new_protected_nopar:Npn \file_add_path_search:nN #1#2
7638 {
7639   \tl_clear:N #2
7640   <*package>
7641   \cs_if_exist:NT \input@path
7642   {
7643     \seq_set_eq:NN \l_file_search_path_saved_seq \l_file_search_path_seq
7644     \clist_map_inline:Nn \input@path
7645     { \seq_put_right:Nn \l_file_search_path_seq {##1} }
7646   }
7647   </package>
7648   \seq_map_inline:Nn \l_file_search_path_seq
7649   {
7650     \ior_open:Nn \g_file_test_stream { ##1 #1 }
7651     \ior_if_eof:NF \g_file_test_stream
7652     {
7653       \tl_set:Nx #2 { ##1 #1 }
7654       \seq_map_break:
7655     }
7656   }
7657   <*package>
7658   \cs_if_exist:NT \input@path
7659   { \seq_set_eq:NN \l_file_search_path_seq \l_file_search_path_saved_seq }
7660   </package>
7661   \ior_close:N \g_file_test_stream
7662 }

```

(End definition for \file_add_path:nN. This function is documented on page ??.)

\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.

```

7663 \prg_new_protected_conditional:Nnn \file_if_exist:n { T , F , TF }
7664 {
7665   \file_add_path:nN {#1} \l_file_name_tl
7666   \tl_if_empty:NTF \l_file_name_tl

```

```

7667     { \prg_return_false: }
7668     { \prg_return_true: }
7669   }

```

(End definition for `\file_if_exist:n`. This function is documented on page 177.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does.

```

7670 \cs_new_protected_nopar:Npn \file_input:n #1
7671 {
7672   \file_add_path:nN {#1} \l_file_name_tl
7673   \tl_if_empty:NF \l_file_name_tl
7674   {
7675     <*initex>
7676     \seq_gput_right:Nx \g_file_record_seq {#1}
7677     </initex>
7678     <*package>
7679     \@addtofilelist {#1}
7680     </package>
7681     \seq_gpush:NV \g_file_stack_seq \g_file_current_name_tl
7682     \tl_gset:Nn \g_file_current_name_tl {#1}
7683     \exp_after:wN \tex_input:D \l_file_name_tl ~
7684     \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
7685   }
7686 }

```

(End definition for `\file_input:n`. This function is documented on page 178.)

`\file_path_include:n` Wrapper functions to manage the search path.
`\file_path_remove:n`

```

7687 \cs_new_protected_nopar:Npn \file_path_include:n #1
7688 {
7689   \seq_if_in:NnF \l_file_search_path_seq {#1}
7690   { \seq_put_right:Nn \l_file_search_path_seq {#1} }
7691 }
7692 \cs_new_protected_nopar:Npn \file_path_remove:n #1
7693 { \seq_remove_all:Nn \l_file_search_path_seq {#1} }

```

(End definition for `\file_path_include:n`. This function is documented on page 178.)

`\file_list:` A function to list all files used to the log.

```

7694 \cs_new_protected_nopar:Npn \file_list:
7695 {
7696   \seq_remove_duplicates:N \g_file_record_seq
7697   \iow_log:n { *-File-List-* }
7698   \seq_map_inline:Nn \g_file_record_seq { \iow_log:n {##1} }
7699   \iow_log:n { ***** }
7700 }

```

(End definition for `\file_list`:. This function is documented on page 178.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```

7701 <*package>
7702 \AtBeginDocument
7703 {
7704   \clist_map_inline:Nn \@filelist
7705     { \seq_put_right:Nn \g_file_record_seq {#1} }
7706 }
7707 </package>

7708 </initex | package>

```

183 l3fp Implementation

The following test files are used for this code: `m3fp003.lvt`.

```

7709 <*initex | package>

7710 <*package>
7711 \ProvidesExplPackage
7712   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
7713 \package_check_loaded_expl:
7714 </package>

```

183.1 Constants

```

\c_forty_four
\c_one_million
\c_one_hundred_million
\c_five_hundred_million
\c_one_thousand_million

```

There is some speed to gain by moving numbers into fixed positions.

```

7715 \int_const:Nn \c_forty_four { 44 }
7716 \int_const:Nn \c_one_million { 1 000 000 }
7717 \int_const:Nn \c_one_hundred_million { 100 000 000 }
7718 \int_const:Nn \c_five_hundred_million { 500 000 000 }
7719 \int_const:Nn \c_one_thousand_million { 1 000 000 000 }

```

```

\c_fp_pi_by_four_decimal_int
\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

```

Parts of π for trigonometric range reduction, implemented as `int` variables for speed.

```

7720 \int_new:N \c_fp_pi_by_four_decimal_int
7721 \int_set:Nn \c_fp_pi_by_four_decimal_int { 785 398 158 }
7722 \int_new:N \c_fp_pi_by_four_extended_int
7723 \int_set:Nn \c_fp_pi_by_four_extended_int { 897 448 310 }
7724 \int_new:N \c_fp_pi_decimal_int
7725 \int_set:Nn \c_fp_pi_decimal_int { 141 592 653 }
7726 \int_new:N \c_fp_pi_extended_int
7727 \int_set:Nn \c_fp_pi_extended_int { 589 793 238 }

```

```

7728 \int_new:N \c_fp_two_pi_decimal_int
7729 \int_set:Nn \c_fp_two_pi_decimal_int { 283 185 307 }
7730 \int_new:N \c_fp_two_pi_extended_int
7731 \int_set:Nn \c_fp_two_pi_extended_int { 179 586 477 }

```

\c_e_fp The value e as a “machine number”.

```

7732 \tl_const:Nn \c_e_fp { + 2.718281828 e 0 }

```

\c_one_fp The constant value 1: used for fast comparisons.

```

7733 \tl_const:Nn \c_one_fp { + 1.000000000 e 0 }

```

\c_pi_fp The value π as a “machine number”.

```

7734 \tl_const:Nn \c_pi_fp { + 3.141592654 e 0 }

```

\c_undefined_fp A marker for undefined values.

```

7735 \tl_const:Nn \c_undefined_fp { X 0.000000000 e 0 }

```

\c_zero_fp The constant zero value.

```

7736 \tl_const:Nn \c_zero_fp { + 0.000000000 e 0 }

```

183.2 Variables

\l_fp_arg_tl A token list to store the formalised representation of the input for transcendental functions.

```

7737 \tl_new:N \l_fp_arg_tl

```

\l_fp_count_int A counter for things like the number of divisions possible.

```

7738 \int_new:N \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.

```

7739 \int_new:N \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

```

7740 \int_new:N \l_fp_exp_integer_int

```

```

7741 \int_new:N \l_fp_exp_decimal_int

```

```

7742 \int_new:N \l_fp_exp_extended_int

```

```

7743 \int_new:N \l_fp_exp_exponent_int

```

<pre> \l_fp_input_a_sign_int \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 </pre>	<p>Storage for the input: two storage areas as there are at most two inputs.</p> <pre> 7744 \int_new:N \l_fp_input_a_sign_int 7745 \int_new:N \l_fp_input_a_integer_int 7746 \int_new:N \l_fp_input_a_decimal_int 7747 \int_new:N \l_fp_input_a_exponent_int 7748 \int_new:N \l_fp_input_b_sign_int 7749 \int_new:N \l_fp_input_b_integer_int 7750 \int_new:N \l_fp_input_b_decimal_int 7751 \int_new:N \l_fp_input_b_exponent_int </pre>
--	---

<pre> \l_fp_input_a_extended_int \l_fp_input_b_extended_int </pre>	<p>For internal use, “extended” floating point numbers are needed.</p> <pre> 7752 \int_new:N \l_fp_input_a_extended_int 7753 \int_new:N \l_fp_input_b_extended_int </pre>
--	---

<pre> \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 </pre>	<p>Multiplication requires that the decimal part is split into parts so that there are no overflows.</p> <pre> 7754 \int_new:N \l_fp_mul_a_i_int 7755 \int_new:N \l_fp_mul_a_ii_int 7756 \int_new:N \l_fp_mul_a_iii_int 7757 \int_new:N \l_fp_mul_a_iv_int 7758 \int_new:N \l_fp_mul_a_v_int 7759 \int_new:N \l_fp_mul_a_vi_int 7760 \int_new:N \l_fp_mul_b_i_int 7761 \int_new:N \l_fp_mul_b_ii_int 7762 \int_new:N \l_fp_mul_b_iii_int 7763 \int_new:N \l_fp_mul_b_iv_int 7764 \int_new:N \l_fp_mul_b_v_int 7765 \int_new:N \l_fp_mul_b_vi_int </pre>
--	---

<pre> \l_fp_mul_output_int \l_fp_mul_output_tl </pre>	<p>Space for multiplication results.</p> <pre> 7766 \int_new:N \l_fp_mul_output_int 7767 \tl_new:N \l_fp_mul_output_tl </pre>
---	---

<pre> \l_fp_output_sign_int \l_fp_output_integer_int \l_fp_output_decimal_int \l_fp_output_exponent_int </pre>	<p>Output is stored in the same way as input.</p> <pre> 7768 \int_new:N \l_fp_output_sign_int 7769 \int_new:N \l_fp_output_integer_int 7770 \int_new:N \l_fp_output_decimal_int 7771 \int_new:N \l_fp_output_exponent_int </pre>
--	--

<pre> \l_fp_output_extended_int </pre>	<p>Again, for calculations an extended part.</p> <pre> 7772 \int_new:N \l_fp_output_extended_int </pre>
--	---

<code>\l_fp_round_carry_bool</code>	To indicate that a digit needs to be carried forward.
<i>7773 \bool_new:N \l_fp_round_carry_bool</i>	
<code>\l_fp_round_decimal_tl</code>	A temporary store when rounding, to build up the decimal part without needing to do any maths.
<i>7774 \tl_new:N \l_fp_round_decimal_tl</i>	
<code>\l_fp_round_position_int</code> <code>\l_fp_round_target_int</code>	Used to check the position for rounding.
<i>7775 \int_new:N \l_fp_round_position_int</i>	
<i>7776 \int_new:N \l_fp_round_target_int</i>	
<code>\l_fp_sign_tl</code>	There are places where the sign needs to be set up “early”, so that the registers can be re-used.
<i>7777 \tl_new:N \l_fp_sign_tl</i>	
<code>\l_fp_split_sign_int</code>	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.
<i>7778 \int_new:N \l_fp_split_sign_int</i>	
<code>\l_fp_tmp_int</code>	A scratch int: used only where the value is not carried forward.
<i>7779 \int_new:N \l_fp_tmp_int</i>	
<code>\l_fp_tmp_tl</code>	A scratch token list variable for expanding material.
<i>7780 \tl_new:N \l_fp_tmp_tl</i>	
<code>\l_fp_trig_octant_int</code>	To track which octant the trigonometric input is in.
<i>7781 \int_new:N \l_fp_trig_octant_int</i>	
<code>\l_fp_trig_sign_int</code> <code>\l_fp_trig_decimal_int</code> <code>\l_fp_trig_extended_int</code>	Used for the calculation of trigonometric values.
<i>7782 \int_new:N \l_fp_trig_sign_int</i>	
<i>7783 \int_new:N \l_fp_trig_decimal_int</i>	
<i>7784 \int_new:N \l_fp_trig_extended_int</i>	

183.3 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`

```

7785 \cs_new_protected_nopar:Npn \fp_read:N #1
7786 { \exp_after:wN \fp_read_aux:w #1 \q_stop }
7787 \cs_new_protected_nopar:Npn \fp_read_aux:w #1#2 . #3 e #4 \q_stop
7788 {
7789   \if:w #1 -
7790     \l_fp_input_a_sign_int \c_minus_one
7791   \else:
7792     \l_fp_input_a_sign_int \c_one
7793   \fi:
7794   \l_fp_input_a_integer_int #2 \scan_stop:
7795   \l_fp_input_a_decimal_int #3 \scan_stop:
7796   \l_fp_input_a_exponent_int #4 \scan_stop:
7797 }

```

(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.

```

7798 \cs_new_protected_nopar:Npn \fp_split:Nn #1#2
7799 {
7800   \tl_set:Nx \l_fp_tmp_tl {#2}
7801   \tl_set_rescan:Nno \l_fp_tmp_tl { \char_set_catcode_ignore:n { 32 } }
7802   { \l_fp_tmp_tl }
7803   \l_fp_split_sign_int \c_one
7804   \fp_split_sign:
7805   \use:c { l_fp_input_ #1 _sign_int } \l_fp_split_sign_int
7806   \exp_after:wN \fp_split_exponent:w \l_fp_tmp_tl e e \q_stop #1
7807 }
7808 \cs_new_protected_nopar:Npn \fp_split_sign:
7809 {
7810   \if_int_compare:w \pdfTeX_strcmp:D
7811   { \exp_after:wN \tl_head:w \l_fp_tmp_tl ? \q_stop } { - }
7812   = \c_zero
7813   \tl_set:Nx \l_fp_tmp_tl
7814   {
7815     \exp_after:wN
7816     \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop
7817   }

```

```

7818     \l_fp_split_sign_int -\l_fp_split_sign_int
7819     \exp_after:wN \fp_split_sign:
7820 \else:
7821     \if_int_compare:w \pdfTeX_strcmp:D
7822     { \exp_after:wN \tl_head:w \l_fp_tmp_tl ? \q_stop } { + }
7823     = \c_zero
7824     \tl_set:Nx \l_fp_tmp_tl
7825     {
7826         \exp_after:wN
7827         \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop
7828     }
7829     \exp_after:wN \exp_after:wN \exp_after:wN \fp_split_sign:
7830 \fi:
7831 \fi:
7832 }
7833 \cs_new_protected_nopar:Npn \fp_split_exponent:w #1 e #2 e #3 \q_stop #4
7834 {
7835     \use:c { l_fp_input_ #4 _exponent_int }
7836     \int_eval:w 0 #2 \scan_stop:
7837     \tex_afterassignment:D \fp_split_aux_i:w
7838     \use:c { l_fp_input_ #4 _integer_int }
7839     \int_eval:w 0 #1 . . \q_stop #4
7840 }
7841 \cs_new_protected_nopar:Npn \fp_split_aux_i:w #1 . #2 . #3 \q_stop
7842 { \fp_split_aux_ii:w #2 000000000 \q_stop }
7843 \cs_new_protected_nopar:Npn \fp_split_aux_ii:w #1#2#3#4#5#6#7#8#9
7844 { \fp_split_aux_iii:w {#1#2#3#4#5#6#7#8#9} }
7845 \cs_new_protected_nopar:Npn \fp_split_aux_iii:w #1#2 \q_stop
7846 {
7847     \l_fp_tmp_int 1 #1 \scan_stop:
7848     \exp_after:wN \fp_split_decimal:w
7849     \int_use:N \l_fp_tmp_int 000000000 \q_stop
7850 }
7851 \cs_new_protected_nopar:Npn \fp_split_decimal:w #1#2#3#4#5#6#7#8#9
7852 { \fp_split_decimal_aux:w {#2#3#4#5#6#7#8#9} }
7853 \cs_new_protected_nopar:Npn \fp_split_decimal_aux:w #1#2#3 \q_stop #4
7854 {
7855     \use:c { l_fp_input_ #4 _decimal_int } #1#2 \scan_stop:
7856     \if_int_compare:w
7857     \int_eval:w
7858     \use:c { l_fp_input_ #4 _integer_int } +
7859     \use:c { l_fp_input_ #4 _decimal_int }
7860     \scan_stop:
7861     = \c_zero
7862     \use:c { l_fp_input_ #4 _sign_int } \c_one
7863 \fi:
7864 \if_int_compare:w
7865     \use:c { l_fp_input_ #4 _integer_int } < \c_one_thousand_million
7866 \else:
7867     \exp_after:wN \fp_overflow_msg:

```

```

7868     \fi:
7869   }

```

(End definition for `\fp_split:Nn`. This function is documented on page ??.)

```

\fp_standardise:NNNN
\fp_standardise_aux:NNNN
\fp_standardise_aux:
\fp_standardise_aux:w

```

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.

```

7870 \cs_new_protected_nopar:Npn \fp_standardise:NNNN #1#2#3#4
7871 {
7872   \if_int_compare:w
7873     \int_eval:w #2 + #3 = \c_zero
7874     #1 \c_one
7875     #4 \c_zero
7876     \exp_after:wN \use_none:nnnn
7877   \else:
7878     \exp_after:wN \fp_standardise_aux:NNNN
7879   \fi:
7880   #1#2#3#4
7881 }
7882 \cs_new_protected_nopar:Npn \fp_standardise_aux:NNNN #1#2#3#4
7883 {
7884   \cs_set_protected_nopar:Npn \fp_standardise_aux:
7885   {
7886     \if_int_compare:w #2 = \c_zero
7887     \tex_advance:D #3 \c_one_thousand_million
7888     \exp_after:wN \fp_standardise_aux:w
7889     \int_use:N #3 \q_stop
7890     \exp_after:wN \fp_standardise_aux:
7891     \fi:
7892   }
7893   \cs_set_protected_nopar:Npn
7894     \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9 \q_stop
7895   {
7896     #2 ##2 \scan_stop:
7897     #3 ##3##4##5##6##7##8##9 0 \scan_stop:
7898     \tex_advance:D #4 \c_minus_one
7899   }
7900   \fp_standardise_aux:
7901   \cs_set_protected_nopar:Npn \fp_standardise_aux:
7902   {
7903     \if_int_compare:w #2 > \c_nine
7904     \tex_advance:D #2 \c_one_thousand_million
7905     \exp_after:wN \use_i:nn \exp_after:wN
7906     \fp_standardise_aux:w \int_use:N #2
7907     \exp_after:wN \fp_standardise_aux:
7908     \fi:
7909   }
7910   \cs_set_protected_nopar:Npn
7911     \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9

```

```

7912 {
7913   #2 ##1##2##3##4##5##6##7##8 \scan_stop:
7914   \tex_advance:D #3 \c_one_thousand_million
7915   \tex_divide:D #3 \c_ten
7916   \tl_set:Nx \l_fp_tmp_tl
7917   {
7918     ##9
7919     \exp_after:wN \use_none:n \int_use:N #3
7920   }
7921   #3 \l_fp_tmp_tl \scan_stop:
7922   \tex_advance:D #4 \c_one
7923 }
7924 \fp_standardise_aux:
7925 \if_int_compare:w #4 < \c_one_hundred
7926 \if_int_compare:w #4 > -\c_one_hundred
7927 \else:
7928   #1 \c_one
7929   #2 \c_zero
7930   #3 \c_zero
7931   #4 \c_zero
7932 \fi:
7933 \else:
7934   \exp_after:wN \fp_overflow_msg:
7935 \fi:
7936 }
7937 \cs_new_protected_nopar:Npn \fp_standardise_aux: { }
7938 \cs_new_protected_nopar:Npn \fp_standardise_aux:w { }

```

(End definition for `\fp_standardise:NNNN`. This function is documented on page ??.)

183.4 Internal utilities

```

\fp_level_input_exponents:
\fp_level_input_exponents_a:
  \fp_level_input_exponents_a:NNNNNNNNN
\fp_level_input_exponents_b:
  \fp_level_input_exponents_b:NNNNNNNNN

```

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.

```

7939 \cs_new_protected_nopar:Npn \fp_level_input_exponents:
7940 {
7941   \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7942   \exp_after:wN \fp_level_input_exponents_a:
7943   \else:
7944     \exp_after:wN \fp_level_input_exponents_b:
7945   \fi:
7946 }
7947 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a:
7948 {
7949   \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7950   \tex_advance:D \l_fp_input_b_integer_int \c_one_thousand_million
7951   \exp_after:wN \use_i:nn \exp_after:wN
7952     \fp_level_input_exponents_a:NNNNNNNNN

```

```

7953         \int_use:N \l_fp_input_b_integer_int
7954         \exp_after:wN \fp_level_input_exponents_a:
7955         \fi:
7956     }
7957 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a:NNNNNNNNN
7958 #1#2#3#4#5#6#7#8#9
7959 {
7960     \l_fp_input_b_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
7961     \tex_advance:D \l_fp_input_b_decimal_int \c_one_thousand_million
7962     \tex_divide:D \l_fp_input_b_decimal_int \c_ten
7963     \tl_set:Nx \l_fp_tmp_tl
7964     {
7965         #9
7966         \exp_after:wN \use_none:n
7967         \int_use:N \l_fp_input_b_decimal_int
7968     }
7969     \l_fp_input_b_decimal_int \l_fp_tmp_tl \scan_stop:
7970     \tex_advance:D \l_fp_input_b_exponent_int \c_one
7971 }
7972 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b:
7973 {
7974     \if_int_compare:w \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
7975     \tex_advance:D \l_fp_input_a_integer_int \c_one_thousand_million
7976     \exp_after:wN \use_i:nn \exp_after:wN
7977     \fp_level_input_exponents_b:NNNNNNNNN
7978     \int_use:N \l_fp_input_a_integer_int
7979     \exp_after:wN \fp_level_input_exponents_b:
7980     \fi:
7981 }
7982 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b:NNNNNNNNN
7983 #1#2#3#4#5#6#7#8#9
7984 {
7985     \l_fp_input_a_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
7986     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7987     \tex_divide:D \l_fp_input_a_decimal_int \c_ten
7988     \tl_set:Nx \l_fp_tmp_tl
7989     {
7990         #9
7991         \exp_after:wN \use_none:n
7992         \int_use:N \l_fp_input_a_decimal_int
7993     }
7994     \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
7995     \tex_advance:D \l_fp_input_a_exponent_int \c_one
7996 }

```

(End definition for `\fp_level_input_exponents:`. This function is documented on page ??.)

`\fp_tmp:w` Used for output of results, cutting down on `\exp_after:wN`. This is just a place holder definition.

```
7997 \cs_new_protected_nopar:Npn \fp_tmp:w #1#2 { }
```

(End definition for \fp_tmp:w.)

183.5 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`

```
7998 \cs_new_protected_nopar:Npn \fp_new:N #1
7999 {
8000   \tl_new:N #1
8001   \tl_gset_eq:NN #1 \c_zero_fp
8002 }
8003 \cs_generate_variant:Nn \fp_new:N { c }
```

(End definition for \fp_new:N and \fp_new:c. These functions are documented on page 180.)

`\fp_const:Nn` A simple wrapper.

`\fp_const:cn`

```
8004 \cs_new_protected_nopar:Npn \fp_const:Nn #1#2
8005 {
8006   \fp_new:N #1
8007   \fp_gset:Nn #1 {#2}
8008 }
8009 \cs_generate_variant:Nn \fp_const:Nn { c }
```

(End definition for \fp_const:Nn and \fp_const:cn. These functions are documented on page 180.)

`\fp_zero:N` Zeroing fixed-points is pretty obvious.

`\fp_zero:c`

`\fp_gzero:N`

`\fp_gzero:c`

```
8010 \cs_new_protected_nopar:Npn \fp_zero:N #1
8011 { \tl_set_eq:NN #1 \c_zero_fp }
8012 \cs_new_protected_nopar:Npn \fp_gzero:N #1
8013 { \tl_gset_eq:NN #1 \c_zero_fp }
8014 \cs_generate_variant:Nn \fp_zero:N { c }
8015 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for \fp_zero:N and \fp_zero:c. These functions are documented on page 180.)

`\fp_set:Nn` To trap any input errors, a very simple version of the parser is run here. This will pick up any invalid characters at this stage, saving issues later. The splitting approach is the same as the more advanced function later.

`\fp_set:cn`

`\fp_gset:Nn`

`\fp_gset:cn`

`\fp_set_aux:Nnn`

```

8016 \cs_new_protected_nopar:Npn \fp_set:Nn { \fp_set_aux:NNn \tl_set:Nn }
8017 \cs_new_protected_nopar:Npn \fp_gset:Nn { \fp_set_aux:NNn \tl_gset:Nn }
8018 \cs_new_protected_nopar:Npn \fp_set_aux:NNn #1#2#3
8019 {
8020   \group_begin:
8021   \fp_split:Nn a {#3}
8022   \fp_standardise:NNNN
8023   \l_fp_input_a_sign_int
8024   \l_fp_input_a_integer_int
8025   \l_fp_input_a_decimal_int
8026   \l_fp_input_a_exponent_int
8027   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8028   \cs_set_protected_nopar:Npx \fp_tmp:w
8029   {
8030     \group_end:
8031     #1 \exp_not:N #2
8032     {
8033       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8034       -
8035       \else:
8036       +
8037       \fi:
8038       \int_use:N \l_fp_input_a_integer_int
8039       .
8040       \exp_after:wN \use_none:n
8041       \int_use:N \l_fp_input_a_decimal_int
8042       e
8043       \int_use:N \l_fp_input_a_exponent_int
8044     }
8045   }
8046   \fp_tmp:w
8047 }
8048 \cs_generate_variant:Nn \fp_set:Nn { c }
8049 \cs_generate_variant:Nn \fp_gset:Nn { c }

```

(End definition for `\fp_set:Nn` and `\fp_set:cn`. These functions are documented on page 181.)

`\fp_set_from_dim:Nn` Here, dimensions are converted to fixed-points *via* a temporary variable. This ensures that they always convert as points. The code is then essentially the same as for `\fp_set:Nn`, but with the dimension passed so that it will be stripped of the `pt` on the way 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
  \l_fp_tmp_dim
  \l_fp_tmp_skip
8050 \cs_new_protected_nopar:Npn \fp_set_from_dim:Nn
8051 { \fp_set_from_dim_aux:NNn \tl_set:Nx }
8052 \cs_new_protected_nopar:Npn \fp_gset_from_dim:Nn
8053 { \fp_set_from_dim_aux:NNn \tl_gset:Nx }
8054 \cs_new_protected_nopar:Npn \fp_set_from_dim_aux:NNn #1#2#3
8055 {
8056   \group_begin:
8057   \l_fp_tmp_skip \etex_glueexpr:D #3 \scan_stop:

```



```

8058 \l_fp_tmp_dim \l_fp_tmp_skip
8059 \fp_split:Nn a
8060 {
8061   \exp_after:wN \fp_set_from_dim_aux:w
8062   \dim_use:N \l_fp_tmp_dim
8063 }
8064 \fp_standardise:NNNN
8065 \l_fp_input_a_sign_int
8066 \l_fp_input_a_integer_int
8067 \l_fp_input_a_decimal_int
8068 \l_fp_input_a_exponent_int
8069 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8070 \cs_set_protected_nopar:Npx \fp_tmp:w
8071 {
8072   \group_end:
8073   #1 \exp_not:N #2
8074   {
8075     \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8076     -
8077     \else:
8078     +
8079     \fi:
8080     \int_use:N \l_fp_input_a_integer_int
8081     .
8082     \exp_after:wN \use_none:n
8083     \int_use:N \l_fp_input_a_decimal_int
8084     e
8085     \int_use:N \l_fp_input_a_exponent_int
8086   }
8087 }
8088 \fp_tmp:w
8089 }
8090 \cs_set_protected_nopar:Npx \fp_set_from_dim_aux:w
8091 {
8092   \cs_set_nopar:Npn \exp_not:N \fp_set_from_dim_aux:w
8093   ##1 \tl_to_str:n { pt } {##1}
8094 }
8095 \fp_set_from_dim_aux:w
8096 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
8097 \cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
8098 \dim_new:N \l_fp_tmp_dim
8099 \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 ??.)

`\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`

```

8100 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
8101 \cs_new_eq:NN \fp_set_eq:cN \tl_set_eq:cN

```

```

8102 \cs_new_eq:NN \fp_set_eq:Nc \tl_set_eq:Nc
8103 \cs_new_eq:NN \fp_set_eq:cc \tl_set_eq:cc
8104 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
8105 \cs_new_eq:NN \fp_gset_eq:cN \tl_gset_eq:cN
8106 \cs_new_eq:NN \fp_gset_eq:Nc \tl_gset_eq:Nc
8107 \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 180.)

\fp_show:N Simple showing of the underlying variable.
\fp_show:c

```

8108 \cs_new_eq:NN \fp_show:N \tl_show:N
8109 \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 181.)

\fp_use:N The idea of the `\fp_use:N` function to convert the stored value into something suitable
\fp_use:c for T_EX 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
8110 \cs_new_nopar:Npn \fp_use:N #1
8111 { \exp_after:wN \fp_use_aux:w #1 \q_stop }
8112 \cs_generate_variant:Nn \fp_use:N { c }
8113 \cs_new_nopar:Npn \fp_use:c #1#2 e #3 \q_stop
8114 {
8115   \if:w #1 -
8116   -
8117   \fi:
8118   \if_int_compare:w #3 > \c_zero
8119   \exp_after:wN \fp_use_large:w
8120   \else:
8121   \if_int_compare:w #3 < \c_zero
8122   \exp_after:wN \exp_after:wN \exp_after:wN
8123   \fp_use_small:w
8124   \else:
8125   \exp_after:wN \exp_after:wN \exp_after:wN \fp_use_none:w
8126   \fi:
8127   \fi:
8128   #2 e #3 \q_stop
8129 }

```

When the exponent is zero, the input is simply returned as output.

```

8130 \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.

```

8131 \cs_new_nopar:Npn \fp_use_small:w #1 . #2 e #3 \q_stop
8132 {

```

```

8133 0 .
8134 \prg_replicate:nn { -#3 - 1 } { 0 }
8135 #1#2
8136 }

```

Life is more complex for large numbers. The decimal point needs to be shuffled, with potentially some zero-filling for very large values.

```

8137 \cs_new_nopar:Npn \fp_use_large:w #1 . #2 e #3 \q_stop
8138 {
8139   \if_int_compare:w #3 < \c_ten
8140     \exp_after:wN \fp_use_large_aux_i:w
8141   \else:
8142     \exp_after:wN \fp_use_large_aux_ii:w
8143   \fi:
8144   #1#2 e #3 \q_stop
8145 }
8146 \cs_new_nopar:Npn \fp_use_large_aux_i:w #1#2 e #3 \q_stop
8147 {
8148   #1
8149   \use:c { fp_use_large_aux_ #3 :w } #2 \q_stop
8150 }
8151 \cs_new_nopar:cpn { fp_use_large_aux_1:w } #1#2 \q_stop { #1 . #2 }
8152 \cs_new_nopar:cpn { fp_use_large_aux_2:w } #1#2#3 \q_stop
8153 { #1#2 . #3 }
8154 \cs_new_nopar:cpn { fp_use_large_aux_3:w } #1#2#3#4 \q_stop
8155 { #1#2#3 . #4 }
8156 \cs_new_nopar:cpn { fp_use_large_aux_4:w } #1#2#3#4#5 \q_stop
8157 { #1#2#3#4 . #5 }
8158 \cs_new_nopar:cpn { fp_use_large_aux_5:w } #1#2#3#4#5#6 \q_stop
8159 { #1#2#3#4#5 . #6 }
8160 \cs_new_nopar:cpn { fp_use_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop
8161 { #1#2#3#4#5#6 . #7 }
8162 \cs_new_nopar:cpn { fp_use_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop
8163 { #1#2#3#4#6#7 . #8 }
8164 \cs_new_nopar:cpn { fp_use_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop
8165 { #1#2#3#4#5#6#7#8 . #9 }
8166 \cs_new_nopar:cpn { fp_use_large_aux_9:w } #1 \q_stop { #1 . }
8167 \cs_new_nopar:Npn \fp_use_large_aux_ii:w #1 e #2 \q_stop
8168 {
8169   #1
8170   \prg_replicate:nn { #2 - 9 } { 0 }
8171   .
8172 }

```

(End definition for `\fp_use:N` and `\fp_use:c`. These functions are documented on page 181.)

183.6 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`

```
8173 \cs_new_nopar:Npn \fp_to_dim:N #1 { \fp_use:N #1 pt }
8174 \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 182.)

`\fp_to_int:N`

`\fp_to_int:c`

Converting to integers in an expandable manner is very similar to simply using floating 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
```

```
8175 \cs_new_nopar:Npn \fp_to_int:N #1
8176 { \exp_after:wN \fp_to_int_aux:w #1 \q_stop }
8177 \cs_generate_variant:Nn \fp_to_int:N { c }
8178 \cs_new_nopar:Npn \fp_to_int_aux:w #1#2 e #3 \q_stop
8179 {
8180   \if:w #1 -
8181     -
8182   \fi:
8183   \if_int_compare:w #3 < \c_zero
8184     \exp_after:wN \fp_to_int_small:w
8185   \else:
8186     \exp_after:wN \fp_to_int_large:w
8187   \fi:
8188   #2 e #3 \q_stop
8189 }
```

For small numbers, if the decimal part is greater than a half then there is rounding up to do.

```
8190 \cs_new_nopar:Npn \fp_to_int_small:w #1 . #2 e #3 \q_stop
8191 {
8192   \if_int_compare:w #3 > \c_one
8193   \else:
8194     \if_int_compare:w #1 < \c_five
8195       0
8196     \else:
8197       1
8198     \fi:
8199   \fi:
8200 }
```

For large numbers, the idea is to split off the part for rounding, do the rounding and fill if needed.

```
8201 \cs_new_nopar:Npn \fp_to_int_large:w #1 . #2 e #3 \q_stop
```

```

8202 {
8203   \if_int_compare:w #3 < \c_ten
8204   \exp_after:wN \fp_to_int_large_aux_i:w
8205   \else:
8206     \exp_after:wN \fp_to_int_large_aux_ii:w
8207     \fi:
8208     #1#2 e #3 \q_stop
8209   }
8210   \cs_new_nopar:Npn \fp_to_int_large_aux_i:w #1#2 e #3 \q_stop
8211   { \use:c { fp_to_int_large_aux_ #3 :w } #2 \q_stop {#1} }
8212   \cs_new_nopar:cpn { fp_to_int_large_aux_1:w } #1#2 \q_stop
8213   { \fp_to_int_large_aux:nnn { #2 0 } {#1} }
8214   \cs_new_nopar:cpn { fp_to_int_large_aux_2:w } #1#2#3 \q_stop
8215   { \fp_to_int_large_aux:nnn { #3 00 } {#1#2} }
8216   \cs_new_nopar:cpn { fp_to_int_large_aux_3:w } #1#2#3#4 \q_stop
8217   { \fp_to_int_large_aux:nnn { #4 000 } {#1#2#3} }
8218   \cs_new_nopar:cpn { fp_to_int_large_aux_4:w } #1#2#3#4#5 \q_stop
8219   { \fp_to_int_large_aux:nnn { #5 0000 } {#1#2#3#4} }
8220   \cs_new_nopar:cpn { fp_to_int_large_aux_5:w } #1#2#3#4#5#6 \q_stop
8221   { \fp_to_int_large_aux:nnn { #6 00000 } {#1#2#3#4#5} }
8222   \cs_new_nopar:cpn { fp_to_int_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop
8223   { \fp_to_int_large_aux:nnn { #7 000000 } {#1#2#3#4#5#6} }
8224   \cs_new_nopar:cpn { fp_to_int_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop
8225   { \fp_to_int_large_aux:nnn { #8 0000000 } {#1#2#3#4#5#6#7} }
8226   \cs_new_nopar:cpn { fp_to_int_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop
8227   { \fp_to_int_large_aux:nnn { #9 00000000 } {#1#2#3#4#5#6#7#8} }
8228   \cs_new_nopar:cpn { fp_to_int_large_aux_9:w } #1 \q_stop {#1}
8229   \cs_new_nopar:Npn \fp_to_int_large_aux:nnn #1#2#3
8230   {
8231     \if_int_compare:w #1 < \c_five_hundred_million
8232     #3#2
8233     \else:
8234       \int_value:w \int_eval:w #3#2 + 1 \int_eval_end:
8235     \fi:
8236   }
8237   \cs_new_nopar:Npn \fp_to_int_large_aux_ii:w #1 e #2 \q_stop
8238   {
8239     #1
8240     \prg_replicate:nn { #2 - 9 } { 0 }
8241   }

```

(End definition for \fp_to_int:N and \fp_to_int:c. These functions are documented on page 182.)

\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

```

```

8242 \cs_new_nopar:Npn \fp_to_tl:N #1
8243 { \exp_after:wN \fp_to_tl_aux:w #1 \q_stop }
8244 \cs_generate_variant:Nn \fp_to_tl:N { c }
8245 \cs_new_nopar:Npn \fp_to_tl_aux:w #1#2 e #3 \q_stop

```

```

8246 {
8247   \if:w #1 -
8248   -
8249   \fi:
8250   \if_int_compare:w #3 < \c_zero
8251     \exp_after:wN \fp_to_tl_small:w
8252   \else:
8253     \exp_after:wN \fp_to_tl_large:w
8254   \fi:
8255   #2 e #3 \q_stop
8256 }

```

For “large” numbers (exponent ≥ 0) there are two cases. For very large exponents (≥ 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.

```

8257 \cs_new_nopar:Npn \fp_to_tl_large:w #1 e #2 \q_stop
8258 {
8259   \if_int_compare:w #2 < \c_ten
8260     \exp_after:wN \fp_to_tl_large_aux_i:w
8261   \else:
8262     \exp_after:wN \fp_to_tl_large_aux_ii:w
8263   \fi:
8264   #1 e #2 \q_stop
8265 }
8266 \cs_new_nopar:Npn \fp_to_tl_large_aux_i:w #1 e #2 \q_stop
8267 { \use:c { fp_to_tl_large_ #2 :w } #1 \q_stop }
8268 \cs_new_nopar:Npn \fp_to_tl_large_aux_ii:w #1 . #2 e #3 \q_stop
8269 {
8270   #1
8271   \fp_to_tl_large_zeros:NNNNNNNN #2
8272   e #3
8273 }
8274 \cs_new_nopar:cpn { fp_to_tl_large_0:w } #1 . #2 \q_stop
8275 {
8276   #1
8277   \fp_to_tl_large_zeros:NNNNNNNN #2
8278 }
8279 \cs_new_nopar:cpn { fp_to_tl_large_1:w } #1 . #2#3 \q_stop
8280 {
8281   #1#2
8282   \fp_to_tl_large_zeros:NNNNNNNN #3 0
8283 }
8284 \cs_new_nopar:cpn { fp_to_tl_large_2:w } #1 . #2#3#4 \q_stop
8285 {
8286   #1#2#3
8287   \fp_to_tl_large_zeros:NNNNNNNN #4 00
8288 }
8289 \cs_new_nopar:cpn { fp_to_tl_large_3:w } #1 . #2#3#4#5 \q_stop
8290 {

```

```

8291     #1#2#3#4
8292     \fp_to_tl_large_zeros:NNNNNNNN #5 000
8293 }
8294 \cs_new_nopar:cpn { fp_to_tl_large_4:w } #1 . #2#3#4#5#6 \q_stop
8295 {
8296     #1#2#3#4#5
8297     \fp_to_tl_large_zeros:NNNNNNNN #6 0000
8298 }
8299 \cs_new_nopar:cpn { fp_to_tl_large_5:w } #1 . #2#3#4#5#6#7 \q_stop
8300 {
8301     #1#2#3#4#5#6
8302     \fp_to_tl_large_zeros:NNNNNNNN #7 00000
8303 }
8304 \cs_new_nopar:cpn { fp_to_tl_large_6:w } #1 . #2#3#4#5#6#7#8 \q_stop
8305 {
8306     #1#2#3#4#5#6#7
8307     \fp_to_tl_large_zeros:NNNNNNNN #8 000000
8308 }
8309 \cs_new_nopar:cpn { fp_to_tl_large_7:w } #1 . #2#3#4#5#6#7#8#9 \q_stop
8310 {
8311     #1#2#3#4#5#6#7#8
8312     \fp_to_tl_large_zeros:NNNNNNNN #9 0000000
8313 }
8314 \cs_new_nopar:cpn { fp_to_tl_large_8:w } #1 .
8315 {
8316     #1
8317     \use:c { fp_to_tl_large_8_aux:w }
8318 }
8319 \cs_new_nopar:cpn { fp_to_tl_large_8_aux:w } #1#2#3#4#5#6#7#8#9 \q_stop
8320 {
8321     #1#2#3#4#5#6#7#8
8322     \fp_to_tl_large_zeros:NNNNNNNN #9 00000000
8323 }
8324 \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.

```

8325 \cs_new_nopar:Npn \fp_to_tl_small:w #1 e #2 \q_stop
8326 {
8327     \if_int_compare:w #2 = \c_minus_one
8328     \exp_after:wN \fp_to_tl_small_one:w
8329     \else:
8330     \if_int_compare:w #2 = -\c_two
8331     \exp_after:wN \exp_after:wN \exp_after:wN \fp_to_tl_small_two:w
8332     \else:
8333     \exp_after:wN \exp_after:wN \exp_after:wN \fp_to_tl_small_aux:w
8334     \fi:
8335     \fi:

```

```

8336     #1 e #2 \q_stop
8337 }
8338 \cs_new_nopar:Npn \fp_to_tl_small_one:w #1 . #2 e #3 \q_stop
8339 {
8340   \if_int_compare:w \fp_use_ix:NNNNNNNN #2 > \c_four
8341     \if_int_compare:w
8342       \int_eval:w #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
8343       < \c_one_thousand_million
8344       0.
8345       \exp_after:wN \fp_to_tl_small_zeros:NNNNNNNN
8346       \int_value:w \int_eval:w
8347       #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
8348       \int_eval_end:
8349     \else:
8350       1
8351     \fi:
8352   \else:
8353     0. #1
8354     \fp_to_tl_small_zeros:NNNNNNNN #2
8355   \fi:
8356 }
8357 \cs_new_nopar:Npn \fp_to_tl_small_two:w #1 . #2 e #3 \q_stop
8358 {
8359   \if_int_compare:w \fp_use_iix_ix:NNNNNNNN #2 > \c_forty_four
8360     \if_int_compare:w
8361       \int_eval:w #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
8362       < \c_one_thousand_million
8363       0.0
8364       \exp_after:wN \fp_to_tl_small_zeros:NNNNNNNN
8365       \int_value:w \int_eval:w
8366       #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
8367       \int_eval_end:
8368     \else:
8369       0.1
8370     \fi:
8371   \else:
8372     0.0
8373     #1
8374     \fp_to_tl_small_zeros:NNNNNNNN #2
8375   \fi:
8376 }
8377 \cs_new_nopar:Npn \fp_to_tl_small_aux:w #1 . #2 e #3 \q_stop
8378 {
8379   #1
8380   \fp_to_tl_large_zeros:NNNNNNNN #2
8381   e #3
8382 }

```

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.

```

8383 \cs_new_nopar:Npn \fp_to_tl_large_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9
8384 {
8385   \if_int_compare:w #9 = \c_zero
8386     \if_int_compare:w #8 = \c_zero
8387       \if_int_compare:w #7 = \c_zero
8388         \if_int_compare:w #6 = \c_zero
8389           \if_int_compare:w #5 = \c_zero
8390             \if_int_compare:w #4 = \c_zero
8391               \if_int_compare:w #3 = \c_zero
8392                 \if_int_compare:w #2 = \c_zero
8393                   \if_int_compare:w #1 = \c_zero
8394                     \else:
8395                       . #1
8396                     \fi:
8397                   \else:
8398                     . #1#2
8399                   \fi:
8400                 \else:
8401                   . #1#2#3
8402                 \fi:
8403               \else:
8404                 . #1#2#3#4
8405               \fi:
8406             \else:
8407               . #1#2#3#4#5
8408             \fi:
8409           \else:
8410             . #1#2#3#4#5#6
8411           \fi:
8412         \else:
8413           . #1#2#3#4#5#6#7
8414         \fi:
8415       \else:
8416         . #1#2#3#4#5#6#7#8
8417       \fi:
8418     \else:
8419       . #1#2#3#4#5#6#7#8#9
8420     \fi:
8421   }
8422 \cs_new_nopar:Npn \fp_to_tl_small_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9
8423 {
8424   \if_int_compare:w #9 = \c_zero
8425     \if_int_compare:w #8 = \c_zero
8426       \if_int_compare:w #7 = \c_zero
8427         \if_int_compare:w #6 = \c_zero
8428           \if_int_compare:w #5 = \c_zero
8429             \if_int_compare:w #4 = \c_zero
8430               \if_int_compare:w #3 = \c_zero

```

```

8431         \if_int_compare:w #2 = \c_zero
8432         \if_int_compare:w #1 = \c_zero
8433         \else:
8434             #1
8435         \fi:
8436     \else:
8437         #1#2
8438     \fi:
8439 \else:
8440     #1#2#3
8441 \fi:
8442 \else:
8443     #1#2#3#4
8444 \fi:
8445 \else:
8446     #1#2#3#4#5
8447 \fi:
8448 \else:
8449     #1#2#3#4#5#6
8450 \fi:
8451 \else:
8452     #1#2#3#4#5#6#7
8453 \fi:
8454 \else:
8455     #1#2#3#4#5#6#7#8
8456 \fi:
8457 \else:
8458     #1#2#3#4#5#6#7#8#9
8459 \fi:
8460 }

```

Some quick “return a few” functions.

```

8461 \cs_new_nopar:Npn \fp_use_iix_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#8#9}
8462 \cs_new_nopar:Npn \fp_use_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#9}
8463 \cs_new_nopar:Npn \fp_use_i_to_vii:NNNNNNNN #1#2#3#4#5#6#7#8#9
8464 {#1#2#3#4#5#6#7}
8465 \cs_new_nopar:Npn \fp_use_i_to_iix:NNNNNNNN #1#2#3#4#5#6#7#8#9
8466 {#1#2#3#4#5#6#7#8}

```

(End definition for `\fp_to_tl:N` and `\fp_to_tl:c`. These functions are documented on page 182.)

183.7 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).

```

8467 \cs_new_protected_nopar:Npn \fp_round_figures:Nn
8468 { \fp_round_figures_aux:NNn \tl_set:Nn }
8469 \cs_generate_variant:Nn \fp_round_figures:Nn { c }
8470 \cs_new_protected_nopar:Npn \fp_ground_figures:Nn
8471 { \fp_round_figures_aux:NNn \tl_gset:Nn }
8472 \cs_generate_variant:Nn \fp_ground_figures:Nn { c }
8473 \cs_new_protected_nopar:Npn \fp_round_figures_aux:NNn #1#2#3
8474 {
8475   \group_begin:
8476   \fp_read:N #2
8477   \int_set:Nn \l_fp_round_target_int { #3 - 1 }
8478   \if_int_compare:w \l_fp_round_target_int < \c_ten
8479     \exp_after:wN \fp_round:
8480   \fi:
8481   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8482   \cs_set_protected_nopar:Npx \fp_tmp:w
8483   {
8484     \group_end:
8485     #1 \exp_not:N #2
8486     {
8487       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8488       -
8489       \else:
8490       +
8491       \fi:
8492       \int_use:N \l_fp_input_a_integer_int
8493       .
8494       \exp_after:wN \use_none:n
8495       \int_use:N \l_fp_input_a_decimal_int
8496       e
8497       \int_use:N \l_fp_input_a_exponent_int
8498     }
8499   }
8500   \fp_tmp:w
8501 }

```

(End definition for `\fp_round_figures:Nn` and `\fp_round_figures:cn`. These functions are documented on page 183.)

`\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.

```

8502 \cs_new_protected_nopar:Npn \fp_round_places:Nn
8503 { \fp_round_places_aux:NNn \tl_set:Nn }
8504 \cs_generate_variant:Nn \fp_round_places:Nn { c }
8505 \cs_new_protected_nopar:Npn \fp_ground_places:Nn
8506 { \fp_round_places_aux:NNn \tl_gset:Nn }
8507 \cs_generate_variant:Nn \fp_ground_places:Nn { c }
8508 \cs_new_protected_nopar:Npn \fp_round_places_aux:NNn #1#2#3
8509 {

```

```

8510 \group_begin:
8511 \fp_read:N #2
8512 \int_set:Nn \l_fp_round_target_int
8513 { #3 + \l_fp_input_a_exponent_int }
8514 \if_int_compare:w \l_fp_round_target_int < \c_ten
8515 \exp_after:wN \fp_round:
8516 \fi:
8517 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8518 \cs_set_protected_nopar:Npx \fp_tmp:w
8519 {
8520 \group_end:
8521 #1 \exp_not:N #2
8522 {
8523 \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8524 -
8525 \else:
8526 +
8527 \fi:
8528 \int_use:N \l_fp_input_a_integer_int
8529 .
8530 \exp_after:wN \use_none:n
8531 \int_use:N \l_fp_input_a_decimal_int
8532 e
8533 \int_use:N \l_fp_input_a_exponent_int
8534 }
8535 }
8536 \fp_tmp:w
8537 }

```

(End definition for `\fp_round_places:Nn` and `\fp_round_places:cn`. These functions are documented on page 183.)

`\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.

```

8538 \cs_new_protected_nopar:Npn \fp_round:
8539 {
8540 \bool_set_false:N \l_fp_round_carry_bool
8541 \l_fp_round_position_int \c_eight
8542 \tl_clear:N \l_fp_round_decimal_tl
8543 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8544 \exp_after:wN \use_i:nn \exp_after:wN
8545 \fp_round_aux:NNNNNNNNN \int_use:N \l_fp_input_a_decimal_int
8546 }
8547 \cs_new_protected_nopar:Npn \fp_round_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9
8548 {
8549 \fp_round_loop:N #9#8#7#6#5#4#3#2#1
8550 \bool_if:NT \l_fp_round_carry_bool

```

```

8551     { \tex_advance:D \l_fp_input_a_integer_int \c_one }
8552 \l_fp_input_a_decimal_int \l_fp_round_decimal_tl \scan_stop:
8553 \if_int_compare:w \l_fp_input_a_integer_int < \c_ten
8554 \else:
8555     \l_fp_input_a_integer_int \c_one
8556     \tex_divide:D \l_fp_input_a_decimal_int \c_ten
8557     \tex_advance:D \l_fp_input_a_exponent_int \c_one
8558 \fi:
8559 }
8560 \cs_new_protected_nopar:Npn \fp_round_loop:N #1
8561 {
8562     \if_int_compare:w \l_fp_round_position_int < \l_fp_round_target_int
8563     \bool_if:NTF \l_fp_round_carry_bool
8564     { \l_fp_tmp_int \int_eval:w #1 + \c_one \scan_stop: }
8565     { \l_fp_tmp_int \int_eval:w #1 \scan_stop: }
8566     \if_int_compare:w \l_fp_tmp_int = \c_ten
8567     \l_fp_tmp_int \c_zero
8568     \else:
8569     \bool_set_false:N \l_fp_round_carry_bool
8570     \fi:
8571     \tl_set:Nx \l_fp_round_decimal_tl
8572     { \int_use:N \l_fp_tmp_int \l_fp_round_decimal_tl }
8573 \else:
8574     \tl_set:Nx \l_fp_round_decimal_tl { 0 \l_fp_round_decimal_tl }
8575     \if_int_compare:w \l_fp_round_position_int = \l_fp_round_target_int
8576     \if_int_compare:w #1 > \c_four
8577     \bool_set_true:N \l_fp_round_carry_bool
8578     \fi:
8579     \fi:
8580 \fi:
8581 \tex_advance:D \l_fp_round_position_int \c_minus_one
8582 \if_int_compare:w \l_fp_round_position_int > \c_minus_one
8583 \exp_after:wN \fp_round_loop:N
8584 \fi:
8585 }

```

(End definition for `\fp_round:`. This function is documented on page ??.)

183.8 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`

```

8586 \cs_new_protected_nopar:Npn \fp_abs:N { \fp_abs_aux:NN \tl_set:Nn }
8587 \cs_new_protected_nopar:Npn \fp_gabs:N { \fp_abs_aux:NN \tl_gset:Nn }
8588 \cs_generate_variant:Nn \fp_abs:N { c }
8589 \cs_generate_variant:Nn \fp_gabs:N { c }
8590 \cs_new_protected_nopar:Npn \fp_abs_aux:NN #1#2
8591 {
8592     \group_begin:

```

```

8593 \fp_read:N #2
8594 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8595 \cs_set_protected_nopar:Npx \fp_tmp:w
8596 {
8597   \group_end:
8598   #1 \exp_not:N #2
8599   {
8600     +
8601     \int_use:N \l_fp_input_a_integer_int
8602     .
8603     \exp_after:wN \use_none:n
8604     \int_use:N \l_fp_input_a_decimal_int
8605     e
8606     \int_use:N \l_fp_input_a_exponent_int
8607   }
8608 }
8609 \fp_tmp:w
8610 }

```

(End definition for `\fp_abs:N` and `\fp_abs:c`. These functions are documented on page 185.)

`\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`

```

8611 \cs_new_protected_nopar:Npn \fp_neg:N { \fp_neg_aux:NN \tl_set:Nn }
8612 \cs_new_protected_nopar:Npn \fp_gneg:N { \fp_neg_aux:NN \tl_gset:Nn }
8613 \cs_generate_variant:Nn \fp_neg:N { c }
8614 \cs_generate_variant:Nn \fp_gneg:N { c }
8615 \cs_new_protected_nopar:Npn \fp_neg_aux:NN #1#2
8616 {
8617   \group_begin:
8618   \fp_read:N #2
8619   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8620   \tl_set:Nx \l_fp_tmp_tl
8621   {
8622     \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8623     +
8624     \else:
8625     -
8626     \fi:
8627     \int_use:N \l_fp_input_a_integer_int
8628     .
8629     \exp_after:wN \use_none:n
8630     \int_use:N \l_fp_input_a_decimal_int
8631     e
8632     \int_use:N \l_fp_input_a_exponent_int
8633   }
8634   \exp_after:wN \group_end: \exp_after:wN
8635   #1 \exp_after:wN #2 \exp_after:wN { \l_fp_tmp_tl }
8636 }

```

(End definition for `\fp_neg:N` and `\fp_neg:c`. These functions are documented on page 185.)

183.9 Basic arithmetic

`\fp_add:Nn`

`\fp_add:cn`

`\fp_gadd:Nn`

`\fp_gadd:cn`

`\fp_add_aux:NNn`

`\fp_add_core:`

`\fp_add_sum:`

`\fp_add_difference:`

The various addition functions are simply different ways to call the single master function below. This pattern is repeated for the other arithmetic functions.

```

8637 \cs_new_protected_nopar:Npn \fp_add:Nn { \fp_add_aux:NNn \tl_set:Nn }
8638 \cs_new_protected_nopar:Npn \fp_gadd:Nn { \fp_add_aux:NNn \tl_gset:Nn }
8639 \cs_generate_variant:Nn \fp_add:Nn { c }
8640 \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.

```

8641 \cs_new_protected_nopar:Npn \fp_add_aux:NNn #1#2#3
8642 {
8643   \group_begin:
8644     \fp_read:N #2
8645     \fp_split:Nn b {#3}
8646     \fp_standardise:NNNN
8647     \l_fp_input_b_sign_int
8648     \l_fp_input_b_integer_int
8649     \l_fp_input_b_decimal_int
8650     \l_fp_input_b_exponent_int
8651     \fp_add_core:
8652     \fp_tmp:w #1#2
8653   }
8654   \cs_new_protected_nopar:Npn \fp_add_core:
8655   {
8656     \fp_level_input_exponents:
8657     \if_int_compare:w
8658       \int_eval:w
8659         \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8660       > \c_zero
8661     \exp_after:wN \fp_add_sum:
8662   \else:
8663     \exp_after:wN \fp_add_difference:
8664   \fi:
8665   \l_fp_output_exponent_int \l_fp_input_a_exponent_int
8666   \fp_standardise:NNNN
8667   \l_fp_output_sign_int
8668   \l_fp_output_integer_int
8669   \l_fp_output_decimal_int
8670   \l_fp_output_exponent_int
8671   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8672   {
8673     \group_end:
8674     ##1 ##2
8675     {
8676       \if_int_compare:w \l_fp_output_sign_int < \c_zero

```

```

8677         -
8678         \else:
8679         +
8680         \fi:
8681         \int_use:N \l_fp_output_integer_int
8682         .
8683         \exp_after:wN \use_none:n
8684         \int_value:w \int_eval:w
8685         \l_fp_output_decimal_int + \c_one_thousand_million
8686         e
8687         \int_use:N \l_fp_output_exponent_int
8688     }
8689 }
8690 }

```

Finding the sum of two numbers is trivially easy.

```

8691 \cs_new_protected_nopar:Npn \fp_add_sum:
8692 {
8693     \l_fp_output_sign_int \l_fp_input_a_sign_int
8694     \l_fp_output_integer_int
8695     \int_eval:w
8696     \l_fp_input_a_integer_int + \l_fp_input_b_integer_int
8697     \scan_stop:
8698     \l_fp_output_decimal_int
8699     \int_eval:w
8700     \l_fp_input_a_decimal_int + \l_fp_input_b_decimal_int
8701     \scan_stop:
8702     \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
8703     \else:
8704         \tex_advance:D \l_fp_output_integer_int \c_one
8705         \tex_advance:D \l_fp_output_decimal_int -\c_one_thousand_million
8706     \fi:
8707 }

```

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.

```

8708 \cs_new_protected_nopar:Npn \fp_add_difference:
8709 {
8710     \l_fp_output_integer_int
8711     \int_eval:w
8712     \l_fp_input_a_integer_int - \l_fp_input_b_integer_int
8713     \scan_stop:
8714     \l_fp_output_decimal_int
8715     \int_eval:w
8716     \l_fp_input_a_decimal_int - \l_fp_input_b_decimal_int
8717     \scan_stop:

```



```

8718 \if_int_compare:w \l_fp_output_decimal_int < \c_zero
8719 \tex_advance:D \l_fp_output_integer_int \c_minus_one
8720 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8721 \fi:
8722 \if_int_compare:w \l_fp_output_integer_int < \c_zero
8723 \l_fp_output_sign_int \l_fp_input_b_sign_int
8724 \if_int_compare:w \l_fp_output_decimal_int = \c_zero
8725 \l_fp_output_integer_int -\l_fp_output_integer_int
8726 \else:
8727 \l_fp_output_decimal_int
8728 \int_eval:w
8729 \c_one_thousand_million - \l_fp_output_decimal_int
8730 \scan_stop:
8731 \l_fp_output_integer_int
8732 \int_eval:w
8733 - \l_fp_output_integer_int - \c_one
8734 \scan_stop:
8735 \fi:
8736 \else:
8737 \l_fp_output_sign_int \l_fp_input_a_sign_int
8738 \fi:
8739 }

```

(End definition for `\fp_add:Nn` and `\fp_add:cn`. These functions are documented on page 185.)

`\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`

```

8740 \cs_new_protected_nopar:Npn \fp_sub:Nn { \fp_sub_aux:NNn \tl_set:Nn }
8741 \cs_new_protected_nopar:Npn \fp_gsub:Nn { \fp_sub_aux:NNn \tl_gset:Nn }
8742 \cs_generate_variant:Nn \fp_sub:Nn { c }
8743 \cs_generate_variant:Nn \fp_gsub:Nn { c }
8744 \cs_new_protected_nopar:Npn \fp_sub_aux:NNn #1#2#3
8745 {
8746 \group_begin:
8747 \fp_read:N #2
8748 \fp_split:Nn b {#3}
8749 \fp_standardise:NNNN
8750 \l_fp_input_b_sign_int
8751 \l_fp_input_b_integer_int
8752 \l_fp_input_b_decimal_int
8753 \l_fp_input_b_exponent_int
8754 \tex_multiply:D \l_fp_input_b_sign_int \c_minus_one
8755 \fp_add_core:
8756 \fp_tmp:w #1#2
8757 }

```

(End definition for `\fp_sub:Nn` and `\fp_sub:cn`. These functions are documented on page 186.)

```

\fp_mul:Nn
\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:NNNNNNNNN

```

The pattern is much the same for multiplication.

```

8758 \cs_new_protected_nopar:Npn \fp_mul:Nn { \fp_mul_aux:NNn \tl_set:Nn }
8759 \cs_new_protected_nopar:Npn \fp_gmul:Nn { \fp_mul_aux:NNn \tl_gset:Nn }
8760 \cs_generate_variant:Nn \fp_mul:Nn { c }
8761 \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

```

8762 \cs_new_protected_nopar:Npn \fp_mul_aux:NNn #1#2#3
8763 {
8764   \group_begin:
8765   \fp_read:N #2
8766   \fp_split:Nn b {#3}
8767   \fp_standardise:NNNN
8768   \l_fp_input_b_sign_int
8769   \l_fp_input_b_integer_int
8770   \l_fp_input_b_decimal_int
8771   \l_fp_input_b_exponent_int
8772   \fp_mul_internal:
8773   \l_fp_output_exponent_int
8774   \int_eval:w
8775   \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
8776   \scan_stop:
8777   \fp_standardise:NNNN
8778   \l_fp_output_sign_int
8779   \l_fp_output_integer_int
8780   \l_fp_output_decimal_int
8781   \l_fp_output_exponent_int
8782   \cs_set_protected_nopar:Npx \fp_tmp:w
8783   {
8784     \group_end:
8785     #1 \exp_not:N #2
8786     {
8787       \if_int_compare:w
8788       \int_eval:w
8789       \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8790       < \c_zero
8791       \if_int_compare:w
8792       \int_eval:w
8793       \l_fp_output_integer_int + \l_fp_output_decimal_int
8794       = \c_zero
8795       +
8796       \else:
8797       -
8798       \fi:
8799     \else:

```

```

8800          +
8801          \fi:
8802          \int_use:N \l_fp_output_integer_int
8803          .
8804          \exp_after:wN \use_none:n
8805          \int_value:w \int_eval:w
8806          \l_fp_output_decimal_int + \c_one_thousand_million
8807          e
8808          \int_use:N \l_fp_output_exponent_int
8809        }
8810      }
8811    \fp_tmp:w
8812  }

```

Done separately so that the internal use is a bit easier.

```

8813 \cs_new_protected_nopar:Npn \fp_mul_internal:
8814 {
8815   \fp_mul_split:NNNN \l_fp_input_a_decimal_int
8816   \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8817   \fp_mul_split:NNNN \l_fp_input_b_decimal_int
8818   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8819   \l_fp_mul_output_int \c_zero
8820   \tl_clear:N \l_fp_mul_output_tl
8821   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
8822   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8823   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8824   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8825   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_iii_int
8826   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8827   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8828   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_input_b_integer_int
8829   \fp_mul_end_level:
8830   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_ii_int
8831   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8832   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_input_b_integer_int
8833   \fp_mul_end_level:
8834   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_i_int
8835   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_input_b_integer_int
8836   \fp_mul_end_level:
8837   \l_fp_output_decimal_int 0 \l_fp_mul_output_tl \scan_stop:
8838   \tl_clear:N \l_fp_mul_output_tl
8839   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_input_b_integer_int
8840   \fp_mul_end_level:
8841   \l_fp_output_integer_int 0 \l_fp_mul_output_tl \scan_stop:
8842 }

```

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.

```

8843 \cs_new_protected_nopar:Npn \fp_mul_split:NNNN #1#2#3#4
8844 {
8845   \tex_advance:D #1 \c_one_thousand_million
8846   \cs_set_protected_nopar:Npn \fp_mul_split_aux:w
8847     ##1##2##3##4##5##6##7##8##9 \q_stop {
8848     #2 ##2##3##4 \scan_stop:
8849     #3 ##5##6##7 \scan_stop:
8850     #4 ##8##9 \scan_stop:
8851   }
8852   \exp_after:wN \fp_mul_split_aux:w \int_use:N #1 \q_stop
8853   \tex_advance:D #1 -\c_one_thousand_million
8854 }
8855 \cs_new_protected_nopar:Npn \fp_mul_product:NN #1#2
8856 {
8857   \l_fp_mul_output_int
8858   \int_eval:w \l_fp_mul_output_int + #1 * #2 \scan_stop:
8859 }

```

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.

