

The L^AT_EX3 Sources

The L^AT_EX3 Project*

August 28, 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
3	Formal language conventions which apply generally	5
II	The <code>l3bootstrap</code> package: Bootstrap code	5
4	Using the <code>L^AT_EX3</code> modules	5
III	The <code>l3names</code> package: Namespace for primitives	7
5	Setting up the <code>L^AT_EX3</code> programming language	7
IV	The <code>l3basics</code> package: Basic definitions	7
6	No operation functions	8
7	Grouping material	8
8	Control sequences and functions	8
8.1	Defining functions	9
8.2	Defining new functions using primitive parameter text	9
8.3	Defining new functions using the signature	12
8.4	Copying control sequences	15
8.5	Deleting control sequences	16
8.6	Showing control sequences	16
8.7	Converting to and from control sequences	16

9	Using or removing tokens and arguments	18
9.1	Selecting tokens from delimited arguments	20
9.2	Decomposing control sequences	21
10	Predicates and conditionals	21
10.1	Tests on control sequences	23
10.2	Testing string equality	24
10.3	Engine-specific conditionals	24
10.4	Primitive conditionals	25
11	Internal kernel functions	26
V	The l3expan package: Argument expansion	27
12	Defining new variants	27
13	Methods for defining variants	28
14	Introducing the variants	29
15	Manipulating the first argument	30
16	Manipulating two arguments	31
17	Manipulating three arguments	32
18	Unbraced expansion	33
19	Preventing expansion	34
20	Internal functions and variables	35
VI	The l3prg package: Control structures	36
21	Defining a set of conditional functions	37

22	The boolean data type	39
23	Boolean expressions	41
24	Logical loops	43
25	Switching by case	43
26	Producing n copies	45
27	Detecting TeX's mode	46
28	Internal programming functions	46
29	Experimental programmings functions	47
VII	The l3quark package: Quarks	48
30	Defining quarks	48
31	Quark tests	49
32	Recursion	50
33	Internal quark functions	51
VIII	The l3token package: Token manipulation	51
34	All possible tokens	52
35	Character tokens	53
36	Generic tokens	56
37	Converting tokens	57
38	Token conditionals	57

39	Peeking ahead at the next token	61
40	Decomposing a macro definition	64
41	Experimental token functions	65
IX	The <code>l3int</code> package: Integers	66
42	Integer expressions	66
43	Creating and initialising integers	68
44	Setting and incrementing integers	68
45	Using integers	70
46	Integer expression conditionals	70
47	Integer expression loops	71
48	Formatting integers	72
49	Converting from other formats to integers	75
50	Viewing integers	75
51	Constant integers	76
52	Scratch integers	76
53	Internal functions	77
X	The <code>l3skip</code> package: Dimensions and skips	78
54	Creating and initialising <code>dim</code> variables	79
55	Setting <code>dim</code> variables	79

56	Utilities for dimension calculations	81
57	Dimension expression conditionals	82
58	Dimension expression loops	82
59	Using dim expressions and variables	84
60	Viewing dim variables	84
61	Constant dimensions	84
62	Scratch dimensions	85
63	Creating and initialising skip variables	85
64	Setting skip variables	85
65	Skip expression conditionals	87
66	Using skip expressions and variables	87
67	Viewing skip variables	88
68	Constant skips	88
69	Scratch skips	88
70	Creating and initialising muskip variables	88
71	Setting muskip variables	89
72	Using muskip expressions and variables	90
73	Inserting skips into the output	91
74	Viewing muskip variables	91
75	Internal functions	91

76	Experimental skip functions	92
XI	The l3tl package: Token lists	92
77	Creating and initialising token list variables	93
78	Adding data to token list variables	94
79	Modifying token list variables	96
80	Reassigning token list category codes	97
81	Reassigning token list character codes	98
82	Token list conditionals	99
83	Mapping to token lists	101
84	Using token lists	102
85	Working with the content of token lists	103
86	The first token from a token list	104
87	Viewing token lists	107
88	Constant token lists	107
89	Scratch token lists	108
90	Experimental token list functions	108
91	Internal functions	109
XII	The l3seq package: Sequences and stacks	109
92	Creating and initialising sequences	109

93	Appending data to sequences	111
94	Recovering items from sequences	112
95	Modifying sequences	113
96	Sequence conditionals	115
97	Mapping to sequences	115
98	Sequences as stacks	117
99	Viewing sequences	118
100	Experimental sequence functions	118
101	Internal sequence functions	121
XIII	The l3clist package: Comma separated lists	122
102	Creating and initialising comma lists	122
103	Appending items to comma lists	124
104	Comma lists as stacks	125
105	Using comma lists	127
106	Modifying comma lists	127
107	Comma list conditionals	128
108	Mapping to comma lists	129
109	Comma lists as stacks	131
110	Viewing comma lists	132
111	Experimental comma list functions	132

XIV The l3prop package: Property lists	133
112 Creating and initialising property lists	134
113 Adding entries to property lists	135
114 Recovering values from property lists	136
115 Modifying property lists	137
116 Property list conditionals	137
117 Recovering values from property lists with branching	138
118 Mapping to property lists	139
119 Viewing property lists	140
120 Experimental property list functions	140
121 Internal property list functions	141
XV The l3box package: Boxes	141
122 Creating and initialising boxes	142
123 Using boxes	143
124 Measuring and setting box dimensions	144
125 Box conditionals	145
126 The last box inserted	146
127 Constant boxes	146
128 Scratch boxes	146
129 Viewing box contents	146

130	Horizontal mode boxes	147
131	Vertical mode boxes	149
132	Primitive box conditionals	151
XVI	The <code>l3io</code> package: Input–output operations	152
133	Opening and closing streams	153
134	Writing to files	153
135	Wrapping lines in output	155
136	Reading from files	155
137	Internal input–output functions	157
XVII	The <code>l3msg</code> package: Messages	157
138	Creating new messages	158
139	Contextual information for messages	158
140	Issuing messages	160
141	Redirecting messages	162
142	Low-level message functions	163
143	Kernel-specific functions	164
144	Expandable errors	166
XVIII	The <code>l3keys</code> package: Key–value interfaces	166
145	Creating keys	168

146	Sub-dividing keys	172
147	Choice and multiple choice keys	173
148	Setting keys	175
149	Setting known keys only	176
150	Utility functions for keys	176
151	Low-level interface for parsing key-val lists	177
XIX	The <code>l3file</code> package: File operations	178
152	File operation functions	179
153	Internal file functions	180
XX	The <code>l3fp</code> package: Floating-point operations	180
154	Floating-point variables	181
155	Conversion of floating point values to other formats	183
156	Rounding floating point values	184
157	Floating-point conditionals	184
158	Unary floating-point operations	185
159	Floating-point arithmetic	186
160	Floating-point power operations	187
161	Exponential and logarithm functions	188
162	Trigonometric functions	188

163	Constant floating point values	189
164	Notes on the floating point unit	190
XXI	The <code>l3luatex</code> package: LuaTeX-specific functions	190
165	Breaking out to Lua	190
166	Category code tables	191
XXII	Implementation	192
167	Bootstrap code	193
167.1	Format-specific code	193
167.2	Package-specific code	194
167.3	Dealing with package-mode meta-data	195
167.4	The <code>\pdfstrcmp</code> primitive in X _Y TeX	199
167.5	Engine requirements	199
167.6	The L ^A T _E X3 code environment	200
168	<code>l3names</code> implementation	202
169	<code>l3basics</code> implementation	212
169.1	Renaming some T _E X primitives (again)	212
169.2	Defining functions	214
169.3	Selecting tokens	215
169.4	Gobbling tokens from input	217
169.5	Conditional processing and definitions	217
169.6	Dissecting a control sequence	222
169.7	Exist or free	224
169.8	Defining and checking (new) functions	226
169.9	More new definitions	229
169.10	Copying definitions	231

169.1	Undefining functions	231
169.1	Defining functions from a given number of arguments	232
169.1	Using the signature to define functions	234
169.1	Checking control sequence equality	236
169.1	Diagnostic wrapper functions	236
169.1	Engine specific definitions	237
169.1	Doing nothing functions	238
169.1	String comparisons	238
169.1	Deprecated functions	238
170	l3expan implementation	239
170.1	General expansion	239
170.2	Hand-tuned definitions	243
170.3	Definitions with the automated technique	246
170.4	Last-unbraced versions	247
170.5	Preventing expansion	248
170.6	Defining function variants	249
170.7	Variants which cannot be created earlier	252
171	l3prg implementation	252
171.1	Defining a set of conditional functions	252
171.2	The boolean data type	252
171.3	Boolean expressions	254
171.4	Logical loops	260
171.5	Switching by case	261
171.6	Producing n copies	263
171.7	Detecting T _E X's mode	266
171.8	Internal programming functions	267
171.9	Experimental programmings functions	268
171.10	Deprecated functions	271
172	l3quark implementation	271

173	l3token implementation	275
173.1	Character tokens	275
173.2	Generic tokens	277
173.3	Token conditionals	278
173.4	Peeking ahead at the next token	287
173.5	Decomposing a macro definition	292
173.6	Experimental token functions	293
173.7	Deprecated functions	294
174	l3int implementation	297
174.1	Integer expressions	297
174.2	Creating and initialising integers	299
174.3	Setting and incrementing integers	300
174.4	Using integers	301
174.5	Integer expression conditionals	302
174.6	Integer expression loops	305
174.7	Formatting integers	306
174.8	Converting from other formats to integers	311
174.9	Viewing integer	315
174.10	Constant integers	315
174.11	Scratch integers	316
174.12	Registers for earlier modules	316
174.13	Deprecated functions	316
175	l3skip implementation	317
175.1	Length primitives renamed	318
175.2	Creating and initialising <code>dim</code> variables	318
175.3	Setting <code>dim</code> variables	318
175.4	Utilities for dimension calculations	320
175.5	Dimension expression conditionals	320
175.6	Dimension expression loops	322
175.7	Using <code>dim</code> expressions and variables	323

175.8	Viewing <code>dim</code> variables	323
175.9	Constant dimensions	324
175.10	Scratch dimensions	324
175.11	Creating and initialising <code>skip</code> variables	324
175.12	Setting <code>skip</code> variables	325
175.13	<code>skip</code> expression conditionals	325
175.14	Using <code>skip</code> expressions and variables	326
175.15	Inserting skips into the output	326
175.16	Viewing <code>skip</code> variables	327
175.17	Constant skips	327
175.18	Scratch skips	327
175.19	Creating and initialising <code>muskip</code> variables	327
175.20	Setting <code>muskip</code> variables	328
175.21	Using <code>muskip</code> expressions and variables	329
175.22	Viewing <code>muskip</code> variables	329
175.23	Experimental skip functions	329
176	l3tl implementation	330
176.1	Functions	330
176.2	Adding to token list variables	331
176.3	Reassigning token list category codes	333
176.4	Reassigning token list character codes	335
176.5	Modifying token list variables	335
176.6	Token list conditionals	337
176.7	Mapping to token lists	341
176.8	Using token lists	342
176.9	Working with the contents of token lists	343
176.10	The first token from a token list	345
176.11	Viewing token lists	349
176.12	Constant token lists	350
176.13	Scratch token lists	350
176.14	Experimental functions	351
176.15	Deprecated functions	356

177	l3seq implementation	357
177.1	Allocation and initialisation	358
177.2	Appending data to either end	359
177.3	Modifying sequences	360
177.4	Sequence conditionals	361
177.5	Recovering data from sequences	362
177.6	Mapping to sequences	365
177.7	Sequence stacks	367
177.8	Viewing sequences	368
177.9	Experimental functions	369
177.10	Deprecated interfaces	374
178	l3clist implementation	375
178.1	Allocation and initialisation	375
178.2	Appending items to comma lists	377
178.3	Comma lists as stacks	377
178.4	Using comma lists	378
178.5	Modifying comma lists	379
178.6	Comma list conditionals	380
178.7	Mapping to comma lists	382
179	Viewing comma lists	384
179.1	Experimental functions	384
179.2	Deprecated interfaces	386
180	l3prop implementation	387
180.1	Allocation and initialisation	387
180.2	Accessing data in property lists	388
180.3	Property list conditionals	392
180.4	Recovering values from property lists with branching	393
180.5	Mapping to property lists	394
180.6	Viewing property lists	395
180.7	Experimental functions	395
180.8	Deprecated interfaces	397

181	l3box implementation	398
181.1	Creating and initialising boxes	398
181.2	Measuring and setting box dimensions	399
181.3	Using boxes	400
181.4	Box conditionals	400
181.5	The last box inserted	401
181.6	Constant boxes	401
181.7	Scratch boxes	402
181.8	Viewing box contents	402
181.9	Horizontal mode boxes	402
181.10	Vertical mode boxes	404
182	l3io implementation	405
182.1	Primitives	405
182.2	Variables and constants	406
182.3	Stream management	407
182.4	Deferred writing	412
182.5	Immediate writing	412
182.6	Hard-wrapping lines based on length	413
182.7	Special characters for writing	417
182.8	Reading input	417
182.9	Deprecated functions	418
183	l3msg implementation	419
184	Creating messages	419
184.1	Messages: support functions and text	420
184.2	Showing messages: low level mechanism	421
184.3	Displaying messages	423
184.4	Kernel-specific functions	429
184.5	Expandable errors	433
184.6	Deprecated functions	434

185	l3keys Implementation	435
185.1	Low-level interface	435
185.2	Constants and variables	439
185.3	The key defining mechanism	440
185.4	Turning properties into actions	442
185.5	Creating key properties	447
185.6	Setting keys	450
185.7	Utilities	453
185.8	Messages	454
185.9	Deprecated functions	455
186	l3file implementation	456
187	l3fp Implementation	459
187.1	Constants	460
187.2	Variables	461
187.3	Parsing numbers	463
187.4	Internal utilities	467
187.5	Operations for fp variables	468
187.6	Transferring to other types	473
187.7	Rounding numbers	480
187.8	Unary functions	483
187.9	Basic arithmetic	484
187.10	Arithmetic for internal use	493
187.11	Trigonometric functions	500
187.12	Exponent and logarithm functions	513
187.13	Tests for special values	535
187.14	Floating-point conditionals	535
187.15	Messages	541
188	l3luatex implementation	542
188.1	Category code tables	543
	Index	547

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 *cname*, and indicates that the argument will be turned into a `cname` before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying `TEX` structure containing the data. A `V` argument will be a single token (similar to `N`), for example `\foo:V \MyVariable`; on the other hand, using `v` a `cname` is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

- o** This means *expansion once*. In general, the **V** and **v** specifiers are favoured over **o** for recovering stored information. However, **o** is useful for correctly processing information with delimited arguments.
- x** The **x** specifier stands for *exhaustive expansion*: the plain $\text{\TeX}\ \backslash\text{edef}$.
- 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 $\text{\TeX}\ \textit{parameters}$. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w** Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some arbitrary string).

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

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

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

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

bool Either true or false.

box Box register.

clist Comma separated list.

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

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.

dim “Rigid” lengths.

fp floating-point values;

int Integer-valued count register.

prop Property list.

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

skip “Rubber” lengths.

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

tl Token list variables: placeholder for a token list.

1.1 Terminological inexactitude

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

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

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

2 Documentation conventions

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

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

<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 L^AT_EX 2_ε or plain T_EX. In these cases, the text will include an extra “**T_EXhackers note**” section:

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

The normal description text.

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

3 Formal language conventions which apply generally

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

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

Part II

The l3bootstrap package

Bootstrap code

4 Using the L^AT_EX3 modules

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

As the modules use a coding syntax different from standard $\text{\LaTeX} 2_{\epsilon}$ it provides a few functions for setting it up.

$\backslash\text{ExplSyntaxOn}$ $\backslash\text{ExplSyntaxOff}$	$\backslash\text{ExplSyntaxOn} \langle code \rangle \quad \backslash\text{ExplSyntaxOff}$
---	---

The $\backslash\text{ExplSyntaxOn}$ 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 $\backslash\text{ExplSyntaxOff}$ reverts to the document category code régime.

$\backslash\text{ExplSyntaxNamesOn}$ $\backslash\text{ExplSyntaxNamesOff}$	$\backslash\text{ExplSyntaxNamesOn} \langle code \rangle \quad \backslash\text{ExplSyntaxNamesOff}$
---	---

The $\backslash\text{ExplSyntaxOn}$ 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 $\backslash\text{ExplSyntaxOn}$, using $\backslash\text{ExplSyntaxNamesOn}$ does not cause spaces to be ignored. The $\backslash\text{ExplSyntaxNamesOff}$ reverts to the document category code régime.

$\backslash\text{ProvidesExplPackage}$ $\backslash\text{ProvidesExplClass}$ $\backslash\text{ProvidesExplFile}$	$\backslash\text{RequirePackage}\{\text{expl3}\}$ $\backslash\text{ProvidesExplPackage} \{\langle package \rangle\} \{\langle date \rangle\} \{\langle version \rangle\}$ $\{\langle description \rangle\}$
---	---

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

$\backslash\text{GetIdInfo}$	$\backslash\text{RequirePackage}\{\text{l3names}\}$ $\backslash\text{GetIdInfo} \$\text{Id}: \langle SVN info field \rangle \$ \{\langle description \rangle\}$
------------------------------	--

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with $\backslash\text{ExplFileName}$ for the part of the file name leading up to the period, $\backslash\text{ExplFileDate}$ for date, $\backslash\text{ExplFileVersion}$ for version and $\backslash\text{ExplFileDescription}$ 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 $\backslash\text{RequirePackage}$ or alike are loaded with usual $\text{\LaTeX} 2_{\epsilon}$ category codes and the $\text{\LaTeX} 3$ category code scheme is reloaded when needed afterwards. See implementation for details. If you use the $\backslash\text{GetIdInfo}$ command you can use the information when loading a package with

$\backslash\text{ProvidesExplPackage}\{\backslash\text{ExplFileName}\}\{\backslash\text{ExplFileDate}\}\{\backslash\text{ExplFileVersion}\}\{\backslash\text{ExplFileDescription}\}$

Part III

The l3names package

Namespace for primitives

5 Setting up the L^AT_EX3 programming language

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

- defines new names for all T_EX primitives;
- switches to the category code 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_ƎT_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_ƎT_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.

6 No operation functions

<code>\prg_do_nothing: *</code>

`\prg_do_nothing:`

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

<code>\scan_stop:</code>

`\scan_stop:`

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

7 Grouping material

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

`\group_begin:`
`\group_end:`

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

<code>\group_insert_after:N</code>

`\group_insert_after:N <token>`

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

8 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 \mathbf{x} expansion. In contrast, “protected” functions are not expanded within \mathbf{x} expansions.

8.1 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.

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

$\backslash\text{cs_set:Npn}$
$\backslash\text{cs_set:cpn}$
$\backslash\text{cs_set:Npx}$
$\backslash\text{cs_set:cpx}$

 $\backslash\text{cs_set:Npn } \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$

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.

$\backslash\text{cs_set_nopar:Npn}$
$\backslash\text{cs_set_nopar:cpn}$
$\backslash\text{cs_set_nopar:Npx}$
$\backslash\text{cs_set_nopar:cpx}$

 $\backslash\text{cs_set_nopar:Npn } \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$

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 $\backslash\text{par}$ tokens. The assignment of a meaning to $\langle function \rangle$ is restricted to the current T_EX group level.

$\backslash\text{cs_set_protected:Npn}$
$\backslash\text{cs_set_protected:cpn}$
$\backslash\text{cs_set_protected:Npx}$
$\backslash\text{cs_set_protected:cpx}$

 $\backslash\text{cs_set_protected:Npn } \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$

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.

$\backslash\text{cs_set_protected_nopar:Npn}$
$\backslash\text{cs_set_protected_nopar:cpn}$
$\backslash\text{cs_set_protected_nopar:Npx}$
$\backslash\text{cs_set_protected_nopar:cpx}$

 $\backslash\text{cs_set_protected_nopar:Npn } \langle function \rangle \langle parameters \rangle \{ \langle code \rangle \}$

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 $\backslash\text{par}$ tokens. The as-

signment of a meaning to $\langle function \rangle$ is restricted to the current \TeX group level. The $\langle function \rangle$ will not expand within an x -type argument.

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

`\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.

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

`\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 `\par` tokens. The assignment of a meaning to $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

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

`\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.

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

`\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 `\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.

8.3 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 T_EX 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 T_EX 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 T_EX 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`.

8.4 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.

8.5 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.

8.6 Showing control sequences

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

8.7 Converting to and from control sequences

<code>\use:c</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>\cs_end:</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>	★
---------------------------	---

`\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.

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

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

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

will result in the input stream containing

```
abc { def }
```

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

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

9.1 Selecting tokens from delimited arguments

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

<code>\use_none_delimit_by_q_nil:w</code>	★	<code>\use_none_delimit_by_q_nil:w <balanced text> \q_nil</code>
<code>\use_none_delimit_by_q_stop:w</code>	★	
<code>\use_none_delimit_by_q_recursion_stop:w</code>	★	

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

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

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

<code>\use_i_after_fi:nw</code>	★	<code>{<inserted tokens>} \fi:</code>
<code>\use_i_after_else:nw</code>	★	<code>{<inserted tokens>} \else:</code>
<code>\use_i_after_or:nw</code>	★	<code>{<inserted tokens>} \or:</code>
<code>\use_i_after_orelse:nw</code>	★	<code>{<inserted tokens>} \or: or \else:</code>

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:`.

9.2 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.

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

10.1 Tests on control sequences

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

Compares the definition of two *<control sequences>* and is logically `true` if the two are the same.

<code>\cs_if_exist_p:N *</code>	<code>\cs_if_exist_p:N <control sequence></code>
<code>\cs_if_exist:NTF *</code>	<code>\cs_if_exist:NNTF <control sequence> <true code></code>
<code>\cs_if_exist:c *</code>	<code><false code></code>

Tests whether the *<control sequence>* is currently defined (whether as a function or another control sequence type). Any valid definition of *<control sequence>* will evaluate as `true`.

<code>\cs_if_free_p:N *</code>	<code>\cs_if_free_p:N <control sequence></code>
<code>\cs_if_free:NNTF *</code>	<code>\cs_if_free:NNTF <control sequence> <true code></code>
<code>\cs_if_free:c *</code>	<code><false code></code>

Tests whether the *<control sequence>* is currently free to be defined. This test will be `false` if the *<control sequence>* currently exists (as defined by `\cs_if_exist:N`).

²If defined using the interface provided.

10.2 Testing string equality

<code>\str_if_eq_p:nn *</code>	<code>\str_if_eq_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq:nnTF *</code>	<code>\str_if_eq:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</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:xxTF *</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`. All versions of these functions are fully expandable (including those involving an `x`-type expansion).

10.3 Engine-specific conditionals

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

Detects if the document is being compiled using LuaTeX.

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

Detects if the document is being compiled using pdfTeX.

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

Detects if the document is being compiled using XeTeX.

<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.
--	--

10.4 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>*</code>	
<code>\if_false:</code>	<code>*</code>	
<code>\or:</code>	<code>*</code>	
<code>\else:</code>	<code>*</code>	
<code>\fi:</code>	<code>*</code>	<code>\if_true: <true code> \else: <false code> \fi:</code>
<code>\reverse_if:N</code>	<code>*</code>	<code>\if_false: <true code> \else: <false code> \fi:</code>
		<code>\reverse_if:N <primitive conditional></code>

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

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

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

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

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

<code>\if:w</code>	<code>*</code>	
<code>\if_charcode:w</code>	<code>*</code>	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	<code>*</code>	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>

These conditionals will expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with `\exp_not:N`. `\if_catcode:w` tests if the category codes of the two tokens are the same whereas `\if:w` tests if the character codes are identical. `\if_charcode:w` is an alternative name for `\if:w`.

<code>\if_predicate:w</code>	<code>*</code>	<code>\if_predicate:w <predicate> <true code> \else: <false code></code>
		<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 $\backslash\text{if_bool:N}$.)

$\backslash\text{if_bool:N} \star$	$\backslash\text{if_bool:N} \langle boolean \rangle \langle true\ code \rangle \backslash\text{else:} \langle false\ code \rangle \backslash\text{fi:}$
-------------------------------------	--

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

$\backslash\text{if_cs_exist:N} \star$	$\backslash\text{if_cs_exist:N} \langle cs \rangle \langle true\ code \rangle \backslash\text{else:} \langle false\ code \rangle \backslash\text{fi:}$
$\backslash\text{if_cs_exist:w} \star$	$\backslash\text{if_cs_exist:w} \langle tokens \rangle \backslash\text{cs_end:} \langle true\ code \rangle \backslash\text{else:} \langle false\ code \rangle \backslash\text{fi:}$

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 $\backslash\text{scan_stop:}$! This can be useful when dealing with control sequences which cannot be entered as a single token.

$\backslash\text{if_mode_horizontal:} \star$	$\backslash\text{if_mode_horizontal:} \langle true\ code \rangle \backslash\text{else:} \langle false\ code \rangle \backslash\text{fi:}$
$\backslash\text{if_mode_vertical:} \star$	
$\backslash\text{if_mode_math:} \star$	
$\backslash\text{if_mode_inner:} \star$	

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

11 Internal kernel functions

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

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

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

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

$\backslash\text{pref_global:D}$	$\backslash\text{pref_global:D} \backslash\text{cs_set_nopar:Npn}$
$\backslash\text{pref_long:D}$	
$\backslash\text{pref_protected:D}$	

Prefix functions that can be used in front of some definition functions (namely ...). The result of prefixing a function definition with $\backslash\text{pref_global:D}$ makes the definition global, $\backslash\text{pref_long:D}$ change the argument scanning mechanism so that it allows $\backslash\text{par}$ tokens

in the argument of the prefixed function, and `\pref_protected:D` makes the definition robust inside `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` ç.

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

Part V

The l3expan package

Argument expansion

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

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

12 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.

13 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.

14 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:Nc \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!

15 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>} {<tokens2>} ...`

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>} {<tokens2>} ...`

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others will be left unchanged.

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

`\exp_args:Nv *` `\exp_args:Nv <function> <variable> {<tokens2>} ...`

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

`\exp_args:Nv *` `\exp_args:Nv <function> {<tokens>} {<tokens2>} ...`

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). This control sequence should be the name of a `<variable>`. The content of the `<variable>` are recovered and placed inside braces into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others will be left unchanged.

`\exp_args:Nf *` `\exp_args:Nf <function> {<tokens>} {<tokens2>} ...`

This function absorbs two arguments (the `<function>` name and the `<tokens>`). 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.

<code>\exp_args:Nx</code>

`\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.

16 Manipulating two arguments

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

`\exp_args:NNc $\langle token1 \rangle$ $\langle token2 \rangle$ { $\langle tokens \rangle$ }`

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

`\exp_args:Noo $\langle token \rangle$ { $\langle tokens_1 \rangle$ } { $\langle tokens_2 \rangle$ }`

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.

17 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>

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

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>

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

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> $\langle token1 \rangle$ $\langle token2 \rangle$ $\langle tokens1 \rangle$ $\{ \langle tokens2 \rangle \}$
---	--

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.*

18 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>	<code>\exp_last_unbraced:Nno</code> $\langle token \rangle$ $\langle tokens1 \rangle$ $\langle tokens2 \rangle$
---	---

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> \star	<code>\exp_last_two_unbraced:Noo</code> $\langle token \rangle$ $\langle tokens1 \rangle$ $\{ \langle tokens2 \rangle \}$
---	---

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

<code>\exp_after:wN</code> \star	<code>\exp_after:wN</code> $\langle token1 \rangle$ $\langle token2 \rangle$
------------------------------------	--

Carries out a single expansion of $\langle token2 \rangle$ prior to expansion of $\langle token1 \rangle$. If $\langle token2 \rangle$ is a \TeX primitive, it will be executed rather than expanded, while if $\langle token2 \rangle$ has not

expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that $\langle token1 \rangle$ may be *any* single token, including group-opening and -closing tokens ($\{$ or $\}$ " assuming normal \TeX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate \exp_arg:N function.

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

19 Preventing expansion

\exp_not:N $\text{\exp_not:N} \langle token \rangle$

Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an \mathbf{x} -type argument.

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

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

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

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

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

\TeX hackers note: This is the ε - \TeX \unexpanded primitive.

\exp_not:V $\text{\exp_not:V} \langle variable \rangle$

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

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

Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence (which should be a $\langle variable \rangle$ name). The content of the $\langle variable \rangle$

is recovered, and further expansion is prevented in a context where it would otherwise be expanded, for example an **x**-type argument.

`\exp_not:o` `\exp_not:o {⟨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 (`_`).

20 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.

`\exp_eval_register:N *`
`\exp_eval_register:c *` `\exp_eval_register:N ⟨variable⟩`

These functions evaluates a *⟨variable⟩* as part of a **V** or **v** expansion (respectively), preceded by `\c_zero` which stops the expansion of a previous `\tex_romannumeral:D`. A *⟨variable⟩* 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>\cs_set_nopar:Npn \exp_args:Ncof { \::c \::o \::f \::: }</code>
<code>\N::</code>	
<code>\c::</code>	
<code>\o::</code>	
<code>\f::</code>	
<code>\x::</code>	
<code>\v::</code>	
<code>\V::</code>	
<code>\:::</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 <arg spec></code>
--	---

Tests if the function `\exp_args:N<arg spec>` exists, and defines it if it does not. The `<arg spec>` 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 `<true>` and `<false>` but other states are possible, say an `<error>` state for erroneous input, *e.g.*, text as input in a function comparing integers.

L^AT_EX3 has two primary forms of conditional flow processing based on these states. One type is predicate functions that turn the returned state into a boolean `<true>` or `<false>`. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean `<true>` or `<false>` values to be used in testing with `\if_predicate:w` or in functions to be described below. The 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 `<true>` or `<false>` depending on the result. Important to note here is that the arguments are executed after exiting the underlying `\if...\fi:` structure

21 Defining a set of conditional functions

<pre> \prg_new_conditional:Npnn \prg_set_conditional:Npnn \prg_new_conditional:Nnn \prg_set_conditional:Nnn </pre>	<pre> \prg_set_conditional:Npnn \<name>:\<arg spec> \<parameters> \<conditions> \<code> \prg_set_conditional:Nnn \<name>:\<arg spec> \<conditions> \<code> </pre>
--	---

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. $\backslash cs_new:Npn$) whereas the **set** version will not (cf. $\backslash cs_set:Npn$). 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:

- $\backslash \langle name \rangle_p:\langle arg\ spec \rangle$, 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.
- $\backslash \langle name \rangle:\langle arg\ spec \rangle T$, 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**.
- $\backslash \langle name \rangle:\langle arg\ spec \rangle F$, 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**.
- $\backslash \langle name \rangle:\langle arg\ spec \rangle TF$, 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 $\backslash prg_set_conditional:Npnn$: 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. If $\langle code \rangle$ is expandable then $\backslash prg_set_conditional:Npnn$ 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.

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

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

The `<code>` of the test may use `<parameters>` as specified by the second argument to `\prg_set_conditional:Npn`: 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. `\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.

<pre>\prg_new_eq_conditional:NN \prg_set_eq_conditional:NN</pre>	<pre>\prg_new_eq_conditional:NN \<name1>:<arg spec1> \<name2>:<arg spec2></pre>
--	---

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.

<pre>\prg_return_true: * \prg_return_false: *</pre>	<pre>\prg_return_true: \prg_return_false:</pre>
--	---

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 `\prg_set_conditional:Npnn`, *etc.*

22 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 `<boolean1>` equal to that of `<boolean2>`. This assignment is restricted to the current `TeX` group level.

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

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.

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

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

$\backslash\text{l_tmpa_bool}$

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.

23 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.

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

Tests the current truth of *⟨boolean expression⟩*, and continues expansion based on this result. The *⟨boolean expression⟩* should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. 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.

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

Function version of `!(⟨boolean expression⟩)` within a boolean expression.

<code>\bool_xor_p:nn *</code>	<code>\bool_xor_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₁⟩}</code>
-------------------------------	---

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

24 Logical loops

Loops using either boolean expressions or stored boolean values.

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

This function firsts checks the logical value of the $\langle\textit{boolean}\rangle$. If it is **false** the $\langle\textit{code}\rangle$ is placed in the input stream and expanded. After the completion of the $\langle\textit{code}\rangle$ the truth of the $\langle\textit{boolean}\rangle$ is re-evaluated. The process will then loop until the $\langle\textit{boolean}\rangle$ is **true**.

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

This function firsts checks the logical value of the $\langle\textit{boolean}\rangle$. If it is **true** the $\langle\textit{code}\rangle$ is placed in the input stream and expanded. After the completion of the $\langle\textit{code}\rangle$ the truth of the $\langle\textit{boolean}\rangle$ is re-evaluated. The process will then loop until the $\langle\textit{boolean}\rangle$ is **false**.

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

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

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

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

25 Switching by case

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

<code>\prg_case_int:nnn *</code>	<pre> \prg_case_int:nnn {<test integer expression>} { {<intexpr case₁>} {<code case₁>} {<intexpr case₂>} {<code case₂>} ... {<intexpr case_n>} {<code case_n>} } {<else case>} </pre>
----------------------------------	---

This function evaluates the $\langle\textit{test integer expression}\rangle$ 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
{ $\langle test\ dimension\ expression \rangle$ }
{
  { $\langle dime\!xpr\ case_1 \rangle$ } { $\langle code\ case_1 \rangle$ }
  { $\langle dime\!xpr\ case_2 \rangle$ } { $\langle code\ case_2 \rangle$ }
  ...
  { $\langle dime\!xpr\ case_n \rangle$ } { $\langle code\ case_n \rangle$ }
}
{ $\langle else\ case \rangle$ }
```

$\prg_case_dim:nnn$ ★

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.

```
\prg_case_str:nnn
{ $\langle test\ string \rangle$ }
{
  { $\langle string\ case_1 \rangle$ } { $\langle code\ case_1 \rangle$ }
  { $\langle string\ case_2 \rangle$ } { $\langle code\ case_2 \rangle$ }
  ...
  { $\langle string\ case_n \rangle$ } { $\langle code\ case_n \rangle$ }
}
{ $\langle else\ case \rangle$ }
```

$\prg_case_str:nnn$ ★
 $\prg_case_str:onnn$ ★
 $\prg_case_str:xxnn$ ★

This function compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$. If the two are equal (as described for $\backslash str_if_eq:nnTF$ 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. The **xx** variant is fully expandable, in the same way as the underlying

`\str_if_eq:xxTF` test.

```

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

```

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.

26 Producing n copies

```

\prg_replicate:nn ★ \prg_replicate:nn {\integer expression} {\tokens}

```

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.

```

\prg_stepwise_function:nnnN ★ \prg_stepwise_function:nnnN {\initial value} {\step}
                               {\final value} {\function}

```

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer expressions. The *<function>* is then placed in front of each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*). Thus *<function>* 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 {<initial value>} {<step>} {<final value>} {<code>}</code>
--	--

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

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

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

27 Detecting T_EX's mode

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

Detects if T_EX is currently in horizontal mode.

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

Detects if T_EX is currently in inner mode.

<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.

<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.

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

$\text{\texttt{\scan_align_safe_stop:}}$	$\text{\texttt{\scan_align_safe_stop:}}$
---	---

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

$\text{\texttt{\prg_variable_get_scope:N *}}$	$\text{\texttt{\prg_variable_get_scope:N}} \langle variable \rangle$
---	---

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

$\text{\texttt{\prg_variable_get_type:N *}}$	$\text{\texttt{\prg_variable_get_type:N}} \langle variable \rangle$
--	--

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

29 Experimental programmings functions

$\text{\texttt{\prg_quicksort:n}}$	$\text{\texttt{\prg_quicksort:n}} \{ \{ \langle item_1 \rangle \} \{ \langle item_2 \rangle \} \dots \{ \langle item_n \rangle \} \}$
-------------------------------------	--

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

$\text{\texttt{\prg_quicksort_function:n}}$ $\text{\texttt{\prg_quicksort_compare:nnTF}}$	$\text{\texttt{\prg_quicksort_function:n}} \{ \langle element \rangle \}$ $\text{\texttt{\prg_quicksort_compare:nnTF}} \{ \langle element_1 \rangle \} \{ \langle element_2 \rangle \}$
--	--

The two functions the programmer must define before calling $\text{\texttt{\prg_quicksort:n}}$. 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_`.

30 Defining quarks

`\quark_new:N` `\quark_new:N` $\langle quark \rangle$

Creates a new $\langle quark \rangle$ which expands only to $\langle quark \rangle$. The $\langle quark \rangle$ 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.

31 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.

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

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

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

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).

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

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

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

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

32 Recursion

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

`\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.

33 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.

34 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:

35 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 `\char_make_<type>` 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 `\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 $\backslash\mathrm{tl_to_uppeer}\mathrm{case:n}$, 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.

36 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.
-----------------------------------	--

37 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.

38 Token conditionals

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

39 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 $\backslash peek_after:Nw$ 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.

$\backslash peek_after:Nw$	$\backslash peek_after:Nw \langle function \rangle \langle token \rangle$
-----------------------------	--

Locally sets the test variable $\backslash l_peek_token$ 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.

$\backslash peek_gafter:Nw$	$\backslash peek_gafter:Nw \langle function \rangle \langle token \rangle$
------------------------------	---

Globally sets the test variable $\backslash g_peek_token$ 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 \square , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

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

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

`\peek_catcode:NTF` `\peek_catcode:NTF $\langle test token \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`
 Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ 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).

`\peek_catcode_ignore_spaces:NTF` `\peek_catcode_ignore_spaces:NTF $\langle test token \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_catcode: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).

`\peek_catcode_remove:NTF` `\peek_catcode_remove:NTF $\langle test token \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ 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).

`\peek_catcode_remove_ignore_spaces:NTF` `\peek_catcode_remove_ignore_spaces:NTF $\langle test token \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_catcode: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).

`\peek_charcode:NTF` `\peek_charcode:NTF $\langle test token \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

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

token) (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:NTF</code>	<code>\peek_charcode_ignore_spaces:NNTF</code> <i>test token</i> $\{ \langle \textit{true code} \rangle \} \{ \langle \textit{false code} \rangle \}$
---	--

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 *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_remove:NNTF</code>	<code>\peek_charcode_remove:NNTF</code> <i>test token</i> $\{ \langle \textit{true code} \rangle \} \{ \langle \textit{false code} \rangle \}$
---	---

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

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

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

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

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

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

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

40 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

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

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

`\token_get_replacement_text:N <token>`

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

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

`\token_get_prefix_spec:N <token>`

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

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

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

41 Experimental token functions

<code>\char_active_set:Npn</code>
<code>\char_active_set:Npx</code>

`\char_active_set:Npn <char> <parameters> {\code}`

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 \TeX group.

<code>\char_active_gset:Npn</code>
<code>\char_active_gset:Npx</code>

`\char_active_gset:Npn <char> <parameters> {\code}`

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.

<code>\char_active_set_eq:NN</code>

`\char_active_set_eq:NN <char> <function>`

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 \TeX group.

`\char_active_gset_eq:NN` $\backslash\text{char_active_gset_eq:NN}$ $\langle char \rangle$ $\langle function \rangle$

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.

`\peek_N_type:TF` $\backslash\text{peek_N_type:TF}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if the next $\langle token \rangle$ in the input stream can be safely grabbed as an N-type argument. The test will be $\langle false \rangle$ if the next $\langle token \rangle$ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), and $\langle true \rangle$ in all other cases. Note that a $\langle true \rangle$ result ensures that the next $\langle token \rangle$ is a valid N-type argument. However, if the next $\langle token \rangle$ is for instance `\c_space_token`, the test will take the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ will be left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).

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`”).

42 Integer expressions

`\int_eval:n *` $\backslash\text{int_eval:n}$ $\{\langle integer\ expression \rangle\}$

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

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

and

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

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

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

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

<code>\int_div_round:nn *</code>	<code>\int_div_round:nn {$\langle \textit{intexpr}_1 \rangle$} {$\langle \textit{intexpr}_2 \rangle$}</code>
----------------------------------	--

Evaluates the two $\langle \textit{integer expressions} \rangle$ 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 $\langle \textit{integer expression} \rangle$. The result is left in the input stream as a $\langle \textit{integer denotation} \rangle$ after two expansions.

<code>\int_div_truncate:nn *</code>	<code>\int_div_truncate:nn {$\langle \textit{intexpr}_1 \rangle$} {$\langle \textit{intexpr}_2 \rangle$}</code>
-------------------------------------	---

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

<code>\int_max:nn *</code>	<code>\int_max:nn {$\langle \textit{intexpr}_1 \rangle$} {$\langle \textit{intexpr}_2 \rangle$}</code>
<code>\int_min:nn *</code>	<code>\int_min:nn {$\langle \textit{intexpr}_1 \rangle$} {$\langle \textit{intexpr}_2 \rangle$}</code>

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

<code>\int_mod:nn *</code>	<code>\int_mod:nn {$\langle \textit{intexpr}_1 \rangle$} {$\langle \textit{intexpr}_2 \rangle$}</code>
----------------------------	--

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

43 Creating and initialising integers

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

`\int_new:N <integer>`

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 <integer> {<integer expression>}`

Creates a new constant *<integer>* or raises an error if the name is already taken. The value of the *<integer>* will be set globally to the *<integer expression>*.

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

`\int_zero:N <integer>`

Sets *<integer>* to 0 within the scope of the current TeX group.

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

`\int_gzero:N <integer>`

Sets *<integer>* 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 <integer1> <integer2>`

Sets the content of *<integer1>* equal to that of *<integer2>*. 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 <integer1> <integer2>`

Sets the content of *<integer1>* equal to that of *<integer2>*. This assignment is global and so is not limited by the current TeX group level.

44 Setting and incrementing integers

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

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

Adds the result of the *<integer expression>* to the current content of the *<integer>*. This

assignment is local.

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

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

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

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

`\int_decr:N <integer>`

Decreases the value stored in *<integer>* 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 *<integer>* 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 *<integer>* 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 *<integer>* 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 *<integer>* to the value of *<integer expression>*, 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 *<integer>* to the value of *<integer expression>*, 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 *<integer expression>* to the current content of the *<integer>*.

This assignment is local.

<code>\int_gsub:Nn</code> <code>\int_gsub:cn</code>	<code>\int_gsub:Nn <integer> {<integer expression>}</code>
--	--

Subtracts the result of the *<integer expression>* to the current content of the *<integer>*. This assignment is global.

45 Using integers

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

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.

46 Integer expression conditionals

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

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

Equal	=
Greater than	>
Less than	<

<code>\int_compare_p:n *</code> <code>\int_compare:nTF *</code>	<code>\int_compare_p:n</code> <code>{ <intexpr1> <relation> <intexpr2> }</code> <code>\int_compare:nTF</code> <code>{ <intexpr1> <relation> <intexpr2> }</code> <code>{<true code>} {<false code>}</code>
--	---

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

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

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

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

47 Integer expression loops

<code>\int_do_while:nNnn *</code>	<code>\int_do_while:nNnn</code> <code>{⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

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

<code>\int_do_until:nNnn *</code>	<code>\int_do_until:nNnn</code> <code>{⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

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

<code>\int_until_do:nNnn *</code>	<code>\int_until_do:nNnn</code> <code>{⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

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

<code>\int_while_do:nNnn *</code>	<code>\int_while_do:nNnn</code> <code>{⟨intexpr₂⟩} {⟨code⟩}</code>	<code>{⟨intexpr₁⟩} ⟨relation⟩</code>
-----------------------------------	--	---

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

<code>\int_do_while:nn *</code>	<code>\int_do_while:nNnn { $\langle intexpr1 \rangle$ $\langle relation \rangle$ $\langle intexpr2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	---

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

<code>\int_do_until:nn *</code>	<code>\int_do_until:nn { $\langle intexpr1 \rangle$ $\langle relation \rangle$ $\langle intexpr2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	---

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

<code>\int_until_do:nn *</code>	<code>\int_until_do:nn { $\langle intexpr1 \rangle$ $\langle relation \rangle$ $\langle intexpr2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	---

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle integer expressions \rangle$ as described for `\int_compare: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>\int_while_do:nn *</code>	<code>\int_while_do:nn { $\langle intexpr1 \rangle$ $\langle relation \rangle$ $\langle intexpr2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	---

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

48 Formatting integers

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

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

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

<code>\int_to_alph:n *</code>	<code>\int_to_alph:n {$\langle integer expression \rangle$}</code>
<code>\int_to_Alph:n *</code>	

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_binary: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).

49 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.

50 Viewing integers

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

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

51 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.

52 Scratch integers