```

8860 \cs_new_protected_nopar:Npn \fp_mul_end_level:
8861 {
8862   \tex_advance:D \l_fp_mul_output_int \c_one_thousand_million
8863   \exp_after:wN \use_i:nn \exp_after:wN
8864   \fp_mul_end_level:NNNNNNNNN \int_use:N \l_fp_mul_output_int
8865 }
8866 \cs_new_protected_nopar:Npn \fp_mul_end_level:NNNNNNNNN #1#2#3#4#5#6#7#8#9
8867 {
8868   \tl_set:Nx \l_fp_mul_output_tl { #7#8#9 \l_fp_mul_output_tl }
8869   \l_fp_mul_output_int #1#2#3#4#5#6 \scan_stop:
8870 }

```

(End definition for `\fp_mul:Nn` and `\fp_mul:cn`. These functions are documented on page 186.)

`\fp_div:Nn`

`\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:`

The pattern is much the same for multiplication.

```

8871 \cs_new_protected_nopar:Npn \fp_div:Nn { \fp_div_aux:NNn \tl_set:Nn }
8872 \cs_new_protected_nopar:Npn \fp_gdiv:Nn { \fp_div_aux:NNn \tl_gset:Nn }
8873 \cs_generate_variant:Nn \fp_div:Nn { c }
8874 \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.

```

8875 \cs_new_protected_nopar:Npn \fp_div_aux:NNn #1#2#3
8876 {

```

```

8877 \group_begin:
8878 \fp_read:N #2
8879 \fp_split:Nn b {#3}
8880 \fp_standardise:NNNN
8881 \l_fp_input_b_sign_int
8882 \l_fp_input_b_integer_int
8883 \l_fp_input_b_decimal_int
8884 \l_fp_input_b_exponent_int
8885 \if_int_compare:w
8886 \int_eval:w
8887 \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
8888 = \c_zero
8889 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8890 {
8891 \group_end:
8892 #1 \exp_not:N #2 { \c_undefined_fp }
8893 }
8894 \else:
8895 \if_int_compare:w
8896 \int_eval:w
8897 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8898 = \c_zero
8899 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8900 {
8901 \group_end:
8902 #1 \exp_not:N #2 { \c_zero_fp }
8903 }
8904 \else:
8905 \exp_after:wN \exp_after:wN \exp_after:wN \fp_div_internal:
8906 \fi:
8907 \fi:
8908 \fp_tmp:w #1##2
8909 }

```

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.

```

8910 \cs_new_protected_nopar:Npn \fp_div_internal: {
8911 \l_fp_output_integer_int \c_zero
8912 \l_fp_output_decimal_int \c_zero
8913 \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
8914 \l_fp_div_offset_int \c_one_hundred_million
8915 \fp_div_loop:
8916 \l_fp_output_exponent_int
8917 \int_eval:w
8918 \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
8919 \scan_stop:

```

```

8920 \fp_standardise:NNNN
8921 \l_fp_output_sign_int
8922 \l_fp_output_integer_int
8923 \l_fp_output_decimal_int
8924 \l_fp_output_exponent_int
8925 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8926 {
8927   \group_end:
8928   ##1 ##2
8929   {
8930     \if_int_compare:w
8931       \int_eval:w
8932         \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8933         < \c_zero
8934     \if_int_compare:w
8935       \int_eval:w
8936         \l_fp_output_integer_int + \l_fp_output_decimal_int
8937         = \c_zero
8938       +
8939     \else:
8940       -
8941     \fi:
8942   \else:
8943     +
8944   \fi:
8945   \int_use:N \l_fp_output_integer_int
8946   .
8947   \exp_after:wN \use_none:n
8948   \int_value:w \int_eval:w
8949     \l_fp_output_decimal_int + \c_one_thousand_million
8950   \int_eval_end:
8951   e
8952   \int_use:N \l_fp_output_exponent_int
8953 }
8954 }
8955 }

```

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.

```

8956 \cs_new_protected_nopar:Npn \fp_div_loop:
8957 {
8958   \l_fp_count_int \c_zero
8959   \fp_div_divide:
8960   \fp_div_store:
8961   \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8962   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8963   \exp_after:wN \fp_div_loop_step:w
8964   \int_use:N \l_fp_input_a_decimal_int \q_stop
8965   \if_int_compare:w

```

```

8966     \int_eval:w \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8967     > \c_zero
8968     \if_int_compare:w \l_fp_div_offset_int > \c_zero
8969     \exp_after:wN \exp_after:wN \exp_after:wN
8970     \fp_div_loop:
8971     \fi:
8972 \fi:
8973 }

```

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 than 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.

```

8974 \cs_new_protected_nopar:Npn \fp_div_divide:
8975 {
8976   \if_int_compare:w \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
8977   \exp_after:wN \fp_div_divide_aux:
8978   \else:
8979   \if_int_compare:w \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
8980   \else:
8981   \if_int_compare:w
8982   \l_fp_input_a_decimal_int < \l_fp_input_b_decimal_int
8983   \else:
8984   \exp_after:wN \exp_after:wN \exp_after:wN
8985   \exp_after:wN \exp_after:wN \exp_after:wN
8986   \exp_after:wN \fp_div_divide_aux:
8987   \fi:
8988   \fi:
8989   \fi:
8990 }
8991 \cs_new_protected_nopar:Npn \fp_div_divide_aux:
8992 {
8993   \tex_advance:D \l_fp_count_int \c_one
8994   \tex_advance:D \l_fp_input_a_integer_int -\l_fp_input_b_integer_int
8995   \tex_advance:D \l_fp_input_a_decimal_int -\l_fp_input_b_decimal_int
8996   \if_int_compare:w \l_fp_input_a_decimal_int < \c_zero
8997   \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
8998   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8999   \fi:
9000   \fp_div_divide:
9001 }

```

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.

```

9002 \cs_new_protected_nopar:Npn \fp_div_store: { }
9003 \cs_new_protected_nopar:Npn \fp_div_store_integer:
9004 {

```

```

9005     \l_fp_output_integer_int \l_fp_count_int
9006     \cs_set_eq:NN \fp_div_store: \fp_div_store_decimal:
9007   }
9008   \cs_new_protected_nopar:Npn \fp_div_store_decimal:
9009   {
9010     \l_fp_output_decimal_int
9011     \int_eval:w
9012       \l_fp_output_decimal_int +
9013       \l_fp_count_int * \l_fp_div_offset_int
9014     \int_eval_end:
9015     \tex_divide:D \l_fp_div_offset_int \c_ten
9016   }
9017   \cs_new_protected_nopar:Npn \fp_div_loop_step:w #1#2#3#4#5#6#7#8#9 \q_stop
9018   {
9019     \l_fp_input_a_integer_int
9020     \int_eval:w #2 + \l_fp_input_a_integer_int \int_eval_end:
9021     \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
9022   }

```

(End definition for `\fp_div:Nn` and `\fp_div:cn`. These functions are documented on page ??.)

183.10 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:NNNNNNNNN` The internal sum is always exactly that: it is always a sum and there is no sign check.

```

9023 \cs_new_protected_nopar:Npn \fp_add:NNNNNNNNN #1#2#3#4#5#6#7#8#9
9024 {
9025   #7 \int_eval:w #1 + #4 \int_eval_end:
9026   #8 \int_eval:w #2 + #5 \int_eval_end:
9027   #9 \int_eval:w #3 + #6 \int_eval_end:
9028   \if_int_compare:w #9 < \c_one_thousand_million
9029   \else:
9030     \tex_advance:D #8 \c_one
9031     \tex_advance:D #9 -\c_one_thousand_million
9032   \fi:
9033   \if_int_compare:w #8 < \c_one_thousand_million
9034   \else:
9035     \tex_advance:D #7 \c_one
9036     \tex_advance:D #8 -\c_one_thousand_million
9037   \fi:
9038 }

```

(End definition for `\fp_add:NNNNNNNNN`. This function is documented on page ??.)

`\fp_sub:NNNNNNNNN` 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.

```

9039 \cs_new_protected_nopar:Npn \fp_sub:NNNNNNNNN #1#2#3#4#5#6#7#8#9
9040 {
9041   #7 \int_eval:w #1 - #4 \int_eval_end:
9042   #8 \int_eval:w #2 - #5 \int_eval_end:
9043   #9 \int_eval:w #3 - #6 \int_eval_end:
9044   \if_int_compare:w #9 < \c_zero
9045     \tex_advance:D #8 \c_minus_one
9046     \tex_advance:D #9 \c_one_thousand_million
9047   \fi:
9048   \if_int_compare:w #8 < \c_zero
9049     \tex_advance:D #7 \c_minus_one
9050     \tex_advance:D #8 \c_one_thousand_million
9051   \fi:
9052   \if_int_compare:w #7 < \c_zero
9053     \if_int_compare:w \int_eval:w #8 + #9 = \c_zero
9054     #7 -#7
9055   \else:
9056     \tex_advance:D #7 \c_one
9057     #8 \int_eval:w \c_one_thousand_million - #8 \int_eval_end:
9058     #9 \int_eval:w \c_one_thousand_million - #9 \int_eval_end:
9059   \fi:
9060   \fi:
9061 }

```

(End definition for `\fp_sub:NNNNNNNNN`. This function is documented on page ??.)

`\fp_mul:NNNNNN` Decimal-part only multiplication but with higher accuracy than the user version.

```

9062 \cs_new_protected_nopar:Npn \fp_mul:NNNNNN #1#2#3#4#5#6
9063 {
9064   \fp_mul_split:NNNN #1
9065   \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
9066   \fp_mul_split:NNNN #2
9067   \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
9068   \fp_mul_split:NNNN #3
9069   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
9070   \fp_mul_split:NNNN #4
9071   \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
9072   \l_fp_mul_output_int \c_zero
9073   \tl_clear:N \l_fp_mul_output_tl
9074   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
9075   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
9076   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
9077   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
9078   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
9079   \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
9080   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
9081   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
9082   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
9083   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
9084   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
9085   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
9086   \fp_mul_end_level:
9087   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
9088   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
9089   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
9090   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
9091   \fp_mul_end_level:
9092   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
9093   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
9094   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
9095   \fp_mul_end_level:
9096   #6 0 \l_fp_mul_output_tl \scan_stop:
9097   \tl_clear:N \l_fp_mul_output_tl
9098   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
9099   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
9100   \fp_mul_end_level:
9101   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
9102   \fp_mul_end_level:
9103   \fp_mul_end_level:
9104   #5 0 \l_fp_mul_output_tl \scan_stop:
9105 }

```

(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.

```

9106 \cs_new_protected_nopar:Npn \fp_mul:NNNNNNNN #1#2#3#4#5#6#7#8#9
9107 {
9108   \fp_mul_split:NNNN #2
9109   \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
9110   \fp_mul_split:NNNN #3
9111   \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
9112   \fp_mul_split:NNNN #5
9113   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
9114   \fp_mul_split:NNNN #6
9115   \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
9116   \l_fp_mul_output_int \c_zero
9117   \tl_clear:N \l_fp_mul_output_tl
9118   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
9119   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
9120   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
9121   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
9122   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
9123   \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
9124   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
9125   \fp_mul_product:NN #1 \l_fp_mul_b_vi_int
9126   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
9127   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
9128   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
9129   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
9130   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
9131   \fp_mul_product:NN \l_fp_mul_a_vi_int #4
9132   \fp_mul_end_level:
9133   \fp_mul_product:NN #1 \l_fp_mul_b_v_int
9134   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
9135   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
9136   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
9137   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
9138   \fp_mul_product:NN \l_fp_mul_a_v_int #4
9139   \fp_mul_end_level:
9140   \fp_mul_product:NN #1 \l_fp_mul_b_iv_int
9141   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
9142   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
9143   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
9144   \fp_mul_product:NN \l_fp_mul_a_iv_int #4
9145   \fp_mul_end_level:
9146   #9 0 \l_fp_mul_output_tl \scan_stop:
9147   \tl_clear:N \l_fp_mul_output_tl
9148   \fp_mul_product:NN #1 \l_fp_mul_b_iii_int
9149   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
9150   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
9151   \fp_mul_product:NN \l_fp_mul_a_iii_int #4
9152   \fp_mul_end_level:
9153   \fp_mul_product:NN #1 \l_fp_mul_b_ii_int

```

```

9154 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
9155 \fp_mul_product:NN \l_fp_mul_a_ii_int #4
9156 \fp_mul_end_level:
9157 \fp_mul_product:NN #1 \l_fp_mul_b_i_int
9158 \fp_mul_product:NN \l_fp_mul_a_i_int #4
9159 \fp_mul_end_level:
9160 #8 0 \l_fp_mul_output_tl \scan_stop:
9161 \tl_clear:N \l_fp_mul_output_tl
9162 \fp_mul_product:NN #1 #4
9163 \fp_mul_end_level:
9164 #7 0 \l_fp_mul_output_tl \scan_stop:
9165 }

```

(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.

```

9166 \cs_new_protected_nopar:Npn \fp_div_integer:NNNNN #1#2#3#4#5
9167 {
9168   \l_fp_tmp_int #1
9169   \tex_divide:D \l_fp_tmp_int #3
9170   \l_fp_tmp_int \int_eval:w #1 - \l_fp_tmp_int * #3 \int_eval_end:
9171   #4 #1
9172   \tex_divide:D #4 #3
9173   #5 #2
9174   \tex_divide:D #5 #3
9175   \tex_multiply:D \l_fp_tmp_int \c_one_thousand
9176   \tex_divide:D \l_fp_tmp_int #3
9177   #5 \int_eval:w #5 + \l_fp_tmp_int * \c_one_million \int_eval_end:
9178   \if_int_compare:w #5 > \c_one_thousand_million
9179     \tex_advance:D #4 \c_one
9180     \tex_advance:D #5 -\c_one_thousand_million
9181   \fi:
9182 }

```

(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:NNNNNNNN
9183 \cs_new_protected_nopar:Npn \fp_extended_normalise:
9184 {
9185   \fp_extended_normalise_aux_i:
9186   \fp_extended_normalise_aux_ii:
9187 }

```

```

9188 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i:
9189 {
9190   \if_int_compare:w \l_fp_input_a_exponent_int > \c_zero
9191     \tex_multiply:D \l_fp_input_a_integer_int \c_ten
9192     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9193     \exp_after:wN \fp_extended_normalise_aux_i:w
9194     \int_use:N \l_fp_input_a_decimal_int \q_stop
9195     \exp_after:wN \fp_extended_normalise_aux_i:
9196   \fi:
9197 }
9198 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i:w
9199 #1#2#3#4#5#6#7#8#9 \q_stop
9200 {
9201   \l_fp_input_a_integer_int
9202   \int_eval:w \l_fp_input_a_integer_int + #2 \scan_stop:
9203   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
9204   \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9205   \exp_after:wN \fp_extended_normalise_aux_ii:w
9206   \int_use:N \l_fp_input_a_extended_int \q_stop
9207 }
9208 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii:w
9209 #1#2#3#4#5#6#7#8#9 \q_stop
9210 {
9211   \l_fp_input_a_decimal_int
9212   \int_eval:w \l_fp_input_a_decimal_int + #2 \scan_stop:
9213   \l_fp_input_a_extended_int #3#4#5#6#7#8#9 0 \scan_stop:
9214   \tex_advance:D \l_fp_input_a_exponent_int \c_minus_one
9215 }
9216 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii:
9217 {
9218   \if_int_compare:w \l_fp_input_a_exponent_int < \c_zero
9219     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9220     \exp_after:wN \use_i:nn \exp_after:wN
9221     \fp_extended_normalise_ii_aux:NNNNNNNNN
9222     \int_use:N \l_fp_input_a_decimal_int
9223     \exp_after:wN \fp_extended_normalise_aux_ii:
9224   \fi:
9225 }
9226 \cs_new_protected_nopar:Npn \fp_extended_normalise_ii_aux:NNNNNNNNN
9227 #1#2#3#4#5#6#7#8#9
9228 {
9229   \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
9230     \l_fp_input_a_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9231   \else:
9232     \tl_set:Nx \l_fp_tmp_tl
9233     {
9234       \int_use:N \l_fp_input_a_integer_int
9235       #1#2#3#4#5#6#7#8
9236     }
9237     \l_fp_input_a_integer_int \c_zero

```

```

9238     \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
9239     \fi:
9240     \tex_divide:D \l_fp_input_a_extended_int \c_ten
9241     \tl_set:Nx \l_fp_tmp_tl
9242     {
9243         #9
9244         \int_use:N \l_fp_input_a_extended_int
9245     }
9246     \l_fp_input_a_extended_int \l_fp_tmp_tl \scan_stop:
9247     \tex_advance:D \l_fp_input_a_exponent_int \c_one
9248 }

```

(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.

```

9249 \cs_new_protected_nopar:Npn \fp_extended_normalise_output:
9250 {
9251     \if_int_compare:w \l_fp_output_integer_int > \c_nine
9252     \tex_advance:D \l_fp_output_integer_int \c_one_thousand_million
9253     \exp_after:wN \use_i:nn \exp_after:wN
9254     \fp_extended_normalise_output_aux_i:NNNNNNNNN
9255     \int_use:N \l_fp_output_integer_int
9256     \exp_after:wN \fp_extended_normalise_output:
9257     \fi:
9258 }
9259 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux_i:NNNNNNNNN
9260 #1#2#3#4#5#6#7#8#9
9261 {
9262     \l_fp_output_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
9263     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9264     \tl_set:Nx \l_fp_tmp_tl
9265     {
9266         #9
9267         \exp_after:wN \use_none:n
9268         \int_use:N \l_fp_output_decimal_int
9269     }
9270     \exp_after:wN \fp_extended_normalise_output_aux_ii:NNNNNNNNN
9271     \l_fp_tmp_tl
9272 }
9273 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux_ii:NNNNNNNNN
9274 #1#2#3#4#5#6#7#8#9
9275 {
9276     \l_fp_output_decimal_int #1#2#3#4#5#6#7#8#9 \scan_stop:
9277     \fp_extended_normalise_output_aux:N
9278 }
9279 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux:N #1
9280 {

```

```

9281 \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
9282 \tex_divide:D \l_fp_output_extended_int \c_ten
9283 \tl_set:Nx \l_fp_tmp_tl
9284 {
9285   #1
9286   \exp_after:wN \use_none:n
9287   \int_use:N \l_fp_output_extended_int
9288 }
9289 \l_fp_output_extended_int \l_fp_tmp_tl \scan_stop:
9290 \tex_advance:D \l_fp_output_exponent_int \c_one
9291 }

```

(End definition for `\fp_extended_normalise_output:`. This function is documented on page ??.)

183.11 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`

```

9292 \cs_new_protected_nopar:Npn \fp_trig_normalise:
9293 {
9294   \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
9295     \l_fp_input_a_extended_int \c_zero
9296     \fp_extended_normalise:
9297     \fp_trig_normalise_aux:
9298     \if_int_compare:w \l_fp_input_a_integer_int < \c_zero
9299       \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
9300       \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
9301     \fi:
9302     \exp_after:wN \fp_trig_octant:
9303   \else:
9304     \l_fp_input_a_sign_int \c_one
9305     \l_fp_output_integer_int \c_zero
9306     \l_fp_output_decimal_int \c_zero
9307     \l_fp_output_exponent_int \c_zero
9308     \exp_after:wN \fp_trig_overflow_msg:
9309   \fi:
9310 }
9311 \cs_new_protected_nopar:Npn \fp_trig_normalise_aux:
9312 {
9313   \if_int_compare:w \l_fp_input_a_integer_int > \c_three
9314     \fp_trig_sub:NNN
9315     \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
9316     \exp_after:wN \fp_trig_normalise_aux:
9317   \else:
9318     \if_int_compare:w \l_fp_input_a_integer_int > \c_two
9319       \if_int_compare:w \l_fp_input_a_decimal_int > \c_fp_pi_decimal_int
9320         \fp_trig_sub:NNN

```

```

9321         \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
9322         \exp_after:wN \exp_after:wN \exp_after:wN
9323         \exp_after:wN \exp_after:wN \exp_after:wN
9324         \exp_after:wN \fp_trig_normalise_aux:
9325         \fi:
9326     \fi:
9327 \fi:
9328 }

```

Here, there may be a sign change but there will never be any variation in the input. So a dedicated function can be used.

```

9329 \cs_new_protected_nopar:Npn \fp_trig_sub:NNN #1#2#3
9330 {
9331     \l_fp_input_a_integer_int
9332     \int_eval:w \l_fp_input_a_integer_int - #1 \int_eval_end:
9333     \l_fp_input_a_decimal_int
9334     \int_eval:w \l_fp_input_a_decimal_int - #2 \int_eval_end:
9335     \l_fp_input_a_extended_int
9336     \int_eval:w \l_fp_input_a_extended_int - #3 \int_eval_end:
9337     \if_int_compare:w \l_fp_input_a_extended_int < \c_zero
9338         \tex_advance:D \l_fp_input_a_decimal_int \c_minus_one
9339         \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9340     \fi:
9341     \if_int_compare:w \l_fp_input_a_decimal_int < \c_zero
9342         \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
9343         \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9344     \fi:
9345     \if_int_compare:w \l_fp_input_a_integer_int < \c_zero
9346         \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
9347     \if_int_compare:w
9348         \int_eval:w
9349         \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
9350         = \c_zero
9351         \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
9352     \else:
9353         \l_fp_input_a_integer_int
9354         \int_eval:w
9355         - \l_fp_input_a_integer_int - \c_one
9356         \int_eval_end:
9357         \l_fp_input_a_decimal_int
9358         \int_eval:w
9359         \c_one_thousand_million - \l_fp_input_a_decimal_int
9360         \int_eval_end:
9361         \l_fp_input_a_extended_int
9362         \int_eval:w
9363         \c_one_thousand_million - \l_fp_input_a_extended_int
9364         \int_eval_end:
9365     \fi:
9366 \fi:

```


9367 }

(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 $\pi/2$.

```

9368 \cs_new_protected_nopar:Npn \fp_trig_octant:
9369 {
9370   \l_fp_trig_octant_int \c_one
9371   \fp_trig_octant_aux:
9372   \if_int_compare:w \l_fp_input_a_decimal_int < \c_ten
9373     \l_fp_input_a_decimal_int \c_zero
9374     \l_fp_input_a_extended_int \c_zero
9375   \fi:
9376   \if_int_odd:w \l_fp_trig_octant_int
9377   \else:
9378     \fp_sub:NNNNNNNNN
9379     \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
9380     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9381     \l_fp_input_a_extended_int
9382     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9383     \l_fp_input_a_extended_int
9384   \fi:
9385 }
9386 \cs_new_protected_nopar:Npn \fp_trig_octant_aux:
9387 {
9388   \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
9389     \fp_sub:NNNNNNNNN
9390     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9391     \l_fp_input_a_extended_int
9392     \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
9393     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9394     \l_fp_input_a_extended_int
9395     \tex_advance:D \l_fp_trig_octant_int \c_one
9396     \exp_after:wN \fp_trig_octant_aux:
9397   \else:
9398     \if_int_compare:w
9399       \l_fp_input_a_decimal_int > \c_fp_pi_by_four_decimal_int
9400     \fp_sub:NNNNNNNNN
9401     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9402     \l_fp_input_a_extended_int
9403     \c_zero \c_fp_pi_by_four_decimal_int
9404     \c_fp_pi_by_four_extended_int
9405     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9406     \l_fp_input_a_extended_int
9407     \tex_advance:D \l_fp_trig_octant_int \c_one
9408     \exp_after:wN \exp_after:wN \exp_after:wN

```

```

9409         \fp_trig_octant_aux:
9410         \fi:
9411     \fi:
9412 }

```

(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` 9413 `\cs_new_protected_nopar:Npn \fp_sin:Nn { \fp_sin_aux:NNn \tl_set:Nn }`
`\fp_sin_aux:NNn` 9414 `\cs_new_protected_nopar:Npn \fp_gsin:Nn { \fp_sin_aux:NNn \tl_gset:Nn }`
`\fp_sin_aux_i:` 9415 `\cs_generate_variant:Nn \fp_sin:Nn { c }`
`\fp_sin_aux_ii:` 9416 `\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} .

```

9417 \cs_new_protected_nopar:Npn \fp_sin_aux:NNn #1#2#3
9418 {
9419     \group_begin:
9420     \fp_split:Nn a {#3}
9421     \fp_standardise:NNNN
9422     \l_fp_input_a_sign_int
9423     \l_fp_input_a_integer_int
9424     \l_fp_input_a_decimal_int
9425     \l_fp_input_a_exponent_int
9426     \tl_set:Nx \l_fp_arg_tl
9427     {
9428         \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9429         -
9430         \else:
9431         +
9432         \fi:
9433         \int_use:N \l_fp_input_a_integer_int
9434         .
9435         \exp_after:wN \use_none:n
9436         \int_value:w \int_eval:w
9437         \l_fp_input_a_decimal_int + \c_one_thousand_million
9438         e
9439         \int_use:N \l_fp_input_a_exponent_int
9440     }
9441     \if_int_compare:w \l_fp_input_a_exponent_int < -\c_five
9442     \cs_set_protected_nopar:Npx \fp_tmp:w
9443     {
9444         \group_end:
9445         #1 \exp_not:N #2 { \l_fp_arg_tl }
9446     }
9447     \else:

```

```

9448     \if_cs_exist:w
9449         c_fp_sin ( \l_fp_arg_tl ) _fp
9450     \cs_end:
9451     \else:
9452         \exp_after:wN \exp_after:wN \exp_after:wN
9453         \fp_sin_aux_i:
9454     \fi:
9455     \cs_set_protected_nopar:Npx \fp_tmp:w
9456     {
9457         \group_end:
9458         #1 \exp_not:N #2
9459         { \use:c { c_fp_sin ( \l_fp_arg_tl ) _fp } }
9460     }
9461     \fi:
9462     \fp_tmp:w
9463 }

```

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.

```

9464 \cs_new_protected_nopar:Npn \fp_sin_aux_i:
9465 {
9466     \fp_trig_normalise:
9467     \fp_sin_aux_ii:
9468     \if_int_compare:w \l_fp_output_integer_int = \c_one
9469         \l_fp_output_exponent_int \c_zero
9470     \else:
9471         \l_fp_output_integer_int \l_fp_output_decimal_int
9472         \l_fp_output_decimal_int \l_fp_output_extended_int
9473         \l_fp_output_exponent_int -\c_nine
9474     \fi:
9475     \fp_standardise:NNNN
9476     \l_fp_input_a_sign_int
9477     \l_fp_output_integer_int
9478     \l_fp_output_decimal_int
9479     \l_fp_output_exponent_int
9480     \tl_new:c { c_fp_sin ( \l_fp_arg_tl ) _fp }
9481     \tl_gset:cx { c_fp_sin ( \l_fp_arg_tl ) _fp }
9482     {
9483         \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9484             +
9485         \else:
9486             -
9487         \fi:
9488         \int_use:N \l_fp_output_integer_int
9489         .
9490         \exp_after:wN \use_none:n
9491         \int_value:w \int_eval:w
9492         \l_fp_output_decimal_int + \c_one_thousand_million

```

```

9493         e
9494         \int_use:N \l_fp_output_exponent_int
9495     }
9496 }
9497 \cs_new_protected_nopar:Npn \fp_sin_aux_ii:
9498 {
9499     \if_case:w \l_fp_trig_octant_int
9500     \or:
9501         \exp_after:wN \fp_trig_calc_sin:
9502     \or:
9503         \exp_after:wN \fp_trig_calc_cos:
9504     \or:
9505         \exp_after:wN \fp_trig_calc_cos:
9506     \or:
9507         \exp_after:wN \fp_trig_calc_sin:
9508     \fi:
9509 }

```

(End definition for `\fp_sin:Nn` and `\fp_sin:cn`. These functions are documented on page 187.)

```

\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:
9510 \cs_new_protected_nopar:Npn \fp_cos:Nn { \fp_cos_aux:NNn \tl_set:Nn }
9511 \cs_new_protected_nopar:Npn \fp_gcos:Nn { \fp_cos_aux:NNn \tl_gset:Nn }
9512 \cs_generate_variant:Nn \fp_cos:Nn { c }
9513 \cs_generate_variant:Nn \fp_gcos:Nn { c }
9514 \cs_new_protected_nopar:Npn \fp_cos_aux:NNn #1#2#3
9515 {
9516     \group_begin:
9517     \fp_split:Nn a {#3}
9518     \fp_standardise:NNNN
9519     \l_fp_input_a_sign_int
9520     \l_fp_input_a_integer_int
9521     \l_fp_input_a_decimal_int
9522     \l_fp_input_a_exponent_int
9523     \tl_set:Nx \l_fp_arg_tl
9524     {
9525         \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9526         -
9527         \else:
9528         +
9529         \fi:
9530         \int_use:N \l_fp_input_a_integer_int
9531         .
9532         \exp_after:wN \use_none:n
9533         \int_value:w \int_eval:w
9534         \l_fp_input_a_decimal_int + \c_one_thousand_million
9535         e
9536         \int_use:N \l_fp_input_a_exponent_int
9537     }

```

```

9538     \if_cs_exist:w c_fp_cos ( \l_fp_arg_tl ) _fp \cs_end:
9539     \else:
9540         \exp_after:wN \fp_cos_aux_i:
9541     \fi:
9542     \cs_set_protected_nopar:Npx \fp_tmp:w
9543     {
9544         \group_end:
9545         #1 \exp_not:N #2
9546         { \use:c { c_fp_cos ( \l_fp_arg_tl ) _fp } }
9547     }
9548     \fp_tmp:w
9549 }

```

Almost the same as for sine: just a bit of correction for the sign of the output.

```

9550 \cs_new_protected_nopar:Npn \fp_cos_aux_i:
9551 {
9552     \fp_trig_normalise:
9553     \fp_cos_aux_ii:
9554     \if_int_compare:w \l_fp_output_integer_int = \c_one
9555     \l_fp_output_exponent_int \c_zero
9556     \else:
9557         \l_fp_output_integer_int \l_fp_output_decimal_int
9558         \l_fp_output_decimal_int \l_fp_output_extended_int
9559         \l_fp_output_exponent_int -\c_nine
9560     \fi:
9561     \fp_standardise:NNNN
9562     \l_fp_input_a_sign_int
9563     \l_fp_output_integer_int
9564     \l_fp_output_decimal_int
9565     \l_fp_output_exponent_int
9566     \tl_new:c { c_fp_cos ( \l_fp_arg_tl ) _fp }
9567     \tl_gset:cx { c_fp_cos ( \l_fp_arg_tl ) _fp }
9568     {
9569         \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9570             +
9571         \else:
9572             -
9573         \fi:
9574         \int_use:N \l_fp_output_integer_int
9575         .
9576         \exp_after:wN \use_none:n
9577         \int_value:w \int_eval:w
9578         \l_fp_output_decimal_int + \c_one_thousand_million
9579         e
9580         \int_use:N \l_fp_output_exponent_int
9581     }
9582 }
9583 \cs_new_protected_nopar:Npn \fp_cos_aux_ii:
9584 {

```

```

9585     \if_case:w \l_fp_trig_octant_int
9586     \or:
9587         \exp_after:wN \fp_trig_calc_cos:
9588     \or:
9589         \exp_after:wN \fp_trig_calc_sin:
9590     \or:
9591         \exp_after:wN \fp_trig_calc_sin:
9592     \or:
9593         \exp_after:wN \fp_trig_calc_cos:
9594     \fi:
9595     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9596         \if_int_compare:w \l_fp_trig_octant_int > \c_two
9597             \l_fp_input_a_sign_int \c_minus_one
9598         \fi:
9599     \else:
9600         \if_int_compare:w \l_fp_trig_octant_int > \c_two
9601         \else:
9602             \l_fp_input_a_sign_int \c_one
9603         \fi:
9604     \fi:
9605 }

```

(End definition for `\fp_cos:Nn` and `\fp_cos:cn`. These functions are documented on page 188.)

`\fp_trig_calc_cos:`
`\fp_trig_calc_sin:`
`\fp_trig_calc_Taylor:`

These functions actually do the calculation for sine and cosine.

```

9606 \cs_new_protected_nopar:Npn \fp_trig_calc_cos:
9607 {
9608     \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
9609         \l_fp_output_integer_int \c_one
9610         \l_fp_output_decimal_int \c_zero
9611     \else:
9612         \l_fp_trig_sign_int \c_minus_one
9613         \fp_mul:NNNNNN
9614             \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9615             \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9616             \l_fp_trig_decimal_int \l_fp_trig_extended_int
9617         \fp_div_integer:NNNNN
9618             \l_fp_trig_decimal_int \l_fp_trig_extended_int
9619             \c_two
9620             \l_fp_trig_decimal_int \l_fp_trig_extended_int
9621         \l_fp_count_int \c_three
9622         \if_int_compare:w \l_fp_trig_extended_int = \c_zero
9623             \if_int_compare:w \l_fp_trig_decimal_int = \c_zero
9624                 \l_fp_output_integer_int \c_one
9625                 \l_fp_output_decimal_int \c_zero
9626                 \l_fp_output_extended_int \c_zero
9627             \else:
9628                 \l_fp_output_integer_int \c_zero
9629                 \l_fp_output_decimal_int \c_one_thousand_million

```

```

9630         \l_fp_output_extended_int \c_zero
9631         \fi:
9632     \else:
9633         \l_fp_output_integer_int \c_zero
9634         \l_fp_output_decimal_int 999999999 \scan_stop:
9635         \l_fp_output_extended_int \c_one_thousand_million
9636         \fi:
9637         \tex_advance:D \l_fp_output_extended_int -\l_fp_trig_extended_int
9638         \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9639         \exp_after:wN \fp_trig_calc_Taylor:
9640     \fi:
9641 }
9642 \cs_new_protected_nopar:Npn \fp_trig_calc_sin:
9643 {
9644     \l_fp_output_integer_int \c_zero
9645     \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
9646         \l_fp_output_decimal_int \c_zero
9647     \else:
9648         \l_fp_output_decimal_int \l_fp_input_a_decimal_int
9649         \l_fp_output_extended_int \l_fp_input_a_extended_int
9650         \l_fp_trig_sign_int \c_one
9651         \l_fp_trig_decimal_int \l_fp_input_a_decimal_int
9652         \l_fp_trig_extended_int \l_fp_input_a_extended_int
9653         \l_fp_count_int \c_two
9654         \exp_after:wN \fp_trig_calc_Taylor:
9655     \fi:
9656 }

```

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.

```

9657 \cs_new_protected_nopar:Npn \fp_trig_calc_Taylor:
9658 {
9659     \l_fp_trig_sign_int -\l_fp_trig_sign_int
9660     \fp_mul:NNNNNN
9661     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9662     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9663     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9664     \fp_mul:NNNNNN
9665     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9666     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9667     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9668     \fp_div_integer:NNNNN
9669     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9670     \l_fp_count_int
9671     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9672     \tex_advance:D \l_fp_count_int \c_one
9673     \fp_div_integer:NNNNN
9674     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9675     \l_fp_count_int

```

```

9676 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9677 \tex_advance:D \l_fp_count_int \c_one
9678 \if_int_compare:w \l_fp_trig_decimal_int > \c_zero
9679 \if_int_compare:w \l_fp_trig_sign_int > \c_zero
9680 \tex_advance:D \l_fp_output_decimal_int \l_fp_trig_decimal_int
9681 \tex_advance:D \l_fp_output_extended_int
9682 \l_fp_trig_extended_int
9683 \if_int_compare:w \l_fp_output_extended_int < \c_one_thousand_million
9684 \else:
9685 \tex_advance:D \l_fp_output_decimal_int \c_one
9686 \tex_advance:D \l_fp_output_extended_int
9687 -\c_one_thousand_million
9688 \fi:
9689 \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
9690 \else:
9691 \tex_advance:D \l_fp_output_integer_int \c_one
9692 \tex_advance:D \l_fp_output_decimal_int
9693 -\c_one_thousand_million
9694 \fi:
9695 \else:
9696 \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9697 \tex_advance:D \l_fp_output_extended_int
9698 -\l_fp_input_a_extended_int
9699 \if_int_compare:w \l_fp_output_extended_int < \c_zero
9700 \tex_advance:D \l_fp_output_decimal_int \c_minus_one
9701 \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
9702 \fi:
9703 \if_int_compare:w \l_fp_output_decimal_int < \c_zero
9704 \tex_advance:D \l_fp_output_integer_int \c_minus_one
9705 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9706 \fi:
9707 \fi:
9708 \exp_after:wN \fp_trig_calc_Taylor:
9709 \fi:
9710 }

```

(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: 9711 \cs_new_protected_nopar:Npn \fp_tan:Nn { \fp_tan_aux:NNn \tl_set:Nn }
\fp_tan_aux_ii: 9712 \cs_new_protected_nopar:Npn \fp_gtan:Nn { \fp_tan_aux:NNn \tl_gset:Nn }
\fp_tan_aux_iii: 9713 \cs_generate_variant:Nn \fp_tan:Nn { c }
\fp_tan_aux_iv: 9714 \cs_generate_variant:Nn \fp_gtan:Nn { c }
9715 \cs_new_protected_nopar:Npn \fp_tan_aux:NNn #1#2#3
9716 {
9717 \group_begin:

```



```

9718 \fp_split:Nn a {#3}
9719 \fp_standardise:NNNN
9720 \l_fp_input_a_sign_int
9721 \l_fp_input_a_integer_int
9722 \l_fp_input_a_decimal_int
9723 \l_fp_input_a_exponent_int
9724 \tl_set:Nx \l_fp_arg_tl
9725 {
9726   \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9727   -
9728   \else:
9729   +
9730   \fi:
9731   \int_use:N \l_fp_input_a_integer_int
9732   .
9733   \exp_after:wN \use_none:n
9734   \int_value:w \int_eval:w
9735   \l_fp_input_a_decimal_int + \c_one_thousand_million
9736   e
9737   \int_use:N \l_fp_input_a_exponent_int
9738 }
9739 \if_int_compare:w \l_fp_input_a_exponent_int < -\c_five
9740 \cs_set_protected_nopar:Npx \fp_tmp:w
9741 {
9742   \group_end:
9743   #1 \exp_not:N #2 { \l_fp_arg_tl }
9744 }
9745 \else:
9746   \if_cs_exist:w
9747     c_fp_tan ( \l_fp_arg_tl ) _fp
9748   \cs_end:
9749   \else:
9750     \exp_after:wN \exp_after:wN \exp_after:wN
9751     \fp_tan_aux_i:
9752   \fi:
9753   \cs_set_protected_nopar:Npx \fp_tmp:w
9754   {
9755     \group_end:
9756     #1 \exp_not:N #2
9757     { \use:c { c_fp_tan ( \l_fp_arg_tl ) _fp } }
9758   }
9759   \fi:
9760   \fp_tmp:w
9761 }

```

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.

```

9762 \cs_new_protected_nopar:Npn \fp_tan_aux_i:
9763 {
9764   \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
9765     \exp_after:wN \fp_tan_aux_ii:
9766   \else:
9767     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9768     \c_zero_fp
9769     \exp_after:wN \fp_trig_overflow_msg:
9770   \fi:
9771 }
9772 \cs_new_protected_nopar:Npn \fp_tan_aux_ii:
9773 {
9774   \fp_trig_normalise:
9775   \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9776     \if_int_compare:w \l_fp_trig_octant_int > \c_two
9777       \l_fp_output_sign_int \c_minus_one
9778     \else:
9779       \l_fp_output_sign_int \c_one
9780   \fi:
9781   \else:
9782     \if_int_compare:w \l_fp_trig_octant_int > \c_two
9783       \l_fp_output_sign_int \c_one
9784     \else:
9785       \l_fp_output_sign_int \c_minus_one
9786   \fi:
9787   \fi:
9788   \fp_cos_aux_ii:
9789   \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
9790     \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
9791       \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9792       \c_undefined_fp
9793     \else:
9794       \exp_after:wN \exp_after:wN \exp_after:wN
9795       \fp_tan_aux_iii:
9796     \fi:
9797   \else:
9798     \exp_after:wN \fp_tan_aux_iii:
9799   \fi:
9800 }

```

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.

```

9801 \cs_new_protected_nopar:Npn \fp_tan_aux_iii:
9802 {
9803   \l_fp_input_b_integer_int \l_fp_output_decimal_int
9804   \l_fp_input_b_decimal_int \l_fp_output_extended_int
9805   \l_fp_input_b_exponent_int -\c_nine
9806   \fp_standardise:NNNN
9807   \l_fp_input_b_sign_int

```

```

9808     \l_fp_input_b_integer_int
9809     \l_fp_input_b_decimal_int
9810     \l_fp_input_b_exponent_int
9811 \fp_sin_aux_ii:
9812 \l_fp_input_a_integer_int \l_fp_output_decimal_int
9813 \l_fp_input_a_decimal_int \l_fp_output_extended_int
9814 \l_fp_input_a_exponent_int -\c_nine
9815 \fp_standardise:NNNN
9816     \l_fp_input_a_sign_int
9817     \l_fp_input_a_integer_int
9818     \l_fp_input_a_decimal_int
9819     \l_fp_input_a_exponent_int
9820 \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
9821 \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
9822     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9823     \c_zero_fp
9824 \else:
9825     \exp_after:wN \exp_after:wN \exp_after:wN \fp_tan_aux_iv:
9826 \fi:
9827 \else:
9828     \exp_after:wN \fp_tan_aux_iv:
9829 \fi:
9830 }
9831 \cs_new_protected_nopar:Npn \fp_tan_aux_iv:
9832 {
9833     \l_fp_output_integer_int \c_zero
9834     \l_fp_output_decimal_int \c_zero
9835     \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
9836     \l_fp_div_offset_int \c_one_hundred_million
9837     \fp_div_loop:
9838     \l_fp_output_exponent_int
9839     \int_eval:w
9840         \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
9841     \int_eval_end:
9842     \fp_standardise:NNNN
9843     \l_fp_output_sign_int
9844     \l_fp_output_integer_int
9845     \l_fp_output_decimal_int
9846     \l_fp_output_exponent_int
9847 \tl_new:c { c_fp_tan ( \l_fp_arg_tl ) _fp }
9848 \tl_gset:cx { c_fp_tan ( \l_fp_arg_tl ) _fp }
9849 {
9850     \if_int_compare:w \l_fp_output_sign_int > \c_zero
9851     +
9852     \else:
9853     -
9854     \fi:
9855     \int_use:N \l_fp_output_integer_int
9856     .
9857     \exp_after:wN \use_none:n

```

```

9858         \int_value:w \int_eval:w
9859         \l_fp_output_decimal_int + \c_one_thousand_million
9860     e
9861     \int_use:N \l_fp_output_exponent_int
9862 }
9863 }

```

(End definition for `\fp_tan:Nn` and `\fp_tan:cn`. These functions are documented on page 188.)

183.12 Exponent and logarithm functions

`\c_fp_exp_1_tl` Calculation of exponentials requires a number of precomputed values: first the positive integers.

```

9864 \tl_const:cn { c_fp_exp_1_tl } { { 2 } { 718281828 } { 459045235 } { 0 } }
9865 \tl_const:cn { c_fp_exp_2_tl } { { 7 } { 389056098 } { 930650227 } { 0 } }
9866 \tl_const:cn { c_fp_exp_3_tl } { { 2 } { 008553692 } { 318766774 } { 1 } }
9867 \tl_const:cn { c_fp_exp_4_tl } { { 5 } { 459815003 } { 314423908 } { 1 } }
9868 \tl_const:cn { c_fp_exp_5_tl } { { 1 } { 484131591 } { 025766034 } { 2 } }
9869 \tl_const:cn { c_fp_exp_6_tl } { { 4 } { 034287934 } { 927351226 } { 2 } }
9870 \tl_const:cn { c_fp_exp_7_tl } { { 1 } { 096633158 } { 428458599 } { 3 } }
9871 \tl_const:cn { c_fp_exp_8_tl } { { 2 } { 980957987 } { 041728275 } { 3 } }
9872 \tl_const:cn { c_fp_exp_9_tl } { { 8 } { 103083927 } { 575384008 } { 3 } }
9873 \tl_const:cn { c_fp_exp_10_tl } { { 2 } { 202646579 } { 480671652 } { 4 } }
9874 \tl_const:cn { c_fp_exp_20_tl } { { 4 } { 851651954 } { 097902280 } { 8 } }
9875 \tl_const:cn { c_fp_exp_30_tl } { { 1 } { 068647458 } { 152446215 } { 13 } }
9876 \tl_const:cn { c_fp_exp_40_tl } { { 2 } { 353852668 } { 370199854 } { 17 } }
9877 \tl_const:cn { c_fp_exp_50_tl } { { 5 } { 184705528 } { 587072464 } { 21 } }
9878 \tl_const:cn { c_fp_exp_60_tl } { { 1 } { 142007389 } { 815684284 } { 26 } }
9879 \tl_const:cn { c_fp_exp_70_tl } { { 2 } { 515438670 } { 919167006 } { 30 } }
9880 \tl_const:cn { c_fp_exp_80_tl } { { 5 } { 540622384 } { 393510053 } { 34 } }
9881 \tl_const:cn { c_fp_exp_90_tl } { { 1 } { 220403294 } { 317840802 } { 39 } }
9882 \tl_const:cn { c_fp_exp_100_tl } { { 2 } { 688117141 } { 816135448 } { 43 } }
9883 \tl_const:cn { c_fp_exp_200_tl } { { 7 } { 225973768 } { 125749258 } { 86 } }

```

`\c_fp_exp_-1_tl` Now the negative integers.

```

9884 \tl_const:cn { c_fp_exp_-1_tl } { { 3 } { 678794411 } { 71442322 } { -1 } }
9885 \tl_const:cn { c_fp_exp_-2_tl } { { 1 } { 353352832 } { 366132692 } { -1 } }
9886 \tl_const:cn { c_fp_exp_-3_tl } { { 4 } { 978706836 } { 786394298 } { -2 } }
9887 \tl_const:cn { c_fp_exp_-4_tl } { { 1 } { 831563888 } { 873418029 } { -2 } }
9888 \tl_const:cn { c_fp_exp_-5_tl } { { 6 } { 737946999 } { 085467097 } { -3 } }
9889 \tl_const:cn { c_fp_exp_-6_tl } { { 2 } { 478752176 } { 666358423 } { -3 } }
9890 \tl_const:cn { c_fp_exp_-7_tl } { { 9 } { 118819655 } { 545162080 } { -4 } }
9891 \tl_const:cn { c_fp_exp_-8_tl } { { 3 } { 354626279 } { 025118388 } { -4 } }
9892 \tl_const:cn { c_fp_exp_-9_tl } { { 1 } { 234098040 } { 866795495 } { -4 } }
9893 \tl_const:cn { c_fp_exp_-10_tl } { { 4 } { 539992976 } { 248451536 } { -5 } }
9894 \tl_const:cn { c_fp_exp_-20_tl } { { 2 } { 061153622 } { 438557828 } { -9 } }
9895 \tl_const:cn { c_fp_exp_-30_tl } { { 9 } { 357622968 } { 840174605 } { -14 } }

```

```

9896 \tl_const:cn { c_fp_exp-40_tl } { { 4 } { 248354255 } { 291588995 } { -18 } }
9897 \tl_const:cn { c_fp_exp-50_tl } { { 1 } { 928749847 } { 963917783 } { -22 } }
9898 \tl_const:cn { c_fp_exp-60_tl } { { 8 } { 756510762 } { 696520338 } { -27 } }
9899 \tl_const:cn { c_fp_exp-70_tl } { { 3 } { 975449735 } { 908646808 } { -31 } }
9900 \tl_const:cn { c_fp_exp-80_tl } { { 1 } { 804851387 } { 845415172 } { -35 } }
9901 \tl_const:cn { c_fp_exp-90_tl } { { 8 } { 194012623 } { 990515430 } { -40 } }
9902 \tl_const:cn { c_fp_exp-100_tl } { { 3 } { 720075976 } { 020835963 } { -44 } }
9903 \tl_const:cn { c_fp_exp-200_tl } { { 1 } { 383896526 } { 736737530 } { -87 } }

```

\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
9904 \cs_new_protected_nopar:Npn \fp_exp:Nn { \fp_exp_aux:NNn \tl_set:Nn }
9905 \cs_new_protected_nopar:Npn \fp_gexp:Nn { \fp_exp_aux:NNn \tl_gset:Nn }
9906 \cs_generate_variant:Nn \fp_exp:Nn { c }
9907 \cs_generate_variant:Nn \fp_gexp:Nn { c }
9908 \cs_new_protected_nopar:Npn \fp_exp_aux:NNn #1#2#3
9909 {
9910   \group_begin:
9911   \fp_split:Nn a {#3}
9912   \fp_standardise:NNNN
9913   \l_fp_input_a_sign_int
9914   \l_fp_input_a_integer_int
9915   \l_fp_input_a_decimal_int
9916   \l_fp_input_a_exponent_int
9917   \l_fp_input_a_extended_int \c_zero
9918   \tl_set:Nx \l_fp_arg_tl
9919   {
9920     \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9921     -
9922     \else:
9923     +
9924     \fi:
9925     \int_use:N \l_fp_input_a_integer_int
9926     .
9927     \exp_after:wN \use_none:n
9928     \int_value:w \int_eval:w
9929     \l_fp_input_a_decimal_int + \c_one_thousand_million
9930     e
9931     \int_use:N \l_fp_input_a_exponent_int
9932   }
9933   \if_cs_exist:w c_fp_exp ( \l_fp_arg_tl ) _fp \cs_end:
9934   \else:
9935     \exp_after:wN \fp_exp_internal:
9936     \fi:
9937   \cs_set_protected_nopar:Npx \fp_tmp:w
9938   {
9939     \group_end:

```

```

9940         #1 \exp_not:N #2
9941         {
9942             \if_cs_exist:w c_fp_exp ( \l_fp_arg_tl ) _fp
9943             \cs_end:
9944             \use:c { c_fp_exp ( \l_fp_arg_tl ) _fp }
9945             \else:
9946             \c_zero_fp
9947             \fi:
9948         }
9949     }
9950     \fp_tmp:w
9951 }

```

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

```

9952 \cs_new_protected_nopar:Npn \fp_exp_internal:
9953 {
9954     \if_int_compare:w \l_fp_input_a_exponent_int < \c_three
9955     \fp_extended_normalise:
9956     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9957     \if_int_compare:w \l_fp_input_a_integer_int < 230 \scan_stop:
9958     \exp_after:wN \exp_after:wN \exp_after:wN
9959     \exp_after:wN \exp_after:wN \exp_after:wN
9960     \exp_after:wN \fp_exp_aux:
9961     \else:
9962     \exp_after:wN \exp_after:wN \exp_after:wN
9963     \exp_after:wN \exp_after:wN \exp_after:wN
9964     \exp_after:wN \fp_exp_overflow_msg:
9965     \fi:
9966     \else:
9967     \if_int_compare:w \l_fp_input_a_integer_int < 230 \scan_stop:
9968     \exp_after:wN \exp_after:wN \exp_after:wN
9969     \exp_after:wN \exp_after:wN \exp_after:wN
9970     \exp_after:wN \fp_exp_aux:
9971     \else:
9972     \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9973     { \c_zero_fp }
9974     \fi:
9975     \fi:
9976     \else:
9977     \exp_after:wN \fp_exp_overflow_msg:
9978     \fi:
9979 }

```

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.

```

9980 \cs_new_protected_nopar:Npn \fp_exp_aux:
9981 {
9982   \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
9983     \exp_after:wN \fp_exp_integer:
9984   \else:
9985     \l_fp_output_integer_int \c_one
9986     \l_fp_output_decimal_int \c_zero
9987     \l_fp_output_extended_int \c_zero
9988     \l_fp_output_exponent_int \c_zero
9989     \exp_after:wN \fp_exp_decimal:
9990   \fi:
9991 }

```

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).

```

9992 \cs_new_protected_nopar:Npn \fp_exp_integer:
9993 {
9994   \if_int_compare:w \l_fp_input_a_integer_int < \c_one_hundred
9995     \l_fp_exp_integer_int \c_one
9996     \l_fp_exp_decimal_int \c_zero
9997     \l_fp_exp_extended_int \c_zero
9998     \l_fp_exp_exponent_int \c_zero
9999     \exp_after:wN \fp_exp_integer_tens:
10000   \else:
10001     \tl_set:Nx \l_fp_tmp_tl
10002     {
10003       \exp_after:wN \use_i:nnn
10004       \int_use:N \l_fp_input_a_integer_int
10005     }
10006     \l_fp_input_a_integer_int
10007     \int_eval:w
10008     \l_fp_input_a_integer_int - \l_fp_tmp_tl 00
10009     \int_eval_end:
10010     \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
10011       \if_int_compare:w \l_fp_output_integer_int > 200 \scan_stop:
10012       \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
10013       { \c_zero_fp }
10014     \else:
10015       \fp_exp_integer_const:n { - \l_fp_tmp_tl 00 }
10016       \exp_after:wN \exp_after:wN \exp_after:wN
10017       \exp_after:wN \exp_after:wN \exp_after:wN

```

```

10018         \exp_after:wN \fp_exp_integer_tens:
10019         \fi:
10020     \else:
10021         \fp_exp_integer_const:n { \l_fp_tmp_tl 00 }
10022         \exp_after:wN \exp_after:wN \exp_after:wN
10023         \exp_after:wN \fp_exp_integer_tens:
10024     \fi:
10025 \fi:
10026 }

```

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.

```

10027 \cs_new_protected_nopar:Npn \fp_exp_integer_tens:
10028 {
10029     \l_fp_output_integer_int \l_fp_exp_integer_int
10030     \l_fp_output_decimal_int \l_fp_exp_decimal_int
10031     \l_fp_output_extended_int \l_fp_exp_extended_int
10032     \l_fp_output_exponent_int \l_fp_exp_exponent_int
10033     \if_int_compare:w \l_fp_input_a_integer_int > \c_nine
10034         \tl_set:Nx \l_fp_tmp_tl
10035         {
10036             \exp_after:wN \use_i:nn
10037             \int_use:N \l_fp_input_a_integer_int
10038         }
10039     \l_fp_input_a_integer_int
10040     \int_eval:w
10041         \l_fp_input_a_integer_int - \l_fp_tmp_tl 0
10042     \int_eval_end:
10043     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10044         \fp_exp_integer_const:n { \l_fp_tmp_tl 0 }
10045     \else:
10046         \fp_exp_integer_const:n { - \l_fp_tmp_tl 0 }
10047     \fi:
10048     \fp_mul:NNNNNNNNN
10049         \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10050         \l_fp_output_integer_int \l_fp_output_decimal_int
10051         \l_fp_output_extended_int
10052         \l_fp_output_integer_int \l_fp_output_decimal_int
10053         \l_fp_output_extended_int
10054     \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
10055     \fp_extended_normalise_output:
10056     \fi:
10057     \fp_exp_integer_units:
10058 }
10059 \cs_new_protected_nopar:Npn \fp_exp_integer_units:
10060 {
10061     \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
10062     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero

```



```

10063     \fp_exp_integer_const:n { \int_use:N \l_fp_input_a_integer_int }
10064 \else:
10065     \fp_exp_integer_const:n
10066     { - \int_use:N \l_fp_input_a_integer_int }
10067 \fi:
10068 \fp_mul:NNNNNNNNN
10069     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10070     \l_fp_output_integer_int \l_fp_output_decimal_int
10071     \l_fp_output_extended_int
10072     \l_fp_output_integer_int \l_fp_output_decimal_int
10073     \l_fp_output_extended_int
10074     \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
10075     \fp_extended_normalise_output:
10076 \fi:
10077 \fp_exp_decimal:
10078 }

```

Recovery of the stored constant values into the separate registers is done with a simple expansion then assignment.

```

10079 \cs_new_protected_nopar:Npn \fp_exp_integer_const:n #1
10080 {
10081     \exp_after:wN \exp_after:wN \exp_after:wN
10082     \fp_exp_integer_const:nnnn
10083     \cs:w c_fp_exp_ #1 _tl \cs_end:
10084 }
10085 \cs_new_protected_nopar:Npn \fp_exp_integer_const:nnnn #1#2#3#4
10086 {
10087     \l_fp_exp_integer_int #1 \scan_stop:
10088     \l_fp_exp_decimal_int #2 \scan_stop:
10089     \l_fp_exp_extended_int #3 \scan_stop:
10090     \l_fp_exp_exponent_int #4 \scan_stop:
10091 }

```

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.

```

10092 \cs_new_protected_nopar:Npn \fp_exp_decimal:
10093 {
10094     \if_int_compare:w \l_fp_input_a_decimal_int > \c_zero
10095     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10096         \l_fp_exp_integer_int \c_one
10097         \l_fp_exp_decimal_int \l_fp_input_a_decimal_int
10098         \l_fp_exp_extended_int \l_fp_input_a_extended_int
10099     \else:
10100         \l_fp_exp_integer_int \c_zero
10101         \if_int_compare:w \l_fp_exp_extended_int = \c_zero
10102             \l_fp_exp_decimal_int

```

```

10103         \int_eval:w
10104         \c_one_thousand_million - \l_fp_input_a_decimal_int
10105         \int_eval_end:
10106         \l_fp_exp_extended_int \c_zero
10107     \else:
10108         \l_fp_exp_decimal_int
10109         \int_eval:w
10110         999999999 - \l_fp_input_a_decimal_int
10111         \scan_stop:
10112         \l_fp_exp_extended_int
10113         \int_eval:w
10114         \c_one_thousand_million - \l_fp_input_a_extended_int
10115         \int_eval_end:
10116     \fi:
10117 \fi:
10118 \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10119 \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
10120 \l_fp_input_b_extended_int  \l_fp_input_a_extended_int
10121 \l_fp_count_int \c_one
10122 \fp_exp_Taylor:
10123 \fp_mul:NNNNNNNNN
10124     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10125     \l_fp_output_integer_int \l_fp_output_decimal_int
10126     \l_fp_output_extended_int
10127     \l_fp_output_integer_int \l_fp_output_decimal_int
10128     \l_fp_output_extended_int
10129 \fi:
10130 \if_int_compare:w \l_fp_output_extended_int < \c_five_hundred_million
10131 \else:
10132     \tex_advance:D \l_fp_output_decimal_int \c_one
10133     \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
10134     \else:
10135         \l_fp_output_decimal_int \c_zero
10136         \tex_advance:D \l_fp_output_integer_int \c_one
10137     \fi:
10138 \fi:
10139 \fp_standardise:NNNN
10140     \l_fp_output_sign_int
10141     \l_fp_output_integer_int
10142     \l_fp_output_decimal_int
10143     \l_fp_output_exponent_int
10144 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
10145 {
10146     +
10147     \int_use:N \l_fp_output_integer_int
10148     .
10149     \exp_after:wN \use_none:n
10150     \int_value:w \int_eval:w
10151         \l_fp_output_decimal_int + \c_one_thousand_million
10152     e

```

```

10153         \int_use:N \l_fp_output_exponent_int
10154     }
10155 }

```

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`.

```

10156 \cs_new_protected_nopar:Npn \fp_exp_Taylor:
10157 {
10158     \tex_advance:D \l_fp_count_int \c_one
10159     \tex_multiply:D \l_fp_input_b_sign_int \l_fp_input_a_sign_int
10160     \fp_mul:NNNNNN
10161     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10162     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10163     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10164     \fp_div_integer:NNNNN
10165     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10166     \l_fp_count_int
10167     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10168     \if_int_compare:w
10169         \int_eval:w
10170         \l_fp_input_b_decimal_int + \l_fp_input_b_extended_int
10171         > \c_zero
10172     \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
10173         \tex_advance:D \l_fp_exp_decimal_int \l_fp_input_b_decimal_int
10174         \tex_advance:D \l_fp_exp_extended_int
10175         \l_fp_input_b_extended_int
10176         \if_int_compare:w \l_fp_exp_extended_int < \c_one_thousand_million
10177     \else:
10178         \tex_advance:D \l_fp_exp_decimal_int \c_one
10179         \tex_advance:D \l_fp_exp_extended_int
10180         -\c_one_thousand_million
10181     \fi:
10182     \if_int_compare:w \l_fp_exp_decimal_int < \c_one_thousand_million
10183     \else:
10184         \tex_advance:D \l_fp_exp_integer_int \c_one
10185         \tex_advance:D \l_fp_exp_decimal_int
10186         -\c_one_thousand_million
10187     \fi:
10188     \else:
10189         \tex_advance:D \l_fp_exp_decimal_int -\l_fp_input_b_decimal_int
10190         \tex_advance:D \l_fp_exp_extended_int
10191         -\l_fp_input_a_extended_int

```

```

10192     \if_int_compare:w \l_fp_exp_extended_int < \c_zero
10193     \tex_advance:D \l_fp_exp_decimal_int \c_minus_one
10194     \tex_advance:D \l_fp_exp_extended_int \c_one_thousand_million
10195     \fi:
10196     \if_int_compare:w \l_fp_exp_decimal_int < \c_zero
10197     \tex_advance:D \l_fp_exp_integer_int \c_minus_one
10198     \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
10199     \fi:
10200     \fi:
10201     \exp_after:wN \fp_exp_Taylor:
10202     \fi:
10203 }

```

This is set up as a function so that the power code can redirect the effect.

```

10204 \cs_new_protected_nopar:Npn \fp_exp_const:Nx #1#2
10205 {
10206     \tl_new:N #1
10207     \tl_gset:Nx #1 {#2}
10208 }
10209 \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 [187](#).)

```

\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.

```

10210 \tl_const:cn { c_fp_ln_10_1_tl } { { 2 } { 302585092 } { 994045684 } { 0 } }
10211 \tl_const:cn { c_fp_ln_10_2_tl } { { 4 } { 605170185 } { 988091368 } { 0 } }
10212 \tl_const:cn { c_fp_ln_10_3_tl } { { 6 } { 907755278 } { 982137052 } { 0 } }
10213 \tl_const:cn { c_fp_ln_10_4_tl } { { 9 } { 210340371 } { 976182736 } { 0 } }
10214 \tl_const:cn { c_fp_ln_10_5_tl } { { 1 } { 151292546 } { 497022842 } { 1 } }
10215 \tl_const:cn { c_fp_ln_10_6_tl } { { 1 } { 381551055 } { 796427410 } { 1 } }
10216 \tl_const:cn { c_fp_ln_10_7_tl } { { 1 } { 611809565 } { 095831979 } { 1 } }
10217 \tl_const:cn { c_fp_ln_10_8_tl } { { 1 } { 842068074 } { 395226547 } { 1 } }
10218 \tl_const:cn { c_fp_ln_10_9_tl } { { 2 } { 072326583 } { 694641116 } { 1 } }

```

```

\c_fp_ln_2_1_tl
\c_fp_ln_2_2_tl
\c_fp_ln_2_3_tl

```

The smaller set for powers of two.