```
\l_tmpa_int  
\l_tmpb_int  
\l_tmpc_int
```

Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be

overwritten by other non-kernel code and so should only be used for short-term storage.

<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.

53 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 it lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions *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>\if_int_compare:w *</code>	<code>\if_num:w <integer1> <relation> <integer2></code> <i><>true code></i> <code>\else:</code> <i><>false code></i> <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>	
<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\if_case:w *</code></div>	<code>\else:</code>	<code><default></code>	
<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\or: *</code></div>	<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>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\int_value:w *</code></div>	<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`.

<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\int_eval:w *</code></div>	<code>\int_eval:w</code>	<code><integer expression></code>	<code>\int_eval_end:</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\int_eval_end: *</code></div>			

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>	
<div style="border: 1px solid black; padding: 2px; display: inline-block;"><code>\if_int_odd:w *</code></div>	<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 μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

54 Creating and initialising dim variables

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

`\dim_new:N <dimension>`

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

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

`\dim_zero:N <dimension>`

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

55 Setting dim variables

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

`\dim_add: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 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` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

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

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` $\langle dimension1 \rangle$ $\langle dimension2 \rangle$

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` $\langle dimension1 \rangle$ $\langle dimension2 \rangle$

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

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

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

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

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

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

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

56 Utilities for dimension calculations

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

`\dim_ratio:nn { $\langle dimexpr_1 \rangle$ } { $\langle dimexpr_2 \rangle$ }`

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

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

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

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

will display 327680/655360 on the terminal.

57 Dimension expression conditionals

	<code>\dim_compare_p:nNn</code>	<code>{\dimexpr1} \langle relation \rangle {\dimexpr2}</code>
	<code>\dim_compare:nNnTF</code>	<code>{\dimexpr1} \langle relation \rangle {\dimexpr2}</code>
<code>\dim_compare_p:nNn *</code>		<code>{\true code} {\false code}</code>
<code>\dim_compare:nNnTF *</code>		

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

Equal	=
Greater than	>
Less than	<

	<code>\dim_compare_p:n</code>	<code>{ \dimexpr1 \langle relation \rangle \dimexpr2 }</code>
	<code>\dim_compare:nTF</code>	<code>{ \dimexpr1 \langle relation \rangle \dimexpr2 }</code>
<code>\dim_compare_p:n *</code>		<code>{\true code} {\false code}</code>
<code>\dim_compare:nTF *</code>		

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

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

58 Dimension expression loops

<code>\dim_do_while:nNnn *</code>	<code>\dim_do_while:nNnn</code>	<code>{\dimexpr1} \langle relation \rangle {\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 **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} \langle relation \rangle {\dimexpr2} {\code}</code>
-----------------------------------	---------------------------------	---

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

<code>\dim_until_do:nNnn *</code>	<code>\dim_until_do:nNnn {$\langle dimexpr_1 \rangle$} $\langle relation \rangle$ {$\langle dimexpr_2 \rangle$} {$\langle code \rangle$}</code>
-----------------------------------	---

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:nNnTF`. If the test is **false** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **true**.

<code>\dim_while_do:nNnn *</code>	<code>\dim_while_do:nNnn {$\langle dimexpr_1 \rangle$} $\langle relation \rangle$ {$\langle dimexpr_2 \rangle$} {$\langle code \rangle$}</code>
-----------------------------------	---

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:nNnTF`. If the test is **true** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **false**.

<code>\dim_do_while:nn *</code>	<code>\dim_do_while:nn {$\langle dimexpr_1 \rangle$ $\langle relation \rangle$ $\langle dimexpr_2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	--

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

<code>\dim_do_until:nn *</code>	<code>\dim_do_until:nn {$\langle dimexpr_1 \rangle$ $\langle relation \rangle$ $\langle dimexpr_2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	--

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

<code>\dim_until_do:nn *</code>	<code>\dim_until_do:nn {$\langle dimexpr_1 \rangle$ $\langle relation \rangle$ $\langle dimexpr_2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	--

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 **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 {$\langle dimexpr_1 \rangle$ $\langle relation \rangle$ $\langle dimexpr_2 \rangle$ } {$\langle code \rangle$}</code>
---------------------------------	--

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**.

59 Using dim expressions and variables

`\dim_eval:n *` `\dim_eval:n {⟨dimension expression⟩}`

Evaluates the *⟨dimension expression⟩*, expanding any dimensions and token list variables within the *⟨expression⟩* to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *⟨dimension denotation⟩* after two expansions. This 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 dimension⟩*.

`\dim_use:N *`
`\dim_use:c *` `\dim_use:N ⟨dimension⟩`

Recovers the content of a *⟨dimension⟩* 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 `\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.

60 Viewing dim variables

`\dim_show:N`
`\dim_show:c` `\dim_show:N ⟨dimension⟩`

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

61 Constant dimensions

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

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

62 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.

63 Creating and initialising skip variables

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

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

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

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

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

Sets $\langle skip \rangle$ to 0pt within the scope of the current T_EX group.

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

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

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

64 Setting skip variables

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

`\skip_add:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$

Adds the result of the $\langle skip \text{ expression} \rangle$ to the current content of the $\langle skip \rangle$. This assign-

ment 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.

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

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

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

65 Skip expression conditionals

	<code>\skip_if_eq_p:nn</code>	<code>{\<skipexpr₁>} {\<skipexpr₂>}</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\skip_if_eq_p:nn *</code> <code>\skip_if_eq:nnTF *</code> </div>	<code>\dim_compare:nTF</code>	<code>{\<skipexpr₁>} {\<skipexpr₂>}</code> <code>{\<true code>} {\<false code>}</code>

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

<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\skip_if_infinite_glue_p:n *</code> <code>\skip_if_infinite_glue:nTF *</code> </div>	<code>\skip_if_infinite_glue_p:n {\<skipexpr>}</code> <code>\skip_if_infinite_glue:nTF {\<skipexpr>}</code> <code>{\<true code>} {\<false code>}</code>
---	---

Evaluates the $\langle skip\ expression \rangle$ as described for `\skip_eval:n`, and then tests if this contains an infinite stretch or shrink component (or both).

66 Using skip expressions and variables

<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\skip_eval:n *</code> </div>	<code>\skip_eval:n {\<skip expression>}</code>
--	--

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

<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>\skip_use:N *</code> <code>\skip_use:c *</code> </div>	<code>\skip_use:N \<skip></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 `\skip_eval:n`).

\TeX hackers note: `\skip_use:N` is the \TeX primitive `\the`: this is one of several \LaTeX 3 names for this primitive.

67 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.

68 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).

69 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 $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<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 $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

70 Creating and initialising muskip variables

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

`\muskip_new:N <muskip>`
Creates a new `<muskip>` 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 $\langle muskip \rangle$`

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 $\langle muskip \rangle$`

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

71 Setting muskip variables

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

`\muskip_add:Nn $\langle muskip \rangle$ { $\langle muskip expression \rangle$ }`

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

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

`\muskip_gadd:Nn $\langle muskip \rangle$ { $\langle muskip expression \rangle$ }`

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

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

`\muskip_set:Nn $\langle muskip \rangle$ { $\langle muskip expression \rangle$ }`

Sets $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu. This assignment is restricted to the current T_EX group.

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

`\muskip_gset:Nn $\langle muskip \rangle$ { $\langle muskip expression \rangle$ }`

Sets $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu. 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>	

`\muskip_set_eq:NN $\langle muskip1 \rangle$ $\langle muskip2 \rangle$`

Sets the content of $\langle muskip1 \rangle$ equal to that of $\langle muskip2 \rangle$. This assignment is restricted

to the current \TeX group level.

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

 $\backslash\text{muskip_gset_eq:N} \langle\text{muskip1}\rangle \langle\text{muskip2}\rangle$

Sets the content of $\langle\text{muskip1}\rangle$ equal to that of $\langle\text{muskip2}\rangle$. This assignment is global and so is not limited by the current \TeX group level.

$\backslash\text{muskip_sub:Nn}$ $\backslash\text{muskip_sub:cn}$
--

 $\backslash\text{muskip_sub:Nn} \langle\text{muskip}\rangle \{\langle\text{muskip expression}\rangle\}$

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

$\backslash\text{muskip_gsub:Nn}$ $\backslash\text{muskip_gsub:cn}$
--

 $\backslash\text{muskip_gsub:Nn} \langle\text{muskip}\rangle \{\langle\text{muskip expression}\rangle\}$

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

72 Using muskip expressions and variables

$\backslash\text{muskip_eval:n} \star$

 $\backslash\text{muskip_eval:n} \{\langle\text{muskip expression}\rangle\}$

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

$\backslash\text{muskip_use:N} \star$ $\backslash\text{muskip_use:c} \star$
--

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

Recovers the content of a $\langle\text{skip}\rangle$ 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 $\langle\text{dimension}\rangle$ is required (such as in the argument of $\backslash\text{muskip_eval:n}$).

\TeX hackers note: $\backslash\text{muskip_use:N}$ is the \TeX primitive $\backslash\text{the}$: this is one of several \LaTeX 3 names for this primitive.

73 Inserting skips into the output

<code>\skip_horizontal:N</code>	
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:N <skip></code>
<code>\skip_horizontal:n</code>	<code>\skip_horizontal:n {\<skipexpr>}</code>

Inserts a horizontal *<skip>* 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 <skip></code>
<code>\skip_vertical:n</code>	<code>\skip_vertical:n {\<skipexpr>}</code>

Inserts a vertical *<skip>* into the current list.

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

74 Viewing muskip variables

<code>\muskip_show:N</code>	
<code>\muskip_show:c</code>	<code>\muskip_show:N <muskip></code>

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

75 Internal functions

<code>\if_dim:w</code>	<code><dimen1> <relation> <dimen1></code>
<code>\else:</code>	<code><true code></code>
<code>\fi:</code>	<code><false></code>

Compare two dimensions. The *<relation>* 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>

Evaluates *<dimension expression>* 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`.

76 Experimental skip functions

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN</code> $\langle\langle skipexpr \rangle\rangle$ $\langle\langle action \rangle\rangle$ $\langle dimen1 \rangle$ $\langle dimen2 \rangle$
--	---

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.

A token list can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use_none:n` grabs as its argument: either a single token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `{`, `{`, or `}` (assuming normal T_EX category codes). Functions which act on items are often faster than their analog acting directly on tokens.

77 Creating and initialising token list variables

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

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>	<code>\tl_const:Nn <tl var> {<token list>}</code>
--	---

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>	<code>\tl_clear:N <tl var></code>
--	---

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>	<code>\tl_gclear:N <tl var></code>
--	--

Clears all entries from the $\langle tl\ var \rangle$ globally.

<code>\tl_clear_new:N</code> <code>\tl_clear_new:c</code>	<code>\tl_clear_new:N <tl var></code>
--	---

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>	<code>\tl_gclear_new:N <tl var></code>
--	--

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>	<code>\tl_set_eq:NN <tl var1> <tl var2></code>
--	--

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.

78 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 \TeX grouping.

79 Modifying token list variables

$\backslash\text{tl_replace_once:Nnn}$ $\backslash\text{tl_replace_once:cnn}$	$\backslash\text{tl_replace_once:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	--

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 \TeX category codes). The assignment is restricted to the current \TeX group.

$\backslash\text{tl_greplace_once:Nnn}$ $\backslash\text{tl_greplace_once:cnn}$	$\backslash\text{tl_greplace_once:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	---

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 \TeX category codes). The assignment is applied globally.

$\backslash\text{tl_replace_all:Nnn}$ $\backslash\text{tl_replace_all:cnn}$	$\backslash\text{tl_replace_all:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	---

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 \TeX category codes). As this function operates from left to right, the pattern $\langle old tokens \rangle$ may remain after the replacement (see $\backslash\text{tl_remove_all:Nn}$ for an example). The assignment is restricted to the current \TeX group.

$\backslash\text{tl_greplace_all:Nnn}$ $\backslash\text{tl_greplace_all:cnn}$	$\backslash\text{tl_greplace_all:Nnn} \langle tl var \rangle \{ \langle old tokens \rangle \}$ $\{ \langle new tokens \rangle \}$
--	--

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 \TeX category codes). As this function operates from left to right, the pattern $\langle old tokens \rangle$ may remain after the replacement (see $\backslash\text{tl_remove_all:Nn}$ for an example). The assignment is applied globally.

$\backslash\text{tl_remove_once:Nn}$ $\backslash\text{tl_remove_once:cn}$	$\backslash\text{tl_remove_once:Nn} \langle tl var \rangle \{ \langle tokens \rangle \}$
--	--

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.

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

Removes the first (leftmost) occurrence of `<tokens>` from the `<tl var>`. `<Tokens>` 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). As this function operates from left to right, the pattern `<tokens>` may remain after the removal, for instance,

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

will result in `\l_tmpa_tl` containing `abcd`. 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). As this function operates from left to right, the pattern `<tokens>` may remain after the removal (see `\tl_remove_all:Nn` for an example). The assignment is applied globally.

80 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:cn</code> <code>\tl_set_rescan:cn</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 $\langle tokens \rangle$ are absorbed. The assignment is local to the current \TeX group. See also `\tl_rescan:nn`.

<code>\tl_gset_rescan:Nnn</code>	<code>\tl_gset_rescan:Nnn <tl var> {<setup>} {<tokens>}</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>	

Sets $\langle tl var \rangle$ to contain $\langle tokens \rangle$, applying the category code régime specified in the $\langle setup \rangle$ before carrying out the assignment. This allows the $\langle tl var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. The assignment is global. See also `\tl_rescan:nn`.

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

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

81 Reassigning token list character codes

<code>\tl_to_lowercase:n</code>	<code>\tl_to_lowercase:n {<tokens>}</code>
---------------------------------	--

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 $\langle tokens \rangle$.

\TeX hackers note: This is the \TeX primitive `\lowercase` renamed. As a result, this function takes place on execution and not on expansion.

<code>\tl_to_uppercase:n</code>	<code>\tl_to_uppercase:n {<tokens>}</code>
---------------------------------	--

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 $\langle tokens \rangle$.

\TeX hackers note: This is the \TeX primitive `\uppercase` renamed. As a result, this function takes place on execution and not on expansion.

82 Token list conditionals

<pre>\tl_if_blank_p:n *</pre>	<pre>\tl_if_blank_p:n {<token list>}</pre>
<pre>\tl_if_blank:nTF *</pre>	<pre>\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}</pre>
<pre>\tl_if_blank_p:V *</pre>	
<pre>\tl_if_blank:VTF *</pre>	
<pre>\tl_if_blank_p:o *</pre>	
<pre>\tl_if_blank:oTF *</pre>	

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

<pre>\tl_if_empty_p:N *</pre>	<pre>\tl_if_empty_p:N <tl var></pre>
<pre>\tl_if_empty:NTF *</pre>	<pre>\tl_if_empty:NTF <tl var> {<true code>} {<false code>}</pre>
<pre>\tl_if_empty_p:c *</pre>	
<pre>\tl_if_empty:cTF *</pre>	

Tests if the *<token list variable>* is entirely empty (*i.e.* contains no tokens at all).

<pre>\tl_if_empty_p:n *</pre>	<pre>\tl_if_empty_p:n {<token list>}</pre>
<pre>\tl_if_empty:nTF *</pre>	<pre>\tl_if_empty:nTF {<token list>} {<true code>} {<false code>}</pre>
<pre>\tl_if_empty_p:V *</pre>	
<pre>\tl_if_empty:VTF *</pre>	
<pre>\tl_if_empty_p:o *</pre>	
<pre>\tl_if_empty:oTF *</pre>	

Tests if the *<token list>* is entirely empty (*i.e.* contains no tokens at all).

<pre>\tl_if_eq_p:NN *</pre>	<pre>\tl_if_eq_p:NN {<tl var₁>} {<tl var₂>}</pre>
<pre>\tl_if_eq:NNTF *</pre>	<pre>\tl_if_eq:NNTF {<tl var₁>} {<tl var₂>} {<true code>}</pre>
<pre>\tl_if_eq_p:Nc *</pre>	
<pre>\tl_if_eq:NcTF *</pre>	
<pre>\tl_if_eq_p:cN *</pre>	
<pre>\tl_if_eq:cNTF *</pre>	
<pre>\tl_if_eq_p:cc *</pre>	
<pre>\tl_if_eq:ccTF *</pre>	<pre>{<false code>}</pre>

Compares the content of two *<token list variables>* 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**.

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

Tests if $\langle token list1 \rangle$ and $\langle token list2 \rangle$ are equal, both in respect of character codes and category codes.

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

Tests if the $\langle token list \rangle$ is found in the content of the $\langle token list variable \rangle$. The $\langle token list \rangle$ cannot contain the tokens `{`, `}` or `#` (assuming the usual \TeX category codes apply).

<code>\tl_if_in:nnTF</code> <code>\tl_if_in:VnTF</code> <code>\tl_if_in:onTF</code> <code>\tl_if_in:cnTF</code>	<code>\tl_if_in:nnTF {<token list1>} {<token list2>} {<true code>} {<false code>}</code>
--	--

Tests if $\langle token list2 \rangle$ is found inside $\langle token list1 \rangle$. The $\langle token list \rangle$ cannot contain the tokens `{`, `}` or `#` (assuming the usual \TeX category codes apply).

<code>\tl_if_single_p:N *</code> <code>\tl_if_single:NnTF *</code> <code>\tl_if_single_p:c *</code> <code>\tl_if_single:cnTF *</code>	<code>\tl_if_single_p:N {<tl var>}</code> <code>\tl_if_single:NnTF {<tl var>} {<true code>} {<false code>}</code>
--	--

Tests if the content of the $\langle tl var \rangle$ consists of a single item, *i.e.* is either a single normal token (excluding spaces, and brace tokens) 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`.

<code>\tl_if_single_p:n *</code> <code>\tl_if_single:nTF *</code>	<code>\tl_if_single_p:n {<token list>}</code> <code>\tl_if_single:nTF {<token list>} {<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`.

<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>} {<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 $\{ \dots \}$ are not single tokens.

83 Mapping to token lists

<code>\tl_map_function:NN *</code> <code>\tl_map_function:cN *</code>	<code>\tl_map_function:NN <tl var> <function></code>
--	--

Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle tl var \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also `\tl_map_function:nN`.

<code>\tl_map_function:nN *</code>	<code>\tl_map_function:nN <token list> <function></code>
------------------------------------	--

Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle token list \rangle$, The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also `\tl_map_function:NN`.

<code>\tl_map_inline:Nn</code> <code>\tl_map_inline:cN</code>	<code>\tl_map_inline:Nn <tl var> {\langle inline function \rangle}</code>
--	---

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

<code>\tl_map_inline:nn</code>	<code>\tl_map_inline:nn <token list> {\langle inline function \rangle}</code>
--------------------------------	---

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

<code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code>	<code>\tl_map_variable:NNn <tl var> <variable> {\langle function \rangle}</code>
--	--

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

<code>\tl_map_variable:nNn</code>	<code>\tl_map_variable:nNn <token list> <variable> {\langle function \rangle}</code>
-----------------------------------	--

Applies the $\langle function \rangle$ to every $\langle item \rangle$ stored within the $\langle token list \rangle$. The $\langle function \rangle$

should consist of code which will receive the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also `\tl_map_inline:nn`.

`\tl_map_break: *` `\tl_map_break:`

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 T_EX errors.

84 Using token lists

`\tl_to_str:N *`
`\tl_to_str:c *` `\tl_to_str:N $\langle tl var \rangle$`

Converts the content of the $\langle tl var \rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string \rangle$ is then left in the input stream.

`\tl_to_str:n *` `\tl_to_str:n $\{ \langle tokens \rangle \}$`

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

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

`\tl_use:N *`
`\tl_use:c *` `\tl_use:N $\langle tl var \rangle$`

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

85 Working with the content of token lists

<code>\tl_length:n *</code>
<code>\tl_length:V *</code>
<code>\tl_length:o *</code>

`\tl_length:n {⟨tokens⟩}`

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

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

`\tl_length:N {⟨tl var⟩}`

Counts the number of token groups in the $\langle tl var \rangle$ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group $\{\dots\}$. This process 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:V *</code>
<code>\tl_reverse:o *</code>

`\tl_reverse:n {⟨token list⟩}`

Reverses the order of the $\langle items \rangle$ in the $\langle token list \rangle$, so that $\langle item1 \rangle \langle item2 \rangle \langle item3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item3 \rangle \langle item2 \rangle \langle item1 \rangle$. This process will preserve unprotected space within the $\langle token list \rangle$. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

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

`\tl_reverse:N {⟨tl var⟩}`

Reverses the order of the $\langle items \rangle$ stored in $\langle tl var \rangle$, so that $\langle item1 \rangle \langle item2 \rangle \langle item3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item3 \rangle \langle item2 \rangle \langle item1 \rangle$. This process will preserve unprotected spaces within the $\langle token list variable \rangle$. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. The reversal is local to the current $\text{T}_{\text{E}}\text{X}$ group. See also `\tl_reverse:n`.

<code>\tl_reverse_items:n *</code>

`\tl_reverse_items:n {⟨token list⟩}`

Reverses the order of the $\langle items \rangle$ stored in $\langle tl var \rangle$, so that $\{\langle item1 \rangle\} \{\langle item2 \rangle\} \{\langle item3 \rangle\} \dots \{\langle item_n \rangle\}$ becomes $\{\langle item_n \rangle\} \dots \{\langle item3 \rangle\} \{\langle item2 \rangle\} \{\langle item1 \rangle\}$. This process will remove any unprotected space within the $\langle token list \rangle$. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are

initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider `\tl_reverse:n` or `\tl_reverse_tokens:n`.

<code>\tl_trim_spaces:n *</code>

`\tl_trim_spaces:n <token list>`

Removes any leading and trailing explicit space characters from the *<token list>* and leaves the result in the input stream. This process requires two expansions.

TeXhackers note: The result is return within the `\etex_unexpanded:D` primitive (`\exp_not:n`), which means that the token list will not expand further when appearing in an x-type argument expansion.

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

`\tl_trim_spaces:N <tl var>`

Removes any leading and trailing explicit space characters from the content of the *<tl var>* within the current TeX group.

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

`\tl_gtrim_spaces:N <tl var>`

Removes any leading and trailing explicit space characters from the content of the *<tl var>* globally.

86 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 in the input stream the first non-space token from the *<tokens>*. Any leading space tokens will be discarded, and thus for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

will both leave `a` in the input stream. An empty list of $\langle tokens \rangle$ or one which consists only of space (category code 10) tokens will result in `\tl_head:n` leaving nothing in the input stream.

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

`\tl_head:w $\langle tokens \rangle$ \q_stop`

Leaves in the input stream the first non-space token from the $\langle tokens \rangle$. An empty list of $\langle tokens \rangle$ or one which consists only of space (category code 10) tokens will result in an error, and thus $\langle tokens \rangle$ must *not* be “blank” as determined by `\tl_if_blank:n(TF)`. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<code>\tl_tail:n *</code>
<code>\tl_tail:V *</code>
<code>\tl_tail:v *</code>
<code>\tl_tail:f *</code>

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

Discards the all leading space tokens and the first non-space token in the $\langle tokens \rangle$, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { abc }
```

and

```
\tl_tail:n { ~ abc }
```

will both leave `bc` in the input stream. An empty list of $\langle tokens \rangle$ or one which consists only of space (category code 10) tokens will result in `\tl_tail:n` leaving nothing in the input stream.

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

`\tl_tail:w { $\langle tokens \rangle$ } \q_stop`

Discards the all leading space tokens and the first non-space token in the $\langle tokens \rangle$, and leaves the remaining tokens in the input stream. An empty list of $\langle tokens \rangle$ or one which consists only of space (category code 10) tokens will result in an error, and thus $\langle tokens \rangle$ must *not* be “blank” as determined by `\tl_if_blank:n(TF)`. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_tail:n` should be preferred if the number of expansions is not critical.

<code>\str_head:n *</code>
<code>\str_tail:n *</code>

`\str_head:n { $\langle tokens \rangle$ }`
`\str_tail:n { $\langle tokens \rangle$ }`

Converts the $\langle tokens \rangle$ into a string, as described for `\tl_to_str:n`. The `\str_head:n`

function then leaves the first character of this string in the input stream. The `\str_tail:n` function leaves all characters except the first in the input stream. The first character may be a space. If the $\langle tokens \rangle$ argument is entirely empty, nothing is left in the input stream.

<code>\tl_if_head_eq_catcode_p:nN *</code>	<code>\tl_if_head_eq_catcode_p:nN {$\langle token list \rangle$} $\langle test token \rangle$</code>
<code>\tl_if_head_eq_catcode:nNTF *</code>	<code>\tl_if_head_eq_catcode:nNTF {$\langle token list \rangle$} $\langle test token \rangle$</code>
	<code>{$\langle true code \rangle$} {$\langle false code \rangle$}</code>

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same category code as the $\langle test token \rangle$. In the case where $\langle token list \rangle$ is empty, its head is considered to be `\q_nil`, and the test will be true if $\langle test token \rangle$ is a control sequence.

<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_p:nN {$\langle token list \rangle$} $\langle test token \rangle$</code>
<code>\tl_if_head_eq_charcode:fNTF *</code>	<code>\tl_if_head_eq_charcode:nNTF {$\langle token list \rangle$} $\langle test token \rangle$</code>
	<code>{$\langle true code \rangle$} {$\langle false code \rangle$}</code>

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same character code as the $\langle test token \rangle$. In the case where $\langle token list \rangle$ is empty, its head is considered to be `\q_nil`, and the test will be true if $\langle test token \rangle$ is a control sequence.

<code>\tl_if_head_eq_meaning_p:nN *</code>	<code>\tl_if_head_eq_meaning_p:nN {$\langle token list \rangle$} $\langle test token \rangle$</code>
<code>\tl_if_head_eq_meaning:nNTF *</code>	<code>\tl_if_head_eq_meaning:nNTF {$\langle token list \rangle$} $\langle test token \rangle$</code>
	<code>{$\langle true code \rangle$} {$\langle false code \rangle$}</code>

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same meaning as the $\langle test token \rangle$. In the case where $\langle token list \rangle$ is empty, its head is considered to be `\q_nil`, and the test will be true if $\langle test token \rangle$ has the same meaning as `\q_nil`.

<code>\tl_if_head_group_p:n *</code>	<code>\tl_if_head_group_p:n {$\langle token list \rangle$}</code>
<code>\tl_if_head_group:nNTF *</code>	<code>\tl_if_head_group:nNTF {$\langle token list \rangle$}</code>
	<code>{$\langle true code \rangle$} {$\langle false code \rangle$}</code>

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

<code>\tl_if_head_N_type_p:n *</code>	<code>\tl_if_head_N_type_p:n {$\langle token list \rangle$}</code>
<code>\tl_if_head_N_type:nNTF *</code>	<code>\tl_if_head_N_type:nNTF {$\langle token list \rangle$}</code>
	<code>{$\langle true code \rangle$} {$\langle false code \rangle$}</code>

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is a normal N-type argument. In other words,

it is neither an explicit space character (with category code 10 and character code 32) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields false, as it does not have a “normal” first token. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_space_p:n *</code>	<code>\tl_if_head_space_p:n {<token list>}</code>
<code>\tl_if_head_space:nTF *</code>	<code>\tl_if_head_space:nTF {<token list>}</code> <code>{<true code>} {<false code>}</code>

Tests if the first *<token>* in the *<token list>* is an explicit space character (with category code 10 and character code 32). If *<token list>* starts with an implicit token such as `\c_space_token`, the test will yield false, as well as if the argument is empty. This function is useful to implement actions on token lists on a token by token basis.

T_EXhackers note: When T_EX 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.

87 Viewing token lists

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

Displays the content of the *<tl var>* on the terminal.

T_EXhackers note: `\tl_show:N` is the T_EX primitive `\show`.

<code>\tl_show:n</code>	<code>\tl_show:n <token list></code>
-------------------------	--

Displays the *<token list>* on the terminal.

T_EXhackers note: `\tl_show:n` is the ε -T_EX primitive `\showtokens`.

88 Constant token lists

<code>\c_job_name_tl</code>	Constant that gets the “job name” assigned when T _E X starts.
-----------------------------	--

T_EXhackers note: This is the new name for the primitive `\jobname`. It is a constant that is set by T_EX and should not be overwritten by the package.

`\c_empty_tl` Constant that is always empty.

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

89 Scratch token lists

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

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

90 Experimental token list functions

`\tl_reverse_tokens:n *` `\tl_reverse_tokens:n {⟨tokens⟩}`

This function, which works directly on T_EX tokens, reverses the order of the `⟨tokens⟩`: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{() (b)~a` in the input stream. This function requires two steps of expansion.

`\tl_length_tokens:n *` `\tl_length_tokens:n {⟨tokens⟩}`

Counts the number of T_EX tokens in the `⟨tokens⟩` and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total;

thus for instance, the length of `a~{bc}` is 6. This function requires three expansions, giving an *integer denotation*.

<code>\tl_expandable_uppercase:n *</code> <code>\tl_expandable_lowercase:n *</code>	<code>\tl_expandable_uppercase:n {<tokens>}</code> <code>\tl_expandable_lowercase:n {<tokens>}</code>
--	--

The `\tl_expandable_uppercase:n` function works through all of the *<tokens>*, replacing characters in the range `a–z` (with arbitrary category code) by the corresponding letter in the range `A–Z`, with category code 11 (letter). Similarly, `\tl_expandable_lowercase:n` replaces characters in the range `A–Z` by letters in the range `a–z`, and leaves other tokens unchanged. This function requires two steps of expansion.

T_EXhackers note: Begin-group and end-group characters are normalized and become `{` and `}`, respectively.

91 Internal functions

<code>\q_tl_act_mark</code> <code>\q_tl_act_stop</code>	Quarks which are only used for the particular purposes of <code>\tl_act...</code> functions.
--	--

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.

92 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.

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

`\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.

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

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

Clears all entries from the $\langle sequence \rangle$ globally.

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

`\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 `\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.

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

`\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 `\seq_new:N`). Thus the sequence is guaranteed to be available and globally clear.

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

`\seq_set_eq:NN $\langle sequence1 \rangle$ $\langle sequence2 \rangle$`

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.

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

`\seq_gset_eq:NN $\langle sequence1 \rangle$ $\langle sequence2 \rangle$`

Sets the content of $\langle sequence1 \rangle$ equal to that of $\langle sequence2 \rangle$. This assignment is global and so is not limited by the current $\text{T}_{\text{E}}\text{X}$ group level.

<code>\seq_concat:NNN</code>
<code>\seq_concat:ccc</code>

`\seq_concat:NNN $\langle sequence1 \rangle$ $\langle sequence2 \rangle$ $\langle sequence3 \rangle$`

Concatenates the content of $\langle sequence2 \rangle$ and $\langle sequence3 \rangle$ together and saves the result in

$\langle sequence1 \rangle$. The items in $\langle sequence2 \rangle$ will be placed at the left side of the new sequence. This operation is local to the current \TeX group and will remove any existing content in $\langle sequence1 \rangle$.

$\backslash seq_gconcat:Nn$ $\backslash seq_gconcat:ccc$	$\backslash seq_gconcat:Nn \langle sequence1 \rangle \langle sequence2 \rangle \langle sequence3 \rangle$
---	--

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 $\langle sequence1 \rangle$.

93 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.

94 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.

95 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>

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

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

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

Removes every occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is local to the current T_EX group.

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

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

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.

96 Sequence conditionals

<code>\seq_if_empty_p:N</code>	★	
<code>\seq_if_empty:N</code>	<u><i>TF</i></u>	★
<code>\seq_if_empty_p:c</code>	★	
<code>\seq_if_empty:c</code>	<u><i>TF</i></u>	★
<code>\seq_if_empty_p:N</code> <i><sequence></i>		
<code>\seq_if_empty:N</code> <i><sequence></i> <i>{<true code>}</i> <i>{<false code>}</i>		

Tests if the *<sequence>* is empty (containing no items).

<code>\seq_if_in:Nn</code>	<u><i>TF</i></u>	
<code>\seq_if_in:N</code>	<u><i>V</i></u>	<u><i>TF</i></u>
<code>\seq_if_in:Nv</code>	<u><i>TF</i></u>	
<code>\seq_if_in:No</code>	<u><i>TF</i></u>	
<code>\seq_if_in:Nx</code>	<u><i>TF</i></u>	
<code>\seq_if_in:cn</code>	<u><i>TF</i></u>	
<code>\seq_if_in:cV</code>	<u><i>TF</i></u>	
<code>\seq_if_in:cv</code>	<u><i>TF</i></u>	
<code>\seq_if_in:co</code>	<u><i>TF</i></u>	
<code>\seq_if_in:cx</code>	<u><i>TF</i></u>	
<code>\seq_if_in:Nn</code> <i><sequence></i> <i>{<item>}</i>		
<i>{<true code>}</i> <i>{<false code>}</i>		

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

97 Mapping to sequences

<code>\seq_map_function:NN</code>	★	
<code>\seq_map_function:cN</code>	★	
<code>\seq_map_function:NN</code> <i><sequence></i> <i><function></i>		

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

<code>\seq_map_inline:Nn</code>		
<code>\seq_map_inline:cn</code>		
<code>\seq_map_inline:Nn</code> <i><sequence></i> <i>{<inline function>}</i>		

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

<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</code> <i><sequence></i>		
<i><tl var.></i> <i>{<function using tl var.>}</i>		

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.

98 Sequences as stacks

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

`\seq_get:NN`

`\seq_get:cN`

`\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.

`\seq_pop:NN`

`\seq_pop:cN`

`\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.

`\seq_gpop:NN`

`\seq_gpop:cN`

`\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.

`\seq_push:Nn`

`\seq_push:NV`

`\seq_push:Nv`

`\seq_push:No`

`\seq_push:Nx`

`seq_push:cn`

`\seq_push:cV`

`\seq_push:cv`

`\seq_push:co`

`\seq_push:cx`

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

Adds the $\{\langle item \rangle\}$ to the top of the $\langle sequence \rangle$. The assignment is restricted to the

current \TeX group.

<code>\seq_gpush:Nn</code>
<code>\seq_gpush:NV</code>
<code>\seq_gpush:Nv</code>
<code>\seq_gpush:No</code>
<code>\seq_gpush:Nx</code>
<code>\seq_gpush:cn</code>
<code>\seq_gpush:cV</code>
<code>\seq_gpush:cv</code>
<code>\seq_gpush:co</code>
<code>\seq_gpush:cx</code>

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

Pushes the $\langle item \rangle$ onto the end of the top of the $\langle sequence \rangle$. The assignment is global.

99 Viewing sequences

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

`\seq_show:N <sequence>`

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

100 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 <sequence> <token list variable> {<true code>} {<false code>}`

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 $\langle token list variable \rangle$ is assigned locally.

<code>\seq_get_right:NNTF</code>
<code>\seq_get_right:cNTF</code>

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

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 $\langle token list variable \rangle$ 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 $\langle sequence \rangle$. 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 $\langle sequence \rangle$ 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 $\langle sequence \rangle$. 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 $\langle sequence \rangle$ 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 $\langle integer denotation \rangle$. The total number of items in a $\langle sequence \rangle$ will include those which are empty and duplicates, *i.e.* every item in a $\langle sequence \rangle$ is unique.

$\backslash seq_item: Nn \star$ $\backslash seq_item: cn \star$	$\backslash seq_item: Nn \langle sequence \rangle \{ \langle integer expression \rangle \}$
--	--

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>	<code>\seq_use:N</code>	$\langle sequence \rangle$
--	-------------------------	----------------------------

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>	<code>\seq_mapthread_function:NNN</code>	$\langle seq1 \rangle$	$\langle seq2 \rangle$	$\langle function \rangle$
--	--	------------------------	------------------------	----------------------------

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>	<code>\seq_set_from_clist:NN</code>	$\langle sequence \rangle$	$\langle comma-list \rangle$
--	-------------------------------------	----------------------------	------------------------------

Sets the $\langle sequence \rangle$ within the current $\text{T}_{\text{E}}\text{X}$ 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 $\langle comma-list \rangle$.

$\backslash seq_set_reverse:N$ $\backslash seq_gset_reverse:N$	$\backslash seq_set_reverse:N \langle sequence \rangle$
---	---

Reverses the order of items in the $\langle sequence \rangle$, and assigns the result to $\langle sequence \rangle$, locally or globally according to the variant chosen.

$\backslash seq_set_split:Nnn$ $\backslash seq_gset_split:Nnn$	$\backslash seq_set_split:Nnn \langle sequence \rangle$ $\{ \langle delimiter \rangle \} \{ \langle token list \rangle \}$
---	---

This function splits the $\langle token list \rangle$ into $\langle items \rangle$ separated by $\langle delimiter \rangle$, ignoring all explicit space characters from both sides of each $\langle item \rangle$, then removing one set of outer braces if any. The result is assigned to $\langle sequence \rangle$, locally or globally according to the function chosen. The $\langle delimiter \rangle$ may not contain $\{$, $\}$ or $\#$ (assuming T_EX's normal category code régime).

101 Internal sequence functions

$\backslash seq_if_empty_err_break:N$	$\backslash seq_if_empty_err_break:N \langle sequence \rangle$
---	--

Tests if the $\langle sequence \rangle$ is empty, and if so issues an error message before skipping over any tokens up to $\backslash seq_break_point:n$. This function is used to avoid more serious errors which would otherwise occur if some internal functions were applied to an empty $\langle sequence \rangle$.

$\backslash seq_item:n \star$	$\backslash seq_item:n \langle item \rangle$
--------------------------------	---

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

$\backslash seq_push_item_def:n$ $\backslash seq_push_item_def:x$	$\backslash seq_push_item_def:n \{ \langle code \rangle \}$
--	--

Saves the definition of $\backslash seq_item:n$ and redefines it to accept one parameter and expand to $\langle code \rangle$. This function should always be balanced by use of $\backslash seq_pop_item_def:$.

$\backslash seq_pop_item_def:$	$\backslash seq_pop_item_def:$
-----------------------------------	-----------------------------------

Restores the definition of $\backslash seq_item:n$ most recently saved by $\backslash seq_push_item_def:n$. This function should always be used in a balanced pair with $\backslash seq_push_item_def:n$.

$\backslash seq_break: \star$	$\backslash seq_break:$
--------------------------------	--------------------------

Used to terminate sequence functions by gobbling all tokens up to $\backslash seq_break_point:n$.

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.

102 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ group. The $\langle comma list \rangle$ will exist globally, but the content outside of the current $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ 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 T_EX group and will remove any existing content in $\langle comma list1 \rangle$.

<code>\clist_gconcat:NNN</code> <code>\clist_gconcat:ccc</code>	<code>\clist_gconcat:NNN</code> $\langle comma list1 \rangle$ $\langle comma list2 \rangle$ $\langle comma list3 \rangle$
--	--

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 global and will remove any existing content in $\langle comma list1 \rangle$.

103 Appending items to comma lists

<code>\clist_put_left:Nn</code> <code>\clist_put_left:NV</code> <code>\clist_put_left:No</code> <code>\clist_put_left:Nx</code> <code>\clist_put_left:cn</code> <code>\clist_put_left:cV</code> <code>\clist_put_left:co</code> <code>\clist_put_left:cx</code>	<code>\clist_put_left:Nn</code> $\langle comma list \rangle$ $\{ \langle item \rangle \}$
--	---

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.

104 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.

105 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>*.

106 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.

107 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).

<code>\clist_if_eq_p:NN *</code>	
<code>\clist_if_eq:NNTF *</code>	
<code>\clist_if_eq_p:Nc *</code>	
<code>\clist_if_eq:NcTF *</code>	
<code>\clist_if_eq_p:cN *</code>	
<code>\clist_if_eq:cNTF *</code>	
<code>\clist_if_eq_p:cc *</code>	<code>\clist_if_eq_p:NN {$\langle clist_1 \rangle$} {$\langle clist_2 \rangle$}</code>
<code>\clist_if_eq:ccTF *</code>	<code>\clist_if_eq:NNTF {$\langle clist_1 \rangle$} {$\langle clist_2 \rangle$} {$\langle true code \rangle$}</code> <code>{$\langle false code \rangle$}</code>

Compares the content of two $\langle comma lists \rangle$ and is logically **true** if the two contain the same list of entries in the same order.

<code>\clist_if_in:NnTF</code>	
<code>\clist_if_in:NVTF</code>	
<code>\clist_if_in:NoTF</code>	
<code>\clist_if_in:cnTF</code>	
<code>\clist_if_in:cVTF</code>	
<code>\clist_if_in:coTF</code>	
<code>\clist_if_in:nnTF</code>	
<code>\clist_if_in:nVTF</code>	
<code>\clist_if_in:noTF</code>	<code>\clist_if_in:NnTF $\langle comma list \rangle$ {$\langle item \rangle$}</code> <code>{$\langle true code \rangle$} {$\langle false code \rangle$}</code>

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

108 Mapping to comma lists

<code>\clist_map_function:NN *</code>	
<code>\clist_map_function:cN *</code>	
<code>\clist_map_function:nN *</code>	<code>\clist_map_function:NN $\langle comma list \rangle$ $\langle function \rangle$</code>

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

<code>\clist_map_inline:Nn</code>	
<code>\clist_map_inline:cn</code>	
<code>\clist_map_inline:nn</code>	<code>\clist_map_inline:Nn $\langle comma list \rangle$ {$\langle inline function \rangle$}</code>

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle comma list \rangle$. The $\langle inline$

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</code> <i>⟨comma list⟩</i> <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 T_EX errors.

T_EXhackers 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 *⟨tokens⟩* 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 *tokens* are inserted into the input stream. This will depend on the design of the mapping function.

109 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> <i><comma list></i> <i><token list variable></i>
--	---

Reads the top item from a *<comma list>* into the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally. If *<comma list>* is empty an error will be raised.

<code>\clist_pop:NN</code> <code>\clist_pop:cN</code>	<code>\clist_pop:NN</code> <i><comma list></i> <i><token list variable></i>
--	---

Pops the top item from a *<comma list>* into the *<token list variable>*. Both of the variables are assigned locally. If *<comma list>* is empty an error will be raised.

<code>\clist_gpop:NN</code> <code>\clist_gpop:cN</code>	<code>\clist_gpop:NN</code> <i><comma list></i> <i><token list variable></i>
--	--

Pops the top item from a *<comma list>* into the *<token list variable>*. The *<comma list>* is modified globally, while the *<token list variable>* is assigned locally. If *<comma list>* 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 <comma list> {<item>}`

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 <comma list> {<item>}`

Pushes the $\langle item \rangle$ onto the end of the top of the $\langle comma list \rangle$. The assignment is global.

110 Viewing comma lists

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

`\clist_show:N <comma list>`

Displays the entries in the $\langle comma list \rangle$ in the terminal.

111 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 <comma list>`

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`.

112 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.

113 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 T_EX 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.

114 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 T_EX group. See also $\backslash prop_get:NnNTF$.

$\backslash prop_pop:NnN$ $\backslash prop_pop:NoN$ $\backslash prop_pop:cnN$ $\backslash prop_pop:coN$
--

 $\backslash prop_pop:NnN \langle property list \rangle \{ \langle key \rangle \} \langle tl var \rangle$

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.

115 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.

116 Property list conditionals

<code>\prop_if_empty_p:N *</code>
<code>\prop_if_empty:N\overline{TF} *</code>
<code>\prop_if_empty_p:c *</code>
<code>\prop_if_empty:c\overline{TF} *</code>

`\prop_if_empty_p:N $\langle property list \rangle$`
`\prop_if_empty:N \overline{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).

<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>	
	<code>\prop_if_in:NnTF $\langle property list \rangle$ {$\langle key \rangle$} {$\langle true code \rangle$} {$\langle false code \rangle$}</code>

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nnTF`.

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`.

117 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated value. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

<code>\prop_get:NnNTF</code>	<code>\prop_get:NnNTF $\langle property list \rangle$ {$\langle key \rangle$} $\langle token list variable \rangle$ {$\langle true code \rangle$} {$\langle false code \rangle$}</code>
<code>\prop_get:cnNTF</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 $\langle token list variable \rangle$ is assigned locally.

<code>\prop_pop:NnNTF</code>	<code>\prop_pop:NnNTF $\langle property list \rangle$ {$\langle key \rangle$} $\langle token list variable \rangle$ {$\langle true code \rangle$} {$\langle false code \rangle$}</code>
<code>\prop_pop:cnNTF</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⟩, pops the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*property list*⟩. Both the ⟨*property list*⟩ and the ⟨*token list variable*⟩ are assigned locally.

118 Mapping to property lists

<code>\prop_map_function:NN *</code> <code>\prop_map_function:cN *</code>	<code>\prop_map_function:NN</code> ⟨ <i>property list</i> ⟩ ⟨ <i>function</i> ⟩
--	---

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

<code>\prop_map_inline:Nn</code> <code>\prop_map_inline:cN</code>	<code>\prop_map_inline:Nn</code> ⟨ <i>property list</i> ⟩ {⟨ <i>inline function</i> ⟩}
--	--

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

<code>\prop_map_break: *</code>	<code>\prop_map_break:</code>
---------------------------------	-------------------------------

Used to terminate a `\prop_map...` function before all entries in the ⟨*property list*⟩ have been processed. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario will lead low level TeX errors.

<code>\prop_map_break:n *</code>	<code>\prop_map_break:n</code> {⟨ <i>tokens</i> ⟩}
----------------------------------	--

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.

119 Viewing property lists

<code>\prop_show:N</code> <code>\prop_show:c</code>	<code>\prop_show:N <property list></code>
--	---

Displays the entries in the *<property list>* in the terminal.

120 Experimental property list functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

<code>\prop_gpop:NnTF</code> <code>\prop_gpop:cnTF</code>	<code>\prop_gpop:NnTF <property list> {<key>}</code> <code><token list variable> {<true code>} {<false code>}</code>
--	---

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

<code>\prop_map_tokens:Nn *</code> <code>\prop_map_tokens:cn *</code>	<code>\prop_map_tokens:Nn <property list> {<code>}</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.

<code>\prop_get:Nn *</code> <code>\prop_get:cn *</code>	<code>\prop_get:Nn <property list> {<key>}</code>
--	---

Expands to the *<value>* corresponding to the *<key>* in the *<property list>*. If the *<key>* is missing, this has an empty expansion.

T_EXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`.

121 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_`.

122 Creating and initialising boxes

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

`\box_new:N <box>`

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

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

`\box_clear:N <box>`

Clears the content of the `<box>` by setting the box equal to `\c_void_box` within the current `TeX` group level.

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

`\box_gclear:N <box>`

Clears the content of the `<box>` 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 `<box>` is not defined, globally creates it. If the `<box>` is defined, clears the content of the `<box>` by setting the box equal to `\c_void_box` within the current `TeX` group level.

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

`\box_gclear_new:N <box>`

If the `<box>` is not defined, globally creates it. If the `<box>` is defined, clears the content of the `<box>` 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 `<box1>` equal to that of `<box2>`. This assignment is restricted to the

current $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 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 $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 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.

123 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.

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$ hackers note: This is the $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 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 $\langle box \rangle$.

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

124 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.

125 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`).

<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.

<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 `<box>` is a vertical box.

126 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* settable by the programmer but is instead varied by \TeX .

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

127 Constant boxes

`\c_empty_box` This is a permanently empty box, which is neither set as horizontal nor vertical.

128 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 \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

129 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.

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

130 Horizontal mode boxes

`\hbox:n` `\hbox:n {⟨contents⟩}`

Typesets the $\langle contents \rangle$ 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 $\langle contents \rangle$ into a horizontal box of width $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.

`\hbox_to_zero:n` `\hbox_to_zero:n {⟨contents⟩}`

Typesets the $\langle contents \rangle$ 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 $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. The assignment is local.

`\hbox_gset:Nn`
`\hbox_gset:cn` `\hbox_gset:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. The assignment is global.

`\hbox_set_to_wd:Nnn`
`\hbox_set_to_wd:cnn` `\hbox_set_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. The assignment is local.

`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn` `\hbox_gset_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. 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`.

131 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:nn {<dimexpr>} {<contents>}`

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`.

132 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 if $\langle box \rangle$ 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 if $\langle box \rangle$ 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).

133 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.

134 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`).

135 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.

136 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.

<code>\ior_gto:NN</code>	<code>\ior_gto:NN</code> $\langle stream \rangle$ $\langle token\ list\ variable \rangle$
--------------------------	---

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.

<code>\ior_str_to:NN</code>	<code>\ior_str_to:NN</code> $\langle stream \rangle$ $\langle token\ list\ variable \rangle$
-----------------------------	--

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.

<code>\ior_str_gto:NN</code>	<code>\ior_str_gto:NN</code> $\langle stream \rangle$ $\langle token\ list\ variable \rangle$
------------------------------	---

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.

<code>\ior_if_eof_p:N *</code>	<code>\ior_if_eof_p:N</code> $\langle stream \rangle$
<code>\ior_if_eof:NTF *</code>	<code>\ior_if_eof:N</code> $\langle stream \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

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.

137 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.

138 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.

139 Contextual information for messages

<code>\msg_line_context: *</code>	<code>\msg_line_context:</code>
-----------------------------------	---------------------------------

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text `on line`.

`\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 give, 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.

140 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 *code* may be any arbitrary material. The *code* 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 an

no errors if the number of arguments supplied here does not match the number in the definition of the message (although of course the sense of the message may be impaired).

<code>\msg_fatal:nnxxxx</code>	
<code>\msg_fatal:nnxxx</code>	
<code>\msg_fatal:nnxx</code>	
<code>\msg_fatal:nnx</code>	
<code>\msg_fatal:nn</code>	<code>\msg_fatal:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. After issuing a fatal error the T_EX run will halt.

<code>\msg_critical:nnxxxx</code>	
<code>\msg_critical:nnxxx</code>	
<code>\msg_critical:nnxx</code>	
<code>\msg_critical:nnx</code>	
<code>\msg_critical:nn</code>	<code>\msg_critical:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` 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.

<code>\msg_error:nnxxxx</code>	
<code>\msg_error:nnxxx</code>	
<code>\msg_error:nnxx</code>	
<code>\msg_error:nnx</code>	
<code>\msg_error:nn</code>	<code>\msg_error:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue.

<code>\msg_warning:nnxxxx</code>	
<code>\msg_warning:nnxxx</code>	
<code>\msg_warning:nnxx</code>	
<code>\msg_warning:nnx</code>	
<code>\msg_warning:nn</code>	<code>\msg_warning:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Issues `<module>` warning `<message>`, passing `<arg one>` to `<arg four>` 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 $\langle module \rangle$ 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.

<pre>\msg_log:nnxxxx \msg_log:nnxxx \msg_log:nnxx \msg_log:nnx \msg_log:nn</pre>	<pre>\msg_log: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 $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file: the output is briefer than `\msg_info:nnxxxx`.

<pre>\msg_none:nnxxxx \msg_none:nnxxx \msg_none:nnxx \msg_none:nnx \msg_none:nn</pre>	<pre>\msg_none:nnxxxx {$\langle module \rangle$} {$\langle message \rangle$} {$\langle arg one \rangle$} {$\langle arg two \rangle$} {$\langle arg three \rangle$} {$\langle arg four \rangle$}</pre>
---	---

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

141 Redirecting messages

<pre>\msg_redirect_class:nn</pre>	<pre>\msg_redirect_class:nn {$\langle class one \rangle$} {$\langle class two \rangle$}</pre>
-----------------------------------	---

Changes the behaviour of messages of $\langle class one \rangle$ so that they are processed using the code for those of $\langle class two \rangle$. Multiple redirections are possible. Redirection to a missing class or infinite loops will raise errors when the messages are used, rather than at the point of redirection.

<pre>\msg_redirect_module:nnn</pre>	<pre>\msg_redirect_module:nnn {$\langle module \rangle$} {$\langle class one \rangle$} {$\langle class two \rangle$}</pre>
-------------------------------------	---

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 {<module>} {<message>} {<class>}</code>
-------------------------------------	---

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* 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 }
```

142 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>*</code>
<code>\msg_two_newlines:</code>	<code>*</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 {<first line>} {<text>} {<extra text>}</code>
---------------------------------	--

Interrupts the T_EX run, issuing a formatted message comprising *<first line>* and *<text>* laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!.....
```

where the *<text>* will be wrapped to fit within the current line length. The user may then request more information, at which stage the *<extra text>* will be shown in the terminal in the format

```
|,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
| <extra text>
|.....
```

where the $\langle extra\ text \rangle$ will be wrapped to fit within the current line length.

<code>\msg_log:x</code>	<code>\msg_log:x {$\langle text \rangle$}</code>
-------------------------	---

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.

<code>\msg_term:x</code>	<code>\msg_term:x {$\langle text \rangle$}</code>
--------------------------	--

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.

143 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.

<code>\msg_kernel_new:nnnn</code>	<code>\msg_kernel_new:nnnn {$\langle module \rangle$} {$\langle message \rangle$} {$\langle text \rangle$} {$\langle more\ text \rangle$}</code>
<code>\msg_kernel_new:nnn</code>	

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 $\langle more\ text \rangle$ `\` can be used to start a new line. An error will be raised if the $\langle message \rangle$ already exists.

<code>\msg_kernel_set:nnnn</code>	<code>\msg_kernel_set:nnnn {$\langle module \rangle$} {$\langle message \rangle$} {$\langle text \rangle$} {$\langle more\ text \rangle$}</code>
<code>\msg_kernel_set:nnn</code>	

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.

<pre> \msg_kernel_fatal:nnxxxx \msg_kernel_fatal:nnxxx \msg_kernel_fatal:nnxx \msg_kernel_fatal:nnx \msg_kernel_fatal:nn </pre>	<pre> \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} </pre>
---	---

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.

<pre> \msg_kernel_error:nnxxxx \msg_kernel_error:nnxxx \msg_kernel_error:nnxx \msg_kernel_error:nnx \msg_kernel_error:nn </pre>	<pre> \msg_kernel_error: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 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.

<pre> \msg_kernel_warning:nnxxxx \msg_kernel_warning:nnxxx \msg_kernel_warning:nnxx \msg_kernel_warning:nnx \msg_kernel_warning:nn </pre>	<pre> \msg_kernel_warning: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 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 {\langle module \rangle} {\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle} </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.

144 Expandable errors

In a few places, the L^AT_EX3 kernel needs to produce errors in an expansion only context. This must be handled very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable.

```
\msg_expandable_error:n \msg_expandable_error:n {<error message>}
```

Issues an “Undefined error” message from T_EX itself, and prints the *<error message>*. The *<error message>* must be short: it is cropped at the end of one line.

T_EXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to T_EX’s prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

Part XVIII

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 }
{
  \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 146, 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`.

145 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 *<key>*, which when used may be supplied with a *<value>*. All key *definitions* are local.

`.bool_set:N` `<key> .bool_set:N = <boolean>`

Defines *<key>* to set *<boolean>* to *<value>* (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 *<boolean>* will be assigned locally.

`.bool_gset:N` `<key> .bool_gset:N = <boolean>`

Defines *<key>* to set *<boolean>* to *<value>* (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 *<boolean>* will be assigned globally.

`.bool_set_inverse:N` `<key> .bool_set_inverse:N = <boolean>`

Defines *<key>* to set *<boolean>* to the logical inverse of *<value>* (which must be either **true** or **false**). If the *<boolean>* does not exist, it will be created at the point that the key is set up. The *<boolean>* will be assigned locally.

This property is experimental.

`.bool_gset_inverse:N` `<key> .bool_gset_inverse:N = <boolean>`

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either `true` or `false`). If the $\langle boolean \rangle$ does not exist, it will be created at the point that the key is set up. The $\langle boolean \rangle$ will be assigned globally.

This property is experimental.

```
.choice:  $\langle key \rangle$  .choice:
```

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 147.

```
.choices:nn  $\langle key \rangle$  .choices:nn  $\langle choices \rangle$   $\langle code \rangle$ 
```

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_t1` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 0). Choices are discussed in detail in section 147.

This property is experimental.

```
.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_t1` 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 147.

```
.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 floating point number). 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 floating point number). 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 [147](#).

```


.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 be an 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>	$\langle key \rangle$	<code>.int_gset:N = $\langle integer \rangle$</code>
<code>.int_gset:c</code>		

Defines $\langle key \rangle$ to set $\langle integer \rangle$ to $\langle value \rangle$ (which must be an 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>	$\langle key \rangle$	<code>.meta:n = {\mathit{keyval list}}</code>
<code>.meta:x</code>		

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>.multichoice:</code>	$\langle key \rangle$	<code>.multichoice:</code>
----------------------------	-----------------------	----------------------------

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 147.

This property is experimental.

<code>.multichoice:nn</code>	$\langle key \rangle$	<code>.multichoice:nn</code>	$\langle choices \rangle$	$\langle code \rangle$
------------------------------	-----------------------	------------------------------	---------------------------	------------------------

Sets $\langle key \rangle$ to act as a multiple choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 0). Choices are discussed in detail in section 147.

This property is experimental.

<code>.skip_set:N</code>	$\langle key \rangle$	<code>.skip_set:N = $\langle skip \rangle$</code>
<code>.skip_set:c</code>		

Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must be 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>	$\langle key \rangle$	<code>.skip_gset:N = $\langle skip \rangle$</code>
<code>.skip_gset:c</code>		

Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must be 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>	$\langle key \rangle$	<code>.tl_set:N = $\langle token list variable \rangle$</code>
<code>.tl_set:c</code>		

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>	$\langle key \rangle$ <code>.tl_gset:N</code> = $\langle token\ list\ variable \rangle$
--	---

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>	$\langle key \rangle$ <code>.tl_set_x:N</code> = $\langle token\ list\ variable \rangle$
--	--

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 $\langle token\ list\ variable \rangle$ 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> = $\langle token\ list\ variable \rangle$
--	---

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 $\langle token\ list\ variable \rangle$ 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.

146 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:nm { 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.

147 Choice and multiple choice keys

The `l3keys` system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { 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. The same approach is also implemented by the *experimental* property `.choices:nn`. This combines the functionality of `.choice_code:n` and `.generate_choices:n` into one property:

```
\keys_define:nn { module }
{
  key .choices:nn =
    { choice-a, choice-b, choice-c }
    {
      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.
    }
}
```

Note that the `.choices:nn` property should *not* be mixed with use of `.generate_choices:n`.

<code>\l_keys_choice_int</code> <code>\l_keys_choice_tl</code>

Inside the code block for a choice generated using `.generate_choice:` or `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 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:,
  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).

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoice:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```

\keys_define:nn { module }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      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.
    }
}

```

and

```

\keys_define:nn { module }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

are valid. The `.multichoices:nn` property causes `\l_keys_choice_tl` and `\l_keys_choice_int` to be set in exactly the same way as described for `.choices:nn`.

When multiple choice keys are set, the value is treated as a comma-separated list:

```

\keys_set:nn { module }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}

```

Each choice will be applied in turn, with the usual handling of unknown values.

148 Setting keys

<pre> \keys_set:nn \keys_set:nV \keys_set:nv \keys_set:no </pre>	<pre> \keys_set:nn {<module>} {<keyval list>} </pre>
--	--

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.

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`.

`\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`.

`\l_keys_value_tl` When processing an unknown key, the value of the key is available as `\l_keys_value_tl`. Note that this will be empty if no value was given for the key.

149 Setting known keys only

The functionality described in this section is experimental and may be altered or removed, depending on feedback.

<pre>\keys_set_known:nnN \keys_set_known:nVN \keys_set_known:nvN \keys_set_known:noN</pre>	<code>\keys_set_known:nn {<module>} {<keyval list>} <clist></code>
--	--

Parses the *<keyval list>*, and sets those keys which are defined for *<module>*. Any keys which are unknown are not processed further by the parser. The key–value pairs for each *unknown* key name will be stored in the *<clist>*.

150 Utility functions for keys

<pre>\keys_if_exist_p:nn *</pre>	<code>\keys_if_exist_p:nn <module> <key></code>
<pre>\keys_if_exist:nnTF *</pre>	<code>\keys_if_exist:nnTF <module> <key></code>
	<code>{<true code>} {<false code>}</code>

Tests if the $\langle key \rangle$ exists for $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle$.

$\backslash\text{keys_if_choice_exist_p:nn} \star$ $\backslash\text{keys_if_choice_exist:nnTF} \star$	$\backslash\text{keys_if_exist_p:nnn} \langle module \rangle \langle key \rangle \langle choice \rangle$ $\backslash\text{keys_if_exist:nnnTF} \langle module \rangle \langle key \rangle \langle choice \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

$\backslash\text{keys_show:nn}$	$\backslash\text{keys_show:nn} \{\langle module \rangle\} \{\langle key \rangle\}$
----------------------------------	---

Shows the function which is used to actually implement a $\langle key \rangle$ for a $\langle module \rangle$.

151 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in especial circumstances you may wish to use a lower-level approach. 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 processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (*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.

```
\keyval_parse:NNn \keyval_parse:NNn <function1> <function2>
{<key-value list>}
```

Parses the $\langle key\text{-}value\ list\rangle$ into a series of $\langle keys\rangle$ and associated $\langle values\rangle$, or keys alone (if no $\langle value\rangle$ was given). $\langle function1\rangle$ should take one argument, while $\langle function2\rangle$ should absorb two arguments. After `\keyval_parse:NNn` has parsed the $\langle key\text{-}value\ list\rangle$, $\langle function1\rangle$ will be used to process keys given with no value and $\langle function2\rangle$ will be used to process keys given with a value. The order of the $\langle keys\rangle$ in the $\langle key\text{-}value\ list\rangle$ will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

will be converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all).

Part XIX

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 $\text{T}_{\text{E}}\text{X}$ will attempt to locate files using both the operating system path and entries in the $\text{T}_{\text{E}}\text{X}$ file database (most $\text{T}_{\text{E}}\text{X}$ systems use such a database). Thus the “current path” for $\text{T}_{\text{E}}\text{X}$ is somewhat broader than that for other programs.

152 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 {<file name>} {<true code>} {<false code>}`
Searches for `<file name>` using the current T_EX search path and the additional paths controlled by `\file_path_include:n`.

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_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 $\langle path \rangle$ from the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

`\file_list:` `\file_list:`

This function will list all files loaded using `\file_input:n` in the log file.

153 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 XX

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.

154 Floating-point variables

<code>\fp_new:N</code> <code>\fp_new:c</code>
--

`\fp_new:N` *<floating point variable>*

Creates a new *<floating point variable>* or raises an error if the name is already taken. The declaration global. The *<floating point>* will initially be set to `+0.000000000e0` (the zero floating point).

<code>\fp_const:Nn</code> <code>\fp_const:cn</code>
--

`\fp_const:Nn` *<floating point variable>* {*<value>*}

Creates a new constant *<floating point variable>* or raises an error if the name is already taken. The value of the *<floating point variable>* will be set globally to the *<value>*.

<code>\fp_set_eq:NN</code> <code>\fp_set_eq:cN</code> <code>\fp_set_eq:Nc</code> <code>\fp_set_eq:cc</code>
--

`\fp_set_eq:NN` *<fp var1>* *<fp var2>*

Sets the value of *<floating point variable1>* equal to that of *<floating point variable2>*. This assignment is restricted to the current \TeX group level.

<code>\fp_gset_eq:NN</code> <code>\fp_gset_eq:cN</code> <code>\fp_gset_eq:Nc</code> <code>\fp_gset_eq:cc</code>
--

`\fp_gset_eq:NN` *<fp var1>* *<fp var2>*

Sets the value of $\langle floating\ point\ variable1 \rangle$ equal to that of $\langle floating\ point\ variable2 \rangle$. 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 $\langle floating\ point\ variable \rangle$`

Sets the $\langle floating\ point\ variable \rangle$ to +0.000000000e0 within the current scope.

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

`\fp_gzero:N $\langle floating\ point\ variable \rangle$`

Sets the $\langle floating\ point\ variable \rangle$ to +0.000000000e0 globally.

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

`\fp_set:Nn $\langle floating\ point\ variable \rangle$ { $\langle value \rangle$ }`

Sets the $\langle floating\ point\ variable \rangle$ variable to $\langle value \rangle$ within the scope of the current T_EX group.

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

`\fp_gset:Nn $\langle floating\ point\ variable \rangle$ { $\langle value \rangle$ }`

Sets the $\langle floating\ point\ variable \rangle$ variable to $\langle value \rangle$ globally.

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

`\fp_set_from_dim:Nn $\langle floating\ point\ variable \rangle$ { $\langle dimexpr \rangle$ }`

Sets the $\langle floating\ point\ variable \rangle$ to the distance represented by the $\langle dimension\ expression \rangle$ in the units points. This means that distances given in other units are first converted to points before being assigned to the $\langle floating\ point\ variable \rangle$. The assignment is local.

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

`\fp_gset_from_dim:Nn $\langle floating\ point\ variable \rangle$ { $\langle dimexpr \rangle$ }`

Sets the $\langle floating\ point\ variable \rangle$ to the distance represented by the $\langle dimension\ expression \rangle$ in the units points. This means that distances given in other units are first converted to points before being assigned to the $\langle floating\ point\ variable \rangle$. The assignment is global.

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

`\fp_use:N $\langle floating\ point\ variable \rangle$`

Inserts the value of the $\langle floating\ point\ variable \rangle$ 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.

155 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.

156 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> { <i><target></i> }
--	--

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> { <i><target></i> }
--	---

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> { <i><target></i> }
--	---

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> { <i><target></i> }
--	--

Rounds the *<floating point variable>* to the *<target>* number of decimal places (an integer expression). The rounding is carried out globally.

157 Floating-point conditionals

<code>\fp_if_undefined_p:N</code> \star <code>\fp_if_undefined:NTF</code> \star	<code>\fp_if_undefined_p:N</code> <i><fixed-point></i> <code>\fp_if_undefined:NTF</code> <i><fixed-point></i> { <i><true code></i> } { <i><false code></i> }
--	--

Tests if $\langle floating\ point \rangle$ is undefined (*i.e.* equal to the special `\c_undefined_fp` variable).

<code>\fp_if_zero:N *</code>	<code>\fp_if_zero_p:N</code> $\langle fixed-point \rangle$
	<code>\fp_if_zero:NTF</code> $\langle fixed-point \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle floating\ point \rangle$ is equal to zero (*i.e.* equal to the special `\c_zero_fp` variable).

	<code>\fp_compare:nNnTF</code>
<code>\fp_compare:nNnTF</code>	$\{\langle floating\ point_1 \rangle\}$ $\langle relation \rangle$ $\{\langle floating\ point_2 \rangle\}$
	$\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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	<

The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

	<code>\fp_compare:nTF</code>
<code>\fp_compare:nTF</code>	$\{\langle floating\ point_1 \rangle\}$ $\langle relation \rangle$ $\langle floating\ point_2 \rangle$ $\}$
	$\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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	!=

The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

158 Unary floating-point operations

The unary operations alter the value stored within an `fp` variable.

<code>\fp_abs:N</code>	
<code>\fp_abs:c</code>	<code>\fp_abs:N</code> $\langle floating\ point\ variable \rangle$

Converts the $\langle floating\ point\ variable \rangle$ to its absolute value, assigning the result within

the current $\text{T}_{\text{E}}\text{X}$ group.

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

`\fp_gabs:N` *floating point variable*

Converts the *floating point variable* to its absolute value, assigning the result globally.

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

`\fp_neg:N` *floating point variable*

Reverse the sign of the *floating point variable*, assigning the result within the current $\text{T}_{\text{E}}\text{X}$ group.

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

`\fp_gneg:N` *floating point variable*

Reverse the sign of the *floating point variable*, assigning the result globally.

159 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).

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

`\fp_add:Nn` *floating point* {*value*}

Adds the *value* to the *floating point*, making the assignment within the current $\text{T}_{\text{E}}\text{X}$ group level.

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

`\fp_gadd:Nn` *floating point* {*value*}

Adds the *value* to the *floating point*, making the assignment globally.

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

`\fp_sub:Nn` *floating point* {*value*}

Subtracts the *value* from the *floating point*, making the assignment within the current

TeX group level.

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

`\fp_gsub:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Subtracts the $\langle value \rangle$ from the $\langle floating\ point \rangle$, making the assignment globally.

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

`\fp_mul:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Multiplies the $\langle floating\ point \rangle$ by the $\langle value \rangle$, making the assignment within the current TeX group level.

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

`\fp_gmul:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Multiplies the $\langle floating\ point \rangle$ by the $\langle value \rangle$, making the assignment globally.

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

`\fp_div:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Divides the $\langle floating\ point \rangle$ by the $\langle value \rangle$, making the assignment within the current TeX group level. If the $\langle value \rangle$ is zero, the $\langle floating\ point \rangle$ will be set to `\c_undefined_fp`. The assignment is local.

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

`\fp_gdiv:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Divides the $\langle floating\ point \rangle$ by the $\langle value \rangle$, making the assignment globally. If the $\langle value \rangle$ is zero, the $\langle floating\ point \rangle$ will be set to `\c_undefined_fp`. The assignment is global.

160 Floating-point power operations

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

`\fp_pow:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Raises the $\langle floating\ point \rangle$ to the given $\langle value \rangle$. If the $\langle floating\ point \rangle$ 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 `\c_undefined_fp`. The assignment is local.

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

`\fp_gpow:Nn` $\langle floating\ point \rangle$ $\{\langle value \rangle\}$
Raises the $\langle floating\ point \rangle$ to the given $\langle value \rangle$. If the $\langle floating\ point \rangle$ 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.

161 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.

162 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.

163 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.

164 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 XXI

The `l3luatex` package Lua \TeX -specific functions

165 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 { $\langle \textit{token list} \rangle$ }`

The $\langle \textit{token list} \rangle$ 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 { $\langle \textit{token list} \rangle$ }`

The $\langle \textit{token list} \rangle$ 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”.

166 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 XXII

Implementation

167 Bootstrap code

```
1 <*initex | package>
```

167.1 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>

```

167.2 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 6.)

\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 ??.)

167.3 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{9999/99/99}%
113   \def\ExplFileDescription{#2}%
114   \def\ExplFileName{[unknown name]}%
115   \def\ExplFileVersion{999}%
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>

```

167.4 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

```

167.5 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε -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

```

167.6 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_other:n}\endcsname{58}%
277     \expandafter\noexpand
278     \csname\detokenize{char_set_catcode_math_subscript:n}\endcsname{95}%
279 }

```

(End definition for `\ExplSyntaxNamesOn` and `\ExplSyntaxNamesOff`. These functions are documented on page 6.)

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 6.)

`\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>

```

168 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 related to PDF output.

```

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_YTeX-specific primitives. Note that X_YTeX’s `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`.

```

754 \name_primitive:NN \XeTeXversion \xetex_XeTeXversion:D

```

Primitives from Lua_YTeX.

```

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>

```

169 l3basics implementation

```

776 \<*initex | package>
777 \<*package>
778 \ProvidesExplPackage
779   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
780 \package_check_loaded_expl:
781 \</package>

```

169.1 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.

```

\if_true:
\if_false:
  \or:
\else:
  \fi:
\reverse_if:N
  \if:w
\if_bool:N
\if_predicate:w
\if_charcode:w
\if_catcode:w

```

```

784 \cs_set_eq:NwN \if_true: \tex_iftrue:D
785 \cs_set_eq:NwN \if_false: \tex_iffalse:D
786 \cs_set_eq:NwN \or: \tex_or:D
787 \cs_set_eq:NwN \else: \tex_else:D
788 \cs_set_eq:NwN \fi: \tex_fi:D
789 \cs_set_eq:NwN \reverse_if:N \etex_unless:D
790 \cs_set_eq:NwN \if:w \tex_if:D
791 \cs_set_eq:NwN \if_bool:N \tex_ifodd:D
792 \cs_set_eq:NwN \if_predicate:w \tex_ifodd:D
793 \cs_set_eq:NwN \if_charcode:w \tex_if:D
794 \cs_set_eq:NwN \if_catcode:w \tex_ifcat:D

```

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 25.)

`\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 34.)

`\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 77.)

`\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 26.)

`\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`
`\cs_meaning:c`
`\cs_show:c`

A small number of variants by hand.

```
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.)

169.2 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.)

169.3 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 }
```

169.4 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 { }

```

169.5 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~

```

169.6 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`
`\cs_get_function_signature:N`

Now returning the name is trivial: just discard the last two arguments. Similar for 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 }

```

169.7 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`
`\cs_if_exist_p:c`
`\cs_if_exist:NTF`
`\cs_if_exist:cTF`

Two versions for checking existence. For the N form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as TeX will only ever skip input in case the token tested against is `\scan_stop:`.

```

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     \else:
1151       \prg_return_false:
1152     \fi:
1153   }
1154   { \prg_return_true: }
1155 }

```

(End definition for `\cs_if_free:N` and `\cs_if_free:c`. These functions are documented on page 23.)

169.8 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 76.)

`\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 76.)

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 154.)

`\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 165.)

`\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 158.)

`\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 26.)

`\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 26.)

169.9 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:.`)

169.10 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.)

169.11 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.)

169.12 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`.)

169.13 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.)

169.14 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:NNTF }
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:NNTF }
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:NNTF }
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 ??.)

169.15 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 ??.)

169.16 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.)

169.17 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 8.)

169.18 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_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.)

169.19 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>

```

170 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 34.)*
`\exp_not:n`

170.1 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 170.3. In section 170.2 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.
<argument> \LaTeX3 error:
                               Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
1571 \cs_new:Npn \exp_eval_error_msg:w #1 \tex_the:D #2
1572 {
1573     \fi:
1574     \fi:
1575     \msg_expandable_error:n { Erroneous ~ variable ~ #2 used! }
1576     \c_zero
1577 }
```

(End definition for `\exp_eval_register:N` and `\exp_eval_register:c`. These functions are documented on page 35.)

170.2 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!

```
1578 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
1579 \cs_new:Npn \exp_args:NNo #1#2#3
1580 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
1581 \cs_new:Npn \exp_args:NNNo #1#2#3#4
1582 { \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 32.)

`\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.

```
1583 \cs_new:Npn \exp_args:cc #1#2
1584 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
1585 \cs_new:Npn \exp_args:NNc #1#2#3
1586 { \exp_after:wN #1 \exp_after:wN #2 \cs:w #3 \cs_end: }
1587 \cs_new:Npn \exp_args:Ncc #1#2#3
1588 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
1589 \cs_new:Npn \exp_args:Nccc #1#2#3#4
1590 {
```

```

1591 \exp_after:wN #1
1592 \cs:w #2 \exp_after:wN \cs_end:
1593 \cs:w #3 \exp_after:wN \cs_end:
1594 \cs:w #4 \cs_end:
1595 }

```

(End definition for `\exp_args:cc` and others. These functions are documented on page 32.)

`\exp_args:Nf`
`\exp_args:Nv`
`\exp_args:Nf`
`\exp_args:Nx`

```

1596 \cs_new:Npn \exp_args:Nf #1#2
1597 { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -'0 #2 } }
1598 \cs_new:Npn \exp_args:Nv #1#2
1599 {
1600   \exp_after:wN #1 \exp_after:wN
1601   { \tex_romannumeral:D \exp_eval_register:c {#2} }
1602 }
1603 \cs_new:Npn \exp_args:Nv #1#2
1604 {
1605   \exp_after:wN #1 \exp_after:wN
1606   { \tex_romannumeral:D \exp_eval_register:N #2 }
1607 }

```

(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:NVV`
`\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.

```

1608 \cs_new:Npn \exp_args:NNf #1#2#3
1609 {
1610   \exp_after:wN #1
1611   \exp_after:wN #2
1612   \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1613 }
1614 \cs_new:Npn \exp_args:NNv #1#2#3
1615 {
1616   \exp_after:wN #1
1617   \exp_after:wN #2
1618   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#3} }
1619 }
1620 \cs_new:Npn \exp_args:NNV #1#2#3
1621 {
1622   \exp_after:wN #1
1623   \exp_after:wN #2
1624   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
1625 }
1626 \cs_new:Npn \exp_args:Nco #1#2#3
1627 {
1628   \exp_after:wN #1

```

```

1629     \cs:w #2 \exp_after:wN \cs_end:
1630     \exp_after:wN {#3}
1631   }
1632   \cs_new:Npn \exp_args:Ncf #1#2#3
1633   {
1634     \exp_after:wN #1
1635     \cs:w #2 \exp_after:wN \cs_end:
1636     \exp_after:wN { \tex_romannumeral:D -'0 #3 }
1637   }
1638   \cs_new_nopar:Npn \exp_args:NVV #1#2#3
1639   {
1640     \exp_after:wN #1
1641     \exp_after:wN { \tex_romannumeral:D \exp_after:wN
1642       \exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
1643     \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
1644   }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 31.)

`\exp_args:Ncco`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:NNNV`

A few more that we can hand-tune.

```

1645   \cs_new:Npn \exp_args:NNNV #1#2#3#4
1646   {
1647     \exp_after:wN #1
1648     \exp_after:wN #2
1649     \exp_after:wN #3
1650     \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #4 }
1651   }
1652   \cs_new:Npn \exp_args:NcNc #1#2#3#4
1653   {
1654     \exp_after:wN #1
1655     \cs:w #2 \exp_after:wN \cs_end:
1656     \exp_after:wN #3
1657     \cs:w #4 \cs_end:
1658   }
1659   \cs_new:Npn \exp_args:NcNo #1#2#3#4
1660   {
1661     \exp_after:wN #1
1662     \cs:w #2 \exp_after:wN \cs_end:
1663     \exp_after:wN #3
1664     \exp_after:wN {#4}
1665   }
1666   \cs_new:Npn \exp_args:Ncco #1#2#3#4
1667   {
1668     \exp_after:wN #1
1669     \cs:w #2 \exp_after:wN \cs_end:
1670     \cs:w #3 \exp_after:wN \cs_end:
1671     \exp_after:wN {#4}
1672   }

```

(End definition for `\exp_args:Ncco` and others. These functions are documented on page 32.)

170.3 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`

```
1673 \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.

```
1674 \cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
1675 \cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
1676 \cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
1677 \cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
1678 \cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
1679 \cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
1680 \cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
1681 \cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
1682 \cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
1683 \cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
1684 \cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
1685 \cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
1686 \cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
1687 \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:Noox`

`\exp_args:Nnnx`

`\exp_args:NNnx`

`\exp_args:NNoo`

`\exp_args:NNox`

```
1688 \cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
1689 \cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
1690 \cs_new_nopar:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
1691 \cs_new_nopar:Npn \exp_args:Nnno { \::n \::n \::o \::: }
1692 \cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
1693 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
1694 \cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
1695 \cs_new_protected_nopar:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
1696 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
1697 \cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
1698 \cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
1699 \cs_new_protected_nopar:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(End definition for `\exp_args:Nccx` and others. These functions are documented on page 33.)

170.4 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.

```

1700 \cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
1701 \cs_new:Npn \::f_unbraced \::: #1#2
1702 {
1703   \exp_after:wN \exp_arg_last_unbraced:nn
1704   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1705 }
1706 \cs_new:Npn \::o_unbraced \::: #1#2
1707 { \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
1708 \cs_new:Npn \::V_unbraced \::: #1#2
1709 {
1710   \exp_after:wN \exp_arg_last_unbraced:nn
1711   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #2 } {#1}
1712 }
1713 \cs_new:Npn \::v_unbraced \::: #1#2
1714 {
1715   \exp_after:wN \exp_arg_last_unbraced:nn
1716   \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#2} } {#1}
1717 }
```

(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.

```

1718 \cs_new:Npn \exp_last_unbraced:NV #1#2
1719 { \exp_after:wN #1 \tex_romannumeral:D \exp_eval_register:N #2 }
1720 \cs_new:Npn \exp_last_unbraced:Nv #1#2
1721 { \exp_after:wN #1 \tex_romannumeral:D \exp_eval_register:c {#2} }
1722 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
1723 \cs_new:Npn \exp_last_unbraced:Nf #1#2
1724 { \exp_after:wN #1 \tex_romannumeral:D -'0 #2 }
1725 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
1726 {
1727   \exp_after:wN #1
1728   \cs:w #2 \exp_after:wN \cs_end:
1729   \tex_romannumeral:D \exp_eval_register:N #3
1730 }
1731 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
1732 {
1733   \exp_after:wN #1
1734   \exp_after:wN #2
1735   \tex_romannumeral:D \exp_eval_register:N #3
1736 }
1737 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
1738 { \exp_after:wN #1 \exp_after:wN #2 #3 }
```

```

1739 \cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
1740 \cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
1741 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
1742 {
1743   \exp_after:wN #1
1744   \exp_after:wN #2
1745   \exp_after:wN #3
1746   \tex_romannumeral:D \exp_eval_register:N #4
1747 }
1748 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
1749 { \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 33.)

`\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.

```

1750 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
1751 { \exp_after:wN \exp_last_two_unbraced_aux:nnN \exp_after:wN {#3} {#2} #1 }
1752 \cs_new:Npn \exp_last_two_unbraced_aux:nnN #1#2#3
1753 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo`. This function is documented on page 33.)

170.5 Preventing expansion

```

\exp_not:o
\exp_not:f
\exp_not:V
\exp_not:v
1754 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
1755 \cs_new:Npn \exp_not:f #1
1756 { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -'0 #1 } }
1757 \cs_new:Npn \exp_not:V #1
1758 {
1759   \etex_unexpanded:D \exp_after:wN
1760   { \tex_romannumeral:D \exp_eval_register:N #1 }
1761 }
1762 \cs_new:Npn \exp_not:v #1
1763 {
1764   \etex_unexpanded:D \exp_after:wN
1765   { \tex_romannumeral:D \exp_eval_register:c {#1} }
1766 }

```

(End definition for `\exp_not:o`. This function is documented on page 34.)

`\exp_not:c` A helper function.

```
1767 \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 34.)

170.6 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`.

```
1768 \cs_new_protected:Npn \cs_generate_variant:Nn #1
1769 {
1770   \chk_if_exist_cs:N #1
1771   \cs_split_function:NN #1 \cs_generate_variant_aux:nnNNn
1772   #1
1773 }
```

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.

```
1774 \cs_new:Npn \cs_generate_variant_aux:nnNNn #1#2#3#4#5
1775 { \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.

```
1776 \cs_new:Npn \cs_generate_variant_aux:Nnnw #1#2#3#4 ,
1777 {
1778   \if:w ? #4
1779   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1780   \fi:
1781   \exp_args:NNc \cs_generate_variant_aux:NNn
1782   #1
```

```

1783     { #2 : #4 \use:c { use_none: \cs_generate_variant_aux:N #4 ? } #3 }
1784     {#4}
1785     \cs_generate_variant_aux:Nnnw #1 {#2} {#3}
1786 }

```

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.

```

1787 \cs_new:Npn \cs_generate_variant_aux:NNn #1 #2 #3
1788 {
1789   \cs_if_free:NTF #2
1790   {
1791     \cs_generate_variant_aux:NNpx #1 #2
1792     { \exp_not:c { exp_args:N #3 } \exp_not:N #1 }
1793     \cs_generate_internal_variant:n {#3}
1794   }
1795   {
1796     \iow_log:x
1797     {
1798       Variant~\token_to_str:N #2~%
1799       already~defined;~ not~ changing~ it~on~line~%
1800       \tex_the:D \tex_inputlineno:D
1801     }
1802   }
1803 }

```

The small loop for defining the required number of ns. Break when seeing a ?.

```

1804 \cs_new:Npn \cs_generate_variant_aux:N #1
1805 {
1806   \if:w ? #1
1807   \exp_after:wN \use_none:nn
1808   \fi:
1809   n
1810   \cs_generate_variant_aux:N
1811 }

```

(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.

```

1812 \group_begin:
1813 \tex_lccode:D ‘Z = ‘\d \scan_stop:
1814 \tex_lccode:D ‘? = ‘\ \scan_stop:
1815 \tex_catcode:D ‘P = 12 \scan_stop:
1816 \tex_catcode:D ‘R = 12 \scan_stop:

```

```

1817 \tex_catcode:D '\0 = 12 \scan_stop:
1818 \tex_catcode:D '\T = 12 \scan_stop:
1819 \tex_catcode:D '\E = 12 \scan_stop:
1820 \tex_catcode:D '\C = 12 \scan_stop:
1821 \tex_catcode:D '\Z = 12 \scan_stop:
1822 \tex_lowercase:D
1823 {
1824   \group_end:
1825   \cs_new_nopar:Npn \cs_generate_variant_aux:NNpx #1
1826   {
1827     \exp_after:wN \cs_generate_variant_aux:w
1828     \token_to_meaning:N #1 ? PROTECTEZ \q_stop
1829   }
1830   \cs_new:Npn \cs_generate_variant_aux:w #1 ? PROTECTEZ #2 \q_stop
1831   {
1832     \if_catcode:w a \etex_detokenize:D \exp_after:wN {#1} a
1833     \exp_after:wN \cs_new_protected_nopar:Npx
1834     \else:
1835     \exp_after:wN \cs_new_nopar:Npx
1836     \fi:
1837   }
1838 }

```

(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`

```

1839 \cs_new_protected:Npn \cs_generate_internal_variant:n #1
1840 {
1841   \cs_if_free:cT { \exp_args:N #1 }
1842   {
1843     \cs_new:cpx { \exp_args:N #1 }
1844     { \cs_generate_internal_variant_aux:N #1 : }
1845   }
1846 }

```

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.

```

1847 \cs_new:Npn \cs_generate_internal_variant_aux:N #1
1848 {
1849   \exp_not:c { :: #1 }
1850   \if_meaning:w #1 :
1851   \exp_after:wN \use_none:n
1852   \fi:
1853   \cs_generate_internal_variant_aux:N
1854 }

```

(End definition for `\cs_generate_internal_variant:n`. This function is documented on page 36.)

170.7 Variants which cannot be created earlier

```

\str_if_eq_p:Vn These cannot come earlier as they need \cs_generate_variant:Nn.
\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.)

```

1863 </initex | package>

```

171 l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

```

1864 <*initex | package>

1865 <*package>
1866 \ProvidesExplPackage
1867   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
1868 \package_check_loaded_expl:
1869 </package>

```

171.1 Defining a set of conditional functions

```

\prg_set_conditional:Npnn These are all defined in l3basics, as they are needed “early”. This is just a reminder that
\prg_new_conditional:Npnn that is the case!
\prg_set_protected_conditional:Npnn (End definition for \prg_set_conditional:Npnn and others. These functions are documented on page
\prg_new_protected_conditional:Npnn 39.)
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn

```

171.2 The boolean data type

```

\prg_set_eq_conditional:Nnn Boolean variables have to be initiated when they are created. Other than that there is
\prg_new_eq_conditional:Nnn not much to say here.
\prg_return_true:
\prg_return_false:

```

```

1870 \cs_new_protected_nopar:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
1871 \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
1872 \cs_new_protected_nopar:Npn \bool_set_true:N #1
1873 { \cs_set_eq:NN #1 \c_true_bool }
1874 \cs_new_protected_nopar:Npn \bool_set_false:N #1
1875 { \cs_set_eq:NN #1 \c_false_bool }
1876 \cs_new_protected_nopar:Npn \bool_gset_true:N #1
1877 { \cs_gset_eq:NN #1 \c_true_bool }
1878 \cs_new_protected_nopar:Npn \bool_gset_false:N #1
1879 { \cs_gset_eq:NN #1 \c_false_bool }
1880 \cs_generate_variant:Nn \bool_set_true:N { c }
1881 \cs_generate_variant:Nn \bool_set_false:N { c }
1882 \cs_generate_variant:Nn \bool_gset_true:N { c }
1883 \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
1884 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
1885 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
1886 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
1887 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
1888 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
1889 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
1890 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
1891 \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 40.)

`\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
1892 \cs_new:Npn \bool_set:Nn #1#2
1893 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
1894 \cs_new:Npn \bool_gset:Nn #1#2
1895 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
1896 \cs_generate_variant:Nn \bool_set:Nn { c }
1897 \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:NTF
\bool_if:cTF
1898 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
1899 {
1900   \if_bool:N #1
1901     \prg_return_true:
1902   \else:
1903     \prg_return_false:
1904   \fi:

```

```

1905     }
1906     \cs_generate_variant:Nn \bool_if_p:N { c }
1907     \cs_generate_variant:Nn \bool_if:NT { c }
1908     \cs_generate_variant:Nn \bool_if:NF { c }
1909     \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`

```

1910 \bool_new:N \l_tmpa_bool
1911 \bool_new:N \g_tmpa_bool

```

171.3 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.

```

1912 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
1913 {
1914   \if_predicate:w \bool_if_p:n {#1}
1915   \prg_return_true:
1916   \else:
1917     \prg_return_false:
1918   \fi:
1919 }
1920 \cs_new:Npn \bool_if_p:n #1
1921 {
1922   \group_align_safe_begin:
1923   \bool_get_next:N ( #1 ) S
1924 }
```

The GetNext operation. We make it a switch: If not a `!` or `(`, we assume it is a predicate.

```

1925 \cs_new:Npn \bool_get_next:N #1
1926 {
1927   \use:c
1928   {
1929     bool_
1930     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
1931     :w
1932   }
1933   #1
1934 }
```

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.

```

1935 \cs_new:Npn \bool_get_not_next:N #1
1936 {
```

```

1937     \use:c
1938     {
1939         bool_not_
1940         \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
1941         :w
1942     }
1943     #1
1944 }

```

We need these later on to nullify the unity operation !!.

```

1945 \cs_new:Npn \bool_get_next:NN #1#2 { \bool_get_next:N #2 }
1946 \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.

```

1947 \cs_new:cpn { bool_!:w } #1#2
1948 {
1949     \if_meaning:w ( #2
1950     \exp_after:wN \bool_Not:w
1951     \else:
1952     \if_meaning:w ! #2
1953     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_next:NN
1954     \else:
1955     \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
1956     \fi:
1957     \fi:
1958     #2
1959 }

```

Variant called when already inside a Not. Essentially the opposite of the above.

```

1960 \cs_new:cpn { bool_not_!:w } #1#2
1961 {
1962     \if_meaning:w ( #2
1963     \exp_after:wN \bool_not_Not:w
1964     \else:
1965     \if_meaning:w ! #2
1966     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_not_next:NN
1967     \else:
1968     \exp_after:wN \exp_after:wN \exp_after:wN \bool_not_Not:N
1969     \fi:
1970     \fi:
1971     #2
1972 }

```

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.

```
1973 \cs_new:Npn \bool_Not:w { \exp_after:wN \int_value:w \bool_get_not_next:N }
1974 \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.

```
1975 \cs_new:Npn \bool_Not:N #1
1976 {
1977   \exp_after:wN \bool_p:w
1978   \if_meaning:w #1 \c_true_bool
1979   \c_false_bool
1980   \else:
1981     \c_true_bool
1982   \fi:
1983 }
1984 \cs_new:Npn \bool_not_Not:N #1
1985 {
1986   \exp_after:wN \bool_p:w
1987   \if_meaning:w #1 \c_true_bool
1988   \c_true_bool
1989   \else:
1990     \c_false_bool
1991   \fi:
1992 }
```

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 !(...).

```
1993 \cs_new:cpn { bool_( :w } #1
1994 { \exp_after:wN \bool_cleanup:N \int_value:w \bool_get_next:N }
1995 \cs_new:cpn { bool_not_( :w } #1
1996 { \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.

```
1997 \cs_new:cpn { bool_p:w } { \exp_after:wN \bool_cleanup:N \int_value:w }
1998 \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.

```
1999 \cs_new:Npn \bool_cleanup:N #1
2000 {
2001   \exp_after:wN \bool_choose:NN \exp_after:wN #1
2002   \int_to_roman:w - '\q
2003 }
2004 \cs_new:Npn \bool_not_cleanup:N #1
2005 {
2006   \exp_after:wN \bool_not_choose:NN \exp_after:wN #1
2007   \int_to_roman:w - '\q
2008 }
```

Branching the six way switch. Reversals should be reasonably straightforward.

```
2009 \cs_new_nopar:Npn \bool_choose:NN #1#2 { \use:c { bool_ #2 _ #1 :w } }
2010 \cs_new_nopar:Npn \bool_not_choose:NN #1#2 { \use:c { bool_not_ #2 _ #1 :w } }
```

Continues scanning. Must remove the second & or |.

```
2011 \cs_new_nopar:cpn { bool_&_1:w } & { \bool_get_next:N }
2012 \cs_new_nopar:cpn { bool_|_0:w } | { \bool_get_next:N }
2013 \cs_new_nopar:cpn { bool_not_&_0:w } & { \bool_get_next:N }
2014 \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.

```
2015 \cs_new_nopar:cpn { bool_)_0:w } { \c_false_bool }
2016 \cs_new_nopar:cpn { bool_)_1:w } { \c_true_bool }
2017 \cs_new_nopar:cpn { bool_not_)_0:w } { \c_true_bool }
2018 \cs_new_nopar:cpn { bool_not_)_1:w } { \c_false_bool }
2019 \cs_new_nopar:cpn { bool_S_0:w } { \group_align_safe_end: \c_false_bool }
2020 \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.

```
2021 \cs_new_nopar:cpn { bool_&_0:w } & { \bool_eval_skip_to_end:Nw \c_false_bool }
2022 \cs_new_nopar:cpn { bool_|_1:w } | { \bool_eval_skip_to_end:Nw \c_true_bool }
2023 \cs_new_nopar:cpn { bool_not_&_1:w } &
2024 { \bool_eval_skip_to_end:Nw \c_false_bool }
2025 \cs_new_nopar:cpn { bool_not_|_0:w } |
2026 { \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.

```
2027 %% (
2028 \cs_new:Npn \bool_eval_skip_to_end:Nw #1#2 )
2029 {
2030   \bool_eval_skip_to_end_aux:Nw #1#2 ( % )
2031   \q_no_value \q_stop
2032   {#2}
2033 }
```

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

```
2034 \cs_new:Npn \bool_eval_skip_to_end_aux:Nw #1#2 ( #3#4 \q_stop #5 % )
2035 {
2036   \quark_if_no_value:NTF #3
2037   {#1}
2038   { \bool_eval_skip_to_end_aux_ii:Nw #1 #5 }
2039 }
```

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!

```

2040 \cs_new:Npn \bool_eval_skip_to_end_aux_ii:Nw #1#2 ( #3 )
2041 { % (
2042   \bool_eval_skip_to_end:Nw #1#3 )
2043 }

```

\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.

```

2044 \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.

```

2045 \cs_new:Npn \bool_xor_p:nn #1#2
2046 {
2047   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2048     \c_false_bool
2049     \c_true_bool
2050 }

```

171.4 Logical loops

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The “while”
\bool_while_do:cn version executes the code as long as the boolean is true; the “until” version executes the
\bool_until_do:Nn code as long as the boolean is false.
\bool_until_do:cn

```

2051 \cs_new:Npn \bool_while_do:Nn #1#2
2052 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2053 \cs_new:Npn \bool_until_do:Nn #1#2
2054 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2055 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2056 \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
\bool_do_while:cn after executing the body. Otherwise identical to the above functions.
\bool_do_until:Nn
\bool_do_until:cn

```

2057 \cs_new:Npn \bool_do_while:Nn #1#2
2058 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2059 \cs_new:Npn \bool_do_until:Nn #1#2
2060 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2061 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2062 \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`

```

2063 \cs_new:Npn \bool_while_do:nn #1#2
2064 {
2065   \bool_if:nT {#1}
2066   {
2067     #2
2068     \bool_while_do:nn {#1} {#2}
2069   }
2070 }
2071 \cs_new:Npn \bool_do_while:nn #1#2
2072 {
2073   #2
2074   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2075 }
2076 \cs_new:Npn \bool_until_do:nn #1#2
2077 {
2078   \bool_if:nF {#1}
2079   {
2080     #2
2081     \bool_until_do:nn {#1} {#2}
2082   }
2083 }
2084 \cs_new:Npn \bool_do_until:nn #1#2
2085 {
2086   #2
2087   \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
2088 }

```

171.5 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.

```

2089 \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`

```

2090 \cs_new:Npn \prg_case_int:nnn #1
2091 { \exp_args:Nf \prg_case_int_aux:nnn { \int_eval:n {#1} } }
2092 \cs_new:Npn \prg_case_int_aux:nnn #1 #2 #3
2093 { \prg_case_int_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }

```

```

2094 \cs_new:Npn \prg_case_int_aux:nw #1#2#3
2095 {
2096   \int_compare:nNnTF {#1} = {#2}
2097   { \prg_case_end:nw {#3} }
2098   { \prg_case_int_aux:nw {#1} }
2099 }

```

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

```

```

2100 \cs_new:Npn \prg_case_dim:nnn #1
2101 { \exp_args:Nf \prg_case_dim_aux:nnn { \dim_eval:n {#1} } }
2102 \cs_new:Npn \prg_case_dim_aux:nnn #1 #2 #3
2103 { \prg_case_dim_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
2104 \cs_new:Npn \prg_case_dim_aux:nw #1#2#3
2105 {
2106   \dim_compare:nNnTF {#1} = {#2}
2107   { \prg_case_end:nw {#3} }
2108   { \prg_case_dim_aux:nw {#1} }
2109 }

```

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

```

```

2110 \cs_new:Npn \prg_case_str:nnn #1#2#3
2111 { \prg_case_str_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
2112 \cs_new:Npn \prg_case_str_aux:nw #1#2#3
2113 {
2114   \str_if_eq:nnTF {#1} {#2}
2115   { \prg_case_end:nw {#3} }
2116   { \prg_case_str_aux:nw {#1} }
2117 }
2118 \cs_generate_variant:Nn \prg_case_str:nnn { o }
2119 \cs_new:Npn \prg_case_str:xxn #1#2#3
2120 { \prg_case_str_x_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
2121 \cs_new:Npn \prg_case_str_x_aux:nw #1#2#3
2122 {
2123   \str_if_eq:xxTF {#1} {#2}
2124   { \prg_case_end:nw {#3} }
2125   { \prg_case_str_x_aux:nw {#1} }
2126 }

```

Similar again, but this time with some variants.

```

\prg_case_tl:Nnn
\prg_case_tl:cnm
\prg_case_tl_aux:Nw

```

```

2127 \cs_new:Npn \prg_case_tl:Nnn #1#2#3
2128 { \prg_case_tl_aux:Nw #1 #2 #1 {#3} \q_recursion_stop }
2129 \cs_new:Npn \prg_case_tl_aux:Nw #1#2#3
2130 {
2131   \tl_if_eq:NNTF #1 #2
2132   { \prg_case_end:nw {#3} }
2133   { \prg_case_tl_aux:Nw #1 }
2134 }
2135 \cs_generate_variant:Nn \prg_case_tl:Nnn { c }

```

171.6 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 csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\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 `cnames` are well spent I would say.

```

2136 \cs_new_nopar:Npn \prg_replicate:nn #1
2137 {
2138   \int_to_roman:w
2139   \exp_after:wN \prg_replicate_first_aux:N
2140   \int_value:w \int_eval:w #1 \int_eval_end:
2141   \cs_end:
2142 }
2143 \cs_new_nopar:Npn \prg_replicate_aux:N #1
2144 { \cs:w prg_replicate_#1 :n \prg_replicate_aux:N }
2145 \cs_new_nopar:Npn \prg_replicate_first_aux:N #1
2146 { \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.

```

2147 \cs_new_nopar:Npn \prg_replicate_ :n #1 { \cs_end: }
2148 \cs_new:cpn { prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 }
2149 \cs_new:cpn { prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1 }
2150 \cs_new:cpn { prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1 }
2151 \cs_new:cpn { prg_replicate_3:n } #1
2152 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1 }
2153 \cs_new:cpn { prg_replicate_4:n } #1
2154 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1 }
2155 \cs_new:cpn { prg_replicate_5:n } #1
2156 { \cs_end: {#1#1#1#1#1#1#1#1#1#1 } #1#1#1#1#1 }

```

```
2157 \cs_new:cpn { prg_replicate_6:n } #1  
2158   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }  
2159 \cs_new:cpn { prg_replicate_7:n } #1  
2160   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }  
2161 \cs_new:cpn { prg_replicate_8:n } #1  
2162   { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }  
2163 \cs_new:cpn { prg_replicate_9:n } #1  
2164   { \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...

```

2165 \cs_new:cpn { prg_replicate_first_~:n } #1 { \c_zero \negative_replication }
2166 \cs_new:cpn { prg_replicate_first_0:n } #1 { \c_zero }
2167 \cs_new:cpn { prg_replicate_first_1:n } #1 { \c_zero #1 }
2168 \cs_new:cpn { prg_replicate_first_2:n } #1 { \c_zero #1#1 }
2169 \cs_new:cpn { prg_replicate_first_3:n } #1 { \c_zero #1#1#1 }
2170 \cs_new:cpn { prg_replicate_first_4:n } #1 { \c_zero #1#1#1#1 }
2171 \cs_new:cpn { prg_replicate_first_5:n } #1 { \c_zero #1#1#1#1#1 }
2172 \cs_new:cpn { prg_replicate_first_6:n } #1 { \c_zero #1#1#1#1#1#1 }
2173 \cs_new:cpn { prg_replicate_first_7:n } #1 { \c_zero #1#1#1#1#1#1#1 }
2174 \cs_new:cpn { prg_replicate_first_8:n } #1 { \c_zero #1#1#1#1#1#1#1#1 }
2175 \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` Repeating a function by steps first needs a check on the direction of the steps. After that, `\prg_stepwise_function_incr:nnnN` do the function for the start value then step and loop around. `\prg_stepwise_function_decr:nnnN`

```

2176 \cs_new:Npn \prg_stepwise_function:nnnN #1#2
2177 {
2178   \int_compare:nNnTF {#2} > { 0 }
2179   { \exp_args:Nf \prg_stepwise_function_incr:nnnN }
2180   { \exp_args:Nf \prg_stepwise_function_decr:nnnN }
2181   { \int_eval:n {#1} } {#2}
2182 }
2183 \cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4
2184 {
2185   \int_compare:nNnF {#1} > {#3}
2186   {
2187     #4 {#1}
2188     \exp_args:Nf \prg_stepwise_function_incr:nnnN
2189     { \int_eval:n { #1 + #2 } } {#2} {#3} #4
2190   }
2191 }
2192 \cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4
2193 {
2194   \int_compare:nNnF {#1} < {#3}
2195   {
2196     #4 {#1}
2197     \exp_args:Nf \prg_stepwise_function_decr:nnnN

```

```

2198         { \int_eval:n { #1 + #2 } } {#2} {#3} #4
2199     }
2200 }

```

(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.

```

2201 \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

```

```

2202 \cs_new_protected:Npn \prg_stepwise_inline:nnnn #1#2#3#4
2203 {
2204     \int_gincr:N \g_prg_stepwise_level_int
2205     \cs_gset_nopar:cpn
2206     { \prg_stepwise_ \int_use:N \g_prg_stepwise_level_int :n }
2207     ##1 {#4}
2208     \int_compare:nNnTF {#2} > { 0 }
2209     { \exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
2210     { \exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
2211     { \prg_stepwise_ \int_use:N \g_prg_stepwise_level_int :n }
2212     { \int_eval:n {#1} } {#2} {#3}
2213     \int_gdecr:N \g_prg_stepwise_level_int
2214 }
2215 \cs_new_protected:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4
2216 {
2217     \int_compare:nNnF {#2} > {#4}
2218     {
2219         #1 {#2}
2220         \exp_args:Nnf \prg_stepwise_inline_incr:Nnnn #1
2221         { \int_eval:n { #2 + #3 } } {#3} {#4}
2222     }
2223 }
2224 \cs_new_protected:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4
2225 {
2226     \int_compare:nNnF {#2} < {#4}
2227     {
2228         #1 {#2}
2229         \exp_args:Nnf \prg_stepwise_inline_decr:Nnnn #1
2230         { \int_eval:n { #2 + #3 } } {#3} {#4}
2231     }
2232 }

```

(End definition for `\prg_stepwise_inline:nnnn`. This function is documented on page 46.)

`\prg_stepwise_variable:nnnNn` A wrapper for the above.

```

2233 \cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2#3#4#5
2234 {
2235   \prg_stepwise_inline:nnnn {#1} {#2} {#3}
2236   {
2237     \tl_set:Nn #4 {##1}
2238     #5
2239   }
2240 }

```

(End definition for `\prg_stepwise_variable:nnnNn`. This function is documented on page ??.)

171.7 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.

```

2241 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2242 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:`. These functions are documented on page 46.)

`\mode_if_horizontal_p:` For testing horizontal mode.
`\mode_if_horizontal:TF`

```

2243 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2244 { \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`

```

2245 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2246 { \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 `\scan_align_safe_stop:` things go wrong in align-
`\mode_if_math:TF` ments.

```

2247 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2248 {
2249   \scan_align_safe_stop:
2250   \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi:
2251 }

```

(End definition for `\mode_if_math:`. These functions are documented on page 46.)

171.8 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*... We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```

2252 \cs_new_nopar:Npn \group_align_safe_begin:
2253   { \if_int_compare:w \if_false: { \fi: ‘} = \c_zero \fi: }
2254 \cs_new_nopar:Npn \group_align_safe_end:
2255   { \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 46.)

`\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.

```

2256 \cs_new_nopar:Npn \scan_align_safe_stop:
2257   {
2258     \int_compare:nNnT \etex_currentgrouptype:D = \c_six
2259     {
2260       \int_compare:nNnF \etex_lastnodetype:D = \c_zero
2261       { \int_compare:nNnF \etex_lastnodetype:D = \c_seven { \scan_stop: } }
2262     }
2263   }

```

(End definition for `\scan_align_safe_stop:`. This function is documented on page 47.)

⁵Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

`\prg_variable_get_scope:N` Expandable functions to find the type of a variable, and to return `g` if the variable is global. The trick for `\prg_variable_get_scope:N` is the same as that in `\cs_split_`
`\prg_variable_get_scope_aux:w` function:NN, but it can be simplified as the requirements here are less complex.
`\prg_variable_get_type:N`
`\prg_variable_get_type:w`

```

2264 \group_begin:
2265   \tex_lccode:D '\& = '\g \scan_stop:
2266   \tex_catcode:D '\& = \c_twelve
2267   \tl_to_lowercase:n
2268   {
2269     \group_end:
2270     \cs_new_nopar:Npn \prg_variable_get_scope:N #1
2271     {
2272       \exp_last_unbraced:Nf \prg_variable_get_scope_aux:w
2273       { \cs_to_str:N #1 \exp_stop_f: \q_stop }
2274     }
2275     \cs_new_nopar:Npn \prg_variable_get_scope_aux:w #1#2 \q_stop
2276     { \token_if_eq_meaning:NNT & #1 { g } }
2277   }
2278 \group_begin:
2279   \tex_lccode:D '\& = '\_ \scan_stop:
2280   \tex_catcode:D '\& = \c_twelve
2281   \tl_to_lowercase:n
2282   {
2283     \group_end:
2284     \cs_new_nopar:Npn \prg_variable_get_type:N #1
2285     {
2286       \exp_after:wN \prg_variable_get_type_aux:w
2287       \token_to_str:N #1 & a \q_stop
2288     }
2289     \cs_new_nopar:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_stop
2290     {
2291       \token_if_eq_meaning:NNTF a #2
2292       {#1}
2293       { \prg_variable_get_type_aux:w #2#3 \q_stop }
2294     }
2295   }

```

(End definition for `\prg_variable_get_scope:N`. This function is documented on page 47.)

171.9 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 T_EX’s Mouth” by Bernd Raichle. Firstly we define the function for parsing the initial list and then the braced list afterwards.