```

10219 \tl_const:cn { c_fp_ln_2_1_tl } { { 0 } { 693147180 } { 559945309 } { 0 } }
10220 \tl_const:cn { c_fp_ln_2_2_tl } { { 1 } { 386294361 } { 119890618 } { 0 } }
10221 \tl_const:cn { c_fp_ln_2_3_tl } { { 2 } { 079441541 } { 679835928 } { 0 } }

```

`\fp_ln:Nn` The approach for logarithms is again based on a mix of tables and Taylor series. Here, the initial validation is a bit easier and so it is set up earlier, meaning less need to escape later on.

`\fp_ln:cn`

`\fp_gln:Nn`

`\fp_gln:cn`

```

\fp_ln_aux:NNn
\fp_ln_aux:

```

```

10222 \cs_new_protected_nopar:Npn \fp_ln:Nn { \fp_ln_aux:NNn \tl_set:Nn }
10223 \cs_new_protected_nopar:Npn \fp_gln:Nn { \fp_ln_aux:NNn \tl_gset:Nn }

```

```

\fp_ln_exponent:
\fp_ln_internal:

```

```

10224 \cs_generate_variant:Nn \fp_ln:Nn { c }
10225 \cs_generate_variant:Nn \fp_gln:Nn { c }

```

`\fp_ln_exponent_tens:`

`\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`

`\fp_ln_Taylor:`

```

10226 \cs_new_protected_nopar:Npn \fp_ln_aux:NNn #1#2#3
10227 {
10228   \group_begin:
10229   \fp_split:Nn a {#3}
10230   \fp_standardise:NNNN
10231   \l_fp_input_a_sign_int
10232   \l_fp_input_a_integer_int
10233   \l_fp_input_a_decimal_int
10234   \l_fp_input_a_exponent_int
10235   \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10236   \if_int_compare:w
10237     \int_eval:w
10238     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10239     > \c_zero
10240     \exp_after:wN \exp_after:wN \exp_after:wN \fp_ln_aux:
10241   \else:
10242     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10243     {
10244       \group_end:
10245       ##1 \exp_not:N ##2 { \c_zero_fp }
10246     }
10247     \exp_after:wN \exp_after:wN \exp_after:wN \fp_ln_error_msg:
10248   \fi:
10249 \else:
10250   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10251   {
10252     \group_end:
10253     ##1 \exp_not:N ##2 { \c_zero_fp }
10254   }
10255   \exp_after:wN \fp_ln_error_msg:
10256 \fi:
10257 \fp_tmp:w #1 #2
10258 }

```

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.

```

10259 \cs_new_protected_nopar:Npn \fp_ln_aux:
10260 {
10261   \tl_set:Nx \l_fp_arg_tl
10262   {
10263     +
10264     \int_use:N \l_fp_input_a_integer_int
10265     .
10266     \exp_after:wN \use_none:n
10267     \int_value:w \int_eval:w
10268     \l_fp_input_a_decimal_int + \c_one_thousand_million
10269     e

```

```

10270         \int_use:N \l_fp_input_a_exponent_int
10271     }
10272     \if_cs_exist:w c_fp_ln ( \l_fp_arg_tl ) _fp \cs_end:
10273     \else:
10274         \exp_after:wN \fp_ln_exponent:
10275     \fi:
10276     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10277     {
10278         \group_end:
10279         ##1 \exp_not:N ##2
10280         { \use:c { c_fp_ln ( \l_fp_arg_tl ) _fp } }
10281     }
10282 }

```

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.

$$\ln(10^n \times 2^m \times x) = \ln(10^n) + \ln(2^m) + \ln(x)$$

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.

```

10283 \cs_new_protected_nopar:Npn \fp_ln_exponent:
10284 {
10285     \fp_ln_internal:
10286     \if_int_compare:w \l_fp_output_extended_int < \c_five_hundred_million
10287     \else:
10288         \tex_advance:D \l_fp_output_decimal_int \c_one
10289         \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
10290         \else:
10291             \l_fp_output_decimal_int \c_zero
10292             \tex_advance:D \l_fp_output_integer_int \c_one
10293         \fi:
10294     \fi:
10295     \fp_standardise:NNNN
10296     \l_fp_output_sign_int
10297     \l_fp_output_integer_int
10298     \l_fp_output_decimal_int
10299     \l_fp_output_exponent_int
10300     \tl_const:cx { c_fp_ln ( \l_fp_arg_tl ) _fp }
10301     {
10302         \if_int_compare:w \l_fp_output_sign_int > \c_zero

```

```

10303         +
10304         \else:
10305         -
10306         \fi:
10307         \int_use:N \l_fp_output_integer_int
10308         .
10309         \exp_after:wN \use_none:n
10310         \int_value:w \int_eval:w
10311         \l_fp_output_decimal_int + \c_one_thousand_million
10312         \scan_stop:
10313         e
10314         \int_use:N \l_fp_output_exponent_int
10315     }
10316 }
10317 \cs_new_protected_nopar:Npn \fp_ln_internal:
10318 {
10319     \if_int_compare:w \l_fp_input_a_exponent_int < \c_zero
10320     \l_fp_input_a_exponent_int -\l_fp_input_a_exponent_int
10321     \l_fp_output_sign_int \c_minus_one
10322     \else:
10323     \l_fp_output_sign_int \c_one
10324     \fi:
10325     \if_int_compare:w \l_fp_input_a_exponent_int > \c_nine
10326     \exp_after:wN \fp_ln_exponent_tens:NN
10327     \int_use:N \l_fp_input_a_exponent_int
10328     \else:
10329     \l_fp_output_integer_int \c_zero
10330     \l_fp_output_decimal_int \c_zero
10331     \l_fp_output_extended_int \c_zero
10332     \l_fp_output_exponent_int \c_zero
10333     \fi:
10334     \fp_ln_exponent_units:
10335 }
10336 \cs_new_protected_nopar:Npn \fp_ln_exponent_tens:NN #1 #2
10337 {
10338     \l_fp_input_a_exponent_int #2 \scan_stop:
10339     \fp_ln_const:nn { 10 } { #1 }
10340     \tex_advance:D \l_fp_exp_exponent_int \c_one
10341     \l_fp_output_integer_int \l_fp_exp_integer_int
10342     \l_fp_output_decimal_int \l_fp_exp_decimal_int
10343     \l_fp_output_extended_int \l_fp_exp_extended_int
10344     \l_fp_output_exponent_int \l_fp_exp_exponent_int
10345 }

```

Next the smaller powers of ten, which will need to be combined with the above: always an additive process.

```

10346 \cs_new_protected_nopar:Npn \fp_ln_exponent_units:
10347 {
10348     \if_int_compare:w \l_fp_input_a_exponent_int > \c_zero

```

```

10349 \fp_ln_const:nn { 10 } { \int_use:N \l_fp_input_a_exponent_int }
10350 \fp_ln_normalise:
10351 \fp_add:NNNNNNNNN
10352 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10353 \l_fp_output_integer_int \l_fp_output_decimal_int
10354 \l_fp_output_extended_int
10355 \l_fp_output_integer_int \l_fp_output_decimal_int
10356 \l_fp_output_extended_int
10357 \fi:
10358 \fp_ln_mantissa:
10359 }

```

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. The only case where the new exponent is larger than the old is if there was no previous part. Then simply set the exponent.

```

10360 \cs_new_protected_nopar:Npn \fp_ln_normalise:
10361 {
10362 \if_int_compare:w \l_fp_exp_exponent_int < \l_fp_output_exponent_int
10363 \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
10364 \exp_after:wN \use_i:nn \exp_after:wN
10365 \fp_ln_normalise_aux:NNNNNNNNN
10366 \int_use:N \l_fp_exp_decimal_int
10367 \exp_after:wN \fp_ln_normalise:
10368 \else:
10369 \l_fp_output_exponent_int \l_fp_exp_exponent_int
10370 \fi:
10371 }
10372 \cs_new_protected_nopar:Npn \fp_ln_normalise_aux:NNNNNNNN #1#2#3#4#5#6#7#8#9
10373 {
10374 \if_int_compare:w \l_fp_exp_integer_int = \c_zero
10375 \l_fp_exp_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
10376 \else:
10377 \tl_set:Nx \l_fp_tmp_tl
10378 {
10379 \int_use:N \l_fp_exp_integer_int
10380 #1#2#3#4#5#6#7#8
10381 }
10382 \l_fp_exp_integer_int \c_zero
10383 \l_fp_exp_decimal_int \l_fp_tmp_tl \scan_stop:
10384 \fi:
10385 \tex_divide:D \l_fp_exp_extended_int \c_ten
10386 \tl_set:Nx \l_fp_tmp_tl
10387 {
10388 #9
10389 \int_use:N \l_fp_exp_extended_int
10390 }
10391 \l_fp_exp_extended_int \l_fp_tmp_tl \scan_stop:
10392 \tex_advance:D \l_fp_exp_exponent_int \c_one

```



```
10393 }
```

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*.

```
10394 \cs_new_protected_nopar:Npn \fp_ln_mantissa:
10395 {
10396   \l_fp_count_int \c_zero
10397   \l_fp_input_a_extended_int \c_zero
10398   \fp_ln_mantissa_aux:
10399   \if_int_compare:w \l_fp_count_int > \c_zero
10400     \fp_ln_const:nn { 2 } { \int_use:N \l_fp_count_int }
10401     \fp_ln_normalise:
10402     \if_int_compare:w \l_fp_output_sign_int > \c_zero
10403       \exp_after:wN \fp_add:NNNNNNNNN
10404     \else:
10405       \exp_after:wN \fp_sub:NNNNNNNNN
10406     \fi:
10407     \l_fp_output_integer_int \l_fp_output_decimal_int
10408     \l_fp_output_extended_int
10409     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10410     \l_fp_output_integer_int \l_fp_output_decimal_int
10411     \l_fp_output_extended_int
10412   \fi:
10413   \if_int_compare:w
10414     \int_eval:w
10415     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int > \c_one
10416     \exp_after:wN \fp_ln_Taylor:
10417   \fi:
10418 }
10419 \cs_new_protected_nopar:Npn \fp_ln_mantissa_aux:
10420 {
10421   \if_int_compare:w \l_fp_input_a_integer_int > \c_one
10422     \tex_advance:D \l_fp_count_int \c_one
10423     \fp_ln_mantissa_divide_two:
10424     \exp_after:wN \fp_ln_mantissa_aux:
10425   \fi:
10426 }
```

A fast one-shot division by two.

```
10427 \cs_new_protected_nopar:Npn \fp_ln_mantissa_divide_two:
10428 {
10429   \if_int_odd:w \l_fp_input_a_decimal_int
10430     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
10431   \fi:
10432   \if_int_odd:w \l_fp_input_a_integer_int
10433     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
10434   \fi:
```

```

10435 \tex_divide:D \l_fp_input_a_integer_int \c_two
10436 \tex_divide:D \l_fp_input_a_decimal_int \c_two
10437 \tex_divide:D \l_fp_input_a_extended_int \c_two
10438 }

```

Recovering constants makes use of the same auxiliary code as for exponents.

```

10439 \cs_new_protected_nopar:Npn \fp_ln_const:nn #1#2
10440 {
10441   \exp_after:wN \exp_after:wN \exp_after:wN
10442   \fp_exp_integer_const:nnnn
10443   \cs:w c_fp_ln_ #1 _ #2 _tl \cs_end:
10444 }

```

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”.

```

10445 \cs_new_protected_nopar:Npn \fp_ln_Taylor:
10446 {
10447   \group_begin:
10448   \l_fp_input_a_integer_int \c_zero
10449   \l_fp_input_a_exponent_int \c_zero
10450   \l_fp_input_b_integer_int \c_two
10451   \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
10452   \l_fp_input_b_exponent_int \c_zero
10453   \fp_div_internal:

```

```

10454 \fp_ln_fixed:
10455 \l_fp_input_a_integer_int \l_fp_output_integer_int
10456 \l_fp_input_a_decimal_int \l_fp_output_decimal_int
10457 \l_fp_input_a_extended_int \c_zero
10458 \l_fp_input_a_exponent_int \l_fp_output_exponent_int
10459 \l_fp_output_decimal_int \c_zero %^^A Bug?
10460 \l_fp_output_decimal_int \l_fp_input_a_decimal_int
10461 \l_fp_output_extended_int \l_fp_input_a_extended_int
10462 \fp_mul:NNNNNN
10463 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10464 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10465 \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10466 \l_fp_count_int \c_one
10467 \fp_ln_Taylor_aux:
10468 \cs_set_protected_nopar:Npx \fp_tmp:w
10469 {
10470 \group_end:
10471 \l_fp_exp_integer_int \c_zero
10472 \exp_not:N \l_fp_exp_decimal_int
10473 \int_use:N \l_fp_output_decimal_int \scan_stop:
10474 \exp_not:N \l_fp_exp_extended_int
10475 \int_use:N \l_fp_output_extended_int \scan_stop:
10476 \exp_not:N \l_fp_exp_exponent_int
10477 \int_use:N \l_fp_output_exponent_int \scan_stop:
10478 }
10479 \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.

```

10480 \tex_advance:D \l_fp_exp_decimal_int \l_fp_exp_decimal_int
10481 \if_int_compare:w \l_fp_exp_extended_int < \c_five_hundred_million
10482 \else:
10483 \tex_advance:D \l_fp_exp_extended_int -\c_five_hundred_million
10484 \tex_advance:D \l_fp_exp_decimal_int \c_one
10485 \fi:
10486 \tex_advance:D \l_fp_exp_extended_int \l_fp_exp_extended_int
10487 \fp_ln_normalise:
10488 \if_int_compare:w \l_fp_output_sign_int > \c_zero
10489 \exp_after:wN \fp_add:NNNNNNNNN
10490 \else:
10491 \exp_after:wN \fp_sub:NNNNNNNNN
10492 \fi:
10493 \l_fp_output_integer_int \l_fp_output_decimal_int
10494 \l_fp_output_extended_int
10495 \c_zero \l_fp_exp_decimal_int \l_fp_exp_extended_int
10496 \l_fp_output_integer_int \l_fp_output_decimal_int
10497 \l_fp_output_extended_int
10498 }

```

The usual shifts to move to fixed-point working. This is done using the output registers

as this saves a reassignment here.

```

10499 \cs_new_protected_nopar:Npn \fp_ln_fixed:
10500 {
10501   \if_int_compare:w \l_fp_output_exponent_int < \c_zero
10502     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
10503     \exp_after:wN \use_i:nn \exp_after:wN
10504     \fp_ln_fixed_aux:NNNNNNNNN
10505     \int_use:N \l_fp_output_decimal_int
10506     \exp_after:wN \fp_ln_fixed:
10507   \fi:
10508 }
10509 \cs_new_protected_nopar:Npn \fp_ln_fixed_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9
10510 {
10511   \if_int_compare:w \l_fp_output_integer_int = \c_zero
10512     \l_fp_output_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
10513   \else:
10514     \tl_set:Nx \l_fp_tmp_tl
10515     {
10516       \int_use:N \l_fp_output_integer_int
10517       #1#2#3#4#5#6#7#8
10518     }
10519     \l_fp_output_integer_int \c_zero
10520     \l_fp_output_decimal_int \l_fp_tmp_tl \scan_stop:
10521   \fi:
10522   \tex_advance:D \l_fp_output_exponent_int \c_one
10523 }

```

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.

```

10524 \cs_new_protected_nopar:Npn \fp_ln_Taylor_aux:
10525 {
10526   \tex_advance:D \l_fp_count_int \c_two
10527   \fp_mul:NNNNNN
10528   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10529   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10530   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10531   \if_int_compare:w
10532     \int_eval:w
10533     \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10534     > \c_zero
10535   \fp_div_integer:NNNNN
10536   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10537   \l_fp_count_int
10538   \l_fp_exp_decimal_int \l_fp_exp_extended_int
10539   \tex_advance:D \l_fp_output_decimal_int \l_fp_exp_decimal_int
10540   \tex_advance:D \l_fp_output_extended_int \l_fp_exp_extended_int
10541   \if_int_compare:w \l_fp_output_extended_int < \c_one_thousand_million

```

```

10542         \else:
10543             \tex_advance:D \l_fp_output_decimal_int \c_one
10544             \tex_advance:D \l_fp_output_extended_int
10545                 -\c_one_thousand_million
10546         \fi:
10547         \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
10548         \else:
10549             \tex_advance:D \l_fp_output_integer_int \c_one
10550             \tex_advance:D \l_fp_output_decimal_int
10551                 -\c_one_thousand_million
10552         \fi:
10553         \exp_after:wN \fp_ln_Taylor_aux:
10554     \fi:
10555 }

```

(End definition for `\fp_ln:Nn` and `\fp_ln:cn`. These functions are documented on page 187.)

`\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:
10556 \cs_new_protected_nopar:Npn \fp_pow:Nn { \fp_pow_aux:NNn \tl_set:Nn }
10557 \cs_new_protected_nopar:Npn \fp_gpow:Nn { \fp_pow_aux:NNn \tl_gset:Nn }
10558 \cs_generate_variant:Nn \fp_pow:Nn { c }
10559 \cs_generate_variant:Nn \fp_gpow:Nn { c }
10560 \cs_new_protected_nopar:Npn \fp_pow_aux:NNn #1#2#3
10561 {
10562     \group_begin:
10563     \fp_read:N #2
10564     \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10565     \l_fp_input_b_integer_int   \l_fp_input_a_integer_int
10566     \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
10567     \l_fp_input_b_exponent_int  \l_fp_input_a_exponent_int
10568     \fp_split:Nn a {#3}
10569     \fp_standardise:NNNN
10570     \l_fp_input_a_sign_int
10571     \l_fp_input_a_integer_int
10572     \l_fp_input_a_decimal_int
10573     \l_fp_input_a_exponent_int
10574     \if_int_compare:w
10575     \int_eval:w
10576     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10577     = \c_zero
10578     \if_int_compare:w
10579     \int_eval:w
10580     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10581     = \c_zero
10582     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10583     {

```

```

10584         \group_end:
10585         ##1 ##2 { \c_undefined_fp }
10586     }
10587 \else:
10588     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10589     {
10590         \group_end:
10591         ##1 ##2 { \c_zero_fp }
10592     }
10593 \fi:
10594 \else:
10595     \if_int_compare:w
10596     \int_eval:w
10597     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10598     = \c_zero
10599     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10600     {
10601         \group_end:
10602         ##1 ##2 { \c_one_fp }
10603     }
10604 \else:
10605     \exp_after:wN \exp_after:wN \exp_after:wN
10606     \fp_pow_aux_i:
10607 \fi:
10608 \fi:
10609 \fp_tmp:w #1 #2
10610 }

```

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.

```

10611 \cs_new_protected_nopar:Npn \fp_pow_aux_i:
10612 {
10613     \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
10614     \tl_set:Nn \l_fp_sign_tl { + }
10615     \exp_after:wN \fp_pow_aux_ii:
10616 \else:
10617     \l_fp_input_a_extended_int \c_zero
10618     \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
10619     \group_begin:
10620     \fp_extended_normalise:
10621     \if_int_compare:w
10622     \int_eval:w
10623     \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10624     = \c_zero
10625     \group_end:
10626     \tl_set:Nn \l_fp_sign_tl { - }
10627     \exp_after:wN \exp_after:wN \exp_after:wN
10628     \exp_after:wN \exp_after:wN \exp_after:wN

```

```

10629         \exp_after:wN \fp_pow_aux_ii:
10630     \else:
10631         \group_end:
10632         \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10633         {
10634             \group_end:
10635             ##1 ##2 { \c_undefined_fp }
10636         }
10637     \fi:
10638 \else:
10639     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10640     {
10641         \group_end:
10642         ##1 ##2 { \c_undefined_fp }
10643     }
10644 \fi:
10645 \fi:
10646 }

```

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.

```

10647 \cs_new_protected_nopar:Npn \fp_pow_aux_ii:
10648 {
10649     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10650     \exp_after:wN \fp_pow_aux_iv:
10651 \else:
10652     \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
10653     \group_begin:
10654     \l_fp_input_a_extended_int \c_zero
10655     \fp_extended_normalise:
10656     \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
10657     \if_int_compare:w \l_fp_input_a_integer_int > \c_ten
10658     \group_end:
10659     \exp_after:wN \exp_after:wN \exp_after:wN
10660     \exp_after:wN \exp_after:wN \exp_after:wN
10661     \exp_after:wN \exp_after:wN \exp_after:wN
10662     \exp_after:wN \exp_after:wN \exp_after:wN
10663     \exp_after:wN \exp_after:wN \exp_after:wN
10664     \fp_pow_aux_iv:
10665 \else:
10666     \group_end:
10667     \exp_after:wN \exp_after:wN \exp_after:wN
10668     \exp_after:wN \exp_after:wN \exp_after:wN
10669     \exp_after:wN \exp_after:wN \exp_after:wN
10670     \exp_after:wN \exp_after:wN \exp_after:wN
10671     \exp_after:wN \exp_after:wN \exp_after:wN

```

```

10672         \exp_after:wN \fp_pow_aux_iii:
10673     \fi:
10674 \else:
10675     \group_end:
10676     \exp_after:wN \exp_after:wN \exp_after:wN
10677     \exp_after:wN \exp_after:wN \exp_after:wN
10678     \exp_after:wN \fp_pow_aux_iv:
10679 \fi:
10680 \else:
10681     \exp_after:wN \exp_after:wN \exp_after:wN
10682     \fp_pow_aux_iv:
10683 \fi:
10684 \fi:
10685 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10686 {
10687     \group_end:
10688     ##1 ##2
10689     {
10690         \l_fp_sign_tl
10691         \int_use:N \l_fp_output_integer_int
10692         .
10693         \exp_after:wN \use_none:n
10694         \int_value:w \int_eval:w
10695         \l_fp_output_decimal_int + \c_one_thousand_million
10696     e
10697     \int_use:N \l_fp_output_exponent_int
10698 }
10699 }
10700 }

```

For the small negative integer powers, the calculation is done for the positive power and the reciprocal is then taken.

```

10701 \cs_new_protected_nopar:Npn \fp_pow_aux_iii:
10702 {
10703     \l_fp_input_a_sign_int \c_one
10704     \fp_pow_aux_iv:
10705     \l_fp_input_a_integer_int \c_one
10706     \l_fp_input_a_decimal_int \c_zero
10707     \l_fp_input_a_exponent_int \c_zero
10708     \l_fp_input_b_integer_int \l_fp_output_integer_int
10709     \l_fp_input_b_decimal_int \l_fp_output_decimal_int
10710     \l_fp_input_b_exponent_int \l_fp_output_exponent_int
10711     \fp_div_internal:
10712 }

```

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.

```

10713 \cs_new_protected_nopar:Npn \fp_pow_aux_iv:
10714 {
10715   \group_begin:
10716     \l_fp_input_a_integer_int \l_fp_input_b_integer_int
10717     \l_fp_input_a_decimal_int \l_fp_input_b_decimal_int
10718     \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10719     \fp_ln_internal:
10720     \cs_set_protected_nopar:Npx \fp_tmp:w
10721     {
10722       \group_end:
10723       \exp_not:N \l_fp_input_b_sign_int
10724       \int_use:N \l_fp_output_sign_int \scan_stop:
10725       \exp_not:N \l_fp_input_b_integer_int
10726       \int_use:N \l_fp_output_integer_int \scan_stop:
10727       \exp_not:N \l_fp_input_b_decimal_int
10728       \int_use:N \l_fp_output_decimal_int \scan_stop:
10729       \exp_not:N \l_fp_input_b_extended_int
10730       \int_use:N \l_fp_output_extended_int \scan_stop:
10731       \exp_not:N \l_fp_input_b_exponent_int
10732       \int_use:N \l_fp_output_exponent_int \scan_stop:
10733     }
10734     \fp_tmp:w
10735     \l_fp_input_a_extended_int \c_zero
10736     \fp_mul:NNNNNNNNN
10737     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10738     \l_fp_input_a_extended_int
10739     \l_fp_input_b_integer_int \l_fp_input_b_decimal_int
10740     \l_fp_input_b_extended_int
10741     \l_fp_output_integer_int \l_fp_output_decimal_int
10742     \l_fp_output_extended_int
10743     \l_fp_output_exponent_int
10744     \int_eval:w
10745     \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
10746     \scan_stop:
10747     \fp_extended_normalise_output:
10748     \tex_multiply:D \l_fp_input_a_sign_int \l_fp_input_b_sign_int
10749     \l_fp_input_a_integer_int \l_fp_output_integer_int
10750     \l_fp_input_a_decimal_int \l_fp_output_decimal_int
10751     \l_fp_input_a_extended_int \l_fp_output_extended_int
10752     \l_fp_input_a_exponent_int \l_fp_output_exponent_int
10753     \l_fp_output_integer_int \c_zero
10754     \l_fp_output_decimal_int \c_zero
10755     \l_fp_output_extended_int \c_zero
10756     \l_fp_output_exponent_int \c_zero
10757     \cs_set_eq:NN \fp_exp_const:Nx \use_none:nn
10758     \fp_exp_internal:

```

```
10759 }
```

(End definition for `\fp_pow:Nn` and `\fp_pow:cn`. These functions are documented on page 186.)

183.13 Tests for special values

`\fp_if_undefined_p:N` Testing for an undefined value is easy.

`\fp_if_undefined:NTF`

```
10760 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { p , T , F , TF }
10761 {
10762   \if_meaning:w #1 \c_undefined_fp
10763   \prg_return_true:
10764   \else:
10765     \prg_return_false:
10766   \fi:
10767 }
```

(End definition for `\fp_if_undefined:N`. These functions are documented on page 183.)

`\fp_if_zero_p:N` Testing for a zero fixed-point is also easy.

`\fp_if_zero:NTF`

```
10768 \prg_new_conditional:Npnn \fp_if_zero:N #1 { p , T , F , TF }
10769 {
10770   \if_meaning:w #1 \c_zero_fp
10771   \prg_return_true:
10772   \else:
10773     \prg_return_false:
10774   \fi:
10775 }
```

(End definition for `\fp_if_zero:N`. These functions are documented on page 183.)

183.14 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.

```
10776 \prg_new_protected_conditional:Npnn \fp_compare:nNn #1#2#3 { T , F , TF }
10777 {
10778   \group_begin:
10779     \fp_split:Nn a {#1}
10780     \fp_standardise:NNNN
10781     \l_fp_input_a_sign_int
10782     \l_fp_input_a_integer_int
10783     \l_fp_input_a_decimal_int
10784     \l_fp_input_a_exponent_int
10785     \fp_split:Nn b {#3}
```

```

10786     \fp_standardise:NNNN
10787     \l_fp_input_b_sign_int
10788     \l_fp_input_b_integer_int
10789     \l_fp_input_b_decimal_int
10790     \l_fp_input_b_exponent_int
10791     \fp_compare_aux:N #2
10792 }
10793 \prg_new_protected_conditional:Npnn \fp_compare:NNN #1#2#3 { T , F , TF }
10794 {
10795     \group_begin:
10796     \fp_read:N #3
10797     \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10798     \l_fp_input_b_integer_int   \l_fp_input_a_integer_int
10799     \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
10800     \l_fp_input_b_exponent_int  \l_fp_input_a_exponent_int
10801     \fp_read:N #1
10802     \fp_compare_aux:N #2
10803 }
10804 \cs_new_protected_nopar:Npn \fp_compare_aux:N #1
10805 {
10806     \cs_if_exist:cTF { fp_compare_#1: }
10807     { \use:c { fp_compare_#1: } }
10808     {
10809         \group_end:
10810         \prg_return_false:
10811     }
10812 }

```

For equality, the test is pretty easy as things are either equal or they are not.

```

10813 \cs_new_protected_nopar:cpn { fp_compare_=: }
10814 {
10815     \if_int_compare:w \l_fp_input_a_sign_int = \l_fp_input_b_sign_int
10816     \if_int_compare:w \l_fp_input_a_integer_int = \l_fp_input_b_integer_int
10817     \if_int_compare:w \l_fp_input_a_decimal_int = \l_fp_input_b_decimal_int
10818     \if_int_compare:w
10819         \l_fp_input_a_exponent_int = \l_fp_input_b_exponent_int
10820     \group_end:
10821     \prg_return_true:
10822     \else:
10823         \group_end:
10824         \prg_return_false:
10825     \fi:
10826     \else:
10827         \group_end:
10828         \prg_return_false:
10829     \fi:
10830     \else:
10831         \group_end:
10832         \prg_return_false:

```

```

10833     \fi:
10834   \else:
10835     \group_end:
10836     \prg_return_false:
10837   \fi:
10838 }

```

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.

```

10839 \cs_new_protected_nopar:cpn { fp_compare_>: }
10840 {
10841   \if_int_compare:w \int_eval:w
10842     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10843     = \c_zero
10844     \if_int_compare:w \int_eval:w
10845       \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10846       = \c_zero
10847     \group_end:
10848     \prg_return_false:
10849   \else:
10850     \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
10851     \group_end:
10852     \prg_return_false:
10853   \else:
10854     \group_end:
10855     \prg_return_true:
10856   \fi:
10857 \fi:
10858 \else:
10859   \if_int_compare:w \int_eval:w
10860     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10861     = \c_zero
10862     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10863     \group_end:
10864     \prg_return_true:
10865   \else:
10866     \group_end:
10867     \prg_return_false:
10868   \fi:
10869 \else:
10870   \use:c { fp_compare_>_aux: }
10871 \fi:
10872 \fi:
10873 }

```

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.

```

10874 \cs_new_protected_nopar:cpn { fp_compare_>_aux: }

```

```

10875 {
10876   \if_int_compare:w \l_fp_input_a_sign_int > \l_fp_input_b_sign_int
10877   \group_end:
10878   \prg_return_true:
10879 \else:
10880   \if_int_compare:w \l_fp_input_a_sign_int < \l_fp_input_b_sign_int
10881   \group_end:
10882   \prg_return_false:
10883 \else:
10884   \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10885   \use:c { fp_compare_absolute_a>b: }
10886   \else:
10887   \use:c { fp_compare_absolute_a<b: }
10888   \fi:
10889 \fi:
10890 \fi:
10891 }

```

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.

```

10892 \cs_new_protected_nopar:cpn { fp_compare_absolute_a>b: }
10893 {
10894   \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
10895   \group_end:
10896   \prg_return_true:
10897 \else:
10898   \if_int_compare:w \l_fp_input_a_exponent_int < \l_fp_input_b_exponent_int
10899   \group_end:
10900   \prg_return_false:
10901 \else:
10902   \if_int_compare:w \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
10903   \group_end:
10904   \prg_return_true:
10905 \else:
10906   \if_int_compare:w
10907   \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
10908   \group_end:
10909   \prg_return_false:
10910 \else:
10911   \if_int_compare:w
10912   \l_fp_input_a_decimal_int > \l_fp_input_b_decimal_int
10913   \group_end:
10914   \prg_return_true:
10915   \else:
10916   \group_end:
10917   \prg_return_false:
10918   \fi:
10919 \fi:

```

```

10920         \fi:
10921     \fi:
10922 \fi:
10923 }
10924 \cs_new_protected_nopar:cpn { fp_compare_absolute_a<b: }
10925 {
10926     \if_int_compare:w \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
10927     \group_end:
10928     \prg_return_true:
10929 \else:
10930     \if_int_compare:w \l_fp_input_b_exponent_int < \l_fp_input_a_exponent_int
10931     \group_end:
10932     \prg_return_false:
10933 \else:
10934     \if_int_compare:w \l_fp_input_b_integer_int > \l_fp_input_a_integer_int
10935     \group_end:
10936     \prg_return_true:
10937 \else:
10938     \if_int_compare:w
10939         \l_fp_input_b_integer_int < \l_fp_input_a_integer_int
10940     \group_end:
10941     \prg_return_false:
10942 \else:
10943     \if_int_compare:w
10944         \l_fp_input_b_decimal_int > \l_fp_input_a_decimal_int
10945     \group_end:
10946     \prg_return_true:
10947 \else:
10948     \group_end:
10949     \prg_return_false:
10950 \fi:
10951 \fi:
10952 \fi:
10953 \fi:
10954 \fi:
10955 }

```

This is just a case of reversing the two input values and then running the tests already defined.

```

10956 \cs_new_protected_nopar:cpn { fp_compare_<: }
10957 {
10958     \tl_set:Nx \l_fp_tmp_tl
10959     {
10960         \int_set:Nn \exp_not:N \l_fp_input_a_sign_int
10961         { \int_use:N \l_fp_input_b_sign_int }
10962         \int_set:Nn \exp_not:N \l_fp_input_a_integer_int
10963         { \int_use:N \l_fp_input_b_integer_int }
10964         \int_set:Nn \exp_not:N \l_fp_input_a_decimal_int
10965         { \int_use:N \l_fp_input_b_decimal_int }

```

```

10966 \int_set:Nn \exp_not:N \l_fp_input_a_exponent_int
10967 { \int_use:N \l_fp_input_b_exponent_int }
10968 \int_set:Nn \exp_not:N \l_fp_input_b_sign_int
10969 { \int_use:N \l_fp_input_a_sign_int }
10970 \int_set:Nn \exp_not:N \l_fp_input_b_integer_int
10971 { \int_use:N \l_fp_input_a_integer_int }
10972 \int_set:Nn \exp_not:N \l_fp_input_b_decimal_int
10973 { \int_use:N \l_fp_input_a_decimal_int }
10974 \int_set:Nn \exp_not:N \l_fp_input_b_exponent_int
10975 { \int_use:N \l_fp_input_a_exponent_int }
10976 }
10977 \l_fp_tmp_tl
10978 \use:c { fp_compare_>: }
10979 }

```

(End definition for \fp_compare:nNn. This function is documented on page ??.)

\fp_compare:nTF

As T_EX cannot help out here, a daisy-chain of delimited functions are used. This is very much a first-generation approach: revision will be needed if these functions are really useful.

```

\fp_compare_aux_i:w
\fp_compare_aux_ii:w
\fp_compare_aux_iii:w
\fp_compare_aux_iv:w
\fp_compare_aux_v:w
\fp_compare_aux_vi:w
\fp_compare_aux_vii:w
10980 \prg_new_protected_conditional:Npnn \fp_compare:n #1 { T , F , TF }
10981 {
10982   \group_begin:
10983     \tl_set:Nx \l_fp_tmp_tl
10984     {
10985       \group_end:
10986       \fp_compare_aux_i:w #1 \exp_not:n { == \q_nil == \q_stop }
10987     }
10988     \l_fp_tmp_tl
10989   }
10990 \cs_new_protected_nopar:Npn \fp_compare_aux_i:w #1 == #2 == #3 \q_stop
10991 {
10992   \quark_if_nil:nTF {#2}
10993   { \fp_compare_aux_ii:w #1 != \q_nil != \q_stop }
10994   { \fp_compare:nNnTF {#1} = {#2} \prg_return_true: \prg_return_false: }
10995 }
10996 \cs_new_protected_nopar:Npn \fp_compare_aux_ii:w #1 != #2 != #3 \q_stop
10997 {
10998   \quark_if_nil:nTF {#2}
10999   { \fp_compare_aux_iii:w #1 <= \q_nil <= \q_stop }
11000   { \fp_compare:nNnTF {#1} = {#2} \prg_return_false: \prg_return_true: }
11001 }
11002 \cs_new_protected_nopar:Npn \fp_compare_aux_iii:w #1 <= #2 <= #3 \q_stop
11003 {
11004   \quark_if_nil:nTF {#2}
11005   { \fp_compare_aux_iv:w #1 >= \q_nil >= \q_stop }
11006   { \fp_compare:nNnTF {#1} > {#2} \prg_return_false: \prg_return_true: }
11007 }
11008 \cs_new_protected_nopar:Npn \fp_compare_aux_iv:w #1 >= #2 >= #3 \q_stop

```

```

11009 {
11010   \quark_if_nil:nTF {#2}
11011   { \fp_compare_aux_v:w #1 = \q_nil \q_stop }
11012   { \fp_compare:nNnTF {#1} < {#2} \prg_return_false: \prg_return_true: }
11013 }
11014 \cs_new_protected_nopar:Npn \fp_compare_aux_v:w #1 = #2 = #3 \q_stop
11015 {
11016   \quark_if_nil:nTF {#2}
11017   { \fp_compare_aux_vi:w #1 < \q_nil < \q_stop }
11018   { \fp_compare:nNnTF {#1} = {#2} \prg_return_true: \prg_return_false: }
11019 }
11020 \cs_new_protected_nopar:Npn \fp_compare_aux_vi:w #1 < #2 < #3 \q_stop
11021 {
11022   \quark_if_nil:nTF {#2}
11023   { \fp_compare_aux_vii:w #1 > \q_nil > \q_stop }
11024   { \fp_compare:nNnTF {#1} < {#2} \prg_return_true: \prg_return_false: }
11025 }
11026 \cs_new_protected_nopar:Npn \fp_compare_aux_vii:w #1 > #2 > #3 \q_stop
11027 {
11028   \quark_if_nil:nTF {#2}
11029   { \prg_return_false: }
11030   { \fp_compare:nNnTF {#1} > {#2} \prg_return_true: \prg_return_false: }
11031 }

```

(End definition for `\fp_compare:n`. This function is documented on page 184.)

183.15 Messages

`\fp_overflow_msg:` A generic overflow message, used whenever there is a possible overflow.

```

11032 \msg_kernel_new:nnnn { fpu } { overflow }
11033 { Number~too~big. }
11034 {
11035   The~input~given~is~too~big~for~the~LaTeX~floating~point~unit. \
11036   Further~errors~may~well~occur!
11037 }
11038 \cs_new_protected_nopar:Npn \fp_overflow_msg:
11039 { \msg_kernel_error:nn { fpu } { overflow } }

```

(End definition for `\fp_overflow_msg:`. This function is documented on page ??.)

`\fp_exp_overflow_msg:` A slightly more helpful message for exponent overflows.

```

11040 \msg_kernel_new:nnnn { fpu } { exponent-overflow }
11041 { Number~too~big~for~exponent~unit. }
11042 {
11043   The~exponent~of~the~input~given~is~too~big~for~the~floating~point~
11044   unit:~the~maximum~input~value~for~an~exponent~is~230.
11045 }

```



```

11046 \cs_new_protected_nopar:Npn \fp_exp_overflow_msg:
11047 { \msg_kernel_error:nn { fpu } { exponent-overflow } }

(End definition for \fp_exp_overflow_msg:.. This function is documented on page ??.)

```

`\fp_ln_error_msg:` Logarithms are only valid for positive number

```

11048 \msg_kernel_new:nnnn { fpu } { logarithm-input-error }
11049 { Invalid~input~to~ln~function. }
11050 { Logarithms~can~only~be~calculated~for~positive~numbers. }
11051 \cs_new_protected_nopar:Npn \fp_ln_error_msg: {
11052 \msg_kernel_error:nn { fpu } { logarithm-input-error }
11053 }

(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.

```

11054 \msg_kernel_new:nnnn { fpu } { trigonometric-overflow }
11055 { Number~too~big~for~trigonometry~unit. }
11056 {
11057   The~trigonometry~code~can~only~work~with~numbers~smaller~
11058   than~1000000000.
11059 }
11060 \cs_new_protected_nopar:Npn \fp_trig_overflow_msg:
11061 { \msg_kernel_error:nn { fpu } { trigonometric-overflow } }

(End definition for \fp_trig_overflow_msg:.. This function is documented on page ??.)

11062 </initex | package>

```

184 l3luatex implementation

```

11063 <*initex | package>

```

Announce and ensure that the required packages are loaded.

```

11064 <*package>
11065 \ProvidesExplPackage
11066 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
11067 \package_check_loaded_expl:
11068 </package>
11069 <*initex | package>

```

`\lua_now:n` When LuaTeX is in use, this is all a question of primitives with new names. On the other
`\lua_now:x` hand, for pdfTeX and XeTeX the argument should be removed from the input stream
`\lua_shipout:x:n` before issuing an error. This needs to be expandable, so the same idea is used as for
`\lua_shipout:x:x` V-type expansion, with an appropriately-named but undefined function.
`\lua_shipout:n`
`\lua_shipout:x` 11070 `\luatex_if_engine:TF`
`\lua_wrong_engine:`

```

11071 {
11072   \cs_new_eq:NN \lua_now:x      \luatex_directlua:D
11073   \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
11074 }
11075 {
11076   \cs_new:Npn \lua_now:x #1 { \lua_wrong_engine: }
11077   \cs_new_protected:Npn \lua_shipout_x:n #1 { \lua_wrong_engine: }
11078 }
11079 \cs_new:Npn \lua_now:n #1
11080 { \lua_now:x { \exp_not:n {#1} } }
11081 \cs_generate_variant:Nn \lua_shipout_x:n { x }
11082 \cs_new_protected:Npn \lua_shipout:n #1
11083 { \lua_shipout_x:n { \exp_not:n {#1} } }
11084 \cs_generate_variant:Nn \lua_shipout:n { x }
11085 \group_begin:
11086 \char_set_catcode_letter:N\!
11087 \char_set_catcode_letter:N\%
11088 \cs_gset:Npn \lua_wrong_engine:{%
11089 \LuaTeX engine not in use!%
11090 }%
11091 \group_end:%

```

(End definition for `\lua_now:n` and `\lua_now:x`. These functions are documented on page 190.)

184.1 Category code tables

`\g_cctab_allocate_int` To allocate category code tables, both the read-only and stack tables need to be followed.
`\g_cctab_stack_int` There is also a sequence stack for the dynamic tables themselves.
`\g_cctab_stack_seq`

```

11092 \int_new:N \g_cctab_allocate_int
11093 \int_set:Nn \g_cctab_allocate_int { -1 }
11094 \int_new:N \g_cctab_stack_int
11095 \seq_new:N \g_cctab_stack_seq

```

`\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.

```

11096 \cs_new_protected_nopar:Npn \cctab_new:N #1
11097 {
11098   \cs_if_free:NTF #1
11099   {
11100     \int_gadd:Nn \g_cctab_allocate_int { 2 }
11101     \int_compare:nNnTF
11102     { \g_cctab_allocate_int } < { \c_max_register_int + 1 }
11103     {
11104       \pref_global:D \tex_mathchardef:D #1 \g_cctab_allocate_int

```

```

11105         \luatex_initcatcodetable:D #1
11106     }
11107     { \msg_kernel_fatal:nxx { alloc } { out-of-registers } { cctab } }
11108 }
11109 {
11110     \msg_kernel_error:nxx { code } { variable-already-defined }
11111     { \token_to_str:N #1 }
11112 }
11113 }
11114 \luatex_if_engine:F
11115 { \cs_set_protected_nopar:Npn \cctab_new:N #1 { \lua_wrong_engine: } }
11116 <*package>
11117 \luatex_if_engine:T
11118 {
11119     \cs_set_protected_nopar:Npn \cctab_new:N #1
11120     {
11121         \newcatcodetable #1
11122         \luatex_initcatcodetable:D #1
11123     }
11124 }
11125 </package>

```

(End definition for `\cctab_new:N`. This function is documented on page 191.)

`\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`

```

11126 \cs_new_protected_nopar:Npn \cctab_begin:N #1
11127 {
11128     \seq_gpush:Nx \g_cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
11129     \luatex_catcodetable:D #1
11130     \int_gadd:Nn \g_cctab_stack_int { 2 }
11131     \int_compare:nNnT { \g_cctab_stack_int } > { 268 435 453 }
11132     { \msg_kernel_error:nn { code } { cctab-stack-full } }
11133     \luatex_savecatcodetable:D \g_cctab_stack_int
11134     \luatex_catcodetable:D \g_cctab_stack_int
11135 }
11136 \cs_new_protected_nopar:Npn \cctab_end:
11137 {
11138     \int_gsub:Nn \g_cctab_stack_int { 2 }
11139     \seq_gpop:NN \g_cctab_stack_seq \l_cctab_tmp_tl
11140     \quark_if_no_value:NT \l_cctab_tmp_tl
11141     { \tl_set:Nn \l_cctab_tmp_tl { 0 } }
11142     \luatex_catcodetable:D \l_cctab_tmp_tl \scan_stop:
11143 }
11144 \luatex_if_engine:F
11145 {
11146     \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \lua_wrong_engine: }
11147     \cs_set_protected_nopar:Npn \cctab_end: { \lua_wrong_engine: }
11148 }

```

```

11149 <*package>
11150 \luatex_if_engine:T
11151 {
11152   \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \BeginCatcodeRegime #1 }
11153   \cs_set_protected_nopar:Npn \cctab_end: { \EndCatcodeRegime }
11154 }
11155 </package>
11156 \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.

```

11157 \cs_new_protected:Npn \cctab_gset:Nn #1#2
11158 {
11159   \group_begin:
11160     #2
11161     \luatex_savecatcodetable:D #1
11162   \group_end:
11163 }
11164 \luatex_if_engine:F
11165 { \cs_set_protected_nopar:Npn \cctab_gset:Nn #1#2 { \lua_wrong_engine: } }

```

(End definition for `\cctab_gset:Nn`. This function is documented on page 191.)

`\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` complex. The table for expl3 category codes is always needed, whereas when in package
`\c_other_cctab` mode the rest can be copied from the existing L^AT_EX 2_ε package `luatex`.
`\c_string_cctab`

```

11166 \luatex_if_engine:T
11167 {
11168   \cctab_new:N \c_code_cctab
11169   \cctab_gset:Nn \c_code_cctab { }
11170 }
11171 <*package>
11172 \luatex_if_engine:T
11173 {
11174   \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
11175   \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
11176   \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
11177   \cs_new_eq:NN \c_string_cctab \CatcodeTableString
11178 }
11179 </package>
11180 <*initex>
11181 \luatex_if_engine:T
11182 {
11183   \cctab_new:N \c_document_cctab

```

```

11184 \cctab_new:N \c_other_cctab
11185 \cctab_new:N \c_string_cctab
11186 \cctab_gset:Nn \c_document_cctab
11187 {
11188   \char_set_catcode_space:n { 9 }
11189   \char_set_catcode_space:n { 32 }
11190   \char_set_catcode_other:n { 58 }
11191   \char_set_catcode_math_subscript:n { 95 }
11192   \char_set_catcode_active:n { 126 }
11193 }
11194 \cctab_gset:Nn \c_other_cctab
11195 {
11196   \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
11197   { \char_set_catcode_other:n {#1} }
11198 }
11199 \cctab_gset:Nn \c_string_cctab
11200 {
11201   \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
11202   { \char_set_catcode_other:n {#1} }
11203   \char_set_catcode_space:n { 32 }
11204 }
11205 }
11206 </initex>

11207 </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	
\!	1572, 4581, 6635, 11086
\#	5
\	2264, 2265, 2278, 2279, 6635, 6636
*	2585, 2587, 2591, 2598
\,	7057, 7059
\-	348
\.	6578, 6583
\.bool_gset:N	<u>7358</u>
\.bool_set:N	<u>7358</u>
\.choice:	<u>7362</u>
\.choice_code:n	<u>7368</u>
\.choice_code:x	<u>7368</u>
\.code:n	<u>7364</u>
\.code:x	<u>7364</u>
\.default:V	<u>7372</u>
\.default:n	<u>7372</u>
\.dim_gset:N	<u>7376</u>
\.dim_gset:c	<u>7376</u>
\.dim_set:N	<u>7376</u>
\.dim_set:c	<u>7376</u>
\.fp_gset:N	<u>7384</u>
\.fp_gset:c	<u>7384</u>
\.fp_set:N	<u>7384</u>
\.fp_set:c	<u>7384</u>
\.generate_choices:n	<u>7392</u>
\.int_gset:N	<u>7394</u>
\.int_gset:c	<u>7394</u>
\.int_set:N	<u>7394</u>
\.int_set:c	<u>7394</u>
\.meta:n	<u>7402</u>
\.meta:x	<u>7402</u>
\.skip_gset:N	<u>7406</u>
\.skip_gset:c	<u>7406</u>
\.skip_set:N	<u>7406</u>
\.skip_set:c	<u>7406</u>
\.tl_gset:N	<u>7414</u>
\.tl_gset:c	<u>7414</u>
\.tl_gset_x:N	<u>7414</u>
\.tl_gset_x:c	<u>7414</u>
\.tl_set:N	<u>7414</u>
\.tl_set:c	<u>7414</u>
\.tl_set_x:N	<u>7414</u>
\.tl_set_x:c	<u>7414</u>
\.value_forbidden:	<u>7430</u>
\.value_required:	<u>7430</u>
\/	347
\:	1073, 2677, 2810
\:::	36,
	1525, 1526, <u>1527</u> , 1527–1530, 1532,
	1534, 1541, 1546, 1552, 1672–1698,
	1700, 1705, 1707, 1712, 1738, 1739
\::N	36, <u>1529</u> ,
	1529, 1681, 1687, 1688, 1692, 1693
\::V	36, <u>1546</u> , 1546, 1678
\::V_unbraced	<u>1699</u> , 1707
\::c	36, <u>1530</u> , 1530,
	1673, 1679, 1682, 1689, 1696, 1697
\::f	36, <u>1534</u> , 1534, 1674–1676, 1739
\::f_unbraced	<u>1699</u> , 1700
\::n	36, <u>1528</u> ,
	1528, 1673, 1676–1678, 1683, 1687,
	1689, 1690, 1692, 1694, 1695, 1697
\::o	36, <u>1532</u> , 1532, 1674, 1677,
	1679, 1680, 1684, 1685, 1687, 1688,
	1690, 1691, 1693, 1695, 1698, 1738
\::o_unbraced	<u>1699</u> , 1705, 1738, 1739
\::v	36, <u>1546</u> , 1552
\::v_unbraced	<u>1699</u> , 1712
\::x	36, <u>1541</u> ,
	1541, 1672, 1681–1686, 1692–1698
\;	2677, 2809, 2810
\=	7056, 7058
\?	1813, 2717
\@	1073, 1074, 4285, 4286
\@end	763
\@hyph	766
\@input	767
\@italiccorr	768
\@underline	769
\@addtofilelist	7679
\@currname	7611
\@filelist	7704
\@ifpackageloaded	216
\@l@expl@log@functions@bool	1201
\@namedef	198

\@nil		193, 201	\batchmode		438
\@popfilename		163, 181, 183	\begin		6598
\@pushfilename		163, 164, 179	\BeginCatcodeRegime		11152
\@tempa		54, 56, 64	\begingroup		12, 61, 100, 228, 232, 338, 376
\@tempboxa		6022	\beginL		730
\[3741, 6578	\beginR		732
\\		1813, 2717,	\belowdisplaysshortskip		482
		6336, 6343, 6382, 6391, 6439, 6446,	\belowdisplayskip		483
		6540, 6558, 6560, 6565, 6566, 6602,	\binoppenalty		506
		6604, 6610, 6612, 6676, 6752, 6760,	\bool_(:w		1911
		6907, 6929, 6949, 6957, 6964, 6971,	\bool_)_0:w		1911
		6980, 6988, 6989, 7026, 7538, 7553,	\bool_)_1:w		1911
		7554, 7560, 7573, 7586, 7592, 11035	\bool_8_0:w		1911
\{		3, 5084, 5547, 5840, 5842	\bool_8_1:w		1911
\}		4, 5084, 5547, 5840, 5842, 6634	\bool_:w		1911
\^		6, 9, 249, 3072, 3079	\bool_choose:Nn		1911, 2000, 2008
_		2278	\bool_cleanup:N		1911, 1993, 1996, 1998
		3847	\bool_do_until:cn		2056
			\bool_do_until:Nn		2056, 2058, 2059, 2061
			\bool_do_until:nm		2062, 2083, 2086
			\bool_do_while:cn		2056
			\bool_do_while:Nn		2056, 2056, 2057, 2060
			\bool_do_while:nm		2062, 2070, 2073
			\bool_eval_skip_to_end:Nw		1911,
					2020, 2021, 2023, 2025, 2027, 2041
_		346, 1065,	\bool_eval_skip_to_end_aux:Nw		
		1573, 4582, 6383, 6634, 6990, 11087			1911, 2029, 2033
			\bool_eval_skip_to_end_aux_ii:Nw		
					1911, 2037, 2039
A			\bool_get_next:N		1911, 1922, 1924,
\A		2676, 2718, 4285, 4287			1944, 1973, 1993, 1995, 2010–2013
\above		467	\bool_get_next:Nn		1944, 1952
\abovedisplaysshortskip		480	\bool_get_not_next:N		1934, 1945, 1972
\abovedisplayskip		481	\bool_get_not_next:Nn		1945, 1965
\abovewithdelims		468	\bool_gset:cn		41, 1891
\accent		518	.bool_gset:N		171
\adjdemerits		555	\bool_gset:Nn		41, 1891, 1893, 1896
\advance		362	\bool_gset_eq:cc		41, 1883, 1890
\afterassignment		372	\bool_gset_eq:cN		41, 1883, 1889
\aftergroup		373	\bool_gset_eq:Nc		41, 1883, 1888
\alloc_new:nnnN		3247, 3879, 4041, 4118, 5917	\bool_gset_eq:NN		41, 1883, 1887
\alloc_reg:NnNN		6140, 6143	\bool_gset_false:c		40, 1871
\alloc_setup_type:nnm		6138, 6141	\bool_gset_false:N		40, 1871, 1877, 1882
\AtBeginDocument		7702	\bool_gset_true:c		40, 1871
\atop		469	\bool_gset_true:N		40, 1871, 1875, 1881
\atopwithdelims		470	\bool_I_0:w		1911
			\bool_I_1:w		1911
B			\bool_if:cTF		41, 1897
\B		4286, 4288	\bool_if:N		1897
\badness		617	\bool_if:n		1911
\baselineskip		545			

\bool_if:NF ... 294, 1907, 2053, 2059, 3678
\bool_if:nF 2077, 2086
\bool_if:NT 1906,
2051, 2057, 3678, 3679, 7484, 8550
\bool_if:nT 2064, 2073
\bool_if:NTF
41, 1897, 1908, 3664, 6425, 7236, 8563
\bool_if:nTF 42, 1911, 2942, 4088, 7459, 7469
\bool_if_p:c 41, 1897
\bool_if_p:N 41, 1897, 1905
\bool_if_p:n 42, 1892,
1894, 1911, 1913, 1919, 2043, 2046
\bool_new:c 40, 1869
\bool_new:N 40, 1869, 1869, 1870,
1909, 1910, 6373, 7185, 7253, 7773
\bool_Not:N 1954, 1974
\bool_Not:w 1911, 1949, 1972
\bool_not_choose:NN 2005, 2009
\bool_not_cleanup:N 1995, 1997, 2003
\bool_not_Not:N 1967, 1983
\bool_not_Not:w 1962, 1973
\bool_not_p:n 43, 2043, 2043
\bool_p:w 1911, 1976, 1985
\bool_S_0:w 1911
\bool_S_1:w 1911
\bool_set:cn 41, 1891
\bool_set:N 171
\bool_set:Nn 41, 1891, 1891, 1895
\bool_set_eq:cc 40, 1883, 1886
\bool_set_eq:cN 40, 1883, 1885
\bool_set_eq:Nc 40, 1883, 1884
\bool_set_eq:NN 40, 1883, 1883
\bool_set_false:c 40, 1871
\bool_set_false:N 40, 309, 1871, 1873,
1880, 6427, 7205, 7451, 8540, 8569
\bool_set_true:c 40, 1871
\bool_set_true:N . 40, 323, 1871, 1871,
1879, 6381, 6449, 7200, 7446, 8577
\bool_until_do:cn 43, 2050
\bool_until_do:Nn 43, 2050, 2052, 2053, 2055
\bool_until_do:nn .. 43, 2062, 2075, 2080
\bool_while_do:cn 43, 2050
\bool_while_do:Nn 43, 2050, 2050, 2051, 2054
\bool_while_do:nn .. 43, 2062, 2062, 2067
\bool_xor_p:nn 43, 2044, 2044
\botmark 453
\botmarks 679
\box 661
\box_clear:c 143, 5928
\box_clear:N 143, 5928, 5928, 5932
\box_clear_new:c 143, 5934
\box_clear_new:N .. 143, 5934, 5934, 5946
\box_dp:c 145, 5960
\box_dp:N 145, 5960, 5961, 5964, 5967
\box_gclear:c 143, 5928
\box_gclear:N 143, 5928, 5930, 5933
\box_gclear_new:c 143, 5934
\box_gclear_new:N . 143, 5934, 5940, 5947
\box_gset_eq:cc 144, 5948
\box_gset_eq:cN 144, 5948
\box_gset_eq:Nc 144, 5948
\box_gset_eq:NN
... 144, 5931, 5943, 5948, 5950, 5953
\box_gset_eq_clear:cc 144, 5954
\box_gset_eq_clear:cN 144, 5954
\box_gset_eq_clear:Nc 144, 5954
\box_gset_eq_clear:NN 144, 5954, 5956, 5959
\box_gset_to_last:c 6009
\box_gset_to_last:N 6009, 6011, 6014
\box_ht:c 145, 5960
\box_ht:N 145, 5960, 5960, 5963, 5969
\box_if_empty:cTF 146, 6002
\box_if_empty:N 6002
\box_if_empty:NF 6006
\box_if_empty:NT 6005
\box_if_empty:NTF 146, 6002, 6007
\box_if_empty_p:c 146, 6002
\box_if_empty_p:N 146, 6002, 6004
\box_if_horizontal:cTF 146, 5990
\box_if_horizontal:N 5990
\box_if_horizontal:NF 5996
\box_if_horizontal:NT 5995
\box_if_horizontal:NTF . 146, 5990, 5997
\box_if_horizontal_p:c 146, 5990
\box_if_horizontal_p:N . 146, 5990, 5994
\box_if_vertical:cTF 146, 5990
\box_if_vertical:N 5992
\box_if_vertical:NF 6000
\box_if_vertical:NT 5999
\box_if_vertical:NTF ... 146, 5990, 6001
\box_if_vertical_p:c 146, 5990
\box_if_vertical_p:N ... 146, 5990, 5998
\box_move_down:nn 145, 5979, 5985
\box_move_left:nn 145, 5979, 5979
\box_move_right:nn 145, 5979, 5981
\box_move_up:nn 145, 5979, 5983
\box_new:c 143, 5916
\box_new:N 143, 5916, 5921,
5927, 5938, 5944, 6019, 6025, 6027
\box_set_dp:cn 146, 5966

- \box_set_dp:Nn 146, 5966, 5966, 5973
 - \box_set_eq:cc 143, 5948
 - \box_set_eq:cN 143, 5948
 - \box_set_eq:Nc 143, 5948
 - \box_set_eq:NN 143, 5929, 5937, 5948, 5948, 5951, 5952
 - \box_set_eq_clear:cc 144, 5954
 - \box_set_eq_clear:cN 144, 5954
 - \box_set_eq_clear:Nc 144, 5954
 - \box_set_eq_clear:NN 144, 5954, 5954, 5957, 5958
 - \box_set_ht:cn 146, 5966
 - \box_set_ht:Nn 146, 5966, 5968, 5972
 - \box_set_to_last:c 6009
 - \box_set_to_last:N 6009, 6009, 6012, 6013
 - \box_set_wd:cn 146, 5966
 - \box_set_wd:Nn 146, 5966, 5970, 5974
 - \box_show:c 147, 6028
 - \box_show:N 147, 6028, 6028, 6029
 - \box_use:c 144, 5975
 - \box_use:N 144, 5975, 5976, 5978
 - \box_use_clear:c 144, 5975
 - \box_use_clear:N 144, 5975, 5975, 5977
 - \box_wd:c 145, 5960
 - \box_wd:N 145, 5960, 5962, 5965, 5971
 - \boxmaxdepth 623
 - \brokenpenalty 580
- C**
- \C 1819, 2718
 - \c_active_char_token 3116, 3116
 - \c_alignment_tab_token 3112, 3112
 - \c_alignment_token 57, 2582, 2588, 2618, 3112
 - \c_catcode_active_tl 58, 2597, 2599, 2656, 3116
 - \c_catcode_letter_token 57, 2582, 2594, 2646, 3114
 - \c_catcode_other_token 57, 2582, 2595, 2651, 3115
 - \c_code_cctab 191, 11166, 11168, 11169
 - \c_document_cctab 191, 11166, 11174, 11183, 11186
 - \c_e_fp 188, 7732, 7732
 - \c_eight 79, 2510, 2542, 3740, 3803, 3808, 6117, 8541
 - \c_eleven 79, 2516, 2548, 3803, 3811, 6120
 - \c_empty_box 147, 5929, 5931, 5937, 5943, 6015, 6016, 6019
 - \c_empty_prop 142, 5633, 5633–5639, 5752
 - \c_empty_tl 108, 4183, 4199, 4201, 4429, 4679, 4679, 4976, 5090
 - \c_false_bool 23, 992, 1021, 1061, 1062, 1090, 1457, 1459, 1468, 1480, 1869, 1874, 1878, 1978, 1989, 2014, 2017, 2018, 2020, 2023, 2047, 2762, 3653, 3658, 3667
 - \c_fifteen 79, 2524, 2556, 3803, 3814, 6124
 - \c_five 79, 2504, 2536, 3803, 3807, 6114, 8194, 9441, 9739
 - \c_five_hundred_million 7715, 7718, 8231, 10130, 10286, 10481, 10483
 - \c_forty_four 7715, 7715, 8359
 - \c_four 79, 2502, 2534, 3803, 3806, 6113, 8340, 8576
 - \c_fourteen 79, 2522, 2554, 3803, 3813, 6123
 - \c_fp_exp-100_tl 9884
 - \c_fp_exp-10_tl 9884
 - \c_fp_exp-1_tl 9884
 - \c_fp_exp-200_tl 9884
 - \c_fp_exp-20_tl 9884
 - \c_fp_exp-2_tl 9884
 - \c_fp_exp-30_tl 9884
 - \c_fp_exp-3_tl 9884
 - \c_fp_exp-40_tl 9884
 - \c_fp_exp-4_tl 9884
 - \c_fp_exp-50_tl 9884
 - \c_fp_exp-5_tl 9884
 - \c_fp_exp-60_tl 9884
 - \c_fp_exp-6_tl 9884
 - \c_fp_exp-70_tl 9884
 - \c_fp_exp-7_tl 9884
 - \c_fp_exp-80_tl 9884
 - \c_fp_exp-8_tl 9884
 - \c_fp_exp-90_tl 9884
 - \c_fp_exp-9_tl 9884
 - \c_fp_exp_100_tl 9864
 - \c_fp_exp_10_tl 9864
 - \c_fp_exp_1_tl 9864
 - \c_fp_exp_200_tl 9864
 - \c_fp_exp_20_tl 9864
 - \c_fp_exp_2_tl 9864
 - \c_fp_exp_30_tl 9864
 - \c_fp_exp_3_tl 9864
 - \c_fp_exp_40_tl 9864
 - \c_fp_exp_4_tl 9864
 - \c_fp_exp_50_tl 9864
 - \c_fp_exp_5_tl 9864
 - \c_fp_exp_60_tl 9864
 - \c_fp_exp_6_tl 9864