```
2296 \cs_new_protected_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
2297   \cs_set:cpx{#1_quicksort:n}##1{
2298     \exp_not:c{#1_quicksort_start_partition:w} ##1
2299     \exp_not:n{#2\q_nil#3\q_stop}
2300   }
2301   \cs_set:cpx{#1_quicksort_braced:n}##1{
2302     \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
2303     \exp_not:N\q_nil\exp_not:N\q_stop
2304   }
2305   \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
2306     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2307     \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{\{}}
2308   }
2309   \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
2310     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2311     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{\{}}
2312   }
```

Now for doing the partitions.

```
2313 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2314   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2315   {
2316     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2317     \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2318     \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2319   }
2320   {##1}{##2}{##3}{##4}
2321 }
2322 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2323   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2324   {
2325     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2326     \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2327     \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2328   }
2329   {##1}{##2}{##3}{##4}
2330 }
```

```

2331 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2332   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2333   {
2334     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2335     \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2336     \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2337   }
2338   {##1}{##2}{##3}{##4}
2339 }
2340 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2341   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2342   {
2343     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2344     \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2345     \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2346   }
2347   {##1}{##2}{##3}{##4}
2348 }

```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```

2349 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2350   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}{##3}}
2351 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2352   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3}{##4}}
2353 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2354   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}{##2}{##3}}
2355 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2356   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##4}{##3}}
2357 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2358   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##4}{##3}}
2359 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2360   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3}{##4}}
2361 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2362   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}{##2}{##3}}
2363 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2364   \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.

```

2365 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2366   \exp_not:c{#1_quicksort_braced:n}{##2}
2367   \exp_not:c{#1_quicksort_function:n}{##1}
2368   \exp_not:c{#1_quicksort_braced:n}{##3}
2369 }
2370 }

```

(End definition for \prg_define_quicksort:nm.)

\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.

```
2371 \prg_define_quicksort:nnn {prg}{-}{-}
```

(End definition for `\prg_quicksort:n`. This function is documented on page 47.)

```
\prg_quicksort_function:n
\prg_quicksort_compare:nnTF
```

```
2372 \cs_set:Npn \prg_quicksort_function:n {\ERROR}
2373 \cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}
```

(End definition for `\prg_quicksort_function:n`. This function is documented on page 47.)

171.10 Deprecated functions

These were deprecated 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.

```
2374 \cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 { \deprecated }
2375 \cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 { \deprecated }
```

(End definition for `\prg_new_map_functions:Nn`. This function is documented on page ??.)

```
2376 \</initex | package>
```

172 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

```
2377 \<*initex | package>
```

```
2378 \<*package>
```

```
2379 \ProvidesExplPackage
```

```
2380 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
```

```
2381 \package_check_loaded_expl:
```

```
2382 \</package>
```

`\quark_new:N` Allocate a new quark.

```
2383 \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 48.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

`\q_mark`

`\q_no_value`

`\q_stop`

```

2384 \quark_new:N \q_nil
2385 \quark_new:N \q_mark
2386 \quark_new:N \q_no_value
2387 \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.

`\q_recursion_stop`

```

2388 \quark_new:N \q_recursion_tail
2389 \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.

`\quark_if_recursion_tail_stop:Nn`

```

2390 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2391 {
2392   \if_meaning:w #1 \q_recursion_tail
2393   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2394   \fi:
2395 }
2396 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2
2397 {
2398   \if_meaning:w #1 \q_recursion_tail
2399   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2400   \else:
2401     \exp_after:wN \use_none:n
2402   \fi:
2403   {#2}
2404 }

```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 50.)

`\quark_if_recursion_tail_stop:n` The same idea applies when testing multiple tokens, but here a little more care is needed.

`\quark_if_recursion_tail_stop:o` 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**.

`\quark_if_recursion_tail_stop_do:nn`

`\quark_if_recursion_tail_stop_do:on`

`\quark_if_recursion_tail_aux:w`

```

2405 \cs_new:Npn \quark_if_recursion_tail_stop:n #1

```

```

2406 {
2407   \if_catcode:w
2408   A
2409   \etex_detokenize:D \exp_after:wN
2410   {
2411     \quark_if_recursion_tail_aux:w #1 \q_recursion_stop
2412     \q_recursion_tail \q_recursion_stop \q_stop
2413   }
2414   A
2415   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2416   \fi:
2417 }
2418 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2
2419 {
2420   \if_catcode:w
2421   A
2422   \etex_detokenize:D \exp_after:wN
2423   {
2424     \quark_if_recursion_tail_aux:w #1 \q_recursion_stop
2425     \q_recursion_tail \q_recursion_stop \q_stop
2426   }
2427   A
2428   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2429   \else:
2430   \exp_after:wN \use_none:n
2431   \fi:
2432   {#2}
2433 }
2434 \cs_new:Npn \quark_if_recursion_tail_aux:w
2435   #1 \q_recursion_tail #2 \q_recursion_stop #3 \q_stop
2436   { #1 #2 }
2437 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2438 \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 50.)

`\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` 2439 \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
`\quark_if_no_value:cTF` 2440 {

```

2441   \if_meaning:w \q_nil #1
2442   \prg_return_true:
2443   \else:
2444   \prg_return_false:
2445   \fi:

```

⁶It may still loop in special circumstances however!

```

2446 }
2447 \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2448 {
2449   \if_meaning:w \q_no_value #1
2450   \prg_return_true:
2451   \else:
2452   \prg_return_false:
2453   \fi:
2454 }
2455 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2456 \cs_generate_variant:Nn \quark_if_no_value:NT { c }
2457 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2458 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for `\quark_if_nil:N`. These functions are documented on page 49.)

`\quark_if_nil_p:n`
`\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`

These are essentially `\str_if_eq:nn` tests but done directly.

```

2459 \prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
2460 {
2461   \if_int_compare:w \pdfTex_strcmp:D
2462   { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero
2463   \prg_return_true:
2464   \else:
2465   \prg_return_false:
2466   \fi:
2467 }
2468 \prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
2469 {
2470   \if_int_compare:w \pdfTex_strcmp:D
2471   { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero
2472   \prg_return_true:
2473   \else:
2474   \prg_return_false:
2475   \fi:
2476 }
2477 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
2478 \cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
2479 \cs_generate_variant:Nn \quark_if_nil:nT { V , o }
2480 \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 49.)

`\q_tl_act_mark`
`\q_tl_act_stop`

These private quarks are needed by `l3tl`, but that is loaded before the quark module, hence their definition is deferred.

```

2481 \quark_new:N \q_tl_act_mark
2482 \quark_new:N \q_tl_act_stop
2483 </initex | package>

```

173 l3token implementation

```
2484 <*initex | package>
2485 <*package>
2486 \ProvidesExplPackage
2487   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
2488 \package_check_loaded_expl:
2489 </package>
```

173.1 Character tokens

Category code changes.

```
\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n
```

```
2490 \cs_new_protected_nopar:Npn \char_set_catcode:nn #1#2
2491   { \tex_catcode:D #1 = \int_eval:w #2 \int_eval_end: }
2492 \cs_new_nopar:Npn \char_value_catcode:n #1
2493   { \tex_the:D \tex_catcode:D \int_eval:w #1 \int_eval_end: }
2494 \cs_new_nopar:Npn \char_show_value_catcode:n #1
2495   { \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 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
```

```
2496 \cs_new_protected_nopar:Npn \char_set_catcode_escape:N #1
2497   { \char_set_catcode:nn { '#1 } \c_zero }
2498 \cs_new_protected_nopar:Npn \char_set_catcode_group_begin:N #1
2499   { \char_set_catcode:nn { '#1 } \c_one }
2500 \cs_new_protected_nopar:Npn \char_set_catcode_group_end:N #1
2501   { \char_set_catcode:nn { '#1 } \c_two }
2502 \cs_new_protected_nopar:Npn \char_set_catcode_math_toggle:N #1
2503   { \char_set_catcode:nn { '#1 } \c_three }
2504 \cs_new_protected_nopar:Npn \char_set_catcode_alignment:N #1
2505   { \char_set_catcode:nn { '#1 } \c_four }
2506 \cs_new_protected_nopar:Npn \char_set_catcode_end_line:N #1
2507   { \char_set_catcode:nn { '#1 } \c_five }
2508 \cs_new_protected_nopar:Npn \char_set_catcode_parameter:N #1
2509   { \char_set_catcode:nn { '#1 } \c_six }
2510 \cs_new_protected_nopar:Npn \char_set_catcode_math_superscript:N #1
2511   { \char_set_catcode:nn { '#1 } \c_seven }
2512 \cs_new_protected_nopar:Npn \char_set_catcode_math_subscript:N #1
2513   { \char_set_catcode:nn { '#1 } \c_eight }
2514 \cs_new_protected_nopar:Npn \char_set_catcode_ignore:N #1
2515   { \char_set_catcode:nn { '#1 } \c_nine }
2516 \cs_new_protected_nopar:Npn \char_set_catcode_space:N #1
2517   { \char_set_catcode:nn { '#1 } \c_ten }
2518 \cs_new_protected_nopar:Npn \char_set_catcode_letter:N #1
2519   { \char_set_catcode:nn { '#1 } \c_eleven }
2520 \cs_new_protected_nopar:Npn \char_set_catcode_other:N #1
```

```

2521 { \char_set_catcode:nn { '#1 } \c_twelve }
2522 \cs_new_protected_nopar:Npn \char_set_catcode_active:N #1
2523 { \char_set_catcode:nn { '#1 } \c_thirteen }
2524 \cs_new_protected_nopar:Npn \char_set_catcode_comment:N #1
2525 { \char_set_catcode:nn { '#1 } \c_fourteen }
2526 \cs_new_protected_nopar:Npn \char_set_catcode_invalid:N #1
2527 { \char_set_catcode:nn { '#1 } \c_fifteen }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 53.)

```

\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

2528 \cs_new_protected_nopar:Npn \char_set_catcode_escape:n #1
2529 { \char_set_catcode:nn {#1} \c_zero }
2530 \cs_new_protected_nopar:Npn \char_set_catcode_group_begin:n #1
2531 { \char_set_catcode:nn {#1} \c_one }
2532 \cs_new_protected_nopar:Npn \char_set_catcode_group_end:n #1
2533 { \char_set_catcode:nn {#1} \c_two }
2534 \cs_new_protected_nopar:Npn \char_set_catcode_math_toggle:n #1
2535 { \char_set_catcode:nn {#1} \c_three }
2536 \cs_new_protected_nopar:Npn \char_set_catcode_alignment:n #1
2537 { \char_set_catcode:nn {#1} \c_four }
2538 \cs_new_protected_nopar:Npn \char_set_catcode_end_line:n #1
2539 { \char_set_catcode:nn {#1} \c_five }
2540 \cs_new_protected_nopar:Npn \char_set_catcode_parameter:n #1
2541 { \char_set_catcode:nn {#1} \c_six }
2542 \cs_new_protected_nopar:Npn \char_set_catcode_math_superscript:n #1
2543 { \char_set_catcode:nn {#1} \c_seven }
2544 \cs_new_protected_nopar:Npn \char_set_catcode_math_subscript:n #1
2545 { \char_set_catcode:nn {#1} \c_eight }
2546 \cs_new_protected_nopar:Npn \char_set_catcode_ignore:n #1
2547 { \char_set_catcode:nn {#1} \c_nine }
2548 \cs_new_protected_nopar:Npn \char_set_catcode_space:n #1
2549 { \char_set_catcode:nn {#1} \c_ten }
2550 \cs_new_protected_nopar:Npn \char_set_catcode_letter:n #1
2551 { \char_set_catcode:nn {#1} \c_eleven }
2552 \cs_new_protected_nopar:Npn \char_set_catcode_other:n #1
2553 { \char_set_catcode:nn {#1} \c_twelve }
2554 \cs_new_protected_nopar:Npn \char_set_catcode_active:n #1
2555 { \char_set_catcode:nn {#1} \c_thirteen }
2556 \cs_new_protected_nopar:Npn \char_set_catcode_comment:n #1
2557 { \char_set_catcode:nn {#1} \c_fourteen }
2558 \cs_new_protected_nopar:Npn \char_set_catcode_invalid:n #1
2559 { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 53.)

Pretty repetitive, but necessary!

```

\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

```

```

2560 \cs_new_protected_nopar:Npn \char_set_mathcode:nn #1#2
2561 { \tex_mathcode:D #1 = \int_eval:w #2 \int_eval_end: }
2562 \cs_new_nopar:Npn \char_value_mathcode:n #1
2563 { \tex_the:D \tex_mathcode:D \int_eval:w #1\int_eval_end: }
2564 \cs_new_nopar:Npn \char_show_value_mathcode:n #1
2565 { \tex_showthe:D \tex_mathcode:D \int_eval:w #1 \int_eval_end: }
2566 \cs_new_protected_nopar:Npn \char_set_lccode:nn #1#2
2567 { \tex_lccode:D #1 = \int_eval:w #2 \int_eval_end: }
2568 \cs_new_nopar:Npn \char_value_lccode:n #1
2569 { \tex_the:D \tex_lccode:D \int_eval:w #1\int_eval_end: }
2570 \cs_new_nopar:Npn \char_show_value_lccode:n #1
2571 { \tex_showthe:D \tex_lccode:D \int_eval:w #1 \int_eval_end: }
2572 \cs_new_protected_nopar:Npn \char_set_uccode:nn #1#2
2573 { \tex_uccode:D #1 = \int_eval:w #2 \int_eval_end: }
2574 \cs_new_nopar:Npn \char_value_uccode:n #1
2575 { \tex_the:D \tex_uccode:D \int_eval:w #1\int_eval_end: }
2576 \cs_new_nopar:Npn \char_show_value_uccode:n #1
2577 { \tex_showthe:D \tex_uccode:D \int_eval:w #1 \int_eval_end: }
2578 \cs_new_protected_nopar:Npn \char_set_sfcode:nn #1#2
2579 { \tex_sfcode:D #1 = \int_eval:w #2 \int_eval_end: }
2580 \cs_new_nopar:Npn \char_value_sfcode:n #1
2581 { \tex_the:D \tex_sfcode:D \int_eval:w #1\int_eval_end: }
2582 \cs_new_nopar:Npn \char_show_value_sfcode:n #1
2583 { \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 56.)

173.2 Generic tokens

`\token_new:Nn` Creates a new token.

```

2584 \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 56.)

`\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` `\c_catcode_letter_token` `\c_catcode_other_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```

2585 \cs_new_eq:NN \c_group_begin_token {
2586 \cs_new_eq:NN \c_group_end_token }
2587 \group_begin:
2588 \char_set_catcode_math_toggle:N \*
2589 \token_new:Nn \c_math_toggle_token { * }
2590 \char_set_catcode_alignment:N \*
2591 \token_new:Nn \c_alignment_token { * }
2592 \token_new:Nn \c_parameter_token { # }
2593 \token_new:Nn \c_math_superscript_token { ^ }
2594 \char_set_catcode_math_subscript:N \*
2595 \token_new:Nn \c_math_subscript_token { * }

```

```

2596 \token_new:Nn \c_space_token { ~ }
2597 \token_new:Nn \c_catcode_letter_token { a }
2598 \token_new:Nn \c_catcode_other_token { 1 }
2599 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 56.)

`\c_catcode_active_tl` Not an implicit token!

```

2600 \group_begin:
2601 \char_set_catcode_active:N \*
2602 \cs_new_nopar:Npn \c_catcode_active_tl { \exp_not:N * }
2603 \group_end:

```

173.3 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`

```

2604 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
2605 {
2606   \if_catcode:w \exp_not:N #1 \c_group_begin_token
2607   \prg_return_true: \else: \prg_return_false: \fi:
2608 }

```

(End definition for `\token_if_group_begin:N`. These functions are documented on page 57.)

`\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`

```

2609 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
2610 {
2611   \if_catcode:w \exp_not:N #1 \c_group_end_token
2612   \prg_return_true: \else: \prg_return_false: \fi:
2613 }

```

(End definition for `\token_if_group_end:N`. These functions are documented on page 57.)

`\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`

```

2614 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
2615 {
2616   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
2617   \prg_return_true: \else: \prg_return_false: \fi:
2618 }

```

(End definition for `\token_if_math_toggle:N`. These functions are documented on page 58.)

`\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`

```

2619 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
2620 {
2621   \if_catcode:w \exp_not:N #1 \c_alignment_token
2622   \prg_return_true: \else: \prg_return_false: \fi:
2623 }

```

(End definition for `\token_if_alignment:N`. These functions are documented on page 58.)

`\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.

```

2624 \group_begin:
2625 \cs_set_eq:NN \c_parameter_token \scan_stop:
2626 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
2627 {
2628   \if_catcode:w \exp_not:N #1 \c_parameter_token
2629   \prg_return_true: \else: \prg_return_false: \fi:
2630 }
2631 \group_end:

```

(End definition for `\token_if_parameter:N`. These functions are documented on page 58.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_superscript_token` for this.
`\token_if_math_superscript:NTF`

```

2632 \prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T , F , TF }
2633 {
2634   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
2635   \prg_return_true: \else: \prg_return_false: \fi:
2636 }

```

(End definition for `\token_if_math_superscript:N`. These functions are documented on page 58.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_subscript_token` for this.
`\token_if_math_subscript:NTF`

```

2637 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
2638 {
2639   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
2640   \prg_return_true: \else: \prg_return_false: \fi:
2641 }

```

(End definition for `\token_if_math_subscript:N`. These functions are documented on page 58.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.
`\token_if_space:NTF`

```
2642 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
2643 {
2644   \if_catcode:w \exp_not:N #1 \c_space_token
2645   \prg_return_true: \else: \prg_return_false: \fi:
2646 }
```

(End definition for \token_if_space:N. These functions are documented on page 58.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_letter_token` for this.
`\token_if_letter:NTF`

```
2647 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
2648 {
2649   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
2650   \prg_return_true: \else: \prg_return_false: \fi:
2651 }
```

(End definition for \token_if_letter:N. These functions are documented on page 59.)

`\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:NTF`

```
2652 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
2653 {
2654   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
2655   \prg_return_true: \else: \prg_return_false: \fi:
2656 }
```

(End definition for \token_if_other:N. These functions are documented on page 59.)

`\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:NTF`

```
2657 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
2658 {
2659   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
2660   \prg_return_true: \else: \prg_return_false: \fi:
2661 }
```

(End definition for \token_if_active:N. These functions are documented on page 59.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NNTF`

```
2662 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
2663 {
2664   \if_meaning:w #1 #2
2665   \prg_return_true: \else: \prg_return_false: \fi:
2666 }
```

(End definition for `\token_if_eq_meaning:NN`. These functions are documented on page 59.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NNTF`

```
2667 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
2668 {
2669   \if_catcode:w \exp_not:N #1 \exp_not:N #2
2670   \prg_return_true: \else: \prg_return_false: \fi:
2671 }
```

(End definition for `\token_if_eq_catcode:NN`. These functions are documented on page 59.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NNTF`

```
2672 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
2673 {
2674   \if_charcode:w \exp_not:N #1 \exp_not:N #2
2675   \prg_return_true: \else: \prg_return_false: \fi:
2676 }
```

(End definition for `\token_if_eq_charcode:NN`. These functions are documented on page 59.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like
`\token_if_macro:N` `\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 `->`.
`\token_if_macro_p_aux:w`

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.

```
2677 \group_begin:
2678 \char_set_catcode_other:N \M
2679 \char_set_catcode_other:N \A
2680 \char_set_lccode:nn { '\; } { '\: }
2681 \char_set_lccode:nn { '\T } { '\T }
2682 \char_set_lccode:nn { '\F } { '\F }
2683 \tl_to_lowercase:n
2684 {
2685   \group_end:
```

```

2686 \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T , F , TF }
2687 {
2688   \exp_after:wN \token_if_macro_p_aux:w
2689   \token_to_meaning:N #1 MA; \q_stop
2690 }
2691 \cs_new_nopar:Npn \token_if_macro_p_aux:w #1 MA #2 ; #3 \q_stop
2692 {
2693   \if_int_compare:w \pdfTeX_strcmp:D { #2 } { cro } = \c_zero
2694   \prg_return_true:
2695   \else:
2696     \prg_return_false:
2697   \fi:
2698 }
2699 }

```

(End definition for `\token_if_macro:N`. These functions are documented on page 59.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:NTF` for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

2700 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
2701 {
2702   \if_catcode:w \exp_not:N #1 \scan_stop:
2703   \prg_return_true: \else: \prg_return_false: \fi:
2704 }

```

(End definition for `\token_if_cs:N`. These functions are documented on page 60.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that \TeX will temporarily convert
`\token_if_expandable:NTF` `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable.

```

2705 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
2706 {
2707   \cs_if_exist:NTF #1
2708   {
2709     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2710     \prg_return_false: \else: \prg_return_true: \fi:
2711   }
2712   { \prg_return_false: }
2713 }

```

(End definition for `\token_if_expandable:N`. These functions are documented on page 60.)

`\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...

```

2714 \group_begin:
2715 \char_set_lccode:nn { '\T } { '\T }

```

```

\token_if_chardef:N\TF
\token_if_mathchardef:N\TF
\token_if_long_macro:N\TF
\token_if_protected_macro:N\TF
\token_if_protected_long_macro:N\TF
\token_if_dim_register:N\TF
\token_if_skip_register:N\TF
\token_if_int_register:N\TF
\token_if_toks_register:N\TF

```

```

2716 \char_set_lccode:nn { '\F } { '\F }
2717 \char_set_lccode:nn { '\X } { '\n }
2718 \char_set_lccode:nn { '\Y } { '\t }
2719 \char_set_lccode:nn { '\Z } { '\d }
2720 \char_set_lccode:nn { '\? } { '\l }
2721 \tl_map_inline:nn { \X \Y \Z \M \C \H \A \R \O \U \S \K \I \P \L \G \P \E }
2722   { \char_set_catcode:nn { '#1 } \c_twelve }

```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```

2723 \tl_to_lowercase:n
2724 {
2725   \group_end:

```

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since \TeX thinks of such tokens as hexadecimal so it stores them as `\char"⟨hex number⟩` or `\mathchar"⟨hex number⟩`.

```

2726 \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T , F , TF }
2727 {
2728   \exp_after:wN \token_if_chardef_aux:w
2729   \token_to_meaning:N #1 ?CHAR" \q_stop
2730 }
2731 \cs_new_nopar:Npn \token_if_chardef_aux:w #1 ?CHAR" #2 \q_stop
2732 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

2733 \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T , F , TF }
2734 {
2735   \exp_after:wN \token_if_mathchardef_aux:w
2736   \token_to_meaning:N #1 ?MAYHCHAR" \q_stop
2737 }
2738 \cs_new_nopar:Npn \token_if_mathchardef_aux:w #1 ?MAYHCHAR" #2 \q_stop
2739 { \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.

```

2740 \prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T , F , TF }
2741 {
2742   \if_meaning:w \tex_countdef:D #1
2743   \prg_return_false:
2744   \else:
2745     \exp_after:wN \token_if_int_register_aux:w
2746     \token_to_meaning:N #1 ?COUXY \q_stop
2747   \fi:
2748 }
2749 \cs_new_nopar:Npn \token_if_int_register_aux:w #1 ?COUXY #2 \q_stop
2750 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Skip registers are done the same way as the integer registers.

```

2751 \prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T , F , TF }
2752 {
2753   \if_meaning:w \tex_skipdef:D #1
2754   \prg_return_false:
2755   \else:
2756     \exp_after:wN \token_if_skip_register_aux:w
2757     \token_to_meaning:N #1?SKIP\q_stop
2758   \fi:
2759 }
2760 \cs_new_nopar:Npn \token_if_skip_register_aux:w #1 ?SKIP #2 \q_stop
2761 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Dim registers. No news here

```

2762 \prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T , F , TF }
2763 {
2764   \if_meaning:w \tex_dimendef:D #1
2765   \c_false_bool
2766   \else:
2767     \exp_after:wN \token_if_dim_register_aux:w
2768     \token_to_meaning:N #1 ?ZIMEX \q_stop
2769   \fi:
2770 }
2771 \cs_new_nopar:Npn \token_if_dim_register_aux:w #1 ?ZIMEX #2 \q_stop
2772 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Toks registers.

```

2773 \prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T , F , TF }
2774 {
2775   \if_meaning:w \tex_toksdef:D #1
2776   \prg_return_false:
2777   \else:
2778     \exp_after:wN \token_if_toks_register_aux:w
2779     \token_to_meaning:N #1 ?YOKS \q_stop
2780   \fi:
2781 }
2782 \cs_new_nopar:Npn \token_if_toks_register_aux:w #1 ?YOKS #2 \q_stop
2783 { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Protected macros.

```

2784 \prg_new_conditional:Npnn \token_if_protected_macro:N #1
2785 { p , T , F , TF }
2786 {
2787   \exp_after:wN \token_if_protected_macro_aux:w
2788   \token_to_meaning:N #1 ?PROYECYEZ~MACRO \q_stop
2789 }
2790 \cs_new_nopar:Npn \token_if_protected_macro_aux:w

```

```

2791      #1 ?PROYECY EZ~MACRO #2 \q_stop
2792      { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Long macros.

```

2793      \prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T , F , TF }
2794      {
2795          \exp_after:wN \token_if_long_macro_aux:w
2796          \token_to_meaning:N #1 ?LOXG~MACRO \q_stop
2797      }
2798      \cs_new_nopar:Npn \token_if_long_macro_aux:w #1 ?LOXG~MACRO #2 \q_stop
2799      { \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.

```

2800      \prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
2801      { p , T , F , TF }
2802      {
2803          \exp_after:wN \token_if_protected_long_macro_aux:w
2804          \token_to_meaning:N #1 ?PROYECY EZ?LOXG~MACRO \q_stop
2805      }
2806      \cs_new_nopar:Npn \token_if_protected_long_macro_aux:w
2807      #1 ?PROYECY EZ?LOXG~MACRO #2 \q_stop
2808      { \tl_if_empty:nTF {#1} { \prg_return_true: } { \prg_return_false: } }

```

Finally the `\tl_to_lowercase:n` ends!

```

2809      }

```

(End definition for `\token_if_chardef:N` and others. These functions are documented on page 61.)

```

\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.

```

2810 \tex_chardef:D \c_token_A_int = 'A ~ %
2811 \group_begin:
2812 \char_set_catcode_other:N \;
2813 \char_set_lccode:nn { '\; } { '\: }
2814 \char_set_lccode:nn { '\T } { '\T }
2815 \char_set_lccode:nn { '\F } { '\F }
2816 \tl_to_lowercase:n {
2817   \group_end:
2818   \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
2819   {
2820     \token_if_macro:NTF #1
2821     \prg_return_false:
2822     {
2823       \exp_after:wN \token_if_primitive_aux:NNw
2824       \token_to_meaning:N #1 ; ; ; \q_stop #1
2825     }
2826   }
2827   \cs_new_nopar:Npn \token_if_primitive_aux:NNw #1#2 #3 ; #4 \q_stop
2828   {
2829     \tl_if_empty:oTF { \token_if_primitive_aux_space:w #3 ~ }
2830     { \token_if_primitive_aux_loop:N #3 ; \q_stop }
2831     { \token_if_primitive_aux_nullfont:N }
2832   }
2833 }
2834 \cs_new_nopar:Npn \token_if_primitive_aux_space:w #1 ~ { }
2835 \cs_new:Npn \token_if_primitive_aux_nullfont:N #1
2836 {
2837   \if_meaning:w \tex_nullfont:D #1
2838   \prg_return_true:
2839   \else:
2840     \prg_return_false:
2841   \fi:
2842 }
2843 \cs_new_nopar:Npn \token_if_primitive_aux_loop:N #1
2844 {

```

```

2845     \if_num:w '#1 < \c_token_A_int %
2846         \exp_after:wN \token_if_primitive_auxii:Nw
2847         \exp_after:wN #1
2848     \else:
2849         \exp_after:wN \token_if_primitive_aux_loop:N
2850     \fi:
2851 }
2852 \cs_new_nopar:Npn \token_if_primitive_auxii:Nw #1 #2 \q_stop
2853 {
2854     \if:w : #1
2855         \exp_after:wN \token_if_primitive_aux_undefined:N
2856     \else:
2857         \prg_return_false:
2858         \exp_after:wN \use_none:n
2859     \fi:
2860 }
2861 \cs_new:Npn \token_if_primitive_aux_undefined:N #1
2862 {
2863     \if_cs_exist:N #1
2864         \prg_return_true:
2865     \else:
2866         \prg_return_false:
2867     \fi:
2868 }

```

(End definition for `\token_if_primitive:N`. These functions are documented on page 61.)

173.4 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`

```

2869 \cs_new_eq:NN \l_peek_token ?
2870 \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`.

```

2871 \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`.

```
2872 \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
2873 \cs_new_nopar:Npn \peek_true:w { }
2874 \cs_new_nopar:Npn \peek_true_aux:w { }
2875 \cs_new_nopar:Npn \peek_false:w { }
2876 \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
2877 \cs_new_protected_nopar:Npn \peek_after:Nw
2878 { \tex_futurelet:D \l_peek_token }
2879 \cs_new_protected_nopar:Npn \peek_gafter:Nw
2880 { \pref_global:D \tex_futurelet:D \g_peek_token }
```

(End definition for `\peek_after:Nw`. This function is documented on page 61.)

`\peek_true_remove:w` A function to remove the next token and then regain control.

```
2881 \cs_new_protected:Npn \peek_true_remove:w
2882 {
2883   \group_align_safe_end:
2884   \tex_afterassignment:D \peek_true_aux:w
2885   \cs_set_eq:NN \peek_tmp:w
2886 }
```

(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.

```
2887 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4
2888 {
2889   \cs_set_eq:NN \l_peek_search_token #2
2890   \tl_set:Nn \l_peek_search_tl {#2}
2891   \cs_set_nopar:Npx \peek_true:w
2892   {
2893     \exp_not:N \group_align_safe_end:
2894     \exp_not:n {#3}
2895   }
2896   \cs_set_nopar:Npx \peek_false:w
2897   {
2898     \exp_not:N \group_align_safe_end:
2899     \exp_not:n {#4}
```

```

2900     }
2901     \group_align_safe_begin:
2902     \peek_after:Nw #1
2903   }
2904   \cs_new_protected:Npn \peek_token_generic:NNT #1#2#3
2905   { \peek_token_generic:NNTF #1 #2 {#3} { } }
2906   \cs_new_protected:Npn \peek_token_generic:NNF #1#2#3
2907   { \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.

```

2908   \cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4
2909   {
2910     \cs_set_eq:NN \l_peek_search_token #2
2911     \tl_set:Nn \l_peek_search_tl {#2}
2912     \cs_set_eq:NN \peek_true:w \peek_true_remove:w
2913     \cs_set_nopar:Npx \peek_true_aux:w { \exp_not:n {#3} }
2914     \cs_set_nopar:Npx \peek_false:w
2915     {
2916       \exp_not:N \group_align_safe_end:
2917       \exp_not:n {#4}
2918     }
2919     \group_align_safe_begin:
2920     \peek_after:Nw #1
2921   }
2922   \cs_new_protected:Npn \peek_token_remove_generic:NNT #1#2#3
2923   { \peek_token_remove_generic:NNTF #1 #2 {#3} { } }
2924   \cs_new_protected:Npn \peek_token_remove_generic:NNF #1#2#3
2925   { \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:
2926   \cs_new_nopar:Npn \peek_execute_branches_catcode:
2927   {
2928     \if_catcode:w
2929       \exp_not:N \l_peek_token \exp_not:N \l_peek_search_token
2930       \exp_after:wN \peek_true:w
2931     \else:
2932       \exp_after:wN \peek_false:w
2933     \fi:
2934   }
2935   \cs_new_nopar:Npn \peek_execute_branches_meaning:
2936   {
2937     \if_meaning:w \l_peek_token \l_peek_search_token
2938       \exp_after:wN \peek_true:w
2939     \else:

```

```

2940     \exp_after:wN \peek_false:w
2941     \fi:
2942 }

```

(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 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.

```

2943 \cs_new_nopar:Npn \peek_execute_branches_charcode:
2944 {
2945     \bool_if:nTF
2946     {
2947         \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token
2948         || \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token
2949         || \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
2950     }
2951     { \peek_false:w }
2952     {
2953         \exp_after:wN \peek_execute_branches_charcode_aux:NN
2954         \l_peek_search_tl
2955     }
2956 }
2957 \cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2
2958 {
2959     \if:w \exp_not:N #1 \exp_not:N #2
2960     \exp_after:wN \peek_true:w
2961     \else:
2962     \exp_after:wN \peek_false:w
2963     \fi:
2964     #2
2965 }

```

(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 things other than spaces.

```

2966 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches:
2967 {
2968     \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
2969     {
2970         \tex_afterassignment:D \peek_ignore_spaces_execute_branches_aux:
2971         \cs_set_eq:NN \peek_tmp:w
2972     }
2973     { \peek_execute_branches: }
2974 }
2975 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches_aux:
2976 { \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.

```

2977 \group_begin:
2978   \cs_set_nopar:Npn \peek_def:nnnn #1#2#3#4
2979   {
2980     \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { TF }
2981     \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { T }
2982     \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { F }
2983   }
2984   \cs_set_nopar:Npn \peek_def_aux:nnnnn #1#2#3#4#5
2985   {
2986     \cs_gset_nopar:cpx { #1 #5 }
2987     {
2988       \tl_if_empty:nF {#2}
2989       { \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 } }
2990       \exp_not:c { #3 #5 }
2991       \exp_not:n {#4}
2992     }
2993   }

```

(End definition for `\peek_def:nnnn`.)

`\peek_catcode:NTF` With everything in place the definitions can take place. First for category codes.
`\peek_catcode_ignore_spaces:NTF`
`\peek_catcode_remove:NTF`
`\peek_catcode_remove_ignore_spaces:NTF`

```

2994 \peek_def:nnnn { peek_catcode:N }
2995 { }
2996 { peek_token_generic:NN }
2997 { \peek_execute_branches_catcode: }
2998 \peek_def:nnnn { peek_catcode_ignore_spaces:N }
2999 { \peek_execute_branches_catcode: }
3000 { peek_token_generic:NN }
3001 { \peek_ignore_spaces_execute_branches: }
3002 \peek_def:nnnn { peek_catcode_remove:N }
3003 { }
3004 { peek_token_remove_generic:NN }
3005 { \peek_execute_branches_catcode: }
3006 \peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
3007 { \peek_execute_branches_catcode: }
3008 { peek_token_remove_generic:NN }
3009 { \peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_catcode:N` and others. These functions are documented on page 62.)

`\peek_charcode:NTF` Then for character codes.
`\peek_charcode_ignore_spaces:NTF`
`\peek_charcode_remove:NTF`
`\peek_charcode_remove_ignore_spaces:NTF`

```

3010 \peek_def:nnnn { peek_charcode:N }

```

```

3011 { }
3012 { peek_token_generic:NN }
3013 { \peek_execute_branches_charcode: }
3014 \peek_def:nnnn { peek_charcode_ignore_spaces:N }
3015 { \peek_execute_branches_charcode: }
3016 { peek_token_generic:NN }
3017 { \peek_ignore_spaces_execute_branches: }
3018 \peek_def:nnnn { peek_charcode_remove:N }
3019 { }
3020 { peek_token_remove_generic:NN }
3021 { \peek_execute_branches_charcode: }
3022 \peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3023 { \peek_execute_branches_charcode: }
3024 { peek_token_remove_generic:NN }
3025 { \peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_charcode:N` and others. These functions are documented on page 63.)

`\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.

```

3026 \peek_def:nnnn { peek_meaning:N }
3027 { }
3028 { peek_token_generic:NN }
3029 { \peek_execute_branches_meaning: }
3030 \peek_def:nnnn { peek_meaning_ignore_spaces:N }
3031 { \peek_execute_branches_meaning: }
3032 { peek_token_generic:NN }
3033 { \peek_ignore_spaces_execute_branches: }
3034 \peek_def:nnnn { peek_meaning_remove:N }
3035 { }
3036 { peek_token_remove_generic:NN }
3037 { \peek_execute_branches_meaning: }
3038 \peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3039 { \peek_execute_branches_meaning: }
3040 { peek_token_remove_generic:NN }
3041 { \peek_ignore_spaces_execute_branches: }
3042 \group_end:

```

(End definition for `\peek_meaning:N` and others. These functions are documented on page 64.)

173.5 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.

```

3043 \exp_args:Nno \use:nn
3044 { \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:wN #1 }
3045 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3046 { #4 {#1} {#2} {#3} }
3047 \cs_new:Npn \token_get_prefix_spec:N #1
3048 {
3049   \token_if_macro:NTF #1
3050   {
3051     \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
3052     \token_to_meaning:N #1 \q_stop \use_i:nnn
3053   }
3054   { \scan_stop: }
3055 }
3056 \cs_new:Npn \token_get_arg_spec:N #1
3057 {
3058   \token_if_macro:NTF #1
3059   {
3060     \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
3061     \token_to_meaning:N #1 \q_stop \use_ii:nnn
3062   }
3063   { \scan_stop: }
3064 }
3065 \cs_new:Npn \token_get_replacement_spec:N #1
3066 {
3067   \token_if_macro:NTF #1
3068   {
3069     \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
3070     \token_to_meaning:N #1 \q_stop \use_iii:nnn
3071   }
3072   { \scan_stop: }
3073 }

```

(End definition for `\token_get_prefix_spec:N`. This function is documented on page ??.)

173.6 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

3074 \group_begin:
3075   \char_set_catcode_active:N ^^@
3076   \cs_set:Npn \char_tmp:NN #1#2
3077   {
3078     \cs_new:Npn #1 ##1
3079     {
3080       \char_set_catcode_active:n { '##1 }
3081       \group_begin:
3082       \char_set_lccode:nn { '\^^@ } { '##1 }
3083       \tl_to_lowercase:n { \group_end: #2 ^^@ }
3084     }

```

```

3085     }
3086     \char_tmp:NN \char_active_set:Npn    \cs_set:Npn
3087     \char_tmp:NN \char_active_set:Npx    \cs_set:Npx
3088     \char_tmp:NN \char_active_gset:Npn    \cs_gset:Npn
3089     \char_tmp:NN \char_active_gset:Npx    \cs_gset:Npx
3090     \char_tmp:NN \char_active_set_eq:NN    \cs_set_eq:NN
3091     \char_tmp:NN \char_active_gset_eq:NN \cs_gset_eq:NN
3092 \group_end:

```

(End definition for `\char_active_set:Npn` and `\char_active_set:Npx`. These functions are documented on page 66.)

`\peek_N_type:TF` The next token is normal if it is neither a begin-group token, nor an end-group token, nor a charcode-32 space token. Note that implicit begin-group tokens, end-group tokens, and spaces are also recognized as non-N-type. Here, there is no *search token*, so we feed a dummy `\scan_stop:` to the `\peek_token_generic::NN` functions.

```

3093 \cs_new_protected_nopar:Npn \peek_execute_branches_N_type:
3094 {
3095     \bool_if:nTF
3096     {
3097         \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
3098         \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token ||
3099         \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
3100     }
3101     { \peek_false:w }
3102     { \peek_true:w }
3103 }
3104 \cs_new_protected_nopar:Npn \peek_N_type:TF
3105 { \peek_token_generic:NNTF \peek_execute_branches_N_type: \scan_stop: }
3106 \cs_new_protected_nopar:Npn \peek_N_type:T
3107 { \peek_token_generic:NNT \peek_execute_branches_N_type: \scan_stop: }
3108 \cs_new_protected_nopar:Npn \peek_N_type:F
3109 { \peek_token_generic:NNF \peek_execute_branches_N_type: \scan_stop: }

```

(End definition for `\peek_N_type:`. This function is documented on page ??.)

173.7 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
\char_set_uccode:w
\char_set_sfcode:w
3110 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
3111 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
3112 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
3113 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
3114 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D

```

(End definition for `\char_set_catcode:w`. This function is documented on page ??.)

<code>\char_value_catcode:w</code>	More w functions we should not have.
<code>\char_show_value_catcode:w</code>	
<code>\char_value_mathcode:w</code>	3115 <code>\cs_new_nopar:Npn \char_value_catcode:w { \tex_the:D \char_set_catcode:w }</code>
<code>\char_show_value_mathcode:w</code>	3116 <code>\cs_new_nopar:Npn \char_show_value_catcode:w</code>
<code>\char_value_lccode:w</code>	3117 <code>{ \tex_showthe:D \char_set_catcode:w }</code>
<code>\char_show_value_lccode:w</code>	3118 <code>\cs_new_nopar:Npn \char_value_mathcode:w { \tex_the:D \char_set_mathcode:w }</code>
<code>\char_value_uccode:w</code>	3119 <code>\cs_new_nopar:Npn \char_show_value_mathcode:w</code>
<code>\char_show_value_uccode:w</code>	3120 <code>{ \tex_showthe:D \char_set_mathcode:w }</code>
<code>\char_value_sfcode:w</code>	3121 <code>\cs_new_nopar:Npn \char_value_lccode:w { \tex_the:D \char_set_lccode:w }</code>
<code>\char_show_value_sfcode:w</code>	3122 <code>\cs_new_nopar:Npn \char_show_value_lccode:w</code>
	3123 <code>{ \tex_showthe:D \char_set_lccode:w }</code>
	3124 <code>\cs_new_nopar:Npn \char_value_uccode:w { \tex_the:D \char_set_uccode:w }</code>
	3125 <code>\cs_new_nopar:Npn \char_show_value_uccode:w</code>
	3126 <code>{ \tex_showthe:D \char_set_uccode:w }</code>
	3127 <code>\cs_new_nopar:Npn \char_value_sfcode:w { \tex_the:D \char_set_sfcode:w }</code>
	3128 <code>\cs_new_nopar:Npn \char_show_value_sfcode:w</code>
	3129 <code>{ \tex_showthe:D \char_set_sfcode:w }</code>

(End definition for `\char_value_catcode:w`. This function is documented on page ??.)

<code>\peek_after:NN</code>	The second argument here must be w.
<code>\peek_gafter:NN</code>	

3130	<code>\cs_new_eq:NN \peek_after:NN \peek_after:Nw</code>
3131	<code>\cs_new_eq:NN \peek_gafter:NN \peek_gafter:Nw</code>

(End definition for `\peek_after:NN`. This function is documented on page ??.)

Functions deprecated 2011-05-28 for removal by 2011-08-31.

<code>\c_alignment_tab_token</code>	
<code>\c_math_shift_token</code>	3132 <code>\cs_new_eq:NN \c_alignment_tab_token \c_alignment_token</code>
<code>\c_letter_token</code>	3133 <code>\cs_new_eq:NN \c_math_shift_token \c_math_toggle_token</code>
<code>\c_other_char_token</code>	3134 <code>\cs_new_eq:NN \c_letter_token \c_catcode_letter_token</code>
	3135 <code>\cs_new_eq:NN \c_other_char_token \c_catcode_other_token</code>

(End definition for `\c_alignment_tab_token`. This function is documented on page ??.)

<code>\c_active_char_token</code>	An odd one: this was never a token!
-----------------------------------	-------------------------------------

3136	<code>\cs_new_eq:NN \c_active_char_token \c_catcode_active_tl</code>
------	--

(End definition for `\c_active_char_token`. This function is documented on page ??.)

<code>\char_make_escape:N</code>	Two renames in one block!	
<code>\char_make_group_begin:N</code>	3137 <code>\cs_new_eq:NN \char_make_escape:N</code>	<code>\char_set_catcode_escape:N</code>
<code>\char_make_group_end:N</code>	3138 <code>\cs_new_eq:NN \char_make_begin_group:N</code>	<code>\char_set_catcode_group_begin:N</code>
<code>\char_make_math_toggle:N</code>	3139 <code>\cs_new_eq:NN \char_make_end_group:N</code>	<code>\char_set_catcode_group_end:N</code>
<code>\char_make_alignment:N</code>		
<code>\char_make_end_line:N</code>		
<code>\char_make_parameter:N</code>		

<code>\char_make_math_superscript:N</code>
<code>\char_make_math_subscript:N</code>
<code>\char_make_ignore:N</code>
<code>\char_make_space:N</code>
<code>\char_make_letter:N</code>
<code>\char_make_other:N</code>
<code>\char_make_active:N</code>

```

3140 \cs_new_eq:NN \char_make_math_shift:N \char_set_catcode_math_toggle:N
3141 \cs_new_eq:NN \char_make_alignment_tab:N \char_set_catcode_alignment:N
3142 \cs_new_eq:NN \char_make_end_line:N \char_set_catcode_end_line:N
3143 \cs_new_eq:NN \char_make_parameter:N \char_set_catcode_parameter:N
3144 \cs_new_eq:NN \char_make_math_superscript:N
3145 \char_set_catcode_math_superscript:N
3146 \cs_new_eq:NN \char_make_math_subscript:N
3147 \char_set_catcode_math_subscript:N
3148 \cs_new_eq:NN \char_make_ignore:N \char_set_catcode_ignore:N
3149 \cs_new_eq:NN \char_make_space:N \char_set_catcode_space:N
3150 \cs_new_eq:NN \char_make_letter:N \char_set_catcode_letter:N
3151 \cs_new_eq:NN \char_make_other:N \char_set_catcode_other:N
3152 \cs_new_eq:NN \char_make_active:N \char_set_catcode_active:N
3153 \cs_new_eq:NN \char_make_comment:N \char_set_catcode_comment:N
3154 \cs_new_eq:NN \char_make_invalid:N \char_set_catcode_invalid:N
3155 \cs_new_eq:NN \char_make_escape:n \char_set_catcode_escape:n
3156 \cs_new_eq:NN \char_make_begin_group:n \char_set_catcode_group_begin:n
3157 \cs_new_eq:NN \char_make_end_group:n \char_set_catcode_group_end:n
3158 \cs_new_eq:NN \char_make_math_shift:n \char_set_catcode_math_toggle:n
3159 \cs_new_eq:NN \char_make_alignment_tab:n \char_set_catcode_alignment:n
3160 \cs_new_eq:NN \char_make_end_line:n \char_set_catcode_end_line:n
3161 \cs_new_eq:NN \char_make_parameter:n \char_set_catcode_parameter:n
3162 \cs_new_eq:NN \char_make_math_superscript:n
3163 \char_set_catcode_math_superscript:n
3164 \cs_new_eq:NN \char_make_math_subscript:n
3165 \char_set_catcode_math_subscript:n
3166 \cs_new_eq:NN \char_make_ignore:n \char_set_catcode_ignore:n
3167 \cs_new_eq:NN \char_make_space:n \char_set_catcode_space:n
3168 \cs_new_eq:NN \char_make_letter:n \char_set_catcode_letter:n
3169 \cs_new_eq:NN \char_make_other:n \char_set_catcode_other:n
3170 \cs_new_eq:NN \char_make_active:n \char_set_catcode_active:n
3171 \cs_new_eq:NN \char_make_comment:n \char_set_catcode_comment:n
3172 \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 $\textit{TF}$ 
\token_if_math_shift_p:N
\token_if_math_shift:N $\textit{TF}$ 
\token_if_other_char_p:N
\token_if_other_char:N $\textit{TF}$ 
\token_if_active_char_p:N
\token_if_active_char:N $\textit{TF}$ 

```

```

3173 \cs_new_eq:NN \token_if_alignment_tab_p:N \token_if_alignment_p:N
3174 \cs_new_eq:NN \token_if_alignment_tab:NT \token_if_alignment:NT
3175 \cs_new_eq:NN \token_if_alignment_tab:NF \token_if_alignment:NF
3176 \cs_new_eq:NN \token_if_alignment_tab:N $\textit{TF}$  \token_if_alignment:N $\textit{TF}$ 
3177 \cs_new_eq:NN \token_if_math_shift_p:N \token_if_math_toggle_p:N
3178 \cs_new_eq:NN \token_if_math_shift:NT \token_if_math_toggle:NT
3179 \cs_new_eq:NN \token_if_math_shift:NF \token_if_math_toggle:NF
3180 \cs_new_eq:NN \token_if_math_shift:N $\textit{TF}$  \token_if_math_toggle:N $\textit{TF}$ 
3181 \cs_new_eq:NN \token_if_other_char_p:N \token_if_other_p:N
3182 \cs_new_eq:NN \token_if_other_char:NT \token_if_other:NT
3183 \cs_new_eq:NN \token_if_other_char:NF \token_if_other:NF
3184 \cs_new_eq:NN \token_if_other_char:N $\textit{TF}$  \token_if_other:N $\textit{TF}$ 

```

```

3185 \cs_new_eq:NN \token_if_active_char_p:N \token_if_active_p:N
3186 \cs_new_eq:NN \token_if_active_char:NT \token_if_active:NT
3187 \cs_new_eq:NN \token_if_active_char:NF \token_if_active:NF
3188 \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 ??.)

```

3189 </initex | package>

```

174 l3int implementation

```

3190 <*initex | package>

```

The following test files are used for this code: `m3int001,m3int002,m3int03`.

```

3191 <*package>
3192 \ProvidesExplPackage
3193   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
3194 \package_check_loaded_expl:
3195 </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 77.)

`\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
3196 \cs_new_eq:NN \int_value:w \tex_number:D
3197 \cs_new_eq:NN \int_eval:w \etex_numexpr:D
3198 \cs_new_eq:NN \int_eval_end: \tex_relax:D
3199 \cs_new_eq:NN \if_num:w \tex_ifnum:D
3200 \cs_new_eq:NN \if_int_odd:w \tex_ifodd:D
3201 \cs_new_eq:NN \if_case:w \tex_ifcase:D

```

(End definition for `\int_value:w`. This function is documented on page 78.)

174.1 Integer expressions

`\int_eval:n` Wrapper for `\int_eval:w`. Can be used in an integer expression or directly in the input stream. In format mode, there is already a definition in `l3alloc` for bookstrapping, which is therefore corrected to the “real” version here.

```

3202 <*initex>
3203 \cs_set:Npn \int_eval:n #1 { \int_value:w \int_eval:w #1 \int_eval_end: }
3204 </initex>
3205 <*package>
3206 \cs_new:Npn \int_eval:n #1 { \int_value:w \int_eval:w #1 \int_eval_end: }
3207 </package>

```

(End definition for `\int_eval:n`. This function is documented on page 66.)