- \c_fp_exp_70_tl 9864
- \c_fp_exp_7_tl 9864
- \c_fp_exp_80_tl 9864
- \c_fp_exp_8_tl 9864
- \c_fp_exp_90_tl 9864
- \c_fp_exp_9_tl 9864
- \c_fp_ln_10_1_tl 10210
- \c_fp_ln_10_2_tl 10210
- \c_fp_ln_10_3_tl 10210
- \c_fp_ln_10_4_tl 10210
- \c_fp_ln_10_5_tl 10210
- \c_fp_ln_10_6_tl 10210
- \c_fp_ln_10_7_tl 10210
- \c_fp_ln_10_8_tl 10210
- \c_fp_ln_10_9_tl 10210
- \c_fp_ln_2_1_tl 10219
- \c_fp_ln_2_2_tl 10219
- \c_fp_ln_2_3_tl 10219
- \c_fp_pi_by_four_decimal_int . 7720,
7720, 7721, 9379, 9392, 9399, 9403
- \c_fp_pi_by_four_extended_int
.. 7720, 7722, 7723, 9379, 9392, 9404
- \c_fp_pi_decimal_int 7720, 7724, 7725, 9319
- \c_fp_pi_extended_int .. 7720, 7726, 7727
- \c_fp_two_pi_decimal_int
..... 7720, 7728, 7729, 9315, 9321
- \c_fp_two_pi_extended_int
..... 7720, 7730, 7731, 9315, 9321
- \c_group_begin_token
57, 2582, 2582, 2603, 2944, 6042, 6082
- \c_group_end_token ... 57, 2582, 2583,
2608, 2945, 6047, 6048, 6087, 6088
- \c_initex_cctab 191, 11166, 11175
- \c_ior_log_stream 6103, 6106
- \c_ior_streams_tl 6107, 6126, 6155
- \c_ior_term_stream 6103, 6104
- \c_iow_log_stream 6103, 6105, 6355, 6356
- \c_iow_streams_tl 6107, 6107, 6126, 6168
- \c_iow_term_stream 6103, 6103, 6357, 6358
- \c_job_name_tl
... 108, 4663, 4664, 4673, 4674, 4676
- \c_keys_code_root_tl 7176, 7176,
7316, 7321, 7512, 7514, 7526, 7531
- \c_keys_props_root_tl 7178,
7178, 7210, 7240, 7247, 7358, 7360,
7362, 7364, 7366, 7368, 7370, 7372,
7374, 7376, 7378, 7380, 7382, 7384,
7386, 7388, 7390, 7392, 7394, 7396,
7398, 7400, 7402, 7404, 7406, 7408,
7410, 7412, 7414, 7416, 7418, 7420,
7422, 7424, 7426, 7428, 7430, 7432
- \c_keys_value_forbidden_tl .. 7179, 7179
- \c_keys_value_required_tl ... 7179, 7180
- \c_keys_vars_root_tl 7176, 7177,
7279, 7298, 7305, 7308, 7310, 7325–
7327, 7330, 7345, 7486, 7489, 7496
- \c_letter_token 3112, 3114
- \c_luatex_is_engine_bool
..... 25, 1448, 1457, 1479
- \c_math_shift_token 3112, 3113
- \c_math_subscript_token
..... 57, 2582, 2592, 2636
- \c_math_superscript_token
..... 57, 2582, 2590, 2631
- \c_math_toggle_token
..... 57, 2582, 2586, 2613, 3113
- \c_max_dim 88, 4026, 4028, 4029, 4033, 4111
- \c_max_int 79, 3821, 3821
- \c_max_register_int
..... 79, 1168, 1168, 3247,
3261, 3879, 4041, 4118, 5917, 11102
- \c_max_skip 91, 4110, 4111
- \c_minus_one 79, 1156,
1157, 1160, 1161, 1170, 3259, 3303,
3803, 4303, 4329, 4341, 6105, 6106,
6273, 6285, 7790, 7898, 8327, 8581,
8582, 8719, 8754, 8997, 9045, 9049,
9214, 9338, 9342, 9597, 9612, 9700,
9704, 9777, 9785, 10193, 10197, 10321
- \c_msg_coding_error_text_tl .. 6537,
6537, 6947, 6956, 6978, 6986, 6995,
7002, 7009, 7544, 7551, 7572, 7579
- \c_msg_continue_text_tl 6537, 6542, 6604
- \c_msg_critical_text_tl 6537, 6544, 6723
- \c_msg_fatal_text_tl 6537, 6546, 6712, 6854
- \c_msg_help_text_tl 6537, 6548, 6612
- \c_msg_hide_tl 6578, 6580–6582, 6649
- \c_msg_kernel_bug_more_text_tl
..... 7013, 7020, 7024
- \c_msg_kernel_bug_text_tl 7013, 7015, 7022
- \c_msg_more_text_prefix_tl ... 6515,
6516, 6532, 6728, 6736, 6867, 6877
- \c_msg_no_info_text_tl . 6537, 6550, 6602
- \c_msg_on_line_text_tl 6555, 6574
- \c_msg_on_line_tl 6537
- \c_msg_return_text_tl 160, 6537, 6553,
6556, 6951, 6959, 6966, 6973, 7028
- \c_msg_text_prefix_tl
... 6515, 6515, 6519, 6530, 6709,

- 6720, 6733, 6742, 6753, 6761, 6767,
6797, 6850, 6872, 6885, 6908, 6930
- \c_msg_trouble_text_tl . 160, 6537, 6563
- \c_nine 79, 2512,
2544, 3803, 3809, 6118, 7903, 9251,
9473, 9559, 9805, 9814, 10033, 10325
- \c_one 79, 2496, 2528,
3301, 3803, 3803, 6110, 7792, 7803,
7862, 7874, 7922, 7928, 7970, 7995,
8192, 8551, 8555, 8557, 8564, 8704,
8733, 8993, 9030, 9035, 9056, 9179,
9247, 9290, 9304, 9355, 9370, 9395,
9407, 9468, 9554, 9602, 9609, 9624,
9650, 9672, 9677, 9685, 9691, 9779,
9783, 9985, 9995, 10096, 10121,
10132, 10136, 10158, 10178, 10184,
10288, 10292, 10323, 10340, 10392,
10415, 10421, 10422, 10466, 10484,
10522, 10543, 10549, 10703, 10705
- \c_one_fp 188, 7733, 7733, 10602
- \c_one_hundred
. 79, 3818, 3818, 7925, 7926, 9994
- \c_one_hundred_million
. 7715, 7717, 8914, 9836
- \c_one_million 7715, 7716, 9177
- \c_one_thousand
79, 3818, 3819, 8824, 9080, 9124, 9175
- \c_one_thousand_million
. 7715, 7719, 7865,
7887, 7904, 7914, 7950, 7961, 7975,
7986, 8027, 8069, 8343, 8362, 8481,
8517, 8543, 8594, 8619, 8685, 8702,
8705, 8720, 8729, 8806, 8845, 8853,
8862, 8949, 8962, 8998, 9028, 9031,
9033, 9036, 9046, 9050, 9057, 9058,
9178, 9180, 9192, 9204, 9219, 9252,
9263, 9281, 9339, 9343, 9359, 9363,
9437, 9492, 9534, 9578, 9629, 9635,
9683, 9687, 9689, 9693, 9701, 9705,
9735, 9859, 9929, 10104, 10114,
10133, 10151, 10176, 10180, 10182,
10186, 10194, 10198, 10268, 10289,
10311, 10363, 10430, 10433, 10502,
10541, 10545, 10547, 10551, 10695
- \c_other_cctab
. 191, 11166, 11176, 11184, 11194
- \c_other_char_token 3112, 3115
- \c_parameter_token
. 57, 2582, 2589, 2622, 2625
- \c_pdftex_is_engine_bool
. 25, 1448, 1458, 1468, 1480
- \c_pi_fp 188, 7734, 7734
- \c_seven 79, 1156,
1166, 2260, 2508, 2540, 3803, 6116
- \c_six 79, 1156, 1165, 2257,
2506, 2538, 3803, 6115, 9315, 9321
- \c_sixteen
. 79, 1156, 1163, 1172, 3738, 3803,
6103, 6104, 6138, 6141, 6154, 6156,
6167, 6169, 6202, 6221, 6239, 6258
- \c_space_tl 108, 4680,
4680, 5066, 5072, 5083, 5529, 5535,
5546, 5822, 5828, 5839, 5841, 6312,
6314, 6382, 6383, 6395, 6396, 6575
- \c_space_token
. 57, 2582, 2593, 2641, 2946, 2965
- \c_string_cctab
. 191, 11166, 11177, 11185, 11199
- \c_ten 79,
2514, 2546, 3803, 3810, 6119, 7915,
7962, 7987, 8139, 8203, 8259, 8361,
8366, 8478, 8514, 8553, 8556, 8566,
8961, 9015, 9191, 9240, 9282, 9294,
9372, 9764, 10385, 10618, 10652, 10657
- \c_ten_thousand 79, 3818, 3820
- \c_thirteen 79, 2520, 2552, 3803, 3812, 6122
- \c_thirty_two 79, 3815, 3815
- \c_three 79, 2500, 2532,
3803, 3805, 6112, 9313, 9621, 9954
- \c_tl_rescan_marker_tl
. 4284, 4292, 4302, 4313, 4340
- \c_token_A_int 2807, 2842
- \c_true_bool
. 23, 992, 1021, 1061, 1061, 1094,
1458, 1469, 1479, 1872, 1876, 1977,
1980, 1986, 1987, 2015, 2016, 2019,
2021, 2025, 2048, 3653, 3658, 3671
- \c_twelve 79, 1156, 1167, 2265,
2279, 2518, 2550, 2719, 3803, 6121
- \c_two 79,
2498, 2530, 3736, 3803, 3804, 6111,
8330, 9318, 9596, 9600, 9619, 9653,
9776, 9782, 10435–10437, 10450, 10526
- \c_two_hundred_fifty_five 79, 3816, 3816
- \c_two_hundred_fifty_six . 79, 3816, 3817
- \c_undefined:D 1297, 1299
- \c_undefined_fp 188, 7735, 7735,
8892, 9792, 10585, 10635, 10642, 10762

<code>\c_xetex_is_engine_bool</code>	<code>\catcodetable</code>	755
..... 25, 1448, 1459, 1469	<code>\CatcodeTableIniTeX</code>	11175
<code>\c_zero</code>	<code>\CatcodeTableLaTeX</code>	11174
79, 991, 999,	<code>\CatcodeTableOther</code>	11176
1007, 1014, 1020, 1028, 1036, 1043,	<code>\CatcodeTableString</code>	11177
1156, 1164, 1486, 1491, 1567, 1575,	<code>\cctab_begin:N</code>	
2164–2174, 2252, 2254, 2259, 2461, 191, 11126, 11126, 11146, 11152	
2470, 2494, 2526, 2690, 3187, 3217,	<code>\cctab_end:</code> 191, 11126, 11136, 11147, 11153	
3221, 3222, 3228, 3278, 3279, 3548,	<code>\cctab_gset:Nn</code> 191, 11157, 11157,	
3803, 3879, 4041, 4080, 4090, 4091,	11165, 11169, 11186, 11194, 11199	
4118, 5172, 5185, 5573, 5583, 5917,	<code>\cctab_new:N</code>	191, 11096, 11096,
6109, 6138, 6141, 6498, 6500, 7812,	11115, 11119, 11168, 11183–11185	
7823, 7861, 7873, 7875, 7886, 7929–	<code>\char</code>	519
7931, 8033, 8075, 8118, 8121, 8183,	<code>\char_active_gset:Npn</code>	68, 3085
8250, 8385–8393, 8424–8432, 8487,	<code>\char_active_gset:Npx</code>	68, 3086
8523, 8567, 8622, 8660, 8676, 8718,	<code>\char_active_gset_eq:NN</code> .	68, 3071, 3088
8722, 8724, 8790, 8794, 8819, 8888,	<code>\char_active_set:Npn</code>	68, 3071, 3083
8898, 8911, 8912, 8933, 8937, 8958,	<code>\char_active_set:Npx</code>	68, 3071, 3084
8967, 8968, 8996, 9044, 9048, 9052,	<code>\char_active_set_eq:NN</code> ..	68, 3071, 3087
9053, 9072, 9116, 9190, 9218, 9229,	<code>\char_make_active:N</code>	3117, 3132
9237, 9295, 9298, 9305–9307, 9337,	<code>\char_make_active:n</code>	3117, 3150
9341, 9345, 9350, 9373, 9374, 9379,	<code>\char_make_alignment:N</code>	3117
9388, 9392, 9403, 9428, 9469, 9483,	<code>\char_make_alignment:n</code>	3117
9525, 9555, 9569, 9595, 9608, 9610,	<code>\char_make_alignment_tab:N</code>	3121
9622, 9623, 9625, 9626, 9628, 9630,	<code>\char_make_alignment_tab:n</code>	3139
9633, 9644–9646, 9678, 9679, 9699,	<code>\char_make_begin_group:N</code>	3118
9703, 9726, 9775, 9789, 9790, 9820,	<code>\char_make_begin_group:n</code>	3136
9821, 9833, 9834, 9850, 9917, 9920,	<code>\char_make_comment:N</code>	3117, 3133
9956, 9982, 9986–9988, 9996–9998,	<code>\char_make_comment:n</code>	3117, 3151
10010, 10043, 10061, 10062, 10094,	<code>\char_make_end_group:N</code>	3119
10095, 10100, 10101, 10106, 10135,	<code>\char_make_end_group:n</code>	3137
10171, 10172, 10192, 10196, 10235,	<code>\char_make_end_line:N</code>	3117, 3122
10239, 10291, 10302, 10319, 10329–	<code>\char_make_end_line:n</code>	3117, 3140
10332, 10348, 10374, 10382, 10396,	<code>\char_make_escape:N</code>	3117, 3117
10397, 10399, 10402, 10448, 10449,	<code>\char_make_escape:n</code>	3117, 3135
10452, 10457, 10459, 10471, 10488,	<code>\char_make_group_begin:N</code>	3117
10495, 10501, 10511, 10519, 10534,	<code>\char_make_group_begin:n</code>	3117
10577, 10581, 10598, 10613, 10617,	<code>\char_make_group_end:N</code>	3117
10624, 10649, 10654, 10656, 10706,	<code>\char_make_group_end:n</code>	3117
10707, 10735, 10753–10756, 10843,	<code>\char_make_ignore:N</code>	3117, 3128
10846, 10850, 10861, 10862, 10884	<code>\char_make_ignore:n</code>	3117, 3146
<code>\c_zero_dim</code>	<code>\char_make_invalid:N</code>	3117, 3134
88, 3889, 4026, 4027, 4032, 4110, 6064	<code>\char_make_invalid:n</code>	3117, 3152
<code>\c_zero_fp</code> 188, 7736, 7736, 8001, 8011,	<code>\char_make_letter:N</code>	3117, 3130
8013, 8902, 9768, 9823, 9946, 9973,	<code>\char_make_letter:n</code>	3117, 3148
10013, 10245, 10253, 10591, 10770	<code>\char_make_math_shift:N</code>	3120
<code>\c_zero_muskip</code>	<code>\char_make_math_shift:n</code>	3138
4129	<code>\char_make_math_subscript:N</code> .	3117, 3126
<code>\c_zero_skip</code>	<code>\char_make_math_subscript:n</code> .	3117, 3144
91, 4051, 4110, 4110, 4164, 4165, 6051	<code>\char_make_math_superscript:N</code>	3117, 3124
<code>\catcode</code>		
3–		
6, 9, 70–78, 84–91, 101, 281–288, 665		

- \char_make_math_superscript:n [3117](#), [3142](#)
- \char_make_math_toggle:N [3117](#)
- \char_make_math_toggle:n [3117](#)
- \char_make_other:N [3117](#), [3131](#)
- \char_make_other:n [3117](#), [3149](#)
- \char_make_parameter:N [3117](#), [3123](#)
- \char_make_parameter:n [3117](#), [3141](#)
- \char_make_space:N [3117](#), [3129](#)
- \char_make_space:n [3117](#), [3147](#)
- \char_set_catcode:nn [55](#), [298–306](#), [2487](#), [2487](#), [2494](#), [2496](#), [2498](#), [2500](#), [2502](#), [2504](#), [2506](#), [2508](#), [2510](#), [2512](#), [2514](#), [2516](#), [2518](#), [2520](#), [2522](#), [2524](#), [2526](#), [2528](#), [2530](#), [2532](#), [2534](#), [2536](#), [2538](#), [2540](#), [2542](#), [2544](#), [2546](#), [2548](#), [2550](#), [2552](#), [2554](#), [2556](#), [2719](#)
- \char_set_catcode:w [3090](#), [3090](#), [3095](#), [3097](#)
- \char_set_catcode_active:N [54](#), [2493](#), [2519](#), [2598](#), [3072](#), [3132](#), [6636](#)
- \char_set_catcode_active:n . [54](#), [2525](#), [2551](#), [3077](#), [3150](#), [7056](#), [7057](#), [11192](#)
- \char_set_catcode_alignment:N [54](#), [2493](#), [2501](#), [2587](#), [3121](#)
- \char_set_catcode_alignment:n [54](#), [316](#), [2525](#), [2533](#), [3139](#)
- \char_set_catcode_comment:N [54](#), [2493](#), [2521](#), [3133](#)
- \char_set_catcode_comment:n [54](#), [2525](#), [2553](#), [3151](#)
- \char_set_catcode_end_line:N [54](#), [2493](#), [2503](#), [3122](#)
- \char_set_catcode_end_line:n [54](#), [2525](#), [2535](#), [3140](#)
- \char_set_catcode_escape:N [54](#), [2493](#), [2493](#), [3117](#)
- \char_set_catcode_escape:n [54](#), [2525](#), [2525](#), [3135](#)
- \char_set_catcode_group_begin:N [54](#), [2493](#), [2495](#), [3118](#)
- \char_set_catcode_group_begin:n [54](#), [2525](#), [2527](#), [3136](#)
- \char_set_catcode_group_end:N [54](#), [2493](#), [2497](#), [3119](#)
- \char_set_catcode_group_end:n [54](#), [2525](#), [2529](#), [3137](#)
- \char_set_catcode_ignore:N [54](#), [2493](#), [2511](#), [3128](#)
- \char_set_catcode_ignore:n [54](#), [313](#), [314](#), [2525](#), [2543](#), [3146](#), [7801](#)
- \char_set_catcode_invalid:N [54](#), [2493](#), [2523](#), [3134](#)
- \char_set_catcode_invalid:n [54](#), [2525](#), [2555](#), [3152](#)
- \char_set_catcode_letter:N [54](#), [2493](#), [2515](#), [3130](#), [6578](#), [11086](#), [11087](#)
- \char_set_catcode_letter:n [54](#), [317](#), [319](#), [2525](#), [2547](#), [3148](#)
- \char_set_catcode_math_subscript:N . [54](#), [2493](#), [2509](#), [2591](#), [3127](#)
- \char_set_catcode_math_subscript:n . [54](#), [2525](#), [2541](#), [3145](#), [11191](#)
- \char_set_catcode_math_superscript:N [54](#), [2493](#), [2507](#), [3125](#)
- \char_set_catcode_math_superscript:n [54](#), [318](#), [2525](#), [2539](#), [3143](#)
- \char_set_catcode_math_toggle:N [54](#), [2493](#), [2499](#), [2585](#), [3120](#)
- \char_set_catcode_math_toggle:n [54](#), [2525](#), [2531](#), [3138](#), [7109](#), [7144](#)
- \char_set_catcode_other:N . [54](#), [2493](#), [2517](#), [2675](#), [2676](#), [2809](#), [3131](#), [6583](#)
- \char_set_catcode_other:n [54](#), [315](#), [320](#), [2525](#), [2549](#), [3149](#), [11190](#), [11197](#), [11202](#)
- \char_set_catcode_parameter:N [54](#), [2493](#), [2505](#), [3123](#)
- \char_set_catcode_parameter:n [54](#), [2525](#), [2537](#), [3141](#)
- \char_set_catcode_space:N [54](#), [2493](#), [2513](#), [3129](#)
- \char_set_catcode_space:n . [54](#), [321](#), [2525](#), [2545](#), [3147](#), [11188](#), [11189](#), [11203](#)
- \char_set_lccode:nn [55](#), [2557](#), [2563](#), [2677–2679](#), [2712–2717](#), [2810–2812](#), [3079](#), [7058](#), [7059](#)
- \char_set_lccode:w [3090](#), [3092](#), [3101](#), [3103](#), [6634](#), [6635](#)
- \char_set_mathcode:nn . . . [56](#), [2557](#), [2557](#)
- \char_set_mathcode:w [3090](#), [3091](#), [3098](#), [3100](#)
- \char_set_sfcode:nn [57](#), [2557](#), [2575](#)
- \char_set_sfcode:w [3090](#), [3094](#), [3107](#), [3109](#)
- \char_set_uccode:nn [56](#), [2557](#), [2569](#)
- \char_set_uccode:w [3090](#), [3093](#), [3104](#), [3106](#)
- \char_show_value_catcode:n [55](#), [2487](#), [2491](#)
- \char_show_value_catcode:w . . [3095](#), [3096](#)
- \char_show_value_lccode:n [55](#), [2557](#), [2567](#)
- \char_show_value_lccode:w . . . [3095](#), [3102](#)
- \char_show_value_mathcode:n [56](#), [2557](#), [2561](#)
- \char_show_value_mathcode:w . [3095](#), [3099](#)

\char_show_value_sfcode:n 57, 2557, 2579
 \char_show_value_sfcode:w ... 3095, 3108
 \char_show_value_uccode:n 56, 2557, 2573
 \char_show_value_uccode:w ... 3095, 3105
 \char_tmp:NN 3073, 3083–3088
 \char_value_catcode:n
 55, 298–306, 2487, 2489
 \char_value_catcode:w 3095, 3095
 \char_value_lccode:n 55, 2557, 2565
 \char_value_lccode:w 3095, 3101
 \char_value_mathcode:n .. 56, 2557, 2559
 \char_value_mathcode:w 3095, 3098
 \char_value_sfcode:n 57, 2557, 2577
 \char_value_sfcode:w 3095, 3107
 \char_value_uccode:n 56, 2557, 2571
 \char_value_uccode:w 3095, 3104
 \chardef 80, 93, 96, 328, 354
 \chk_if_exist_cs:c 27, 1215, 1223
 \chk_if_exist_cs:N
 27, 1215, 1215, 1224, 1769
 \chk_if_free_cs:c 27, 1192, 1213
 \chk_if_free_cs:N 27, 1192, 1192, 1202,
 1214, 1229, 1286, 3252, 3267, 3884,
 4046, 4123, 4182, 4188, 4193, 5923
 .choice: 171
 .choice_code:n 171
 .choice_code:x 171
 \cleaders 537
 \clist_clear:c 124, 5260, 5261
 \clist_clear:N
 ... 124, 5260, 5260, 5365, 5383, 5596
 \clist_clear_new:c 124, 5264, 5265
 \clist_clear_new:N 124, 5264, 5264
 \clist_concat:ccc 124, 5276
 \clist_concat:NNN . 124, 5276, 5276, 5296
 \clist_concat_aux:NNNN
 5276, 5277, 5279, 5280
 \clist_display:c 5623, 5624
 \clist_display:N 5623, 5623
 \clist_gclear:c 124, 5260, 5263
 \clist_gclear:N ... 124, 5260, 5262, 5607
 \clist_gclear_new:c 124, 5264, 5267
 \clist_gclear_new:N 124, 5264, 5266
 \clist_gconcat:ccc 125, 5276
 \clist_gconcat:NNN . 125, 5276, 5278, 5297
 \clist_get:cN .. 126, 127, 132, 5320, 5620
 \clist_get:NN
 126, 127, 132, 5320, 5320, 5324, 5619
 \clist_get_aux:wN 5320, 5321, 5322
 \clist_gpop:cN 127, 132, 5325
 \clist_gpop:NN . 127, 132, 5325, 5327, 5339
 \clist_gpush:cn 127, 133, 5340, 5352
 \clist_gpush:co 127, 133, 5340, 5354
 \clist_gpush:cV 127, 133, 5340, 5353
 \clist_gpush:cx 127, 133, 5340, 5355
 \clist_gpush:Nn 127, 133, 5340, 5348
 \clist_gpush:No 127, 133, 5340, 5350
 \clist_gpush:NV 127, 133, 5340, 5349
 \clist_gpush:Nx 127, 133, 5340, 5351
 \clist_gput_left:cn 126, 5298, 5352
 \clist_gput_left:co 126, 5298, 5354
 \clist_gput_left:cV 126, 5298, 5353
 \clist_gput_left:cx 126, 5298, 5355
 \clist_gput_left:Nn
 ... 126, 5298, 5300, 5310, 5311, 5348
 \clist_gput_left:No 126, 5298, 5350
 \clist_gput_left:NV 126, 5298, 5349
 \clist_gput_left:Nx 126, 5298, 5351
 \clist_gput_right:cn 126, 5312
 \clist_gput_right:co 126, 5312
 \clist_gput_right:cV 126, 5312
 \clist_gput_right:cx 126, 5312
 \clist_gput_right:Nn
 126, 5312, 5314, 5318, 5319
 \clist_gput_right:No 126, 5312
 \clist_gput_right:NV 126, 5312
 \clist_gput_right:Nx 126, 5312
 \clist_gremove_all:cn 129, 5375
 \clist_gremove_all:Nn
 129, 5375, 5377, 5404, 5622
 \clist_gremove_duplicates:c .. 128, 5359
 \clist_gremove_duplicates:N
 128, 5359, 5361, 5374
 \clist_gremove_element:Nn ... 5621, 5622
 \clist_gset_eq:cc 124, 5268, 5275
 \clist_gset_eq:cN 124, 5268, 5274
 \clist_gset_eq:Nc 124, 5268, 5273
 \clist_gset_eq:NN
 124, 5268, 5272, 5362, 5378
 \clist_gset_from_seq:cc 134, 5593
 \clist_gset_from_seq:cN 134, 5593
 \clist_gset_from_seq:Nc 134, 5593
 \clist_gset_from_seq:NN
 134, 5593, 5604, 5617, 5618
 \clist_gtrim_spaces:c 129, 5405
 \clist_gtrim_spaces:N 129, 5405, 5415, 5418
 \clist_if_empty:c 5421
 \clist_if_empty:cTF 129, 5420
 \clist_if_empty:N 5420
 \clist_if_empty:NF 5288, 5381, 5463

\clist_if_empty:NTF 129, 5284, 5304, 5420, 5527
\clist_if_empty_p:c 129, 5420
\clist_if_empty_p:N 129, 5420
\clist_if_eq:cc 5425
\clist_if_eq:ccTF 130, 5422
\clist_if_eq:cN 5424
\clist_if_eq:cNTF 130, 5422
\clist_if_eq:Nc 5423
\clist_if_eq:NcTF 130, 5422
\clist_if_eq:NN 5422
\clist_if_eq:NNTF 130, 5422
\clist_if_eq_p:cc 130, 5422
\clist_if_eq_p:cN 130, 5422
\clist_if_eq_p:Nc 130, 5422
\clist_if_eq_p:NN 130, 5422
\clist_if_in:cnTF 130, 5426
\clist_if_in:coTF 130, 5426
\clist_if_in:cVTF 130, 5426
\clist_if_in:Nn 5426
\clist_if_in:nn 5439
\clist_if_in:NnF 5368, 5454, 5455
\clist_if_in:nnF 5459
\clist_if_in:NnT 5452, 5453
\clist_if_in:nnT 5458
\clist_if_in:NnTF 130, 5426, 5456, 5457
\clist_if_in:nnTF 130, 5460
\clist_if_in:NoTF 130, 5426
\clist_if_in:noTF 130
\clist_if_in:NVTF 130, 5426
\clist_if_in:nVTF 130
\clist_item:cn 134, 5569
\clist_item:Nn 134, 5569, 5569, 5592
\clist_item:nn 134, 5569, 5570, 5571
\clist_item_aux:nnn 5569
\clist_item_aux:nw 5575, 5579, 5581, 5587
\clist_length:c 133, 5551
\clist_length:N 133, 5551, 5551, 5568
\clist_length:n 133, 5551, 5559, 5576
\clist_length_aux:n 5551, 5556, 5564, 5567
\clist_map_break: 131, 5523, 5523
\clist_map_break:n 131, 5523, 5524
\clist_map_function:cN 130, 5461
\clist_map_function:NN 130, 5221, 5231, 5461, 5461, 5483, 5491, 5539, 5556
\clist_map_function:nN 130, 5226, 5236, 5408, 5461, 5469, 5501, 5564, 7283
\clist_map_function_aux:Nw 5461, 5465, 5473, 5477, 5481
\clist_map_inline:cn 131, 5485
\clist_map_inline:Nn 131, 5366, 5485, 5485, 5505, 5508, 7644, 7704
\clist_map_inline:nn 131, 5485, 5495, 5516
\clist_map_variable:cNn 131, 5506
\clist_map_variable:NNn 131, 5506, 5506, 5522
\clist_map_variable:nNn 131, 5514
\clist_new:c 123, 5258, 5259
\clist_new:N 123, 5258, 5258, 5358
\clist_pop:cN 127, 132, 5325
\clist_pop:NN 127, 132, 5325, 5325, 5338
\clist_pop_aux:NNN 5325, 5326, 5328, 5329
\clist_pop_aux:wNNN 5325, 5330, 5331
\clist_push:cn 127, 133, 5340, 5344
\clist_push:co 127, 133, 5340, 5346
\clist_push:cV 127, 133, 5340, 5345
\clist_push:cx 127, 133, 5340, 5347
\clist_push:Nn 127, 133, 5340, 5340
\clist_push:No 127, 133, 5340, 5342
\clist_push:NV 127, 133, 5340, 5341
\clist_push:Nx 127, 133, 5340, 5343
\clist_put_aux:NNnnNn 5299, 5301, 5302, 5313, 5315
\clist_put_left:cn 125, 5298, 5344
\clist_put_left:co 125, 5298, 5346
\clist_put_left:cV 125, 5298, 5345
\clist_put_left:cx 125, 5298, 5347
\clist_put_left:Nn 125, 5298, 5298, 5308, 5309, 5340
\clist_put_left:No 125, 5298, 5342
\clist_put_left:NV 125, 5298, 5341
\clist_put_left:Nx 125, 5298, 5343
\clist_put_right:cn 126, 5312
\clist_put_right:co 126, 5312
\clist_put_right:cV 126, 5312
\clist_put_right:cx 126, 5312
\clist_put_right:Nn 126, 5312, 5312, 5316, 5317, 5369
\clist_put_right:No 126, 5312
\clist_put_right:NV 126, 5312
\clist_put_right:Nx 126, 5312
\clist_remove_all:cn 128, 5375
\clist_remove_all:Nn 128, 5375, 5375, 5403, 5621
\clist_remove_all_aux:NNn 5375, 5376, 5378, 5379
\clist_remove_all_aux:w 5375, 5384, 5389, 5391, 5402
\clist_remove_duplicates:c 128, 5359

- \clist_remove_duplicates:N 128, 5359, 5359, 5373
- \clist_remove_duplicates_aux:NN 5359, 5360, 5362, 5363
- \clist_remove_element:Nn 5621, 5621
- \clist_set_eq:cc 124, 5268, 5271
- \clist_set_eq:cN 124, 5268, 5270
- \clist_set_eq:Nc 124, 5268, 5269
- \clist_set_eq:NN 124, 5268, 5268, 5360, 5376
- \clist_set_from_seq:cc 134, 5593
- \clist_set_from_seq:cN 134, 5593
- \clist_set_from_seq:Nc 134, 5593
- \clist_set_from_seq:NN 134, 5593, 5593, 5615, 5616
- \clist_show:c 133, 5525, 5624
- \clist_show:N . 133, 5525, 5525, 5550, 5623
- \clist_show_aux:n 5525, 5539, 5544
- \clist_show_aux:w 5525, 5541, 5549
- \clist_tmp:w 5419, 5419, 5429, 5437, 5442, 5450
- \clist_top:cN 5619, 5620
- \clist_top:NN 5619, 5619
- \clist_trim_spaces:c 129, 5405
- \clist_trim_spaces:N 129, 5405, 5413, 5417
- \clist_trim_spaces:n 129, 5405, 5405, 5414, 5416
- \clist_trim_spaces_aux_i:n 5405, 5407, 5410
- \clist_trim_spaces_aux_ii:n 5405, 5408, 5411
- \clist_use:c 128, 5356, 5357
- \clist_use:N 128, 5356, 5356
- \closein 413
- \closeout 408
- \clubpenalties 721
- \clubpenalty 548
- .code:n 171
- .code:x 171
- \copy 605
- \count 656
- \countdef 355
- \cr 380
- \crrcr 381
- \cs:w 17, 805, 807, 820, 852, 1120, 1148, 1361, 1393, 1531, 1570, 1583, 1585, 1587, 1591–1593, 1628, 1634, 1654, 1656, 1661, 1668, 1669, 1727, 1766, 2143, 2145, 3318, 4575, 4740, 10083, 10443
- \cs_end: 17, 805, 808, 820, 852, 1114, 1120, 1142, 1148, 1361, 1393, 1531, 1570, 1583, 1585, 1587, 1591–1593, 1628, 1634, 1654, 1656, 1661, 1668, 1669, 1727, 1766, 2140, 2146–2149, 2151, 2153, 2155, 2157, 2159, 2161, 2163, 3318, 4574, 4575, 4740, 9450, 9538, 9748, 9933, 9943, 10083, 10272, 10443
- \cs_generate_from_arg_count:cNnn ... 15, 1018, 1026, 1034, 1042, 1349, 1392
- \cs_generate_from_arg_count:NNnn ... 15, 1318, 1318, 1350, 1360
- \cs_generate_from_arg_count_error_msg:Nn 1318, 1343, 1351
- \cs_generate_internal_variant:n 36, 1792, 1838, 1838
- \cs_generate_internal_variant_aux:N 1838, 1843, 1846, 1852
- \cs_generate_variant:Nn 28, 1767, 1767, 1854–1861, 1870, 1879–1882, 1895, 1896, 1905–1908, 2054, 2055, 2060, 2061, 2117, 2134, 2436, 2437, 2454–2457, 2476–2479, 3256, 3277, 3280, 3281, 3283, 3284, 3286, 3287, 3296–3299, 3308–3311, 3315, 3316, 3683, 3888, 3891, 3892, 3896, 3897, 3899, 3900, 3902, 3903, 3912–3915, 3919, 3920, 3924, 3925, 4023, 4025, 4050, 4053, 4054, 4058, 4059, 4061, 4062, 4064, 4065, 4069, 4070, 4074, 4075, 4099, 4106, 4107, 4109, 4127, 4131, 4132, 4136, 4137, 4139, 4140, 4142, 4143, 4147, 4148, 4152, 4153, 4157, 4159, 4185, 4196, 4197, 4202, 4203, 4208, 4209, 4230–4235, 4252–4259, 4276–4283, 4317–4320, 4335, 4336, 4373, 4374, 4403, 4404, 4409, 4410, 4415, 4416, 4419–4426, 4435–4438, 4446–4449, 4468–4471, 4490–4492, 4499–4507, 4534, 4555, 4566, 4570, 4596, 4597, 4605, 4608, 4625, 4626, 4631, 4632, 4650–4653, 4661, 4735, 4736, 4765, 4803, 4804, 4809–4812, 4817–4820, 4836, 4837, 4862, 4863, 4885–4890, 4899, 4915, 4916, 4940, 4967, 4968, 4993, 5022, 5033, 5034, 5087, 5110–5115, 5144–5155, 5165, 5189, 5191, 5216, 5217, 5239–5244, 5296, 5297,

5308–5311, 5316–5319, 5324, 5338, 5339, 5373, 5374, 5403, 5404, 5417, 5418, 5452–5460, 5483, 5505, 5522, 5550, 5568, 5592, 5615–5618, 5642, 5678–5681, 5690, 5691, 5709–5712, 5727, 5729, 5731, 5733, 5748, 5749, 5758–5761, 5781–5788, 5802, 5803, 5814, 5845, 5898, 5899, 5901–5904, 5927, 5932, 5933, 5946, 5947, 5952, 5953, 5958, 5959, 5963–5965, 5972– 5974, 5977, 5978, 5994–6001, 6004– 6007, 6013, 6014, 6029, 6033, 6034, 6039, 6040, 6045, 6046, 6058, 6059, 6067, 6068, 6073, 6074, 6079, 6080, 6085, 6086, 6091, 6092, 6149, 6150, 6177, 6178, 6293, 6294, 6348, 6351, 6700–6703, 6830, 7197, 7331, 7356, 7357, 7442, 7443, 8003, 8009, 8014, 8015, 8048, 8049, 8096, 8097, 8112, 8174, 8177, 8244, 8469, 8472, 8504, 8507, 8588, 8589, 8613, 8614, 8639, 8640, 8742, 8743, 8760, 8761, 8873, 8874, 9415, 9416, 9512, 9513, 9713, 9714, 9906, 9907, 10209, 10224, 10225, 10558, 10559, 11081, 11084	\cs_gnew:Npn 1495
\cs_generate_variant_aux:N	\cs_gnew:Npx 1499
. 1767, 1782, 1803, 1809	\cs_gnew_eq:cc 1513
\cs_generate_variant_aux:NNn	\cs_gnew_eq:cN 1511
. 1767, 1780, 1786	\cs_gnew_eq:Nc 1512
\cs_generate_variant_aux:nnNNn	\cs_gnew_eq:NN 1510
. 1767, 1770, 1773	\cs_gnew_nopar:cpn 1502
\cs_generate_variant_aux:Nnnw	\cs_gnew_nopar:cpx 1506
. 1767, 1774, 1775, 1784	\cs_gnew_nopar:Npn 1494
\cs_generate_variant_aux:NNpx	\cs_gnew_nopar:Npx 1498
. 1790, 1811, 1824	\cs_gnew_protected:cpn 1505
\cs_generate_variant_aux:w	\cs_gnew_protected:cpx 1509
. 1811, 1826, 1829	\cs_gnew_protected:Npn 1497
\cs_get_arg_count_from_signature:c	\cs_gnew_protected:Npx 1501
. 1316, 1394	\cs_gnew_protected_nopar:cpn 1504
\cs_get_arg_count_from_signature:N	\cs_gnew_protected_nopar:cpx 1508
. 21, 937,	\cs_gnew_protected_nopar:Npn 1496
946, 953, 963, 1300, 1300, 1317, 1362	\cs_gnew_protected_nopar:Npx 1500
\cs_get_arg_count_from_signature_aux:nnN	\cs_gset:cn 14, 1397
. 1300, 1301, 1302	\cs_gset:cpn 11, 1249,
\cs_get_arg_count_from_signature_auxii:w	1251, 4538, 4548, 5488, 5498, 5808
. 1300, 1310, 1315	\cs_gset:cpx 11, 1249, 1252
\cs_get_function_name:N 21, 1096, 1096	\cs_gset:cx 14, 1397
\cs_get_function_signature:N 1096, 1098	\cs_gset:Nn 14, 1356
\cs_gnew:cpn 1503	\cs_gset:Npn 11, 838, 840, 1235,
\cs_gnew:cpx 1507	1251, 3085, 4997, 7120–7122, 11088
	\cs_gset:Npx
 11, 838, 842, 1236, 1252, 3086, 5002
	\cs_gset:Nx 14, 1356
	\cs_gset_eq:cc 15, 1292, 1295, 1890, 4217
	\cs_gset_eq:cN
 15, 1292, 1294, 1299, 1889, 4215,
	5006, 5639, 6206, 6243, 7155, 7156
	\cs_gset_eq:Nc 15, 1292, 1293, 1888,
	4216, 5013, 6198, 6214, 6235, 6251
	\cs_gset_eq:NN
 15, 1292, 1292–1295, 1297, 1876,
	1878, 1887, 3088, 4183, 4214, 5638
	\cs_gset_nopar:cn 14, 1397
	\cs_gset_nopar:cpn 11, 1241, 1245, 2204
	\cs_gset_nopar:cpx 11, 1241, 1246, 2983
	\cs_gset_nopar:cx 14, 1397
	\cs_gset_nopar:Nn 14, 1356
	\cs_gset_nopar:Npn 11, 838,
	838, 841, 845, 849, 1233, 1245, 4583
	\cs_gset_nopar:Npx 11, 838, 839,
	843, 847, 851, 1234, 1246, 4189,
	4194, 4225, 4227, 4229, 4245, 4247,
	4249, 4251, 4269, 4271, 4273, 4275
	\cs_gset_nopar:Nx 14, 1356

- \cs_gset_protected:cn [14](#), [1397](#)
- \cs_gset_protected:cpn .. [11](#), [1261](#), [1263](#)
- \cs_gset_protected:cpx .. [11](#), [1261](#), [1264](#)
- \cs_gset_protected:cx [14](#), [1397](#)
- \cs_gset_protected:Nn [14](#), [1356](#)
- \cs_gset_protected:Npn
..... [11](#), [838](#), [848](#), [1239](#), [1263](#)
- \cs_gset_protected:Npx
..... [11](#), [838](#), [850](#), [1240](#), [1264](#)
- \cs_gset_protected:Nx [14](#), [1356](#)
- \cs_gset_protected_nopar:cn ... [14](#), [1397](#)
- \cs_gset_protected_nopar:cpn
..... [11](#), [1255](#), [1257](#)
- \cs_gset_protected_nopar:cpx
..... [11](#), [1255](#), [1258](#)
- \cs_gset_protected_nopar:cx ... [14](#), [1397](#)
- \cs_gset_protected_nopar:Nn ... [14](#), [1356](#)
- \cs_gset_protected_nopar:Npn
..... [11](#), [838](#), [844](#), [1237](#), [1257](#)
- \cs_gset_protected_nopar:Npx
..... [11](#), [838](#), [846](#), [1238](#), [1258](#)
- \cs_gset_protected_nopar:Nx ... [14](#), [1356](#)
- \cs_gundefine:c [1515](#)
- \cs_gundefine:N [1514](#)
- \cs_if_eq:ccF [1437](#)
- \cs_if_eq:ccT [1436](#)
- \cs_if_eq:ccTF [1421](#), [1435](#)
- \cs_if_eq:cNF [1429](#)
- \cs_if_eq:cNT [1428](#)
- \cs_if_eq:cNTF [1421](#), [1427](#)
- \cs_if_eq:NcF [1433](#)
- \cs_if_eq:NcT [1432](#)
- \cs_if_eq:NcTF [1421](#), [1431](#)
- \cs_if_eq:NN [1421](#)
- \cs_if_eq:NNF [1429](#), [1433](#), [1437](#)
- \cs_if_eq:NNT [1428](#), [1432](#), [1436](#)
- \cs_if_eq:NNTF
.... [23](#), [1421](#), [1427](#), [1431](#), [1435](#), [6695](#)
- \cs_if_eq_p:cc [1421](#), [1434](#)
- \cs_if_eq_p:cN [1421](#), [1426](#)
- \cs_if_eq_p:Nc [1421](#), [1430](#)
- \cs_if_eq_p:NN . [23](#), [1421](#), [1426](#), [1430](#), [1434](#)
- \cs_if_exist:c [1112](#)
- \cs_if_exist:cF .. [3771](#), [3778](#), [3780](#), [7304](#)
- \cs_if_exist:cT [6519](#)
- \cs_if_exist:cTF [23](#),
[1100](#), [6197](#), [6223](#), [6234](#), [6260](#), [6787](#),
[6797](#), [7210](#), [7278](#), [7512](#), [7526](#), [10806](#)
- \cs_if_exist:N [1100](#)
- \cs_if_exist:NF [1217](#), [7253](#), [7350](#)
- \cs_if_exist:NT
.. [1460](#), [1471](#), [6271](#), [6283](#), [7641](#), [7658](#)
- \cs_if_exist:NTF
..... [23](#), [1100](#), [1440](#), [2704](#), [4205](#),
[4207](#), [5641](#), [5644](#), [5936](#), [5942](#), [6479](#)
- \cs_if_exist_p:c [23](#), [1100](#)
- \cs_if_exist_p:N [23](#), [1100](#)
- \cs_if_free:c [1140](#)
- \cs_if_free:cT [1840](#)
- \cs_if_free:cTF [23](#), [1128](#)
- \cs_if_free:N [1128](#)
- \cs_if_free:NF [1194](#), [1204](#)
- \cs_if_free:NTF
... [23](#), [1128](#), [1788](#), [6360](#), [6362](#), [11098](#)
- \cs_if_free_p:c [23](#), [1128](#)
- \cs_if_free_p:N [23](#), [1128](#)
- \cs_meaning:c [16](#), [821](#), [821](#)
- \cs_meaning:N [16](#), [805](#), [809](#), [821](#)
- \cs_new:cn [12](#), [1413](#)
- \cs_new:cpn . [9](#), [1249](#), [1253](#), [1503](#), [1946](#),
[1959](#), [1992](#), [1994](#), [1996](#), [1997](#), [2147](#)–
[2150](#), [2152](#), [2154](#), [2156](#), [2158](#), [2160](#),
[2162](#), [2164](#)–[2174](#), [3333](#), [3341](#), [3349](#),
[3357](#), [3365](#), [3373](#), [3381](#), [3957](#)–[3963](#)
- \cs_new:cpx [9](#), [1249](#), [1254](#), [1507](#), [1842](#)
- \cs_new:cx [12](#), [1413](#)
- \cs_new:Nn [12](#), [1381](#)
- \cs_new:Npn [9](#), [912](#),
[944](#), [1225](#), [1235](#), [1253](#), [1300](#), [1302](#),
[1315](#), [1351](#), [1495](#), [1525](#)–[1530](#), [1532](#),
[1534](#), [1546](#), [1552](#), [1574](#), [1577](#), [1578](#),
[1580](#), [1582](#), [1584](#), [1586](#), [1588](#), [1595](#),
[1597](#), [1602](#), [1607](#), [1613](#), [1619](#), [1625](#),
[1631](#), [1644](#), [1651](#), [1658](#), [1665](#), [1699](#),
[1700](#), [1705](#), [1707](#), [1712](#), [1717](#), [1719](#),
[1721](#), [1722](#), [1724](#), [1730](#), [1736](#), [1740](#),
[1747](#), [1749](#), [1751](#), [1753](#), [1754](#), [1756](#),
[1761](#), [1766](#), [1773](#), [1775](#), [1786](#), [1803](#),
[1829](#), [1846](#), [1891](#), [1893](#), [1919](#), [1924](#),
[1934](#), [1944](#), [1945](#), [1972](#)–[1974](#), [1983](#),
[1998](#), [2003](#), [2027](#), [2033](#), [2039](#), [2043](#),
[2044](#), [2050](#), [2052](#), [2056](#), [2058](#), [2062](#),
[2070](#), [2075](#), [2083](#), [2089](#), [2091](#), [2093](#),
[2099](#), [2101](#), [2103](#), [2109](#), [2111](#), [2118](#),
[2120](#), [2126](#), [2128](#), [2175](#), [2182](#), [2191](#),
[2389](#), [2395](#), [2404](#), [2417](#), [2433](#), [2832](#),
[2858](#), [2873](#), [2954](#), [3044](#), [3053](#), [3062](#),
[3075](#), [3182](#), [3184](#), [3192](#), [3203](#), [3214](#),
[3239](#), [3240](#), [3321](#), [3331](#), [3413](#), [3421](#),
[3429](#), [3435](#), [3441](#), [3449](#), [3457](#), [3463](#),

- 3482, 3514, 3546, 3559, 3575, 3608,
 3610, 3612, 3650, 3655, 3660, 3684,
 3692, 3694, 3703, 3705, 3714, 3716,
 3726, 3735, 3737, 3739, 3838, 3853,
 3945, 4450, 4520, 4522, 4529, 4585,
 4595, 4598, 4600, 4609, 4620, 4622,
 4623, 4627–4630, 4685, 4687, 4689,
 4691, 4709, 4764, 4766, 4774, 4935,
 4969, 4970, 4981, 4987, 5081, 5086,
 5156, 5164, 5209, 5238, 5405, 5410,
 5411, 5469, 5477, 5544, 5549, 5551,
 5559, 5567, 5664, 5768, 5794, 5837,
 5844, 5875, 5880, 5888, 5890, 6310,
 6459, 6466, 6699, 7106, 11076, 11079
- \cs_new:Npx 9, 1225, 1236, 1254, 1499
- \cs_new:Nx 12, 1381
- \cs_new_eq:cc 15, 968, 1284, 1291, 1513
- \cs_new_eq:cN 15, 1284,
 1289, 1511, 5635, 9767, 9791, 9822
- \cs_new_eq:Nc 15, 1284, 1290, 1512
- \cs_new_eq:NN 15, 1284, 1284, 1289–1291,
 1448–1459, 1494–1516, 1540, 1869,
 1883–1890, 2088, 2581–2583, 2866–
 2868, 3090–3094, 3110–3124, 3126,
 3128–3142, 3144, 3146–3168, 3178,
 3179, 3181, 3317, 3801, 3802, 3829–
 3831, 3875–3877, 4022, 4024, 4032,
 4033, 4098, 4100, 4103, 4108, 4110,
 4111, 4156, 4158, 4210–4217, 4347,
 4348, 4567, 4568, 4571, 4662, 4741–
 4763, 4781–4798, 4971–4973, 5035–
 5060, 5245–5248, 5258–5275, 5340–
 5357, 5523, 5524, 5619–5624, 5634,
 5645–5653, 5815, 5816, 5892, 5893,
 5900, 5960–5962, 5975, 5976, 5987–
 5989, 6008, 6016, 6022, 6028, 6047,
 6048, 6056, 6057, 6087–6090, 6102–
 6106, 6126, 6136, 6332, 6347, 6476,
 6501–6506, 7030–7035, 7166–7168,
 8100–8109, 11072, 11073, 11174–11177
- \cs_new_nopar:cn 12, 1413
- \cs_new_nopar:cpn 9, 1241, 1247, 1267–
 1274, 1276, 1278, 1502, 2010–2022,
 2024, 8151, 8152, 8154, 8156, 8158,
 8160, 8162, 8164, 8166, 8212, 8214,
 8216, 8218, 8220, 8222, 8224, 8226,
 8228, 8274, 8279, 8284, 8289, 8294,
 8299, 8304, 8309, 8314, 8319, 8324
- \cs_new_nopar:cpx 9, 1241, 1248, 1506
- \cs_new_nopar:cx 12, 1413
- \cs_new_nopar:Nn 12, 1381
- \cs_new_nopar:Npn 9, 1225, 1233, 1247,
 1316, 1349, 1426–1438, 1447, 1482,
 1494, 1524, 1558, 1569, 1637, 1673–
 1680, 1687–1691, 1738, 1739, 1824,
 2008, 2009, 2135, 2142, 2144, 2146,
 2251, 2253, 2255, 2269, 2274, 2283,
 2288, 2489, 2491, 2559, 2561, 2565,
 2567, 2571, 2573, 2577, 2579, 2599,
 2688, 2728, 2735, 2746, 2757, 2768,
 2779, 2787, 2795, 2803, 2824, 2831,
 2840, 2849, 2869–2872, 2923, 2932,
 2940, 3041, 3095, 3096, 3098, 3099,
 3101, 3102, 3104, 3105, 3107, 3108,
 3290, 3318, 3469, 3470, 3614, 3619,
 3624, 3629, 3634–3649, 3755, 3764,
 3798, 3800, 3832, 3926, 3928, 4020,
 4096, 4101, 4104, 4154, 4160, 4524,
 4569, 4572, 4590, 4884, 5088, 5166,
 5182, 5190, 5192, 5201, 5461, 5789,
 6081, 6457, 6475, 6569–6571, 6670–
 6675, 7251, 7500, 7502, 7510, 7519,
 7530, 8110, 8113, 8130, 8131, 8137,
 8146, 8167, 8173, 8175, 8178, 8190,
 8201, 8210, 8229, 8237, 8242, 8245,
 8257, 8266, 8268, 8325, 8338, 8357,
 8377, 8383, 8422, 8461–8463, 8465
- \cs_new_nopar:Npx
 9, 1225, 1234, 1248, 1498, 1834, 4310
- \cs_new_nopar:Nx 12, 1381
- \cs_new_protected:cn 12, 1413
- \cs_new_protected:cpn 9,
 1261, 1265, 1505, 7364, 7366, 7368,
 7370, 7372, 7374, 7392, 7402, 7404
- \cs_new_protected:cpx 9, 1261, 1266, 1509
- \cs_new_protected:cx 12, 1413
- \cs_new_protected:Nn 12, 1381
- \cs_new_protected:Npn 9,
 927, 961, 1225, 1239, 1265, 1280,
 1284, 1318, 1497, 1541, 1767, 1838,
 2201, 2214, 2223, 2232, 2373, 2374,
 2878, 2884, 2901, 2903, 2905, 2919,
 2921, 4186, 4191, 4218, 4220, 4222,
 4224, 4226, 4228, 4236, 4238, 4240,
 4242, 4244, 4246, 4248, 4250, 4260,
 4262, 4264, 4266, 4268, 4270, 4272,
 4274, 4299, 4337, 4354, 4379, 4535,
 4545, 4556, 4560, 4614, 4616, 4660,
 4730, 4805, 4807, 4813, 4815, 4822,
 4824, 4826, 4838, 4840, 4842, 4897,

- 4910, 4994, 4999, 5004, 5016, 5023,
 5218, 5223, 5228, 5233, 5302, 5312,
 5322, 5331, 5359, 5361, 5363, 5375,
 5377, 5379, 5402, 5413, 5415, 5419,
 5485, 5495, 5506, 5514, 5593, 5604,
 5634–5640, 5643, 5656, 5665, 5672,
 5674, 5676, 5682, 5688, 5692, 5698,
 5704, 5713–5715, 5719, 5739, 5805,
 5852, 5869, 5894, 5896, 5921, 5979,
 5981, 5983, 5985, 6031, 6035, 6049,
 6051, 6052, 6054, 6062, 6064, 6065,
 6069, 6075, 6374, 6401, 6497, 6499,
 6517, 6526, 6528, 6535, 6584, 6599,
 6607, 6615, 6617, 6640, 6655, 6662,
 6774, 6819, 6829, 6845, 6857, 6859,
 6861, 6863, 6865, 6892, 6901, 6914,
 6916, 6918, 6920, 6923, 6936, 6938,
 6940, 6942, 7036–7042, 7063, 7077,
 7089, 7110, 7124, 7145, 7152, 7189,
 7191, 7203, 7208, 7234, 7249, 7276,
 7302, 7313, 7318, 7329, 7434, 7436,
 7449, 7454, 7481, 11077, 11082, 11157
 \cs_new_protected:Npx
 9, [1225](#), 1240, 1266, [1501](#)
 \cs_new_protected:Nx [12](#), [1381](#)
 \cs_new_protected_nopar:cn [12](#), [1413](#)
 \cs_new_protected_nopar:cpn
 9, [1255](#), 1259, 1504,
 7358, 7360, 7362, 7376, 7378, 7380,
 7382, 7384, 7386, 7388, 7390, 7394,
 7396, 7398, 7400, 7406, 7408, 7410,
 7412, 7414, 7416, 7418, 7420, 7422,
 7424, 7426, 7428, 7430, 7432, 10813,
 10839, 10874, 10892, 10924, 10956
 \cs_new_protected_nopar:cpx
 9, [1255](#), 1260, [1508](#)
 \cs_new_protected_nopar:cx [12](#), [1413](#)
 \cs_new_protected_nopar:Nn [12](#), [1381](#)
 \cs_new_protected_nopar:Npn 9, [1225](#),
[1237](#), [1242](#), [1259](#), [1281](#)–[1283](#), [1289](#)–
[1296](#), [1298](#), [1496](#), [1672](#), [1681](#)–[1686](#),
[1692](#)–[1698](#), [1869](#), [1871](#), [1873](#), [1875](#),
[1877](#), [2295](#), [2382](#), [2487](#), [2493](#), [2495](#),
[2497](#), [2499](#), [2501](#), [2503](#), [2505](#), [2507](#),
[2509](#), [2511](#), [2513](#), [2515](#), [2517](#), [2519](#),
[2521](#), [2523](#), [2525](#), [2527](#), [2529](#), [2531](#),
[2533](#), [2535](#), [2537](#), [2539](#), [2541](#), [2543](#),
[2545](#), [2547](#), [2549](#), [2551](#), [2553](#), [2555](#),
[2557](#), [2563](#), [2569](#), [2575](#), [2581](#), [2874](#),
[2876](#), [2963](#), [2972](#), [3250](#), [3257](#), [3278](#),
[3279](#), [3282](#), [3285](#), [3288](#), [3292](#), [3294](#),
[3300](#), [3302](#), [3304](#), [3306](#), [3312](#), [3314](#),
[3882](#), [3889](#), [3890](#), [3893](#), [3895](#), [3898](#),
[3901](#), [3904](#), [3906](#), [3908](#), [3910](#), [3916](#),
[3918](#), [3921](#), [3923](#), [4044](#), [4051](#), [4052](#),
[4055](#), [4057](#), [4060](#), [4063](#), [4066](#), [4068](#),
[4071](#), [4073](#), [4121](#), [4128](#), [4130](#), [4133](#),
[4135](#), [4138](#), [4141](#), [4144](#), [4146](#), [4149](#),
[4151](#), [4180](#), [4198](#), [4200](#), [4204](#), [4206](#),
[4295](#), [4297](#), [4321](#), [4323](#), [4325](#), [4350](#),
[4352](#), [4375](#), [4377](#), [4405](#), [4407](#), [4411](#),
[4413](#), [4487](#)–[4489](#), [4558](#), [4606](#), [4737](#),
[4739](#), [4799](#), [4801](#), [4891](#), [4900](#), [4902](#),
[4904](#), [4917](#), [4923](#), [4941](#), [4943](#), [4945](#),
[4951](#), [4974](#), [5010](#), [5062](#), [5276](#), [5278](#),
[5280](#), [5298](#), [5300](#), [5314](#), [5320](#), [5325](#),
[5327](#), [5329](#), [5525](#), [5735](#), [5737](#), [5818](#),
[5928](#), [5930](#), [5934](#), [5940](#), [5948](#), [5950](#),
[5954](#), [5956](#), [5966](#), [5968](#), [5970](#), [6009](#),
[6011](#), [6030](#), [6032](#), [6037](#), [6041](#), [6043](#),
[6060](#), [6061](#), [6066](#), [6071](#), [6077](#), [6083](#),
[6093](#), [6139](#), [6142](#), [6151](#), [6164](#), [6179](#),
[6187](#), [6195](#), [6218](#), [6232](#), [6255](#), [6269](#),
[6281](#), [6295](#), [6317](#), [6349](#), [6352](#), [6353](#),
[6356](#), [6358](#), [6359](#), [6361](#), [6412](#), [6423](#),
[6436](#), [6443](#), [6452](#), [6489](#), [6491](#), [6493](#),
[6495](#), [6679](#), [6801](#), [6809](#), [6828](#), [6831](#),
[6833](#), [6835](#), [6837](#), [6839](#), [6841](#), [6843](#),
[7198](#), [7217](#), [7224](#), [7266](#), [7290](#), [7323](#),
[7332](#), [7337](#), [7342](#), [7348](#), [7354](#), [7444](#),
[7627](#), [7637](#), [7670](#), [7687](#), [7692](#), [7694](#),
[7785](#), [7787](#), [7798](#), [7808](#), [7833](#), [7841](#),
[7843](#), [7845](#), [7851](#), [7853](#), [7870](#), [7882](#),
[7937](#)–[7939](#), [7947](#), [7957](#), [7972](#), [7982](#),
[7997](#), [7998](#), [8004](#), [8010](#), [8012](#), [8016](#)–
[8018](#), [8050](#), [8052](#), [8054](#), [8467](#), [8470](#),
[8473](#), [8502](#), [8505](#), [8508](#), [8538](#), [8547](#),
[8560](#), [8586](#), [8587](#), [8590](#), [8611](#), [8612](#),
[8615](#), [8637](#), [8638](#), [8641](#), [8654](#), [8691](#),
[8708](#), [8740](#), [8741](#), [8744](#), [8758](#), [8759](#),
[8762](#), [8813](#), [8843](#), [8855](#), [8860](#), [8866](#),
[8871](#), [8872](#), [8875](#), [8910](#), [8956](#), [8974](#),
[8991](#), [9002](#), [9003](#), [9008](#), [9017](#), [9023](#),
[9039](#), [9062](#), [9106](#), [9166](#), [9183](#), [9188](#),
[9198](#), [9208](#), [9216](#), [9226](#), [9249](#), [9259](#),
[9273](#), [9279](#), [9292](#), [9311](#), [9329](#), [9368](#),
[9386](#), [9413](#), [9414](#), [9417](#), [9464](#), [9497](#),
[9510](#), [9511](#), [9514](#), [9550](#), [9583](#), [9606](#),
[9642](#), [9657](#), [9711](#), [9712](#), [9715](#), [9762](#),
[9772](#), [9801](#), [9831](#), [9904](#), [9905](#), [9908](#),

- 9952, 9980, 9992, 10027, 10059,
 10079, 10085, 10092, 10156, 10204,
 10222, 10223, 10226, 10259, 10283,
 10317, 10336, 10346, 10360, 10372,
 10394, 10419, 10427, 10439, 10445,
 10499, 10509, 10524, 10556, 10557,
 10560, 10611, 10647, 10701, 10713,
 10804, 10990, 10996, 11002, 11008,
 11014, 11020, 11026, 11038, 11046,
 11051, 11060, 11096, 11126, 11136
 \cs_new_protected_nopar:Npx
 9, 1225, 1238, 1260, 1500, 1832
 \cs_new_protected_nopar:Nx 12, 1381
 \cs_record_meaning:N 1191, 1191
 \cs_set:cn 13, 1397
 \cs_set:cpn 10, 1249, 1249, 6530, 6532, 7316
 \cs_set:cpx 10, 1249, 1250,
 2296, 2300, 2304, 2308, 2312, 2321,
 2330, 2339, 2348, 2350, 2352, 2354,
 2356, 2358, 2360, 2362, 2364, 7321
 \cs_set:cx 13, 1397
 \cs_set:Nn 13, 1356
 \cs_set:Npn 10, 824,
 826, 852, 861–894, 904, 935, 969,
 970, 1057–1060, 1078, 1083, 1093,
 1096, 1098, 1191, 1225, 1241, 1249,
 1356, 1389, 2371, 2372, 3073, 3083,
 3964, 3972, 3980, 3986, 3992, 4000,
 4008, 4014, 4495, 5571, 5581, 6391
 \cs_set:Npx
 .. 10, 824, 828, 856, 1250, 3084, 6382
 \cs_set:Nx 13, 1356
 \cs_set_eq:cc 15, 966, 1280, 1283, 1886, 4213
 \cs_set_eq:cN
 15, 1280, 1281, 1885, 4211, 5637
 \cs_set_eq:Nc .. 15, 1280, 1282, 1884, 4212
 \cs_set_eq:NN 15, 1280, 1280–
 1283, 1287, 1292, 1462–1469, 1473–
 1480, 1872, 1874, 1883, 2622, 2882,
 2886, 2907, 2909, 2968, 2986, 3087,
 3176, 3177, 3180, 4210, 5636, 6146,
 6147, 6383, 8913, 9006, 9835, 10757
 \cs_set_eq:NwN 782,
 783–819, 824, 825, 838, 839, 1157, 1280
 \cs_set_nopar:cn 13, 1397
 \cs_set_nopar:cpn 10, 1241, 1243
 \cs_set_nopar:cpx 10, 1241, 1244
 \cs_set_nopar:cx 13, 1397
 \cs_set_nopar:Nn 13, 1356
 \cs_set_nopar:Npn
 .. 10, 824, 824, 826–828, 830–832,
 835, 895, 897, 1063, 1070, 1189,
 1243, 2975, 2981, 5569, 6572, 8092
 \cs_set_nopar:Npx 10, 824, 825, 829, 833,
 837, 1244, 1543, 2888, 2893, 2910,
 2911, 4219, 4221, 4223, 4237, 4239,
 4241, 4243, 4261, 4263, 4265, 4267
 \cs_set_nopar:Nx 13, 1356
 \cs_set_protected:cn 13, 1397
 \cs_set_protected:cpn 10, 1261, 1261, 6682
 \cs_set_protected:cpx 10, 1261, 1262,
 1358, 1391, 6684, 6686, 6688, 6690
 \cs_set_protected:cx 13, 1397
 \cs_set_protected:Nn 13, 1356
 \cs_set_protected:Npn
 10, 824, 834, 853, 899,
 907, 915, 919, 922, 930, 939, 949,
 952, 956, 965, 967, 971, 979, 987,
 995, 1003, 1011, 1016, 1024, 1032,
 1040, 1045, 1047, 1261, 4312, 4363,
 4388, 4873, 5384, 5429, 5442, 5654,
 5658, 5666, 6894, 6896, 6898, 7013
 \cs_set_protected:Npx
 . 10, 824, 836, 1262, 4356, 4382, 6781
 \cs_set_protected:Nx 13, 1356
 \cs_set_protected_nopar:cn 13, 1397
 \cs_set_protected_nopar:cpn
 10, 1255, 1255
 \cs_set_protected_nopar:cpx
 10, 1255, 1256
 \cs_set_protected_nopar:cx 13, 1397
 \cs_set_protected_nopar:Nn 13, 1356
 \cs_set_protected_nopar:Npn
 10, 310, 824,
 830, 834, 836, 840, 842, 844, 846,
 848, 850, 1169, 1171, 1173, 1185,
 1187, 1192, 1202, 1213, 1215, 1223,
 1227, 1255, 6355, 6357, 7884, 7893,
 7901, 7910, 8846, 11115, 11119,
 11146, 11147, 11152, 11153, 11165
 \cs_set_protected_nopar:Npx 10,
 296, 824, 832, 1256, 6650, 6776,
 8028, 8070, 8090, 8482, 8518, 8595,
 8671, 8782, 8889, 8899, 8925, 9442,
 9455, 9542, 9740, 9753, 9937, 10242,
 10250, 10276, 10468, 10582, 10588,
 10599, 10632, 10639, 10685, 10720
 \cs_set_protected_nopar:Nx 13, 1356
 \cs_show:c 16, 821, 823, 7531