`\int_max:nn` Functions for min, max, and absolute value.

`\int_min:nn`

`\int_abs:n`

```

3208 \cs_new:Npn \int_abs:n #1
3209 {
3210   \int_value:w
3211   \if_int_compare:w \int_eval:w #1 < \c_zero
3212   -
3213   \fi:
3214   \int_eval:w #1 \int_eval_end:
3215 }
3216 \cs_new:Npn \int_max:nn #1#2
3217 {
3218   \int_value:w \int_eval:w
3219   \if_int_compare:w
3220     \int_eval:w #1 > \int_eval:w #2 \int_eval_end:
3221     #1
3222   \else:
3223     #2
3224   \fi:
3225   \int_eval_end:
3226 }
3227 \cs_new:Npn \int_min:nn #1#2
3228 {
3229   \int_value:w \int_eval:w
3230   \if_int_compare:w
3231     \int_eval:w #1 < \int_eval:w #2 \int_eval_end:
3232     #1
3233   \else:
3234     #2
3235   \fi:
3236   \int_eval_end:
3237 }

```

(End definition for `\int_max:nn`. This function is documented on page 67.)

`\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.

```

3238 \cs_new:Npn \int_div_truncate:nn #1#2
3239 {
3240   \int_value:w \int_eval:w
3241   \if_int_compare:w \int_eval:w #1 = \c_zero
3242   0
3243   \else:
3244     ( #1 % )
3245   \if_int_compare:w \int_eval:w #1 < \c_zero
3246     \if_int_compare:w \int_eval:w #2 < \c_zero

```

```

3247         - ( #2 + % )
3248     \else:
3249         + ( #2 - % )
3250     \fi:
3251 \else:
3252     \if_int_compare:w \int_eval:w #2 < \c_zero
3253         + ( #2 + % )
3254     \else:
3255         - ( #2 - % )
3256     \fi:
3257     \fi: % ( (
3258         1 ) / 2 )
3259     \fi:
3260     / ( #2 )
3261 \int_eval_end:
3262 }

```

For the sake of completeness:

```

3263 \cs_new:Npn \int_div_round:nn #1#2 { \int_eval:n { ( #1 ) / ( #2 ) } }

```

Finally there's the modulus operation.

```

3264 \cs_new:Npn \int_mod:nn #1#2
3265 {
3266     \int_value:w \int_eval:w
3267     #1 - \int_div_truncate:nn {#1} {#2} * ( #2 )
3268     \int_eval_end:
3269 }

```

(End definition for `\int_div_truncate:nn`. This function is documented on page 67.)

174.2 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`

```

3270 <*package>
3271 \cs_new_protected_nopar:Npn \int_new:N #1
3272 {
3273     \chk_if_free_cs:N #1
3274     \newcount #1
3275 }
3276 </package>
3277 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N` and `\int_new:c`. These functions are documented on page 68.)

`\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.

```

3278 \cs_new_protected_nopar:Npn \int_const:Nn #1#2
3279 {
3280   \int_compare:nNnTF {#2} > \c_minus_one
3281   {
3282     \int_compare:nNnTF {#2} > \c_max_register_int
3283     {
3284       \int_new:N #1
3285       \int_gset:Nn #1 {#2}
3286     }
3287     {
3288       \chk_if_free_cs:N #1
3289       \pref_global:D \tex_mathchardef:D #1 =
3290       \int_eval:w #2 \int_eval_end:
3291     }
3292   }
3293   {
3294     \int_new:N #1
3295     \int_gset:Nn #1 {#2}
3296   }
3297 }
3298 \cs_generate_variant:Nn \int_const:Nn { c }

```

(End definition for `\int_const:Nn` and `\int_const:cn`. These functions are documented on page 68.)

`\int_zero:N` Functions that reset an *integer* register to zero.

`\int_zero:c`
`\int_gzero:N`
`\int_gzero:c`

```

3299 \cs_new_protected_nopar:Npn \int_zero:N #1 { #1 = \c_zero }
3300 \cs_new_protected_nopar:Npn \int_gzero:N #1 { \pref_global:D #1 = \c_zero }
3301 \cs_generate_variant:Nn \int_zero:N { c }
3302 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End definition for `\int_zero:N` and `\int_zero:c`. These functions are documented on page 68.)

`\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`

```

3303 \cs_new_protected_nopar:Npn \int_set_eq:NN #1#2 { #1 = #2 }
3304 \cs_generate_variant:Nn \int_set_eq:NN { c }
3305 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
3306 \cs_new_protected_nopar:Npn \int_gset_eq:NN #1#2 { \pref_global:D #1 = #2 }
3307 \cs_generate_variant:Nn \int_gset_eq:NN { c }
3308 \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 68.)

174.3 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`

```

3309 \cs_new_protected_nopar:Npn \int_add:Nn #1#2

```

```

3310 { \tex_advance:D #1 by \int_eval:w #2 \int_eval_end: }
3311 \cs_new_nopar:Npn \int_sub:Nn #1#2
3312 { \tex_advance:D #1 by - \int_eval:w #2 \int_eval_end: }
3313 \cs_new_protected_nopar:Npn \int_gadd:Nn
3314 { \pref_global:D \int_add:Nn }
3315 \cs_new_protected_nopar:Npn \int_gsub:Nn
3316 { \pref_global:D \int_sub:Nn }
3317 \cs_generate_variant:Nn \int_add:Nn { c }
3318 \cs_generate_variant:Nn \int_gadd:Nn { c }
3319 \cs_generate_variant:Nn \int_sub:Nn { c }
3320 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and `\int_add:cn`. These functions are documented on page 70.)

`\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
3321 \cs_new_protected_nopar:Npn \int_incr:N #1
3322 { \tex_advance:D #1 \c_one }
3323 \cs_new_protected_nopar:Npn \int_decr:N #1
3324 { \tex_advance:D #1 \c_minus_one }
3325 \cs_new_protected_nopar:Npn \int_gincr:N
3326 { \pref_global:D \int_incr:N }
3327 \cs_new_protected_nopar:Npn \int_gdecr:N
3328 { \pref_global:D \int_decr:N }
3329 \cs_generate_variant:Nn \int_incr:N { c }
3330 \cs_generate_variant:Nn \int_decr:N { c }
3331 \cs_generate_variant:Nn \int_gincr:N { c }
3332 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and `\int_incr:c`. These functions are documented on page 69.)

`\int_set:Nn` As integers are register-based T_EX 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
3333 \cs_new_protected_nopar:Npn \int_set:Nn #1#2
3334 { #1 ~ \int_eval:w #2\int_eval_end: }
3335 \cs_new_protected_nopar:Npn \int_gset:Nn { \pref_global:D \int_set:Nn }
3336 \cs_generate_variant:Nn \int_set:Nn { c }
3337 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_set:cn`. These functions are documented on page 69.)

174.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

\int_use:c
3338 \cs_new_eq:NN \int_use:N \tex_the:D
3339 \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 70.)

174.5 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.

```

3340 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3341 { \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.

```

3342 \cs_new:Npn \int_compare_aux:nw #1#2 \q_stop
3343 {
3344   \exp_after:wN \int_compare_aux:Nw
3345   \int_to_roman:w
3346   \if:w #1 -
3347   \else:
3348     -
3349   \fi:
3350   #1#2 \q_mark #1#2 \q_stop
3351 }

```

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.

```

3352 \cs_new:Npn \int_compare_aux:Nw #1#2#3 \q_mark
3353 { \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:

```

3354 \cs_new:cpn { int_compare_=:w } #1 = #2 \q_stop
3355 {
3356   \if_int_compare:w #1 = \int_eval:w #2 \int_eval_end:

```

```

3357     \prg_return_true:
3358 \else:
3359     \prg_return_false:
3360 \fi:
3361 }

```

So is the one using == we just have to use == in the parameter text.

```

3362 \cs_new:cpn { int_compare_==:w } #1 == #2 \q_stop
3363 {
3364     \if_int_compare:w #1 = \int_eval:w #2 \int_eval_end:
3365         \prg_return_true:
3366     \else:
3367         \prg_return_false:
3368     \fi:
3369 }

```

Not equal is just about reversing the truth value.

```

3370 \cs_new:cpn { int_compare_!=:w } #1 != #2 \q_stop
3371 {
3372     \if_int_compare:w #1 = \int_eval:w #2 \int_eval_end:
3373         \prg_return_false:
3374     \else:
3375         \prg_return_true:
3376     \fi:
3377 }

```

Less than and greater than are also straight forward.

```

3378 \cs_new:cpn { int_compare_<:w } #1 < #2 \q_stop
3379 {
3380     \if_int_compare:w #1 < \int_eval:w #2 \int_eval_end:
3381         \prg_return_true:
3382     \else:
3383         \prg_return_false:
3384     \fi:
3385 }
3386 \cs_new:cpn { int_compare_>:w } #1 > #2 \q_stop
3387 {
3388     \if_int_compare:w #1 > \int_eval:w #2 \int_eval_end:
3389         \prg_return_true:
3390     \else:
3391         \prg_return_false:
3392     \fi:
3393 }

```

The less than or equal operation is just the opposite of the greater than operation. *Vice versa* for less than or equal.

```

3394 \cs_new:cpn { int_compare_<=:w } #1 <= #2 \q_stop

```

```

3395 {
3396   \if_int_compare:w #1 > \int_eval:w #2 \int_eval_end:
3397   \prg_return_false:
3398   \else:
3399     \prg_return_true:
3400   \fi:
3401 }
3402 \cs_new:cpn { int_compare_>=:w } #1 >= #2 \q_stop
3403 {
3404   \if_int_compare:w #1 < \int_eval:w #2 \int_eval_end:
3405   \prg_return_false:
3406   \else:
3407     \prg_return_true:
3408   \fi:
3409 }

```

(End definition for `\int_compare:n`. These functions are documented on page 70.)

`\int_compare_p:nNn` More efficient but less natural in typing.

`\int_compare:nNnTF`

```

3410 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF}
3411 {
3412   \if_int_compare:w \int_eval:w #1 #2 \int_eval:w #3 \int_eval_end:
3413   \prg_return_true:
3414   \else:
3415     \prg_return_false:
3416   \fi:
3417 }

```

(End definition for `\int_compare:nNn`. These functions are documented on page 70.)

`\int_if_odd_p:n` A predicate function.

`\int_if_odd:nTF`

`\int_if_even_p:n`

`\int_if_even:nTF`

```

3418 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
3419 {
3420   \if_int_odd:w \int_eval:w #1 \int_eval_end:
3421   \prg_return_true:
3422   \else:
3423     \prg_return_false:
3424   \fi:
3425 }
3426 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3427 {
3428   \if_int_odd:w \int_eval:w #1 \int_eval_end:
3429   \prg_return_false:
3430   \else:
3431     \prg_return_true:
3432   \fi:
3433 }

```

(End definition for `\int_if_odd:n`. These functions are documented on page 71.)

174.6 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

3434 \cs_new:Npn \int_while_do:nn #1#2
3435 {
3436   \int_compare:nT {#1}
3437   {
3438     #2
3439     \int_while_do:nn {#1} {#2}
3440   }
3441 }
3442 \cs_new:Npn \int_until_do:nn #1#2
3443 {
3444   \int_compare:nF {#1}
3445   {
3446     #2
3447     \int_until_do:nn {#1} {#2}
3448   }
3449 }
3450 \cs_new:Npn \int_do_while:nn #1#2
3451 {
3452   #2
3453   \int_compare:nT {#1}
3454   { \int_do_while:nn {#1} {#2} }
3455 }
3456 \cs_new:Npn \int_do_until:nn #1#2
3457 {
3458   #2
3459   \int_compare:nF {#1}
3460   { \int_do_until:nn {#1} {#2} }
3461 }
```

(End definition for `\int_while_do:nn`. This function is documented on page 72.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```
\int_until_do:nNnn
\int_do_while:nNnn
\int_do_until:nNnn

3462 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3463 {
3464   \int_compare:nNnT {#1} #2 {#3}
3465   {
3466     #4
3467     \int_while_do:nNnn {#1} #2 {#3} {#4}
3468   }
3469 }
3470 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3471 {
3472   \int_compare:nNnF {#1} #2 {#3}
3473   {
```

```

3474     #4
3475     \int_until_do:nNnn {#1} #2 {#3} {#4}
3476   }
3477 }
3478 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3479 {
3480   #4
3481   \int_compare:nNnT {#1} #2 {#3}
3482   { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3483 }
3484 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3485 {
3486   #4
3487   \int_compare:nNnF {#1} #2 {#3}
3488   { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3489 }

```

(End definition for `\int_while_do:nNnn`. This function is documented on page 71.)

174.7 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

3490 \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 72.)

`\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".

```

3491 \cs_new_nopar:Npn \int_to_symbols:nnn #1#2#3
3492 {
3493   \int_compare:nNnTF {#1} > {#2}
3494   {
3495     \exp_args:Nf \int_to_symbols:nnn
3496     { \int_div_truncate:nn { #1 - 1 } {#2} } {#2} {#3}
3497     \exp_args:Nf \prg_case_int:nnn
3498     { \int_eval:n { 1 + \int_mod:nn { #1 - 1 } {#2} } }
3499     {#3} { }
3500   }
3501   { \exp_args:Nf \prg_case_int:nnn { \int_eval:n {#1} } {#3} { } }
3502 }

```

(End definition for `\int_to_symbols:nnn`. This function is documented on page 73.)

`\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.

```

3503 \cs_new:Npn \int_to_alph:n #1
3504 {
3505   \int_to_symbols:nnn {#1} { 26 }
3506   {
3507     { 1 } { a }
3508     { 2 } { b }
3509     { 3 } { c }
3510     { 4 } { d }
3511     { 5 } { e }
3512     { 6 } { f }
3513     { 7 } { g }
3514     { 8 } { h }
3515     { 9 } { i }
3516     { 10 } { j }
3517     { 11 } { k }
3518     { 12 } { l }
3519     { 13 } { m }
3520     { 14 } { n }
3521     { 15 } { o }
3522     { 16 } { p }
3523     { 17 } { q }
3524     { 18 } { r }
3525     { 19 } { s }
3526     { 20 } { t }
3527     { 21 } { u }
3528     { 22 } { v }
3529     { 23 } { w }
3530     { 24 } { x }
3531     { 25 } { y }
3532     { 26 } { z }
3533   }
3534 }
3535 \cs_new:Npn \int_to_Alph:n #1
3536 {
3537   \int_to_symbols:nnn {#1} { 26 }
3538   {
3539     { 1 } { A }
3540     { 2 } { B }
3541     { 3 } { C }
3542     { 4 } { D }
3543     { 5 } { E }
3544     { 6 } { F }
3545     { 7 } { G }
3546     { 8 } { H }
3547     { 9 } { I }

```

```

3548         { 10 } { J }
3549         { 11 } { K }
3550         { 12 } { L }
3551         { 13 } { M }
3552         { 14 } { N }
3553         { 15 } { O }
3554         { 16 } { P }
3555         { 17 } { Q }
3556         { 18 } { R }
3557         { 19 } { S }
3558         { 20 } { T }
3559         { 21 } { U }
3560         { 22 } { V }
3561         { 23 } { W }
3562         { 24 } { X }
3563         { 25 } { Y }
3564         { 26 } { Z }
3565     }
3566 }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 72.)

```

\int_to_base:nn
\int_to_base_aux_i:nn
\int_to_base_aux_ii:nnN
\int_to_base_aux_iii:nnnN
\int_to_letter:n
3567 \cs_new:Npn \int_to_base:nn #1
3568 { \exp_args:Nf \int_to_base_aux_i:nn { \int_eval:n {#1} } }
3569 \cs_new:Npn \int_to_base_aux_i:nn #1#2
3570 {
3571     \int_compare:nNnTF {#1} < \c_zero
3572     { \exp_args:No \int_to_base_aux_ii:nnN { \use_none:n #1 } {#2} - }
3573     { \int_to_base_aux_ii:nnN {#1} {#2} \c_empty_tl }
3574 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

3575 \cs_new:Npn \int_to_base_aux_ii:nnN #1#2#3
3576 {
3577     \int_compare:nNnTF {#1} < {#2}
3578     { \exp_last_unbraced:Nf #3 { \int_to_letter:n {#1} } }
3579     {
3580         \exp_args:Nf \int_to_base_aux_iii:nnnN
3581         { \int_to_letter:n { \int_mod:nn {#1} {#2} } }
3582         {#1}

```

```

3583         {#2}
3584         #3
3585     }
3586 }
3587 \cs_new:Npn \int_to_base_aux_iii:nnnN #1#2#3#4
3588 {
3589     \exp_args:Nf \int_to_base_aux_ii:nnN
3590     { \int_div_truncate:nn {#2} {#3} }
3591     {#3}
3592     #4
3593     #1
3594 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\prg_case_int:nnn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional.

```

3595 \cs_new:Npn \int_to_letter:n #1
3596 {
3597     \exp_after:wN \exp_after:wN
3598     \if_case:w \int_eval:w #1 - \c_ten \int_eval_end:
3599     A
3600     \or: B
3601     \or: C
3602     \or: D
3603     \or: E
3604     \or: F
3605     \or: G
3606     \or: H
3607     \or: I
3608     \or: J
3609     \or: K
3610     \or: L
3611     \or: M
3612     \or: N
3613     \or: O
3614     \or: P
3615     \or: Q
3616     \or: R
3617     \or: S
3618     \or: T
3619     \or: U
3620     \or: V
3621     \or: W
3622     \or: X
3623     \or: Y
3624     \or: Z
3625     \else: #1

```

```

3626     \fi:
3627 }

```

(End definition for `\int_to_base:nn`. This function is documented on page 77.)

```

\int_to_binary:n
\int_to_hexadecimal:n
\int_to_octal:n

```

Wrappers around the generic function.

```

3628 \cs_new:Npn \int_to_binary:n #1
3629 { \int_to_base:nn {#1} { 2 } }
3630 \cs_new:Npn \int_to_hexadecimal:n #1
3631 { \int_to_base:nn {#1} { 16 } }
3632 \cs_new:Npn \int_to_octal:n #1
3633 { \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 74.)

```

\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.

```

3634 \cs_new_nopar:Npn \int_to_roman:n #1
3635 {
3636   \exp_after:wN \int_to_roman_aux:N
3637   \int_to_roman:w \int_eval:n {#1} Q
3638 }
3639 \cs_new_nopar:Npn \int_to_roman_aux:N #1
3640 {
3641   \use:c { int_to_roman_ #1 :w }
3642   \int_to_roman_aux:N
3643 }
3644 \cs_new_nopar:Npn \int_to_Roman:n #1
3645 {
3646   \exp_after:wN \int_to_Roman_aux:N
3647   \int_to_roman:w \int_eval:n {#1} Q
3648 }
3649 \cs_new_nopar:Npn \int_to_Roman_aux:N #1
3650 {
3651   \use:c { int_to_Roman_ #1 :w }
3652   \int_to_Roman_aux:N
3653 }
3654 \cs_new_nopar:Npn \int_to_roman_i:w { i }
3655 \cs_new_nopar:Npn \int_to_roman_v:w { v }
3656 \cs_new_nopar:Npn \int_to_roman_x:w { x }
3657 \cs_new_nopar:Npn \int_to_roman_l:w { l }
3658 \cs_new_nopar:Npn \int_to_roman_c:w { c }
3659 \cs_new_nopar:Npn \int_to_roman_d:w { d }
3660 \cs_new_nopar:Npn \int_to_roman_m:w { m }
3661 \cs_new_nopar:Npn \int_to_roman_Q:w #1 { }

```

```

3662 \cs_new_nopar:Npn \int_to_Roman_i:w { I }
3663 \cs_new_nopar:Npn \int_to_Roman_v:w { V }
3664 \cs_new_nopar:Npn \int_to_Roman_x:w { X }
3665 \cs_new_nopar:Npn \int_to_Roman_l:w { L }
3666 \cs_new_nopar:Npn \int_to_Roman_c:w { C }
3667 \cs_new_nopar:Npn \int_to_Roman_d:w { D }
3668 \cs_new_nopar:Npn \int_to_Roman_m:w { M }
3669 \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 74.)

174.8 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
`\int_get_sign_and_digits_aux:nNNN` of the input. The sign of the input is tracked by the first Boolean used by the auxiliary
`\int_get_sign_and_digits_aux:oNNN` function.

```

3670 \cs_new:Npn \int_get_sign:n #1
3671 {
3672   \int_get_sign_and_digits_aux:nNNN {#1}
3673   \c_true_bool \c_true_bool \c_false_bool
3674 }
3675 \cs_new:Npn \int_get_digits:n #1
3676 {
3677   \int_get_sign_and_digits_aux:nNNN {#1}
3678   \c_true_bool \c_false_bool \c_true_bool
3679 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3680 \cs_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4
3681 {
3682   \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} -
3683   {
3684     \bool_if:NTF #2
3685     {
3686       \int_get_sign_and_digits_aux:oNNN
3687       { \use_none:n #1 } \c_false_bool #3#4
3688     }
3689     {
3690       \int_get_sign_and_digits_aux:oNNN
3691       { \use_none:n #1 } \c_true_bool #3#4
3692     }
3693   }
3694   {
3695     \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} +

```

```

3696         { \int_get_sign_and_digits_aux:oNNN { \use_none:n #1 } #2#3#4 }
3697         {
3698             \bool_if:NT #3 { \bool_if:NF #2 - }
3699             \bool_if:NT #4 {#1}
3700         }
3701     }
3702 }
3703 \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 77.)

`\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.

```

3704 \cs_new:Npn \int_from_alph:n #1
3705 {
3706     \int_eval:n
3707     {
3708         \int_get_sign:n {#1}
3709         \exp_args:Nf \int_from_alph_aux:n { \int_get_digits:n {#1} }
3710     }
3711 }
3712 \cs_new:Npn \int_from_alph_aux:n #1
3713 { \int_from_alph_aux:nN { 0 } #1 \q_nil }
3714 \cs_new:Npn \int_from_alph_aux:nN #1#2
3715 {
3716     \quark_if_nil:NTF #2
3717     {#1}
3718     {
3719         \exp_args:Nf \int_from_alph_aux:nN
3720         { \int_eval:n { #1 * 26 + \int_from_alph_aux:N #2 } }
3721     }
3722 }
3723 \cs_new:Npn \int_from_alph_aux:N #1
3724 { \int_eval:n { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } } }

```

(End definition for `\int_from_alph:n`. This function is documented on page 75.)

`\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.

```

3725 \cs_new:Npn \int_from_base:nn #1#2
3726 {
3727     \int_eval:n
3728     {
3729         \int_get_sign:n {#1}
3730         \exp_args:Nf \int_from_base_aux:nn
3731         { \int_get_digits:n {#1} } {#2}
3732     }

```

```

3733 }
3734 \cs_new:Npn \int_from_base_aux:nn #1#2
3735 { \int_from_base_aux:nnN { 0 } { #2 } #1 \q_nil }
3736 \cs_new:Npn \int_from_base_aux:nnN #1#2#3
3737 {
3738   \quark_if_nil:NTF #3
3739   {#1}
3740   {
3741     \exp_args:Nf \int_from_base_aux:nnN
3742     { \int_eval:n { #1 * #2 + \int_from_base_aux:N #3 } }
3743     {#2}
3744   }
3745 }

```

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.

```

3746 \cs_new:Npn \int_from_base_aux:N #1
3747 {
3748   \int_compare:nNnTF { '#1 } < { 58 }
3749   {#1}
3750   {
3751     \int_eval:n
3752     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3753   }
3754 }

```

(End definition for `\int_from_base:nn`. This function is documented on page 75.)

`\int_from_binary:n`
`\int_from_hexadecimal:n`
`\int_from_octal:n`

Wrappers around the generic function.

```

3755 \cs_new:Npn \int_from_binary:n #1
3756 { \int_from_base:nn {#1} \c_two }
3757 \cs_new:Npn \int_from_hexadecimal:n #1
3758 { \int_from_base:nn {#1} \c_sixteen }
3759 \cs_new:Npn \int_from_octal:n #1
3760 { \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 75.)

[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 num-
erals to integers.

```

3761 \int_const:cn { c_int_from_roman_i_int } { 1 }
3762 \int_const:cn { c_int_from_roman_v_int } { 5 }

```

```

3763 \int_const:cn { c_int_from_roman_x_int } { 10 }
3764 \int_const:cn { c_int_from_roman_l_int } { 50 }
3765 \int_const:cn { c_int_from_roman_c_int } { 100 }
3766 \int_const:cn { c_int_from_roman_d_int } { 500 }
3767 \int_const:cn { c_int_from_roman_m_int } { 1000 }
3768 \int_const:cn { c_int_from_roman_I_int } { 1 }
3769 \int_const:cn { c_int_from_roman_V_int } { 5 }
3770 \int_const:cn { c_int_from_roman_X_int } { 10 }
3771 \int_const:cn { c_int_from_roman_L_int } { 50 }
3772 \int_const:cn { c_int_from_roman_C_int } { 100 }
3773 \int_const:cn { c_int_from_roman_D_int } { 500 }
3774 \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 .

```

3775 \cs_new_nopar:Npn \int_from_roman:n #1
3776 {
3777   \tl_if_blank:nF {#1}
3778   {
3779     \exp_after:wN \int_from_roman_end:w
3780     \int_value:w \int_eval:w
3781     \int_from_roman_aux:NN #1 Q \q_stop
3782   }
3783 }
3784 \cs_new_nopar:Npn \int_from_roman_aux:NN #1#2
3785 {
3786   \str_if_eq:nnTF {#1} { Q }
3787   {#1#2}
3788   {
3789     \str_if_eq:nnTF {#2} { Q }
3790     {
3791       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3792       { \int_from_roman_clean_up:w }
3793       +
3794       \use:c { c_int_from_roman_ #1 _int }
3795       #2
3796     }
3797     {
3798       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3799       { \int_from_roman_clean_up:w }
3800       \cs_if_exist:cF { c_int_from_roman_ #2 _int }
3801       { \int_from_roman_clean_up:w }
3802       \int_compare:nNnTF
3803       { \use:c { c_int_from_roman_ #1 _int } }
3804       <
3805       { \use:c { c_int_from_roman_ #2 _int } }
3806       {
3807         + \use:c { c_int_from_roman_ #2 _int }
3808         - \use:c { c_int_from_roman_ #1 _int }

```

```

3809             \int_from_roman_aux:NN
3810         }
3811     {
3812         + \use:c { c_int_from_roman_ #1 _int }
3813         \int_from_roman_aux:NN #2
3814     }
3815 }
3816 }
3817 }
3818 \cs_new_nopar:Npn \int_from_roman_end:w #1 Q #2 \q_stop
3819 { \tl_if_empty:nTF {#2} {#1} {#2} }
3820 \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 75.)

174.9 Viewing integer

```

\int_show:N
\int_show:c
3821 \cs_new_eq:NN \int_show:N \kernel_register_show:N
3822 \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 75.)

174.10 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_two`

`\c_three`

`\c_four`

`\c_five`

`\c_eight`

`\c_nine`

`\c_ten`

`\c_eleven`

`\c_thirteen`

`\c_fourteen`

`\c_fifteen`

```

3823 \int_const:Nn \c_one      { 1 }
3824 \int_const:Nn \c_two      { 2 }
3825 \int_const:Nn \c_three    { 3 }
3826 \int_const:Nn \c_four     { 4 }
3827 \int_const:Nn \c_five     { 5 }
3828 \int_const:Nn \c_eight    { 8 }
3829 \int_const:Nn \c_nine     { 9 }
3830 \int_const:Nn \c_ten      { 10 }
3831 \int_const:Nn \c_eleven   { 11 }
3832 \int_const:Nn \c_thirteen { 13 }
3833 \int_const:Nn \c_fourteen { 14 }
3834 \int_const:Nn \c_fifteen { 15 }

```

`\c_thirty_two` One middling value.

```
3835 \int_const:Nn \c_thirty_two { 32 }
```

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

`\c_two_hundred_fifty_six`

```
3836 \int_const:Nn \c_two_hundred_fifty_five { 255 }
```

```
3837 \int_const:Nn \c_two_hundred_fifty_six { 256 }
```

`\c_one_hundred` Simple runs of powers of ten.

`\c_one_thousand`

`\c_ten_thousand`

```
3838 \int_const:Nn \c_one_hundred { 100 }
```

```
3839 \int_const:Nn \c_one_thousand { 1000 }
```

```
3840 \int_const:Nn \c_ten_thousand { 10000 }
```

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
3841 \int_const:Nn \c_max_int { 2 147 483 647 }
```

174.11 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`

```
3842 \int_new:N \l_tmpa_int
```

```
3843 \int_new:N \l_tmpb_int
```

```
3844 \int_new:N \l_tmpc_int
```

```
3845 \int_new:N \g_tmpa_int
```

```
3846 \int_new:N \g_tmpb_int
```

174.12 Registers for earlier modules

Needed from other modules:

```
3847 \int_new:N \g_seq_nesting_depth_int
```

```
3848 \int_new:N \g_tl_inline_level_int
```

174.13 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\int_convert_from_base_ten:nn` Some simple renames.

`\int_convert_to_symbols:nnn`

`\int_convert_to_base_ten:nn`

```
3849 \cs_new_eq:NN \int_convert_from_base_ten:nn \int_to_base:nn
```

```
3850 \cs_new_eq:NN \int_convert_to_symbols:nnn \int_to_symbols:nnn
```

```
3851 \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
3852 \cs_new_nopar:Npn \int_to_symbol:n
3853 {
3854   \mode_if_math:TF
3855     { \int_to_symbol_math:n }
3856     { \int_to_symbol_text:n }
3857 }
3858 \cs_new:Npn \int_to_symbol_math:n #1
3859 {
3860   \int_to_symbols:nnn {#1} { 9 }
3861   {
3862     { 1 } { * }
3863     { 2 } { \dagger }
3864     { 3 } { \ddagger }
3865     { 4 } { \mathsection }
3866     { 5 } { \mathparagraph }
3867     { 6 } { \l }
3868     { 7 } { ** }
3869     { 8 } { \dagger \dagger }
3870     { 9 } { \ddagger \ddagger }
3871   }
3872 }
3873 \cs_new:Npn \int_to_symbol_text:n #1
3874 {
3875   \int_to_symbols:nnn {#1} { 9 }
3876   {
3877     { 1 } { \textasteriskcentered }
3878     { 2 } { \textdagger }
3879     { 3 } { \textdaggerdbl }
3880     { 4 } { \textsection }
3881     { 5 } { \textparagraph }
3882     { 6 } { \textbardbl }
3883     { 7 } { \textasteriskcentered \textasteriskcentered }
3884     { 8 } { \textdagger \textdagger }
3885     { 9 } { \textdaggerdbl \textdaggerdbl }
3886   }
3887 }
3888 \</initex | package>

```

(End definition for `\int_to_symbol:n`. This function is documented on page ??.)

175 l3skip implementation

```

3889 \<*initex | package>

```

```

3890 <*package>
3891 \ProvidesExplPackage
3892   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
3893 \package_check_loaded_expl:
3894 </package>

```

175.1 Length primitives renamed

`\if_dim:w` Primitives renamed.
`\dim_eval:w`
`\dim_eval_end:`

```

3895 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
3896 \cs_new_eq:NN \dim_eval:w \etex_dimexpr:D
3897 \cs_new_eq:NN \dim_eval_end: \tex_relax:D

```

(End definition for `\if_dim:w`. This function is documented on page 91.)

175.2 Creating and initialising dim variables

`\dim_new:N` Allocating $\langle dim \rangle$ registers ...
`\dim_new:c`

```

3898 <*package>
3899 \cs_new_protected_nopar:Npn \dim_new:N #1
3900 {
3901   \chk_if_free_cs:N #1
3902   \newdimen #1
3903 }
3904 </package>
3905 \cs_generate_variant:Nn \dim_new:N { c }

```

(End definition for `\dim_new:N` and `\dim_new:c`. These functions are documented on page 79.)

`\dim_zero:N` Reset the register to zero.

```

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

```

```

3906 \cs_new_protected_nopar:Npn \dim_zero:N #1 { #1 \c_zero_dim }
3907 \cs_new_protected_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N }
3908 \cs_generate_variant:Nn \dim_zero:N { c }
3909 \cs_generate_variant:Nn \dim_gzero:N { c }

```

(End definition for `\dim_zero:N` and `\dim_zero:c`. These functions are documented on page 79.)

175.3 Setting dim variables

`\dim_set:Nn` Setting dimensions is easy enough.

```

\dim_set:cn
\dim_gset:Nn
\dim_gset:cn

```

```

3910 \cs_new_protected_nopar:Npn \dim_set:Nn #1#2
3911 { #1 ~ \dim_eval:w #2 \dim_eval_end: }
3912 \cs_new_protected_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn }
3913 \cs_generate_variant:Nn \dim_set:Nn { c }
3914 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_set:cn`. These functions are documented on page 80.)

```

\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 80.)

```

\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

```

```

3921 \cs_new_protected_nopar:Npn \dim_set_max:Nn #1#2
3922 { \dim_compare:nNnT {#1} < {#2} { \dim_set:Nn #1 {#2} } }
3923 \cs_new_protected_nopar:Npn \dim_gset_max:Nn #1#2
3924 { \dim_compare:nNnT {#1} < {#2} { \dim_gset:Nn #1 {#2} } }
3925 \cs_new_protected_nopar:Npn \dim_set_min:Nn #1#2
3926 { \dim_compare:nNnT {#1} > {#2} { \dim_set:Nn #1 {#2} } }
3927 \cs_new_protected_nopar:Npn \dim_gset_min:Nn #1#2
3928 { \dim_compare:nNnT {#1} > {#2} { \dim_gset:Nn #1 {#2} } }
3929 \cs_generate_variant:Nn \dim_set_max:Nn { c }
3930 \cs_generate_variant:Nn \dim_gset_max:Nn { c }
3931 \cs_generate_variant:Nn \dim_set_min:Nn { c }
3932 \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 81.)

```

\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

```

```

3933 \cs_new_protected_nopar:Npn \dim_add:Nn #1#2
3934 { \tex_advance:D #1 by \dim_eval:w #2 \dim_eval_end: }
3935 \cs_new_protected_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
3936 \cs_generate_variant:Nn \dim_add:Nn { c }
3937 \cs_generate_variant:Nn \dim_gadd:Nn { c }
3938 \cs_new_protected_nopar:Npn \dim_sub:Nn #1#2
3939 { \tex_advance:D #1 by - \dim_eval:w #2 \dim_eval_end: }
3940 \cs_new_protected_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn }
3941 \cs_generate_variant:Nn \dim_sub:Nn { c }
3942 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and `\dim_add:cn`. These functions are documented on page 81.)

175.4 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.

```

3943 \cs_new_nopar:Npn \dim_ratio:nn #1#2
3944 { \dim_ratio_aux:n {#1} / \dim_ratio_aux:n {#2} }
3945 \cs_new_nopar:Npn \dim_ratio_aux:n #1
3946 { \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 81.)

175.5 Dimension expression conditionals

`\dim_compare_p:nNn`
`\dim_compare:nNnTF`

```

3947 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
3948 {
3949   \if_dim:w \dim_eval:w #1 #2 \dim_eval:w #3 \dim_eval_end:
3950   \prg_return_true: \else: \prg_return_false: \fi:
3951 }

```

(End definition for `\dim_compare_p:nNn`. This function is documented on page 82.)

`\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.

```

3952 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
3953 {
3954   \exp_after:wN \dim_compare_aux:wNN \dim_use:N \dim_eval:w #1
3955   \dim_eval_end:
3956   \prg_return_true:

```

```

3957     \else:
3958         \prg_return_false:
3959     \fi:
3960 }

```

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.

```

3961 \exp_args:Nno \use:nn
3962 { \cs_new:Npn \dim_compare_aux:wNN #1 }
3963 { \tl_to_str:n { pt } }
3964 #2 #3
3965 {
3966     \use:c
3967     {
3968         dim_compare_ #2
3969         \if_meaning:w = #3 = \fi:
3970     :nw
3971     }
3972     { #1 pt } #3
3973 }

```

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.

```

3974 \cs_new:cpn { dim_compare_<:nw } #1 { \if_dim:w #1 < \dim_eval:w }
3975 \cs_new:cpn { dim_compare_=:nw } #1 { \if_dim:w #1 = \dim_eval:w }
3976 \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 `=`.

```

3977 \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.

```

3978 \cs_new:cpn {dim_compare_<=:nw} #1#2 {\reverse_if:N \if_dim:w #1 > \dim_eval:w}
3979 \cs_new:cpn {dim_compare_!:=:nw} #1#2 {\reverse_if:N \if_dim:w #1 = \dim_eval:w}
3980 \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 82.)

175.6 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
3981 \cs_set:Npn \dim_while_do:nn #1#2
3982 {
3983   \dim_compare:nT {#1}
3984   {
3985     #2
3986     \dim_while_do:nn {#1} {#2}
3987   }
3988 }
3989 \cs_set:Npn \dim_until_do:nn #1#2
3990 {
3991   \dim_compare:nF {#1}
3992   {
3993     #2
3994     \dim_until_do:nn {#1} {#2}
3995   }
3996 }
3997 \cs_set:Npn \dim_do_while:nn #1#2
3998 {
3999   #2
4000   \dim_compare:nT {#1}
4001   { \dim_do_while:nn {#1} {#2} }
4002 }
4003 \cs_set:Npn \dim_do_until:nn #1#2
4004 {
4005   #2
4006   \dim_compare:nF {#1}
4007   { \dim_do_until:nn {#1} {#2} }
4008 }

```

(End definition for `\dim_while_do:nn`. This function is documented on page 83.)

`\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
4009 \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
4010 {
4011   \dim_compare:nNnT {#1} #2 {#3}
4012   {

```

```

4013         #4
4014         \dim_while_do:nNnn {#1} #2 {#3} {#4}
4015     }
4016 }
4017 \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
4018 {
4019     \dim_compare:nNnF {#1} #2 {#3}
4020     {
4021         #4
4022         \dim_until_do:nNnn {#1} #2 {#3} {#4}
4023     }
4024 }
4025 \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4026 {
4027     #4
4028     \dim_compare:nNnT {#1} #2 {#3}
4029     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4030 }
4031 \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4032 {
4033     #4
4034     \dim_compare:nNnF {#1} #2 {#3}
4035     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4036 }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 82.)

175.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4037 \cs_new_nopar:Npn \dim_eval:n #1
4038 { \dim_use:N \dim_eval:w #1 \dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 84.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

```

\dim_use:c
4039 \cs_new_eq:NN \dim_use:N \tex_the:D
4040 \cs_generate_variant:Nn \dim_use:N { c }

```

(End definition for `\dim_use:N` and `\dim_use:c`. These functions are documented on page 84.)

175.8 Viewing dim variables

`\dim_show:N` Diagnostics.

```

\dim_show:c
4041 \cs_new_eq:NN \dim_show:N \kernel_register_show:N
4042 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N` and `\dim_show:c`. These functions are documented on page 84.)

175.9 Constant dimensions

`\c_zero_dim` The source for these depends on whether we are in package mode.
`\c_max_dim`

```

4043 \*initex>
4044 \dim_new:N \c_zero_dim
4045 \dim_new:N \c_max_dim
4046 \dim_set:Nn \c_max_dim { 16383.99999 pt }
4047 \*initex>
4048 \*package>
4049 \cs_new_eq:NN \c_zero_dim \z@
4050 \cs_new_eq:NN \c_max_dim \maxdimen
4051 \*package>

```

175.10 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`

```

4052 \dim_new:N \l_tmpa_dim
4053 \dim_new:N \l_tmpb_dim
4054 \dim_new:N \l_tmpc_dim
4055 \dim_new:N \g_tmpa_dim
4056 \dim_new:N \g_tmpb_dim

```

175.11 Creating and initialising skip variables

`\skip_new:N` Allocation of a new internal registers.
`\skip_new:c`

```

4057 \*package>
4058 \cs_new_protected_nopar:Npn \skip_new:N #1
4059 {
4060   \chk_if_free_cs:N #1
4061   \newskip #1
4062 }
4063 \*package>
4064 \cs_generate_variant:Nn \skip_new:N { c }

```

(End definition for `\skip_new:N` and `\skip_new:c`. These functions are documented on page 85.)

`\skip_zero:N` Reset the register to zero.
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c`

```

4065 \cs_new_protected_nopar:Npn \skip_zero:N #1 { #1 \c_zero_skip }
4066 \cs_new_protected_nopar:Npn \skip_gzero:N { \pref_global:D \skip_zero:N }
4067 \cs_generate_variant:Nn \skip_zero:N { c }
4068 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End definition for `\skip_zero:N` and `\skip_zero:c`. These functions are documented on page 85.)

175.12 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn`

```

4069 \cs_new_protected_nopar:Npn \skip_set:Nn #1#2
4070 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
4071 \cs_new_protected_nopar:Npn \skip_gset:Nn { \pref_global:D \skip_set:Nn }
4072 \cs_generate_variant:Nn \skip_set:Nn { c }
4073 \cs_generate_variant:Nn \skip_gset:Nn { c }

```

(End definition for `\skip_set:Nn` and `\skip_set:cn`. These functions are documented on page 86.)

`\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`

```

4074 \cs_new_protected_nopar:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
4075 \cs_generate_variant:Nn \skip_set_eq:NN { c }
4076 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
4077 \cs_new_protected_nopar:Npn \skip_gset_eq:NN #1#2 { \pref_global:D #1 = #2 }
4078 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
4079 \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 86.)

`\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`

```

4080 \cs_new_protected_nopar:Npn \skip_add:Nn #1#2
4081 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
4082 \cs_new_protected_nopar:Npn \skip_gadd:Nn { \pref_global:D \skip_add:Nn }
4083 \cs_generate_variant:Nn \skip_add:Nn { c }
4084 \cs_generate_variant:Nn \skip_gadd:Nn { c }
4085 \cs_new_protected_nopar:Npn \skip_sub:Nn #1#2
4086 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4087 \cs_new_protected_nopar:Npn \skip_gsub:Nn { \pref_global:D \skip_sub:Nn }
4088 \cs_generate_variant:Nn \skip_sub:Nn { c }
4089 \cs_generate_variant:Nn \skip_gsub:Nn { c }

```

(End definition for `\skip_add:Nn` and `\skip_add:cn`. These functions are documented on page 86.)

175.13 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.

```

4090 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4091 {
4092   \if_int_compare:w
4093     \pdfTeX_strcmp:D { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4094     = \c_zero
4095     \prg_return_true:

```

```

4096     \else:
4097         \prg_return_false:
4098     \fi:
4099 }

```

(End definition for `\skip_if_eq:nm`. These functions are documented on page 87.)

`\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.

```

4100 \prg_new_conditional:Npnn \skip_if_infinite_glue:n #1 { p , T , F , TF }
4101 {
4102     \bool_if:nTF
4103     {
4104         \int_compare_p:nNn { \etex_gluestretchorder:D #1 } > \c_zero ||
4105         \int_compare_p:nNn { \etex_glueshrinkorder:D #1 } > \c_zero
4106     }
4107     { \prg_return_true: }
4108     { \prg_return_false: }
4109 }

```

(End definition for `\skip_if_infinite_glue:n`. These functions are documented on page 87.)

175.14 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

4110 \cs_new_nopar:Npn \skip_eval:n #1
4111 { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }

```

(End definition for `\skip_eval:n`. This function is documented on page 87.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

`\skip_use:c`

```

4112 \cs_new_eq:NN \skip_use:N \tex_the:D
4113 \cs_generate_variant:Nn \skip_use:N { c }

```

(End definition for `\skip_use:N` and `\skip_use:c`. These functions are documented on page 87.)

175.15 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`

```

4114 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
4115 \cs_new_nopar:Npn \skip_horizontal:n #1

```

```

4116 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
4117 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
4118 \cs_new_nopar:Npn \skip_vertical:n #1
4119 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4120 \cs_generate_variant:Nn \skip_horizontal:N { c }
4121 \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 91.)

175.16 Viewing skip variables

`\skip_show:N` Diagnostics.
`\skip_show:c`

```

4122 \cs_new_eq:NN \skip_show:N \kernel_register_show:N
4123 \cs_generate_variant:Nn \skip_show:N { c }

```

(End definition for `\skip_show:N` and `\skip_show:c`. These functions are documented on page 88.)

175.17 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions
`\c_max_skip`

```

4124 \cs_new_eq:NN \c_zero_skip \c_zero_dim
4125 \cs_new_eq:NN \c_max_skip \c_max_dim

```

(End definition for `\c_zero_skip`. This function is documented on page 88.)

175.18 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`

```

4126 \skip_new:N \l_tmpa_skip
4127 \skip_new:N \l_tmpb_skip
4128 \skip_new:N \l_tmpc_skip
4129 \skip_new:N \g_tmpa_skip
4130 \skip_new:N \g_tmpb_skip

```

175.19 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.
`\muskip_new:c`

```

4131 <*package>
4132 \cs_new_protected_nopar:Npn \muskip_new:N #1
4133 {
4134   \chk_if_free_cs:N #1
4135   \newmuskip #1

```

```

4136 }
4137 </package>
4138 \cs_generate_variant:Nn \muskip_new:N { c }

```

(End definition for `\muskip_new:N` and `\muskip_new:c`. These functions are documented on page 88.)

`\muskip_zero:N` Reset the register to zero.

```

\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c
4139 \cs_new_protected_nopar:Npn \muskip_zero:N #1
4140 { #1 \c_zero_muskip }
4141 \cs_new_protected_nopar:Npn \muskip_gzero:N { \pref_global:D \muskip_zero:N }
4142 \cs_generate_variant:Nn \muskip_zero:N { c }
4143 \cs_generate_variant:Nn \muskip_gzero:N { c }

```

(End definition for `\muskip_zero:N` and `\muskip_zero:c`. These functions are documented on page 89.)

175.20 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.

```

\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn
4144 \cs_new_protected_nopar:Npn \muskip_set:Nn #1#2
4145 { #1 ~ \etex_muexpr:D #2 \scan_stop: }
4146 \cs_new_protected_nopar:Npn \muskip_gset:Nn { \pref_global:D \muskip_set:Nn }
4147 \cs_generate_variant:Nn \muskip_set:Nn { c }
4148 \cs_generate_variant:Nn \muskip_gset:Nn { c }

```

(End definition for `\muskip_set:Nn` and `\muskip_set:cn`. These functions are documented on page 89.)

`\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
4149 \cs_new_protected_nopar:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
4150 \cs_generate_variant:Nn \muskip_set_eq:NN { c }
4151 \cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
4152 \cs_new_protected_nopar:Npn \muskip_gset_eq:NN #1#2 { \pref_global:D #1 = #2 }
4153 \cs_generate_variant:Nn \muskip_gset_eq:NN { c }
4154 \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 90.)

`\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
4155 \cs_new_protected_nopar:Npn \muskip_add:Nn #1#2
4156 { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
4157 \cs_new_protected_nopar:Npn \muskip_gadd:Nn { \pref_global:D \muskip_add:Nn }
4158 \cs_generate_variant:Nn \muskip_add:Nn { c }
4159 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
4160 \cs_new_protected_nopar:Npn \muskip_sub:Nn #1#2
4161 { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
4162 \cs_new_protected_nopar:Npn \muskip_gsub:Nn { \pref_global:D \muskip_sub:Nn }
4163 \cs_generate_variant:Nn \muskip_sub:Nn { c }
4164 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for `\muskip_add:Nn` and `\muskip_add:cn`. These functions are documented on page 90.)

175.21 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```
4165 \cs_new_nopar:Npn \muskip_eval:n #1
4166 { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }
```

(End definition for `\muskip_eval:n`. This function is documented on page 90.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

`\muskip_use:c`

```
4167 \cs_new_eq:NN \muskip_use:N \tex_the:D
4168 \cs_generate_variant:Nn \muskip_use:N { c }
```

(End definition for `\muskip_use:N` and `\muskip_use:c`. These functions are documented on page 90.)

175.22 Viewing muskip variables

`\muskip_show:N` Diagnostics.

`\muskip_show:c`

```
4169 \cs_new_eq:NN \muskip_show:N \kernel_register_show:N
4170 \cs_generate_variant:Nn \muskip_show:N { c }
```

(End definition for `\muskip_show:N` and `\muskip_show:c`. These functions are documented on page 91.)

175.23 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.

```
4171 \cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
4172 {
4173   \skip_if_infinite_glue:nTF {#1}
4174   {
4175     #3 = \c_zero_skip
4176     #4 = \c_zero_skip
4177     #2
4178   }
4179   {
4180     #3 = \etex_gluestretch:D #1 \scan_stop:
4181     #4 = \etex_glueshrink:D #1 \scan_stop:
4182   }
4183 }
```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 92.)

```
4184 </initex | package>
```

176 l3tl implementation

```
4185 <*initex | package>
4186 <*package>
4187 \ProvidesExplPackage
4188   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4189 \package_check_loaded_expl:
4190 </package>
```

A token list variable is a T_EX macro that holds tokens. By using the ε -T_EX primitive `\unexpanded` inside a T_EX `\edef` it is possible to store any tokens, including #, in this way.

176.1 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

```
4191 \cs_new_protected_nopar:Npn \tl_new:N #1
4192 {
4193   \chk_if_free_cs:N #1
4194   \cs_gset_eq:NN #1 \c_empty_tl
4195 }
4196 \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for \tl_new:N and \tl_new:c. These functions are documented on page 93.)

\tl_const:Nn Constants are also easy to generate.

```
\tl_const:Nx
\tl_const:cn
\tl_const:cx
4197 \cs_new_protected:Npn \tl_const:Nn #1#2
4198 {
4199   \chk_if_free_cs:N #1
4200   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4201 }
4202 \cs_new_protected:Npn \tl_const:Nx #1#2
4203 {
4204   \chk_if_free_cs:N #1
4205   \cs_gset_nopar:Npx #1 {#2}
4206 }
4207 \cs_generate_variant:Nn \tl_const:Nn { c }
4208 \cs_generate_variant:Nn \tl_const:Nx { c }
```

(End definition for \tl_const:Nn and others. These functions are documented on page 93.)

\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
4209 \cs_new_protected_nopar:Npn \tl_clear:N #1
4210 { \tl_set_eq:NN #1 \c_empty_tl }
```

```

4211 \cs_new_protected_nopar:Npn \tl_gclear:N #1
4212 { \tl_gset_eq:NN #1 \c_empty_tl }
4213 \cs_generate_variant:Nn \tl_clear:N { c }
4214 \cs_generate_variant:Nn \tl_gclear:N { c }

```

(End definition for `\tl_clear:N` and `\tl_clear:c`. These functions are documented on page 93.)