- \cs_show:N 16, 805, 810, 823, 4660
 - \cs_split_function:NN . 903, 911, 918,
926, 934, 943, 951, 960, 1053, 1054,
1072, 1078, 1097, 1099, 1301, 1770
 - \cs_split_function_aux:w 1072, 1080, 1083
 - \cs_split_function_auxii:w
..... 1072, 1091, 1093
 - \cs_tmp:w 853, 856,
859, 861, 1225, 1233–1241, 1243–
1266, 1356, 1365–1389, 1397–1420
 - \cs_to_str:N 4,
17, 21, 1063, 1063, 1081, 2272, 6476
 - \cs_to_str_aux:w 1063, 1066, 1070
 - \cs_undefine:c 16, 1296, 1298, 1515
 - \cs_undefine:N
..... 16, 1296, 1296, 1514, 6277, 6289
 - \csname . 13, 32, 35, 62, 80, 93, 96, 167,
170, 176, 184, 189, 191, 201, 204,
214, 229, 233, 269, 271, 276, 278, 443
 - \currentgrouplevel 695
 - \currentgrouptype 696
 - \currentifbranch 692
 - \currentiflevel 691
 - \currentiftype 693
- D**
- \d 1812, 2716
 - \dagger 3843, 3849
 - \day 651
 - \ddagger 3844, 3850
 - \deadcycles 585
 - \def 54, 56, 98,
104, 106, 107, 109, 112–115, 118,
126, 128–130, 133, 141–144, 147,
152, 157, 203, 213, 292, 325, 339, 350
 - .default:n 171
 - .default:V 171
 - \defaultthyphenchar 635
 - \defaultskewchar 636
 - \delcode 666
 - \delimiter 460
 - \delimiterfactor 509
 - \delimitershortfall 508
 - \deprecated 7036–7042
 - \deprectiated 2373, 2374
 - \detokenize 32, 35, 80, 93, 96,
167, 170, 176, 185, 190, 192, 198,
201, 204, 214, 269, 271, 276, 278, 683
 - \dim_add:cn 82, 3916
 - \dim_add:Nn 82, 3916, 3916, 3918, 3919
 - \dim_compare:n 3935
 - \dim_compare:nF 3974, 3989
 - \dim_compare:nNn 3930
 - \dim_compare:nNnF 4002, 4017
 - \dim_compare:nNnT
... 3905, 3907, 3909, 3911, 3994, 4011
 - \dim_compare:nNnTF 85, 2105, 3930
 - \dim_compare:nT 3966, 3983
 - \dim_compare:nTF 85, 3935
 - \dim_compare_<:nw 3935
 - \dim_compare_=:nw 3935
 - \dim_compare_>:nw 3935
 - \dim_compare_aux:wNn ... 3935, 3937, 3945
 - \dim_compare_p:n 85, 3935
 - \dim_compare_p:nNn 85, 3930
 - \dim_do_until:nn ... 86, 3964, 3986, 3990
 - \dim_do_until:nNnn .. 86, 3992, 4014, 4018
 - \dim_do_while:nn 86, 3964, 3980
 - \dim_do_while:nNnn
..... 85, 3984, 3992, 4008, 4012
 - \dim_eval:n 87, 2100, 4020, 4020
 - \dim_eval:w
... 95, 3875, 3876, 3894, 3917, 3922,
3929, 3932, 3937, 3957–3963, 4021,
5967, 5969, 5971, 5980, 5982, 5984,
5986, 6036, 6050, 6063, 6076, 6094
 - \dim_eval_end: .. 95, 3875, 3877, 3894,
3917, 3922, 3929, 3932, 3938, 4021,
5967, 5969, 5971, 5980, 5982, 5984,
5986, 6036, 6050, 6063, 6076, 6094
 - \dim_gadd:cn 83, 3916
 - \dim_gadd:Nn 83, 3916, 3918, 3920
 - .dim_gset:c 172
 - \dim_gset:cn 83, 3893
 - .dim_gset:N 172
 - \dim_gset:Nn 83, 3893, 3895, 3897, 3907, 3911
 - \dim_gset_eq:cc 83, 3898
 - \dim_gset_eq:cN 83, 3898
 - \dim_gset_eq:Nc 83, 3898
 - \dim_gset_eq:NN 83, 3898, 3901–3903
 - \dim_gset_max:cn 83, 3904
 - \dim_gset_max:Nn ... 83, 3904, 3906, 3913
 - \dim_gset_min:cn 84, 3904
 - \dim_gset_min:Nn ... 84, 3904, 3910, 3915
 - \dim_gsub:cn 84, 3916
 - \dim_gsub:Nn 84, 3916, 3923, 3925
 - \dim_gzero:c 82, 3889
 - \dim_gzero:N 82, 3889, 3890, 3892
 - \dim_new:c 82, 3878

<code>\dim_new:N</code>	82, <u>3878</u> , 3882, 3888, 4027, 4028, 4035–4039, 8098
<code>\dim_ratio:nn</code>	84, <u>3926</u> , 3926
<code>\dim_ratio_aux:n</code>	<u>3926</u> , 3927, 3928
<code>.dim_set:c</code>	172
<code>\dim_set:cn</code>	83, <u>3893</u>
<code>.dim_set:N</code>	172
<code>\dim_set:Nn</code>	83, <u>3893</u> , 3893, 3895, 3896, 3905, 3909, 4029
<code>\dim_set_eq:cc</code>	83, <u>3898</u>
<code>\dim_set_eq:cN</code>	83, <u>3898</u>
<code>\dim_set_eq:Nc</code>	83, <u>3898</u>
<code>\dim_set_eq:NN</code>	83, <u>3898</u> , 3898–3900
<code>\dim_set_max:cn</code>	83, <u>3904</u>
<code>\dim_set_max:Nn</code>	83, <u>3904</u> , 3904, 3912
<code>\dim_set_min:cn</code>	84, <u>3904</u>
<code>\dim_set_min:Nn</code>	84, <u>3904</u> , 3908, 3914
<code>\dim_show:c</code>	87, <u>4024</u>
<code>\dim_show:N</code>	87, <u>4024</u> , 4024, 4025
<code>\dim_sub:cn</code>	84, <u>3916</u>
<code>\dim_sub:Nn</code>	84, <u>3916</u> , 3921, 3923, 3924
<code>\dim_until_do:nn</code> ...	86, <u>3964</u> , 3972, 3977
<code>\dim_until_do:nNnn</code> ..	86, <u>3992</u> , 4000, 4005
<code>\dim_use:c</code>	87, <u>4022</u>
<code>\dim_use:N</code>	87, 3937, 4021, <u>4022</u> , 4022, 4023, 8062
<code>\dim_while_do:nn</code> ...	87, <u>3964</u> , 3964, 3969
<code>\dim_while_do:nNnn</code> ..	86, <u>3992</u> , 3992, 3997
<code>\dim_zero:c</code>	82, <u>3889</u>
<code>\dim_zero:N</code>	82, <u>3889</u> , 3889–3891
<code>\dimen</code>	657
<code>\dimendef</code>	356
<code>\dimexpr</code>	710
<code>\directlua</code>	15, 756
<code>\discretionary</code>	520
<code>\displayindent</code>	485
<code>\displaylimits</code>	495
<code>\displaystyle</code>	473
<code>\displaywidowpenalties</code>	723
<code>\displaywidowpenalty</code>	484
<code>\displaywidth</code>	486
<code>\divide</code>	363
<code>\doublehyphendemerits</code>	553
<code>\dp</code>	664
<code>\dump</code>	647
E	
<code>\E</code>	1818, 2718
<code>\edef</code>	33, 68, 82, 164, 166, 181, 201, 266, 273, 351
<code>\else</code> ..	14, 63, 117, 137, 172, 187, 207, 404
<code>\else:</code>	25, <u>784</u> , 787, 883, 1087, 1104, 1107, 1116, 1122, 1132, 1135, 1144, 1150, 1306, 1340, 1424, 1487, 1492, 1564, 1833, 1901, 1915, 1929, 1939, 1950, 1953, 1963, 1966, 1979, 1988, 2241, 2243, 2245, 2249, 2399, 2428, 2442, 2450, 2463, 2472, 2604, 2609, 2614, 2619, 2626, 2632, 2637, 2642, 2647, 2652, 2657, 2662, 2667, 2672, 2692, 2700, 2707, 2741, 2752, 2763, 2774, 2836, 2845, 2853, 2862, 2928, 2936, 2958, 3198, 3209, 3219, 3224, 3227, 3230, 3326, 3337, 3345, 3353, 3361, 3369, 3377, 3385, 3393, 3401, 3409, 3933, 3940, 4082, 4431, 4442, 4454, 4464, 4480, 4576, 4637, 4646, 4658, 5433, 5446, 5754, 5991, 5993, 6003, 6470, 6483, 7791, 7820, 7866, 7877, 7927, 7933, 7943, 8035, 8077, 8120, 8124, 8141, 8185, 8193, 8196, 8205, 8233, 8252, 8261, 8329, 8332, 8349, 8352, 8368, 8371, 8394, 8397, 8400, 8403, 8406, 8409, 8412, 8415, 8418, 8433, 8436, 8439, 8442, 8445, 8448, 8451, 8454, 8457, 8489, 8525, 8554, 8568, 8573, 8624, 8662, 8678, 8703, 8726, 8736, 8796, 8799, 8894, 8904, 8939, 8942, 8978, 8980, 8983, 9029, 9034, 9055, 9231, 9303, 9317, 9352, 9377, 9397, 9430, 9447, 9451, 9470, 9485, 9527, 9539, 9556, 9571, 9599, 9601, 9611, 9627, 9632, 9647, 9684, 9690, 9695, 9728, 9745, 9749, 9766, 9778, 9781, 9784, 9793, 9797, 9824, 9827, 9852, 9922, 9934, 9945, 9961, 9966, 9971, 9976, 9984, 10000, 10014, 10020, 10045, 10064, 10099, 10107, 10131, 10134, 10177, 10183, 10188, 10241, 10249, 10273, 10287, 10290, 10304, 10322, 10328, 10368, 10376, 10404, 10482, 10490, 10513, 10542, 10548, 10587, 10594, 10604, 10616, 10630, 10638, 10651, 10665, 10674, 10680, 10764, 10772, 10822, 10826, 10830, 10834, 10849, 10853, 10858, 10865, 10869, 10879, 10883, 10886, 10897, 10901, 10905, 10910, 10915, 10929, 10933, 10937, 10942, 10947

<code>\emergencystretch</code>	568	<code>\etex_ifcsname:D</code>	672, 801
<code>\end</code>	442	<code>\etex_ifdefined:D</code>	671, 800
<code>\EndCatcodeRegime</code>	11153	<code>\etex_iffontchar:D</code>	701
<code>\endcsname</code>	13, 32, 35, 62, 80, 93, 96, 167, 170, 176, 185, 190, 192, 201, 204, 214, 229, 233, 269, 271, 276, 278, 444	<code>\etex_interactionmode:D</code>	699
<code>\endgroup</code>	12, 61, 111, 120, 228, 232, 377	<code>\etex_interlinepenalties:D</code>	720
<code>\endinput</code>	264, 416	<code>\etex_lastlinefit:D</code>	719
<code>\endL</code>	731	<code>\etex_lastnodetype:D</code>	700, 2259, 2260
<code>\endlinechar</code>	79, 92, 289, 458	<code>\etex_marks:D</code>	676
<code>\endR</code>	733	<code>\etex_middle:D</code>	724
<code>seq_push:cn</code>	118	<code>\etex_muexpr:D</code>	712, 4134, 4145, 4150, 4155
<code>\eqno</code>	478	<code>\etex_mutoglua:D</code>	718
<code>\errhelp</code>	250, 424	<code>\etex_numexpr:D</code>	709, 3177
<code>\errmessage</code>	418	<code>\etex_pagediscards:D</code>	727
<code>\erroneous</code>	1575	<code>\etex_parshapedimen:D</code>	708
<code>\ERROR</code>	2371, 2372	<code>\etex_parshapeindent:D</code>	706
<code>\errorcontextlines</code>	425	<code>\etex_parshapelength:D</code>	707
<code>\errorstopmode</code>	439	<code>\etex_predisplaydirection:D</code>	734
<code>\escapechar</code>	457	<code>\etex_protected:D</code>	736, 819
<code>\etex_beginL:D</code>	730	<code>\etex_readline:D</code>	686, 6494, 6496
<code>\etex_beginR:D</code>	732	<code>\etex_savinghyphcodes:D</code>	725
<code>\etex_botmarks:D</code>	679	<code>\etex_savingvdiscards:D</code>	726
<code>\etex_clubpenalties:D</code>	721	<code>\etex_scantokens:D</code>	684, 4306, 4331, 4343
<code>\etex_currentgrouplevel:D</code>	695	<code>\etex_showgroups:D</code>	697
<code>\etex_currentgrouptype:D</code>	696, 2257	<code>\etex_showifs:D</code>	698
<code>\etex_currentifbranch:D</code>	692	<code>\etex_showtokens:D</code>	685, 4662, 5077, 5540, 6306, 6328
<code>\etex_currentiflevel:D</code>	691	<code>\etex_splitbotmarks:D</code>	681
<code>\etex_currentifttype:D</code>	693	<code>\etex_splitdiscards:D</code>	728
<code>\etex_detokenize:D</code>	683, 1831, 2408, 2421, 4568, 4569	<code>\etex_splitfirstmarks:D</code>	680
<code>\etex_dimexpr:D</code>	710, 3876	<code>\etex_TeXXETstate:D</code>	729
<code>\etex_displaywidowpenalties:D</code>	723	<code>\etex_topmarks:D</code>	677
<code>\etex_endL:D</code>	731	<code>\etex_tracingassigns:D</code>	687
<code>\etex_endR:D</code>	733	<code>\etex_tracinggroups:D</code>	694
<code>\etex_eTeXrevision:D</code>	675	<code>\etex_tracingifs:D</code>	690
<code>\etex_eTeXversion:D</code>	674	<code>\etex_tracingnesting:D</code>	689
<code>\etex_everyeof:D</code>	735, 4302, 4328, 4340	<code>\etex_tracingscantokens:D</code>	688
<code>\etex_firstmarks:D</code>	678	<code>\etex_unexpanded:D</code>	682, 804, 1753, 1755, 1758, 1763, 4611
<code>\etex_fontcharhp:D</code>	703	<code>\etex_unless:D</code>	673, 789
<code>\etex_fontcharht:D</code>	702	<code>\etex_widowpenalties:D</code>	722
<code>\etex_fontcharic:D</code>	705	<code>\eTeXrevision</code>	675
<code>\etex_fontcharwd:D</code>	704	<code>\eTeXversion</code>	674
<code>\etex_glueexpr:D</code>	711, 4056, 4067, 4072, 4097, 4102, 4105, 8057	<code>\everycr</code>	386
<code>\etex_glueshrink:D</code>	714, 4170	<code>\everydisplay</code>	487
<code>\etex_glueshrinkorder:D</code>	716, 4091	<code>\everyeof</code>	735
<code>\etex_gluestretch:D</code>	713, 4169	<code>\everyhbox</code>	626
<code>\etex_gluestretchorder:D</code>	715, 4090	<code>\everyjob</code>	31, 655
<code>\etex_gluetomu:D</code>	717	<code>\everymath</code>	511
		<code>\everypar</code>	574
		<code>\everyvbox</code>	627

- `\exhyphenpenalty` 550
- `[EXP]\cs_get_function_signature:N` 21
- `\exp_after:wN`
 - . 34, 802, 802, 820, 857, 896, 898,
 - 982, 1050, 1068, 1080, 1086, 1088,
 - 1115, 1117, 1120, 1143, 1145, 1148,
 - 1305, 1307, 1310, 1361, 1393, 1524,
 - 1531, 1533, 1536, 1537, 1544, 1548,
 - 1549, 1554, 1555, 1560, 1565, 1567,
 - 1570, 1575, 1577, 1579, 1581, 1583,
 - 1585, 1587, 1590–1592, 1596, 1599,
 - 1604, 1609–1611, 1615–1617, 1621–
 - 1623, 1627–1629, 1633–1635, 1639–
 - 1642, 1646–1649, 1653–1655, 1660–
 - 1663, 1667–1670, 1702, 1703, 1706,
 - 1709, 1710, 1714, 1715, 1718, 1720,
 - 1721, 1723, 1726, 1727, 1732, 1733,
 - 1737, 1742–1744, 1748, 1750, 1752,
 - 1753, 1755, 1758, 1763, 1766, 1778,
 - 1806, 1826, 1831, 1832, 1834, 1850,
 - 1949, 1952, 1954, 1962, 1965, 1967,
 - 1972, 1973, 1976, 1985, 1993, 1995–
 - 1997, 2000, 2005, 2138, 2285, 2392,
 - 2398, 2400, 2408, 2414, 2421, 2427,
 - 2429, 2685, 2706, 2725, 2732, 2742,
 - 2753, 2764, 2775, 2784, 2792, 2800,
 - 2820, 2843, 2844, 2846, 2852, 2855,
 - 2927, 2929, 2935, 2937, 2950, 2957,
 - 2959, 3048, 3057, 3066, 3320, 3323,
 - 3616, 3626, 3759, 3929, 3937, 4306,
 - 4343, 4370, 4399, 4440, 4451, 4452,
 - 4526, 4527, 4569, 4575, 4577, 4611,
 - 4615, 4617, 4635, 4643, 4644, 4656,
 - 4667, 4695–4697, 4700, 4712, 4877,
 - 4894, 4907, 4926, 4927, 4955, 4956,
 - 4978, 4983, 5077, 5078, 5092, 5098,
 - 5119, 5126, 5194–5196, 5203, 5204,
 - 5321, 5330, 5391, 5392, 5395–5397,
 - 5540, 5541, 5600, 5611, 5661, 5772,
 - 5797, 5833, 5834, 5883, 6306, 6307,
 - 6328, 6329, 6462, 6469, 6472, 6631,
 - 7071, 7073, 7114, 7118, 7238, 7335,
 - 7604, 7616, 7683, 7786, 7806, 7811,
 - 7815, 7819, 7822, 7826, 7829, 7848,
 - 7867, 7876, 7878, 7888, 7890, 7905,
 - 7907, 7919, 7934, 7942, 7944, 7951,
 - 7954, 7966, 7976, 7979, 7991, 8040,
 - 8061, 8082, 8111, 8119, 8122, 8125,
 - 8140, 8142, 8176, 8184, 8186, 8204,
 - 8206, 8243, 8251, 8253, 8260, 8262,
 - 8328, 8331, 8333, 8345, 8364, 8479,
 - 8494, 8515, 8530, 8544, 8583, 8603,
 - 8629, 8634, 8635, 8661, 8663, 8683,
 - 8804, 8852, 8863, 8905, 8947, 8963,
 - 8969, 8977, 8984–8986, 9193, 9195,
 - 9205, 9220, 9223, 9253, 9256, 9267,
 - 9270, 9286, 9302, 9308, 9316, 9322–
 - 9324, 9396, 9408, 9435, 9452, 9490,
 - 9501, 9503, 9505, 9507, 9532, 9540,
 - 9576, 9587, 9589, 9591, 9593, 9639,
 - 9654, 9708, 9733, 9750, 9765, 9769,
 - 9794, 9798, 9825, 9828, 9857, 9927,
 - 9935, 9958–9960, 9962–9964, 9968–
 - 9970, 9977, 9983, 9989, 9999, 10003,
 - 10016–10018, 10022, 10023, 10036,
 - 10081, 10149, 10201, 10240, 10247,
 - 10255, 10266, 10274, 10309, 10326,
 - 10364, 10367, 10403, 10405, 10416,
 - 10424, 10441, 10489, 10491, 10503,
 - 10506, 10553, 10605, 10615, 10627–
 - 10629, 10650, 10659–10663, 10667–
 - 10672, 10676–10678, 10681, 10693
- `\exp_arg_last_unbraced:nn`
 - . . . 1699, 1699, 1702, 1706, 1709, 1714
- `\exp_arg_next:nnn` 1525,
 - 1525, 1533, 1536, 1544, 1548, 1554
- `\exp_arg_next_nobraces:nnn` 1525, 1526, 1531
- `\exp_args:cc` 30, 1582, 1582
- `\exp_args:Nc` 30, 820,
 - 820–823, 989, 997, 1005, 1013, 1214,
 - 1224, 1242, 1281, 1289, 1294, 1317,
 - 1350, 1426–1429, 1447, 1582, 4540
- `\exp_args:Ncc` 32, 1283,
 - 1291, 1295, 1434–1437, 1582, 1586
- `\exp_args:Nccc` 33, 1582, 1588
- `\exp_args:Ncco` 33, 1644, 1665
- `\exp_args:Nccx` 33, 1687, 1696
- `\exp_args:Ncf` . . . 32, 1607, 1631, 2208, 2209
- `\exp_args:NcNc` 33, 1644, 1651
- `\exp_args:NcNo` 33, 1644, 1658
- `\exp_args:Ncnx` 33, 1687, 1697
- `\exp_args:Nco` 32, 1607, 1625
- `\exp_args:Ncx` 32, 1673, 1682
- `\exp_args:Nf` 31, 1595,
 - 1595, 2090, 2100, 2178, 2179, 2187,
 - 2196, 3474, 3476, 3480, 3689, 3699,
 - 3710, 3721, 5187, 5407, 5575, 5587
- `\exp_args:Nff` 32, 1673, 1675, 3566
- `\exp_args:Nfo` 32, 1673, 1674

- \exp_args:Nnc
 . 32, 1053, 1054, 1282, 1290, 1293,
 1430–1433, 1582, 1584, 1780, 5491
- \exp_args:Nnc 32, 1673, 1673, 5501
- \exp_args:NNf
 32, 1607, 1607, 2219, 2228, 7112, 7147
- \exp_args:Nnf 32, 932,
 941, 950, 958, 1673, 1676, 3551, 3555
- \exp_args:Nnnc 33, 1687, 1689
- \exp_args:NNNo . 33, 1577, 1580, 4307, 4332
- \exp_args:NNno 33, 1687, 1687
- \exp_args:Nnno 33, 1687, 1690
- \exp_args:NNNV 33, 1644, 1644
- \exp_args:NNnx 33, 1687, 1692
- \exp_args:Nnnx 33, 1687, 1694
- \exp_args:NNo
 32, 1577, 1578, 5655, 6398, 7334
- \exp_args:Nno
 32, 1673, 1677, 3040, 3944, 5670, 7514
- \exp_args:NNoo 33, 1687, 1688
- \exp_args:NNox 33, 1687, 1693
- \exp_args:Nnox 33, 1687, 1695
- \exp_args:NNV 32, 1607, 1619
- \exp_args:NNv 32, 1607, 1613
- \exp_args:NnV 32, 1673, 1678
- \exp_args:NNx 32, 1673, 1681
- \exp_args:Nnx 32, 1673, 1683
- \exp_args:No 30, 1577,
 1577, 4302, 4340, 4344, 4487–4489,
 4518, 4559, 5414, 5416, 5570, 6458
- \exp_args:Noc 32, 1673, 1679
- \exp_args:Noo 32, 1673, 1680
- \exp_args:Nooo 33, 1687, 1691
- \exp_args:Noox 33, 1687, 1698
- \exp_args:Nox 32, 1673, 1684
- \exp_args:NV 31, 1595, 1602
- \exp_args:Nv 31, 1595, 1597
- \exp_args:NVV 32, 1607, 1637
- \exp_args:Nx 31, 1595, 1672, 1672
- \exp_args:Nxo 32, 1673, 1685
- \exp_args:Nxx 32, 1673, 1686
- \exp_eval_error_msg:w .. 1558, 1562, 1574
- \exp_eval_register:c . 36, 1555, 1558,
 1569, 1600, 1617, 1715, 1720, 1764
- \exp_eval_register:N
 36, 1549, 1558, 1558,
 1570, 1605, 1623, 1641, 1642, 1649,
 1710, 1718, 1728, 1734, 1745, 1759
- \exp_last_two_unbraced:Noo 34, 1749, 1749
- \exp_last_two_unbraced_aux:nnN
 1750, 1751
- \exp_last_unbraced:NcV 34, 1717, 1724, 4550
- \exp_last_unbraced:Nf 34, 1717, 1722, 2271
- \exp_last_unbraced:Nfo 34, 1717, 1739, 5168
- \exp_last_unbraced:NNNo . 34, 1717, 1747
- \exp_last_unbraced:NNNV . 34, 1717, 1740
- \exp_last_unbraced:NNo
 34, 1717, 1736, 5437, 5465, 5791
- \exp_last_unbraced:Nno 5877
- \exp_last_unbraced:NNV .. 34, 1717, 1730
- \exp_last_unbraced:No ... 34, 1717, 1721
- \exp_last_unbraced:Noo . 1717, 1738, 5764
- \exp_last_unbraced:NV ... 34, 1717, 1717
- \exp_last_unbraced:Nv ... 34, 1717, 1719
- \exp_not:c 35, 1766, 1766, 1791, 1848,
 2297, 2301, 2306, 2310, 2313, 2315–
 2317, 2322, 2324–2326, 2331, 2333–
 2335, 2340, 2342–2344, 2349, 2351,
 2353, 2355, 2357, 2359, 2361, 2363,
 2365–2367, 2987, 6685, 6687, 6689,
 6691, 7096, 7128, 7256, 7258, 7352
- \exp_not:f 35, 1753, 1754
- \exp_not:N 34, 802,
 803, 1360–1362, 1392–1394, 1524,
 1560, 1766, 1791, 2302, 2305, 2309,
 2313, 2322, 2331, 2340, 2461, 2470,
 2599, 2603, 2608, 2613, 2618, 2625,
 2631, 2636, 2641, 2646, 2651, 2656,
 2666, 2671, 2699, 2706, 2890, 2895,
 2913, 2926, 2956, 4312, 4314, 4328,
 4358, 4360, 4384, 4391, 4396, 4621,
 4644, 4656, 4657, 5027, 6394, 6396,
 6778, 6786, 6787, 6789, 7115, 7148,
 7256, 7258, 7340, 7352, 8031, 8073,
 8092, 8485, 8521, 8598, 8785, 8892,
 8902, 9445, 9458, 9545, 9743, 9756,
 9940, 10245, 10253, 10279, 10472,
 10474, 10476, 10723, 10725, 10727,
 10729, 10731, 10960, 10962, 10964,
 10966, 10968, 10970, 10972, 10974
- \exp_not:n 35, 802, 804, 1485, 1524, 2298,
 2461, 2470, 2891, 2896, 2910, 2914,
 2986, 2988, 4189, 4219, 4225, 4237,
 4245, 4261, 4269, 4360, 4361, 4366,
 4367, 4386, 4394–4396, 4856, 4953,
 5028, 5084, 5238, 5547, 5598, 5609,
 5723, 5724, 5745, 5842, 6315, 6350,
 6354, 6498, 6779, 6783, 6790, 7120,
 7294, 7295, 7369, 10986, 11080, 11083

- \exp_not:o 35, 1753, 1753,
4221, 4227, 4237, 4239, 4241, 4243,
4245, 4247, 4249, 4251, 4261, 4263,
4265, 4267, 4269, 4271, 4273, 4275,
4800, 4802, 4852, 5285, 5287, 5291,
6651, 7097, 7129, 7131, 7138, 7149
 - \exp_not:V
35, 1753, 1756, 4239, 4247, 4263, 4271
 - \exp_not:v 35, 1753, 1761, 7297
 - \exp_stop_f: ... 35, 1534, 1540, 2272, 4607
 - \expandafter 12, 13, 31, 35,
61, 62, 64, 96, 136, 138, 166, 169,
175, 179, 183, 184, 188, 189, 191,
201, 203, 206, 208, 213, 228, 229,
232, 233, 264, 268, 270, 275, 277, 374
 - \expl_status_pop:w 200
 - \ExplFileDate 49, 112, 142, 144, 334, 779,
1521, 1866, 2379, 2484, 3173, 3872,
4177, 4771, 5253, 5629, 5913, 6099,
6511, 7047, 7173, 7599, 7712, 11066
 - \ExplFileDescription 113, 130, 334, 779,
1521, 1866, 2379, 2484, 3173, 3872,
4177, 4771, 5253, 5629, 5913, 6099,
6511, 7047, 7173, 7599, 7712, 11066
 - \ExplFileName 114, 128, 334, 779,
1521, 1866, 2379, 2484, 3173, 3872,
4177, 4771, 5253, 5629, 5913, 6099,
6511, 7047, 7173, 7599, 7712, 11066
 - \ExplFileVersion 49, 115, 129, 334, 779,
1521, 1866, 2379, 2484, 3173, 3872,
4177, 4771, 5253, 5629, 5913, 6099,
6511, 7047, 7173, 7599, 7712, 11066
 - \ExplSyntaxNamesOff 5, 266, 273
 - \ExplSyntaxNamesOn 5, 266, 266
 - \ExplSyntaxOff 4, 5,
67, 68, 178, 186, 208, 291, 296, 310, 325
 - \ExplSyntaxOn 4,
5, 67, 82, 150, 155, 160, 206, 291, 292
- F**
- \F 2679, 2713, 2812
 - \fam 366
 - \fi 44, 65,
123, 139, 174, 194, 209, 231, 265, 405
 - \fi: 25, 784, 788, 882–
885, 983, 1051, 1067, 1071, 1089,
1109, 1110, 1118, 1124, 1137, 1138,
1146, 1152, 1211, 1308, 1346, 1424,
1487, 1492, 1563, 1566, 1575, 1779,
1807, 1835, 1851, 1903, 1917, 1929,
1939, 1955, 1956, 1968, 1969, 1981,
1990, 2241, 2243, 2245, 2249, 2252,
2254, 2393, 2401, 2415, 2430, 2444,
2452, 2465, 2474, 2604, 2609, 2614,
2619, 2626, 2632, 2637, 2642, 2647,
2652, 2657, 2662, 2667, 2672, 2694,
2700, 2707, 2744, 2755, 2766, 2777,
2838, 2847, 2856, 2864, 2930, 2938,
2960, 3189, 3200, 3211, 3226, 3232,
3233, 3235, 3328, 3332, 3339, 3347,
3355, 3363, 3371, 3379, 3387, 3395,
3403, 3411, 3933, 3942, 3952, 4084,
4433, 4444, 4456, 4466, 4483, 4578,
4639, 4648, 4658, 4706, 4712, 4848,
4851, 4878, 4954, 4958, 4979, 5093,
5435, 5448, 5756, 5773, 5798, 5884,
5991, 5993, 6003, 6473, 6485, 7793,
7830, 7831, 7863, 7868, 7879, 7891,
7908, 7932, 7935, 7945, 7955, 7980,
8037, 8079, 8117, 8126, 8127, 8143,
8182, 8187, 8198, 8199, 8207, 8235,
8249, 8254, 8263, 8334, 8335, 8351,
8355, 8370, 8375, 8396, 8399, 8402,
8405, 8408, 8411, 8414, 8417, 8420,
8435, 8438, 8441, 8444, 8447, 8450,
8453, 8456, 8459, 8480, 8491, 8516,
8527, 8558, 8570, 8578–8580, 8584,
8626, 8664, 8680, 8706, 8721, 8735,
8738, 8798, 8801, 8906, 8907, 8941,
8944, 8971, 8972, 8987–8989, 8999,
9032, 9037, 9047, 9051, 9059, 9060,
9181, 9196, 9224, 9239, 9257, 9301,
9309, 9325–9327, 9340, 9344, 9365,
9366, 9375, 9384, 9410, 9411, 9432,
9454, 9461, 9474, 9487, 9508, 9529,
9541, 9560, 9573, 9594, 9598, 9603,
9604, 9631, 9636, 9640, 9655, 9688,
9694, 9702, 9706, 9707, 9709, 9730,
9752, 9759, 9770, 9780, 9786, 9787,
9796, 9799, 9826, 9829, 9854, 9924,
9936, 9947, 9965, 9974, 9975, 9978,
9990, 10019, 10024, 10025, 10047,
10056, 10067, 10076, 10116, 10117,
10129, 10137, 10138, 10181, 10187,
10195, 10199, 10200, 10202, 10248,
10256, 10275, 10293, 10294, 10306,
10324, 10333, 10357, 10370, 10384,
10406, 10412, 10417, 10425, 10431,
10434, 10485, 10492, 10507, 10521,
10546, 10552, 10554, 10593, 10607,

10608, 10637, 10644, 10645, 10673,
 10679, 10683, 10684, 10766, 10774,
 10825, 10829, 10833, 10837, 10856,
 10857, 10868, 10871, 10872, 10888–
 10890, 10918–10922, 10950–10954
 \file_add_path:nN
 178, 7627, 7627, 7665, 7672
 \file_add_path_search:nN 7627, 7631, 7637
 \file_if_exist:n 7663
 \file_if_exist:nTF 177, 7663
 \file_input:n 178, 7670, 7670
 \file_list: 178, 7694, 7694
 \file_path_include:n ... 178, 7687, 7687
 \file_path_remove:n 178, 7687, 7692
 \finalhyphdemerits 554
 \firstmark 452
 \firstmarks 678
 \floatingpenalty 599
 \font 365
 \fontchardp 703
 \fontcharht 702
 \fontcharic 705
 \fontcharwd 704
 \fontdimen 632
 \fontname 456
 \fp_abs:c 184, 8586
 \fp_abs:N 184, 8586, 8586, 8588
 \fp_abs_aux:NN ... 8586, 8586, 8587, 8590
 \fp_add:cn 185, 8637
 \fp_add:Nn 185, 8637, 8637, 8639
 \fp_add:NNNNNNNN
 9023, 9023, 10351, 10403, 10489
 \fp_add_aux:NNn .. 8637, 8637, 8638, 8641
 \fp_add_core: 8637, 8651, 8654, 8755
 \fp_add_difference: 8637, 8663, 8708
 \fp_add_sum: 8637, 8661, 8691
 \fp_compare:n 10980
 \fp_compare:NNN 10793
 \fp_compare:nNn 10776
 \fp_compare:NNNTF 10776
 \fp_compare:nNnTF
 184, 10776, 10994, 11000,
 11006, 11012, 11018, 11024, 11030
 \fp_compare:nTF 184, 10980
 \fp_compare_<: 10776
 \fp_compare_<_aux: 10776
 \fp_compare_>: 10776
 \fp_compare_absolute_a<b: 10776
 \fp_compare_absolute_a>b: 10776
 \fp_compare_aux:N
 10776, 10791, 10802, 10804
 \fp_compare_aux_i:w . 10980, 10986, 10990
 \fp_compare_aux_ii:w 10980, 10993, 10996
 \fp_compare_aux_iii:w 10980, 10999, 11002
 \fp_compare_aux_iv:w 10980, 11005, 11008
 \fp_compare_aux_v:w . 10980, 11011, 11014
 \fp_compare_aux_vi:w 10980, 11017, 11020
 \fp_compare_aux_vii:w 10980, 11023, 11026
 \fp_const:cn 180, 8004
 \fp_const:Nn 180, 8004, 8004, 8009
 \fp_cos:cn 188, 9510
 \fp_cos:Nn 188, 9510, 9510, 9512
 \fp_cos_aux:NNn .. 9510, 9510, 9511, 9514
 \fp_cos_aux_i: 9510, 9540, 9550
 \fp_cos_aux_ii: .. 9510, 9553, 9583, 9788
 \fp_div:cn 186, 8871
 \fp_div:Nn 186, 8871, 8871, 8873
 \fp_div_aux:NNn .. 8871, 8871, 8872, 8875
 \fp_div_divide: .. 8871, 8959, 8974, 9000
 \fp_div_divide_aux: 8871, 8977, 8986, 8991
 \fp_div_integer:NNNNN 9166,
 9166, 9617, 9668, 9673, 10164, 10535
 \fp_div_internal:
 8871, 8905, 8910, 10453, 10711
 \fp_div_loop: 8871, 8915, 8956, 8970, 9837
 \fp_div_loop_step:w 8963, 9017
 \fp_div_store:
 .. 8871, 8913, 8960, 9002, 9006, 9835
 \fp_div_store_decimal: . 8871, 9006, 9008
 \fp_div_store_integer:
 8871, 8913, 9003, 9835
 \fp_exp:cn 187, 9904
 \fp_exp:Nn 187, 9904, 9904, 9906
 \fp_exp_aux: 9904, 9960, 9970, 9980
 \fp_exp_aux:NNn .. 9904, 9904, 9905, 9908
 \fp_exp_const:cx 9904, 9972, 10012, 10144
 \fp_exp_const:Nx 9904, 10204, 10209, 10757
 \fp_exp_decimal: 9904, 9989, 10077, 10092
 \fp_exp_integer: 9904, 9983, 9992
 \fp_exp_integer_const:n
 9904, 10015, 10021,
 10044, 10046, 10063, 10065, 10079
 \fp_exp_integer_const:nnnn
 9904, 10082, 10085, 10442
 \fp_exp_integer_tens:
 9904, 9999, 10018, 10023, 10027
 \fp_exp_integer_units: 9904, 10057, 10059
 \fp_exp_internal: 9904, 9935, 9952, 10758

\fp_exp_overflow_msg:	\fp_gset:Nn ... 181, 8007, 8016, 8017, 8049
..... 9964, 9977, 11040, 11046	\fp_gset_eq:cc 180, 8100, 8107
\fp_exp_Taylor: 9904, 10122, 10156, 10201	\fp_gset_eq:cN 180, 8100, 8105
\fp_extended_normalise:	\fp_gset_eq:Nc 180, 8100, 8106
. 9183, 9183, 9296, 9955, 10620, 10655	\fp_gset_eq:NN 180, 8100, 8104
\fp_extended_normalise_aux:NNNNNNNN	\fp_gset_from_dim:cn 181, 8050
..... 9183	\fp_gset_from_dim:Nn 181, 8050, 8052, 8097
\fp_extended_normalise_aux_i:	\fp_gsin:cn 187, 9413
..... 9183, 9185, 9188, 9195	\fp_gsin:Nn 187, 9413, 9414, 9416
\fp_extended_normalise_aux_i:w	\fp_gsub:cn 186, 8740
..... 9183, 9193, 9198	\fp_gsub:Nn 186, 8740, 8741, 8743
\fp_extended_normalise_aux_ii:	\fp_gtan:cn 188, 9711
..... 9183, 9186, 9216, 9223	\fp_gtan:Nn 188, 9711, 9712, 9714
\fp_extended_normalise_aux_ii:w	\fp_gzero:c 180, 8010
..... 9183, 9205, 9208	\fp_gzero:N 180, 8010, 8012, 8015
\fp_extended_normalise_ii_aux:NNNNNNNN	\fp_if_undefined:N 10760
..... 9221, 9226	\fp_if_undefined:NTF 183, 10760
\fp_extended_normalise_output:	\fp_if_undefined_p:N 183, 10760
9249, 9249, 9256, 10055, 10075, 10747	\fp_if_zero:N 183, 10768
\fp_extended_normalise_output_aux:N	\fp_if_zero:NTF 10768
..... 9249, 9277, 9279	\fp_if_zero_p:N 10768
\fp_extended_normalise_output_aux_i:NNNNNNNN	\fp_level_input_exponents:
..... 9249, 9254, 9259 7939, 7939, 8656
\fp_extended_normalise_output_aux_ii:NNNNNNNN	\fp_level_input_exponents_a:
..... 9249, 9270, 9273 7939, 7942, 7947, 7954
\fp_gabs:c 185, 8586	\fp_level_input_exponents_a:NNNNNNNN
\fp_gabs:N 185, 8586, 8587, 8589 7939, 7952, 7957
\fp_gadd:cn 185, 8637	\fp_level_input_exponents_b:
\fp_gadd:Nn 185, 8637, 8638, 8640 7939, 7944, 7972, 7979
\fp_gcos:cn 188, 9510	\fp_level_input_exponents_b:NNNNNNNN
\fp_gcos:Nn 188, 9510, 9511, 9513 7939, 7977, 7982
\fp_gdiv:cn 186, 8871	\fp_ln:cn 187, 10222
\fp_gdiv:Nn 186, 8871, 8872, 8874	\fp_ln:Nn 187, 10222, 10222, 10224
\fp_gexp:cn 187, 9904	\fp_ln_aux: 10222, 10240, 10259
\fp_gexp:Nn 187, 9904, 9905, 9907	\fp_ln_aux:NNn 10222, 10222, 10223, 10226
\fp_gln:cn 187, 10222	\fp_ln_const:nn 10339, 10349, 10400, 10439
\fp_gln:Nn 187, 10222, 10223, 10225	\fp_ln_error_msg:
\fp_gmul:cn 186, 8758 10247, 10255, 11048, 11051
\fp_gmul:Nn 186, 8758, 8759, 8761	\fp_ln_exponent: ... 10222, 10274, 10283
\fp_gneg:c 185, 8611	\fp_ln_exponent_tens: 10222
\fp_gneg:N 185, 8611, 8612, 8614	\fp_ln_exponent_tens:NN .. 10326, 10336
\fp_gpow:cn 186, 10556	\fp_ln_exponent_units: 10222, 10334, 10346
\fp_gpow:Nn 186, 10556, 10557, 10559	\fp_ln_fixed: . 10222, 10454, 10499, 10506
\fp_ground_figures:cn 183, 8467	\fp_ln_fixed_aux:NNNNNNNN
\fp_ground_figures:Nn 183, 8467, 8470, 8472 10222, 10504, 10509
\fp_ground_places:cn 183, 8502	\fp_ln_integer_const:nn 10222
\fp_ground_places:Nn 183, 8502, 8505, 8507	\fp_ln_internal: 10222, 10285, 10317, 10719
.fp_gset:c 172	\fp_ln_mantissa: ... 10222, 10358, 10394
\fp_gset:cn 181, 8016	\fp_ln_mantissa_aux:
.fp_gset:N 172 10222, 10398, 10419, 10424

\fp_ln_mantissa_divide_two: 10222, 10423, 10427
\fp_ln_normalise: 10222, 10350, 10360, 10367, 10401, 10487
\fp_ln_normalise_aux:NNNNNNNN 10365, 10372
\fp_ln_nornalise_aux:NNNNNNNN .. 10222
\fp_ln_Taylor: 10222, 10416, 10445
\fp_ln_Taylor_aux: 10222, 10467, 10524, 10553
\fp_mul:cn 186, 8758
\fp_mul:Nn 186, 8758, 8758, 8760
\fp_mul:NNNNNN 9062, 9062, 9613, 9660, 9664, 10160, 10462, 10527
\fp_mul:NNNNNNNN 9106, 9106, 10048, 10068, 10123, 10736
\fp_mul_aux:NNn .. 8758, 8758, 8759, 8762
\fp_mul_end_level: 8758, 8829, 8833, 8836, 8840, 8860, 9086, 9091, 9095, 9100, 9102, 9103, 9132, 9139, 9145, 9152, 9156, 9159, 9163
\fp_mul_end_level:NNNNNNNN 8758, 8864, 8866
\fp_mul_internal: 8758, 8772, 8813
\fp_mul_product:NN 8821–8823, 8825–8828, 8830–8832, 8834, 8835, 8839, 8855, 9074–9079, 9081–9085, 9087–9090, 9092–9094, 9098, 9099, 9101, 9118–9123, 9125–9131, 9133–9138, 9140–9144, 9148–9151, 9153–9155, 9157, 9158, 9162
\fp_mul_split:NNNN 8758, 8815, 8817, 8843, 9064, 9066, 9068, 9070, 9108, 9110, 9112, 9114
\fp_mul_split:w 8758
\fp_mul_split_aux:w 8846, 8852
\fp_neg:c 185, 8611
\fp_neg:N 185, 8611, 8611, 8613
\fp_neg:NN 8611
\fp_neg_aux:NN 8611, 8612, 8615
\fp_new:c 180, 7998
\fp_new:N 180, 7998, 7998, 8003, 8006
\fp_overflow_msg: 7867, 7934, 11032, 11038
\fp_pow:cn 186, 10556
\fp_pow:Nn 186, 10556, 10556, 10558
\fp_pow_aux:NNn 10556, 10556, 10557, 10560
\fp_pow_aux_i: 10556, 10606, 10611
\fp_pow_aux_ii: 10556, 10615, 10629, 10647
\fp_pow_aux_iii: ... 10556, 10672, 10701
\fp_pow_aux_iv: 10556, 10650, 10664, 10678, 10682, 10704, 10713
\fp_pow_negative: 10556
\fp_pow_positive: 10556
\fp_read:N 7785, 7785, 8476, 8511, 8593, 8618, 8644, 8747, 8765, 8878, 10563, 10796, 10801
\fp_read_aux:w 7785, 7786, 7787
\fp_round: 8479, 8515, 8538, 8538
\fp_round_aux:NNNNNNNN 8538, 8545, 8547
\fp_round_figures:cn 183, 8467
\fp_round_figures:Nn 183, 8467, 8467, 8469
\fp_round_figures_aux:NNn 8467, 8468, 8471, 8473
\fp_round_loop:N . 8538, 8549, 8560, 8583
\fp_round_places:cn 183, 8502
\fp_round_places:Nn 183, 8502, 8502, 8504
\fp_round_places_aux:NNn 8502, 8503, 8506, 8508
.fp_set:c 172
\fp_set:cn 180, 8016
.fp_set:N 172
\fp_set:Nn 180, 8016, 8016, 8048
\fp_set_aux:NNn 8016, 8016–8018
\fp_set_eq:cc 180, 8100, 8103
\fp_set_eq:cN 180, 8100, 8101
\fp_set_eq:Nc 180, 8100, 8102
\fp_set_eq:NN 180, 8100, 8100
\fp_set_from_dim:cn 181, 8050
\fp_set_from_dim:Nn 181, 8050, 8050, 8096
\fp_set_from_dim_aux:NNn 8050, 8051, 8053, 8054
\fp_set_from_dim_aux:w 8050, 8061, 8090, 8092, 8095
\fp_show:c 181, 8108, 8109
\fp_show:N 181, 8108, 8108
\fp_sin:cn 187, 9413
\fp_sin:Nn 187, 9413, 9413, 9415
\fp_sin_aux:NNn .. 9413, 9413, 9414, 9417
\fp_sin_aux_i: 9413, 9453, 9464
\fp_sin_aux_ii: .. 9413, 9467, 9497, 9811
\fp_split:Nn .. 7798, 7798, 8021, 8059, 8645, 8748, 8766, 8879, 9420, 9517, 9718, 9911, 10229, 10568, 10779, 10785
\fp_split_aux_i:w 7798, 7837, 7841
\fp_split_aux_ii:w 7798, 7842, 7843
\fp_split_aux_iii:w 7798, 7844, 7845
\fp_split_decimal:w 7798, 7848, 7851
\fp_split_decimal_aux:w 7798, 7852, 7853
\fp_split_exponent: 7798

\fp_split_exponent:w 7806, 7833
 \fp_split_sign: [7798](#), [7804](#), [7808](#), [7819](#), [7829](#)
 \fp_standardise:NNNN [7870](#), [7870](#), [8022](#),
 [8064](#), [8646](#), [8666](#), [8749](#), [8767](#), [8777](#),
 [8880](#), [8920](#), [9421](#), [9475](#), [9518](#), [9561](#),
 [9719](#), [9806](#), [9815](#), [9842](#), [9912](#), [10139](#),
 [10230](#), [10295](#), [10569](#), [10780](#), [10786](#)
 \fp_standardise_aux: [7870](#), [7884](#),
 [7890](#), [7900](#), [7901](#), [7907](#), [7924](#), [7937](#)
 \fp_standardise_aux:NNNN [7870](#), [7878](#), [7882](#)
 \fp_standardise_aux:w
 .. [7870](#), [7888](#), [7894](#), [7906](#), [7911](#), [7938](#)
 \fp_sub:cn [185](#), [8740](#)
 \fp_sub:Nn [185](#), [8740](#), [8740](#), [8742](#)
 \fp_sub:NNNNNNNN [9039](#),
 [9039](#), [9378](#), [9389](#), [9400](#), [10405](#), [10491](#)
 \fp_sub_aux:NNn .. [8740](#), [8740](#), [8741](#), [8744](#)
 \fp_tan:cn [188](#), [9711](#)
 \fp_tan:Nn [188](#), [9711](#), [9711](#), [9713](#)
 \fp_tan_aux:NNn .. [9711](#), [9711](#), [9712](#), [9715](#)
 \fp_tan_aux_i: [9711](#), [9751](#), [9762](#)
 \fp_tan_aux_ii: [9711](#), [9765](#), [9772](#)
 \fp_tan_aux_iii: . [9711](#), [9795](#), [9798](#), [9801](#)
 \fp_tan_aux_iv: .. [9711](#), [9825](#), [9828](#), [9831](#)
 \fp_tmp:w [7997](#),
 [7997](#), [8028](#), [8046](#), [8070](#), [8088](#), [8482](#),
 [8500](#), [8518](#), [8536](#), [8595](#), [8609](#), [8652](#),
 [8671](#), [8756](#), [8782](#), [8811](#), [8889](#), [8899](#),
 [8908](#), [8925](#), [9442](#), [9455](#), [9462](#), [9542](#),
 [9548](#), [9740](#), [9753](#), [9760](#), [9937](#), [9950](#),
 [10242](#), [10250](#), [10257](#), [10276](#), [10468](#),
 [10479](#), [10582](#), [10588](#), [10599](#), [10609](#),
 [10632](#), [10639](#), [10685](#), [10720](#), [10734](#)
 \fp_to_dim:c [182](#), [8173](#)
 \fp_to_dim:N [182](#), [8173](#), [8173](#), [8174](#)
 \fp_to_int:c [182](#), [8175](#)
 \fp_to_int:N [182](#), [8175](#), [8175](#), [8177](#)
 \fp_to_int_aux:w [8175](#), [8176](#), [8178](#)
 \fp_to_int_large:w [8175](#), [8186](#), [8201](#)
 \fp_to_int_large_aux:nnn
 [8175](#), [8213](#), [8215](#), [8217](#),
 [8219](#), [8221](#), [8223](#), [8225](#), [8227](#), [8229](#)
 \fp_to_int_large_aux_1:w [8175](#)
 \fp_to_int_large_aux_2:w [8175](#)
 \fp_to_int_large_aux_3:w [8175](#)
 \fp_to_int_large_aux_4:w [8175](#)
 \fp_to_int_large_aux_5:w [8175](#)
 \fp_to_int_large_aux_6:w [8175](#)
 \fp_to_int_large_aux_7:w [8175](#)
 \fp_to_int_large_aux_8:w [8175](#)
 \fp_to_int_large_aux_i:w [8175](#), [8204](#), [8210](#)
 \fp_to_int_large_aux_ii:w [8175](#), [8206](#), [8237](#)
 \fp_to_int_none:w [8175](#)
 \fp_to_int_small:w [8175](#), [8184](#), [8190](#)
 \fp_to_tl:c [182](#), [8242](#)
 \fp_to_tl:N [182](#), [8242](#), [8242](#), [8244](#)
 \fp_to_tl_aux:w [8242](#), [8243](#), [8245](#)
 \fp_to_tl_large:w [8242](#), [8253](#), [8257](#)
 \fp_to_tl_large_0:w [8242](#)
 \fp_to_tl_large_1:w [8242](#)
 \fp_to_tl_large_2:w [8242](#)
 \fp_to_tl_large_3:w [8242](#)
 \fp_to_tl_large_4:w [8242](#)
 \fp_to_tl_large_5:w [8242](#)
 \fp_to_tl_large_6:w [8242](#)
 \fp_to_tl_large_7:w [8242](#)
 \fp_to_tl_large_8:w [8242](#)
 \fp_to_tl_large_8_aux:w [8242](#)
 \fp_to_tl_large_9:w [8242](#)
 \fp_to_tl_large_aux_i:w [8242](#), [8260](#), [8266](#)
 \fp_to_tl_large_aux_ii:w [8242](#), [8262](#), [8268](#)
 \fp_to_tl_large_zeros:NNNNNNNN [8242](#),
 [8271](#), [8277](#), [8282](#), [8287](#), [8292](#), [8297](#),
 [8302](#), [8307](#), [8312](#), [8322](#), [8380](#), [8383](#)
 \fp_to_tl_small:w [8242](#), [8251](#), [8325](#)
 \fp_to_tl_small_aux:w .. [8242](#), [8333](#), [8377](#)
 \fp_to_tl_small_one:w .. [8242](#), [8328](#), [8338](#)
 \fp_to_tl_small_two:w .. [8242](#), [8331](#), [8357](#)
 \fp_to_tl_small_zeros:NNNNNNNN
 .. [8242](#), [8345](#), [8354](#), [8364](#), [8374](#), [8422](#)
 \fp_trig_calc_cos:
 .. [9503](#), [9505](#), [9587](#), [9593](#), [9606](#), [9606](#)
 \fp_trig_calc_sin:
 .. [9501](#), [9507](#), [9589](#), [9591](#), [9606](#), [9642](#)
 \fp_trig_calc_Taylor:
 [9606](#), [9639](#), [9654](#), [9657](#), [9708](#)
 \fp_trig_normalise:
 [9292](#), [9292](#), [9466](#), [9552](#), [9774](#)
 \fp_trig_normalise_aux:
 [9292](#), [9297](#), [9311](#), [9316](#), [9324](#)
 \fp_trig_octant: [9302](#), [9368](#), [9368](#)
 \fp_trig_octant_aux:
 [9368](#), [9371](#), [9386](#), [9396](#), [9409](#)
 \fp_trig_overflow_msg:
 [9308](#), [9769](#), [11054](#), [11060](#)
 \fp_trig_sub:NNN . [9292](#), [9314](#), [9320](#), [9329](#)
 \fp_use:c [181](#), [8110](#)
 \fp_use:N [181](#), [8110](#), [8110](#), [8112](#), [8173](#)
 \fp_use_aux:w [8110](#), [8111](#), [8113](#)

- \fp_use_i_to_iix:NNNNNNNNN 8242, 8342, 8347, 8465
 - \fp_use_i_to_vii:NNNNNNNNN 8242, 8361, 8366, 8463
 - \fp_use_iix_ix:NNNNNNNNN 8242, 8359, 8461
 - \fp_use_ix:NNNNNNNNN ... 8242, 8340, 8462
 - \fp_use_large:w 8110, 8119, 8137
 - \fp_use_large_aux_1:w 8110
 - \fp_use_large_aux_2:w 8110
 - \fp_use_large_aux_3:w 8110
 - \fp_use_large_aux_4:w 8110
 - \fp_use_large_aux_5:w 8110
 - \fp_use_large_aux_6:w 8110
 - \fp_use_large_aux_7:w 8110
 - \fp_use_large_aux_8:w 8110
 - \fp_use_large_aux_i:w .. 8110, 8140, 8146
 - \fp_use_large_aux_ii:w . 8110, 8142, 8167
 - \fp_use_none:w 8110, 8125, 8130
 - \fp_use_small:w 8110, 8123, 8131
 - \fp_zero:c 180, 8010
 - \fp_zero:N 180, 8010, 8010, 8014
 - \frozen@everydisplay 764
 - \frozen@everymath 765
 - \futurelet 361
- G**
- \G 2718
 - \g 2264
 - \g_box_allocation_seq 5918
 - \g_cctab_allocate_int 11092, 11092, 11093, 11100, 11102, 11104
 - \g_cctab_stack_int 11092, 11094, 11130, 11131, 11133, 11134, 11138
 - \g_cctab_stack_seq 11092, 11095, 11128, 11139
 - \g_clist_map_inline_int 5484, 5484, 5487, 5488, 5492, 5493, 5497, 5498, 5502, 5503
 - \g_file_current_name_tl 177, 7602, 7602, 7607, 7611, 7619, 7681, 7682, 7684
 - \g_file_record_seq 178, 7614, 7614, 7619, 7676, 7696, 7698, 7705
 - \g_file_stack_seq 178, 7613, 7613, 7681, 7684
 - \g_file_test_stream 7627, 7629, 7630, 7633, 7650, 7651, 7661
 - \g_ior_streams_prop 6127, 6128, 6133, 6160, 6262, 6276, 6297, 6305
 - \g_ior_tmp_stream 6195, 6241, 6242, 6245
 - \g_iow_streams_prop 6127, 6127, 6130–6132, 6173, 6181, 6189, 6225, 6288, 6319, 6327
 - \g_iow_tmp_stream 6195, 6204, 6205, 6208
 - \g_keyval_level_int 7050, 7050, 7096, 7128, 7154–7156, 7158
 - \g_peek_token 64, 2866, 2867, 2877
 - \g_prg_stepwise_level_int 2200, 2200, 2203, 2205, 2210, 2212
 - \g_prop_map_inline_int 5804, 5804, 5807, 5808, 5811, 5812
 - \g_seq_nesting_depth_int 3827, 4994, 5006, 5008, 5012, 5014
 - \g_tl_inline_level_int 3828, 4535, 4537, 4538, 4541, 4543, 4547, 4548, 4551, 4553
 - \g_tmpa_bool 41, 1909, 1910
 - \g_tmpa_dim 88, 4035, 4038
 - \g_tmpa_int 80, 3822, 3825
 - \g_tmpa_skip 91, 4112, 4115
 - \g_tmpa_tl 109, 4681, 4681
 - \g_tmpb_dim 88, 4035, 4039
 - \g_tmpb_int 80, 3822, 3826
 - \g_tmpb_skip 91, 4112, 4116
 - \g_tmpb_tl 109, 4681, 4682
 - \gdef 352
 - .generate_choices:n 172
 - \GetIdInfo 6, 97, 98
 - \GetIdInfoAuxCVS 97, 136, 141
 - \GetIdInfoAuxI 97, 102, 104
 - \GetIdInfoAuxII 97, 121, 126
 - \GetIdInfoAuxIII 97, 131, 133
 - \GetIdInfoAuxSVN 97, 138, 143
 - \GetIdInfoFull 97
 - \global 336, 367
 - \globaldefs 371
 - \glueexpr 711
 - \glueshrink 714
 - \glueshrinkorder 716
 - \gluestretch 713
 - \gluestretchorder 715
 - \gluetomu 717
 - \group_align_safe_begin: 47, 1921, 2251, 2251, 2898, 2916
 - \group_align_safe_end: 47, 2018, 2019, 2251, 2253, 2880, 2890, 2895, 2913
 - \group_begin: 8, 811, 812, 855, 1072, 1571, 1811, 2263, 2277, 2584, 2597, 2621, 2674, 2711, 2808, 2974, 3071, 3078,

4284, 4301, 4327, 4339, 4474, 4580, 4618, 4871, 6376, 6586, 6633, 7055, 7065, 7108, 7143, 8020, 8056, 8475, 8510, 8592, 8617, 8643, 8746, 8764, 8877, 9419, 9516, 9717, 9910, 10228, 10447, 10562, 10619, 10653, 10715, 10778, 10795, 10982, 11085, 11159	\hbox_set_inline_begin:N 149, 6041, 6041, 6044, 6045 \hbox_set_inline_end: .. 149, 6041, 6047 \hbox_set_to_wd:cnn 148, 6035 \hbox_set_to_wd:Nnn 148, 6035, 6035, 6038, 6039 \hbox_to_wd:nn 148, 6049, 6049 \hbox_to_zero:n 148, 6049, 6051, 6053, 6055 \hbox_unpack:c 149, 6056 \hbox_unpack:N 149, 6056, 6056, 6058 \hbox_unpack_clear:c 149, 6056 \hbox_unpack_clear:N 149, 6056, 6057, 6059 \hfil 521 \hfill 523 \hfilneg 522 \hfuzz 620 \hoffset 595 \holdinginserts 598 \hrule 534 \hsize 559 \hskip 524 \hss 525 \ht 663 \hyphenation 649 \hyphenchar 633 \hyphenpenalty 551
\group_end: 8, 811, 813, 858, 1077, 1576, 1823, 2268, 2282, 2596, 2600, 2628, 2682, 2722, 2814, 3039, 3080, 3089, 4291, 4307, 4332, 4344, 4478, 4481, 4584, 4624, 4882, 6398, 6591, 6639, 7062, 7073, 7123, 7151, 8030, 8072, 8484, 8520, 8597, 8634, 8673, 8784, 8891, 8901, 8927, 9444, 9457, 9544, 9742, 9755, 9939, 10244, 10252, 10278, 10470, 10584, 10590, 10601, 10625, 10631, 10634, 10641, 10658, 10666, 10675, 10687, 10722, 10809, 10820, 10823, 10827, 10831, 10835, 10847, 10851, 10854, 10863, 10866, 10877, 10881, 10895, 10899, 10903, 10908, 10913, 10916, 10927, 10931, 10935, 10940, 10945, 10948, 10985, 11091, 11162	
\group_execute_after:N 1516 \group_insert_after:N . 8, 816, 816, 1516	
<div>H</div>	
\H 2718 \halign 378 \hangafter 556 \hangindent 557 \hbadness 618 \hbox 613 \hbox:n 148, 6030, 6030 \hbox_gset:cn 148, 6031 \hbox_gset:Nn 148, 6031, 6032, 6034 \hbox_gset_inline_begin:c 149, 6041 \hbox_gset_inline_begin:N 149, 6041, 6043, 6046 \hbox_gset_inline_end: .. 149, 6041, 6048 \hbox_gset_to_wd:cnn 148, 6035 \hbox_gset_to_wd:Nnn 148, 6035, 6037, 6040 \hbox_overlap_left:n ... 149, 6052, 6052 \hbox_overlap_right:n .. 148, 6052, 6054 \hbox_set:cn 148, 6031 \hbox_set:Nn 148, 6031, 6031–6033 \hbox_set_inline_begin:c 149, 6041	
<div>I</div>	
\I 2718 \if 184, 387 \if:w 26, 784, 790, 981, 1049, 1065, 1777, 1805, 2851, 2956, 3325, 4643, 7789, 8115, 8180, 8247 \if_bool:N 26, 784, 791, 1899 \if_box_empty:N ... 153, 5987, 5989, 6003 \if_case:w 81, 1320, 3176, 3181, 9499, 9585 \if_catcode:w 26, 784, 794, 1831, 2406, 2419, 2603, 2608, 2613, 2618, 2625, 2631, 2636, 2641, 2646, 2651, 2656, 2666, 2699, 2925, 4656, 5770, 5796, 5882, 6468 \if_charcode:w 26, 784, 793, 2671 \if_cs_exist:N 26, 800, 800, 1105, 1133, 2860 \if_cs_exist:w 26, 800, 801, 1114, 1142, 4574, 9448, 9538, 9746, 9933, 9942, 10272 \if_dim:w 94, 3875, 3875, 3932, 3957–3963 \if_eof:w 158, 6102, 6102, 6481 \if_false: 25, 784, 785, 2252, 4706, 4712, 4848, 4851, 4954, 4958	

- \if_hbox:N 152, 5987, 5987, 5991
- \if_int_compare:w
 - 81, 814, 814, 1485, 1491,
 - 2252, 2254, 2460, 2469, 2690, 3176,
 - 3187, 3195, 3206, 3217, 3221, 3222,
 - 3228, 3335, 3343, 3351, 3359, 3367,
 - 3375, 3383, 3391, 4078, 7810, 7821,
 - 7856, 7864, 7872, 7886, 7903, 7925,
 - 7926, 7941, 7949, 7974, 8033, 8075,
 - 8118, 8121, 8139, 8183, 8192, 8194,
 - 8203, 8231, 8250, 8259, 8327, 8330,
 - 8340, 8341, 8359, 8360, 8385–8393,
 - 8424–8432, 8478, 8487, 8514, 8523,
 - 8553, 8562, 8566, 8575, 8576, 8582,
 - 8622, 8657, 8676, 8702, 8718, 8722,
 - 8724, 8787, 8791, 8885, 8895, 8930,
 - 8934, 8965, 8968, 8976, 8979, 8981,
 - 8996, 9028, 9033, 9044, 9048, 9052,
 - 9053, 9178, 9190, 9218, 9229, 9251,
 - 9294, 9298, 9313, 9318, 9319, 9337,
 - 9341, 9345, 9347, 9372, 9388, 9398,
 - 9428, 9441, 9468, 9483, 9525, 9554,
 - 9569, 9595, 9596, 9600, 9608, 9622,
 - 9623, 9645, 9678, 9679, 9683, 9689,
 - 9699, 9703, 9726, 9739, 9764, 9775,
 - 9776, 9782, 9789, 9790, 9820, 9821,
 - 9850, 9920, 9954, 9956, 9957, 9967,
 - 9982, 9994, 10010, 10011, 10033,
 - 10043, 10061, 10062, 10094, 10095,
 - 10101, 10130, 10133, 10168, 10172,
 - 10176, 10182, 10192, 10196, 10235,
 - 10236, 10286, 10289, 10302, 10319,
 - 10325, 10348, 10362, 10374, 10399,
 - 10402, 10413, 10421, 10481, 10488,
 - 10501, 10511, 10531, 10541, 10547,
 - 10574, 10578, 10595, 10613, 10618,
 - 10621, 10649, 10652, 10656, 10657,
 - 10815–10818, 10841, 10844, 10850,
 - 10859, 10862, 10876, 10880, 10884,
 - 10894, 10898, 10902, 10906, 10911,
 - 10926, 10930, 10934, 10938, 10943
- \if_int_odd:w 81, 3176,
- 3180, 3399, 3407, 9376, 10429, 10432
- \if_meaning:w
 - .. 25, 795, 795, 1085, 1102, 1120,
 - 1130, 1148, 1423, 1560, 1561, 1849,
 - 1929, 1939, 1948, 1951, 1961, 1964,
 - 1977, 1986, 2391, 2397, 2440, 2448,
 - 2661, 2706, 2739, 2750, 2761, 2772,
 - 2834, 2934, 3332, 3952, 4429, 4440,
 - 4451, 4462, 4477, 4635, 4876, 4976,
 - 5090, 5431, 5444, 5752, 10762, 10770
- \if_mode_horizontal: .. 26, 796, 797, 2243
- \if_mode_inner: 26, 796, 799, 2245
- \if_mode_math: 26, 796, 796, 2249
- \if_mode_vertical: 26, 796, 798, 2241
- \if_num:w 81, 2842, 3176, 3179
- \if_predicate:w . 26, 784, 792, 1304, 1913
- \if_true: 25, 784, 784
- \if_vbox:N 153, 5987, 5988, 5993
- \ifcase 388
- \ifcat 389
- \ifcsname 672
- \ifdefined 671
- \ifdim 392
- \ifeof 393
- \iffalse 398
- \iffontchar 701
- \ifhbox 394
- \ifhmode 400
- \ifinner 403
- \ifmmode 401
- \ifnum 390
- \ifodd 169, 205, 391
- \iftrue 399
- \ifvbox 395
- \ifvmode 402
- \ifvoid 396
- \ifx 13, 62, 108, 135, 229, 233, 397
- \ignorespaces 445
- \immediate 407
- \indent 541
- \initcatcodetable 757
- \input 415
- \input@path 7641, 7644, 7658
- \inputlineno 417
- \insert 597
- \insertpenalties 600
- \int_abs:n 69, 3184, 3184
- \int_add:cn 71, 3288
- \int_add:Nn 71, 3288, 3288, 3293, 3296, 6416
- \int_compare:n 3319
- \int_compare:nF 3423, 3438
- \int_compare:nNn 3389
- \int_compare:nNnF
 - 2184, 2193, 2216, 2225,
 - 2259, 2260, 3451, 3466, 6273, 6285
- \int_compare:nNnT .. 2257, 3443, 3460,
- 5172, 6202, 6221, 6239, 6258, 11131

\int_compare:nNnTF [73](#), [2046](#), [2095](#), [2177](#),
 [2207](#), [3259](#), [3261](#), [3389](#), [3472](#), [3548](#),
 [3560](#), [3704](#), [3728](#), [3732](#), [3782](#), [5185](#),
 [5573](#), [5583](#), [6156](#), [6169](#), [6417](#), [11101](#)
 \int_compare:nT [3415](#), [3432](#)
 \int_compare:nTF [73](#), [3319](#)
 \int_compare_<:w [3319](#)
 \int_compare_=:w [3319](#)
 \int_compare_>:w [3319](#)
 \int_compare_aux:Nw [3319](#), [3323](#), [3331](#)
 \int_compare_aux:nw [3319](#), [3320](#), [3321](#)
 \int_compare_p:n [73](#), [3319](#)
 \int_compare_p:nNn .. [73](#), [3389](#), [4090](#), [4091](#)
 \int_const:cn [70](#), [3257](#), [3741](#)–[3754](#)
 \int_const:Nn [70](#), [3257](#),
 [3257](#), [3277](#), [3803](#)–[3821](#), [7715](#)–[7719](#)
 \int_convert_from_base_ten:nn [3829](#), [3829](#)
 \int_convert_to_base_ten:nn . [3829](#), [3831](#)
 \int_convert_to_symbols:nnn . [3829](#), [3830](#)
 \int_decr:c [72](#), [3300](#)
 \int_decr:N [72](#), [3300](#), [3302](#), [3307](#), [3309](#)
 \int_div_round:nn [69](#), [3214](#), [3239](#)
 \int_div_truncate:nn
 [70](#), [3214](#), [3214](#), [3243](#), [3475](#), [3571](#)
 \int_do_until:nn ... [75](#), [3413](#), [3435](#), [3439](#)
 \int_do_until:nNnn .. [74](#), [3441](#), [3463](#), [3467](#)
 \int_do_while:nn [75](#), [3413](#), [3429](#)
 \int_do_while:nNnn
 [74](#), [3433](#), [3441](#), [3457](#), [3461](#)
 \int_eval:n [69](#), [1354](#), [2090](#),
 [2180](#), [2188](#), [2197](#), [2211](#), [2220](#), [2229](#),
 [3182](#), [3182](#), [3239](#), [3469](#), [3477](#), [3480](#),
 [3552](#), [3556](#), [3617](#), [3627](#), [3686](#), [3700](#),
 [3704](#), [3707](#), [3722](#), [3731](#), [4587](#), [4592](#),
 [5158](#), [5170](#), [5553](#), [5561](#), [5576](#), [5588](#)
 \int_eval:w [81](#), [1320](#),
 [2139](#), [2488](#), [2490](#), [2492](#), [2558](#), [2560](#),
 [2562](#), [2564](#), [2566](#), [2568](#), [2570](#), [2572](#),
 [2574](#), [2576](#), [2578](#), [2580](#), [3176](#), [3177](#),
 [3183](#), [3187](#), [3190](#), [3194](#), [3196](#), [3205](#),
 [3207](#), [3216](#), [3217](#), [3221](#), [3222](#), [3228](#),
 [3242](#), [3269](#), [3289](#), [3291](#), [3313](#), [3320](#),
 [3335](#), [3343](#), [3351](#), [3359](#), [3367](#), [3375](#),
 [3383](#), [3391](#), [3399](#), [3407](#), [3760](#), [6461](#),
 [7836](#), [7839](#), [7857](#), [7873](#), [8234](#), [8342](#),
 [8346](#), [8361](#), [8365](#), [8564](#), [8565](#), [8658](#),
 [8684](#), [8695](#), [8699](#), [8711](#), [8715](#), [8728](#),
 [8732](#), [8774](#), [8788](#), [8792](#), [8805](#), [8858](#),
 [8886](#), [8896](#), [8917](#), [8931](#), [8935](#), [8948](#),
 [8966](#), [9011](#), [9020](#), [9025](#)–[9027](#), [9041](#)–
 [9043](#), [9053](#), [9057](#), [9058](#), [9170](#), [9177](#),
 [9202](#), [9212](#), [9332](#), [9334](#), [9336](#), [9348](#),
 [9354](#), [9358](#), [9362](#), [9436](#), [9491](#), [9533](#),
 [9577](#), [9734](#), [9839](#), [9858](#), [9928](#), [10007](#),
 [10040](#), [10103](#), [10109](#), [10113](#), [10150](#),
 [10169](#), [10237](#), [10267](#), [10310](#), [10414](#),
 [10532](#), [10575](#), [10579](#), [10596](#), [10622](#),
 [10694](#), [10744](#), [10841](#), [10844](#), [10859](#)
 \int_eval_end: [81](#), [1320](#),
 [2139](#), [2488](#), [2490](#), [2492](#), [2558](#), [2560](#),
 [2562](#), [2564](#), [2566](#), [2568](#), [2570](#), [2572](#),
 [2574](#), [2576](#), [2578](#), [2580](#), [3176](#), [3178](#),
 [3183](#), [3190](#), [3196](#), [3201](#), [3207](#), [3212](#),
 [3237](#), [3244](#), [3269](#), [3289](#), [3291](#), [3313](#),
 [3335](#), [3343](#), [3351](#), [3359](#), [3367](#), [3375](#),
 [3383](#), [3391](#), [3399](#), [3407](#), [6464](#), [8234](#),
 [8348](#), [8367](#), [8950](#), [9014](#), [9020](#), [9025](#)–
 [9027](#), [9041](#)–[9043](#), [9057](#), [9058](#), [9170](#),
 [9177](#), [9332](#), [9334](#), [9336](#), [9356](#), [9360](#),
 [9364](#), [9841](#), [10009](#), [10042](#), [10105](#), [10115](#)
 \int_from_alph:n [78](#), [3684](#), [3684](#)
 \int_from_alph_aux:N ... [3684](#), [3700](#), [3703](#)
 \int_from_alph_aux:n ... [3684](#), [3689](#), [3692](#)
 \int_from_alph_aux:nN
 [3684](#), [3693](#), [3694](#), [3699](#)
 \int_from_base:nn
 [78](#), [3705](#), [3705](#), [3736](#), [3738](#), [3740](#), [3831](#)
 \int_from_base_aux:N ... [3705](#), [3722](#), [3726](#)
 \int_from_base_aux:nn .. [3705](#), [3710](#), [3714](#)
 \int_from_base_aux:nnN
 [3705](#), [3715](#), [3716](#), [3721](#)
 \int_from_binary:n [78](#), [3735](#), [3735](#)
 \int_from_hexadecimal:n . [78](#), [3735](#), [3737](#)
 \int_from_octal:n [78](#), [3735](#), [3739](#)
 \int_from_roman:n [78](#), [3755](#), [3755](#)
 \int_from_roman_aux:NN
 [3755](#), [3761](#), [3764](#), [3789](#), [3793](#)
 \int_from_roman_clean_up:w
 [3755](#), [3772](#), [3779](#), [3781](#), [3800](#)
 \int_from_roman_end:w .. [3755](#), [3759](#), [3798](#)
 \int_gadd:cn [71](#), [3288](#)
 \int_gadd:Nn
 .. [71](#), [3288](#), [3292](#), [3297](#), [11100](#), [11130](#)
 \int_gdecr:c [72](#), [3300](#)
 \int_gdecr:N
 . [72](#), [2212](#), [3300](#), [3306](#), [3311](#), [4543](#),
 [4553](#), [5012](#), [5493](#), [5503](#), [5812](#), [7158](#)
 \int_get_digits:n [3650](#), [3655](#), [3689](#), [3711](#)
 \int_get_sign:n .. [3650](#), [3650](#), [3688](#), [3709](#)

`\int_get_sign_and_digits_aux:nNNN` ..
 [3650](#), [3652](#), [3657](#), [3660](#), [3683](#)
`\int_get_sign_and_digits_aux:oNNN` ..
 [3650](#), [3666](#), [3670](#), [3676](#)
`\int_gincr:c` [72](#), [3300](#)
`\int_gincr:N`
 . [72](#), [2203](#), [3300](#), [3304](#), [3310](#), [4537](#),
 [4547](#), [5008](#), [5487](#), [5497](#), [5807](#), [7154](#)
`.int_gset:c` [173](#)
`\int_gset:cn` [72](#), [3312](#)
`.int_gset:N` [173](#)
`\int_gset:Nn` [72](#), [3264](#), [3274](#), [3312](#), [3314](#), [3316](#)
`\int_gset_eq:cc` [71](#), [3282](#)
`\int_gset_eq:cN` [71](#), [3282](#)
`\int_gset_eq:Nc` [71](#), [3282](#)
`\int_gset_eq:NN` [71](#), [3282](#), [3285](#)–[3287](#)
`\int_gsub:cn` [72](#), [3288](#)
`\int_gsub:Nn` .. [72](#), [3288](#), [3294](#), [3299](#), [11138](#)
`\int_gzero:c` [71](#), [3278](#)
`\int_gzero:N` [71](#), [3278](#), [3279](#), [3281](#)
`\int_if_even:n` [3405](#)
`\int_if_even:nTF` [74](#), [3397](#)
`\int_if_even:p:n` [74](#), [3397](#)
`\int_if_odd:n` [3397](#)
`\int_if_odd:nTF` [74](#), [3397](#)
`\int_if_odd:p:n` [74](#), [3397](#)
`\int_incr:c` [72](#), [3300](#)
`\int_incr:N` [72](#), [3300](#), [3300](#),
 [3305](#), [3308](#), [6220](#), [6257](#), [6433](#), [7300](#)
`\int_max:nn` [70](#), [3184](#), [3192](#)
`\int_min:nn` [70](#), [3184](#), [3203](#)
`\int_mod:nn` [70](#), [3214](#), [3240](#), [3477](#), [3568](#)
`\int_new:c` [70](#), [3246](#)
`\int_new:N` [70](#), [2200](#), [3246](#), [3250](#), [3256](#),
 [3263](#), [3273](#), [3822](#)–[3828](#), [5484](#), [5804](#),
 [6135](#), [6363](#), [6365](#)–[6367](#), [7050](#), [7181](#),
 [7720](#), [7722](#), [7724](#), [7726](#), [7728](#), [7730](#),
 [7738](#)–[7766](#), [7768](#)–[7772](#), [7775](#), [7776](#),
 [7778](#), [7779](#), [7781](#)–[7784](#), [11092](#), [11094](#)
`.int_set:c` [172](#)
`\int_set:cn` [72](#), [3312](#)
`.int_set:N` [172](#)
`\int_set:Nn` [72](#), [3312](#), [3312](#), [3314](#), [3315](#),
 [6154](#), [6167](#), [6183](#), [6191](#), [6205](#), [6242](#),
 [6364](#), [6377](#), [6414](#), [6669](#), [7295](#), [7721](#),
 [7723](#), [7725](#), [7727](#), [7729](#), [7731](#), [8477](#),
 [8512](#), [10960](#), [10962](#), [10964](#), [10966](#),
 [10968](#), [10970](#), [10972](#), [10974](#), [11093](#)
`\int_set_eq:cc` [71](#), [3282](#)
`\int_set_eq:cN` [71](#), [3282](#)
`\int_set_eq:Nc` [71](#), [3282](#)
`\int_set_eq:NN` . [71](#), [3282](#), [3282](#)–[3284](#), [6440](#)
`\int_show:c` [79](#), [3801](#), [3802](#)
`\int_show:N` [79](#), [1447](#), [3801](#), [3801](#)
`\int_sub:cn` [72](#), [3288](#)
`\int_sub:Nn` [72](#), [3288](#), [3290](#), [3295](#), [3298](#)
`\int_to_Alph:n` [75](#), [3482](#), [3514](#)
`\int_to_alph:n` [75](#), [3482](#), [3482](#)
`\int_to_arabic:n` [75](#), [3469](#), [3469](#)
`\int_to_base:nn`
 [77](#), [3546](#), [3546](#), [3609](#), [3611](#), [3613](#), [3829](#)
`\int_to_base_aux:nnn`
 [3546](#), [3551](#), [3555](#), [3559](#), [3566](#)
`\int_to_binary:n` [77](#), [3608](#), [3608](#)
`\int_to_hexadecimal:n` ... [77](#), [3608](#), [3610](#)
`\int_to_letter:n` . [3546](#), [3562](#), [3568](#), [3575](#)
`\int_to_octal:n` [77](#), [3608](#), [3612](#)
`\int_to_Roman:n` [77](#), [3614](#), [3624](#)
`\int_to_roman:n` [77](#), [3614](#), [3614](#)
`\int_to_roman:w` [80](#), [814](#), [815](#),
 [896](#), [898](#), [1065](#), [1071](#), [1081](#), [2001](#),
 [2006](#), [2137](#), [3176](#), [3324](#), [3617](#), [3627](#)
`\int_to_Roman_aux:N` [3626](#), [3629](#), [3632](#)
`\int_to_roman_aux:N` [3614](#), [3616](#), [3619](#), [3622](#)
`\int_to_Roman_c:w` [3614](#), [3646](#)
`\int_to_roman_c:w` [3614](#), [3638](#)
`\int_to_Roman_d:w` [3614](#), [3647](#)
`\int_to_roman_d:w` [3614](#), [3639](#)
`\int_to_Roman_i:w` [3614](#), [3642](#)
`\int_to_roman_i:w` [3614](#), [3634](#)
`\int_to_Roman_l:w` [3614](#), [3645](#)
`\int_to_roman_l:w` [3614](#), [3637](#)
`\int_to_Roman_m:w` [3614](#), [3648](#)
`\int_to_roman_m:w` [3614](#), [3640](#)
`\int_to_Roman_Q:w` [3614](#), [3649](#)
`\int_to_roman_Q:w` [3614](#), [3641](#)
`\int_to_Roman_v:w` [3614](#), [3643](#)
`\int_to_roman_v:w` [3614](#), [3635](#)
`\int_to_Roman_x:w` [3614](#), [3644](#)
`\int_to_roman_x:w` [3614](#), [3636](#)
`\int_to_symbol:n` [78](#), [3832](#), [3832](#)
`\int_to_symbol_math:n` .. [3832](#), [3835](#), [3838](#)
`\int_to_symbol_text:n` .. [3832](#), [3836](#), [3853](#)
`\int_to_symbols:nnn` .. [76](#), [3470](#), [3470](#),
 [3474](#), [3484](#), [3516](#), [3830](#), [3840](#), [3855](#)
`\int_until_do:nn` ... [75](#), [3413](#), [3421](#), [3426](#)
`\int_until_do:nNnn` .. [74](#), [3441](#), [3449](#), [3454](#)
`\int_use:c` [73](#), [3317](#), [3318](#)
`\int_use:N`
 . [73](#), [2205](#), [2210](#), [3317](#), [3317](#), [3318](#),

- 4538, 4541, 4548, 4551, 5006, 5014,
 5488, 5492, 5498, 5502, 5808, 5811,
 6197, 6198, 6207, 6212, 6214, 6223,
 6234, 6235, 6244, 6249, 6251, 6260,
 6571, 7096, 7128, 7155, 7156, 7296,
 7849, 7889, 7906, 7919, 7953, 7967,
 7978, 7992, 8038, 8041, 8043, 8080,
 8083, 8085, 8492, 8495, 8497, 8528,
 8531, 8533, 8545, 8572, 8601, 8604,
 8606, 8627, 8630, 8632, 8681, 8687,
 8802, 8808, 8852, 8864, 8945, 8952,
 8964, 9194, 9206, 9222, 9234, 9244,
 9255, 9268, 9287, 9433, 9439, 9488,
 9494, 9530, 9536, 9574, 9580, 9731,
 9737, 9855, 9861, 9925, 9931, 10004,
 10037, 10063, 10066, 10147, 10153,
 10264, 10270, 10307, 10314, 10327,
 10349, 10366, 10379, 10389, 10400,
 10473, 10475, 10477, 10505, 10516,
 10691, 10697, 10724, 10726, 10728,
 10730, 10732, 10961, 10963, 10965,
 10967, 10969, 10971, 10973, 10975
 \int_value:w
 . 81, 1972, 1973, 1993, 1995–1997,
 2139, 3176, 3176, 3183, 3186, 3194,
 3205, 3216, 3242, 3320, 3760, 3929,
 6461, 8234, 8346, 8365, 8684, 8805,
 8948, 9436, 9491, 9533, 9577, 9734,
 9858, 9928, 10150, 10267, 10310, 10694
 \int_while_do:nn ... 75, 3413, 3413, 3418
 \int_while_do:nNnn .. 74, 3441, 3441, 3446
 \int_zero:c 71, 3278
 \int_zero:N
 71, 3278, 3278, 3280, 6378, 6447, 7282
 \interactionmode 699
 \interlinepenalties 720
 \interlinepenalty 579
 \ior_alloc_read:n 6155, 6179, 6187
 \ior_close:c 154
 \ior_close:N
 ... 154, 6153, 6269, 6293, 7633, 7661
 \ior_gto:NN 157, 6489, 6491
 \ior_if_eof:N 6477
 \ior_if_eof:Nf 7651
 \ior_if_eof:Ntf 157, 7630
 \ior_if_eof_p:N 157
 \ior_if_eof_p:Ntf 6477
 \ior_if_eof_p_p:N 6477
 \ior_list_streams: . 154, 6295, 6295, 6505
 \ior_new:c 6501, 6502
 \ior_new:N 6501, 6501
 \ior_open:cn 154, 6151
 \ior_open:Nn
 ... 154, 6151, 6151, 6177, 7629, 7650
 \ior_open_streams: 6505, 6505
 \ior_raw_new:c 158, 6137, 6249
 \ior_raw_new:N
 ... 158, 6137, 6139, 6147, 6149, 6241
 \ior_show_aux:nn . 6295, 6305, 6310, 6332
 \ior_str_gto:NN 157, 6493, 6495
 \ior_str_to:NN 157, 6493, 6493
 \ior_stream_alloc:N 6159, 6195, 6232
 \ior_stream_alloc_aux:
 6195, 6238, 6255, 6263, 6265
 \ior_to:NN 156, 6489, 6489
 \iow_alloc_write:n 6168, 6179, 6179
 \iow_char:N 155, 5084,
 5547, 5840, 5842, 6313, 6476, 6476
 \iow_close:c 154, 6269
 \iow_close:N .. 154, 6166, 6269, 6281, 6294
 \iow_list_streams: . 154, 6295, 6317, 6506
 \iow_log:n ... 155, 6355, 6356, 7697–7699
 \iow_log:x 155, 1169, 1169, 1209,
 1795, 6355, 6355, 6657, 6659, 6660
 \iow_new:c 6501, 6504
 \iow_new:N 6501, 6503
 \iow_newline:
 156, 5083, 5546, 5839, 6312, 6382,
 6391, 6404, 6475, 6475, 6644, 6646
 \iow_now:Nn 154, 6353,
 6353, 6356, 6358, 6360, 6498, 6500
 \iow_now:Nx
 154, 6352, 6352, 6354, 6355, 6357, 6362
 \iow_now_buffer_safe:Nn 6497, 6497
 \iow_now_buffer_safe:Nx 6497, 6499
 \iow_now_when_avail:Nn . 155, 6359, 6359
 \iow_now_when_avail:Nx . 155, 6359, 6361
 \iow_open:cn 154, 6151
 \iow_open:Nn 154, 6151, 6164, 6178
 \iow_open_streams: 6505, 6506
 \iow_raw_new:c 158, 6137, 6212
 \iow_raw_new:N
 ... 158, 6137, 6142, 6146, 6150, 6204
 \iow_shipout:Nn ... 155, 6349, 6349, 6351
 \iow_shipout:Nx 155, 6349
 \iow_shipout_x:Nn
 ... 155, 6347, 6347, 6348, 6350, 6352
 \iow_shipout_x:Nx 155, 6347
 \iow_show_aux:nn 6295, 6327, 6332
 \iow_stream_alloc:N 6172, 6195, 6195

- \iow_stream_alloc_aux: 6195, 6201, 6218, 6226, 6228
 - \iow_term:n 155, 6355, 6358
 - \iow_term:x 155, 1169, 1171, 5066, 5070, 5529, 5533, 5822, 5826, 6299, 6303, 6321, 6325, 6355, 6357, 6642, 6664, 6666, 6667
 - \iow_wrap:xnnnN 156, 6374, 6374, 6498, 6500, 6601, 6604, 6609, 6612, 6658, 6665
 - \iow_wrap_end: 6374, 6408, 6452
 - \iow_wrap_loop:w 6374, 6394, 6401, 6421, 6450
 - \iow_wrap_newline: 6374, 6405, 6443
 - \iow_wrap_word: 6374, 6409, 6412
 - \iow_wrap_word_fits: ... 6374, 6419, 6423
 - \iow_wrap_word_newline: 6374, 6420, 6436
- J**
- \jobname 654
- K**
- \K 2718
 - \kern 532
 - \kernel_register_show:c 1438, 1447, 3802
 - \kernel_register_show:N 1438, 1438, 3801, 4024, 4108, 4158
 - \keys_bool_set:NN 7251, 7251, 7359, 7361
 - \keys_choice_code_store:x 7302, 7302, 7369, 7371
 - \keys_choice_find:n 7269, 7519, 7519
 - \keys_choice_make: 7254, 7266, 7266, 7281, 7363
 - \keys_choices_generate:n 7276, 7276, 7393
 - \keys_choices_generate_aux:n 7276, 7283, 7290
 - \keys_cmd_set:nn 7259, 7268, 7270, 7313, 7313, 7334, 7365
 - \keys_cmd_set:nx 7255, 7257, 7292, 7313, 7318, 7339, 7351, 7367
 - \keys_cmd_set_aux:n 7313, 7315, 7320, 7323
 - \keys_default_set:n 7264, 7329, 7329, 7331, 7373
 - \keys_default_set:V 7329, 7375
 - \keys_define:nn ... 170, 7189, 7189, 7552
 - \keys_define_aux:nnn ... 7189, 7191, 7197
 - \keys_define_aux:onn 7189, 7190
 - \keys_define_elt:n 7194, 7198, 7198
 - \keys_define_elt:nn 7194, 7198, 7203
 - \keys_define_elt_aux:nn 7198, 7201, 7206, 7208
 - \keys_define_key:n 7211, 7234, 7234
 - \keys_define_key_aux:w . 7234, 7238, 7249
 - \keys_execute: 7478, 7500, 7500
 - \keys_execute:nn 7500, 7501, 7504, 7510, 7521, 7522
 - \keys_execute_unknown: . 7500, 7501, 7502
 - \keys_if_exist:nn 7524
 - \keys_if_exist:nnTF 176, 7524
 - \keys_if_exist_p:nn 176, 7524
 - \keys_if_value:n 7493
 - \keys_if_value_p:n 7461, 7471, 7493
 - \keys_meta_make:n 7332, 7332, 7403
 - \keys_meta_make:x 7332, 7337, 7405
 - \keys_property_find:n .. 7209, 7217, 7217
 - \keys_property_find_aux:w 7217, 7221, 7224, 7230
 - \keys_set:nn 176, 7335, 7340, 7434, 7434, 7442
 - \keys_set:no 176, 7434
 - \keys_set:nV 176, 7434
 - \keys_set:nv 176, 7434
 - \keys_set_aux:nnn 7434, 7436, 7443
 - \keys_set_aux:onn 7434, 7435
 - \keys_set_elt:n 7439, 7444, 7444
 - \keys_set_elt:nn 7439, 7444, 7449
 - \keys_set_elt_aux:nn 7444, 7447, 7452, 7454
 - \keys_show:nn 177, 7530, 7530
 - \keys_value_or_default:n 7458, 7481, 7481
 - \keys_value_requirement:n 7342, 7342, 7431, 7433
 - \keys_variable_set:cnN 7348, 7379, 7387, 7397, 7409, 7417, 7421
 - \keys_variable_set:cnNN 7348, 7383, 7391, 7401, 7413, 7425, 7429
 - \keys_variable_set:NnN 7348, 7354, 7357, 7377, 7385, 7395, 7407, 7415, 7419
 - \keys_variable_set:NnNN 7348, 7348, 7355, 7356, 7381, 7389, 7399, 7411, 7423, 7427
 - \keyval_parse:n 7055, 7063, 7157
 - \keyval_parse:NnN 168, 7152, 7152, 7166-7168, 7194, 7439
 - \keyval_parse_elt:w 7071, 7077, 7077, 7080, 7085
 - \keyval_remove_spaces:w 7108, 7114, 7121, 7148

- \keyval_remove_spaces_aux:w 7108, 7121, 7122
- \keyval_split_key:w 7091, 7108, 7110
- \keyval_split_key_aux:w 7108, 7118, 7120
- \keyval_split_key_value:w 7084, 7089, 7089
- \keyval_split_key_value_aux:wTF 7089, 7101, 7106
- \keyval_split_value:w .. 7102, 7124, 7124
- \keyval_split_value_aux:w ... 7140, 7145
- \KV_process_no_space_removal_no_sanitization:NNn 7166, 7168
- \KV_process_space_removal_no_sanitization:NNn 7166, 7167
- \KV_process_space_removal_sanitization:NNn 7166, 7166

- L**
- \L 2718
- \l_cctab_tmp_tl 11126, 11139–11142, 11156
- \l_clist_remove_clist 5358, 5358, 5365, 5368, 5369, 5371, 5383, 5387, 5393, 5396, 5397, 5399
- \l_clist_show_tl 5538, 5541
- \l_clist_tmpa_tl 5256, 5256
- \l_clist_tmpb_tl 5256, 5257
- \l_exp_tl 36, 1524, 1524, 1543, 1544
- \l_expl_status_bool 96, 294, 309, 323, 327, 328
- \l_expl_status_stack_tl 197
- \l_file_name_tl 179, 7622, 7622, 7665, 7666, 7672, 7673, 7683
- \l_file_search_path_saved_seq 179, 7624, 7625, 7643, 7659
- \l_file_search_path_seq 179, 7623, 7623, 7643, 7645, 7648, 7659, 7689, 7690, 7693
- \l_fp_arg_tl .. 7737, 7737, 9426, 9445, 9449, 9459, 9480, 9481, 9523, 9538, 9546, 9566, 9567, 9724, 9743, 9747, 9757, 9767, 9791, 9822, 9847, 9848, 9918, 9933, 9942, 9944, 9972, 10012, 10144, 10261, 10272, 10280, 10300
- \l_fp_count_int 7738, 7738, 8958, 8993, 9005, 9013, 9621, 9653, 9670, 9672, 9675, 9677, 10121, 10158, 10166, 10396, 10399, 10400, 10422, 10466, 10526, 10537
- \l_fp_div_offset_int 7739, 7739, 8914, 8968, 9013, 9015, 9836
- \l_fp_exp_decimal_int ... 7740, 7741, 9996, 10030, 10049, 10069, 10088, 10097, 10102, 10108, 10124, 10173, 10178, 10182, 10185, 10189, 10193, 10196, 10198, 10342, 10352, 10363, 10366, 10375, 10383, 10409, 10472, 10480, 10484, 10495, 10538, 10539
- \l_fp_exp_exponent_int 7740, 7743, 9998, 10032, 10054, 10074, 10090, 10340, 10344, 10362, 10369, 10392, 10476
- \l_fp_exp_extended_int 7740, 7742, 9997, 10031, 10049, 10069, 10089, 10098, 10101, 10106, 10112, 10124, 10174, 10176, 10179, 10190, 10192, 10194, 10343, 10352, 10385, 10389, 10391, 10409, 10474, 10481, 10483, 10486, 10495, 10538, 10540
- \l_fp_exp_integer_int 7740, 7740, 9995, 10029, 10049, 10069, 10087, 10096, 10100, 10124, 10184, 10197, 10341, 10352, 10374, 10379, 10382, 10409, 10471
- \l_fp_input_a_decimal_int 7744, 7746, 7795, 7986, 7987, 7992, 7994, 8025, 8027, 8041, 8067, 8069, 8083, 8481, 8495, 8517, 8531, 8543, 8545, 8552, 8556, 8594, 8604, 8619, 8630, 8700, 8716, 8815, 8897, 8962, 8964, 8966, 8982, 8995, 8996, 8998, 9021, 9192, 9194, 9203, 9211, 9212, 9219, 9222, 9230, 9238, 9319, 9333, 9334, 9338, 9341, 9343, 9349, 9357, 9359, 9372, 9373, 9380, 9382, 9390, 9393, 9399, 9401, 9405, 9424, 9437, 9521, 9534, 9608, 9614, 9615, 9645, 9648, 9651, 9662, 9666, 9722, 9735, 9789, 9813, 9818, 9820, 9915, 9929, 10094, 10097, 10104, 10110, 10119, 10161, 10233, 10238, 10268, 10415, 10429, 10433, 10436, 10451, 10456, 10460, 10463, 10464, 10528, 10530, 10533, 10536, 10566, 10572, 10580, 10597, 10623, 10656, 10706, 10717, 10737, 10750, 10783, 10799, 10817, 10842, 10912, 10944, 10964, 10973
- \l_fp_input_a_exponent_int 7744, 7747, 7796, 7941, 7949, 7974, 7995, 8026, 8043, 8068, 8085, 8497, 8513, 8533, 8557, 8606, 8632, 8665, 8775, 8918,

- 9190, 9214, 9218, 9247, 9294, 9425,
 9439, 9441, 9522, 9536, 9723, 9737,
 9739, 9764, 9814, 9819, 9840, 9916,
 9931, 9954, 10234, 10270, 10319,
 10320, 10325, 10327, 10338, 10348,
 10349, 10449, 10458, 10567, 10573,
 10618, 10652, 10707, 10718, 10745,
 10752, 10784, 10800, 10819, 10894,
 10898, 10926, 10930, 10966, 10975
 \l_fp_input_a_extended_int
 7752, 7752, 9204,
 9206, 9213, 9240, 9244, 9246, 9295,
 9335-9337, 9339, 9349, 9361, 9363,
 9374, 9381, 9383, 9391, 9394, 9402,
 9406, 9614, 9615, 9649, 9652, 9662,
 9666, 9698, 9917, 10098, 10114,
 10120, 10161, 10191, 10397, 10430,
 10437, 10457, 10461, 10463, 10464,
 10528, 10530, 10533, 10536, 10617,
 10623, 10654, 10735, 10738, 10751
 \l_fp_input_a_integer_int
 7744, 7745, 7794, 7975,
 7978, 7985, 8024, 8038, 8066, 8080,
 8492, 8528, 8551, 8553, 8555, 8601,
 8627, 8696, 8712, 8825, 8830, 8834,
 8839, 8897, 8961, 8966, 8976, 8979,
 8994, 8997, 9019, 9020, 9191, 9201,
 9202, 9229, 9234, 9237, 9298, 9300,
 9313, 9318, 9331, 9332, 9342, 9345,
 9351, 9353, 9355, 9380, 9382, 9388,
 9390, 9393, 9401, 9405, 9423, 9433,
 9520, 9530, 9721, 9731, 9790, 9812,
 9817, 9821, 9914, 9925, 9957, 9967,
 9982, 9994, 10004, 10006, 10008,
 10033, 10037, 10039, 10041, 10061,
 10063, 10066, 10232, 10238, 10264,
 10415, 10421, 10432, 10435, 10448,
 10455, 10565, 10571, 10580, 10597,
 10657, 10705, 10716, 10737, 10749,
 10782, 10798, 10816, 10842, 10902,
 10907, 10934, 10939, 10962, 10971
 \l_fp_input_a_sign_int
 7744, 7744, 7790, 7792,
 8023, 8033, 8065, 8075, 8487, 8523,
 8622, 8659, 8693, 8737, 8789, 8932,
 9299, 9304, 9346, 9422, 9428, 9476,
 9483, 9519, 9525, 9562, 9569, 9595,
 9597, 9602, 9720, 9726, 9775, 9816,
 9913, 9920, 9956, 10010, 10043,
 10062, 10095, 10118, 10159, 10231,
 10235, 10564, 10570, 10649, 10703,
 10748, 10781, 10797, 10815, 10862,
 10876, 10880, 10884, 10960, 10969
 \l_fp_input_b_decimal_int 7744,
 7750, 7961, 7962, 7967, 7969, 8649,
 8700, 8716, 8752, 8770, 8817, 8883,
 8887, 8982, 8995, 9804, 9809, 10119,
 10162, 10163, 10165, 10167, 10170,
 10173, 10189, 10451, 10465, 10529,
 10566, 10576, 10709, 10717, 10727,
 10739, 10789, 10799, 10817, 10845,
 10860, 10912, 10944, 10965, 10972
 \l_fp_input_b_exponent_int
 7744, 7751, 7941,
 7949, 7970, 7974, 8650, 8753, 8771,
 8775, 8884, 8918, 9805, 9810, 9840,
 10452, 10567, 10710, 10718, 10731,
 10745, 10790, 10800, 10819, 10894,
 10898, 10926, 10930, 10967, 10974
 \l_fp_input_b_extended_int
 7752, 7753, 10120,
 10162, 10163, 10165, 10167, 10170,
 10175, 10465, 10529, 10729, 10740
 \l_fp_input_b_integer_int 7744,
 7749, 7950, 7953, 7960, 8648, 8696,
 8712, 8751, 8769, 8828, 8832, 8835,
 8839, 8882, 8887, 8976, 8979, 8994,
 9803, 9808, 10450, 10565, 10576,
 10708, 10716, 10725, 10739, 10788,
 10798, 10816, 10845, 10860, 10902,
 10907, 10934, 10939, 10963, 10970
 \l_fp_input_b_sign_int
 7744, 7748, 8647, 8659, 8723, 8750,
 8754, 8768, 8789, 8881, 8932, 9807,
 10118, 10159, 10172, 10564, 10613,
 10723, 10748, 10787, 10797, 10815,
 10850, 10876, 10880, 10961, 10968
 \l_fp_mul_a_i_int
 7754, 7754, 8816, 8821,
 8826, 8831, 8835, 9065, 9074, 9081,
 9087, 9092, 9098, 9101, 9109, 9118,
 9126, 9134, 9141, 9149, 9154, 9158
 \l_fp_mul_a_ii_int 7754,
 7755, 8816, 8822, 8827, 8832, 9065,
 9075, 9082, 9088, 9093, 9099, 9109,
 9119, 9127, 9135, 9142, 9150, 9155
 \l_fp_mul_a_iii_int
 7754, 7756, 8816, 8823,
 8828, 9065, 9076, 9083, 9089, 9094,
 9109, 9120, 9128, 9136, 9143, 9151

`\l_fp_mul_a_iv_int` 10496, 10502, 10505, 10512, 10520,
 ... [7754](#), 7757, 9067, 9077, 9084,
 9090, 9111, 9121, 9129, 9137, 9144
`\l_fp_mul_a_v_int` .. [7754](#), 7758, 9067,
 9078, 9085, 9111, 9122, 9130, 9138
`\l_fp_mul_a_vi_int` [7754](#),
 7759, 9067, 9079, 9111, 9123, 9131
`\l_fp_mul_b_i_int`
 [7754](#), 7760, 8818, 8823,
 8827, 8831, 8834, 9069, 9079, 9085,
 9090, 9094, 9099, 9101, 9113, 9123,
 9130, 9137, 9143, 9150, 9154, 9157
`\l_fp_mul_b_ii_int` [7754](#),
 7761, 8818, 8822, 8826, 8830, 9069,
 9078, 9084, 9089, 9093, 9098, 9113,
 9122, 9129, 9136, 9142, 9149, 9153
`\l_fp_mul_b_iii_int`
 [7754](#), 7762, 8818, 8821,
 8825, 9069, 9077, 9083, 9088, 9092,
 9113, 9121, 9128, 9135, 9141, 9148
`\l_fp_mul_b_iv_int`
 ... [7754](#), 7763, 9071, 9076, 9082,
 9087, 9115, 9120, 9127, 9134, 9140
`\l_fp_mul_b_v_int` .. [7754](#), 7764, 9071,
 9075, 9081, 9115, 9119, 9126, 9133
`\l_fp_mul_b_vi_int` [7754](#),
 7765, 9071, 9074, 9115, 9118, 9125
`\l_fp_mul_output_int` [7766](#),
 7766, 8819, 8824, 8857, 8858, 8862,
 8864, 8869, 9072, 9080, 9116, 9124
`\l_fp_mul_output_tl`
 ... [7766](#), 7767, 8820, 8837, 8838,
 8841, 8868, 9073, 9096, 9097, 9104,
 9117, 9146, 9147, 9160, 9161, 9164
`\l_fp_output_decimal_int`
 [7768](#), 7770, 8669, 8685, 8698, 8702,
 8705, 8714, 8718, 8720, 8724, 8727,
 8729, 8780, 8793, 8806, 8837, 8912,
 8923, 8936, 8949, 9010, 9012, 9263,
 9268, 9276, 9306, 9471, 9472, 9478,
 9492, 9557, 9558, 9564, 9578, 9610,
 9625, 9629, 9634, 9638, 9646, 9648,
 9680, 9685, 9689, 9692, 9696, 9700,
 9703, 9705, 9803, 9812, 9834, 9845,
 9859, 9986, 10030, 10050, 10052,
 10070, 10072, 10125, 10127, 10132,
 10133, 10135, 10142, 10151, 10288,
 10289, 10291, 10298, 10311, 10330,
 10342, 10353, 10355, 10407, 10410,
 10456, 10459, 10460, 10473, 10493,
 10496, 10502, 10505, 10512, 10520,
 10539, 10543, 10547, 10550, 10695,
 10709, 10728, 10741, 10750, 10754
`\l_fp_output_exponent_int` [7768](#), 7771,
 8665, 8670, 8687, 8773, 8781, 8808,
 8916, 8924, 8952, 9290, 9307, 9469,
 9473, 9479, 9494, 9555, 9559, 9565,
 9580, 9838, 9846, 9861, 9988, 10032,
 10054, 10074, 10143, 10153, 10299,
 10314, 10332, 10344, 10362, 10369,
 10458, 10477, 10501, 10522, 10697,
 10710, 10732, 10743, 10752, 10756
`\l_fp_output_extended_int`
 ... [7772](#), 7772, 9281, 9282, 9287,
 9289, 9472, 9558, 9626, 9630, 9635,
 9637, 9649, 9681, 9683, 9686, 9697,
 9699, 9701, 9804, 9813, 9987, 10031,
 10051, 10053, 10071, 10073, 10126,
 10128, 10130, 10286, 10331, 10343,
 10354, 10356, 10408, 10411, 10461,
 10475, 10494, 10497, 10540, 10541,
 10544, 10730, 10742, 10751, 10755
`\l_fp_output_integer_int` [7768](#),
 7769, 8668, 8681, 8694, 8704, 8710,
 8719, 8722, 8725, 8731, 8733, 8779,
 8793, 8802, 8841, 8911, 8922, 8936,
 8945, 9005, 9251, 9252, 9255, 9262,
 9305, 9468, 9471, 9477, 9488, 9554,
 9557, 9563, 9574, 9609, 9624, 9628,
 9633, 9644, 9691, 9704, 9833, 9844,
 9855, 9985, 10011, 10029, 10050,
 10052, 10070, 10072, 10125, 10127,
 10136, 10141, 10147, 10292, 10297,
 10307, 10329, 10341, 10353, 10355,
 10407, 10410, 10455, 10493, 10496,
 10511, 10516, 10519, 10549, 10691,
 10708, 10726, 10741, 10749, 10753
`\l_fp_output_sign_int` [7768](#),
 7768, 8667, 8676, 8693, 8723, 8737,
 8778, 8921, 9777, 9779, 9783, 9785,
 9843, 9850, 10140, 10296, 10302,
 10321, 10323, 10402, 10488, 10724
`\l_fp_round_carry_bool` [7773](#),
 7773, 8540, 8550, 8563, 8569, 8577
`\l_fp_round_decimal_tl` [7774](#),
 7774, 8542, 8552, 8571, 8572, 8574
`\l_fp_round_position_int` [7775](#),
 7775, 8541, 8562, 8575, 8581, 8582
`\l_fp_round_target_int` .. [7775](#), 7776,
 8477, 8478, 8512, 8514, 8562, 8575

\l_fp_sign_tl 7777, 7777, 10614, 10626, 10690
\l_fp_split_sign_int 7778, 7778, 7803, 7805, 7818
\l_fp_tmp_dim 8050, 8058, 8062, 8098
\l_fp_tmp_int 7779, 7779, 7847, 7849, 8564–8567, 8572, 9168–9170, 9175–9177
\l_fp_tmp_skip ... 8050, 8057, 8058, 8099
\l_fp_tmp_tl 7780, 7780, 7800–7802, 7806, 7811, 7813, 7816, 7822, 7824, 7827, 7916, 7921, 7963, 7969, 7988, 7994, 8620, 8635, 9232, 9238, 9241, 9246, 9264, 9271, 9283, 9289, 10001, 10008, 10015, 10021, 10034, 10041, 10044, 10046, 10377, 10383, 10386, 10391, 10514, 10520, 10958, 10977, 10983, 10988
\l_fp_trig_decimal_int 7782, 7783, 9616, 9618, 9620, 9623, 9638, 9651, 9661, 9663, 9665, 9667, 9669, 9671, 9674, 9676, 9678, 9680, 9696
\l_fp_trig_extended_int 7782, 7784, 9616, 9618, 9620, 9622, 9637, 9652, 9661, 9663, 9665, 9667, 9669, 9671, 9674, 9676, 9682
\l_fp_trig_octant_int 7781, 7781, 9370, 9376, 9395, 9407, 9499, 9585, 9596, 9600, 9776, 9782
\l_fp_trig_sign_int 7782, 7782, 9612, 9650, 9659, 9679
\l_ior_stream_int 6135, 6136, 6154, 6156, 6160, 6191, 6234, 6235, 6239, 6242, 6249, 6251, 6257, 6258, 6260, 6262
\l_iow_current_line_int . 6366, 6366, 6378, 6416, 6417, 6433, 6440, 6447
\l_iow_current_line_tl 6368, 6368, 6379, 6428, 6431, 6439, 6441, 6446, 6448, 6455
\l_iow_current_word_int 6366, 6367, 6414, 6416, 6440
\l_iow_current_word_tl 6368, 6369, 6403, 6404, 6407, 6415, 6428, 6432, 6441
\l_iow_line_length_int 156, 6363, 6363, 6364, 6377
\l_iow_line_start_bool 6373, 6373, 6381, 6425, 6427, 6449
\l_iow_stream_int 6135, 6135, 6136, 6167, 6169, 6173, 6183, 6197, 6198, 6202, 6205, 6207, 6212, 6214, 6220, 6221, 6223, 6225, 6244
\l_iow_target_length_int 6365, 6365, 6377, 6418
\l_iow_wrap_tl 6370, 6370, 6380, 6386, 6389, 6395
\l_iow_wrapped_tl 6371, 6371, 6399, 6438, 6445, 6454
\l_keys_choice_int 175, 7181, 7181, 7282, 7295, 7296, 7300
\l_keys_choice_tl 175, 7294
\l_keys_choices_tl 7181, 7182
\l_keys_key_tl 176, 7183, 7183, 7262, 7456, 7457
\l_keys_module_tl 7184, 7184, 7190, 7193, 7195, 7219, 7335, 7340, 7435, 7438, 7440, 7457, 7504, 7507
\l_keys_no_value_bool 7185, 7185, 7200, 7205, 7236, 7446, 7451, 7462, 7472, 7484
\l_keys_path_tl 176, 7186, 7186, 7214, 7219, 7226, 7229, 7244, 7255, 7257, 7259, 7268, 7270, 7273, 7279, 7287, 7292, 7298, 7305, 7308, 7310, 7330, 7334, 7339, 7345, 7351, 7365, 7367, 7457, 7466, 7476, 7486, 7489, 7496, 7501, 7507, 7521, 7522
\l_keys_property_tl . 7187, 7187, 7210, 7214, 7232, 7239, 7240, 7243, 7247
\l_keys_value_tl 7188, 7188, 7476, 7483, 7488, 7515
\l_keyval_key_tl 7051, 7051, 7097, 7112, 7117, 7118, 7129
\l_keyval_parse_tl . 7053, 7054, 7070, 7074, 7094, 7126, 7133, 7137, 7149
\l_keyval_sanitise_tl 7053, 7053, 7066–7069, 7072
\l_keyval_value_tl 7051, 7052, 7131, 7132, 7135, 7147, 7149
\l_last_box 147, 6008, 6008, 6010
\l_msg_class_tl 6770, 6771, 6821, 6822, 6825
\l_msg_current_class_tl 6770, 6772, 6803, 6822
\l_msg_current_module_tl 6770, 6773, 6804
\l_msg_redirect_classes_prop 6677, 6677
\l_msg_redirect_classes_seq 6770, 6770, 6778, 6783, 6786
\l_msg_redirect_kernel_info_prop . 6922

<code>\l_msg_redirect_kernel_warning_prop</code>	6900	<code>\leaders</code>	536
<code>\l_msg_redirect_names_prop</code>	6677, 6678, 6805, 6836	<code>\left</code>	504
<code>\l_msg_text_tl</code>	6616, 6649, 6651	<code>\lefthyphenmin</code>	560
<code>\l_msg_tmp_tl</code>	6514, 6514, 6620, 6623, 6631	<code>\leftskip</code>	562
<code>\l_peek_search_tl</code>	2869, 2869, 2887, 2908, 2951	<code>\leqno</code>	479
<code>\l_peek_search_token</code>	2868, 2868, 2886, 2907, 2926, 2934	<code>\let</code>	59, 230, 336, 337, 349
<code>\l_peek_token</code>	64, 2866, 2866, 2875, 2926, 2934, 2944–2946, 2965	<code>\limits</code>	496
<code>\l_prop_show_tl</code>	5817, 5817, 5831, 5834, 6304, 6307, 6326, 6329	<code>\linepenalty</code>	552
<code>\l_seq_remove_seq</code>	4821, 4821, 4828, 4831, 4832, 4834	<code>\lineskip</code>	546
<code>\l_seq_show_tl</code>	5061, 5061, 5075, 5078	<code>\lineskiplimit</code>	547
<code>\l_seq_tmpa_tl</code>	4779, 4779, 4853, 4858, 4872, 4876	<code>\long</code>	33, 339, 368
<code>\l_seq_tmpb_tl</code>	4779, 4780, 4849, 4853, 4875, 4876	<code>\looseness</code>	564
<code>\l_tl_replace_tl</code>	4349, 4349, 4360, 4364, 4367, 4381, 4389, 4394, 4401	<code>\lower</code>	601
<code>\l_tl_rescan_tl</code>	4294, 4294, 4305, 4308, 4314, 4331, 4333, 4345	<code>\lowercase</code>	640
<code>\l_tl_tmpa_tl</code>	4472, 4475, 4477, 4485	<code>\lua_now:n</code>	189, 11070, 11079
<code>\l_tl_tmpb_tl</code>	4472, 4476, 4477, 4486	<code>\lua_now:x</code>	189, 11070, 11072, 11076, 11080
<code>\l_tmpa_bool</code>	41, 1909, 1909	<code>\lua_shipout:n</code>	190, 11070, 11082, 11084
<code>\l_tmpa_box</code>	6021, 6022, 6025	<code>\lua_shipout:x</code>	190, 11070
<code>\l_tmpa_dim</code>	88, 4035, 4035	<code>\lua_shipout_x:n</code>	190, 11070, 11073, 11077, 11081, 11083
<code>\l_tmpa_int</code>	80, 3822, 3822	<code>\lua_shipout_x:x</code>	190, 11070
<code>\l_tmpa_skip</code>	91, 4112, 4112	<code>\lua_wrong_engine:</code>	11070, 11076, 11077, 11088, 11115, 11146, 11147, 11165
<code>\l_tmpa_tl</code>	5, 109, 147, 4683, 4683	<code>\luaescapestring</code>	39, 40
<code>\l_tmpb_box</code>	6021, 6027	<code>\LuaTeX</code>	11089
<code>\l_tmpb_dim</code>	88, 4035, 4036	<code>\luatex_catcodetable:D</code>	755, 770, 11128, 11129, 11134, 11142
<code>\l_tmpb_int</code>	80, 3822, 3823	<code>\luatex_directlua:D</code>	756, 1471, 11072
<code>\l_tmpb_skip</code>	91, 4112, 4113	<code>\luatex_if_engine:F</code>	1449, 1474, 11114, 11144, 11164
<code>\l_tmpb_tl</code>	109, 147, 4683, 4684	<code>\luatex_if_engine:T</code>	1448, 1473, 11117, 11150, 11166, 11172, 11181
<code>\l_tmppc_dim</code>	88, 4035, 4037	<code>\luatex_if_engine:TF</code>	24, 1448, 1450, 1475, 4670, 11070
<code>\l_tmppc_int</code>	80, 3822, 3824	<code>\luatex_initcatcodetable:D</code>	757, 771, 11105, 11122
<code>\l_tmppc_skip</code>	91, 4112, 4114	<code>\luatex_latelua:D</code>	758, 772, 11073
<code>\language</code>	449	<code>\luatex luatexversion:D</code>	759
<code>\lastbox</code>	606	<code>\luatex_savecatcodetable:D</code>	760, 773, 11133, 11161
<code>\lastkern</code>	539	<code>\luatexcatcodetable</code>	770
<code>\lastlinefit</code>	719	<code>\luatexinitcatcodetable</code>	771
<code>\lastnodetype</code>	700	<code>\luatexlatelua</code>	772
<code>\lastpenalty</code>	645	<code>\luatexsavecatcodetable</code>	773
<code>\lastskip</code>	540	<code>\luatexversion</code>	759
<code>\latelua</code>	758		
<code>\lccode</code>	668		

M

<code>\M</code>	2675, 2718
<code>\m@ne</code>	1157
<code>\mag</code>	448

<code>\mark</code>	450	<code>\msg_critical:nxxx</code>	162, 6715
<code>\marks</code>	676	<code>\msg_critical:nxxxx</code>	162, 6715
<code>\mathaccent</code>	461	<code>\msg_critical:nxxxxx</code>	162, 6715
<code>\mathbin</code>	491	<code>\msg_critical_text:n</code> 160, 6670, 6671, 6718	
<code>\mathchar</code>	462	<code>\msg_direct_interrupt:xxxxx</code> .	7036, 7040
<code>\mathchardef</code>	359	<code>\msg_direct_log:xx</code>	7036, 7041
<code>\mathchoice</code>	459	<code>\msg_direct_term:xx</code>	7036, 7042
<code>\mathclose</code>	492	<code>\msg_error:nn</code>	162, 6726
<code>\mathcode</code>	670	<code>\msg_error:nxx</code>	162, 6726
<code>\mathinner</code>	493	<code>\msg_error:nxxx</code>	162, 6726
<code>\mathop</code>	494	<code>\msg_error:nxxxx</code>	162, 6726
<code>\mathopen</code>	498	<code>\msg_error:nxxxxx</code>	162, 6726
<code>\mathord</code>	499	<code>\msg_error_text:n</code>	
<code>\mathparagraph</code>	3846		161, 6670, 6672, 6731, 6740, 6870, 6883
<code>\mathpunct</code>	500	<code>\msg_fatal:nn</code>	162, 6704
<code>\mathrel</code>	501	<code>\msg_fatal:nxx</code>	162, 6704
<code>\mathsection</code>	3845	<code>\msg_fatal:nxxx</code>	162, 6704
<code>\mathsurround</code>	512	<code>\msg_fatal:nxxxx</code>	162, 6704
<code>\maxdeadcycles</code>	582	<code>\msg_fatal:nxxxxx</code>	162, 6704
<code>\maxdepth</code>	583	<code>\msg_fatal_text:n</code>	
<code>\maxdimen</code>	4033		160, 6670, 6670, 6707, 6848
<code>\meaning</code>	642	<code>\msg_generic_new:nn</code>	7036, 7037
<code>\medmuskip</code>	513	<code>\msg_generic_new:nnn</code>	7036, 7036
<code>\message</code>	419	<code>\msg_generic_set:nn</code>	7036, 7039
<code>\MessageBreak</code>	222, 238–244	<code>\msg_generic_set:nnn</code>	7036, 7038
<code>.meta:n</code>	173	<code>\msg_if_more_text:cTF</code> ..	6693, 6728, 6867
<code>.meta:x</code>	173	<code>\msg_if_more_text:N</code>	6693
<code>\middle</code>	724	<code>\msg_if_more_text:NF</code>	6702
<code>\mkern</code>	466	<code>\msg_if_more_text:NT</code>	6701
<code>\mode_if_horizontal:</code>	2242	<code>\msg_if_more_text:NTF</code>	6693, 6703
<code>\mode_if_horizontal:TF</code>	46, 2242	<code>\msg_if_more_text_p:c</code>	6693
<code>\mode_if_horizontal_p:</code>	46, 2242	<code>\msg_if_more_text_p:N</code>	6693, 6700
<code>\mode_if_inner:</code>	2244	<code>\msg_info:nn</code>	163, 6756
<code>\mode_if_inner:TF</code>	46, 2244	<code>\msg_info:nxx</code>	163, 6756
<code>\mode_if_inner_p:</code>	46, 2244	<code>\msg_info:nxxx</code>	163, 6756
<code>\mode_if_math:</code>	2246	<code>\msg_info:nxxxx</code>	163, 6756
<code>\mode_if_math:TF</code>	47, 2246, 3834	<code>\msg_info:nxxxxx</code>	163, 6756
<code>\mode_if_math_p:</code>	2246	<code>\msg_info_text:n</code> 161, 6670, 6674, 6760, 6929	
<code>\mode_if_vertical:</code>	2240	<code>\msg_interrupt:xxx</code>	
<code>\mode_if_vertical:TF</code>	47, 2240		164, 6584, 6584, 6706, 6717,
<code>\mode_if_vertical_p:</code>	47, 2240		6730, 6739, 6847, 6869, 6882, 7015
<code>\month</code>	652	<code>\msg_interrupt_aux:</code>	6584, 6590, 6640
<code>\moveleft</code>	602	<code>\msg_interrupt_details:xxx</code>	
<code>\moveright</code>	603		6584, 6589, 6607
<code>\msg_class_new:nn</code>	7030, 7030	<code>\msg_interrupt_more_text:n</code>	
<code>\msg_class_set:nn</code>			6584, 6603, 6611, 6617
	161, 6679, 6679, 6704, 6715,	<code>\msg_interrupt_no_details:xx</code>	
	6726, 6748, 6756, 6764, 6769, 7030		6584, 6588, 6599
<code>\msg_critical:nn</code>	162, 6715	<code>\msg_interrupt_text:n</code>	
<code>\msg_critical:nxx</code>	162, 6715		6584, 6605, 6613, 6615