`\tl_clear_new:N` `\tl_clear_new:c` `\tl_gclear_new:N` `\tl_gclear_new:c` Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```

4215 \cs_new_protected_nopar:Npn \tl_clear_new:N #1
4216 { \cs_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4217 \cs_new_protected_nopar:Npn \tl_gclear_new:N #1
4218 { \cs_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4219 \cs_generate_variant:Nn \tl_clear_new:N { c }
4220 \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 93.)

`\tl_set_eq:NN` For setting token list variables equal to each other.

```

\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc
4221 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
4222 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
4223 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
4224 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
4225 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4226 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
4227 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
4228 \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 94.)

176.2 Adding to token list variables

`\tl_set:Nn` `\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` `\tl_gset:Nf` `\tl_gset:Nx` `\tl_gset:cn` `\tl_gset:NV` `\tl_gset:Nv` `\tl_gset:co` `\tl_gset:cf` 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.

```

4229 \cs_new_protected:Npn \tl_set:Nn #1#2
4230 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4231 \cs_new_protected:Npn \tl_set:No #1#2
4232 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4233 \cs_new_protected:Npn \tl_set:Nx #1#2
4234 { \cs_set_nopar:Npx #1 {#2} }
4235 \cs_new_protected:Npn \tl_gset:Nn #1#2
4236 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4237 \cs_new_protected:Npn \tl_gset:No #1#2
4238 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }

```

```

4239 \cs_new_protected:Npn \tl_gset:Nx #1#2
4240 { \cs_gset_nopar:Npx #1 {#2} }
4241 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
4242 \cs_generate_variant:Nn \tl_set:Nx { c }
4243 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
4244 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
4245 \cs_generate_variant:Nn \tl_gset:Nx { c }
4246 \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 94.)

`\tl_put_left:Nn`

Adding to the left is done directly to gain a little performance.

`\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`

```

4247 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4248 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4249 \cs_new_protected:Npn \tl_put_left:NV #1#2
4250 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4251 \cs_new_protected:Npn \tl_put_left:No #1#2
4252 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4253 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4254 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4255 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4256 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4257 \cs_new_protected:Npn \tl_gput_left:NV #1#2
4258 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4259 \cs_new_protected:Npn \tl_gput_left:No #1#2
4260 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4261 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4262 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4263 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4264 \cs_generate_variant:Nn \tl_put_left:NV { c }
4265 \cs_generate_variant:Nn \tl_put_left:No { c }
4266 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4267 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4268 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4269 \cs_generate_variant:Nn \tl_gput_left:No { c }
4270 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and others. These functions are documented on page 95.)

`\tl_put_right:Nn`

The same on the right.

`\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`

`\tl_gput_right:cV`

`\tl_gput_right:co`

`\tl_gput_right:cx`

```

4271 \cs_new_protected:Npn \tl_put_right:Nn #1#2
4272 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
4273 \cs_new_protected:Npn \tl_put_right:NV #1#2
4274 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
4275 \cs_new_protected:Npn \tl_put_right:No #1#2
4276 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
4277 \cs_new_protected:Npn \tl_put_right:Nx #1#2
4278 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }

```

```

4279 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
4280 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
4281 \cs_new_protected:Npn \tl_gput_right:NV #1#2
4282 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
4283 \cs_new_protected:Npn \tl_gput_right:No #1#2
4284 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
4285 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4286 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4287 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4288 \cs_generate_variant:Nn \tl_put_right:NV { c }
4289 \cs_generate_variant:Nn \tl_put_right:No { c }
4290 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4291 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4292 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4293 \cs_generate_variant:Nn \tl_gput_right:No { c }
4294 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and others. These functions are documented on page 95.)

176.3 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.

```

4295 \group_begin:
4296 \tex_lccode:D '\A = '\@ \scan_stop:
4297 \tex_lccode:D '\B = '\@ \scan_stop:
4298 \tex_catcode:D '\A = 8 \scan_stop:
4299 \tex_catcode:D '\B = 3 \scan_stop:
4300 \tex_lowercase:D
4301 {
4302   \group_end:
4303   \tl_const:Nn \c_tl_rescan_marker_tl { A B }
4304 }

```

`\l_tl_rescan_tl` A token list variable to actually store the material being processed.

```

4305 \tl_new:N \l_tl_rescan_tl

```

`\tl_set_rescan:Nnn` `\tl_set_rescan:Nno` `\tl_set_rescan:cnn` `\tl_set_rescan:cno` 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:

```
! File ended while scanning definition of ...
```

`\tl_gset_rescan:Nnn` `\tl_gset_rescan:Nno` `\tl_gset_rescan:cnn` `\tl_gset_rescan:cno`

`\tl_set_rescan_aux:NNnn` `\tl_rescan_aux:w` 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. As every character with charcode `\tex_newlinechar:D` is replaced by the `\tex_endlinechar:D`, and an extra `\tex_endlinechar:D` is added at the end, we need to set both of those to -1, “unprintable”.

```

4306 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
4307 { \tl_set_rescan_aux:NNnn \tl_set:Nn }
4308 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
4309 { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
4310 \cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4
4311 {
4312   \group_begin:
4313     \exp_args:No \etex_everyeof:D { \c_tl_rescan_marker_tl }
4314     \tex_endlinechar:D \c_minus_one
4315     \tex_newlinechar:D \c_minus_one
4316     #3
4317     \tl_clear:N \l_tl_rescan_tl
4318     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
4319     \exp_args:NNNo \group_end:
4320     #1 #2 \l_tl_rescan_tl
4321   }
4322 \cs_new_nopar:Npx \tl_rescan_aux:w
4323 {
4324   \cs_set_protected:Npn \exp_not:N \tl_rescan_aux:w ##1
4325     \c_tl_rescan_marker_tl
4326     { \tl_set:Nn \exp_not:N \l_tl_rescan_tl {##1} }
4327 }
4328 \tl_rescan_aux:w
4329 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno }
4330 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno }
4331 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno }
4332 \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 98.)

```

\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.

```

4333 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnx
4334 { \tl_set_rescan_aux:NNnx \tl_set:Nn }
4335 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnx
4336 { \tl_set_rescan_aux:NNnx \tl_gset:Nn }
4337 \cs_new_protected_nopar:Npn \tl_set_rescan_aux:NNnx #1#2#3#4
4338 {
4339   \group_begin:
4340     \etex_everyeof:D { \exp_not:N }
4341     \tex_endlinechar:D \c_minus_one
4342     \tex_newlinechar:D \c_minus_one
4343     #3
4344     \tl_set:Nx \l_tl_rescan_tl { \etex_scantokens:D {#4} }

```

```

4345 \exp_args:NNNo \group_end:
4346 #1 #2 \l_tl_rescan_tl
4347 }
4348 \cs_generate_variant:Nn \tl_set_rescan:Nnx { c }
4349 \cs_generate_variant:Nn \tl_gset_rescan:Nnx { c }

```

(End definition for `\tl_set_rescan:Nmx` and `\tl_set_rescan:cnx`. These functions are documented on page 98.)

`\tl_rescan:nn` The same idea is also applied to in line token lists.

```

4350 \cs_new_protected:Npn \tl_rescan:nn #1#2
4351 {
4352   \group_begin:
4353   \exp_args:No \etex_everyeof:D { \c_tl_rescan_marker_tl }
4354   \tex_endlinechar:D \c_minus_one
4355   \tex_newlinechar:D \c_minus_one
4356   #1
4357   \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
4358   \exp_args:No \group_end:
4359   \l_tl_rescan_tl
4360 }

```

(End definition for `\tl_rescan:nn`. This function is documented on page 98.)

176.4 Reassigning token list character codes

`\tl_to_lowercase:n` Just some names for a few primitives.

`\tl_to_uppercase:n`

```

4361 \cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
4362 \cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D

```

(End definition for `\tl_to_lowercase:n`. This function is documented on page 98.)

176.5 Modifying token list variables

`\l_tl_replace_tl` A scratch variable for doing token replacement.

```

4363 \tl_new:N \l_tl_replace_tl

```

`\tl_replace_all:Nnn` All of the replace functions are based on `\tl_replace_aux:NNNnn`, whose arguments are:
`\tl_replace_all:cnx` $\langle function \rangle$, $\langle \text{tl} \rangle \text{set:Nx}$, $\langle \text{tl var} \rangle$, $\langle search\ tokens \rangle$, $\langle replacement\ tokens \rangle$.

```

\l_greplace_all:Nnn
\l_greplace_all:cnx
\tl_replace_once:Nnn
\tl_replace_once:cnx
\tl_greplace_once:Nnn
\tl_greplace_once:cnx
4364 \cs_new_protected_nopar:Npn \tl_replace_once:Nnn
4365 { \tl_replace_aux:NNNnn \tl_replace_once_aux: \tl_set:Nx }
4366 \cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
4367 { \tl_replace_aux:NNNnn \tl_replace_once_aux: \tl_gset:Nx }
4368 \cs_new_protected_nopar:Npn \tl_replace_all:Nnn

```

```

\tl_replace_aux:NNNnn
\tl_replace_aux_ii:w
\tl_replace_all_aux:
\tl_replace_once_aux:
\tl_replace_once_aux_end:w

```

```

4369 { \tl_replace_aux:NNNnn \tl_replace_all_aux: \tl_set:Nx }
4370 \cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
4371 { \tl_replace_aux:NNNnn \tl_replace_all_aux: \tl_gset:Nx }
4372 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
4373 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
4374 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
4375 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

The idea is easier to understand by considering the case of `\tl_replace_all:Nnn`. The replacement happens within an `x`-type expansion. We use an auxiliary function `\tl_tmp:w`, which essentially replaces the next *search tokens* by *replacement tokens*. To avoid runaway arguments, we expand something like `\tl_tmp:w <token list> \q_mark <search tokens> \q_stop`, repeating until the end. How do we detect that we have reached the last occurrence of *search tokens*? The last replacement is characterized by the fact that the argument of `\tl_tmp:w` contains `\q_mark`. In the code below, `\tl_replace_aux_ii:w` takes an argument delimited by `\q_mark`, and removes the following token. Before we reach the end, this gobbles `\q_mark \use_none_delimit_by_q_stop:w` which appear in the definition of `\tl_tmp:w`, and leaves the *replacement tokens*, passed to `\exp_not:n`, to be included in the `x`-expanding definition. At the end, the first `\q_mark` is within the argument of `\tl_tmp:w`, and `\tl_replace_aux_ii:w` gobbles the second `\q_mark` as well, leaving `\use_none_delimit_by_q_stop:w`, which ends the recursion cleanly.

```

4376 \cs_new_protected:Npn \tl_replace_aux:NNNnn #1#2#3#4#5
4377 {
4378   \tl_if_empty:nTF {#4}
4379   {
4380     \msg_kernel_error:nxx { tl } { empty-search-pattern }
4381     { \tl_to_str:n {#5} }
4382   }
4383   {
4384     \cs_set:Npx \tl_tmp:w ##1##2 #4
4385     {
4386       ##2
4387       \exp_not:N \q_mark
4388       \exp_not:N \use_none_delimit_by_q_stop:w
4389       \exp_not:n { \exp_not:n {#5} }
4390       ##1
4391     }
4392     #2 #3
4393     {
4394       \exp_after:wN #1
4395       #3 \q_mark #4 \q_stop
4396     }
4397   }
4398 }
4399 \cs_new:Npn \tl_replace_aux_ii:w #1 \q_mark #2 { \exp_not:o {#1} }

```

The first argument of `\tl_tmp:w` is responsible for repeating the replacement in the case

of `replace_all`, and stopping it early for `replace_once`. Note also that we build `\tl_tmp:w` within an `x`-expansion so that the *<replacement tokens>* can contain `#`. The second `\exp_not:n` ensures that the *<replacement tokens>* are not expanded by `\tl_(g)set:Nx`.

Now on to the difference between “once” and “all”. The `\prg_do_nothing:` and accompanying `o`-expansion ensure that we don’t lose braces in case the tokens between two occurrences of the *<search tokens>* form a brace group.

```

4400 \cs_new:Npn \tl_replace_all_aux:
4401 {
4402   \exp_after:wN \tl_replace_aux_ii:w
4403   \tl_tmp:w \tl_replace_all_aux: \prg_do_nothing:
4404 }
4405 \cs_new_nopar:Npn \tl_replace_once_aux:
4406 {
4407   \exp_after:wN \tl_replace_aux_ii:w
4408   \tl_tmp:w { \tl_replace_once_aux_end:w \prg_do_nothing: } \prg_do_nothing:
4409 }
4410 \cs_new:Npn \tl_replace_once_aux_end:w #1 \q_mark #2 \q_stop
4411 { \exp_not:o {#1} }

```

(End definition for `\tl_replace_all:Nnn` and `\tl_replace_all:cnn`. These functions are documented on page 96.)

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

Removal is just a special case of replacement.

```

4412 \cs_new_protected_nopar:Npn \tl_remove_once:Nn #1#2
4413 { \tl_replace_once:Nnn #1 {#2} { } }
4414 \cs_new_protected_nopar:Npn \tl_gremove_once:Nn #1#2
4415 { \tl_greplace_once:Nnn #1 {#2} { } }
4416 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
4417 \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 97.)

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

Removal is just a special case of replacement.

```

4418 \cs_new_protected_nopar:Npn \tl_remove_all:Nn #1#2
4419 { \tl_replace_all:Nnn #1 {#2} { } }
4420 \cs_new_protected_nopar:Npn \tl_gremove_all:Nn #1#2
4421 { \tl_greplace_all:Nnn #1 {#2} { } }
4422 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
4423 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

176.6 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`

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

`\tl_if_blank_p_aux:NNw`

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.

```

4424 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
4425 { \tl_if_empty_return:o { \use_none:n #1 ? } }
4426 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
4427 \cs_generate_variant:Nn \tl_if_blank:nT { V }
4428 \cs_generate_variant:Nn \tl_if_blank:nF { V }
4429 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
4430 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
4431 \cs_generate_variant:Nn \tl_if_blank:nT { o }
4432 \cs_generate_variant:Nn \tl_if_blank:nF { o }
4433 \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 99.)

`\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`

```

4434 \prg_set_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
4435 {
4436   \if_meaning:w #1 \c_empty_tl
4437   \prg_return_true:
4438   \else:
4439     \prg_return_false:
4440   \fi:
4441 }
4442 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
4443 \cs_generate_variant:Nn \tl_if_empty:N { T }
4444 \cs_generate_variant:Nn \tl_if_empty:N { F }
4445 \cs_generate_variant:Nn \tl_if_empty:N { TF }

```

(End definition for `\tl_if_empty:N` and `\tl_if_empty:c`. These functions are documented on page 99.)

`\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.

```

4446 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }

```

```

4447 {
4448   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4449   \prg_return_true:
4450   \else:
4451     \prg_return_false:
4452   \fi:
4453 }
4454 \cs_generate_variant:Nn \tl_if_empty_p:n { V }
4455 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
4456 \cs_generate_variant:Nn \tl_if_empty:nT { V }
4457 \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 99.)

`\tl_if_empty_p:o`
`\tl_if_empty:oTF`
`\tl_if_empty_return: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.

```

4458 \cs_new:Npn \tl_if_empty_return:o #1
4459 {
4460   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4461   \tl_to_str:n \exp_after:wN {#1} \q_nil
4462   \prg_return_true:
4463   \else:
4464     \prg_return_false:
4465   \fi:
4466 }
4467 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
4468 { \tl_if_empty_return:o {#1} }

```

(End definition for `\tl_if_empty:o`. These functions are documented on page 99.)

`\tl_if_eq_p:NN`
`\tl_if_eq_p:Nc`
`\tl_if_eq_p:cN`
`\tl_if_eq_p:cc`
`\tl_if_eq:NNTF`
`\tl_if_eq:NcTF`
`\tl_if_eq:cNTF`
`\tl_if_eq:ccTF`

Returns `\c_true_bool` if and only if the two token list variables are equal.

```

4469 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
4470 {
4471   \if_meaning:w #1 #2
4472   \prg_return_true:
4473   \else:
4474     \prg_return_false:
4475   \fi:
4476 }
4477 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
4478 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
4479 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
4480 \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 99.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

\l_tl_tmpa_tl
\l_tl_tmpb_tl
4481 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
4482 {
4483   \group_begin:
4484     \tl_set:Nn \l_tl_tmpa_tl {#1}
4485     \tl_set:Nn \l_tl_tmpb_tl {#2}
4486     \if_meaning:w \l_tl_tmpa_tl \l_tl_tmpb_tl
4487     \group_end:
4488     \prg_return_true:
4489   \else:
4490     \group_end:
4491     \prg_return_false:
4492   \fi:
4493 }
4494 \tl_new:N \l_tl_tmpa_tl
4495 \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)`.

```

4496 \cs_new_protected_nopar:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4497 \cs_new_protected_nopar:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4498 \cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4499 \cs_generate_variant:Nn \tl_if_in:NnT { c }
4500 \cs_generate_variant:Nn \tl_if_in:NnF { c }
4501 \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 100.)

`\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 TeX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments.

```

4502 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
4503 {
4504   \cs_set:Npn \tl_tmp:w ##1 #2 { }
4505   \tl_if_empty:oTF { \tl_tmp:w #1 {} {} #2 }
4506   { \prg_return_false: } { \prg_return_true: }

```

```

4507 }
4508 \cs_generate_variant:Nn \tl_if_in:nnT { V }
4509 \cs_generate_variant:Nn \tl_if_in:nnF { V }
4510 \cs_generate_variant:Nn \tl_if_in:nnTF { V }
4511 \cs_generate_variant:Nn \tl_if_in:nnT { o }
4512 \cs_generate_variant:Nn \tl_if_in:nnF { o }
4513 \cs_generate_variant:Nn \tl_if_in:nnTF { o }
4514 \cs_generate_variant:Nn \tl_if_in:nnT { no }
4515 \cs_generate_variant:Nn \tl_if_in:nnF { no }
4516 \cs_generate_variant:Nn \tl_if_in:nnTF { no }

```

(End definition for `\tl_if_in:nn` and others. These functions are documented on page ??.)

176.7 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:Nn
4517 \cs_new:Npn \tl_map_function:nN #1#2
4518 { \tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop }
4519 \cs_new_nopar:Npn \tl_map_function:NN #1#2
4520 {
4521   \exp_after:wN \tl_map_function_aux:Nn
4522   \exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop
4523 }
4524 \cs_new:Npn \tl_map_function_aux:Nn #1#2
4525 {
4526   \quark_if_recursion_tail_stop:n {#2}
4527   #1 {#2} \tl_map_function_aux:Nn #1
4528 }
4529 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for `\tl_map_function:nN`. This function is documented on page 101.)

`\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
4530 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4531 {
4532   \int_gincr:N \g_tl_inline_level_int
4533   \cs_gset:cpn { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4534   ##1 {#2}
4535   \exp_args:Nc \tl_map_function_aux:Nn
4536   { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4537   #1 \q_recursion_tail \q_recursion_stop
4538   \int_gdecr:N \g_tl_inline_level_int

```

```

4539 }
4540 \cs_new_protected:Npn \tl_map_inline:Nn #1#2
4541 {
4542   \int_gincr:N \g_tl_inline_level_int
4543   \cs_gset:cpn { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4544     ##1 {#2}
4545   \exp_last_unbraced:NcV \tl_map_function_aux:Nn
4546     { tl_map_inline_ \int_use:N \g_tl_inline_level_int :n }
4547     #1 \q_recursion_tail\q_recursion_stop
4548   \int_gdecr:N \g_tl_inline_level_int
4549 }
4550 \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` `\tl_map_variable:cNn`
`\tl_map_variable_aux:NnN`

```

4551 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
4552 { \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop }
4553 \cs_new_protected_nopar:Npn \tl_map_variable:NNn
4554 { \exp_args:No \tl_map_variable:nNn }
4555 \cs_new_protected:Npn \tl_map_variable:cNn #1#2#3
4556 {
4557   \tl_set:Nn #1 {#3}
4558   \quark_if_recursion_tail_stop:N #1
4559   #2 \tl_map_variable_aux:Nnn #1 {#2}
4560 }
4561 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`. This function is documented on page 101.)

`\tl_map_break:` The break statement.

```

4562 \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 102.)

176.8 Using token lists

`\tl_to_str:n` Another name for a primitive.

```

4563 \cs_new_eq:NN \tl_to_str:n \etex_detokenize:D

```

(End definition for `\tl_to_str:n`. This function is documented on page 102.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

`\tl_to_str:c`

```

4564 \cs_new_nopar:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
4565 \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 102.)

`\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.

```

4566 \cs_new_eq:NN \tl_use:N \prg_do_nothing:
4567 \cs_new_nopar:Npn \tl_use:c #1
4568 {
4569   \if_cs_exist:w #1 \cs_end:
4570   \cs:w #1 \exp_after:wN \cs_end:
4571   \else:
4572     \msg_expandable_error:n { Undefined~variable~name~'~#1~'! }
4573   \fi:
4574 }
```

(End definition for `\tl_use:N` and `\tl_use:c`. These functions are documented on page 102.)

176.9 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
`\tl_length:V` the list are read as a single element. Spaces are ignored. `\tl_length_aux:n` grabs the
`\tl_length:o` element and replaces it by +1. The 0 to ensure it works on an empty list.
`\tl_length:N`
`\tl_length:c`
`\tl_length_aux:n`

```

4575 \cs_new:Npn \tl_length:n #1
4576 {
4577   \int_eval:n
4578   { 0 \tl_map_function:nN {#1} \tl_length_aux:n }
4579 }
4580 \cs_new_nopar:Npn \tl_length:N #1
4581 {
4582   \int_eval:n
4583   { 0 \tl_map_function:NN #1 \tl_length_aux:n }
4584 }
4585 \cs_new:Npn \tl_length_aux:n #1 { + 1 }
4586 \cs_generate_variant:Nn \tl_length:n { V , o }
4587 \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 103.)

`\tl_reverse_items:n` Reversal of a token list is done by taking one item at a time and putting it after `\q_`-
`\tl_reverse_items_aux:nN` `recursion_stop`.

```

4588 \cs_new:Npn \tl_reverse_items:n #1
4589 { \tl_reverse_items_aux:nw #1 \q_recursion_tail \q_recursion_stop }
4590 \cs_new:Npn \tl_reverse_items_aux:nw #1 #2 \q_recursion_stop
4591 {
4592   \quark_if_recursion_tail_stop_do:nn {#1} { \use_none:n }
```

```

4593     \tl_reverse_items_aux:nw #2 \q_recursion_stop
4594     {#1}
4595   }

```

(End definition for `\tl_reverse_items:n`. This function is documented on page 103.)

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `\tl_tmp:w`, which then receives a single space as its argument: `#1` is `␣`. Removing leading spaces is done with `\tl_trim_spaces_aux_i:w`, which loops until `\q_mark␣` matches the end of the token list: then `##1` is the token list and `##3` is `\tl_trim_spaces_aux_ii:w`. This hands the relevant tokens to the loop `\tl_trim_spaces_aux_iii:w`, responsible for trimming trailing spaces. The end is reached when `␣\q_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `\tl_trim_spaces_aux_iv:w` puts the token list into a group, as the argument of the initial `\etex_unexpanded:D`. The `\etex_unexpanded:D` here is used so that space trimming will behave correctly within an x-type expansion.

```

4596 \cs_set:Npn \tl_tmp:w #1
4597 {
4598   \cs_new:Npn \tl_trim_spaces:n ##1
4599   {
4600     \etex_unexpanded:D
4601     \tl_trim_spaces_aux_i:w
4602     \q_mark
4603     ##1
4604     \q_nil
4605     \q_mark #1 { }
4606     \q_mark \tl_trim_spaces_aux_ii:w
4607     \tl_trim_spaces_aux_iii:w
4608     #1 \q_nil
4609     \tl_trim_spaces_aux_iv:w
4610     \q_stop
4611   }
4612   \cs_new:Npn \tl_trim_spaces_aux_i:w ##1 \q_mark #1 ##2 \q_mark ##3
4613   {
4614     ##3
4615     \tl_trim_spaces_aux_i:w
4616     \q_mark
4617     ##2
4618     \q_mark #1 {##1}
4619   }
4620   \cs_new:Npn \tl_trim_spaces_aux_ii:w ##1 \q_mark \q_mark ##2
4621   {
4622     \tl_trim_spaces_aux_iii:w
4623     ##2
4624   }
4625   \cs_new:Npn \tl_trim_spaces_aux_iii:w ##1 #1 \q_nil ##2
4626   {
4627     ##2

```

```

4628     ##1 \q_nil
4629     \tl_trim_spaces_aux_iii:w
4630   }
4631   \cs_new:Npn \tl_trim_spaces_aux_iv:w ##1 \q_nil ##2 \q_stop
4632     { \exp_after:wN { \use_none:n ##1 } }
4633   }
4634   \tl_tmp:w { ~ }
4635   \cs_new_protected:Npn \tl_trim_spaces:N #1
4636     { \tl_set:Nx #1 { \exp_after:wN \tl_trim_spaces:n \exp_after:wN {#1} } }
4637   \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4638     { \tl_gset:Nx #1 { \exp_after:wN \tl_trim_spaces:n \exp_after:wN {#1} } }
4639   \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4640   \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

(End definition for `\tl_trim_spaces:n`. This function is documented on page 104.)

176.10 The first token from a token list

`\tl_head:n` These functions pick up either the head or the tail of a list. The empty brace groups in `\tl_head:n` and `\tl_tail:n` ensure that a blank argument gives an empty result.

```

\tl_head:n
\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
4641 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4642 \cs_new:Npn \tl_tail:w #1#2 \q_stop {#2}
4643 \cs_new:Npn \tl_head:n #1
4644   { \tl_head:w #1 { } \q_stop }
4645 \cs_new:Npn \tl_tail:n #1
4646   { \tl_tail_aux:w #1 \q_mark { } \q_mark \q_stop }
4647 \cs_new:Npn \tl_tail_aux:w #1 #2 \q_mark #3 \q_stop { #2 }
4648 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4649 \cs_generate_variant:Nn \tl_tail:n { V , v , f }

```

(End definition for `\tl_head:n` and others. These functions are documented on page 105.)

`\str_head:n` After `\tl_to_str:n`, we have a list of character tokens, all with category code 12, except
`\str_tail:n` the space, which has category code 10. Directly using `\tl_head:w` would thus lose leading
`\str_head_aux:w` spaces. Instead, we take an argument delimited by an explicit space, and then only use `\tl_head:w`. If the string started with a space, then the argument of `\str_head_aux:w` is empty, and the function correctly returns a space character. Otherwise, it returns the first token of #1, which is the first token of the string. If the string is empty, we return an empty result.

To remove the first character of `\tl_to_str:n {#1}`, we test it using `\if_charcode:w \scan_stop:`, always false for characters. If the argument was non-empty, then `\str_tail_aux:w` returns everything until the first X (with category code letter, no risk of confusing with the user input). If the argument was empty, the first X is taken by `\if_charcode:w`, and nothing is returned. We use X as a *marker*, rather than a quark because the test `\if_charcode:w \scan_stop: <marker>` has to be false.

```

4650 \cs_new:Npn \str_head:n #1
4651 {
4652   \exp_after:wN \str_head_aux:w
4653   \tl_to_str:n {#1}
4654   { { } } ~ \q_stop
4655 }
4656 \cs_new_nopar:Npn \str_head_aux:w #1 ~ %
4657 { \tl_head:w #1 { ~ } }
4658 \cs_new:Npn \str_tail:n #1
4659 {
4660   \exp_after:wN \str_tail_aux:w
4661   \reverse_if:N \if_charcode:w
4662   \scan_stop: \tl_to_str:n {#1} X X \q_stop
4663 }
4664 \cs_new_nopar:Npn \str_tail_aux:w #1 X #2 \q_stop { \fi: #1 }

```

(End definition for `\str_head:n` and `\str_tail:n`. These functions are documented on page 105.)

```

\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

```

Accessing the first token of a token list is tricky in two cases: when it has category code 1 (begin-group token), or when it is an explicit space, with category code 10 and character code 32.

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, an empty `#1` argument yields `\q_nil`, otherwise the first token of the token list.

```

\if_charcode:w
  \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
\exp_not:N #2

```

The special cases are detected using `\tl_if_head_N_type:n` (the extra `?` takes care of empty arguments). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character).

```

4665 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4666 {
4667   \if_charcode:w
4668     \exp_not:N #2
4669     \tl_if_head_N_type:nTF { #1 ? }
4670     { \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop }
4671     { \str_head:n {#1} }
4672   \prg_return_true:
4673   \else:
4674     \prg_return_false:
4675   \fi:
4676 }
4677 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }

```

```

4678 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
4679 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
4680 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF { f }

```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_N_type`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`.

```

4681 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
4682 {
4683   \if_catcode:w
4684     \exp_not:N #2
4685     \tl_if_head_N_type:nTF { #1 ? }
4686     { \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop }
4687     {
4688       \tl_if_head_group:nTF {#1}
4689       { \c_group_begin_token }
4690       { \c_space_token }
4691     }
4692     \prg_return_true:
4693   \else:
4694     \prg_return_false:
4695   \fi:
4696 }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse.

```

4697 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
4698 {
4699   \tl_if_head_N_type:nTF { #1 ? }
4700   { \tl_if_head_eq_meaning_aux_normal:nN }
4701   { \tl_if_head_eq_meaning_aux_special:nN }
4702   {#1} #2
4703 }
4704 \cs_new:Npn \tl_if_head_eq_meaning_aux_normal:nN #1 #2
4705 {
4706   \exp_after:wN \if_meaning:w \tl_head:w #1 \q_nil \q_stop #2
4707   \prg_return_true:
4708   \else:
4709     \prg_return_false:
4710   \fi:
4711 }
4712 \cs_new:Npn \tl_if_head_eq_meaning_aux_special:nN #1 #2
4713 {
4714   \if_charcode:w \str_head:n {#1} \exp_not:N #2

```

```

4715     \exp_after:wN \use:n
4716 \else:
4717     \prg_return_false:
4718     \exp_after:wN \use_none:n
4719 \fi:
4720 {
4721     \if_catcode:w \exp_not:N #2
4722         \tl_if_head_group:nTF {#1}
4723         { \c_group_begin_token }
4724         { \c_space_token }
4725     \prg_return_true:
4726 \else:
4727     \prg_return_false:
4728 \fi:
4729 }
4730 }

```

(End definition for `\tl_if_head_eq_meaning:nN`. These functions are documented on page 106.)

`\tl_if_head_N_type_p:n` The first token of a token list can be either an N-type argument, a begin-group token (catcode 1), or an explicit space token (catcode 10 and charcode 32). These two cases are characterized by the fact that `\use:n` removes some tokens from #1, hence changing its string representation (no token can have an empty string representation). The extra brace group covers the case of an empty argument, whose head is not “normal”.

`\tl_if_head_N_type:nTF`

```

4731 \prg_new_conditional:Npnn \tl_if_head_N_type:n #1 { p , T , F , TF }
4732 { \str_if_eq_return:on { \use:n #1 { } } { #1 { } } }

```

(End definition for `\tl_if_head_N_type:n`. These functions are documented on page 106.)

`\tl_if_head_group_p:n` Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance.

`\tl_if_head_group:nTF` The extra ? caters for an empty argument.⁷

```

4733 \prg_new_conditional:Npnn \tl_if_head_group:n #1 { p , T , F , TF }
4734 {
4735     \if_predicate:w
4736         \exp_after:wN \use_none:n
4737         \exp_after:wN {
4738             \exp_after:wN {
4739                 \token_to_str:N #1 ?
4740             }
4741             \c_false_bool
4742         }
4743         \c_true_bool
4744     \prg_return_false:
4745 \else:

```

⁷Bruno: this could be made faster, but we don’t: if we hope to ever have an e-type argument, we need all brace “tricks” to happen in one step of expansion, keeping the token list brace balanced at all times.

```

4746     \prg_return_true:
4747     \fi:
4748 }

```

(End definition for `\tl_if_head_group:n`. These functions are documented on page 106.)

`\tl_if_head_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_space:nTF`

`\tl_if_head_space_aux:w`

```

4749 \prg_new_conditional:Npnn \tl_if_head_space:n #1 { p , T , F , TF }
4750 {
4751     \if_int_compare:w
4752         \pdfTeX_strcmp:D
4753         { }
4754         { \tl_if_head_space_aux:w \prg_do_nothing: #1 ? ~ }
4755         = \c_zero
4756         \prg_return_true:
4757     \else:
4758         \prg_return_false:
4759     \fi:
4760 }
4761 \cs_new:Npn \tl_if_head_space_aux:w #1 ~ %
4762 {
4763     \exp_not:o {#1}
4764     \if_false: { \fi: }
4765     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4766 }

```

(End definition for `\tl_if_head_space:n`. These functions are documented on page 107.)

176.11 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`

```

4767 \cs_new_protected:Npn \tl_show:N #1 { \cs_show:N #1 }
4768 \cs_generate_variant:Nn \tl_show:N { c }

```

(End definition for `\tl_show:N` and `\tl_show:c`. These functions are documented on page 107.)

`\tl_show:n` For literal token lists, life is easy.

```

4769 \cs_new_eq:NN \tl_show:n \etex_showtokens:D

```

(End definition for `\tl_show:n`. This function is documented on page 107.)

176.12 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. Lua_TE_X does not quote file names containing spaces, whereas pdf_TE_X and X_T_TE_X do. So there may be a correction to make in the Lua_TE_X case.

```
4770 <*initex>
4771 \tex_everyjob:D \exp_after:wN
4772 {
4773   \tex_the:D \tex_everyjob:D
4774   \luatex_if_engine:T
4775   {
4776     \lua_now:x
4777     { dofile ( assert ( kpse.find_file ("lualatexquotejobname.lua" ) ) ) }
4778   }
4779 }
4780 </initex>
4781 \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
```

`\c_empty_tl` Never full.

```
4782 \tl_const:Nn \c_empty_tl { }
```

`\c_space_tl` A space as a token list (as opposed to as a character).

```
4783 \tl_const:Nn \c_space_tl { ~ }
```

176.13 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.

```
4784 \tl_new:N \g_tmpa_tl
4785 \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.

```
4786 \tl_new:N \l_tmpa_tl
4787 \tl_new:N \l_tmpb_tl
```

176.14 Experimental functions

`\str_if_eq_return_p:on` It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:.` This test is similar to `\str_if_eq:nnTF`, but hard-coded for speed.

```

4788 \cs_new:Npn \str_if_eq_return:on #1 #2
4789 {
4790   \if_int_compare:w
4791     \pdfTeX_strcmp:D { \exp_not:o {#1} } { \exp_not:n {#2} }
4792     = \c_zero
4793     \prg_return_true:
4794   \else:
4795     \prg_return_false:
4796   \fi:
4797 }
```

(End definition for `\str_if_eq_return:on`. These functions are documented on page ??.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:n`.
`\tl_if_single:NTF`

```

4798 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
4799 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
4800 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
4801 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
```

(End definition for `\tl_if_single:n`. These functions are documented on page 100.)

`\tl_if_single_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.

```

4802 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4803 { \str_if_eq_return:on { \use_none:nn #1 ?? } {?} }
```

(End definition for `\tl_if_single:n`. These functions are documented on page 100.)

`\tl_if_single_token_p:n` There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. Otherwise, compare with a single space, only case where we have a single token.

```

4804 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
4805 {
4806   \tl_if_head_N_type:nTF {#1}
4807   { \str_if_eq_return:on { \use_none:n #1 } { } }
4808   { \str_if_eq_return:on { ~ } { #1 } }
4809 }
```

(End definition for `\tl_if_single_token:n`. These functions are documented on page 100.)

`\q_tl_act_mark` The `\tl_act` functions may be applied to any token list. Hence, we use two private
`\q_tl_act_stop` quarks, to allow any token, even quarks, in the token list. Only `\q_tl_act_mark` and
`\q_tl_act_stop` may not appear in the token lists manipulated by `\tl_act` functions.
The quarks are effectively defined in `l3quark`.

`\tl_act:NNNnn` To help control the expansion, `\tl_act:NNNnn` starts with `\tex_romannumeral:D` and
`\tl_act_aux:NNNnn` ends by producing `\c_zero` once the result has been obtained. Then loop over tokens,
`\tl_act_output:n` groups, and spaces in #5. The marker `\q_tl_act_mark` is used both to avoid losing outer
`\tl_act_reverse_output:n` braces and to detect the end of the token list more easily. The result is stored as an
`\tl_act_group_recurse:Nnn` argument for the dummy function `\tl_act_result:n`.

```

\q_tl_act_mark
\q_tl_act_stop

\tl_act:NNNnn
\tl_act_aux:NNNnn
\tl_act_output:n
\tl_act_reverse_output:n
\tl_act_group_recurse:Nnn
\tl_act_loop:w
\tl_act_normal:NwnNNN
\tl_act_group:nwnNNN
\tl_act_space:wwnNNN
\tl_act_end:w

4810 \cs_new:Npn \tl_act:NNNnn { \tex_romannumeral:D \tl_act_aux:NNNnn }
4811 \cs_new:Npn \tl_act_aux:NNNnn #1 #2 #3 #4 #5
4812 {
4813   \tl_act_loop:w #5 \q_tl_act_mark \q_tl_act_stop
4814   {#4} #1 #2 #3
4815   \tl_act_result:n { }
4816 }

```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q_tl_act_mark`, the end of the list. Then leave `\c_zero` and the result in the input stream, to terminate the expansion of `\tex_romannumeral:D`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with a group or with a space. Some extra work is needed to make `\tl_act_space:wwnNNN` gobble the space.

```

4817 \cs_new:Npn \tl_act_loop:w #1 \q_tl_act_stop
4818 {
4819   \tl_if_head_N_type:nTF {#1}
4820   { \tl_act_normal:NwnNNN }
4821   {
4822     \tl_if_head_group:nTF {#1}
4823     { \tl_act_group:nwnNNN }
4824     { \tl_act_space:wwnNNN }
4825   }
4826   #1 \q_tl_act_stop
4827 }
4828 \cs_new:Npn \tl_act_normal:NwnNNN #1 #2 \q_tl_act_stop #3#4
4829 {
4830   \if_meaning:w \q_tl_act_mark #1
4831   \exp_after:wN \tl_act_end:wn
4832   \fi:
4833   #4 {#3} #1
4834   \tl_act_loop:w #2 \q_tl_act_stop
4835   {#3} #4

```

```

4836 }
4837 \cs_new:Npn \tl_act_end:wn #1 \tl_act_result:n #2 { \c_zero #2 }
4838 \cs_new:Npn \tl_act_group:nwnNNN #1 #2 \q_tl_act_stop #3#4#5
4839 {
4840   #5 {#3} {#1}
4841   \tl_act_loop:w #2 \q_tl_act_stop
4842   {#3} #4 #5
4843 }
4844 \exp_last_unbraced:NNo
4845 \cs_new:Npn \tl_act_space:wwnNNN \c_space_tl #1 \q_tl_act_stop #2#3#4#5
4846 {
4847   #5 {#2}
4848   \tl_act_loop:w #1 \q_tl_act_stop
4849   {#2} #3 #4 #5
4850 }

```

Typically, the output is done to the right of what was already output, using `\tl_act_output:n`, but for the `\tl_act_reverse` functions, it should be done to the left.

```

4851 \cs_new:Npn \tl_act_output:n #1 #2 \tl_act_result:n #3
4852 { #2 \tl_act_result:n { #3 #1 } }
4853 \cs_new:Npn \tl_act_reverse_output:n #1 #2 \tl_act_result:n #3
4854 { #2 \tl_act_result:n { #1 #3 } }

```

In many applications of `\tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, `#1` is the output function, `#2` is the transformation, which should expand in two steps, and `#3` is the group.

```

4855 \cs_new:Npn \tl_act_group_recurse:Nnn #1#2#3
4856 {
4857   \exp_args:Nf #1
4858   { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
4859 }

```

(End definition for `\tl_act:NNNnn` and `\tl_act_aux:NNNnn`. These functions are documented on page ??.)

```

\tl_reverse_tokens:n
\tl_act_reverse_normal:nN
\tl_act_reverse_group:nn
\tl_act_reverse_space:n

```

The goal is to reverse a token list. This is done by feeding `\tl_act_aux:NNNnn` three functions, an empty fourth argument (we don't use it for `\tl_act_reverse_tokens:n`), and as a fifth argument the token list to be reversed. Spaces and normal tokens are output to the left of the current output. For groups, we must recursively apply `\tl_act_reverse_tokens:n` to the group, and output, still on the left. Note that in all three cases, we throw one argument away: this *parameter* is where for instance the upper/lowercasing action stores the information of whether it is uppercasing or lowercasing.

```

4860 \cs_new:Npn \tl_reverse_tokens:n
4861 {
4862   \tex_romannumeral:D
4863   \tl_act_aux:NNNnn
4864   \tl_act_reverse_normal:nN

```

```

4865     \tl_act_reverse_group:nn
4866     \tl_act_reverse_space:n
4867     { }
4868   }
4869   \cs_new:Npn \tl_act_reverse_space:n #1
4870     { \tl_act_reverse_output:n {~} }
4871   \cs_new:Npn \tl_act_reverse_normal:nN #1 #2
4872     { \tl_act_reverse_output:n {#2} }
4873   \cs_new:Npn \tl_act_reverse_group:nn #1
4874     {
4875       \tl_act_group_recurse:Nnn
4876       \tl_act_reverse_output:n
4877       { \tl_reverse_tokens:n }
4878     }

```

(End definition for `\tl_reverse_tokens:n`. This function is documented on page 108.)

`\tl_reverse:n` The goal here is to reverse without losing spaces nor braces. The only difference with
`\tl_reverse:o` `\tl_reverse_tokens:n` is that we now simply output groups without entering them.
`\tl_reverse:V`

```

\tl_reverse_group_preserve:nn 4879   \cs_new:Npn \tl_reverse:n
4880   {
4881     \tex_romannumeral:D
4882     \tl_act_aux:NNNnn
4883     \tl_act_reverse_normal:nN
4884     \tl_act_reverse_group_preserve:nn
4885     \tl_act_reverse_space:n
4886     { }
4887   }
4888   \cs_new:Npn \tl_act_reverse_group_preserve:nn #1 #2
4889     { \tl_act_reverse_output:n { {#2} } }
4890   \cs_generate_variant:Nn \tl_reverse:n { o , V }

```

(End definition for `\tl_reverse:n`, `\tl_reverse:o`, and `\tl_reverse:V`. These functions are documented on page 103.)

`\tl_reverse:N` This reverses the list, leaving `{}` in front, which in turn is removed by the `\etex_-`
`\tl_reverse:c` `unexpanded:D` primitive.

```

4891   \cs_new_protected_nopar:Npn \tl_reverse:N #1
4892     { \tl_set:No #1 { \etex_unexpanded:D \tl_reverse:o { #1 { } } } }
4893   \cs_generate_variant:Nn \tl_reverse:N { c }

```

(End definition for `\tl_reverse:N` and `\tl_reverse:c`. These functions are documented on page 103.)

`\tl_length_tokens:n` The length is computed through an `\int_eval:n` construction. Each `1+` is output to
`\tl_act_length_normal:nN` the left, into the integer expression, and the sum is ended by the `\c_zero` inserted by
`\tl_act_length_group:nn` `\tl_act_end:wn`. Somewhat a hack.
`\tl_act_length_space:n`

```

4894   \cs_new:Npn \tl_length_tokens:n #1

```

```

4895 {
4896   \int_eval:n
4897   {
4898     \tl_act_aux:NNNnn
4899     \tl_act_length_normal:nN
4900     \tl_act_length_group:nn
4901     \tl_act_length_space:n
4902     { }
4903     {#1}
4904   }
4905 }
4906 \cs_new:Npn \tl_act_length_normal:nN #1 #2 { 1 + }
4907 \cs_new:Npn \tl_act_length_space:n #1 { 1 + }
4908 \cs_new:Npn \tl_act_length_group:nn #1 #2
4909   { 2 + \tl_length_tokens:n {#2} + }

```

(End definition for `\tl_length_tokens:n`. This function is documented on page 108.)

`\c_tl_act_uppercase_tl` These constants contain the correspondance between lowercase and uppercase letters, in the form aAbBcC... and AaBbCc... respectively.

`\c_tl_act_lowercase_tl`

```

4910 \tl_const:Nn \c_tl_act_uppercase_tl
4911 {
4912   aA bB cC dD eE fF gG hH iI jJ kK lL mM
4913   nN oO pP qQ rR sS tT uU vV wW xX yY zZ
4914 }
4915 \tl_const:Nn \c_tl_act_lowercase_tl
4916 {
4917   Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
4918   Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
4919 }

```

`\tl_expandable_uppercase:n` The only difference between uppercasing and lowercasing is the table of correspondance that is used. As for other token list actions, we feed `\tl_act_aux:NNNnn` three functions, and this time, we use the *parameters* argument to carry which case-changing we are applying. A space is simply output. A normal token is compared to each letter in the alphabet using `\str_if_eq:nn` tests, and converted if necessary to upper/lowercase, before being output. For a group, we must perform the conversion within the group (the `\exp_after:wN` trigger `\tex_romannumeral:D`, which expands fully to give the converted group), then output.

`\tl_expandable_lowercase:n`

`\tl_act_case_normal:nN`

`\tl_act_case_group:nn`

`\tl_act_case_space:n`

```

4920 \cs_new:Npn \tl_expandable_uppercase:n
4921 { \tex_romannumeral:D \tl_act_case_aux:nn { \c_tl_act_uppercase_tl } }
4922 \cs_new:Npn \tl_expandable_lowercase:n
4923 { \tex_romannumeral:D \tl_act_case_aux:nn { \c_tl_act_lowercase_tl } }
4924 \cs_new:Npn \tl_act_case_aux:nn
4925 {
4926   \tl_act_aux:NNNnn
4927   \tl_act_case_normal:nN

```

```

4928     \tl_act_case_group:nn
4929     \tl_act_case_space:n
4930   }
4931   \cs_new:Npn \tl_act_case_space:n #1 { \tl_act_output:n {~} }
4932   \cs_new:Npn \tl_act_case_normal:nN #1 #2
4933   {
4934     \exp_args:Nf \tl_act_output:n
4935     {
4936       \exp_args:NNo \prg_case_str:nnn #2 {#1}
4937       { \exp_stop_f: #2 }
4938     }
4939   }
4940   \cs_new:Npn \tl_act_case_group:nn #1 #2
4941   {
4942     \exp_after:wN \tl_act_output:n \exp_after:wN
4943     { \exp_after:wN { \tex_romannumeral:D \tl_act_case_aux:nn {#1} {#2} } }
4944   }

```

(End definition for `\tl_expandable_uppercase:n` and `\tl_expandable_lowercase:n`. These functions are documented on page 109.)

176.15 Deprecated functions

`\tl_new:Nn` Use either `\tl_const:Nn` or `\tl_new:N`.

`\tl_new:cn`

`\tl_new:Nx`

```

4945   \cs_new_protected:Npn \tl_new:Nn #1#2
4946   {
4947     \tl_new:N #1
4948     \tl_gset:Nn #1 {#2}
4949   }
4950   \cs_generate_variant:Nn \tl_new:Nn { c }
4951   \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`

```

4952   \cs_new_protected_nopar:Npn \tl_gset:Nc
4953   { \pref_global:D \tl_set:Nc }
4954   \cs_new_protected_nopar:Npn \tl_set:Nc #1#2
4955   { \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`

`\tl_replace_all_in:Nnn`

`\tl_replace_all_in:cnn`

`\tl_greplace_all_in:Nnn`

`\tl_greplace_all_in:cnn`

```

4956   \cs_new_eq:NN \tl_replace_in:Nnn \tl_replace_once:Nnn
4957   \cs_new_eq:NN \tl_replace_in:cnn \tl_replace_once:cnn
4958   \cs_new_eq:NN \tl_greplace_in:Nnn \tl_greplace_once:Nnn

```

```

4959 \cs_new_eq:NN \tl_greplace_in:cnn \tl_greplace_once:cnn
4960 \cs_new_eq:NN \tl_replace_all_in:Nnn \tl_replace_all:Nnn
4961 \cs_new_eq:NN \tl_replace_all_in:cnn \tl_replace_all:cnn
4962 \cs_new_eq:NN \tl_greplace_all_in:Nnn \tl_greplace_all:Nnn
4963 \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
\tl_remove_all_in:Nn
\tl_remove_all_in:cn
\tl_gremove_all_in:Nn
\tl_gremove_all_in:cn
4964 \cs_new_eq:NN \tl_remove_in:Nn \tl_remove_once:Nn
4965 \cs_new_eq:NN \tl_remove_in:cn \tl_remove_once:cn
4966 \cs_new_eq:NN \tl_gremove_in:Nn \tl_gremove_once:Nn
4967 \cs_new_eq:NN \tl_gremove_in:cn \tl_gremove_once:cn
4968 \cs_new_eq:NN \tl_remove_all_in:Nn \tl_remove_all:Nn
4969 \cs_new_eq:NN \tl_remove_all_in:cn \tl_remove_all:cn
4970 \cs_new_eq:NN \tl_gremove_all_in:Nn \tl_gremove_all:Nn
4971 \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
\tl_elt_count:N
\tl_elt_count:c
4972 \cs_new_eq:NN \tl_elt_count:n \tl_length:n
4973 \cs_new_eq:NN \tl_elt_count:V \tl_length:V
4974 \cs_new_eq:NN \tl_elt_count:o \tl_length:o
4975 \cs_new_eq:NN \tl_elt_count:N \tl_length:N
4976 \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
\tl_head_iii:f
\tl_head_iii:w
4977 \cs_new_eq:NN \tl_head_i:n \tl_head:n
4978 \cs_new_eq:NN \tl_head_i:w \tl_head:w
4979 \cs_new:Npn \tl_head_iii:n #1 { \tl_head_iii:w #1 \q_stop }
4980 \cs_generate_variant:Nn \tl_head_iii:n { f }
4981 \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 ??.)

```

4982 </initex | package>

```

177 l3seq implementation

The following test files are used for this code: `m3seq002`, `m3seq003`.

```

4983 <*initex | package>

```

```

4984 <*package>
4985 \ProvidesExplPackage
4986   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4987 \package_check_loaded_expl:
4988 </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.

```

4989 \cs_new:Npn \seq_item:n
4990 {
4991   \msg_expandable_error:n { A-sequence-was-used-incorrectly. }
4992   \use_none:n
4993 }

```

(End definition for `\seq_item:n`. This function is documented on page 121.)

`\l_seq_tmpa_tl` Scratch space for various internal uses.

`\l_seq_tmpp_tl`

```

4994 \tl_new:N \l_seq_tmpa_tl
4995 \tl_new:N \l_seq_tmpp_tl

```

177.1 Allocation and initialisation

`\seq_new:N` Internally, sequences are just token lists.

`\seq_new:c`

```

4996 \cs_new_eq:NN \seq_new:N \tl_new:N
4997 \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 109.)

`\seq_clear:N` Clearing sequences is just the same as clearing token lists.

`\seq_clear:c`

`\seq_gclear:N`

`\seq_gclear:c`

```

4998 \cs_new_eq:NN \seq_clear:N \tl_clear:N
4999 \cs_new_eq:NN \seq_clear:c \tl_clear:c
5000 \cs_new_eq:NN \seq_gclear:N \tl_gclear:N
5001 \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 110.)

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

Once again a copy from the token list functions.

```

5002 \cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N
5003 \cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c
5004 \cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N
5005 \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 110.)

`\seq_set_eq:NN`
`\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`

Once again, these are simple copies from the token list functions.

```

5006 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
5007 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
5008 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
5009 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
5010 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
5011 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
5012 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
5013 \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 110.)

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

Concatenating sequences is easy.

```

5014 \cs_new_protected_nopar:Npn \seq_concat:NNN #1#2#3
5015 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
5016 \cs_new_protected_nopar:Npn \seq_gconcat:NNN #1#2#3
5017 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
5018 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
5019 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_concat:ccc`. These functions are documented on page 111.)

177.2 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_put_right:Nv`
`\seq_put_right:No`
`\seq_put_right:Nx`
`\seq_put_right:cn`
`\seq_put_right:cV`
`\seq_put_right:cV`
`\seq_put_right:co`
`\seq_put_right:cx`

The code here is just a wrapper for adding to token lists.

```

5020 \cs_new_protected:Npn \seq_put_left:Nn #1#2
5021 { \tl_put_left:Nn #1 { \seq_item:n {#2} } }
5022 \cs_new_protected:Npn \seq_put_right:Nn #1#2
5023 { \tl_put_right:Nn #1 { \seq_item:n {#2} } }
5024 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
5025 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cV , co , cx }
5026 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
5027 \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 112.)

```

\seq_gput_left:Nn
\seq_gput_left:Nv
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\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:No
\seq_gput_right:Nx
\l_seq_remove_seq
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_remove_duplicates_aux:NN

```

The same for global addition.

```

5028 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
5029 { \tl_gput_left:Nn #1 { \seq_item:n {#2} } }
5030 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
5031 { \tl_gput_right:Nn #1 { \seq_item:n {#2} } }
5032 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
5033 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
5034 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
5035 \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 112.)

177.3 Modifying sequences

An internal sequence for the removal routines.

```

5036 \seq_new:N \l_seq_remove_seq

```

Removing duplicates means making a new list then copying it.

```

5037 \cs_new_protected:Npn \seq_remove_duplicates:N
5038 { \seq_remove_duplicates_aux:NN \seq_set_eq:NN }
5039 \cs_new_protected:Npn \seq_gremove_duplicates:N
5040 { \seq_remove_duplicates_aux:NN \seq_gset_eq:NN }
5041 \cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2
5042 {
5043   \seq_clear:N \l_seq_remove_seq
5044   \seq_map_inline:Nn #2
5045   {
5046     \seq_if_in:NnF \l_seq_remove_seq {##1}
5047     { \seq_put_right:Nn \l_seq_remove_seq {##1} }
5048   }
5049   #1 #2 \l_seq_remove_seq
5050 }
5051 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
5052 \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 114.)

```

\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.

```

5053 \cs_new_protected:Npn \seq_remove_all:Nn
5054 { \seq_remove_all_aux:NNn \tl_set:Nx }
5055 \cs_new_protected:Npn \seq_gremove_all:Nn
5056 { \seq_remove_all_aux:NNn \tl_gset:Nx }
5057 \cs_new_protected:Npn \seq_remove_all_aux:NNn #1#2#3
5058 {
5059   \seq_push_item_def:n
5060   {
5061     \str_if_eq:nnT {##1} {#3}
5062     {
5063       \if_false: { \fi: }
5064       \tl_set:Nn \l_seq_tmpb_tl {##1}
5065       #1 #2
5066       { \if_false: } \fi:
5067       \exp_not:o {#2}
5068       \tl_if_eq:NNT \l_seq_tmpa_tl \l_seq_tmpb_tl
5069       { \use_none:nn }
5070     }
5071     \exp_not:n { \seq_item:n {##1} }
5072   }
5073   \tl_set:Nn \l_seq_tmpa_tl {#3}
5074   #1 #2 {#2}
5075   \seq_pop_item_def:
5076 }
5077 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
5078 \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 114.)

177.4 Sequence conditionals

`\seq_if_empty_p:N`
`\seq_if_empty_p:c`
`\seq_if_empty:NTF`
`\seq_if_empty:cTF`

Simple copies from the token list variable material.

```

5079 \prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N
5080 { p , T , F , TF }
5081 \prg_new_eq_conditional:NNn \seq_if_empty:c \tl_if_empty:c
5082 { 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`
`\seq_if_in:NvTF`
`\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:`

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.

```

5083 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
5084 { T , F , TF }
5085 {
5086   \group_begin:
5087     \tl_set:Nn \l_seq_tmpa_tl {#2}
5088     \cs_set_protected:Npn \seq_item:n ##1
5089     {
5090       \tl_set:Nn \l_seq_tmpb_tl {##1}
5091       \if_meaning:w \l_seq_tmpa_tl \l_seq_tmpb_tl
5092       \exp_after:wN \seq_if_in_aux:
5093       \fi:
5094     }
5095     #1
5096     \seq_break:n { \prg_return_false: }
5097     \seq_break_point:n { \group_end: }
5098   }
5099   \cs_new_nopar:Npn \seq_if_in_aux: { \seq_break:n { \prg_return_true: } }
5100   \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
5101   \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
5102   \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
5103   \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
5104   \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
5105   \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }

```

(End definition for `\seq_if_in:Nn` and others. These functions are documented on page 115.)

177.5 Recovering data from sequences

`\seq_get_left:NN` `\seq_get_left:cN` `\seq_get_left_aux:NnwN` 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.

```

5106 \cs_new_protected_nopar:Npn \seq_get_left:NN #1#2
5107 {
5108   \seq_if_empty_err_break:N #1
5109   \exp_after:wN \seq_get_left_aux:NnwN #1 \q_stop #2
5110   \seq_break_point:n { }
5111 }
5112 \cs_new_protected:Npn \seq_get_left_aux:NnwN \seq_item:n #1#2 \q_stop #3
5113 { \tl_set:Nn #3 {#1} }
5114 \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 118.)

`\seq_pop_left:NN` `\seq_pop_left:cN` `\seq_gpop_left:NN` `\seq_gpop_left:cN` `\seq_pop_left_aux:NNN` `\seq_pop_left_aux:NnwNNN` 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.

```

5115 \cs_new_protected_nopar:Npn \seq_pop_left:NN
5116 { \seq_pop_left_aux:NNN \tl_set:Nn }
5117 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
5118 { \seq_pop_left_aux:NNN \tl_gset:Nn }
5119 \cs_new_protected_nopar:Npn \seq_pop_left_aux:NNN #1#2#3
5120 {
5121   \seq_if_empty_err_break:N #2
5122   \exp_after:wN \seq_pop_left_aux:NnwNNN #2 \q_stop #1#2#3
5123   \seq_break_point:n { }
5124 }
5125 \cs_new_protected:Npn \seq_pop_left_aux:NnwNNN \seq_item:n #1#2 \q_stop #3#4#5
5126 {
5127   #3 #4 {#2}
5128   \tl_set:Nn #5 {#1}
5129 }
5130 \cs_generate_variant:Nn \seq_pop_left:NN { c }
5131 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and `\seq_pop_left:cN`. These functions are documented on page 119.)

`\seq_get_right:NN` The idea here is to remove the very first `\seq_item:n` from the sequence, leaving a token
`\seq_get_right:cN` list starting with the first braced entry. Two arguments at a time are then grabbed: apart
`\seq_get_right_aux:NN` from the right-hand end of the sequence, this will be a brace group followed by `\seq_`
`\seq_get_right_loop:nn` `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`.

```

5132 \cs_new_protected_nopar:Npn \seq_get_right:NN #1#2
5133 {
5134   \seq_if_empty_err_break:N #1
5135   \seq_get_right_aux:NN #1#2
5136   \seq_break_point:n { }
5137 }
5138 \cs_new_protected_nopar:Npn \seq_get_right_aux:NN #1#2
5139 {
5140   \seq_push_item_def:n { }
5141   \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
5142   \exp_after:wN \use_none:n #1
5143   { \tl_set:Nn #2 }
5144   { }
5145   {
5146     \seq_pop_item_def:
5147     \seq_break:
5148   }
5149 }
5150 \cs_new:Npn \seq_get_right_loop:nn #1#2
5151 {
5152   #2 {#1}
5153   \seq_get_right_loop:nn

```

```

5154 }
5155 \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 118.)

```

\seq_pop_right:NN
\seq_pop_right:cN
\seq_gpop_right:NN
\seq_gpop_right:cN
\seq_pop_right_aux:NNN
\seq_pop_right_aux_ii:NNN

```

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.

```

5156 \cs_new_protected_nopar:Npn \seq_pop_right:NN
5157 { \seq_pop_right_aux:NNN \tl_set:Nx }
5158 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
5159 { \seq_pop_right_aux:NNN \tl_gset:Nx }
5160 \cs_new_protected_nopar:Npn \seq_pop_right_aux:NNN #1#2#3
5161 {
5162   \seq_if_empty_err_break:N #2
5163   \seq_pop_right_aux_ii:NNN #1 #2 #3
5164   \seq_break_point:n { }
5165 }
5166 \cs_new_protected_nopar:Npn \seq_pop_right_aux_ii:NNN #1#2#3
5167 {
5168   \seq_push_item_def:n { \exp_not:n { \seq_item:n {##1} } }
5169   #1 #2 { \if_false: } \fi:
5170   \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
5171   \exp_after:wN \use_none:n #2
5172   {
5173     \if_false: { \fi: }
5174     \tl_set:Nn #3
5175   }
5176   { }
5177   {
5178     \seq_pop_item_def:
5179     \seq_break:
5180   }
5181 }
5182 \cs_generate_variant:Nn \seq_pop_right:NN { c }
5183 \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 119.)

177.6 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.

```
5184 \cs_new:Npn \seq_break: #1 \seq_break_point:n #2 {#2}
5185 \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 122.)

\seq_map_break: Semantically-logical copies of the break functions for use inside mappings.
\seq_map_break:n

```
5186 \cs_new_eq:NN \seq_map_break: \seq_break:
5187 \cs_new_eq:NN \seq_map_break:n \seq_break:n
```

(End definition for \seq_map_break:. This function is documented on page 116.)

\seq_break_point:n Normally, the marker token will not be executed, but if it is then the end code is simply inserted.

```
5188 \cs_new_eq:NN \seq_break_point:n \use:n
```

(End definition for \seq_break_point:n. This function is documented on page 122.)

\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.

```
5189 \cs_new_protected_nopar:Npn \seq_if_empty_err_break:N #1
5190 {
5191   \if_meaning:w #1 \c_empty_tl
5192     \msg_kernel_error:nnx { seq } { empty-sequence } { \token_to_str:N #1 }
5193   \exp_after:wN \seq_break:
5194   \fi:
5195 }
```

(End definition for \seq_if_empty_err_break:N. This function is documented on page 121.)

\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.

```
5196 \cs_new:Npn \seq_map_function:NN #1#2
5197 {
5198   \exp_after:wN \seq_map_function_aux:NNn \exp_after:wN #2 #1
5199   { ? \seq_map_break: } { }
5200   \seq_break_point:n { }
5201 }
```

```

5202 \cs_new:Npn \seq_map_function_aux:NNn #1#2#3
5203 {
5204   \use_none:n #2
5205   #1 {#3}
5206   \seq_map_function_aux:NNn #1
5207 }
5208 \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 115.)

`\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 the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

\seq_push_item_def:x
\seq_push_item_def_aux:
\seq_pop_item_def:
5209 \cs_new_protected:Npn \seq_push_item_def:n
5210 {
5211   \seq_push_item_def_aux:
5212   \cs_gset:Npn \seq_item:n ##1
5213 }
5214 \cs_new_protected:Npn \seq_push_item_def:x
5215 {
5216   \seq_push_item_def_aux:
5217   \cs_gset:Npx \seq_item:n ##1
5218 }
5219 \cs_new_protected:Npn \seq_push_item_def_aux:
5220 {
5221   \cs_gset_eq:cN { seq_item_ \int_use:N \g_seq_nesting_depth_int :n }
5222   \seq_item:n
5223   \int_gincr:N \g_seq_nesting_depth_int
5224 }
5225 \cs_new_protected_nopar:Npn \seq_pop_item_def:
5226 {
5227   \int_gdecr:N \g_seq_nesting_depth_int
5228   \cs_gset_eq:Nc \seq_item:n
5229   { seq_item_ \int_use:N \g_seq_nesting_depth_int :n }
5230 }

```

(End definition for `\seq_push_item_def:n` and `\seq_push_item_def:x`. These functions are documented on page 121.)

`\seq_map_inline:Nn` The idea here is that `\seq_item:n` is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining `\seq_item:n`.

```

\seq_map_inline:cn
5231 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
5232 {
5233   \seq_push_item_def:n {#2}
5234   #1

```

```

5235     \seq_break_point:n { \seq_pop_item_def: }
5236   }
5237   \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 115.)

`\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
5238   \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
5239   {
5240     \seq_push_item_def:x
5241     {
5242       \tl_set:Nn \exp_not:N #2 {##1}
5243       \exp_not:n {#3}
5244     }
5245     #1
5246     \seq_break_point:n { \seq_pop_item_def: }
5247   }
5248   \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
5249   \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 115.)

177.7 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
\seq_push:No
\seq_push:Nx
\seq_push:cn
\seq_push:cV
\seq_push:cV
\seq_push:co
\seq_push:cx
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:Nv
\seq_gpush:No
\seq_gpush:Nx
\seq_gpush:cn
\seq_gpush:cV
\seq_gpush:cv
\seq_gpush:co
\seq_gpush:cx

```

```

5250   \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
5251   \cs_new_eq:NN \seq_push:NV \seq_put_left:NV
5252   \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
5253   \cs_new_eq:NN \seq_push:No \seq_put_left:No
5254   \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
5255   \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
5256   \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
5257   \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
5258   \cs_new_eq:NN \seq_push:co \seq_put_left:co
5259   \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
5260   \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
5261   \cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
5262   \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
5263   \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
5264   \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
5265   \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
5266   \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
5267   \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv

```

```

5268 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
5269 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx

```

(End definition for `\seq_push:Nn` and others. These functions are documented on page 118.)

`\seq_get:NN` In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```

\seq_get:cN
\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN
5270 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
5271 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
5272 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
5273 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
5274 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
5275 \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 117.)

177.8 Viewing sequences

`\l_seq_show_tl` Used to store the material for display.

```

5276 \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 auxiliary. To avoid a low-level TeX error if there is an empty sequence, a simple test is used to keep the output “clean”.

```

\seq_show_aux:n
\seq_show_aux:w
5277 \cs_new_protected_nopar:Npn \seq_show:N #1
5278 {
5279   \seq_if_empty:NTF #1
5280   {
5281     \iow_term:x { Sequence~\token_to_str:N #1 \c_space_tl is-empty }
5282     \tl_show:n { }
5283   }
5284   {
5285     \iow_term:x
5286     {
5287       Sequence~\token_to_str:N #1 \c_space_tl
5288       contains~the~items~(without~outer~braces):
5289     }
5290     \tl_set:Nx \l_seq_show_tl
5291     { \seq_map_function:NN #1 \seq_show_aux:n }
5292     \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
5293     { \exp_after:wN \seq_show_aux:w \l_seq_show_tl }
5294   }
5295 }
5296 \cs_new:Npn \seq_show_aux:n #1
5297 {

```

```

5298 \iow_newline: > \c_space_tl \c_space_tl
5299 \iow_char:N \{ \exp_not:n {#1} \iow_char:N \}
5300 }
5301 \cs_new:Npn \seq_show_aux:w #1 > ~ { }
5302 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page 118.)

177.9 Experimental functions

`\seq_if_empty_break_return_false:N` The name says it all: of the sequence is empty, returns logical false.

```

5303 \cs_new_nopar:Npn \seq_if_empty_break_return_false:N #1
5304 {
5305   \if_meaning:w #1 \c_empty_tl
5306   \prg_return_false:
5307   \exp_after:wN \seq_break:
5308   \fi:
5309 }

```

(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`

```

5310 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1 #2 { T , F , TF }
5311 {
5312   \seq_if_empty_break_return_false:N #1
5313   \exp_after:wN \seq_get_left_aux:Nw #1 \q_stop #2
5314   \prg_return_true:
5315   \seq_break:
5316   \seq_break_point:n { }
5317 }
5318 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
5319 {
5320   \seq_if_empty_break_return_false:N #1
5321   \seq_get_right_aux:NN #1#2
5322   \prg_return_true: \seq_break:
5323   \seq_break_point:n { }
5324 }
5325 \cs_generate_variant:Nn \seq_get_left:NNT { c }
5326 \cs_generate_variant:Nn \seq_get_left:NNF { c }
5327 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
5328 \cs_generate_variant:Nn \seq_get_right:NNT { c }
5329 \cs_generate_variant:Nn \seq_get_right:NNF { c }
5330 \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 118.)

`\seq_pop_left:NNTF` More or less the same for popping.
`\seq_pop_left:cNTF`
`\seq_gpop_left:NNTF` 5331 `\prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }`
`\seq_gpop_left:cNTF` 5332 `{`
`\seq_pop_right:NNTF` 5333 `\seq_if_empty_break_return_false:N #1`
`\seq_pop_right:cNTF` 5334 `\exp_after:wN \seq_pop_left_aux:NnwNNN #1 \q_stop \tl_set:Nn #1#2`
`\seq_gpop_right:NNTF` 5335 `\prg_return_true: \seq_break:`
`\seq_gpop_right:cNTF` 5336 `\seq_break_point:n { }`
5337 `}`
5338 `\prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }`
5339 `{`
5340 `\seq_if_empty_break_return_false:N #1`
5341 `\exp_after:wN \seq_pop_left_aux:NnwNNN #1 \q_stop \tl_gset:Nn #1#2`
5342 `\prg_return_true: \seq_break:`
5343 `\seq_break_point:n { }`
5344 `}`
5345 `\prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }`
5346 `{`
5347 `\seq_if_empty_break_return_false:N #1`
5348 `\seq_pop_right_aux_ii:NNN \tl_set:Nx #1 #2`
5349 `\prg_return_true: \seq_break:`
5350 `\seq_break_point:n { }`
5351 `}`
5352 `\prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }`
5353 `{`
5354 `\seq_if_empty_break_return_false:N #1`
5355 `\seq_pop_right_aux_ii:NNN \tl_gset:Nx #1 #2`
5356 `\prg_return_true: \seq_break:`
5357 `\seq_break_point:n { }`
5358 `}`
5359 `\cs_generate_variant:Nn \seq_pop_left:NNT { c }`
5360 `\cs_generate_variant:Nn \seq_pop_left:NNF { c }`
5361 `\cs_generate_variant:Nn \seq_pop_left:NNTF { c }`
5362 `\cs_generate_variant:Nn \seq_gpop_left:NNT { c }`
5363 `\cs_generate_variant:Nn \seq_gpop_left:NNF { c }`
5364 `\cs_generate_variant:Nn \seq_gpop_left:NNTF { c }`
5365 `\cs_generate_variant:Nn \seq_pop_right:NNT { c }`
5366 `\cs_generate_variant:Nn \seq_pop_right:NNF { c }`
5367 `\cs_generate_variant:Nn \seq_pop_right:NNTF { c }`
5368 `\cs_generate_variant:Nn \seq_gpop_right:NNT { c }`
5369 `\cs_generate_variant:Nn \seq_gpop_right:NNF { c }`
5370 `\cs_generate_variant:Nn \seq_gpop_right:NNTF { c }`

(End definition for `\seq_pop_left:NN` and `\seq_pop_left:cN`. These functions are documented on page 119.)

`\seq_length:N` Counting the items in a sequence is done using the same approach as for other length
`\seq_length:c` functions: turn each entry into a +1 then use integer evaluation to actually do the math-
`\seq_length_aux:n` ematics.

```

5371 \cs_new:Npn \seq_length:N #1
5372 {
5373   \int_eval:n
5374   {
5375     0
5376     \seq_map_function:NN #1 \seq_length_aux:n
5377   }
5378 }
5379 \cs_new:Npn \seq_length_aux:n #1 { +1 }
5380 \cs_generate_variant:Nn \seq_length:N { c }

```

(End definition for `\seq_length:N` and `\seq_length:c`. These functions are documented on page 119.)

`\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`

```

5381 \cs_new_nopar:Npn \seq_item:Nn #1#2
5382 {
5383   \exp_last_unbraced:Nfo \seq_item_aux:nnn
5384   {
5385     \int_eval:n
5386     {
5387       \int_compare:nNnT {#2} < \c_zero
5388       { \seq_length:N #1 + }
5389       #2
5390     }
5391   }
5392   #1
5393   { ? \seq_break: }
5394   { }
5395   \seq_break_point:n { }
5396 }
5397 \cs_new_nopar:Npn \seq_item_aux:nnn #1#2#3
5398 {
5399   \use_none:n #2
5400   \int_compare:nNnTF {#1} = \c_zero
5401   { \seq_break:n {#3} }
5402   { \exp_args:Nf \seq_item_aux:nnn { #1 - 1 } }
5403 }
5404 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and `\seq_item:cn`. These functions are documented on page 119.)

`\seq_use:N` A simple short cut for a mapping.

`\seq_use:c`

```

5405 \cs_new_nopar:Npn \seq_use:N #1 { \seq_map_function:NN #1 \use:n }
5406 \cs_generate_variant:Nn \seq_use:N { c }

```

(End definition for `\seq_use:N` and `\seq_use:c`. These functions are documented on page 120.)

```

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
  \seq_mapthread_function_aux:NN
  \seq_mapthread_function_aux:Nnnwnn

```

The idea here is to first expand both of the sequences, adding the usual `{ ? \seq_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first 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.

```

5407 \cs_new_nopar:Npn \seq_mapthread_function:NNN #1#2#3
5408 {
5409   \exp_after:wN \seq_mapthread_function_aux:NN
5410   \exp_after:wN #3
5411   \exp_after:wN #1
5412   #2
5413   { ? \seq_break: } { }
5414   \seq_break_point:n { }
5415 }
5416 \cs_new_nopar:Npn \seq_mapthread_function_aux:NN #1#2
5417 {
5418   \exp_after:wN \seq_mapthread_function_aux:Nnnwnn
5419   \exp_after:wN #1
5420   #2
5421   { ? \seq_break: } { }
5422   \q_stop
5423 }
5424 \cs_new:Npn \seq_mapthread_function_aux:Nnnwnn #1#2#3#4 \q_stop #5#6
5425 {
5426   \use_none:n #2
5427   \use_none:n #5
5428   #1 {#3} {#6}
5429   \seq_mapthread_function_aux:Nnnwnn #1 #4 \q_stop
5430 }
5431 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }
5432 \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 120.)

```

\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.

```

5433 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
5434 {
5435   \tl_set:Nx #1
5436   { \clist_map_function:NN #2 \seq_wrap_item:n }
5437 }
5438 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
5439 {
5440   \tl_set:Nx #1
5441   { \clist_map_function:nN {#2} \seq_wrap_item:n }
5442 }

```

```

5443 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
5444 {
5445   \tl_gset:Nx #1
5446   { \clist_map_function:NN #2 \seq_wrap_item:n }
5447 }
5448 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
5449 {
5450   \tl_gset:Nx #1
5451   { \clist_map_function:nN {#2} \seq_wrap_item:n }
5452 }
5453 \cs_new:Npn \seq_wrap_item:n #1 { \exp_not:n { \seq_item:n {#1} } }
5454 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
5455 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
5456 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
5457 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
5458 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
5459 \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 120.)

`\seq_set_reverse:N` Define `\seq_item:n` to place its argument after a marker, `\prg_do_nothing:`. Then
`\seq_gset_reverse:N` x-expand the sequence.

```

5460 \cs_new_protected_nopar:Npn \seq_tmp:w
5461 { \msg_expandable_error:n { There-is-a-bug-in-LaTeX3! } }
5462 \cs_new_protected_nopar:Npn \seq_set_reverse:N
5463 { \seq_reverse_aux:NN \tl_set:Nx }
5464 \cs_new_protected_nopar:Npn \seq_gset_reverse:N
5465 { \seq_reverse_aux:NN \tl_gset:Nx }
5466 \cs_new_protected_nopar:Npn \seq_reverse_aux:NN #1 #2
5467 {
5468   \cs_set_eq:NN \seq_tmp:w \seq_item:n
5469   \cs_set_eq:NN \seq_item:n \seq_reverse_aux_item:w
5470   #1 #2 { #2 \prg_do_nothing: }
5471   \cs_set_eq:NN \seq_item:n \seq_tmp:w
5472 }
5473 \cs_new:Npn \seq_reverse_aux_item:w #1 #2 \prg_do_nothing:
5474 {
5475   #2
5476   \prg_do_nothing:
5477   \exp_not:n { \seq_item:n {#1} }
5478 }

```

(End definition for `\seq_set_reverse:N` and `\seq_gset_reverse:N`. These functions are documented on page 121.)

`\seq_set_split:Nnn` The goal is to split a given token list at a marker, strip spaces from each item, and
`\seq_gset_split:Nnn` remove one set of outer braces if after removing leading and trailing spaces the item
`\seq_set_split_aux:Nnn` is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l_seq_tmpa_tl`
`\seq_set_split_aux_i:w` is a repetition of the pattern `\seq_set_split_aux_i:w \prg_do_nothing: <item with`
`\seq_set_split_aux_ii:w`
`\seq_set_split_aux_end:`

spaces) `\seq_set_split_aux_end:`. Then, x-expansion causes `\seq_set_split_aux_i:w` to trim spaces, and leaves its result as `\seq_set_split_aux_ii:w` *<trimmed item>* `\seq_set_split_aux_end:`. This is then converted to the `l3seq` internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

5479 \cs_new_protected_nopar:Npn \seq_set_split:Nnn
5480 { \seq_set_split_aux:NNnn \tl_set:Nx }
5481 \cs_new_protected_nopar:Npn \seq_gset_split:Nnn
5482 { \seq_set_split_aux:NNnn \tl_gset:Nx }
5483 \cs_new_protected_nopar:Npn \seq_set_split_aux:NNnn #1 #2 #3 #4
5484 {
5485   \tl_if_empty:nTF {#4}
5486     { #1 #2 { } }
5487     {
5488       \tl_set:Nn \l_seq_tmpa_tl
5489       {
5490         \seq_set_split_aux_i:w \prg_do_nothing:
5491         #4
5492         \seq_set_split_aux_end:
5493       }
5494       \tl_replace_all:Nnn \l_seq_tmpa_tl { #3 }
5495       {
5496         \seq_set_split_aux_end:
5497         \seq_set_split_aux_i:w \prg_do_nothing:
5498       }
5499       \tl_set:Nx \l_seq_tmpa_tl { \l_seq_tmpa_tl }
5500       #1 #2 { \l_seq_tmpa_tl }
5501     }
5502   }
5503 \cs_new:Npn \seq_set_split_aux_i:w #1 \seq_set_split_aux_end:
5504 {
5505   \exp_not:N \seq_set_split_aux_ii:w
5506   \exp_args:No \tl_trim_spaces:n {#1}
5507   \exp_not:N \seq_set_split_aux_end:
5508 }
5509 \cs_new:Npn \seq_set_split_aux_ii:w #1 \seq_set_split_aux_end:
5510 { \exp_not:n { \seq_item:n {#1} } }

```

(End definition for `\seq_set_split:Nnn` and `\seq_gset_split:Nnn`. These functions are documented on page [121](#).)

177.10 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`

```
5511 \cs_new_eq:NN \seq_top:NN \seq_get_left:NN
5512 \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`

```
5513 \cs_new_eq:NN \seq_display:N \seq_show:N
5514 \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 ??.)

```
5515 </initex | package>
```

178 l3clist implementation

The following test files are used for this code: *m3clist002*.

```
5516 <*initex | package>
```

```
5517 <*package>
```

```
5518 \ProvidesExplPackage
```

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

```
5520 \package_check_loaded_expl:
```

```
5521 </package>
```

`\l_clist_tmpa_tl` Scratch space for various internal uses.

`\l_clist_tmpb_tl`

```
5522 \tl_new:N \l_clist_tmpa_tl
```

```
5523 \tl_new:N \l_clist_tmpb_tl
```

178.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

`\clist_new:c`

```
5524 \cs_new_eq:NN \clist_new:N \tl_new:N
```

```
5525 \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 122.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

`\clist_clear:c`

`\clist_gclear:N`

`\clist_gclear:c`

```
5526 \cs_new_eq:NN \clist_clear:N \tl_clear:N
```

```
5527 \cs_new_eq:NN \clist_clear:c \tl_clear:c
```

```
5528 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
```

```
5529 \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 123.)

```
\clist_clear_new:N Once again a copy from the token list functions.
\clist_clear_new:c
\clist_gclear_new:N 5530 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:c 5531 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
                    5532 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
                    5533 \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 123.)

```
\clist_set_eq:NN Once again, these are simple copies from the token list functions.
\clist_set_eq:cN
\clist_set_eq:Nc 5534 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:cc 5535 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_gset_eq:NN 5536 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:cN 5537 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:Nc 5538 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:cc 5539 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
                    5540 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
                    5541 \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 123.)

```
\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 5542 \cs_new_protected_nopar:Npn \clist_concat:NNN
                    5543 { \clist_concat_aux:NNNN \tl_set:Nx }
                    5544 \cs_new_protected_nopar:Npn \clist_gconcat:NNN
                    5545 { \clist_concat_aux:NNNN \tl_gset:Nx }
                    5546 \cs_new_protected_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4
                    5547 {
                    5548   #1 #2
                    5549   {
                    5550     \clist_if_empty:NTF #3
                    5551     { \exp_not:o #4 }
                    5552     {
                    5553       \exp_not:o #3
                    5554       \clist_if_empty:NF #4
                    5555       {
                    5556         ,
                    5557         \exp_not:o #4
                    5558       }
                    5559     }
                    5560   }
                    5561 }
                    5562 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
                    5563 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(End definition for `\clist_concat:NNN` and `\clist_concat:ccc`. These functions are documented on page 124.)

178.2 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.

```
5564 \cs_new_protected_nopar:Npn \clist_put_left:Nn
5565 { \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn { } , }
5566 \cs_new_protected_nopar:Npn \clist_gput_left:Nn
5567 { \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_left:Nn { } , }
5568 \cs_new_protected:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6
5569 {
5570   \clist_if_empty:NTF #5
5571     { #1 #5 {#6} }
5572     { \tl_if_empty:nF {#6} { #2 #5 { #3 #6 #4 } } }
5573 }
5574 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
5575 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
5576 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
5577 \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 125.)

`\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`
`\clist_get_aux:wN`

The same for the right side.

```
5578 \cs_new_protected:Npn \clist_put_right:Nn
5579 { \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn , { } }
5580 \cs_new_protected_nopar:Npn \clist_gput_right:Nn
5581 { \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , { } }
5582 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
5583 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
5584 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
5585 \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 125.)

178.3 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.

```
5586 \cs_new_protected_nopar:Npn \clist_get:NN #1#2
5587 { \exp_after:wN \clist_get_aux:wN #1 , \q_stop #2 }
5588 \cs_new_protected:Npn \clist_get_aux:wN #1 , #2 \q_stop #3
5589 { \tl_set:Nn #3 {#1} }
5590 \cs_generate_variant:Nn \clist_get:NN { c }
```

(End definition for `\clist_get:NN` and `\clist_get:cN`. These functions are documented on page 131.)

`\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`

```

5591 \cs_new_protected_nopar:Npn \clist_pop:NN
5592   { \clist_pop_aux:NNN \tl_set:Nn }
5593 \cs_new_protected_nopar:Npn \clist_gpop:NN
5594   { \clist_pop_aux:NNN \tl_gset:Nn }
5595 \cs_new_protected_nopar:Npn \clist_pop_aux:NNN #1#2#3
5596   { \exp_after:wN \clist_pop_aux:wNNN #2 , \q_nil , \q_nil , \q_stop #1#2#3 }
5597 \cs_new_protected:Npn \clist_pop_aux:wNNN #1 , #2 , \q_nil , #3 \q_stop #4#5#6
5598   {
5599     \quark_if_nil:nTF {#2}
5600       { #4 #5 { } }
5601       { #4 #5 {#2} }
5602     \tl_set:Nn #6 {#1}
5603   }
5604 \cs_generate_variant:Nn \clist_pop:NN { c }
5605 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and `\clist_pop:cN`. These functions are documented on page 131.)

`\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`

```

5606 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
5607 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
5608 \cs_new_eq:NN \clist_push:No \clist_put_left:No
5609 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
5610 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
5611 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
5612 \cs_new_eq:NN \clist_push:co \clist_put_left:co
5613 \cs_new_eq:NN \clist_push:cx \clist_put_left:cx
5614 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
5615 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
5616 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
5617 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
5618 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
5619 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
5620 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
5621 \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 132.)

178.4 Using comma lists

`\clist_use:N` The approach is the same as for `\tl_use:N`.
`\clist_use:c`

```

5622 \cs_new_eq:NN \clist_use:N \tl_use:N
5623 \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 127.)

178.5 Modifying comma lists

`\l_clist_remove_clist` An internal comma list for the removal routines.

```
5624 \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

5625 \cs_new_protected:Npn \clist_remove_duplicates:N
5626 { \clist_remove_duplicates_aux:NN \clist_set_eq:NN }
5627 \cs_new_protected:Npn \clist_gremove_duplicates:N
5628 { \clist_remove_duplicates_aux:NN \clist_gset_eq:NN }
5629 \cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2
5630 {
5631   \clist_clear:N \l_clist_remove_clist
5632   \clist_map_inline:Nn #2
5633   {
5634     \clist_if_in:NnF \l_clist_remove_clist {##1}
5635     { \clist_put_right:Nn \l_clist_remove_clist {##1} }
5636   }
5637   #1 #2 \l_clist_remove_clist
5638 }
5639 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
5640 \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 127.)

`\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`

```
5641 \cs_new_protected:Npn \clist_remove_all:Nn
5642 { \clist_remove_all_aux:NNn \clist_set_eq:NN }
5643 \cs_new_protected:Npn \clist_gremove_all:Nn
5644 { \clist_remove_all_aux:NNn \clist_gset_eq:NN }
5645 \cs_new_protected:Npn \clist_remove_all_aux:NNn #1#2#3
5646 {
5647   \clist_if_empty:NF #2
5648   {
5649     \clist_clear:N \l_clist_remove_clist
5650     \cs_set_protected:Npn \clist_remove_all_aux:w
5651     ##1 , #3 , ##2 \q_stop
5652     {
5653       \tl_put_right:Nn \l_clist_remove_clist {##1}
```

```

5654     \quark_if_no_value:nF {##2}
5655     { \clist_remove_all_aux:w , ##2 \q_stop }
5656   }
5657   \exp_after:wN \clist_remove_all_aux:w
5658   \exp_after:wN , #2 , #3 , \q_no_value \q_stop
5659   \tl_if_empty:NF \l_clist_remove_clist
5660   {
5661     \exp_after:wN \tl_set:No \exp_after:wN
5662     \l_clist_remove_clist \exp_after:wN
5663     { \exp_after:wN \use_none:n \l_clist_remove_clist }
5664   }
5665   #1 #2 \l_clist_remove_clist
5666 }
5667 }
5668 \cs_new_protected:Npn \clist_remove_all_aux:w { }
5669 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
5670 \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 128.)

`\clist_trim_spaces: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.

`\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`

```

5671 \cs_new:Npn \clist_trim_spaces:n #1
5672 {
5673   \exp_args:Nf \clist_trim_spaces_aux_i:n
5674   { \clist_map_function:nN {#1} \clist_trim_spaces_aux_ii:n }
5675 }
5676 \cs_new:Npn \clist_trim_spaces_aux_i:n #1 { \use_ii:nn #1 }
5677 \cs_new:Npn \clist_trim_spaces_aux_ii:n #1
5678 { , \tl_trim_spaces:n {#1} }
5679 \cs_new_protected:Npn \clist_trim_spaces:N #1
5680 { \tl_set:Nf #1 { \exp_args:No \clist_trim_spaces:n #1 } }
5681 \cs_new_protected:Npn \clist_gtrim_spaces:N #1
5682 { \tl_gset:Nf #1 { \exp_args:No \clist_trim_spaces:n #1 } }
5683 \cs_generate_variant:Nn \clist_trim_spaces:N { c }
5684 \cs_generate_variant:Nn \clist_gtrim_spaces:N { c }

```

(End definition for `\clist_trim_spaces:n`. This function is documented on page ??.)

178.6 Comma list conditionals

`\clist_tmp:w` A temporary function for comparison.

```

5685 \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
\clist_if_empty:NTF
\clist_if_empty:cTF
5686 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }
5687 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c { p , T , F , TF }

```

(End definition for `\clist_if_empty:N` and `\clist_if_empty:c`. These functions are documented on page 128.)

`\clist_if_eq_p:NN` Simple copies from the token list variable material.

```

\clist_if_eq_p:Nc
\clist_if_eq_p:cN
\clist_if_eq_p:cc
\clist_if_eq:NTF
\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
5688 \prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
5689 \prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
5690 \prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
5691 \prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc { p , T , F , TF }

```

(End definition for `\clist_if_eq:NN` and others. These functions are documented on page 129.)

```

5692 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2
5693 { T , F , TF }
5694 {
5695   \cs_set_protected:Npn \clist_tmp:w ##1 , #2 , ##2##3 \q_stop
5696   {
5697     \if_meaning:w \q_no_value ##2
5698     \prg_return_false:
5699     \else:
5700       \prg_return_true:
5701     \fi:
5702   }
5703   \exp_last_unbraced:NNo \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
5704 }
5705 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2
5706 { T , F , TF }
5707 {
5708   \cs_set_protected:Npn \clist_tmp:w ##1 , #2 , ##2##3 \q_stop
5709   {
5710     \if_meaning:w \q_no_value ##2
5711     \prg_return_false:
5712     \else:
5713       \prg_return_true:
5714     \fi:
5715   }
5716   \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
5717 }
5718 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }
5719 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
5720 \cs_generate_variant:Nn \clist_if_in:NnF { NV , No }
5721 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
5722 \cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }
5723 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }

```

```

5724 \cs_generate_variant:Nn \clist_if_in:nnT { nV , no }
5725 \cs_generate_variant:Nn \clist_if_in:nnF { nV , no }
5726 \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 129.)

178.7 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.

```

5727 \cs_new_nopar:Npn \clist_map_function:NN #1#2
5728 {
5729   \clist_if_empty:NF #1
5730   {
5731     \exp_last_unbraced:NNo \clist_map_function_aux:Nw #2 #1
5732     , \q_recursion_tail , \q_recursion_stop
5733   }
5734 }
5735 \cs_new:Npn \clist_map_function:nN #1#2
5736 {
5737   \tl_if_empty:nF {#1}
5738   {
5739     \clist_map_function_aux:Nw #2 #1
5740     , \q_recursion_tail , \q_recursion_stop
5741   }
5742 }
5743 \cs_new:Npn \clist_map_function_aux:Nw #1#2 ,
5744 {
5745   \quark_if_recursion_tail_stop:n {#2}
5746   #1 {#2}
5747   \clist_map_function_aux:Nw #1
5748 }
5749 \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 129.)

`\g_clist_map_inline_int` For the nesting of mappings.

```

5750 \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.

```

5751 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
5752 {
5753   \int_gincr:N \g_clist_map_inline_int
5754   \cs_gset:cpn { clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5755   ##1

```

```

5756     {#2}
5757     \exp_args:Nnc \clist_map_function:NN #1
5758     { \clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5759     \int_gdecr:N \g_clist_map_inline_int
5760   }
5761   \cs_new_protected:Npn \clist_map_inline:nn #1#2
5762   {
5763     \int_gincr:N \g_clist_map_inline_int
5764     \cs_gset:cpn { \clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5765     ##1
5766     {#2}
5767     \exp_args:Nnc \clist_map_function:nN {#1}
5768     { \clist_map_inline_ \int_use:N \g_clist_map_inline_int :n }
5769     \int_gdecr:N \g_clist_map_inline_int
5770   }
5771   \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 129.)

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

This is just a dedicated version of the inline mapping.

```

5772   \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
5773   {
5774     \clist_map_inline:Nn #1
5775     {
5776       \tl_set:Nn #2 {##1}
5777       #3
5778     }
5779   }
5780   \cs_new_protected:Npn \clist_map_variable:nNn #1#2#3
5781   {
5782     \clist_map_inline:nn {#1}
5783     {
5784       \tl_set:Nn #2 {##1}
5785       #3
5786     }
5787   }
5788   \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 130.)

`\clist_map_break:`
`\clist_map_break:n`

Both are simple renaming.

```

5789   \cs_new_eq:NN \clist_map_break: \use_none_delimit_by_q_recursion_stop:w
5790   \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 130.)

179 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`

```

5791 \cs_new_protected_nopar:Npn \clist_show:N #1
5792 {
5793   \clist_if_empty:NTF #1
5794   {
5795     \iow_term:x { Comma-list~\token_to_str:N #1 \c_space_tl is~empty }
5796     \tl_show:n { }
5797   }
5798   {
5799     \iow_term:x
5800     {
5801       Comma-list~\token_to_str:N #1 \c_space_tl
5802       contains~the~items~(without~outer~braces):
5803     }
5804     \tl_set:Nx \l_clist_show_tl
5805     { \clist_map_function:NN #1 \clist_show_aux:n }
5806     \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
5807     { \exp_after:wN \clist_show_aux:w \l_clist_show_tl }
5808   }
5809 }
5810 \cs_new:Npn \clist_show_aux:n #1
5811 {
5812   \iow_newline: > \c_space_tl \c_space_tl
5813   \iow_char:N \{ \exp_not:n {#1} \iow_char:N \}
5814 }
5815 \cs_new:Npn \clist_show_aux:w #1 > ~ { }
5816 \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for `\clist_show:N` and `\clist_show:c`. These functions are documented on page 132.)

179.1 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`

```

5817 \cs_new:Npn \clist_length:N #1
5818 {
5819   \int_eval:n
5820   {
5821     0
5822     \clist_map_function:NN #1 \clist_length_aux:n

```

```

5823     }
5824   }
5825   \cs_new:Npn \clist_length:n #1
5826   {
5827     \int_eval:n
5828     {
5829       0
5830       \clist_map_function:nN {#1} \clist_length_aux:n
5831     }
5832   }
5833   \cs_new:Npn \clist_length_aux:n #1 { +1 }
5834   \cs_generate_variant:Nn \clist_length:N { c }

```

(End definition for `\clist_length:N` and `\clist_length:c`. These functions are documented on page 132.)

`\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`

```

5835   \cs_set_nopar:Npn \clist_item:Nn #1#2
5836   { \exp_args:No \clist_item:nn #1 {#2} }
5837   \cs_set:Npn \clist_item:nn #1#2
5838   {
5839     \int_compare:nNnTF {#2} < \c_zero
5840     {
5841       \exp_args:Nf \clist_item_aux:nw
5842       { \int_eval:n { \clist_length:n {#1} + #2 } }
5843       #1 , \q_recursion_tail \q_recursion_stop
5844     }
5845     { \clist_item_aux:nw {#2} #1 , \q_recursion_tail \q_recursion_stop }
5846   }
5847   \cs_set:Npn \clist_item_aux:nw #1#2 , #3
5848   {
5849     \int_compare:nNnTF {#1} = \c_zero
5850     { \use_i_delimit_by_q_recursion_stop:nw {#2} }
5851     {
5852       \quark_if_recursion_tail_stop:n {#3}
5853       \exp_args:Nf \clist_item_aux:nw
5854       { \int_eval:n { #1 - 1 } }
5855       #3
5856     }
5857   }
5858   \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn` and `\clist_item:cn`. These functions are documented on page 133.)

`\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.

```

5859 \cs_new_protected:Npn \clist_set_from_seq:NN #1#2
5860 {
5861   \seq_if_empty:NTF #2
5862   { \clist_clear:N #1 }
5863   {
5864     \seq_push_item_def:n { , \exp_not:n {{##1}} }
5865     \tl_set:Nx #1
5866     { \exp_after:wN \use_none:n \tex_romannumeral:D -'\0 #2 }
5867     \seq_pop_item_def:
5868   }
5869 }
5870 \cs_new_protected:Npn \clist_gset_from_seq:NN #1#2
5871 {
5872   \seq_if_empty:NTF #2
5873   { \clist_gclear:N #1 }
5874   {
5875     \seq_push_item_def:n { , \exp_not:n {{##1}} }
5876     \tl_gset:Nx #1
5877     { \exp_after:wN \use_none:n \tex_romannumeral:D -'\0 #2 }
5878     \seq_pop_item_def:
5879   }
5880 }
5881 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
5882 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
5883 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
5884 \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 [133](#).)

179.2 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\clist_top:NN` These are old stack functions.

`\clist_top:cN`

```

5885 \cs_new_eq:NN \clist_top:NN \clist_get:NN
5886 \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`

```

5887 \cs_new_eq:NN \clist_remove_element:Nn \clist_remove_all:Nn
5888 \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`

```

5889 \cs_new_eq:NN \clist_display:N \clist_show:N
5890 \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
??.)

5891 </initex | package>

```

180 l3prop implementation

The following test files are used for this code: *m3prop001*.

```

5892 <*initex | package>

5893 <*package>
5894 \ProvidesExplPackage
5895   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5896 \package_check_loaded_expl:
5897 </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.

```

5898 \quark_new:N \q_prop

(End definition for \q_prop. This function is documented on page 141.)

```

`\c_empty_prop` An empty prop contains exactly one `\q_prop`.

```

5899 \tl_const:Nn \c_empty_prop { \q_prop }

```

180.1 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.

```

5900 \cs_new_protected:Npn \prop_new:N #1 { \cs_new_eq:NN #1 \c_empty_prop }
5901 \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 134.)

```

`\prop_clear:N` The same idea for clearing

`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`

```
5902 \cs_new_protected:Npn \prop_clear:N #1 { \cs_set_eq:NN #1 \c_empty_prop }
5903 \cs_new_protected:Npn \prop_clear:c #1 { \cs_set_eq:cN {#1} \c_empty_prop }
5904 \cs_new_protected:Npn \prop_gclear:N #1 { \cs_gset_eq:NN #1 \c_empty_prop }
5905 \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 134.)

`\prop_clear_new:N` Once again a simple copy from the token list functions.

`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

```
5906 \cs_new_protected:Npn \prop_clear_new:N #1
5907 { \cs_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
5908 \cs_generate_variant:Nn \prop_clear_new:N {c}
5909 \cs_new_protected:Npn \prop_gclear_new:N #1
5910 { \cs_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
5911 \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 134.)

`\prop_set_eq:NN` Once again, these are simply copies from the token list functions.

`\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`

```
5912 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
5913 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
5914 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
5915 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
5916 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
5917 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
5918 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
5919 \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 134.)

180.2 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, `{<extract1>}` `{<value>}` `{<extract2>}`. 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.

```

5920 \cs_set_protected:Npn \prop_split:NnTF #1#2
5921 { \exp_args:NNo \prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
5922 \cs_new_protected:Npn \prop_split_aux:NnTF #1#2
5923 {
5924   \cs_set_protected:Npn \prop_split_aux:w
5925     ##1 \q_prop #2 \q_prop ##2 ##3 ##4 \q_mark ##5 \q_stop
5926     { \prop_split_aux:nnnn ##3 { {##1 \q_prop } {##2} {##4} } }
5927   \exp_after:wN \prop_split_aux:w #1 \q_mark
5928     \q_prop #2 \q_prop { } { ? \use_ii:nn { } } \q_mark \q_stop
5929 }
5930 \cs_new:Npn \prop_split_aux:nnnn #1#2#3#4 { #3 #2 }
5931 \cs_new_protected:Npn \prop_split_aux:w { }

```

(End definition for $\backslash prop_split:NnTF$. This function is documented on page 141.)

$\backslash prop_split:Nnn$ 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 $\backslash q_prop$.