\msg_kernel_bug:x 7013, 7013
 \msg_kernel_error:nn
 166, 1173, 1187, 6157,
 6170, 6521, 6784, 6865, 6898, 7103,
 11039, 11047, 11052, 11061, 11132
 \msg_kernel_error:nnx 166,
 1173, 1185, 1443, 4977, 6793, 6865,
 6896, 7222, 7261, 7286, 7465, 11110
 \msg_kernel_error:nnxx
 166, 1173, 1173, 1186, 1188,
 1196, 1206, 1219, 1353, 6799, 6865,
 6894, 7213, 7242, 7272, 7475, 7506
 \msg_kernel_error:nnxxx 166, 6865, 6892
 \msg_kernel_error:nnxxxx
 166, 6865, 6865, 6893, 6895, 6897, 6899
 \msg_kernel_fatal:nn ... 166, 6845, 6863
 \msg_kernel_fatal:nnx
 166, 6845, 6861, 11107
 \msg_kernel_fatal:nnxx . 166, 6845, 6859
 \msg_kernel_fatal:nnxxx 166, 6845, 6857
 \msg_kernel_fatal:nnxxxx
 166, 6845, 6845, 6858, 6860, 6862, 6864
 \msg_kernel_info:nn 167, 6900, 6942
 \msg_kernel_info:nnx ... 167, 6900, 6940
 \msg_kernel_info:nnxx . 167, 6900, 6938
 \msg_kernel_info:nnxxx . 167, 6900, 6936
 \msg_kernel_info:nnxxxx
 167, 6900, 6923, 6937, 6939, 6941, 6943
 \msg_kernel_new:nnn 165, 6837, 6839
 \msg_kernel_new:nnnn
 165, 6333, 6340, 6837, 6837,
 6944, 6953, 6961, 6968, 6975, 6983,
 6992, 6999, 7006, 7160, 7532, 7535,
 7541, 7548, 7557, 7563, 7569, 7576,
 7583, 7589, 11032, 11040, 11048, 11054
 \msg_kernel_set:nnn 166, 6837, 6843
 \msg_kernel_set:nnnn ... 166, 6837, 6841
 \msg_kernel_warning:nn . 166, 6900, 6920
 \msg_kernel_warning:nnx 166, 6900, 6918
 \msg_kernel_warning:nnxx 166, 6900, 6916
 \msg_kernel_warning:nnxxx 166, 6900, 6914
 \msg_kernel_warning:nnxxxx
 166, 6900, 6901, 6915, 6917, 6919, 6921
 \msg_line_context:
 ... 160, 1189, 1189, 1209, 6571, 6572
 \msg_line_number:
 160, 6571, 6571, 6576, 7161
 \msg_log:nn 163, 6764, 7035
 \msg_log:nnx 163, 6764, 7034
 \msg_log:nnxx 163, 6764, 7033
 \msg_log:nnxxx 163, 6764, 7032
 \msg_log:nnxxxx 163, 6764, 7031
 \msg_log:x 165, 6655, 6655, 6758, 6766, 6927
 \msg_new:nnn 159, 6517, 6526, 6840
 \msg_new:nnnn . 159, 6517, 6517, 6527, 6838
 \msg_newline: 164, 6569, 6569, 6628
 \msg_no_more_text:xxxx . 6693, 6695, 6699
 \msg_none:nn 163, 6769
 \msg_none:nnx 163, 6769
 \msg_none:nnxx 163, 6769
 \msg_none:nnxxx 163, 6769
 \msg_none:nnxxxx 163, 6769
 \msg_redirect_class:nn . 163, 6831, 6831
 \msg_redirect_module:nnn 163, 6833, 6833
 \msg_redirect_name:nnn . 164, 6835, 6835
 \msg_see_documentation_text:n
 6675, 6675, 6710, 6721,
 6734, 6743, 6852, 6874, 6887, 7018
 \msg_set:nnn 159, 6517, 6535, 6844
 \msg_set:nnnn
 ... 159, 6517, 6524, 6528, 6536, 6842
 \msg_term:x ... 165, 6655, 6662, 6750, 6905
 \msg_trace:nn 7031, 7035
 \msg_trace:nnx 7031, 7034
 \msg_trace:nnxx 7031, 7033
 \msg_trace:nnxxx 7031, 7032
 \msg_trace:nnxxxx 7031, 7031
 \msg_two_newlines: 164, 6569, 6570
 \msg_use:nnnnnnnn
 6683, 6774, 6774, 6903, 6925
 \msg_use_aux:nn 6774, 6807, 6809
 \msg_use_aux:nnn 6774, 6798, 6801
 \msg_use_code: 6774, 6776, 6816, 6824, 6828
 \msg_use_loop:n . 6774, 6781, 6829, 6830
 \msg_use_loop:o 6774, 6825
 \msg_use_loop_check:nn
 6774, 6806, 6812, 6815, 6819
 \msg_warning:nn 162, 6748
 \msg_warning:nnx 162, 6748
 \msg_warning:nnxx 162, 6748
 \msg_warning:nnxxx 162, 6748
 \msg_warning:nnxxxx 162, 6748
 \msg_warning_text:n
 161, 6670, 6673, 6752, 6907
 \mskip 463
 \muexpr 712
 \multiply 364
 \muskip 660
 \muskip_add:cn 92, 4144
 \muskip_add:Nn . 92, 4144, 4144, 4146, 4147

\muskip_eval:n	93, 4154, 4154	\noindent	543
\muskip_gadd:cn	92, 4144	\nolimits	497
\muskip_gadd:Nn	92, 4144, 4146, 4148	\nonscript	477
\muskip_gset:cn	92, 4133	\nonstopmode	440
\muskip_gset:Nn	92, 4133, 4135, 4137	int_get_digits:n	80
\muskip_gset_eq:cc	93, 4138	int_get_sign:n	80
\muskip_gset_eq:cN	93, 4138	int_to_letter:n	80
\muskip_gset_eq:Nc	93, 4138	\nulldelimiterspace	510
\muskip_gset_eq:NN	93, 4138, 4141–4143	\nullfont	623
\muskip_gsub:cn	93, 4144	\number	637
\muskip_gsub:Nn	93, 4144, 4151, 4153	\numexpr	709
\muskip_gzero:c	92, 4128		
\muskip_gzero:N	92, 4128, 4130, 4132	O	
\muskip_new:c	92, 4117	\O	1816, 2718
\muskip_new:N	92, 4117, 4121, 4127	\omit	383
\muskip_set:cn	92, 4133	\openin	409
\muskip_set:Nn	92, 4133, 4133, 4135, 4136	\openout	410
\muskip_set_eq:cc	93, 4138	\or	406
\muskip_set_eq:cN	93, 4138	\or:	25, 81, 784, 786, 884, 1322, 1324, 1326, 1328, 1330, 1332, 1334, 1336, 1338, 9500, 9502, 9504, 9506, 9586, 9588, 9590, 9592
\muskip_set_eq:Nc	93, 4138	\outer	369
\muskip_set_eq:NN	93, 4138, 4138–4140	\output	584
\muskip_show:c	94, 4158	\outputpenalty	594
\muskip_show:N	94, 4158, 4158, 4159	\over	471
\muskip_sub:cn	93, 4144	\overfullrule	622
\muskip_sub:Nn	93, 4144, 4149, 4151, 4152	\overline	502
\muskip_use:c	93, 4156	\overwithdelims	472
\muskip_use:N	93, 4155, 4156, 4156, 4157	P	
\muskip_zero:c	92, 4128	\P	1814, 2718
\muskip_zero:N	92, 4128, 4128, 4130, 4131	\package_check_loaded_expl:	780, 1522, 1867, 2380, 2485, 3174, 3873, 4178, 4772, 5254, 5630, 5914, 6100, 6512, 7048, 7174, 7600, 7713, 11067
\muskipdef	358	\PackageError	219, 235
\mutogluue	718	\pagedepth	586
N		\pagediscards	727
\n	2714	\pagefillllstretch	590
\name_primitive:NN	339, 339, 346–760	\pagefillstretch	589
\negative_replication	2164	\pagefilstretch	588
\newbox	5924	\pagegoal	592
\newcatcodetable	11121	\pageshrink	591
\newcount	3253	\pagestretch	587
\newdimen	3885	\pagetotal	593
\newlinechar	249, 414	\par	542
\newmuskip	4124	\parfillskip	573
\newread	6147	\parindent	566
\newskip	4047	\parshape	558
\newwrite	6146		
\noalign	382		
\noboundary	517		
\noexpand	35, 39, 40, 166, 169, 172, 174, 175, 184, 187–189, 191, 193, 194, 203, 205–209, 268, 270, 275, 277, 375		

<code>\parshapedimen</code>	708	<code>\peek_catcode_ignore_spaces:N</code> TF 65 , 2991
<code>\parshapeindent</code>	706	<code>\peek_catcode_remove:N</code> TF 65 , 2991
<code>\parshapelength</code>	707	<code>\peek_catcode_remove_ignore_spaces:N</code> TF
<code>\parskip</code>	565	65 , 2991
<code>\patterns</code>	648	<code>\peek_charcode:N</code> TF 65 , 3007
<code>\pausing</code>	435	<code>\peek_charcode_ignore_spaces:N</code> TF ...
<code>\pdf@strcmp</code>	59	65 , 3007
<code>\pdfcolorstack</code>	738	<code>\peek_charcode_remove:N</code> TF 66 , 3007
<code>\pdfcompresslevel</code>	739	<code>\peek_charcode_remove_ignore_spaces:N</code> TF
<code>\pdfcreationdate</code>	737	66 , 3007
<code>\pdfdecimaldigits</code>	740	<code>\peek_def:n</code> nnnn 2974 , 2975 , 2991 , 2995 , 2999 , 3003 , 3007 , 3011 , 3015 , 3019 , 3023 , 3027 , 3031 , 3035
<code>\pdfhorigin</code>	741	<code>\peek_def_aux:n</code> nnnnn 2974 , 2977 – 2979 , 2981
<code>\pdfinfo</code>	742	<code>\peek_execute_branches:</code> 2970 , 2986
<code>\pdfliteral</code>	743	<code>\peek_execute_branches_catcode:</code>
<code>\pdfminorversion</code>	744	2923 , 2923 , 2994 , 2996 , 3002 , 3004
<code>\pdfobjcompresslevel</code>	745	<code>\peek_execute_branches_charcode:</code> ...
<code>\pdfoutput</code>	746	2940 , 2940 , 3010 , 3012 , 3018 , 3020
<code>\pdfpkresolution</code>	750	<code>\peek_execute_branches_charcode:NN</code> 2940
<code>\pdfrestore</code>	747	<code>\peek_execute_branches_charcode_aux:NN</code>
<code>\pdfsave</code>	748	2950 , 2954
<code>\pdfsetmatrix</code>	749	<code>\peek_execute_branches_meaning:</code>
<code>\pdfstrcmp</code> . 33 , 59 , 230 , 235 , 238 , 252 , 753		2923 , 2932 , 3026 , 3028 , 3034 , 3036
<code>\pdftex_if_engine:F</code>	1452 , 1463 , 1477	<code>\peek_false:w</code> 2870 , 2872 , 2893 , 2911 , 2929 , 2937 , 2948 , 2959
<code>\pdftex_if_engine:T</code>	1451 , 1462 , 1476	<code>\peek_gafter:NN</code> 3110 , 3111
<code>\pdftex_if_engine:TF</code>	24 , 1448 , 1453 , 1464 , 1478	<code>\peek_gafter:Nw</code> 64 , 2876 , 3111
<code>\pdftex_pdfcolorstack:D</code>	738	<code>\peek_ignore_spaces_execute_branches:</code>
<code>\pdftex_pdfcompresslevel:D</code>	739	2963 , 2963 , 2973 , 2998 , 3006 , 3014 , 3022 , 3030 , 3038
<code>\pdftex_pdfcreationdate:D</code>	737	<code>\peek_ignore_spaces_execute_branches_aux:</code>
<code>\pdftex_pdfdecimaldigits:D</code>	740	2963 , 2967 , 2972
<code>\pdftex_pdfhorigin:D</code>	741	<code>\peek_meaning:N</code> TF 66 , 3023
<code>\pdftex_pdfinfo:D</code>	742	<code>\peek_meaning_ignore_spaces:N</code> TF 66 , 3023
<code>\pdftex_pdfliteral:D</code>	743	<code>\peek_meaning_remove:N</code> TF 66 , 3023
<code>\pdftex_pdfminorversion:D</code>	744	<code>\peek_meaning_remove_ignore_spaces:N</code> TF
<code>\pdftex_pdfobjcompresslevel:D</code>	745	67 , 3023
<code>\pdftex_pdfoutput:D</code>	746	<code>\peek_tmp:w</code> 2870 , 2873 , 2882 , 2968
<code>\pdftex_pdfpkresolution:D</code>	750	<code>\peek_token_generic:NNF</code> 2903
<code>\pdftex_pdfrestore:D</code>	747	<code>\peek_token_generic:NNT</code> 2901
<code>\pdftex_pdfsave:D</code>	748	<code>\peek_token_generic:NNTF</code>
<code>\pdftex_pdfsetmatrix:D</code>	749	2884 , 2884 , 2902 , 2904
<code>\pdftex_pdftextrevision:D</code>	751	<code>\peek_token_remove_generic:NNF</code> .. 2921
<code>\pdftex_pdfvorigin:D</code>	752	<code>\peek_token_remove_generic:NNT</code> .. 2919
<code>\pdftex_strcmp:D</code> ...	753 , 1485 , 1491 , 2460 , 2469 , 2690 , 4079 , 7810 , 7821	<code>\peek_token_remove_generic:NNTF</code>
<code>\pdftexrevision</code>	751	2905 , 2905 , 2920 , 2922
<code>\pdfvorigin</code>	752	<code>\peek_true:w</code> 2870 , 2870 , 2888 , 2909 , 2927 , 2935 , 2957
<code>\peek_after:NN</code>	3110 , 3110	<code>\peek_true_aux:w</code> . 2870 , 2871 , 2881 , 2910
<code>\peek_after:Nw</code>	64 , 2874 , 2874 , 2899 , 2917 , 2973 , 3110	
<code>\peek_catcode:N</code> TF	64 , 2991	

- \peek_true_remove:w [2878](#), [2878](#), [2909](#)
- \penalty [643](#)
- \postdisplaypenalty [490](#)
- \predisplaydirection [734](#)
- \predisdisplaypenalty [489](#)
- \predisplaysize [488](#)
- \pref_global:D [27](#),
[817](#), [817](#), [1287](#), [1292](#), [2877](#), [3268](#),
[3279](#), [3285](#), [3293](#), [3295](#), [3305](#), [3307](#),
[3314](#), [3890](#), [3895](#), [3901](#), [3918](#), [3923](#),
[4052](#), [4057](#), [4063](#), [4068](#), [4073](#), [4130](#),
[4135](#), [4141](#), [4146](#), [4151](#), [4738](#), [5951](#),
[5957](#), [6012](#), [6032](#), [6038](#), [6044](#), [6066](#),
[6072](#), [6078](#), [6084](#), [6492](#), [6496](#), [11104](#)
- \pref_long:D [27](#), [817](#), [818](#),
[827](#), [829](#), [835](#), [837](#), [841](#), [843](#), [849](#), [851](#)
- \pref_protected:D [27](#), [817](#), [819](#), [826](#), [828](#),
[830–833](#), [835](#), [837](#), [845](#), [847](#), [849](#), [851](#)
- \pretolerance [569](#)
- \prevdepth [616](#)
- \prevgraf [575](#)
- \prg_case_dim:nnn [44](#), [2099](#), [2099](#)
- \prg_case_dim_aux:nnn .. [2099](#), [2100](#), [2101](#)
- \prg_case_dim_aux:nw [2099](#), [2102](#), [2103](#), [2107](#)
- \prg_case_end:nw [2088](#),
[2088](#), [2096](#), [2106](#), [2114](#), [2123](#), [2131](#)
- \prg_case_int:nnn
..... [44](#), [2089](#), [2089](#), [3476](#), [3480](#), [3577](#)
- \prg_case_int_aux:nnn .. [2089](#), [2090](#), [2091](#)
- \prg_case_int_aux:nw [2089](#), [2092](#), [2093](#), [2097](#)
- \prg_case_str:nnn .. [45](#), [2109](#), [2109](#), [2117](#)
- \prg_case_str:onn [45](#), [2109](#)
- \prg_case_str:xxn [45](#), [2109](#), [2118](#)
- \prg_case_str_aux:nw
..... [2109](#), [2110](#), [2111](#), [2115](#), [2124](#)
- \prg_case_str_x_aux:nw . [2109](#), [2119](#), [2120](#)
- \prg_case_tl:cnn [45](#), [2126](#)
- \prg_case_tl:Nnn ... [45](#), [2126](#), [2126](#), [2134](#)
- \prg_case_tl_aux:Nw [2126](#), [2127](#), [2128](#), [2132](#)
- \prg_conditional_form_F:nnn [1060](#)
- \prg_conditional_form_p:nnn [1057](#)
- \prg_conditional_form_T:nnn [1059](#)
- \prg_conditional_form_TF:nnn [1058](#)
- \prg_define_quicksort:nnn [2295](#), [2295](#), [2370](#)
- \prg_do_nothing:
..... [7](#), [1482](#), [1482](#), [4366](#), [4370](#),
[4396](#), [4399](#), [4571](#), [4707](#), [7816](#), [7827](#)
- \prg_generate_conditional_aux:nnNNnnnn
..... [903](#),
[911](#), [918](#), [926](#), [934](#), [943](#), [951](#), [960](#), [971](#)
- \prg_generate_conditional_aux:nnw ..
..... [973](#), [979](#), [985](#)
- \prg_generate_conditional_parm_aux:nnNNnnnn
..... [971](#)
- \prg_generate_conditional_parm_aux:nw
..... [971](#)
- \prg_generate_F_form_count:Nnnnn ...
..... [1016](#), [1032](#)
- \prg_generate_F_form_parm:Nnnnn
..... [987](#), [1003](#)
- \prg_generate_p_form_count:Nnnnn ...
..... [1016](#), [1016](#)
- \prg_generate_p_form_parm:Nnnnn [987](#), [987](#)
- \prg_generate_T_form_count:Nnnnn ...
..... [1016](#), [1024](#)
- \prg_generate_T_form_parm:Nnnnn [987](#), [995](#)
- \prg_generate_TF_form_count:Nnnnn ..
..... [1016](#), [1040](#)
- \prg_generate_TF_form_parm:Nnnnn ...
..... [987](#), [1011](#)
- \prg_get_count_aux:nn
..... [932](#), [941](#), [950](#), [958](#), [969](#), [969](#)
- \prg_get_parm_aux:nw
..... [901](#), [909](#), [917](#), [924](#), [969](#), [970](#)
- \prg_new_conditional:Nnn ... [37](#), [930](#),
[939](#), [1869](#), [2438](#), [2446](#), [2458](#), [2467](#), [6477](#)
- \prg_new_conditional:Npnn [37](#), [899](#), [907](#),
[1421](#), [1483](#), [1489](#), [1869](#), [1897](#), [1911](#),
[2240](#), [2242](#), [2244](#), [2246](#), [2601](#), [2606](#),
[2611](#), [2616](#), [2623](#), [2629](#), [2634](#), [2639](#),
[2644](#), [2649](#), [2654](#), [2659](#), [2664](#), [2669](#),
[2683](#), [2697](#), [2702](#), [2723](#), [2730](#), [2737](#),
[2748](#), [2759](#), [2770](#), [2781](#), [2790](#), [2797](#),
[2815](#), [3319](#), [3389](#), [3397](#), [3405](#), [3930](#),
[3935](#), [4076](#), [4086](#), [4417](#), [4439](#), [4458](#),
[4460](#), [4508](#), [4514](#), [4633](#), [4641](#), [4654](#),
[4693](#), [4704](#), [4714](#), [5750](#), [5762](#), [5990](#),
[5992](#), [6002](#), [7493](#), [7524](#), [10760](#), [10768](#)
- \prg_new_eq_conditional:NN [39](#)
- \prg_new_eq_conditional:NNn
..... [965](#), [967](#), [1869](#),
[4864](#), [4866](#), [5420–5425](#), [5905–5908](#)
- \prg_new_map_functions:Nn ... [2373](#), [2373](#)
- \prg_new_protected_conditional:Nnn .
..... [38](#), [930](#), [956](#), [1869](#), [7663](#)
- \prg_new_protected_conditional:Npnn
..... [38](#), [899](#), [922](#),
[1869](#), [4472](#), [4493](#), [4868](#), [5095](#), [5103](#),
[5116](#), [5123](#), [5130](#), [5137](#), [5426](#), [5439](#),
[5846](#), [5857](#), [5863](#), [10776](#), [10793](#), [10980](#)

\prg_quicksort:n [48](#), [2370](#)
 \prg_quicksort_compare:nnTF
 [48](#), [2371](#), [2372](#)
 \prg_quicksort_function:n [48](#), [2371](#), [2371](#)
 \prg_replicate:nn
 [45](#), [2135](#), [2135](#), [8134](#), [8170](#), [8240](#)
 \prg_replicate_ [2135](#), [2146](#)
 \prg_replicate_0:n [2135](#)
 \prg_replicate_1:n [2135](#)
 \prg_replicate_2:n [2135](#)
 \prg_replicate_3:n [2135](#)
 \prg_replicate_4:n [2135](#)
 \prg_replicate_5:n [2135](#)
 \prg_replicate_6:n [2135](#)
 \prg_replicate_7:n [2135](#)
 \prg_replicate_8:n [2135](#)
 \prg_replicate_9:n [2135](#)
 \prg_replicate_aux:N [2135](#), [2142](#), [2143](#), [2145](#)
 \prg_replicate_first_-:n [2135](#)
 \prg_replicate_first_0:n [2135](#)
 \prg_replicate_first_1:n [2135](#)
 \prg_replicate_first_2:n [2135](#)
 \prg_replicate_first_3:n [2135](#)
 \prg_replicate_first_4:n [2135](#)
 \prg_replicate_first_5:n [2135](#)
 \prg_replicate_first_6:n [2135](#)
 \prg_replicate_first_7:n [2135](#)
 \prg_replicate_first_8:n [2135](#)
 \prg_replicate_first_9:n [2135](#)
 \prg_replicate_first_aux:N
 [2135](#), [2138](#), [2144](#)
 \prg_return_false: [39](#),
 [895](#), [897](#), [1103](#), [1108](#), [1121](#), [1126](#),
 [1134](#), [1151](#), [1424](#), [1487](#), [1492](#), [1869](#),
 [1902](#), [1916](#), [2241](#), [2243](#), [2245](#), [2249](#),
 [2443](#), [2451](#), [2464](#), [2473](#), [2604](#), [2609](#),
 [2614](#), [2619](#), [2626](#), [2632](#), [2637](#), [2642](#),
 [2647](#), [2652](#), [2657](#), [2662](#), [2667](#), [2672](#),
 [2693](#), [2700](#), [2707](#), [2709](#), [2729](#), [2736](#),
 [2740](#), [2747](#), [2751](#), [2758](#), [2769](#), [2773](#),
 [2780](#), [2789](#), [2796](#), [2805](#), [2818](#), [2837](#),
 [2854](#), [2863](#), [3338](#), [3346](#), [3352](#), [3362](#),
 [3370](#), [3376](#), [3384](#), [3394](#), [3402](#), [3408](#),
 [3933](#), [3941](#), [4083](#), [4094](#), [4432](#), [4443](#),
 [4455](#), [4465](#), [4482](#), [4497](#), [4511](#), [4517](#),
 [4638](#), [4647](#), [4658](#), [4702](#), [4719](#), [4725](#),
 [4727](#), [4881](#), [5091](#), [5432](#), [5445](#), [5755](#),
 [5771](#), [5850](#), [5861](#), [5867](#), [5991](#), [5993](#),
 [6003](#), [6484](#), [6696](#), [7498](#), [7528](#), [7667](#),
 [10765](#), [10773](#), [10810](#), [10824](#), [10828](#),
 [10832](#), [10836](#), [10848](#), [10852](#), [10867](#),
 [10882](#), [10900](#), [10909](#), [10917](#), [10932](#),
 [10941](#), [10949](#), [10994](#), [11000](#), [11006](#),
 [11012](#), [11018](#), [11024](#), [11029](#), [11030](#)
 \prg_return_true: [39](#), [895](#), [895](#), [1106](#),
 [1123](#), [1131](#), [1136](#), [1149](#), [1154](#), [1424](#),
 [1487](#), [1492](#), [1869](#), [1900](#), [1914](#), [2241](#),
 [2243](#), [2245](#), [2249](#), [2441](#), [2449](#), [2462](#),
 [2471](#), [2604](#), [2609](#), [2614](#), [2619](#), [2626](#),
 [2632](#), [2637](#), [2642](#), [2647](#), [2652](#), [2657](#),
 [2662](#), [2667](#), [2672](#), [2691](#), [2700](#), [2707](#),
 [2729](#), [2736](#), [2747](#), [2758](#), [2769](#), [2780](#),
 [2789](#), [2796](#), [2805](#), [2835](#), [2861](#), [3336](#),
 [3344](#), [3354](#), [3360](#), [3368](#), [3378](#), [3386](#),
 [3392](#), [3400](#), [3410](#), [3933](#), [3939](#), [4081](#),
 [4093](#), [4430](#), [4441](#), [4453](#), [4463](#), [4479](#),
 [4497](#), [4511](#), [4517](#), [4636](#), [4645](#), [4658](#),
 [4700](#), [4719](#), [4725](#), [4884](#), [5099](#), [5107](#),
 [5120](#), [5127](#), [5134](#), [5141](#), [5434](#), [5447](#),
 [5753](#), [5776](#), [5855](#), [5873](#), [5991](#), [5993](#),
 [6003](#), [6482](#), [6487](#), [6697](#), [7497](#), [7527](#),
 [7668](#), [10763](#), [10771](#), [10821](#), [10855](#),
 [10864](#), [10878](#), [10896](#), [10904](#), [10914](#),
 [10928](#), [10936](#), [10946](#), [10994](#), [11000](#),
 [11006](#), [11012](#), [11018](#), [11024](#), [11030](#)
 \prg_set_conditional:Nnn
 [37](#), [930](#), [930](#), [1869](#)
 \prg_set_conditional:Npnn
 [37](#), [899](#), [899](#), [1100](#),
 [1112](#), [1128](#), [1140](#), [1869](#), [4427](#), [6693](#)
 \prg_set_eq_conditional:NN [39](#)
 \prg_set_eq_conditional:NNn
 [965](#), [965](#), [1869](#)
 \prg_set_eq_conditional_aux:NNNn ...
 [966](#), [968](#), [1045](#), [1045](#)
 \prg_set_eq_conditional_aux:NNNw ...
 [1045](#), [1046](#), [1047](#), [1055](#)
 \prg_set_map_functions:Nn ... [2373](#), [2374](#)
 \prg_set_protected_conditional:Nnn .
 [38](#), [930](#), [949](#), [1869](#)
 \prg_set_protected_conditional:Npnn
 [38](#), [899](#), [915](#), [1869](#)
 \prg_stepwise_function:nnnN
 [45](#), [2175](#), [2175](#)
 \prg_stepwise_function_decr:nnnN ...
 [2175](#), [2179](#), [2191](#), [2196](#)
 \prg_stepwise_function_incr:nnnN ...
 [2175](#), [2178](#), [2182](#), [2187](#)
 \prg_stepwise_inline:nnnn
 .. [46](#), [2201](#), [2201](#), [2234](#), [11196](#), [11201](#)

\prg_stepwise_inline_decr:Nnnn	\prop_gpop:cnN	138, 5692	
.....	2201, 2209, 2223, 2228	\prop_gpop:coN	138, 5692	
\prg_stepwise_inline_incr:Nnnn	\prop_gpop:NnN	138, 5692, 5698, 5711, 5712, 5863, 5900	
.....	2201, 2208, 2214, 2219	\prop_gpop:NnNTF	141, 5857	
\prg_stepwise_variable:nnnn	46	\prop_gpop:NoN	138, 5692
\prg_stepwise_variable:nnnNn	2232, 2232	\prop_gput:ccx	5904	
\prg_variable_get_scope:N	47, 2263, 2269	\prop_gput:cnn	136, 5713	
\prg_variable_get_scope_aux:w	\prop_gput:cno	136, 5713	
.....	2263, 2271, 2274	\prop_gput:cnV	136, 5713	
\prg_variable_get_type:N	48, 2263, 2283	\prop_gput:cnx	136, 5713	
\prg_variable_get_type:w	2263	\prop_gput:con	136, 5713
\prg_variable_get_type_aux:w	2285, 2288, 2292	\prop_gput:coo	136, 5713
.....	2285, 2288, 2292	\prop_gput:cVn	136, 5713	
\prop_clear:c	135, 5636, 5637	\prop_gput:cVV	136, 5713
\prop_clear:N	135, 5636, 5636, 5641	\prop_gput:Nnn	136, 5713, 5714, 5731, 5733, 5904
\prop_clear_new:c	135, 5640, 6681	...	136, 5713, 5714, 5731, 5733, 5904	
\prop_clear_new:N	135, 5640, 5640, 5642	\prop_gput:Nno	136, 5713
\prop_del:cn	138, 5672	\prop_gput:NnV	136, 5713
\prop_del:cV	138, 5672	\prop_gput:Nnx	136, 5713
\prop_del:Nn	138, 5672, 5672, 5678, 5679	\prop_gput:Non	136, 5713
\prop_del:NV	138, 5672	\prop_gput:Noo	136, 5713
\prop_del_aux:Nnnnn	5672, 5673, 5675, 5676	\prop_gput:NVn	136, 5713, 6160, 6173	
\prop_display:c	5892, 5893	\prop_gput:NVV	136, 5713
\prop_display:N	5892, 5892	\prop_gput_if_new:cnn	137, 5735
\prop_gclear:c	135, 5636, 5639, 5645	\prop_gput_if_new:Nnn	137, 5735, 5737, 5749	
\prop_gclear:N	135, 5636, 5638, 5644	\prop_gset_eq:cc	135, 5646, 5653
\prop_gclear_new:c	135, 5640, 5645	\prop_gset_eq:cN	135, 5646, 5652
\prop_gclear_new:N	135, 5640, 5643	\prop_gset_eq:Nc	135, 5646, 5651
\prop_gdel:cn	138, 5672	\prop_gset_eq:NN	135, 5646, 5650
\prop_gdel:cV	138, 5672	\prop_if_empty:cTF	138, 5750
\prop_gdel:Nn	138, 5672, 5674, 5680, 5681	\prop_if_empty:N	5750
\prop_gdel:NV	138, 5672, 6276, 6288	\prop_if_empty:Nf	5761
\prop_get:cnN	137, 5682, 6821	\prop_if_empty:NT	5760
\prop_get:coN	137	\prop_if_empty:NTF	138, 5750, 5759, 5820, 6297, 6319
\prop_get:cVN	137, 5682	...	138, 5750, 5759, 5820, 6297, 6319	
\prop_get:Nn	141, 5888, 5888	\prop_if_empty_p:c	138, 5750
\prop_get:NnN	137, 5682, 5682, 5690, 5691, 5846	\prop_if_empty_p:N	138, 5750, 5758
...	137, 5682, 5682, 5690, 5691, 5846	\prop_if_eq:cc	5908	
\prop_get:NnNTF	141, 5846	\prop_if_eq:ccTF	5905
\prop_get:NoN	137, 5682	\prop_if_eq:cN	5906
\prop_get:NVN	137, 5682	\prop_if_eq:cNTF	5905
\prop_get_aux:nnn	5888, 5889, 5890	\prop_if_eq:Nc	5907
\prop_get_aux:Nnnn	5682, 5685, 5688	\prop_if_eq:NcTF	5905
\prop_get_aux_true:Nnnn	5846, 5849, 5852	\prop_if_eq:NN	5905	
\prop_get_gdel:NnN	5900, 5900	\prop_if_eq:NNTF	5905
\prop_gget:cnN	5894	\prop_if_eq_p:cc	5905
\prop_gget:cVN	5894	\prop_if_eq_p:cN	5905
\prop_gget:NnN	5894, 5894, 5898, 5899	\prop_if_eq_p:Nc	5905
\prop_gget:NVN	5894	\prop_if_eq_p:NN	5905
\prop_gget_aux:Nnnn	5894, 5895, 5896			

\prop_if_in:ccTF	5901	\prop_put:con	136, 5713
\prop_if_in:cnTF	139, 5762, 6811, 6814	\prop_put:coo	136, 5713
\prop_if_in:coTF	139, 5762	\prop_put:cVn	136, 5713
\prop_if_in:cVTF	139, 5762	\prop_put:cVV	136, 5713
\prop_if_in:Nn	5762	\prop_put:Nnn	136, 5713, 5713, 5727, 5729, 6130–6133, 6836
\prop_if_in:NnF	5785, 5786, 5902, 6181, 6189	\prop_put:Nno	136, 5713
\prop_if_in:NnT	5783, 5784, 5901	\prop_put:NnV	136, 5713
\prop_if_in:NnTF	139, 5762, 5787, 5788, 5903, 6805	\prop_put:Nnx	136, 5713
\prop_if_in:NoTF	139, 5762	\prop_put:Non	136, 5713
\prop_if_in:NVT	6225, 6262	\prop_put:Noo	136, 5713
\prop_if_in:NVTF	139, 5762	\prop_put:NVn	136, 5713
\prop_if_in_aux:nwn	5764, 5768, 5779	\prop_put:NVV	136, 5713
\prop_if_in_aux:w	5762	\prop_put_aux:NNnn	5713–5715
\prop_if_in_p:cn	139, 5762	\prop_put_aux:NNnnnnn	5713, 5717, 5719
\prop_if_in_p:co	139, 5762	\prop_put_if_new:cnn	137, 5735
\prop_if_in_p:cV	139, 5762	\prop_put_if_new:Nnn	137, 5735, 5735, 5748
\prop_if_in_p:Nn	139, 5762, 5781, 5782	\prop_put_if_new_aux:NNnn	5736, 5738, 5739
\prop_if_in_p:No	139, 5762	\prop_set_eq:cc	135, 5646, 5649
\prop_if_in_p:NV	139, 5762	\prop_set_eq:cN	135, 5646, 5648
\prop_map_break:	139, 5797, 5815, 5815, 5883	\prop_set_eq:Nc	135, 5646, 5647
\prop_map_break:n	140, 5816, 5816, 5891	\prop_set_eq:NN	135, 5646, 5646
\prop_map_function:cc	5789	\prop_show:c	140, 5818, 5893
\prop_map_function:cN	139, 5789	\prop_show:N	140, 5818, 5818, 5845, 5892
\prop_map_function:Nc	5789, 5810	\prop_show_aux:n	5818
\prop_map_function:NN	139, 5789, 5789, 5802, 5803, 5832, 6305, 6327	\prop_show_aux:nn	5832, 5837
\prop_map_function_aux:Nwn	5789, 5791, 5794, 5800	\prop_show_aux:w	5818, 5834, 5844, 6307, 6329
\prop_map_inline:cn	139, 5805	\prop_split:Nnn	142, 5666, 5666, 5717, 5895
\prop_map_inline:Nn	139, 5805, 5805, 5814	\prop_split:NnTF	142, 5654, 5654, 5668, 5673, 5675, 5684, 5694, 5700, 5741, 5848, 5859, 5865
\prop_map_tokens:Nn	141, 5875, 5875, 5889	\prop_split_aux:nnnn	5654, 5660, 5664
\prop_map_tokens_aux:nwn	5875, 5877, 5880, 5886	\prop_split_aux:NnTF	5654, 5655, 5656
\prop_new:c	135, 5634, 5635	\prop_split_aux:w	5654, 5658, 5661, 5665
\prop_new:N	135, 5634, 5634, 5641, 5644, 6127, 6128, 6677, 6678, 6900, 6922	\protect	235
\prop_pop:cnN	137, 5692	\protected	68, 82, 98, 104, 126, 133, 141, 143, 147, 152, 157, 213, 266, 273, 292, 325, 736
\prop_pop:coN	137, 5692	\protected@edef	6389, 6623
\prop_pop:NnN	137, 5692, 5692, 5709, 5710, 5857	\ProvidesClass	154
\prop_pop:NnNTF	141, 5857	\ProvidesExplClass	6, 146, 152
\prop_pop:NoN	137, 5692	\ProvidesExplFile	6, 146, 157
\prop_pop_aux:NNNnnn	5692, 5695, 5701, 5704	\ProvidesExplPackage	6, 146, 147, 333, 778, 1520, 1865, 2378, 2483, 3172, 3871, 4176, 4770, 5252, 5628, 5912, 6098, 6510, 7046, 7172, 7598, 7711, 11065
\prop_pop_aux_true:NNNnnn	5857, 5860, 5866, 5869	\ProvidesFile	159
\prop_put:cnn	136, 5713, 6832, 6834	\ProvidesPackage	47, 149
\prop_put:cno	136, 5713		
\prop_put:cnV	136, 5713		
\prop_put:cnx	136, 5713		

Q

- \Q 4619, 7109, 7144
- \q 1081, 2001, 2006
- \q_iow_stop [6372](#), 6372, 6396, 6407
- \q_mark [49](#), [2383](#),
2384, 3329, 3331, 5659, 5661, 5662
- \q_nil [49](#), 876, 879, 2298,
2302, [2383](#), 2383, 2440, 2461, 3693,
3715, 4363, 4371, 4388, 4400, 4440,
4451, 4452, 5330, 5331, 7072, 7102,
7131, 7138, 7145, 10986, 10993,
10999, 11005, 11011, 11017, 11023
- \q_no_value
. [49](#), 2030, [2383](#), 2385, 2448, 2470,
4363, 4371, 4400, 5392, 5431, 5437,
5444, 5450, 5670, 5686, 5696, 5702,
7072, 7080, 7085, 7101, 7107, 7326
- \q_prop [142](#), [5632](#), 5632,
5633, 5659, 5660, 5662, 5724, 5745,
5766, 5768, 5792, 5794, 5878, 5880
- \q_recursion_stop
... [51](#), 878, 881, 977, 1046, 1774,
2092, 2102, 2110, 2119, 2127, [2387](#),
2388, 2410, 2411, 2423, 2424, 2434,
4523, 4527, 4542, 4552, 4557, 4599,
5466, 5474, 5577, 5579, 5792, 5878
- \q_recursion_tail
. [51](#), [2387](#), 2387, 2391, 2397, 2411,
2424, 2434, 4523, 4527, 4542, 4552,
4557, 4599, 5466, 5474, 5577, 5579
- \q_stop [49](#), 877, 880,
1081, 1083, 1091, 1093, 1311, 1315,
1827, 1829, 2030, 2033, 2272, 2274,
2286, 2288, 2292, 2298, 2302, 2364,
[2383](#), 2386, 2411, 2424, 2434, 2686,
2688, 2726, 2728, 2733, 2735, 2743,
2746, 2754, 2757, 2765, 2768, 2776,
2779, 2785, 2788, 2793, 2795, 2801,
2804, 2821, 2824, 2827, 2849, 3042,
3049, 3058, 3067, 3320, 3321, 3329,
3333, 3341, 3349, 3357, 3365, 3373,
3381, 3761, 3798, 4356, 4371, 4382,
4388, 4391, 4396, 4400, 4627–4630,
4635, 4644, 4657, 4764, 4766, 4894,
4897, 4907, 4910, 5098, 5119, 5126,
5207, 5209, 5214, 5321, 5322, 5330,
5331, 5385, 5389, 5392, 5429, 5437,
5442, 5450, 5659, 5662, 5766, 6463,
7084, 7089, 7091, 7101, 7106, 7110,
7118, 7120, 7140, 7145, 7221, 7224,
7230, 7239, 7249, 7786, 7787, 7806,
7811, 7816, 7822, 7827, 7833, 7839,
7841, 7842, 7845, 7849, 7853, 7889,
7894, 8111, 8113, 8128, 8130, 8131,
8137, 8144, 8146, 8149, 8151, 8152,
8154, 8156, 8158, 8160, 8162, 8164,
8166, 8167, 8176, 8178, 8188, 8190,
8201, 8208, 8210–8212, 8214, 8216,
8218, 8220, 8222, 8224, 8226, 8228,
8237, 8243, 8245, 8255, 8257, 8264,
8266–8268, 8274, 8279, 8284, 8289,
8294, 8299, 8304, 8309, 8319, 8324,
8325, 8336, 8338, 8357, 8377, 8847,
8852, 8964, 9017, 9194, 9199, 9206,
9209, 10986, 10990, 10993, 10996,
10999, 11002, 11005, 11008, 11011,
11014, 11017, 11020, 11023, 11026
- \quark_if_nil:N 2438
- \quark_if_nil:n 2458
- \quark_if_nil:nF 2479
- \quark_if_nil:nT 2305, 2309, 2478
- \quark_if_nil:NTF [50](#), [2438](#), 3696, 3718, 7135
- \quark_if_nil:nTF
..... [50](#), 2313, 2322, 2331, 2340,
[2458](#), 2477, 5333, 10992, 10998,
11004, 11010, 11016, 11022, 11028
- \quark_if_nil:oF 7082
- \quark_if_nil:oTF [50](#), [2458](#)
- \quark_if_nil:VTF [50](#), [2458](#)
- \quark_if_nil_p:N [50](#), [2438](#)
- \quark_if_nil_p:n [50](#), [2458](#), 2476
- \quark_if_nil_p:o [50](#), [2458](#)
- \quark_if_nil_p:V [50](#), [2458](#)
- \quark_if_no_value:cF 7486
- \quark_if_no_value:cTF [50](#), [2438](#)
- \quark_if_no_value:N 2446
- \quark_if_no_value:n 2467
- \quark_if_no_value:N.TF [2438](#)
- \quark_if_no_value:NF 2456
- \quark_if_no_value:nF 4358, 5388
- \quark_if_no_value:NT 2455, 11140
- \quark_if_no_value:NTF .. [50](#), 2035, 2457
- \quark_if_no_value:nTF .. [50](#), [2458](#), 4384
- \quark_if_no_value_p:c [50](#), [2438](#)
- \quark_if_no_value_p:N [50](#), 2454
- \quark_if_no_value_p:n [50](#), [2458](#)
- \quark_if_no_value_p:N. [2438](#)
- \quark_if_recursion_tail_aux:w
..... [2404](#), 2410, 2423, 2433

\quark_if_recursion_tail_stop:N	9104, 9146, 9160, 9164, 9202, 9203,
. 51, 2389, 2389, 4563	9212, 9213, 9230, 9238, 9246, 9262,
\quark_if_recursion_tail_stop:n	9276, 9289, 9634, 9957, 9967, 10011,
51, 2404, 2404, 2436, 4531, 5479, 5586	10087–10090, 10111, 10312, 10338,
\quark_if_recursion_tail_stop:o 51, 2404	10375, 10383, 10391, 10473, 10475,
\quark_if_recursion_tail_stop_do:Nn	10477, 10512, 10520, 10724, 10726,
. 51, 2389, 2395	10728, 10730, 10732, 10746, 11142
\quark_if_recursion_tail_stop_do:n	\scantokens 684
. 51, 2404, 2417, 2437, 4602	\scriptfont 630
\quark_if_recursion_tail_stop_do:on	\scriptscriptfont 631
. 51, 2404	\scriptscriptstyle 476
\quark_new:N 49, 2382, 2382–2388, 5632, 6372	\scriptspace 516
	\scriptstyle 475
	\scrollmode 441
R	\seq_break: 122, 4932, 4964, 4969, 4969,
\R 1815, 2718	4971, 4978, 5092, 5100, 5107, 5120,
\radical 464	5127, 5134, 5141, 5178, 5198, 5206
\raise 604	\seq_break:n
\read 411	123, 4881, 4884, 4969, 4970, 4972, 5186
\readline 686	\seq_break_point:n . . 123, 4882, 4895,
\relax 3–6, 9, 13,	4908, 4921, 4949, 4969, 4970, 4973,
62, 70–80, 84–93, 96, 101, 131, 133,	4973, 4985, 5020, 5031, 5101, 5108,
141, 143, 229, 233, 249, 281–289, 446	5121, 5128, 5135, 5142, 5180, 5199
\repenalty 507	\seq_clear:c 111, 4783, 4784
\RequirePackage 57, 58	\seq_clear:N . . 111, 4783, 4783, 4828, 6778
\reverse_if:N . . . 25, 784, 789, 3961–3963	\seq_clear_new:c 111, 4787, 4788
right 505	\seq_clear_new:N 111, 4787, 4787
\righthyphenmin 561	\seq_concat:ccc 111, 4799
\rightskip 563	\seq_concat:NNN . . . 111, 4799, 4799, 4803
\romannumeral 638	\seq_display:c 5247, 5248
S	\seq_display:N 5247, 5247
\S 2718	\seq_gclear:c 111, 4783, 4786
\savecatcodetable 760	\seq_gclear:N 111, 4783, 4785
\savinghyphcodes 725	\seq_gclear_new:c 111, 4787, 4790
\savingdiscards 726	\seq_gclear_new:N 111, 4787, 4789
\scan_align_safe_stop: 47, 2248, 2255, 2255	\seq_gconcat:ccc 112, 4799
\scan_stop: 7, 308, 322, 811, 811,	\seq_gconcat:NNN . . 112, 4799, 4801, 4804
1049, 1073, 1102, 1120, 1130, 1148,	\seq_get:cN 118, 5055, 5056
1168, 1561, 1572, 1573, 1812–1820,	\seq_get:NN 118, 5055, 5055
2260, 2264, 2278, 2622, 2699, 3051,	\seq_get_left:cN . . 113, 4891, 5056, 5246
3060, 3069, 4056, 4067, 4072, 4097,	\seq_get_left:cNTF 119, 5095
4102, 4105, 4134, 4145, 4150, 4155,	\seq_get_left:NN
4169, 4170, 4285–4288, 4581, 4582,	113, 4891, 4891, 4899, 5055, 5095, 5245
4619, 6030, 6060, 6061, 6161, 6174,	\seq_get_left:NNF 5111
6634, 6635, 7794–7796, 7836, 7847,	\seq_get_left:NNT 5110
7855, 7860, 7896, 7897, 7913, 7921,	\seq_get_left:NNTF 119, 5095, 5112
7960, 7969, 7985, 7994, 8057, 8552,	\seq_get_left_aux:NnwN . 4891, 4894, 4897
8564, 8565, 8697, 8701, 8713, 8717,	\seq_get_left_aux:Nw 5098
8730, 8734, 8776, 8837, 8841, 8848–	\seq_get_right:cN 113, 4917
8850, 8858, 8869, 8919, 9021, 9096,	\seq_get_right:cNTF 119, 5095

<code>\seq_get_right:NN</code>	<code>\seq_gput_right:cx</code>
..... 113 , 4917 , 4917 , 4940 , 5103	113 , 4813
<code>\seq_get_right:NNF</code>	<code>\seq_gput_right:Nn</code>
..... 5114 113 , 4813 , 4815 , 4819 , 4820
<code>\seq_get_right:NNT</code>	<code>\seq_gput_right:No</code>
..... 5113	113 , 4813
<code>\seq_get_right:NNTF</code>	<code>\seq_gput_right:Nv</code>
..... 119 , 5095 , 5115	113 , 4813 , 7619
<code>\seq_get_right_aux:NN</code>	<code>\seq_gput_right:Nx</code>
..... 4917 , 4920 , 4923 , 5106	113 , 4813 , 7676
<code>\seq_get_right_loop:nn</code>	<code>\seq_gremove_all:cn</code>
..... 4917 , 4926 , 4935 , 4938 , 4955	115 , 4838
<code>\seq_gpop:cN</code>	<code>\seq_gremove_all:Nn</code>
..... 118 , 5055 , 5060	115 , 4838 , 4840 , 4863
<code>\seq_gpop:NN</code> .	<code>\seq_gremove_duplicates:c</code>
..... 118 , 5055 , 5059 , 7684 , 11139	115 , 4822
<code>\seq_gpop_left:cN</code>	<code>\seq_gremove_duplicates:N</code>
..... 114 , 4900 , 5060 115 , 4822 , 4824 , 4837
<code>\seq_gpop_left:cNTF</code>	<code>\seq_gset_eq:cc</code>
..... 120 , 5116	111 , 4791 , 4798
<code>\seq_gpop_left:NN</code>	<code>\seq_gset_eq:cN</code>
..... 114 , 4900 , 4902 , 4916 , 5059 , 5123	111 , 4791 , 4797
<code>\seq_gpop_left:NNF</code>	<code>\seq_gset_eq:Nc</code>
..... 5148	111 , 4791 , 4796
<code>\seq_gpop_left:NNT</code>	<code>\seq_gset_eq:NN</code> ...
..... 5147	111 , 4791 , 4795 , 4825
<code>\seq_gpop_left:NNTF</code>	<code>\seq_gset_from_clist:cc</code>
..... 120 , 5116 , 5149	122 , 5218
<code>\seq_gpop_right:cN</code>	<code>\seq_gset_from_clist:cN</code>
..... 114 , 4941	122 , 5218
<code>\seq_gpop_right:cNTF</code>	<code>\seq_gset_from_clist:cn</code>
..... 120 , 5116	122 , 5218
<code>\seq_gpop_right:NN</code>	<code>\seq_gset_from_clist:Nc</code>
..... 114 , 4941 , 4943 , 4968 , 5137	122 , 5218
<code>\seq_gpop_right:NNF</code>	<code>\seq_gset_from_clist:NN</code>
..... 5154 122 , 5218 , 5228 , 5242 , 5243
<code>\seq_gpop_right:NNT</code>	<code>\seq_gset_from_clist:Nn</code>
..... 5153 122 , 5218 , 5233 , 5244
<code>\seq_gpop_right:NNTF</code> ...	<code>\seq_if_empty:c</code>
..... 120 , 5116 , 5155 4866
<code>\seq_gpush:cn</code>	<code>\seq_if_empty:cTF</code>
..... 119 , 5035 , 5050	115 , 4864
<code>\seq_gpush:co</code>	<code>\seq_if_empty:N</code>
..... 119 , 5035 , 5053 4864
<code>\seq_gpush:cV</code>	<code>\seq_if_empty:NTF</code>
..... 119 , 5035 , 5051 115 , 4864 , 5064 , 5595 , 5606
<code>\seq_gpush:cv</code>	<code>\seq_if_empty_break_return_false:N</code> .
..... 119 , 5035 , 5052 5088 , 5088 ,
<code>\seq_gpush:cx</code> 5097 , 5105 , 5118 , 5125 , 5132 , 5139
..... 119 , 5035 , 5054	<code>\seq_if_empty_err_break:N</code>
<code>\seq_gpush:Nn</code> 122 , 4893 , 4906 , 4919 , 4947 , 4974 , 4974
..... 119 , 5035 , 5045	<code>\seq_if_empty_p:c</code>
<code>\seq_gpush:No</code>	115 , 4864
..... 119 , 5035 , 5048	<code>\seq_if_empty_p:N</code>
<code>\seq_gpush:Nv</code>	115 , 4864
..... 119 , 5035 , 5046 , 7681	<code>\seq_if_in:cnTF</code>
<code>\seq_gpush:Nv</code>	116 , 4868
..... 119 , 5035 , 5047	<code>\seq_if_in:coTF</code>
<code>\seq_gpush:Nx</code>	116 , 4868
..... 119 , 5035 , 5049 , 11128	<code>\seq_if_in:cVTF</code>
<code>\seq_gput_left:cn</code>	116 , 4868
..... 112 , 4813 , 5050	<code>\seq_if_in:cvTF</code>
<code>\seq_gput_left:co</code>	116 , 4868
..... 112 , 4813 , 5053	<code>\seq_if_in:cxTF</code>
<code>\seq_gput_left:cV</code>	116 , 4868
..... 112 , 4813 , 5051	<code>\seq_if_in:Nn</code>
<code>\seq_gput_left:cv</code>	4868
..... 112 , 4813 , 5052	<code>\seq_if_in:NnF</code> ...
<code>\seq_gput_left:cx</code>	4831 , 4887 , 4888 , 7689
..... 112 , 4813 , 5054	<code>\seq_if_in:NnT</code>
<code>\seq_gput_left:Nn</code>	4885 , 4886
..... 112 , 4813 , 4813 , 4817 , 4818 , 5045	<code>\seq_if_in:NnTF</code> 116 , 4868 , 4889 , 4890 , 6783
<code>\seq_gput_left:No</code>	<code>\seq_if_in:NoTF</code>
..... 112 , 4813 , 5048	116 , 4868
<code>\seq_gput_left:Nv</code>	<code>\seq_if_in:NvTF</code>
..... 112 , 4813 , 5046	116 , 4868
<code>\seq_gput_left:Nv</code>	<code>\seq_if_in:NvTF</code>
..... 112 , 4813 , 5047	116 , 4868
<code>\seq_gput_left:Nx</code>	<code>\seq_if_in:NxTF</code>
..... 112 , 4813 , 5049	116 , 4868
<code>\seq_gput_right:cn</code>	<code>\seq_if_in:NxTF</code>
..... 113 , 4813	116 , 4868
<code>\seq_gput_right:co</code>	<code>\seq_if_in_aux:</code>
..... 113 , 4813	4868 , 4877 , 4884
<code>\seq_gput_right:cV</code>	
..... 113 , 4813	
<code>\seq_gput_right:cv</code>	
..... 113 , 4813	

`\seq_item:cn` [121](#), [5166](#)
`\seq_item:n` [122](#), [4774](#), [4774](#), [4806](#), [4808](#),
[4814](#), [4816](#), [4856](#), [4873](#), [4897](#), [4910](#),
[4953](#), [4997](#), [5002](#), [5007](#), [5013](#), [5238](#)
`\seq_item:Nn` [121](#), [5166](#), [5166](#), [5189](#)
`\seq_item_aux:nnn` [5166](#), [5168](#), [5182](#), [5187](#)
`\seq_length:c` [120](#), [5156](#)
`\seq_length:N` . [120](#), [5156](#), [5156](#), [5165](#), [5173](#)
`\seq_length_aux:n` [5156](#), [5161](#), [5164](#)
`\seq_map_break:` [117](#), [4971](#), [4971](#), [4984](#), [7654](#)
`\seq_map_break:n` [117](#), [4971](#), [4972](#)
`\seq_map_function:cn` [116](#), [4981](#)
`\seq_map_function:NN`
[116](#), [4981](#), [4981](#), [4993](#), [5076](#), [5161](#), [5190](#)
`\seq_map_function_aux:NNn`
..... [4981](#), [4983](#), [4987](#), [4991](#)
`\seq_map_inline:cn` [116](#), [5016](#)
`\seq_map_inline:Nn`
[116](#), [4829](#), [5016](#), [5016](#), [5022](#), [7648](#), [7698](#)
`\seq_map_variable:ccn` [116](#), [5023](#)
`\seq_map_variable:cNn` [116](#), [5023](#)
`\seq_map_variable:Ncn` [116](#), [5023](#)
`\seq_map_variable:NNn`
..... [116](#), [5023](#), [5023](#), [5033](#), [5034](#)
`\seq_mapthread_function:ccN` . [121](#), [5192](#)
`\seq_mapthread_function:cNN` . [121](#), [5192](#)
`\seq_mapthread_function:NcN` . [121](#), [5192](#)
`\seq_mapthread_function:NNN`
..... [121](#), [5192](#), [5192](#), [5216](#), [5217](#)
`\seq_mapthread_function_aux:NN`
..... [5192](#), [5194](#), [5201](#)
`\seq_mapthread_function_aux:Nnnwnn` .
..... [5192](#), [5203](#), [5209](#), [5214](#)
`\seq_new:c` [4](#), [110](#), [4781](#), [4782](#)
`\seq_new:N` . [4](#), [110](#), [4781](#), [4781](#), [4821](#),
[6770](#), [7613](#), [7614](#), [7623](#), [7625](#), [11095](#)
`\seq_pop:cn` [118](#), [5055](#), [5058](#)
`\seq_pop:NN` [118](#), [5055](#), [5057](#)
`\seq_pop_item_def:`
..... [122](#), [4860](#), [4931](#), [4963](#),
[4994](#), [5010](#), [5020](#), [5031](#), [5601](#), [5612](#)
`\seq_pop_left:cn` [114](#), [4900](#), [5058](#)
`\seq_pop_left:cNTF` [120](#), [5116](#)
`\seq_pop_left:NN`
... [114](#), [4900](#), [4900](#), [4915](#), [5057](#), [5116](#)
`\seq_pop_left:NNF` [5145](#)
`\seq_pop_left:NNT` [5144](#)
`\seq_pop_left:NNTF` [120](#), [5116](#), [5146](#)
`\seq_pop_left_aux:NNN`
..... [4900](#), [4901](#), [4903](#), [4904](#)
`\seq_pop_left_aux:NnwNNN`
..... [4900](#), [4907](#), [4910](#), [5119](#), [5126](#)
`\seq_pop_right:cn` [114](#), [4941](#)
`\seq_pop_right:cNTF` [120](#), [5116](#)
`\seq_pop_right:NN`
..... [114](#), [4941](#), [4941](#), [4967](#), [5130](#)
`\seq_pop_right:NNF` [5151](#)
`\seq_pop_right:NNT` [5150](#)
`\seq_pop_right:NNTF` [120](#), [5116](#), [5152](#)
`\seq_pop_right_aux:NNN`
..... [4941](#), [4942](#), [4944](#), [4945](#)
`\seq_pop_right_aux_ii:NNN`
..... [4941](#), [4948](#), [4951](#), [5133](#), [5140](#)
`\seq_push:cn` [5035](#), [5040](#)
`\seq_push:co` [118](#), [5035](#), [5043](#)
`\seq_push:cV` [118](#), [5035](#), [5041](#)
`\seq_push:cv` [118](#), [5042](#)
`\seq_push:cx` [118](#), [5035](#), [5044](#)
`\seq_push:Nn` [118](#), [5035](#), [5035](#)
`\seq_push:No` [118](#), [5035](#), [5038](#)
`\seq_push:Nv` [118](#), [5035](#), [5036](#)
`\seq_push:Nv` [118](#), [5035](#), [5037](#)
`\seq_push:Nx` [118](#), [5035](#), [5039](#)
`\seq_push_item_def:n` [122](#), [4844](#), [4925](#),
[4953](#), [4994](#), [4994](#), [5018](#), [5598](#), [5609](#)
`\seq_push_item_def:x` [122](#), [4994](#), [4999](#), [5025](#)
`\seq_push_item_def_aux:`
..... [4994](#), [4996](#), [5001](#), [5004](#)
`\seq_put_left:cn` [112](#), [4805](#), [5040](#)
`\seq_put_left:co` [112](#), [4805](#), [5043](#)
`\seq_put_left:cV` [112](#), [4805](#), [5041](#)
`\seq_put_left:cv` [112](#), [4805](#), [5042](#)
`\seq_put_left:cx` [112](#), [4805](#), [5044](#)
`\seq_put_left:Nn`
... [112](#), [4805](#), [4805](#), [4809](#), [4810](#), [5035](#)
`\seq_put_left:No` [112](#), [4805](#), [5038](#)
`\seq_put_left:Nv` [112](#), [4805](#), [5036](#)
`\seq_put_left:Nv` [112](#), [4805](#), [5037](#)
`\seq_put_left:Nx` [112](#), [4805](#), [5039](#)
`\seq_put_right:cn` [113](#), [4805](#)
`\seq_put_right:co` [113](#), [4805](#)
`\seq_put_right:cV` [113](#), [4805](#)
`\seq_put_right:cv` [113](#), [4805](#)
`\seq_put_right:cx` [113](#), [4805](#)
`\seq_put_right:Nn`
... [113](#), [4805](#), [4807](#), [4811](#), [4812](#),
[4832](#), [5918](#), [6786](#), [7645](#), [7690](#), [7705](#)
`\seq_put_right:No` [113](#), [4805](#)
`\seq_put_right:Nv` [113](#), [4805](#)
`\seq_put_right:Nv` [113](#), [4805](#)

<code>\seq_put_right:Nx</code>	113, 4805	<code>\skip_add:cn</code>	89, 4066
<code>\seq_remove_all:cn</code>	115, 4838	<code>\skip_add:Nn</code> ...	89, 4066, 4066, 4068, 4069
<code>\seq_remove_all:Nn</code>	115, 4838, 4838, 4862, 7693	<code>\skip_eval:n</code>	90, 4079, 4096, 4096
<code>\seq_remove_all_aux:NNn</code>	4838, 4839, 4841, 4842	<code>\skip_gadd:cn</code>	89, 4066
<code>\seq_remove_duplicates:c</code>	114, 4822	<code>\skip_gadd:Nn</code>	89, 4066, 4068, 4070
<code>\seq_remove_duplicates:N</code>	114, 4822, 4822, 4836, 7696	<code>.skip_gset:c</code>	173
<code>\seq_remove_duplicates_aux:NN</code>	4822, 4823, 4825, 4826	<code>\skip_gset:cn</code>	89, 4055
<code>\seq_set_eq:cc</code>	111, 4791, 4794	<code>.skip_gset:N</code>	173
<code>\seq_set_eq:cN</code>	111, 4791, 4793	<code>\skip_gset:Nn</code>	89, 4055, 4057, 4059
<code>\seq_set_eq:Nc</code>	111, 4791, 4792	<code>\skip_gset_eq:cc</code>	89, 4060
<code>\seq_set_eq:NN</code>	111, 4791, 4791, 4823, 7643, 7659	<code>\skip_gset_eq:cN</code>	89, 4060
<code>\seq_set_from_clist:cc</code>	121, 5218	<code>\skip_gset_eq:Nc</code>	89, 4060
<code>\seq_set_from_clist:cN</code>	121, 5218	<code>\skip_gset_eq:NN</code> ...	89, 4060, 4063–4065
<code>\seq_set_from_clist:cn</code>	121, 5218	<code>\skip_gsub:cn</code>	90, 4066
<code>\seq_set_from_clist:Nc</code>	121, 5218	<code>\skip_gsub:Nn</code>	90, 4066, 4073, 4075
<code>\seq_set_from_clist:NN</code>	121, 5218, 5218, 5239, 5240	<code>\skip_gzero:c</code>	88, 4051
<code>\seq_set_from_clist:Nn</code>	121, 5218, 5223, 5241	<code>\skip_gzero:N</code>	88, 4051, 4052, 4054
<code>\seq_show:c</code>	119, 5062, 5248	<code>\skip_horizontal:c</code>	94, 4100
<code>\seq_show:N</code> ...	119, 5062, 5062, 5087, 5247	<code>\skip_horizontal:N</code>	94, 4100, 4100, 4102, 4106
<code>\seq_show_aux:n</code>	5062, 5076, 5081	<code>\skip_horizontal:n</code>	94, 4100, 4101
<code>\seq_show_aux:w</code>	5062, 5078, 5086	<code>\skip_if_eq:nn</code>	4076
<code>\seq_top:cN</code>	5245, 5246	<code>\skip_if_eq:nnTF</code>	90, 4076
<code>\seq_top:NN</code>	5245, 5245	<code>\skip_if_eq_p:nn</code>	90, 4076
<code>\seq_use:c</code>	121, 5190	<code>\skip_if_infinite_glue:n</code>	4086
<code>\seq_use:N</code>	121, 5190, 5190, 5191	<code>\skip_if_infinite_glue:nTF</code>	90, 4086, 4162
<code>\seq_use_error:</code>	4776	<code>\skip_if_infinite_glue_p:n</code> ...	90, 4086
<code>\seq_wrap_item:n</code>	5218, 5221, 5226, 5231, 5236, 5238	<code>\skip_new:c</code>	88, 4040
<code>\setbox</code>	612	<code>\skip_new:N</code>	88, 4040, 4044, 4050, 4112–4116, 8099
<code>\setlanguage</code>	370	<code>.skip_set:c</code>	173
<code>\sfcode</code>	667	<code>\skip_set:cn</code>	89, 4055
<code>\shipout</code>	577	<code>.skip_set:N</code>	173
<code>\show</code>	420	<code>\skip_set:Nn</code> ...	89, 4055, 4055, 4057, 4058
<code>\showbox</code>	422	<code>\skip_set_eq:cc</code>	89, 4060
<code>\showboxbreadth</code>	436	<code>\skip_set_eq:cN</code>	89, 4060
<code>\showboxdepth</code>	437	<code>\skip_set_eq:Nc</code>	89, 4060
<code>\showgroups</code>	697	<code>\skip_set_eq:NN</code> ...	89, 4060, 4060–4062
<code>\showifs</code>	698	<code>\skip_show:c</code>	91, 4108
<code>\showlists</code>	423	<code>\skip_show:N</code>	91, 4108, 4108, 4109
<code>\showthe</code>	421	<code>\skip_split_finite_else_action:nnNN</code>	95, 4160, 4160
<code>\showtokens</code>	685	<code>\skip_sub:cn</code>	89, 4066
<code>\skewchar</code>	634	<code>\skip_sub:Nn</code> ...	89, 4066, 4071, 4073, 4074
<code>\skip</code>	658	<code>\skip_use:c</code>	91, 4098
		<code>\skip_use:N</code> ...	91, 4097, 4098, 4098, 4099
		<code>\skip_vertical:c</code>	94, 4100
		<code>\skip_vertical:N</code>	94, 4100, 4103, 4105, 4107
		<code>\skip_vertical:n</code>	94, 4100, 4104
		<code>\skip_zero:c</code>	88, 4051

<code>\skip_zero:N</code>	88, 4051 , 4051–4053	<code>\tex_abovedisplayskip:D</code>	481
<code>\skipdef</code>	357	<code>\tex_abovewithdelims:D</code>	468
<code>\space</code>	49, 205	<code>\tex_accent:D</code>	518
<code>\spacefactor</code>	576	<code>\tex_adjdemerits:D</code>	555
<code>\spaceskip</code>	571	<code>\tex_advance:D</code>	
<code>\span</code>	384	362, 3289, 3291, 3301, 3303, 3917,	
<code>\special</code>	646	3922, 4067, 4072, 4145, 4150, 7887,	
<code>\splitbotmark</code>	455	7898, 7904, 7914, 7922, 7950, 7961,	
<code>\splitbotmarks</code>	681	7970, 7975, 7986, 7995, 8027, 8069,	
<code>\splitdiscards</code>	728	8481, 8517, 8543, 8551, 8557, 8581,	
<code>\splitfirstmark</code>	454	8594, 8619, 8704, 8705, 8719, 8720,	
<code>\splitfirstmarks</code>	680	8845, 8853, 8862, 8962, 8993–8995,	
<code>\splitmaxdepth</code>	624	8997, 8998, 9030, 9031, 9035, 9036,	
<code>\splittopskip</code>	625	9045, 9046, 9049, 9050, 9056, 9179,	
<code>\str_if_eq:nn</code>	1483	9180, 9192, 9204, 9214, 9219, 9247,	
<code>\str_if_eq:nnF</code>	1858, 1859	9252, 9263, 9281, 9290, 9338, 9339,	
<code>\str_if_eq:nnT</code> 1856, 1857, 4846, 5774, 5891		9342, 9343, 9395, 9407, 9637, 9638,	
<code>\str_if_eq:nnTF</code>	24, 1483 , 1860,	9672, 9677, 9680, 9681, 9685, 9686,	
1861, 2113, 3766, 3769, 4718, 7092		9691, 9692, 9696, 9697, 9700, 9701,	
<code>\str_if_eq:noTF</code>	24, 1854	9704, 9705, 10054, 10074, 10132,	
<code>\str_if_eq:nVTF</code>	24, 1854	10136, 10158, 10173, 10174, 10178,	
<code>\str_if_eq:onF</code>	4690	10179, 10184, 10185, 10189, 10190,	
<code>\str_if_eq:onT</code>	4688	10193, 10194, 10197, 10198, 10288,	
<code>\str_if_eq:onTF</code>	24, 1854 , 4686, 4722	10292, 10340, 10363, 10392, 10422,	
<code>\str_if_eq:VnTF</code>	24, 1854	10430, 10433, 10480, 10483, 10484,	
<code>\str_if_eq:VVTF</code>	24, 1854	10486, 10502, 10522, 10526, 10539,	
<code>\str_if_eq:xx</code>	1489	10540, 10543, 10544, 10549, 10550	
<code>\str_if_eq:xxTF</code>	24, 1483 , 2122	<code>\tex_afterassignment:D</code>	
<code>\str_if_eq_p:nn</code>	24, 1483 , 1854, 1855	372, 2881, 2967, 7837
<code>\str_if_eq_p:no</code>	24, 1854	<code>\tex_aftergroup:D</code>	373, 816
<code>\str_if_eq_p:nV</code>	24, 1854	<code>\tex_atop:D</code>	469
<code>\str_if_eq_p:on</code>	24, 1854 , 4692	<code>\tex_atopwithdelims:D</code>	470
<code>\str_if_eq_p:Vn</code>	24, 1854	<code>\tex_badness:D</code>	617
<code>\str_if_eq_p:VV</code>	24, 1854	<code>\tex_baselineskip:D</code>	545
<code>\str_if_eq_p:xx</code>	24, 1483	<code>\tex_batchmode:D</code>	438
<code>\str_length_loop:NNNNNNNN</code>		<code>\tex_begingroup:D</code>	376, 812
.	6457 , 6462, 6466, 6472	<code>\tex_belowdisplayshortskip:D</code>	482
<code>\str_length_skip_spaces:N</code> 6415 , 6457 , 6457		<code>\tex_belowdisplayskip:D</code>	483
<code>\str_length_skip_spaces:n</code> 6457 , 6458 , 6459		<code>\tex_binoppenalty:D</code>	506
<code>\strcmp</code>	230	<code>\tex_botmark:D</code>	453
<code>\string</code>	223, 238, 252, 639	<code>\tex_box:D</code>	661, 5955, 5975
T			
<code>\T</code>	1817, 2678, 2712, 2811	<code>\tex_boxmaxdepth:D</code>	623
<code>\t</code>	2715	<code>\tex_brokenpenalty:D</code>	580
<code>\tabskip</code>	385	<code>\tex_catcode:D</code>	
<code>\tempa</code>	106, 108, 109, 118, 124	665, 1074, 1572, 1573, 1814–
<code>\tempb</code>	107, 108	1820, 2265, 2279, 2488, 2490, 2492,	
<code>\tex_above:D</code>	467	3090, 4287, 4288, 4581, 4582, 4619	
<code>\tex_abovedisplayshortskip:D</code>	480	<code>\tex_char:D</code>	519
		<code>\tex_chardef:D</code>	354, 1061, 1062, 1163–
		1167, 1892, 1894, 2807, 6140, 6143	

<code>\tex_cleaders:D</code>	537	<code>\tex_everydisplay:D</code>	487, 764
<code>\tex_closein:D</code>	413, 6275	<code>\tex_everyhbox:D</code>	626
<code>\tex_closeout:D</code>	408, 6287	<code>\tex_everyjob:D</code>	
<code>\tex_clubpenalty:D</code>	548		655, 4667, 4669, 7604, 7606, 7616, 7618
<code>\tex_copy:D</code>	605, 5949, 5976	<code>\tex_everymath:D</code>	511, 765
<code>\tex_count:D</code>	656	<code>\tex_everypar:D</code>	574
<code>\tex_countdef:D</code> ...	355, 1160, 2739, 3247	<code>\tex_everyvbox:D</code>	627
<code>\tex_cr:D</code>	380	<code>\tex_exhyphenpenalty:D</code>	550
<code>\tex_crcr:D</code>	381	<code>\tex_expandafter:D</code>	374, 802
<code>\tex_csname:D</code>	443, 807	<code>\tex_fam:D</code>	366
<code>\tex_day:D</code>	651	<code>\tex_fi:D</code>	405, 788
<code>\tex_deadcycles:D</code>	585	<code>\tex_finalhyphendemerits:D</code>	554
<code>\tex_def:D</code>	350, 820–824	<code>\tex_firstmark:D</code>	452
<code>\tex_defaulthyphenchar:D</code>	635	<code>\tex_floatingpenalty:D</code>	599
<code>\tex_defaultskewchar:D</code>	636	<code>\tex_font:D</code>	365
<code>\tex_delcode:D</code>	666	<code>\tex_fontdimen:D</code>	632
<code>\tex_delimiter:D</code>	460	<code>\tex_fontname:D</code>	456
<code>\tex_delimiterfactor:D</code>	509	<code>\tex_futurelet:D</code>	361, 2875, 2877
<code>\tex_delimitershortfall:D</code>	508	<code>\tex_gdef:D</code>	352, 838
<code>\tex_dimen:D</code>	657	<code>\tex_global:D</code> 336, 341, 343, 367, 817, 1894	
<code>\tex_dimendef:D</code>	356, 2761, 3879	<code>\tex_globaldefs:D</code>	371
<code>\tex_discretionary:D</code>	520	<code>\tex_halign:D</code>	378
<code>\tex_displayindent:D</code>	485	<code>\tex_hangafter:D</code>	556
<code>\tex_displaylimits:D</code>	495	<code>\tex_hangindent:D</code>	557
<code>\tex_displaystyle:D</code>	473	<code>\tex_hbadness:D</code>	618
<code>\tex_displaywidowpenalty:D</code>	484	<code>\tex_hbox:D</code>	
<code>\tex_displaywidth:D</code>	486		613, 6030, 6031, 6036, 6042, 6050, 6051
<code>\tex_divide:D</code>		<code>\tex_hfil:D</code>	521
	363, 7915, 7962, 7987, 8556, 8824,	<code>\tex_hfill:D</code>	523
	9015, 9080, 9124, 9169, 9172, 9174,	<code>\tex_hfilneg:D</code>	522
	9176, 9240, 9282, 10385, 10435–10437	<code>\tex_hfuzz:D</code>	620
<code>\tex_doublehyphendemerits:D</code>	553	<code>\tex_hoffset:D</code>	595
<code>\tex_dp:D</code>	664, 5961	<code>\tex_holdinginserts:D</code>	598
<code>\tex_dump:D</code>	647	<code>\tex_hrule:D</code>	534
<code>\tex_edef:D</code>	351, 825	<code>\tex_hsize:D</code>	559
<code>\tex_else:D</code>	404, 787	<code>\tex_hskip:D</code>	524, 4100
<code>\tex_emergencystretch:D</code>	568	<code>\tex_hss:D</code>	525, 6053, 6055
<code>\tex_end:D</code>	442, 763, 1183, 6713, 6855	<code>\tex_ht:D</code>	663, 5960
<code>\tex_endcsname:D</code>	444, 808	<code>\tex_hyphen:D</code>	348, 766
<code>\tex_endgroup:D</code>	377, 761, 813	<code>\tex_hyphenation:D</code>	649
<code>\tex_endinput:D</code>	416, 6724	<code>\tex_hyphenchar:D</code>	633
<code>\tex_endlinechar:D</code>		<code>\tex_hyphenpenalty:D</code>	551
	307, 308, 322, 458, 4303, 4329, 4341	<code>\tex_if:D</code>	387, 790, 793
<code>\tex_eqno:D</code>	478	<code>\tex_ifcase:D</code>	388, 3181
<code>\tex_errhelp:D</code>	424, 6631	<code>\tex_ifcat:D</code>	389, 794
<code>\tex_errmessage:D</code>	418, 1175, 6651	<code>\tex_ifdim:D</code>	392, 3875
<code>\tex_errorcontextlines:D</code>	425, 6669	<code>\tex_ifeof:D</code>	393, 6102
<code>\tex_errorstopmode:D</code>	439	<code>\tex_iffalse:D</code>	398, 785
<code>\tex_escapechar:D</code>	457	<code>\tex_ifhbox:D</code>	394, 5987
<code>\tex_everycr:D</code>	386	<code>\tex_ifhmode:D</code>	400, 797

<code>\tex_ifinner:D</code>	403, 799	<code>\tex_mathchardef:D</code>	
<code>\tex_ifmmode:D</code>	401, 796	359, 1168, 3268, 5917, 11104
<code>\tex_ifnum:D</code>	390, 814, 3179	<code>\tex_mathchoice:D</code>	459
<code>\tex_ifodd:D</code> ...	391, 791, 792, 1201, 3180	<code>\tex_mathclose:D</code>	492
<code>\tex_iftrue:D</code>	399, 784	<code>\tex_mathcode:D</code>	670, 2558, 2560, 2562, 3091
<code>\tex_ifvbox:D</code>	395, 5988	<code>\tex_mathinner:D</code>	493
<code>\tex_ifvmode:D</code>	402, 798	<code>\tex_mathop:D</code>	494
<code>\tex_ifvoid:D</code>	396, 5989	<code>\tex_mathopen:D</code>	498
<code>\tex_ifx:D</code>	397, 795	<code>\tex_mathord:D</code>	499
<code>\tex_ignorespaces:D</code>	445	<code>\tex_mathpunct:D</code>	500
<code>\tex_immediate:D</code>		<code>\tex_mathrel:D</code>	501
...	407, 1170, 1172, 6174, 6287, 6352	<code>\tex_mathsurround:D</code>	512
<code>\tex_indent:D</code>	541	<code>\tex_maxdeadcycles:D</code>	582
<code>\tex_input:D</code>	415, 767, 7683	<code>\tex_maxdepth:D</code>	583
<code>\tex_inputlineno:D</code> .	417, 1190, 1799, 6571	<code>\tex_meaning:D</code>	642, 805, 809
<code>\tex_insert:D</code>	597	<code>\tex_medmuskip:D</code>	513
<code>\tex_insertpenalties:D</code>	600	<code>\tex_message:D</code>	419
<code>\tex_interlinepenalty:D</code>	579	<code>\tex_mkern:D</code>	466
<code>\tex_italic_correction:D</code>	768	<code>\tex_month:D</code>	652
<code>\tex_italiccor:D</code>	347	<code>\tex_moveleft:D</code>	602, 5980
<code>\tex_jobname:D</code>		<code>\tex_moveright:D</code>	603, 5982
...	654, 4664, 4672–4674, 4676, 7607	<code>\tex_mskip:D</code>	463
<code>\tex_kern:D</code>	532	<code>\tex_multiply:D</code>	364,
<code>\tex_language:D</code>	449	8754, 8961, 9175, 9191, 10159, 10748	
<code>\tex_lastbox:D</code>	606, 6008	<code>\tex_muskip:D</code>	660
<code>\tex_lastkern:D</code>	539	<code>\tex_muskipdef:D</code>	358, 4118
<code>\tex_lastpenalty:D</code>	645	<code>\tex_newlinechar:D</code>	414
<code>\tex_lastskip:D</code>	540	<code>\tex_noalign:D</code>	382
<code>\tex_lccode:D</code>		<code>\tex_noboundary:D</code>	517
668, 1073, 1812, 1813, 2264, 2278,		<code>\tex_noexpand:D</code>	375, 803
2564, 2566, 2568, 3092, 4285, 4286		<code>\tex_noindent:D</code>	543
<code>\tex_leaders:D</code>	536	<code>\tex_nolimits:D</code>	497
<code>\tex_left:D</code>	504	<code>\tex_nonscript:D</code>	477
<code>\tex_lefthyphenmin:D</code>	560	<code>\tex_nonstopmode:D</code>	440
<code>\tex_leftskip:D</code>	562	<code>\tex_nulldelimiterspace:D</code>	510
<code>\tex_leqno:D</code>	479	<code>\tex_nullfont:D</code>	628, 2834
<code>\tex_let:D</code> 337, 341, 343, 349, 763–773, 783		<code>\tex_number:D</code>	637, 3176
<code>\tex_limits:D</code>	496	<code>\tex_omit:D</code>	383
<code>\tex_linepenalty:D</code>	552	<code>\tex_openin:D</code>	409, 6161
<code>\tex_lineskip:D</code>	546	<code>\tex_openout:D</code>	410, 6174
<code>\tex_lineskiplimit:D</code>	547	<code>\tex_or:D</code>	406, 786
<code>\tex_long:D</code>	368, 818, 820	<code>\tex_outer:D</code>	369
<code>\tex_looseness:D</code>	564	<code>\tex_output:D</code>	584
<code>\tex_lower:D</code>	601, 5986	<code>\tex_outputpenalty:D</code>	594
<code>\tex_lowercase:D</code>	640, 1075, 1821, 4289, 4347	<code>\tex_over:D</code>	471
<code>\tex_mag:D</code>	448	<code>\tex_overfullrule:D</code>	622
<code>\tex_mark:D</code>	450	<code>\tex_overline:D</code>	502
<code>\tex_mathaccent:D</code>	461	<code>\tex_overwithdelims:D</code>	472
<code>\tex_mathbin:D</code>	491	<code>\tex_pagedepth:D</code>	586
<code>\tex_mathchar:D</code>	462	<code>\tex_pagefilllstretch:D</code>	590

<code>\tex_pagefillstretch:D</code>	589	<code>\tex_showlists:D</code>	423
<code>\tex_pagefilstretch:D</code>	588	<code>\tex_showthe:D</code>	
<code>\tex_pagegoal:D</code>	592		421, 1441, 2492, 2562, 2568, 2574,
<code>\tex_pageshrink:D</code>	591		2580, 3097, 3100, 3103, 3106, 3109
<code>\tex_pagestretch:D</code>	587	<code>\tex_skewchar:D</code>	634
<code>\tex_pagetotal:D</code>	593	<code>\tex_skip:D</code>	658
<code>\tex_par:D</code>	542	<code>\tex_skipdef:D</code>	357, 2750, 4041
<code>\tex_parfillskip:D</code>	573	<code>\tex_space:D</code>	346
<code>\tex_parindent:D</code>	566	<code>\tex_spacefactor:D</code>	576
<code>\tex_parshape:D</code>	558	<code>\tex_spaceskip:D</code>	571
<code>\tex_parskip:D</code>	565	<code>\tex_span:D</code>	384
<code>\tex_patterns:D</code>	648	<code>\tex_special:D</code>	646
<code>\tex_pausing:D</code>	435	<code>\tex_splitbotmark:D</code>	455
<code>\tex_penalty:D</code>	643	<code>\tex_splitfirstmark:D</code>	454
<code>\tex_postdisplaypenalty:D</code>	490	<code>\tex_splitmaxdepth:D</code>	624
<code>\tex_predisplaypenalty:D</code>	489	<code>\tex_splittopskip:D</code>	625
<code>\tex_predisplaysize:D</code>	488	<code>\tex_string:D</code>	639, 806
<code>\tex_pretolerance:D</code>	569	<code>\tex_tabskip:D</code>	385
<code>\tex_prevdepth:D</code>	616	<code>\tex_textfont:D</code>	629
<code>\tex_prevgraf:D</code>	575	<code>\tex_textstyle:D</code>	474
<code>\tex_radical:D</code>	464	<code>\tex_the:D</code>	
<code>\tex_raise:D</code>	604, 5984		. 308, 447, 1190, 1567, 1574, 1799,
<code>\tex_read:D</code>	411, 6490, 6492		2490, 2560, 2566, 2572, 2578, 3095,
<code>\tex_relax:D</code>	446, 811, 3178, 3877		3098, 3101, 3104, 3107, 3317, 4022,
<code>\tex_relpentalty:D</code>	507		4098, 4156, 4669, 7606, 7618, 11128
<code>\tex_right:D</code>	505	<code>\tex_thickmuskip:D</code>	515
<code>\tex_righthyphenmin:D</code>	561	<code>\tex_thinmuskip:D</code>	514
<code>\tex_rightskip:D</code>	563	<code>\tex_time:D</code>	650
<code>\tex_romannumeral:D</code>		<code>\tex_toks:D</code>	659
	. 638, 815, 1537, 1549, 1555, 1596,	<code>\tex_toksdef:D</code>	360, 2772
	1600, 1605, 1611, 1617, 1623, 1635,	<code>\tex_tolerance:D</code>	570
	1640, 1642, 1649, 1703, 1710, 1715,	<code>\tex_topmark:D</code>	451
	1718, 1720, 1723, 1728, 1734, 1745,	<code>\tex_topskip:D</code>	581
	1755, 1759, 1764, 4621, 5600, 5611	<code>\tex_tracingcommands:D</code>	426
<code>\tex_scriptfont:D</code>	630	<code>\tex_tracinglostchars:D</code>	427
<code>\tex_scriptscriptfont:D</code>	631	<code>\tex_tracingmacros:D</code>	428
<code>\tex_scriptscriptstyle:D</code>	476	<code>\tex_tracingonline:D</code>	429
<code>\tex_scriptspace:D</code>	516	<code>\tex_tracingoutput:D</code>	430
<code>\tex_scriptstyle:D</code>	475	<code>\tex_tracingpages:D</code>	431
<code>\tex_scrollmode:D</code>	441	<code>\tex_tracingparagraphs:D</code>	432
<code>\tex_setbox:D</code>		<code>\tex_tracingrestores:D</code>	433
	612, 5949, 5955, 6010, 6031, 6036,	<code>\tex_tracingstats:D</code>	434
	6042, 6065, 6070, 6076, 6082, 6094	<code>\tex_uccode:D</code> .	669, 2570, 2572, 2574, 3093
<code>\tex_setlanguage:D</code>	370	<code>\tex_uchyph:D</code>	567
<code>\tex_sfcode:D</code> .	667, 2576, 2578, 2580, 3094	<code>\tex_undefined:D</code>	336, 343
<code>\tex_shipout:D</code>	577	<code>\tex_underline:D</code>	503, 769
<code>\tex_show:D</code>	420, 810	<code>\tex_unhbox:D</code>	608, 6057
<code>\tex_showbox:D</code>	422, 6028	<code>\tex_unhcopy:D</code>	609, 6056
<code>\tex_showboxbreadth:D</code>	436	<code>\tex_unkern:D</code>	533
<code>\tex_showboxdepth:D</code>	437	<code>\tex_unpenalty:D</code>	644

- \tex_unskip:D 531
- \tex_unvbox:D 610, 6090
- \tex_unvcopy:D 611, 6089
- \tex_uppercase:D 641, 4348
- \tex_vadjust:D 544
- \tex_valign:D 379
- \tex_vbadness:D 619
- \tex_vbox:D 614, 6060, 6063–6065, 6076, 6082
- \tex_vcenter:D 465
- \tex_vfil:D 526
- \tex_vfill:D 528
- \tex_vfilneg:D 527
- \tex_vfuzz:D 621
- \tex_voffset:D 596
- \tex_vrule:D 535
- \tex_vsize:D 578
- \tex_vskip:D 529, 4103
- \tex_vsplit:D 607, 6094
- \tex_vss:D 530
- \tex_vtop:D 615, 6061, 6070
- \tex_wd:D 662, 5962
- \tex_widowpenalty:D 549
- \tex_write:D 412, 1170, 1172, 6347
- \tex_xdef:D 353, 839
- \tex_xleaders:D 538
- \tex_xspaceskip:D 572
- \tex_year:D 653
- \textasteriskcentered 3857, 3863
- \textbardbl 3862
- \textdagger 3858, 3864
- \textdaggerdbl 3859, 3865
- \textfont 629
- \textparagraph 3861
- \textsection 3860
- \textstyle 474
- \texttt 6596
- \TeXETstate 729
- \the 70–79, 447
- \thickmuskip 515
- \thinmuskip 514
- \time 650
- \tl_clear:c 96, 4198, 4784, 5261
- \tl_clear:N 96, 4198, 4198, 4202, 4205,
4305, 4381, 4783, 5260, 6379, 6380,
6448, 7066, 7070, 7639, 8542, 8820,
8838, 9073, 9097, 9117, 9147, 9161
- \tl_clear_new:c
.... 96, 4204, 4788, 5265, 7325, 7327
- \tl_clear_new:N
.... 96, 4204, 4204, 4208, 4787, 5264
- \tl_const:cn
.... 96, 4186, 9864–9903, 10210–10221
- \tl_const:cx 96, 4186, 10300
- \tl_const:Nn 96, 2382, 4186, 4186,
4196, 4292, 4679, 4680, 5633, 6107,
6515, 6516, 6537, 6542, 6544, 6546,
6548, 6550, 6555, 6556, 6563, 6581,
7022, 7024, 7176–7180, 7732–7736
- \tl_const:Nx 96, 4186,
4191, 4197, 4664, 4673, 4674, 4676
- \tl_elt_count:c 4757, 4761
- \tl_elt_count:N 4757, 4760
- \tl_elt_count:n 4757, 4757
- \tl_elt_count:o 4757, 4759
- \tl_elt_count:V 4757, 4758
- \tl_error_message: 4571, 4577, 4583
- \tl_gclear:c 96, 4198, 4786, 5263
- \tl_gclear:N
96, 4198, 4200, 4203, 4207, 4785, 5262
- \tl_gclear_new:c ... 96, 4204, 4790, 5267
- \tl_gclear_new:N
.... 96, 4204, 4206, 4209, 4789, 5266
- \tl_gput_left:cn 98, 4236
- \tl_gput_left:co 98, 4236
- \tl_gput_left:cV 98, 4236
- \tl_gput_left:cx 98, 4236
- \tl_gput_left:Nn
.... 98, 4236, 4244, 4256, 4814, 5301
- \tl_gput_left:No ... 98, 4236, 4248, 4258
- \tl_gput_left:NV ... 98, 4236, 4246, 4257
- \tl_gput_left:Nx ... 98, 4236, 4250, 4259
- \tl_gput_right:cn 98, 4260
- \tl_gput_right:co 98, 4260
- \tl_gput_right:cV 98, 4260
- \tl_gput_right:cx 98, 4260
- \tl_gput_right:Nn
.... 98, 4260, 4268, 4280, 4816, 5315
- \tl_gput_right:No .. 98, 4260, 4272, 4282
- \tl_gput_right:NV .. 98, 4260, 4270, 4281
- \tl_gput_right:Nx 98, 4260, 4274, 4283, 5738
- \tl_gremove_all:cn 100, 4411, 4756
- \tl_gremove_all:Nn
.... 100, 4411, 4413, 4416, 4755
- \tl_gremove_all_in:cn 4749, 4756
- \tl_gremove_all_in:Nn 4749, 4755
- \tl_gremove_in:cn 4749, 4752
- \tl_gremove_in:Nn 4749, 4751
- \tl_gremove_once:cn 99, 4405, 4752
- \tl_gremove_once:Nn
.... 99, 4405, 4407, 4410, 4751

<code>\tl_greplace_all:cnn</code>	99 , 4375 , 4748	<code>\tl_gtrim_spaces:c</code>	106 , 4609
<code>\tl_greplace_all:Nnn</code>		<code>\tl_gtrim_spaces:N</code>	106 , 4609 , 4616 , 4626
.	99 , 4375 , 4377 , 4404 , 4414 , 4747	<code>\tl_head:f</code>	107 , 4627
<code>\tl_greplace_all_in:cnn</code>	4741 , 4748	<code>\tl_head:n</code>	107 , 4627 , 4627 , 4631 , 4762
<code>\tl_greplace_all_in:Nnn</code>	4741 , 4747	<code>\tl_head:V</code>	107 , 4627
<code>\tl_greplace_in:cnn</code>	4741 , 4744	<code>\tl_head:v</code>	107 , 4627
<code>\tl_greplace_in:Nnn</code>	4741 , 4743	<code>\tl_head:w</code>	107 , 4627 , 4627 , 4629 , 4635 , 4644 , 4657 , 4763 , 7811 , 7822
<code>\tl_greplace_once:cnn</code>	99 , 4350 , 4744	<code>\tl_head_i:n</code>	4762 , 4762
<code>\tl_greplace_once:Nnn</code>		<code>\tl_head_i:w</code>	4762 , 4763
.	99 , 4350 , 4352 , 4374 , 4408 , 4743	<code>\tl_head_iii:f</code>	4762
<code>.tl_gset:c</code>	173	<code>\tl_head_iii:n</code>	4762 , 4764 , 4765
<code>\tl_gset:cf</code>	97 , 4218	<code>\tl_head_iii:w</code>	4762 , 4764 , 4766
<code>\tl_gset:cn</code>	97 , 4218	<code>\tl_if_blank:n</code>	4417
<code>\tl_gset:co</code>	97 , 4218	<code>\tl_if_blank:nF</code>	3757 , 4421 , 4425
<code>\tl_gset:cV</code>	97	<code>\tl_if_blank:nT</code>	4420 , 4424
<code>\tl_gset:cv</code>	97	<code>\tl_if_blank:nTF</code> 101 , 4417 , 4422 , 4426 , 4510	
<code>\tl_gset:cx</code>	97 , 4218 , 9481 , 9567 , 9848	<code>\tl_if_blank:oTF</code>	101 , 4417 , 4516 , 7079
<code>.tl_gset:N</code>	173	<code>\tl_if_blank:VTF</code>	101 , 4417
<code>\tl_gset:Nc</code>	4737 , 4737	<code>\tl_if_blank_p:n</code>	101 , 4417 , 4419 , 4423
<code>\tl_gset:Nf</code>	97 , 4218 , 5416	<code>\tl_if_blank_p:o</code>	101 , 4417
<code>\tl_gset:Nn</code>	97 , 4218 , 4224 , 4233 , 4235 , 4298 , 4324 , 4733 , 4903 , 5126 , 5301 , 5315 , 5328 , 5675 , 5701 , 5866 , 5897 , 7682 , 8017 , 8471 , 8506 , 8587 , 8612 , 8638 , 8741 , 8759 , 8872 , 9414 , 9511 , 9712 , 9905 , 10223 , 10557	<code>\tl_if_blank_p:V</code>	101 , 4417
<code>\tl_gset:No</code>	97 , 4218 , 4226	<code>\tl_if_blank_p_aux:NNw</code>	4417
<code>\tl_gset:Nv</code>	97 , 4218	<code>\tl_if_empty:c</code>	4866 , 5421
<code>\tl_gset:Nv</code>	97 , 4218	<code>\tl_if_empty:cTF</code>	101 , 4427
<code>\tl_gset:Nx</code> 97 , 4218 , 4228 , 4234 , 4617 , 4802 , 4841 , 4944 , 5140 , 5230 , 5235 , 5279 , 5610 , 5714 , 7607 , 8053 , 10207		<code>\tl_if_empty:N</code>	4427 , 4864 , 5420
<code>\tl_gset_eq:cc</code>		<code>\tl_if_empty:n</code>	4439
.	97 , 4210 , 4217 , 4798 , 5275 , 5653 , 8107	<code>\tl_if_empty:Nf</code>	4437 , 5393 , 7673
<code>\tl_gset_eq:cN</code>		<code>\tl_if_empty:nF</code>	2985 , 4449 , 5306 , 5471
.	97 , 4210 , 4215 , 4797 , 5274 , 5652 , 8105	<code>\tl_if_empty:NT</code>	4436
<code>\tl_gset_eq:Nc</code>		<code>\tl_if_empty:nT</code>	4448
.	97 , 4210 , 4216 , 4796 , 5273 , 5651 , 8106	<code>\tl_if_empty:NTF</code>	
<code>\tl_gset_eq:NN</code>	97 , 4201 , 4210 , 4214 , 4353 , 4378 , 4795 , 5272 , 5650 , 7611 , 8001 , 8013 , 8104	101 , 4427 , 4438 , 4517 , 7132 , 7666
<code>\tl_gset_rescan:cnn</code>	100 , 4295	<code>\tl_if_empty:nTF</code> 102 , 2729 , 2736 , 2747 , 2758 , 2769 , 2780 , 2789 , 2796 , 2805 , 3799 , 4439 , 4447 , 4511 , 6587 , 7250	
<code>\tl_gset_rescan:cno</code>	100 , 4295	<code>\tl_if_empty:o</code>	4458
<code>\tl_gset_rescan:cnx</code>	100 , 4321	<code>\tl_if_empty:oTF</code>	102 , 2826 , 4450 , 4496
<code>\tl_gset_rescan:Nnn</code>		<code>\tl_if_empty:VTF</code>	102 , 4439
.	100 , 4295 , 4297 , 4319 , 4320	<code>\tl_if_empty_p:c</code>	101 , 4427
<code>\tl_gset_rescan:Nno</code>	100 , 4295	<code>\tl_if_empty_p:N</code>	101 , 4427 , 4435
<code>\tl_gset_rescan:Nnx</code> 100 , 4321 , 4323 , 4336		<code>\tl_if_empty_p:n</code>	102 , 4439 , 4446
<code>.tl_gset_x:c</code>	174	<code>\tl_if_empty_p:o</code>	102 , 4450
<code>.tl_gset_x:N</code>	174	<code>\tl_if_empty_p:V</code>	102 , 4439
		<code>\tl_if_empty_return:o</code>	
		4418 , 4450 , 4450 , 4459 , 4521 , 4711
		<code>\tl_if_eq:cc</code>	5425 , 5908
		<code>\tl_if_eq:ccTF</code>	102 , 4460 , 7495
		<code>\tl_if_eq:cN</code>	5424 , 5906
		<code>\tl_if_eq:cNTF</code>	102 , 4460

<code>\tl_if_eq:Nc</code>	5423, 5907	<code>\tl_if_in:VnTF</code>	103, 4493
<code>\tl_if_eq:NcTF</code>	102, 4460	<code>\tl_if_single:cTF</code>	103
<code>\tl_if_eq:NN</code>	4460, 5422, 5905	<code>\tl_if_single:N</code>	4514
<code>\tl_if_eq:nn</code>	4472	<code>\tl_if_single:n</code>	4508
<code>\tl_if_eq:NNF</code>	4471	<code>\tl_if_single:NTF</code>	103
<code>\tl_if_eq:NNT</code>	4470, 4853	<code>\tl_if_single:nTF</code>	103, 4508
<code>\tl_if_eq:NNTF</code>	102, 2130, 4460, 4469, 6404, 6407, 6822	<code>\tl_if_single_aux:n</code>	4508, 4512, 4518, 4520
<code>\tl_if_eq:nnTF</code>	102, 4472	<code>\tl_if_single_item:nF</code>	4689
<code>\tl_if_eq_p:cc</code>	102, 4460	<code>\tl_if_single_item:nT</code>	4687
<code>\tl_if_eq_p:cN</code>	102, 4460	<code>\tl_if_single_item:nTF</code>	109, 4685, 4685
<code>\tl_if_eq_p:Nc</code>	102, 4460	<code>\tl_if_single_item_p:n</code>	109, 4685, 4691
<code>\tl_if_eq_p:NN</code>	102, 4460, 4468	<code>\tl_if_single_p:c</code>	103
<code>\tl_if_head_begin_group:n</code>	4693	<code>\tl_if_single_p:N</code>	103
<code>\tl_if_head_begin_group:nTF</code>	109, 4693, 4724	<code>\tl_if_single_p:n</code>	103, 4508
<code>\tl_if_head_begin_group_p:n</code>	109, 4693	<code>\tl_if_single_token:n</code>	4714
<code>\tl_if_head_eq_catcode:nN</code>	4654	<code>\tl_if_single_token:nTF</code>	110, 4714
<code>\tl_if_head_eq_catcode:nNTF</code>	107, 4633	<code>\tl_if_single_token_p:n</code>	110, 4714
<code>\tl_if_head_eq_catcode_p:nN</code>	107, 4633	<code>\tl_length:c</code>	106, 4585, 4761
<code>\tl_if_head_eq_charcode:fNTF</code>	107, 3662, 3675, 4633	<code>\tl_length:N</code>	106, 4585, 4590, 4597, 4760
<code>\tl_if_head_eq_charcode:nN</code>	4641	<code>\tl_length:n</code>	105, 4585, 4585, 4596, 4757
<code>\tl_if_head_eq_charcode:nNF</code>	4653	<code>\tl_length:o</code>	105, 4585, 4759
<code>\tl_if_head_eq_charcode:nNT</code>	4652	<code>\tl_length:V</code>	105, 4585, 4758
<code>\tl_if_head_eq_charcode:nNTF</code>	107, 4633, 4651	<code>\tl_length_aux:n</code>	4585, 4588, 4593, 4595
<code>\tl_if_head_eq_charcode_p:fN</code>	107, 4633	<code>\tl_map_break:</code>	104, 4567, 4567, 6184, 6192
<code>\tl_if_head_eq_charcode_p:nN</code>	107, 4633, 4650	<code>\tl_map_function:cN</code>	103, 4522
<code>\tl_if_head_eq_meaning:nN</code>	4633	<code>\tl_map_function:NN</code>	103, 4522, 4524, 4534, 4593, 6155, 6168
<code>\tl_if_head_eq_meaning:nNTF</code>	108, 4633, 7107	<code>\tl_map_function:nN</code>	104, 4522, 4522, 4588
<code>\tl_if_head_eq_meaning_p:nN</code>	108, 4633	<code>\tl_map_function_aux:NN</code>	4522
<code>\tl_if_head_eq_space:n</code>	4704	<code>\tl_map_function_aux:Nn</code>	4523, 4526, 4529, 4532, 4540, 4550
<code>\tl_if_head_eq_space:nTF</code>	109, 4704, 4716	<code>\tl_map_inline:cn</code>	104, 4535
<code>\tl_if_head_eq_space_aux:w</code>	4704, 4707, 4709	<code>\tl_map_inline:Nn</code>	104, 4535, 4545, 4555
<code>\tl_if_head_eq_space_p:n</code>	109, 4704	<code>\tl_map_inline:nn</code>	104, 2718, 4535, 4535
<code>\tl_if_in:cnTF</code>	102, 4487	<code>\tl_map_inline_aux:n</code>	4535
<code>\tl_if_in:nn</code>	4493	<code>\tl_map_variable:cNn</code>	104, 4556
<code>\tl_if_in:NnF</code>	4488, 4491	<code>\tl_map_variable:NNn</code>	104, 4556, 4558, 4566
<code>\tl_if_in:nnF</code>	4488, 4500, 4503, 4506	<code>\tl_map_variable:nNn</code>	104, 4556, 4556, 4559
<code>\tl_if_in:NnT</code>	4487, 4490	<code>\tl_map_variable_aux:NnN</code>	4556
<code>\tl_if_in:nnT</code>	4487, 4499, 4502, 4505	<code>\tl_map_variable_aux:Nnn</code>	4557, 4560, 4564
<code>\tl_if_in:NnTF</code>	102, 4487, 4489, 4492	<code>\tl_new:c</code>	96, 4180, 4782, 5259, 7307, 9480, 9566, 9847
<code>\tl_if_in:nnTF</code>	103, 4489, 4493, 4501, 4504, 4507, 7220, 7227	<code>\tl_new:cn</code>	4730
<code>\tl_if_in:noTF</code>	4493	<code>\tl_new:N</code>	96, 4180, 4180, 4185, 4205, 4207, 4294, 4349, 4485, 4486, 4681–4684, 4732, 4779–4781, 5061, 5256–5258, 5817, 6368–6371, 6514, 6579, 6771–6773, 7051–7054, 7182–7184, 7186–7188, 7602, 7622, 7737, 7767, 7774, 7777, 7780, 8000, 10206, 11156
<code>\tl_if_in:onTF</code>	103, 4493, 4672		

`\tl_new:Nn` [4730](#), [4730](#), [4735](#), [4736](#)
`\tl_new:Nx` [4730](#)
`\tl_put_left:cn` [98](#), [4236](#)
`\tl_put_left:co` [98](#), [4236](#)
`\tl_put_left:cV` [98](#), [4236](#)
`\tl_put_left:cx` [98](#), [4236](#)
`\tl_put_left:Nn`
 [98](#), [4236](#), [4236](#), [4252](#), [4806](#), [5299](#)
`\tl_put_left:No` [98](#), [4236](#), [4240](#), [4254](#)
`\tl_put_left:Nv` [98](#), [4236](#), [4238](#), [4253](#)
`\tl_put_left:Nx` [98](#), [4236](#), [4242](#), [4255](#)
`\tl_put_right:cn` [98](#), [4260](#)
`\tl_put_right:co` [98](#), [4260](#)
`\tl_put_right:cV` [98](#), [4260](#)
`\tl_put_right:cx` [98](#), [4260](#)
`\tl_put_right:Nn` [98](#), [4260](#),
 [4260](#), [4276](#), [4808](#), [5313](#), [5387](#), [7133](#)
`\tl_put_right:No` [98](#), [4260](#),
 [4264](#), [4278](#), [4364](#), [4389](#), [4394](#), [6649](#)
`\tl_put_right:Nv` ... [98](#), [4260](#), [4262](#), [4277](#)
`\tl_put_right:Nx` [98](#),
 [4260](#), [4266](#), [4279](#), [5736](#), [6431](#), [6438](#),
 [6445](#), [6454](#), [7094](#), [7126](#), [7137](#), [7149](#)
`\tl_remove_all:cn` [100](#), [4411](#), [4754](#)
`\tl_remove_all:Nn`
 [100](#), [4411](#), [4411](#), [4415](#), [4753](#)
`\tl_remove_all_in:cn` [4749](#), [4754](#)
`\tl_remove_all_in:Nn` [4749](#), [4753](#)
`\tl_remove_in:cn` [4749](#), [4750](#)
`\tl_remove_in:Nn` [4749](#), [4749](#)
`\tl_remove_once:cn` [99](#), [4405](#), [4750](#)
`\tl_remove_once:Nn`
 [99](#), [4405](#), [4405](#), [4409](#), [4749](#)
`\tl_replace_all:cnn` [99](#), [4375](#), [4746](#)
`\tl_replace_all:Nnn`
 [99](#), [4375](#), [4375](#), [4403](#), [4412](#), [4745](#)
`\tl_replace_all_aux:NNnn`
 [4375](#), [4376](#), [4378](#), [4379](#)
`\tl_replace_all_in:cnn` [4741](#), [4746](#)
`\tl_replace_all_in:Nnn`
 [4741](#), [4745](#), [7068](#), [7069](#)
`\tl_replace_in:cnn` [4741](#), [4742](#)
`\tl_replace_in:Nnn` [4741](#), [4741](#)
`\tl_replace_once:cnn` [99](#), [4350](#), [4742](#)
`\tl_replace_once:Nnn`
 [99](#), [4350](#), [4350](#), [4373](#), [4406](#), [4741](#)
`\tl_replace_once_aux:NNnn`
 [4350](#), [4351](#), [4353](#), [4354](#)
`\tl_rescan:nn` [100](#), [4337](#), [4337](#)
`\tl_rescan_aux:w`
 ... [4295](#), [4306](#), [4310](#), [4312](#), [4316](#), [4343](#)
`\tl_reverse:c` [106](#), [4606](#)
`\tl_reverse:N` [106](#), [4606](#), [4606](#), [4608](#)
`\tl_reverse:n` [106](#), [4598](#), [4598](#), [4605](#)
`\tl_reverse:o` [106](#), [4598](#), [4607](#)
`\tl_reverse:V` [106](#), [4598](#)
`\tl_reverse_aux:nN` [4598](#), [4599](#), [4600](#), [4603](#)
`.tl_set:c` [173](#)
`\tl_set:cf` [97](#), [4218](#)
`\tl_set:cn` [97](#), [4218](#), [7326](#), [7330](#)
`\tl_set:co` [97](#), [4218](#)
`\tl_set:cx` [97](#), [4218](#), [7310](#)
`.tl_set:N` [173](#)
`\tl_set:Nc` [4737](#), [4738](#), [4739](#)
`\tl_set:Nf` [97](#), [4218](#), [4607](#), [5414](#)
`\tl_set:Nn` [97](#), [2236](#),
 [2887](#), [2908](#), [4218](#), [4218](#), [4230](#), [4232](#),
 [4296](#), [4314](#), [4322](#), [4475](#), [4476](#), [4562](#),
 [4849](#), [4858](#), [4872](#), [4875](#), [4898](#), [4901](#),
 [4913](#), [4928](#), [4959](#), [5027](#), [5119](#), [5299](#),
 [5313](#), [5323](#), [5326](#), [5336](#), [5510](#), [5518](#),
 [5673](#), [5686](#), [5689](#), [5695](#), [5696](#), [5702](#),
 [5706](#), [5854](#), [5860](#), [5871](#), [6403](#), [6616](#),
 [6803](#), [6804](#), [7067](#), [7112](#), [7147](#), [7195](#),
 [7232](#), [7294](#), [7438](#), [7440](#), [7483](#), [8016](#),
 [8468](#), [8503](#), [8586](#), [8611](#), [8637](#), [8740](#),
 [8758](#), [8871](#), [9413](#), [9510](#), [9711](#), [9904](#),
 [10222](#), [10556](#), [10614](#), [10626](#), [11141](#)
`\tl_set:No` [97](#), [4218](#), [4220](#), [4360](#), [4740](#), [5395](#)
`\tl_set:Nv` [97](#), [4218](#)
`\tl_set:Nv` [97](#), [4218](#)
`\tl_set:Nx` [97](#),
 [4218](#), [4222](#), [4231](#), [4331](#), [4615](#), [4800](#),
 [4839](#), [4942](#), [5075](#), [5133](#), [5220](#), [5225](#),
 [5277](#), [5538](#), [5599](#), [5713](#), [5831](#), [6304](#),
 [6326](#), [6386](#), [6620](#), [7117](#), [7131](#), [7193](#),
 [7219](#), [7226](#), [7229](#), [7456](#), [7457](#), [7634](#),
 [7653](#), [7800](#), [7813](#), [7824](#), [7916](#), [7963](#),
 [7988](#), [8051](#), [8571](#), [8574](#), [8620](#), [8868](#),
 [9232](#), [9241](#), [9264](#), [9283](#), [9426](#), [9523](#),
 [9724](#), [9918](#), [10001](#), [10034](#), [10261](#),
 [10377](#), [10386](#), [10514](#), [10958](#), [10983](#)
`\tl_set_eq:cc` [96](#), [4210](#),
 [4213](#), [4794](#), [5271](#), [5649](#), [7344](#), [8103](#)
`\tl_set_eq:cN`
 [96](#), [4210](#), [4211](#), [4793](#), [5270](#), [5648](#), [8101](#)
`\tl_set_eq:Nc` [96](#), [4210](#),
 [4212](#), [4792](#), [5269](#), [5647](#), [7488](#), [8102](#)

- \tl_set_eq:NN 96,
4199, 4210, 4210, 4351, 4376, 4791,
5268, 5646, 6428, 6441, 8011, 8100
- \tl_set_rescan:cnm 100, 4295
- \tl_set_rescan:cno 100, 4295
- \tl_set_rescan:cnx 100, 4321
- \tl_set_rescan:Nnn
..... 100, 4295, 4295, 4317, 4318
- \tl_set_rescan:Nno 100, 4295, 7801
- \tl_set_rescan:Nnx . 100, 4321, 4321, 4335
- \tl_set_rescan_aux:NNnn
..... 4295, 4296, 4298, 4299
- \tl_set_rescan_aux:NNnx
..... 4321, 4322, 4324, 4325
- .tl_set_x:c 173
- .tl_set_x:N 173
- \tl_show:c 108, 4660, 8109
- \tl_show:N ... 108, 4660, 4660, 4661, 8108
- \tl_show:n 108, 4662, 4662,
5067, 5530, 5823, 5833, 6300, 6322
- \tl_tail:f 107, 4627
- \tl_tail:n 107, 4627, 4628, 4632
- \tl_tail:V 107, 4627
- \tl_tail:v 107, 4627
- \tl_tail:w 107, 4627, 4628, 4630, 7816, 7827
- \tl_tmp:w 4356, 4363, 4366, 4370, 4382,
4388, 4391, 4396, 4399, 4495, 4496
- \tl_to_lowercase:n
.... 101, 2266, 2280, 2680, 2720,
2813, 3080, 4347, 4347, 6637, 7060
- \tl_to_str:c 105, 4569
- \tl_to_str:N .. 105, 4569, 4569, 4570, 6395
- \tl_to_str:n
105, 3042, 3946, 4440, 4452, 4568,
4568, 5655, 5724, 5745, 5765, 6462,
7193, 7226, 7456, 7521, 7531, 8093
- \tl_to_uppercase:n 101, 4347, 4348
- \tl_trim_spaces:c 106, 4609
- \tl_trim_spaces:N . 106, 4609, 4614, 4625
- \tl_trim_spaces:n
... 106, 4609, 4609, 4615, 4617, 5412
- \tl_trim_spaces_aux_i:w 4609, 4621, 4622
- \tl_trim_spaces_aux_ii:w 4609, 4622, 4623
- \tl_trim_spaces_exp:n .. 4609, 4612, 4620
- \tl_use:c 105, 4571, 4572, 5357
- \tl_use:N 105, 4571, 4571, 5356
- \token_get_arg_spec:N ... 67, 3040, 3053
- \token_get_prefix_arg_replacement_aux:wN
..... 3040, 3041, 3048, 3057, 3066
- \token_get_prefix_spec:N . 68, 3040, 3044
- \token_get_replacement_spec:N 3040, 3062
- \token_get_replacement_text:N 67
- \token_if_active:N 2654
- \token_if_active:NF 3167
- \token_if_active:NT 3166
- \token_if_active:NTF 60, 2654, 3168
- \token_if_active_char:NF 3167
- \token_if_active_char:NT 3166
- \token_if_active_char:NTF ... 3153, 3168
- \token_if_active_char_p:N ... 3153, 3165
- \token_if_active_p:N 60, 2654, 3165
- \token_if_alignment:N 2616
- \token_if_alignment:NF 3155
- \token_if_alignment:NT 3154
- \token_if_alignment:NTF . 59, 2616, 3156
- \token_if_alignment_p:N . 59, 2616, 3153
- \token_if_alignment_tab:NF 3155
- \token_if_alignment_tab:NT 3154
- \token_if_alignment_tab:NTF . 3153, 3156
- \token_if_alignment_tab_p:N . 3153, 3153
- \token_if_chardef:N 2723
- \token_if_chardef:NTF 62, 2711
- \token_if_chardef_aux:w 2725, 2728
- \token_if_chardef_p:N 62, 2711
- \token_if_chardef_p_aux:w 2711
- \token_if_cs:N 2697
- \token_if_cs:NTF 61, 2697
- \token_if_cs_p:N 61, 2697
- \token_if_dim_register:N 2759
- \token_if_dim_register:NTF 63, 2711
- \token_if_dim_register_aux:w 2764, 2768
- \token_if_dim_register_p:N 63, 2711
- \token_if_dim_register_p_aux:w .. 2711
- \token_if_eq_catcode:NN 2664
- \token_if_eq_catcode:NNTF 61, 2664
- \token_if_eq_catcode_p:NN
..... 61, 2664, 2944, 2945
- \token_if_eq_charcode:NN 2669
- \token_if_eq_charcode:NNTF 61, 2669
- \token_if_eq_charcode_p:NN 61, 2669
- \token_if_eq_meaning:NN 2659
- \token_if_eq_meaning:NNT 2275
- \token_if_eq_meaning:NNTF
..... 61, 2290, 2659, 2965
- \token_if_eq_meaning_p:NN 61, 2659, 2946
- \token_if_expandable:N 2702
- \token_if_expandable:NTF 62, 2702
- \token_if_expandable_p:N 62, 2702
- \token_if_group_begin:N 2601
- \token_if_group_begin:NTF 58, 2601

- \token_if_group_begin_p:N 58, 2601
- \token_if_group_end:N 2606
- \token_if_group_end:NTF 59, 2606
- \token_if_group_end_p:N 59, 2606
- \token_if_int_register:N 2737
- \token_if_int_register:NTF 63, 2711
- \token_if_int_register_aux:w 2742, 2746
- \token_if_int_register_p:N 63, 2711
- \token_if_int_register_p_aux:w . . 2711
- \token_if_letter:N 2644
- \token_if_letter:NTF 60, 2644
- \token_if_letter_p:N 60, 2644
- \token_if_long_macro:N 2790
- \token_if_long_macro:NTF 62, 2711
- \token_if_long_macro_aux:w . . 2792, 2795
- \token_if_long_macro_p:N 62, 2711
- \token_if_long_macro_p_aux:w . . . 2711
- \token_if_macro:N 2683
- \token_if_macro:NTF 61, 2674, 2817, 3046, 3055, 3064
- \token_if_macro_p:N 61, 2674
- \token_if_macro_p_aux:w 2674, 2685, 2688
- \token_if_math_shift:N 3159
- \token_if_math_shift:NT 3158
- \token_if_math_shift:NTF 3153, 3160
- \token_if_math_shift_p:N 3153, 3157
- \token_if_math_subscript:N 2634
- \token_if_math_subscript:NTF . . 60, 2634
- \token_if_math_subscript_p:N . . 60, 2634
- \token_if_math_superscript:N 2629
- \token_if_math_superscript:NTF 59, 2629
- \token_if_math_superscript_p:N 59, 2629
- \token_if_math_toggle:N 2611
- \token_if_math_toggle:N 3159
- \token_if_math_toggle:NT 3158
- \token_if_math_toggle:NTF 59, 2611, 3160
- \token_if_math_toggle_p:N 59, 2611, 3157
- \token_if_mathchardef:N 2730
- \token_if_mathchardef:NTF 63, 2711
- \token_if_mathchardef_aux:w . 2732, 2735
- \token_if_mathchardef_p:N 63, 2711
- \token_if_mathchardef_p_aux:w . . 2711
- \token_if_other:N 2649
- \token_if_other:N 3163
- \token_if_other:NT 3162
- \token_if_other:NTF 60, 2649, 3164
- \token_if_other_char:N 3163
- \token_if_other_char:NT 3162
- \token_if_other_char:NTF 3153, 3164
- \token_if_other_char_p:N 3153, 3161
- \token_if_other_p:N 60, 2649, 3161
- \token_if_parameter:N 2623
- \token_if_parameter:NTF 59, 2621
- \token_if_parameter_p:N 59, 2621
- \token_if_primitive:N 2815
- \token_if_primitive:NTF 64, 2807
- \token_if_primitive_aux:NNw 2807, 2820, 2824
- \token_if_primitive_aux_loop:N 2807, 2827, 2840, 2846
- \token_if_primitive_aux_nullfont:N 2807, 2828, 2832
- \token_if_primitive_aux_space:w 2807, 2826, 2831
- \token_if_primitive_aux_undefined:N 2807, 2852, 2858
- \token_if_primitive_auxii:Nw 2807, 2843, 2849
- \token_if_primitive_p:N 64, 2807
- \token_if_protected_long_macro:N . . 2797
- \token_if_protected_long_macro:NTF 62, 2711
- \token_if_protected_long_macro_aux:w 2800, 2803
- \token_if_protected_long_macro_p:N 62, 2711
- \token_if_protected_long_macro_p_aux:w 2711
- \token_if_protected_macro:N 2781
- \token_if_protected_macro:NTF . 62, 2711
- \token_if_protected_macro_aux:w 2784, 2787
- \token_if_protected_macro_p:N . 62, 2711
- \token_if_protected_macro_p_aux:w 2711
- \token_if_skip_register:N 2748
- \token_if_skip_register:NTF . . . 63, 2711
- \token_if_skip_register_aux:w 2753, 2757
- \token_if_skip_register_p:N . . . 63, 2711
- \token_if_skip_register_p_aux:w . . 2711
- \token_if_space:N 2639
- \token_if_space:NTF 60, 2639
- \token_if_space_p:N 60, 2639
- \token_if_toks_register:N 2770
- \token_if_toks_register:NTF . . . 63, 2711
- \token_if_toks_register_aux:w 2775, 2779
- \token_if_toks_register_p:N . . . 63, 2711
- \token_if_toks_register_p_aux:w . . 2711
- \token_new:Nn 57, 2581, 2581, 2586, 2588–2590, 2592–2595

- \token_to_meaning:N 58, 805, 805,
1197, 1207, 1220, 1827, 2686, 2726,
2733, 2743, 2754, 2765, 2776, 2785,
2793, 2801, 2821, 3049, 3058, 3067
- \token_to_str:c 58, 821, 822
- \token_to_str:N
. 5, 58, 805, 806, 822, 1065, 1068,
1197, 1207, 1209, 1220, 1354, 1444,
1797, 2286, 4698, 4700, 4977, 5066,
5072, 5529, 5535, 5822, 5828, 11111
- \toks 659
- \toksdef 360
- \tolerance 570
- \topmark 451
- \topmarks 677
- \topskip 581
- \tracingassigns 687
- \tracingcommands 426
- \tracinggroups 694
- \tracingifs 690
- \tracinglostchars 427
- \tracingmacros 428
- \tracingnesting 689
- \tracingonline 429
- \tracingoutput 430
- \tracingpages 431
- \tracingparagraphs 432
- \tracingrestores 433
- \tracingscantokens 688
- \tracingstats 434
- U**
- \U 2718
- \uccode 669
- \uchyph 567
- \undefined 4583
- \underline 503
- \unexpanded 179, 183, 682
- \unhbox 608
- \unhcopy 609
- \unkern 533
- \unless 673
- \unpenalty 644
- \unskip 531
- \unvbox 610
- \unvcopy 611
- \uppercase 641
- \use:c 16, 852, 852, 984, 1782, 1926, 1936,
2008, 2009, 3332, 3621, 3631, 3774,
3783, 3785, 3787, 3788, 3792, 3949,
6709, 6720, 6733, 6736, 6742, 6753,
6761, 6767, 6789, 6850, 6872, 6877,
6885, 6908, 6930, 7240, 7247, 7350,
7514, 7805, 7835, 7838, 7855, 7858,
7859, 7862, 7865, 8149, 8211, 8267,
8317, 9459, 9546, 9757, 9944, 10280,
10807, 10870, 10885, 10887, 10978
- \use:n 18, 862, 862, 1000,
1029, 1449, 1451, 1455, 1463, 1465,
1473, 1477, 4973, 5190, 5670, 5885
- \use:nn 18, 862, 863, 1540, 3040, 3944
- \use:nnn 18, 862, 864
- \use:nnnn 18, 862, 865
- \use:x 19, 853, 853, 6392
- \use_0_parameter: 1267
- \use_1_parameter: 1267
- \use_2_parameter: 1267
- \use_3_parameter: 1267
- \use_4_parameter: 1267
- \use_5_parameter: 1267
- \use_6_parameter: 1267
- \use_7_parameter: 1267
- \use_8_parameter: 1267
- \use_9_parameter: 1267
- \use_i:nn 18, 866, 866,
896, 1086, 1115, 1143, 1305, 1453,
1467, 1475, 7905, 7951, 7976, 8544,
8863, 9220, 9253, 10036, 10364, 10503
- \use_i:nnn 18, 868, 868, 1097, 3049, 10003
- \use_i:nnnn 19, 868, 872
- \use_i_after_else:nw 20, 882, 883
- \use_i_after_fi:nw 20, 882, 882, 1341
- \use_i_after_or:nw 20, 882, 884
- \use_i_after_orelse:nw 20,
882, 885, 1321, 1323, 1325, 1327,
1329, 1331, 1333, 1335, 1337, 1339
- \use_i_delimit_by_q_nil:nw . 20, 879, 879
- \use_i_delimit_by_q_recursion_stop:nw
. 20, 52, 879,
881, 2088, 2398, 2427, 5524, 5584, 5816
- \use_i_delimit_by_q_stop:nw 20, 879, 880
- \use_i_ii:nnn 19, 868, 871, 1565
- \use_ii:nn 18,
866, 867, 898, 1088, 1117, 1145,
1307, 1450, 1456, 1464, 1478, 4686,
4688, 4690, 4692, 5410, 5662, 7082
- \use_ii:nnn . 18, 868, 869, 1099, 3058, 7138
- \use_ii:nnnn 19, 868, 873
- \use_iii:nnn 18, 868, 870, 3067
- \use_iii:nnnn 19, 868, 874

<code>\use_iv:nnnn</code>	19, 868, 875	<code>\vbox_set:Nn</code>	150, 6065, 6065–6067
<code>\use_none:n</code>	19, 886, 886, 1000, 1029, 1068, 1070, 1344, 1448, 1452, 1454, 1462, 1466, 1474, 1476, 1850, 2400, 2429, 2855, 3667, 3671, 3676, 4418, 4521, 4695, 4712, 4722, 4777, 4927, 4956, 4989, 5184, 5211, 5212, 5397, 5600, 5611, 6360, 6362, 6501–6504, 7079, 7115, 7919, 7966, 7991, 8040, 8082, 8494, 8530, 8603, 8629, 8683, 8804, 8947, 9267, 9286, 9435, 9490, 9532, 9576, 9733, 9857, 9927, 10149, 10266, 10309, 10693	<code>\vbox_set_inline_begin:c</code>	151, 6081
<code>\use_none:nn</code>	19, 886, 887, 1806, 4700, 4854, 10757	<code>\vbox_set_inline_begin:N</code>	151, 6081, 6081, 6084, 6085
<code>\use_none:nnn</code>	19, 886, 888, 5742, 7131	<code>\vbox_set_inline_end:</code>	151, 6081, 6087
<code>\use_none:nnnn</code>	19, 886, 889, 7876	<code>\vbox_set_split_to_ht:Nn</code>	152, 6093, 6093
<code>\use_none:nnnnn</code>	19, 886, 890	<code>\vbox_set_to_ht:cnn</code>	151, 6075
<code>\use_none:nnnnnn</code>	19, 886, 891	<code>\vbox_set_to_ht:Nnn</code>	151, 6075, 6075, 6078, 6079
<code>\use_none:nnnnnnn</code>	19, 886, 892	<code>\vbox_set_top:cn</code>	151, 6069
<code>\use_none:nnnnnnnn</code>	19, 886, 893	<code>\vbox_set_top:Nn</code>	151, 6069, 6069, 6072, 6073
<code>\use_none:nnnnnnnnn</code>	19, 886, 894, 1311	<code>\vbox_to_ht:nn</code>	150, 6062, 6062
<code>\use_none_delimit_by_q_nil:w</code>	20, 876, 876	<code>\vbox_to_zero:n</code>	150, 6062, 6064
<code>\use_none_delimit_by_q_recursion_stop:w</code>	20, 52, 876, 878, 982, 1050, 1778, 2392, 2414, 4567, 5523, 5815	<code>\vbox_top:n</code>	150, 6061
<code>\use_none_delimit_by_q_stop:w</code>	20, 876, 877, 2305, 2309, 5772, 5777, 6469	<code>\vbox_unpack:c</code>	152, 6089
<code>\usepackage</code>	223	<code>\vbox_unpack:N</code>	152, 6089, 6089, 6091
V		<code>\vbox_unpack_clear:c</code>	152, 6089
<code>\vadjust</code>	544	<code>\vbox_unpack_clear:N</code>	152, 6089, 6090, 6092
<code>\valign</code>	379	<code>\vcenter</code>	465
<code>.value_forbidden:</code>	174	<code>\vfil</code>	526
<code>.value_required:</code>	174	<code>\vfill</code>	528
<code>\vbadness</code>	619	<code>\vfildneg</code>	527
<code>\vbox</code>	614	<code>\vfuzz</code>	621
<code>\vbox:n</code>	150, 6060, 6060	<code>\voffset</code>	596
<code>\vbox:n\vbox_top:n</code>	6060	<code>\voidb@x</code>	6016
<code>\vbox_gset:cn</code>	150, 6065	<code>\vrule</code>	535
<code>\vbox_gset:Nn</code>	150, 6065, 6066, 6068	<code>\vsize</code>	578
<code>\vbox_gset_inline_begin:c</code>	151, 6081	<code>\vskip</code>	529
<code>\vbox_gset_inline_begin:N</code>	151, 6081, 6083, 6086	<code>\vsplit</code>	607
<code>\vbox_gset_inline_end:</code>	151, 6081, 6088	<code>\vss</code>	530
<code>\vbox_gset_to_ht:cnn</code>	151, 6075	<code>\vtop</code>	615
<code>\vbox_gset_to_ht:Nnn</code>	151, 6075, 6077, 6080	W	
<code>\vbox_gset_top:cn</code>	151, 6069	<code>\wd</code>	662
<code>\vbox_gset_top:Nn</code>	151, 6069, 6071, 6074	<code>\widowpenalties</code>	722
<code>\vbox_set:cn</code>	150, 6065	<code>\widowpenalty</code>	549
		<code>\write</code>	412
		X	
		<code>\X</code>	2714, 2718
		<code>\xdef</code>	353
		<code>\xetex_if_engine:F</code>	1455, 1466
		<code>\xetex_if_engine:T</code>	1454, 1465
		<code>\xetex_if_engine:TF</code>	4, 25, 1448, 1456, 1467
		<code>\xetex_if_engine_p:</code>	4
		<code>\xetex_XeTeXversion:D</code>	754, 1460
		<code>\XeTeXversion</code>	754
		<code>\xleaders</code>	538
		<code>\xspaceskip</code>	572

Y		Z	
<code>\Y</code> 2715, 2718	<code>\Z</code> 1812, 1820, 2716, 2718
<code>\year</code> 653	<code>\z@</code> 4032