```

5932 \cs_set_protected:Npn \prop_split:Nnn #1#2#3
5933 {
5934   \prop_split:NnTF #1 {#2}
5935   {#3}
5936   { \exp_args:Nno \use:n {#3} {#1} { \q_no_value } { } }
5937 }

```

(End definition for $\backslash prop_split:Nnn$. This function is documented on page 141.)

$\backslash prop_del:Nn$ 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.

$\backslash prop_del:NV$
 $\backslash prop_del:cn$
 $\backslash prop_del:cV$
 $\backslash prop_gdel:Nn$
 $\backslash prop_gdel:NV$
 $\backslash prop_gdel:cn$
 $\backslash prop_gdel:cV$

```

5938 \cs_new_protected:Npn \prop_del:Nn #1#2
5939 { \prop_split:NnTF #1 {#2} { \prop_del_aux:NNnnn \tl_set:Nn #1 { } } }
5940 \cs_new_protected:Npn \prop_gdel:Nn #1#2
5941 { \prop_split:NnTF #1 {#2} { \prop_del_aux:NNnnn \tl_gset:Nn #1 { } } }
5942 \cs_new_protected:Npn \prop_del_aux:NNnnn #1#2#3#4#5
5943 { #1 #2 { #3 #5 } }
5944 \cs_generate_variant:Nn \prop_del:Nn { NV }
5945 \cs_generate_variant:Nn \prop_del:Nn { c , cV }

```

$\backslash prop_del_aux:NNnnn$

```

5946 \cs_generate_variant:Nn \prop_gdel:Nn { NV }
5947 \cs_generate_variant:Nn \prop_gdel:Nn { c , cV }

```

(End definition for `\prop_del:Nn` and others. These functions are documented on page 137.)

`\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`

```

5948 \cs_new_protected:Npn \prop_get:NnN #1#2#3
5949 {
5950   \prop_split:NnTF #1 {#2}
5951   { \prop_get_aux:Nnnn #3 }
5952   { \tl_set:Nn #3 { \q_no_value } }
5953 }
5954 \cs_new_protected:Npn \prop_get_aux:Nnnn #1#2#3#4
5955 { \tl_set:Nn #1 {#3} }
5956 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
5957 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

```

(End definition for `\prop_get:NnN` and others. These functions are documented on page 136.)

`\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`

```

5958 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
5959 {
5960   \prop_split:NnTF #1 {#2}
5961   { \prop_pop_aux:NNNnnn \tl_set:Nn #1 #3 }
5962   { \tl_set:Nn #3 { \q_no_value } }
5963 }
5964 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
5965 {
5966   \prop_split:NnTF #1 {#2}
5967   { \prop_pop_aux:NNNnnn \tl_gset:Nn #1 #3 }
5968   { \tl_set:Nn #3 { \q_no_value } }
5969 }
5970 \cs_new_protected:Npn \prop_pop_aux:NNNnnn #1#2#3#4#5#6
5971 {
5972   \tl_set:Nn #3 {#5}
5973   #1 #2 { #4 #6 }
5974 }
5975 \cs_generate_variant:Nn \prop_pop:NnN { No }
5976 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
5977 \cs_generate_variant:Nn \prop_gpop:NnN { No }
5978 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and others. These functions are documented on page 137.)

<code>\prop_put:Nnn</code>	Putting a key–value pair in a property list starts by splitting to remove any existing
<code>\prop_put:NnV</code>	value. The property list is then reconstructed with the two remaining parts #5 and #7
<code>\prop_put:Nno</code>	first, followed by the new or updated entry.

```

1979  \prop_put:Nnn { \cs_new_protected:Npn \prop_put:Nnn { \prop_put_aux:NNnn \tl_set:Nx }
1980  \cs_new_protected:Npn \prop_gput:Nnn { \prop_put_aux:NNnn \tl_gset:Nx }
1981  \cs_new_protected:Npn \prop_put_aux:NNnn #1#2#3#4
1982  {
1983    \prop_split:Nnn #2 {#3} { \prop_put_aux:NNnnnnn #1 #2 {#3} {#4} }
1984  }
1985  \cs_new_protected:Npn \prop_put_aux:NNnnnnn #1#2#3#4#5#6#7
1986  {
1987    #1 #2
1988    {
1989      \exp_not:n { #5 #7 }
1990      \tl_to_str:n {#3} \exp_not:n { \q_prop {#4} \q_prop }
1991    }
1992  }
1993  \cs_generate_variant:Nn \prop_put:Nnn
1994  { NnV , Nno , Nnx , NV , NVV , No , Noo }
1995  \cs_generate_variant:Nn \prop_put:Nnn
1996  { c , cnV , cno , cnx , cV , cVV , co , coo }
1997  \cs_generate_variant:Nn \prop_gput:Nnn
1998  { NnV , Nno , Nnx , NV , NVV , No , Noo }
1999  \cs_generate_variant:Nn \prop_gput:Nnn
2000  { c , cnV , cno , cnx , cV , cVV , co , coo }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 135.)

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`.

```

\prop_gput_if_new:cnm
\prop_gput:con
\prop_gput:coo
\prop_put_aux:NNnnnnn
6001 \cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
6002 { \prop_put_if_new_aux:NNnn \tl_put_right:Nx }
6003 \cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
6004 { \prop_put_if_new_aux:NNnn \tl_gput_right:Nx }
6005 \cs_new_protected:Npn \prop_put_if_new_aux:NNnn #1#2#3#4
6006 {
6007     \prop_split:NnTF #2 {#3}
6008     { \use_none:nnn }
6009     {
6010         #1 #2
6011         { \tl_to_str:n {#3} \exp_not:n { \q_prop {#4} \q_prop } }
6012     }
6013 }
6014 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
6015 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn` and `\prop_put_if_new:cnn`. These functions are documented on page 136.)

180.3 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!

```
6016 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p, T , F , TF }
6017 {
6018   \if_meaning:w #1 \c_empty_prop
6019   \prg_return_true:
6020   \else:
6021     \prg_return_false:
6022   \fi:
6023 }
6024 \cs_generate_variant:Nn \prop_if_empty_p:N {c}
6025 \cs_generate_variant:Nn \prop_if_empty:NTF {c}
6026 \cs_generate_variant:Nn \prop_if_empty:NT {c}
6027 \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 137.)

`\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`

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: }
}
```

`\prop_if_in_aux:w`

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.

```
6028 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
6029 {
6030   \exp_last_unbraced:Noo \prop_if_in_aux:nwn
6031   { \tl_to_str:n {#2} } #1
6032   A \q_prop { } \q_stop
6033 }
6034 \cs_new:Npn \prop_if_in_aux:nwn #1 \q_prop #2 \q_prop #3
6035 {
```

```

6036 \if_catcode:w A #2
6037 \prg_return_false:
6038 \exp_after:wN \use_none_delimit_by_q_stop:w
6039 \fi:
6040 \str_if_eq:nnT {#1} {#2}
6041 {
6042 \prg_return_true:
6043 \use_none_delimit_by_q_stop:w
6044 }
6045 \prop_if_in_aux:nwn {#1}
6046 }
6047 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
6048 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
6049 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
6050 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
6051 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
6052 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
6053 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
6054 \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 138.)

180.4 Recovering values from property lists with branching

`\prop_get:NnNTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

`\prop_get:cnNTF`

`\prop_get_aux_true:Nnnn`

```

6055 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
6056 {
6057 \prop_split:NnTF #1 {#2}
6058 { \prop_get_aux_true:Nnnn #3 }
6059 { \prg_return_false: }
6060 }
6061 \cs_new_protected:Npn \prop_get_aux_true:Nnnn #1#2#3#4
6062 {
6063 \tl_set:Nn #1 {#3}
6064 \prg_return_true:
6065 }
6066 \cs_generate_variant:Nn \prop_get:NnNT { c }
6067 \cs_generate_variant:Nn \prop_get:NnNF { c }
6068 \cs_generate_variant:Nn \prop_get:NnNTF { c }

```

(End definition for `\prop_get:NnN` and `\prop_get:cnN`. These functions are documented on page 138.)

180.5 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

```

6069 \cs_new_nopar:Npn \prop_map_function:NN #1#2
6070 {
6071   \exp_last_unbraced:NNo \prop_map_function_aux:Nwn #2
6072   #1 A \q_prop { } \q_recursion_stop
6073 }
6074 \cs_new:Npn \prop_map_function_aux:Nwn #1 \q_prop #2 \q_prop #3
6075 {
6076   \if_catcode:w A #2
6077   \exp_after:wN \prop_map_break:
6078   \fi:
6079   #1 {#2} {#3}
6080   \prop_map_function_aux:Nwn #1
6081 }
6082 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
6083 \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.

```

6084 \int_new:N \g_prop_map_inline_int

```

\prop_map_inline:Nn Mapping in line requires a nesting level counter.

\prop_map_inline:cn

```

6085 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
6086 {
6087   \int_gincr:N \g_prop_map_inline_int
6088   \cs_gset:cpn { prop_map_inline_ \int_use:N \g_prop_map_inline_int :nn }
6089   ##1##2 {#2}
6090   \prop_map_function:Nc #1
6091   { prop_map_inline_ \int_use:N \g_prop_map_inline_int :nn }
6092   \int_gdecr:N \g_prop_map_inline_int
6093 }
6094 \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.

```

6095 \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.

```

6096 \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 139.)

180.6 Viewing property lists

`\l_prop_show_tl` Used to store the material for display.

```
6097 \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_aux:n` a w-type auxiliary. To avoid a low-level `TeX` error if there is an empty property list, a simple test is used to keep the output “clean”.

`\prop_show:c`

`\prop_show_aux:n`

`\prop_show_aux:w`

```
6098 \cs_new_protected_nopar:Npn \prop_show:N #1
6099 {
6100   \prop_if_empty:NTF #1
6101   {
6102     \iow_term:x { Property-list~\token_to_str:N #1 \c_space_tl is~empty }
6103     \tl_show:n { }
6104   }
6105   {
6106     \iow_term:x
6107     {
6108       Property-list~\token_to_str:N #1 \c_space_tl
6109       contains~the~pairs~(without~outer~braces):
6110     }
6111     \tl_set:Nx \l_prop_show_tl
6112     { \prop_map_function:NN #1 \prop_show_aux:nn }
6113     \tl_show:n \exp_after:wN \exp_after:wN \exp_after:wN
6114     { \exp_after:wN \prop_show_aux:w \l_prop_show_tl }
6115   }
6116 }
6117 \cs_new:Npn \prop_show_aux:nn #1#2
6118 {
6119   \iow_newline: > \c_space_tl \c_space_tl
6120   \iow_char:N \{ #1 \iow_char:N \}
6121   \c_space_tl \c_space_tl => \c_space_tl \c_space_tl
6122   \iow_char:N \{ \exp_not:n {#2} \iow_char:N \}
6123 }
6124 \cs_new:Npn \prop_show_aux:w #1 > ~ { }
6125 \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.)

180.7 Experimental functions

`\prop_pop:NnNTF` Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

`\prop_pop:cnNTF`

`\prop_gpop:cnNTF`

`\prop_gpop:cnNTF`

`\prop_pop_aux_true:NNNnnn`

```

6126 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
6127 {
6128   \prop_split:NnTF #1 {#2}
6129   { \prop_pop_aux_true:NNNnnn \tl_set:Nn #1 #3 }
6130   { \prg_return_false: }
6131 }
6132 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
6133 {
6134   \prop_split:NnTF #1 {#2}
6135   { \prop_pop_aux_true:NNNnnn \tl_gset:Nn #1 #3 }
6136   { \prg_return_false: }
6137 }
6138 \cs_new_protected:Npn \prop_pop_aux_true:NNNnnn #1#2#3#4#5#6
6139 {
6140   \tl_set:Nn #3 {#5}
6141   #1 #2 { #4 #6 }
6142   \prg_return_true:
6143 }
6144 \cs_generate_variant:Nn \prop_pop:NnNT { c }
6145 \cs_generate_variant:Nn \prop_pop:NnNF { c }
6146 \cs_generate_variant:Nn \prop_pop:NnNTF { c }
6147 \cs_generate_variant:Nn \prop_gpop:NnNT { c }
6148 \cs_generate_variant:Nn \prop_gpop:NnNF { c }
6149 \cs_generate_variant:Nn \prop_gpop:NnNTF { c }

```

(End definition for `\prop_pop:NnN` and others. These functions are documented on page 140.)

`\prop_map_tokens:Nn` The mapping grabs one key–value pair at a time, and stops when reaching the marker
`\prop_map_tokens:cn` key A, with catcode “letter”, which cannot appear in normal keys since those are strings.
`\prop_map_tokens_aux:nwn` The odd construction `\use:n {#1}` allows #1 to contain any token.

```

6150 \cs_new:Npn \prop_map_tokens:Nn #1#2
6151 {
6152   \exp_last_unbraced:Nno \prop_map_tokens_aux:nwn {#2} #1
6153   A \q_prop { } \q_recursion_stop
6154 }
6155 \cs_new:Npn \prop_map_tokens_aux:nwn #1 \q_prop #2 \q_prop #3
6156 {
6157   \if_catcode:w A #2
6158   \exp_after:wN \prop_map_break:
6159   \fi:
6160   \use:n {#1} {#2} {#3}
6161   \prop_map_tokens_aux:nwn {#1}
6162 }
6163 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `\prop_map_tokens:cn`. These functions are documented on page 140.)

`\prop_get:Nn` Getting the value corresponding to a key in a property list in an expandable fashion is a
`\prop_get:Nn` simple instance of mapping some tokens. Map the function `\prop_get_aux:nnn` which
`\prop_get_aux:nnn`

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.

```

6164 \cs_new:Npn \prop_get:Nn #1#2
6165   { \prop_map_tokens:Nn #1 { \prop_get_aux:nnn {#2} } }
6166 \cs_new:Npn \prop_get_aux:nnn #1#2#3
6167   { \str_if_eq:nnT {#1} {#2} { \prop_map_break:n {#3} } }
6168 \cs_generate_variant:Nn \prop_get:Nn { c }

```

(End definition for `\prop_get:Nn` and `\prop_get:Nn`. These functions are documented on page 140.)

180.8 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`

```

6169 \cs_new_eq:NN \prop_display:N \prop_show:N
6170 \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.

```

6171 \cs_new_protected:Npn \prop_gget:NnN #1#2#3
6172   { \prop_split:Nnn #1 {#2} { \prop_gget_aux:Nnnn #3 } }
6173 \cs_new_protected:Npn \prop_gget_aux:Nnnn #1#2#3#4
6174   { \tl_gset:Nn #1 {#3} }
6175 \cs_generate_variant:Nn \prop_gget:NnN { NV }
6176 \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.

```

6177 \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

```

6178 \cs_generate_variant:Nn \prop_if_in:NnT { cc }
6179 \cs_generate_variant:Nn \prop_if_in:NnF { cc }
6180 \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.

```
6181 \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` These ones do no even make sense!

```
\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
```

```
6182 \prg_new_eq_conditional:NNn \prop_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
6183 \prg_new_eq_conditional:NNn \prop_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
6184 \prg_new_eq_conditional:NNn \prop_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
6185 \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 ??.)

```
6186 </initex | package>
```

181 l3box implementation

```
6187 <*initex | package>
6188 <*package>
6189 \ProvidesExplPackage
6190 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
6191 \package_check_loaded_expl:
6192 </package>
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

181.1 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

`\box_new:N` Defining a new `<box>` register: remember that box 255 is not generally available.

```
\box_new:c
6193 <*package>
6194 \cs_new_protected:Npn \box_new:N #1
6195 {
6196   \chk_if_free_cs:N #1
6197   \newbox #1
6198 }
6199 </package>
6200 \cs_generate_variant:Nn \box_new:N { c }
```

`\box_clear:N` Clear a `<box>` register.

```
\box_clear:c
\box_gclear:N
\box_gclear:c
```

```
6201 \cs_new_protected_nopar:Npn \box_clear:N #1
6202 { \box_set_eq:NN #1 \c_empty_box }
```

```

6203 \cs_new_protected_nopar:Npn \box_gclear:N #1
6204 { \box_gset_eq:NN #1 \c_empty_box }
6205 \cs_generate_variant:Nn \box_clear:N { c }
6206 \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
6207 \cs_new_protected_nopar:Npn \box_clear_new:N #1
6208 {
6209   \cs_if_exist:NTF #1
6210     { \box_set_eq:NN #1 \c_empty_box }
6211     { \box_new:N #1 }
6212 }
6213 \cs_new_protected_nopar:Npn \box_gclear_new:N #1
6214 {
6215   \cs_if_exist:NTF #1
6216     { \box_gset_eq:NN #1 \c_empty_box }
6217     { \box_new:N #1 }
6218 }
6219 \cs_generate_variant:Nn \box_clear_new:N { c }
6220 \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
6221 \cs_new_protected_nopar:Npn \box_set_eq:NN #1#2
6222 { \tex_setbox:D #1 \tex_copy:D #2 }
6223 \cs_new_protected_nopar:Npn \box_gset_eq:NN
6224 { \pref_global:D \box_set_eq:NN }
6225 \cs_generate_variant:Nn \box_set_eq:NN { cN , Nc , cc }
6226 \cs_generate_variant:Nn \box_gset_eq:NN { cN , Nc , cc }
\box_gset_eq: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
6227 \cs_new_protected_nopar:Npn \box_set_eq_clear:NN #1#2
6228 { \tex_setbox:D #1 \tex_box:D #2 }
6229 \cs_new_protected_nopar:Npn \box_gset_eq_clear:NN
6230 { \pref_global:D \box_set_eq_clear:NN }
6231 \cs_generate_variant:Nn \box_set_eq_clear:NN { cN , Nc , cc }
6232 \cs_generate_variant:Nn \box_gset_eq_clear:NN { cN , Nc , cc }
\box_gset_eq_clear:cc

```

181.2 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
6233 \cs_new_eq:NN \box_ht:N \tex_ht:D
6234 \cs_new_eq:NN \box_dp:N \tex_dp:D
6235 \cs_new_eq:NN \box_wd:N \tex_wd:D

```

```

6236 \cs_generate_variant:Nn \box_ht:N { c }
6237 \cs_generate_variant:Nn \box_dp:N { c }
6238 \cs_generate_variant:Nn \box_wd:N { c }

```

`\box_set_ht:Nn` `\box_set_ht:cn` `\box_set_dp:Nn` `\box_set_dp:cn` `\box_set_wd:Nn` `\box_set_wd:cn` Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```

6239 \cs_new_protected_nopar:Npn \box_set_dp:Nn #1#2
6240 { \box_dp:N #1 \dim_eval:w #2 \dim_eval_end: }
6241 \cs_new_protected_nopar:Npn \box_set_ht:Nn #1#2
6242 { \box_ht:N #1 \dim_eval:w #2 \dim_eval_end: }
6243 \cs_new_protected_nopar:Npn \box_set_wd:Nn #1#2
6244 { \box_wd:N #1 \dim_eval:w #2 \dim_eval_end: }
6245 \cs_generate_variant:Nn \box_set_ht:Nn { c }
6246 \cs_generate_variant:Nn \box_set_dp:Nn { c }
6247 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

181.3 Using boxes

`\box_use_clear:N` `\box_use_clear:c` `\box_use:N` `\box_use:c` Using a $\langle box \rangle$. These are just TeX primitives with meaningful names.

```

6248 \cs_new_eq:NN \box_use_clear:N \tex_box:D
6249 \cs_new_eq:NN \box_use:N \tex_copy:D
6250 \cs_generate_variant:Nn \box_use_clear:N { c }
6251 \cs_generate_variant:Nn \box_use:N { c }

```

`\box_move_left:nn` `\box_move_right:nn` `\box_move_up:nn` `\box_move_down:nn` Move box material in different directions.

```

6252 \cs_new_protected:Npn \box_move_left:nn #1#2
6253 { \tex_moveleft:D \dim_eval:w #1 \dim_eval_end: #2 }
6254 \cs_new_protected:Npn \box_move_right:nn #1#2
6255 { \tex_moveright:D \dim_eval:w #1 \dim_eval_end: #2 }
6256 \cs_new_protected:Npn \box_move_up:nn #1#2
6257 { \tex_raise:D \dim_eval:w #1 \dim_eval_end: #2 }
6258 \cs_new_protected:Npn \box_move_down:nn #1#2
6259 { \tex_lower:D \dim_eval:w #1 \dim_eval_end: #2 }

```

181.4 Box conditionals

`\if_hbox:N` `\if_vbox:N` `\if_box_empty:N` The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

6260 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
6261 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
6262 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:N $\underline{TF}$ 
\box_if_horizontal:c $\underline{TF}$ 
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N $\underline{TF}$ 
\box_if_vertical:c $\underline{TF}$ 

```

```

6263 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
6264 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6265 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
6266 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6267 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
6268 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
6269 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
6270 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
6271 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
6272 \cs_generate_variant:Nn \box_if_vertical:NT { c }
6273 \cs_generate_variant:Nn \box_if_vertical:NF { c }
6274 \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:N \underline{TF}`

`\box_if_empty:c \underline{TF}`

```

6275 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
6276 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
6277 \cs_generate_variant:Nn \box_if_empty_p:N { c }
6278 \cs_generate_variant:Nn \box_if_empty:NT { c }
6279 \cs_generate_variant:Nn \box_if_empty:NF { c }
6280 \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 145.)

181.5 The last box inserted

`\l_last_box` A different name for this read-only primitive.

```

6281 \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`

```

6282 \cs_new_protected_nopar:Npn \box_set_to_last:N #1
6283 { \tex_setbox:D #1 \l_last_box }
6284 \cs_new_protected_nopar:Npn \box_gset_to_last:N
6285 { \pref_global:D \box_set_to_last:N }
6286 \cs_generate_variant:Nn \box_set_to_last:N { c }
6287 \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 ??.)

181.6 Constant boxes

`\c_empty_box`

```

6288 \<package>
6289 \cs_new_eq:NN \c_empty_box \voidb@x

```

```

6290 </package>
6291 <*initex>
6292 \box_new:N \c_empty_box
6293 </initex>

```

181.7 Scratch boxes

```

\l_tmpa_box
\l_tmpb_box
6294 <*package>
6295 \cs_new_eq:NN \l_tmpa_box \@tempboxa
6296 </package>
6297 <*initex>
6298 \box_new:N \l_tmpa_box
6299 </initex>
6300 \box_new:N \l_tmpb_box

```

181.8 Viewing box contents

\box_show:N Show the contents of a box and write it into the log file.
\box_show:c

```

6301 \cs_new_eq:NN \box_show:N \tex_showbox:D
6302 \cs_generate_variant:Nn \box_show:N { c }

```

(End definition for `\box_show:N` and `\box_show:c`. These functions are documented on page 146.)

181.9 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.

```

6303 \cs_new_protected_nopar:Npn \hbox:n { \tex_hbox:D \scan_stop: }

```

(End definition for `\hbox:n`. This function is documented on page 147.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
6304 \cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
6305 \cs_new_protected_nopar:Npn \hbox_gset:Nn { \pref_global:D \hbox_set:Nn }
6306 \cs_generate_variant:Nn \hbox_set:Nn { c }
6307 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End definition for `\hbox_set:Nn` and `\hbox_set:cn`. These functions are documented on page 147.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width.

`\hbox_set_to_wd:cnn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn`

```
6308 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
6309 { \tex_setbox:D #1 \tex_hbox:D to \dim_eval:w #2 \dim_eval_end: {#3} }
6310 \cs_new_protected_nopar:Npn \hbox_gset_to_wd:Nnn
6311 { \pref_global:D \hbox_set_to_wd:Nnn }
6312 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
6313 \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 147.)

`\hbox_set_inline_begin:N` Storing material in a horizontal box. This type is useful in environment definitions.

`\hbox_set_inline_begin:c`
`\hbox_gset_inline_begin:N`
`\hbox_gset_inline_begin:c`
`\hbox_set_inline_end:`
`\hbox_gset_inline_end:`

```
6314 \cs_new_protected_nopar:Npn \hbox_set_inline_begin:N #1
6315 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
6316 \cs_new_protected_nopar:Npn \hbox_gset_inline_begin:N
6317 { \pref_global:D \hbox_set_inline_begin:N }
6318 \cs_generate_variant:Nn \hbox_set_inline_begin:N { c }
6319 \cs_generate_variant:Nn \hbox_gset_inline_begin:N { c }
6320 \cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token
6321 \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 148.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

`\hbox_to_zero:n`

```
6322 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
6323 { \tex_hbox:D to \dim_eval:w #1 \dim_eval_end: {#2} }
6324 \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 147.)

`\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.

```
6325 \cs_new_protected:Npn \hbox_overlap_left:n #1
6326 { \hbox_to_zero:n { \tex_hss:D #1 } }
6327 \cs_new_protected:Npn \hbox_overlap_right:n #1
6328 { \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 147.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

`\hbox_unpack:c`
`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

```
6329 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
6330 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
6331 \cs_generate_variant:Nn \hbox_unpack:N { c }
6332 \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 148.)

181.10 Vertical mode boxes

\vbox:n *The following test files are used for this code: m3box003.lvt.*

\vbox_top:n *The following test files are used for this code: m3box003.lvt.*

Put a vertical box directly into the input stream.

```
6333 \cs_new_protected_nopar:Npn \vbox:n { \tex_vbox:D \scan_stop: }
6334 \cs_new_protected_nopar:Npn \vbox_top:n { \tex_vtop:D \scan_stop: }
```

(End definition for \vbox:n. This function is documented on page 149.)

\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

```
6335 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
6336 { \tex_vbox:D to \dim_eval:w #1 \dim_eval_end: {#2} }
6337 \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 149.)

\vbox_set:Nn Storing material in a vertical box with a natural height.

\vbox_set:cn

\vbox_gset:Nn

\vbox_gset:cn

```
6338 \cs_new_protected:Npn \vbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_vbox:D {#2} }
6339 \cs_new_protected_nopar:Npn \vbox_gset:Nn { \pref_global:D \vbox_set:Nn }
6340 \cs_generate_variant:Nn \vbox_set:Nn { c }
6341 \cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(End definition for \vbox_set:Nn and \vbox_set:cn. These functions are documented on page 149.)

\vbox_set_top:Nn Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

\vbox_set_top:cn

\vbox_gset_top:Nn

\vbox_gset_top:cn

```
6342 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
6343 { \tex_setbox:D #1 \tex_vtop:D {#2} }
6344 \cs_new_protected_nopar:Npn \vbox_gset_top:Nn
6345 { \pref_global:D \vbox_set_top:Nn }
6346 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
6347 \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 150.)

\vbox_set_to_ht:Nnn Storing material in a vertical box with a specified height.

\vbox_set_to_ht:cn

\vbox_gset_to_ht:Nnn

\vbox_gset_to_ht:cn

```
6348 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
6349 { \tex_setbox:D #1 \tex_vbox:D to \dim_eval:w #2 \dim_eval_end: {#3} }
6350 \cs_new_protected_nopar:Npn \vbox_gset_to_ht:Nnn
6351 { \pref_global:D \vbox_set_to_ht:Nnn }
6352 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
6353 \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 150.)

`\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:`

```
6354 \cs_new_nopar:Npn \vbox_set_inline_begin:N #1
6355 { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
6356 \cs_new_protected_nopar:Npn \vbox_gset_inline_begin:N
6357 { \pref_global:D \vbox_set_inline_begin:N }
6358 \cs_generate_variant:Nn \vbox_set_inline_begin:N { c }
6359 \cs_generate_variant:Nn \vbox_gset_inline_begin:N { c }
6360 \cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token
6361 \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 150.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.
`\vbox_unpack:c`
`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

```
6362 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
6363 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
6364 \cs_generate_variant:Nn \vbox_unpack:N { c }
6365 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }
```

(End definition for `\vbox_unpack:N` and `\vbox_unpack:c`. These functions are documented on page 151.)

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

```
6366 \cs_new_protected_nopar:Npn \vbox_set_split_to_ht:NNn #1#2#3
6367 { \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 151.)

```
6368 </initex | package>
```

182 l3io implementation

```
6369 <*initex | package>
6370 <*package>
6371 \ProvidesExplPackage
6372 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
6373 \package_check_loaded_expl:
6374 </package>
```

182.1 Primitives

`\if_eof:w` The primitive conditional

```
6375 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for `\if_eof:w`. This function is documented on page 157.)

182.2 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.

```
6376 \cs_new_eq:NN \c_iow_term_stream \c_sixteen
6377 \cs_new_eq:NN \c_ior_term_stream \c_sixteen
6378 \cs_new_eq:NN \c_iow_log_stream \c_minus_one
6379 \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
6380 \tl_const:Nn \c_iow_streams_tl
6381 {
6382   \c_zero
6383   \c_one
6384   \c_two
6385   \c_three
6386   \c_four
6387   \c_five
6388   \c_six
6389   \c_seven
6390   \c_eight
6391   \c_nine
6392   \c_ten
6393   \c_eleven
6394   \c_twelve
6395   \c_thirteen
6396   \c_fourteen
6397   \c_fifteen
6398 }
6399 \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” set of allocations from the start. In package mode, a few slots are always taken, so these are blocked off from use.

```
6400 \prop_new:N \g_iow_streams_prop
6401 \prop_new:N \g_ior_streams_prop
6402 \*package
6403 \prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e-reserved }
6404 \prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e-reserved }
6405 \prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e-reserved }
6406 \prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e-reserved }
6407 \*package
```

`\l_iow_stream_int` Used to track the number allocated to the stream being created: this is taken from the property list but does alter.

`\l_ior_stream_int`

```

6408 \int_new:N \l_iow_stream_int
6409 \cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int

```

182.3 Stream management

`\ior_raw_new:N` The lowest level for stream management is actually creating raw T_EX streams. As these are very limited (even with ε -T_EX), this should not be addressed directly.

```

\ior_raw_new:c
\ior_raw_new:N
\ior_raw_new:c
6410 <*initex>
6411 \alloc_setup_type:nnn { ior } \c_zero \c_sixteen
6412 \cs_new_protected_nopar:Npn \ior_raw_new:N #1
6413 { \alloc_reg:nnn { ior } \tex_chardef:D #1 }
6414 \alloc_setup_type:nnn { iow } \c_zero \c_sixteen
6415 \cs_new_protected_nopar:Npn \iow_raw_new:N #1
6416 { \alloc_reg:nnn { iow } \tex_chardef:D #1 }
6417 </initex>
6418 <*package>
6419 \cs_set_eq:NN \iow_raw_new:N \newwrite
6420 \cs_set_eq:NN \ior_raw_new:N \newread
6421 </package>
6422 \cs_generate_variant:Nn \ior_raw_new:N { c }
6423 \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 157.)

`\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.

```

6424 \cs_new_protected_nopar:Npn \ior_open:Nn #1#2
6425 {
6426   \ior_close:N #1
6427   \int_set:Nn \l_ior_stream_int \c_sixteen
6428   \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
6429   \int_compare:nNnTF \l_ior_stream_int = \c_sixteen
6430   { \msg_kernel_error:nn { ior } { streams-exhausted } }
6431   {
6432     \ior_stream_alloc:N #1
6433     \prop_gput:NVn \g_ior_streams_prop \l_ior_stream_int {#2}
6434     \tex_openin:D #1#2 \scan_stop:
6435   }
6436 }
6437 \cs_new_protected_nopar:Npn \iow_open:Nn #1#2
6438 {
6439   \iow_close:N #1
6440   \int_set:Nn \l_iow_stream_int \c_sixteen
6441   \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n

```

```

6442 \int_compare:nNnTF \l_iow_stream_int = \c_sixteen
6443 { \msg_kernel_error:nn { iow } { streams-exhausted } }
6444 {
6445   \iow_stream_alloc:N #1
6446   \prop_gput:NVn \g_iow_streams_prop \l_iow_stream_int {#2}
6447   \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
6448 }
6449 }
6450 \cs_generate_variant:Nn \ior_open:Nn { c }
6451 \cs_generate_variant:Nn \iow_open:Nn { c }

```

(End definition for `\ior_open:Nn` and `\iow_open:Nn`. These functions are documented on page 153.)

`\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.

```

6452 \cs_new_protected_nopar:Npn \iow_alloc_write:n #1
6453 {
6454   \prop_if_in:NnF \g_iow_streams_prop {#1}
6455   {
6456     \int_set:Nn \l_iow_stream_int {#1}
6457     \tl_map_break:
6458   }
6459 }
6460 \cs_new_protected_nopar:Npn \ior_alloc_read:n #1
6461 {
6462   \prop_if_in:NnF \g_iow_streams_prop {#1}
6463   {
6464     \int_set:Nn \l_ior_stream_int {#1}
6465     \tl_map_break:
6466   }
6467 }

```

(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.

```

6468 \cs_new_protected_nopar:Npn \iow_stream_alloc:N #1
6469 {
6470   \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
6471   { \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream } }
6472   {
6473     <*package>
6474     \iow_stream_alloc_aux:
6475     \int_compare:nNnT \l_iow_stream_int = \c_sixteen

```

```

6476         {
6477             \iow_raw_new:N \g_iow_tmp_stream
6478             \int_set:Nn \l_iow_stream_int { \g_iow_tmp_stream }
6479             \cs_gset_eq:cN
6480                 { \g_iow_ \int_use:N \l_iow_stream_int _stream }
6481             \g_iow_tmp_stream
6482         }
6483     </package>
6484     <*initex>
6485         \iow_raw_new:c { \g_iow_ \int_use:N \l_iow_stream_int _stream }
6486     </initex>
6487     \cs_gset_eq:Nc #1 { \g_iow_ \int_use:N \l_iow_stream_int _stream }
6488 }
6489 }
6490 <*package>
6491 \cs_new_protected_nopar:Npn \iow_stream_alloc_aux:
6492 {
6493     \int_incr:N \l_iow_stream_int
6494     \int_compare:nNnT \l_iow_stream_int < \c_sixteen
6495     {
6496         \cs_if_exist:cTF { \g_iow_ \int_use:N \l_iow_stream_int _stream }
6497         {
6498             \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
6499                 { \iow_stream_alloc_aux: }
6500         }
6501         { \iow_stream_alloc_aux: }
6502     }
6503 }
6504 </package>
6505 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1
6506 {
6507     \cs_if_exist:cTF { \g_ior_ \int_use:N \l_ior_stream_int _stream }
6508     { \cs_gset_eq:Nc #1 { \g_ior_ \int_use:N \l_ior_stream_int _stream } }
6509     {
6510 <*package>
6511         \ior_stream_alloc_aux:
6512         \int_compare:nNnT \l_ior_stream_int = \c_sixteen
6513         {
6514             \ior_raw_new:N \g_ior_tmp_stream
6515             \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }
6516             \cs_gset_eq:cN
6517                 { \g_ior_ \int_use:N \l_ior_stream_int _stream }
6518             \g_ior_tmp_stream
6519         }
6520 </package>
6521 <*initex>
6522         \ior_raw_new:c { \g_ior_ \int_use:N \l_ior_stream_int _stream }
6523 </initex>
6524     \cs_gset_eq:Nc #1 { \g_ior_ \int_use:N \l_ior_stream_int _stream }
6525 }

```

```

6526 }
6527 <*package>
6528 \cs_new_protected_nopar:Npn \ior_stream_alloc_aux:
6529 {
6530   \int_incr:N \l_ior_stream_int
6531   \int_compare:nNnT \l_ior_stream_int < \c_sixteen
6532   {
6533     \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
6534     {
6535       \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int
6536       { \ior_stream_alloc_aux: }
6537     }
6538     { \ior_stream_alloc_aux: }
6539   }
6540 }
6541 </package>

```

(End definition for `\ior_stream_alloc:N`.)

`\ior_close:N` Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```

\ior_close:c
\ior_close:N
\ior_close:c
6542 \cs_new_protected_nopar:Npn \ior_close:N #1
6543 {
6544   \cs_if_exist:NT #1
6545   {
6546     \int_compare:nNnF #1 = \c_minus_one
6547     {
6548       \tex_closein:D #1
6549       \prop_gdel:NV \g_ior_streams_prop #1
6550       \cs_undefine:N #1
6551     }
6552   }
6553 }
6554 \cs_new_protected_nopar:Npn \ior_close:N #1
6555 {
6556   \cs_if_exist:NT #1
6557   {
6558     \int_compare:nNnF #1 = \c_minus_one
6559     {
6560       \tex_immediate:D \tex_closeout:D #1
6561       \prop_gdel:NV \g_iow_streams_prop #1
6562       \cs_undefine:N #1
6563     }
6564   }
6565 }
6566 \cs_generate_variant:Nn \ior_close:N { c }
6567 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N` and `\ior_close:c`. These functions are documented on page 153.)

`\ior_list_streams:` Show the property lists, but with some “pretty printing”.

```

\ior_list_streams:
\ior_list_streams:
\ior_show_aux:nn
\ior_show_aux:nn
6568 \cs_new_protected_nopar:Npn \ior_list_streams:
6569 {
6570   \prop_if_empty:NTF \g_ior_streams_prop
6571   {
6572     \ior_term:x { No~input~streams~are~open }
6573     \tl_show:n { }
6574   }
6575   {
6576     \ior_term:x { The~following~input~streams~are~in~use: }
6577     \tl_set:Nx \l_prop_show_tl
6578     { \prop_map_function:NN \g_ior_streams_prop \ior_show_aux:nn }
6579     \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
6580     { \exp_after:wN \prop_show_aux:w \l_prop_show_tl }
6581   }
6582 }
6583 \cs_new:Npn \ior_show_aux:nn #1#2
6584 {
6585   \ior_newline: > \c_space_tl \c_space_tl
6586   #1 \ior_char:N
6587   \c_space_tl \c_space_tl => \c_space_tl \c_space_tl
6588   \exp_not:n {#2}
6589 }
6590 \cs_new_protected_nopar:Npn \ior_list_streams:
6591 {
6592   \prop_if_empty:NTF \g_iow_streams_prop
6593   {
6594     \ior_term:x { No~output~streams~are~open }
6595     \tl_show:n { }
6596   }
6597   {
6598     \ior_term:x { The~following~output~streams~are~in~use: }
6599     \tl_set:Nx \l_prop_show_tl
6600     { \prop_map_function:NN \g_iow_streams_prop \ior_show_aux:nn }
6601     \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
6602     { \exp_after:wN \prop_show_aux:w \l_prop_show_tl }
6603   }
6604 }
6605 \cs_new_eq:NN \ior_show_aux:nn \ior_show_aux:nn

```

(End definition for `\ior_list_streams:`. This function is documented on page 153.)

Text for the error messages.

```

6606 \msg_kernel_new:nnnn { iow } { streams-exhausted }
6607 { Output~streams~exhausted }
6608 {
6609   TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
6610   All~16~are~currently~in~use,~and~something~wanted~to~open
6611   another~one.

```

```

6612 }
6613 \msg_kernel_new:nnnn { ior } { streams-exhausted }
6614 { Input~streams-exhausted }
6615 {
6616   TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
6617   All~16 are currently~in~use,~and~something~wanted~to~open
6618   another~one.
6619 }

```

182.4 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive.

`\iow_shipout_x:Nx`

```

6620 \cs_new_eq:NN \iow_shipout_x:Nn \tex_write:D
6621 \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 154.)

`\iow_shipout:Nn` With ε -TeX available deferred writing is easy.

`\iow_shipout:Nx`

```

6622 \cs_new_protected_nopar:Npn \iow_shipout:Nn #1#2
6623 { \iow_shipout_x:Nn #1 { \exp_not:n {#2} } }
6624 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }

```

(End definition for `\iow_shipout:Nn` and `\iow_shipout:Nx`. These functions are documented on page 154.)

182.5 Immediate writing

`\iow_now:Nx` An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```

6625 \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 153.)

`\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.

```

6626 \cs_new_protected_nopar:Npn \iow_now:Nn #1#2
6627 { \iow_now:Nx #1 { \exp_not:n {#2} } }

```

(End definition for `\iow_now:Nn`. This function is documented on page 153.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.

`\iow_log:x`

`\iow_term:n`

`\iow_term:x`

```

6628 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_iow_log_stream }
6629 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_iow_log_stream }
6630 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
6631 \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 154.)

`\iow_now_when_avail:Nn` For writing only if the stream requested is open at all.

`\iow_now_when_avail:Nx`

```

6632 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nn #1
6633 { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 } }
6634 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nx #1
6635 { \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 154.)

182.6 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 `TeXLive` and `MikTeX`.

```

6636 \int_new:N \l_iow_line_length_int
6637 \int_set:Nn \l_iow_line_length_int { 78 }

```

(End definition for `\l_iow_line_length_int`. This function is documented on page 155.)

`\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.

```

6638 \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, respectively.

`\l_iow_current_word_int`

```

6639 \int_new:N \l_iow_current_line_int
6640 \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`

```

6641 \tl_new:N \l_iow_current_line_tl
6642 \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.

```
6643 \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.

```
6644 \tl_new:N \l_iow_wrapped_tl
```

(End definition for `\l_iow_wrapped_tl`.)

`\q_iow_stop` A quark which will not appear elsewhere.

```
6645 \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.

```
6646 \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.

```
\iow_wrap_loop:w
\iow_wrap_word:
\iow_wrap_word_fits:
\iow_wrap_word_newline:
\iow_wrap_newline:
\iow_wrap_end:
```

```
6647 \cs_new_protected:Npn \iow_wrap:xnnnN #1#2#3#4#5
6648 {
6649   \group_begin:
6650   \int_set:Nn \l_iow_target_length_int { \l_iow_line_length_int - ( #3 ) }
6651   \int_zero:N \l_iow_current_line_int
6652   \tl_clear:N \l_iow_current_line_tl
6653   \tl_clear:N \l_iow_wrap_tl
6654   \bool_set_true:N \l_iow_line_start_bool
6655   \cs_set:Npx \\ { \c_space_tl \iow_newline: \c_space_tl }
6656   \cs_set_eq:NN \ \c_space_tl
6657   #4
6658   \<initex>
6659   \tl_set:Nx \l_iow_wrap_tl {#1}
6660   \</initex>
6661   \<package>
6662   \protected@edef \l_iow_wrap_tl {#1}
6663   \</package>
```

```

6664 \cs_set:Npn \\\ { \iow_newline: #2 }
6665 \use:x
6666 {
6667   \exp_not:N \iow_wrap_loop:w
6668   \tl_to_str:N \l_iow_wrap_tl \c_space_tl
6669   \exp_not:N \q_iow_stop \c_space_tl
6670 }
6671 \exp_args:NNo \group_end:
6672 #5 \l_iow_wrapped_tl
6673 }

```

The loop grabs one word in the input, and checks whether it is the end, or a forced new line, or a normal word.

```

6674 \cs_new_protected:Npn \iow_wrap_loop:w #1 ~ %
6675 {
6676   \tl_set:Nn \l_iow_current_word_tl {#1}
6677   \tl_if_eq:NNTF \l_iow_current_word_tl \iow_newline:
6678   { \iow_wrap_newline: }
6679   {
6680     \tl_if_eq:NNTF \l_iow_current_word_tl \q_iow_stop
6681     { \iow_wrap_end: }
6682     { \iow_wrap_word: }
6683   }
6684 }

```

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.

```

6685 \cs_new_protected_nopar:Npn \iow_wrap_word:
6686 {
6687   \int_set:Nn \l_iow_current_word_int
6688   { \str_length_skip_spaces:N \l_iow_current_word_tl }
6689   \int_add:Nn \l_iow_current_line_int { \l_iow_current_word_int }
6690   \int_compare:nNnTF \l_iow_current_line_int
6691   < \l_iow_target_length_int
6692   { \iow_wrap_word_fits: }
6693   { \iow_wrap_word_newline: }
6694   \iow_wrap_loop:w
6695 }

```

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.

```

6696 \cs_new_protected_nopar:Npn \iow_wrap_word_fits:
6697 {
6698   \bool_if:NNTF \l_iow_line_start_bool
6699   {
6700     \bool_set_false:N \l_iow_line_start_bool
6701     \tl_set_eq:NN \l_iow_current_line_tl \l_iow_current_word_tl
6702   }

```

```

6703     {
6704         \tl_put_right:Nx \l_iow_current_line_tl
6705         { ~ \l_iow_current_word_tl }
6706         \int_incr:N \l_iow_current_line_int
6707     }
6708 }

```

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.

```

6709 \cs_new_protected_nopar:Npn \iow_wrap_word_newline:
6710 {
6711     \tl_put_right:Nx \l_iow_wrapped_tl
6712     { \l_iow_current_line_tl \ }
6713     \int_set_eq:NN \l_iow_current_line_int \l_iow_current_word_int
6714     \tl_set_eq:NN \l_iow_current_line_tl \l_iow_current_word_tl
6715 }

```

Forced newlines are almost identical to those caused by overflow, except that here the word is empty. And remember to continue the loop!

```

6716 \cs_new_protected_nopar:Npn \iow_wrap_newline:
6717 {
6718     \tl_put_right:Nx \l_iow_wrapped_tl
6719     { \l_iow_current_line_tl \ }
6720     \int_zero:N \l_iow_current_line_int
6721     \tl_clear:N \l_iow_current_line_tl
6722     \bool_set_true:N \l_iow_line_start_bool
6723     \iow_wrap_loop:w
6724 }

```

At the end, we simply save the last line (without the run-on text).

```

6725 \cs_new_protected_nopar:Npn \iow_wrap_end:
6726 {
6727     \tl_put_right:Nx \l_iow_wrapped_tl
6728     { \l_iow_current_line_tl }
6729 }

```

(End definition for `\iow_wrap:nnnn`. This function is documented on page 155.)

```

\str_length_skip_spaces:N
\str_length_skip_spaces:n
\str_length_loop:NNNNNNNN

```

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.

```

6730 \cs_new_nopar:Npn \str_length_skip_spaces:N
6731 { \exp_args:No \str_length_skip_spaces:n }
6732 \cs_new:Npn \str_length_skip_spaces:n #1
6733 {
6734     \int_value:w \int_eval:w
6735     \exp_after:wN \str_length_loop:NNNNNNNN \tl_to_str:n {#1}
6736     {X8}{X7}{X6}{X5}{X4}{X3}{X2}{X1}{X0} \q_stop

```

```

6737     \int_eval_end:
6738   }
6739   \cs_new:Npn \str_length_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
6740   {
6741     \if_catcode:w X #9
6742       \exp_after:wN \use_none_delimit_by_q_stop:w
6743     \else:
6744       9 +
6745       \exp_after:wN \str_length_loop:NNNNNNNNN
6746     \fi:
6747   }

```

(End definition for `\str_length_skip_spaces:N`. This function is documented on page ??.)

182.7 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream

```

6748 \cs_new_nopar:Npn \iow_newline: { ^^J }

```

(End definition for `\iow_newline:.` This function is documented on page 155.)

`\iow_char:N` Function to write any escaped char to an output stream.

```

6749 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End definition for `\iow_char:N`. This function is documented on page 154.)

182.8 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:NTF` As the pool model means that closed streams are undefined control sequences, the test has two parts.

```

6750 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
6751 {
6752   \cs_if_exist:NTF #1
6753   {
6754     \if_eof:w #1
6755       \prg_return_true:
6756     \else:
6757       \prg_return_false:
6758     \fi:
6759   }
6760   { \prg_return_true: }
6761 }

```

(End definition for `\ior_if_eof_p:N`. These functions are documented on page 156.)

`\ior_to:NN` And here we read from files.

`\ior_gto:NN`

```
6762 \cs_new_protected_nopar:Npn \ior_to:NN #1#2
6763 { \tex_read:D #1 to #2 }
6764 \cs_new_protected_nopar:Npn \ior_gto:NN #1#2
6765 { \pref_global:D \tex_read:D #1 to #2 }
```

(End definition for \ior_to:NN. This function is documented on page 156.)

`\ior_str_to:NN` Reading as strings is also a primitive wrapper.

`\ior_str_gto:NN`

```
6766 \cs_new_protected_nopar:Npn \ior_str_to:NN #1#2
6767 { \etex_readline:D #1 to #2 }
6768 \cs_new_protected_nopar:Npn \ior_str_gto:NN #1#2
6769 { \pref_global:D \etex_readline:D #1 to #2 }
```

(End definition for \ior_str_to:NN. This function is documented on page 156.)

182.9 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.

```
6770 \cs_new_protected:Npn \iow_now_buffer_safe:Nn #1#2
6771 { \iow_wrap:xnnnN { \exp_not:n {#2} } { } \c_zero { } \iow_now:Nn #1 }
6772 \cs_new_protected:Npn \iow_now_buffer_safe:Nx #1#2
6773 { \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`

```
6774 \cs_new_eq:NN \ior_new:N \use_none:n
6775 \cs_new_eq:NN \ior_new:c \use_none:n
6776 \cs_new_eq:NN \iow_new:N \use_none:n
6777 \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:`

```
6778 \cs_new_eq:NN \ior_open_streams: \ior_list_streams:
6779 \cs_new_eq:NN \iow_open_streams: \iow_list_streams:
```

(End definition for \ior_open_streams:. This function is documented on page ??.)

```
6780 </initex | package>
```

183 l3msg implementation

```

6781 <*initex | package>
6782 <*package>
6783 \ProvidesExplPackage
6784   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
6785   \package_check_loaded_expl:
6786 </package>

```

\l_msg_tmp_tl A general scratch for the module.

```

6787 \tl_new:N \l_msg_tmp_tl

```

184 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.

```

6788 \tl_const:Nn \c_msg_text_prefix_tl { msg-text~>~ }
6789 \tl_const:Nn \c_msg_more_text_prefix_tl { msg-extra~text~>~ }

```

\msg_new:nnnn
\msg_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```

6790 \cs_new_protected:Npn \msg_new:nnnn #1#2
6791 {
6792   \cs_if_exist:cT { \c_msg_text_prefix_tl #1 / #2 }
6793   {
6794     \msg_kernel_error:nn { msg } { message-already-defined }
6795     {#1} {#2}
6796   }
6797   \msg_gset:nnnn {#1} {#2}
6798 }
6799 \cs_new_protected:Npn \msg_new:nnn #1#2#3
6800 { \msg_new:nnnn {#1} {#2} {#3} { } }
6801 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
6802 {
6803   \cs_set:cpn { \c_msg_text_prefix_tl #1 / #2 }
6804   ##1##2##3##4 {#3}
6805   \cs_set:cpn { \c_msg_more_text_prefix_tl #1 / #2 }
6806   ##1##2##3##4 {#4}
6807 }
6808 \cs_new_protected:Npn \msg_set:nnn #1#2#3
6809 { \msg_set:nnnn {#1} {#2} {#3} { } }

```

```

6810 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
6811 {
6812   \cs_gset:cpn { \c_msg_text_prefix_tl #1 / #2 }
6813     ##1##2##3##4 {#3}
6814   \cs_gset:cpn { \c_msg_more_text_prefix_tl #1 / #2 }
6815     ##1##2##3##4 {#4}
6816 }
6817 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
6818 { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for `\msg_new:nnnn` and `\msg_new:nnn`. These functions are documented on page 158.)

184.1 Messages: support functions and text

```

\c_msg_coding_error_text_tl Simple pieces of text for messages.
\c_msg_continue_text_tl
\c_msg_critical_text_tl
\c_msg_fatal_text_tl
\c_msg_help_text_tl
\c_msg_no_info_text_tl
\c_msg_on_line_tl
\c_msg_return_text_tl
\c_msg_trouble_text_tl
6819 \tl_const:Nn \c_msg_coding_error_text_tl
6820 {
6821   This-is-a-coding-error.
6822   \\ \\
6823 }
6824 \tl_const:Nn \c_msg_continue_text_tl
6825 { Type-<return>-to-continue }
6826 \tl_const:Nn \c_msg_critical_text_tl
6827 { Reading-the-current-file-will-stop }
6828 \tl_const:Nn \c_msg_fatal_text_tl
6829 { This-is-a-fatal-error:-LaTeX-will-abort }
6830 \tl_const:Nn \c_msg_help_text_tl
6831 { For-immediate-help-type-H-<return> }
6832 \tl_const:Nn \c_msg_no_info_text_tl
6833 {
6834   LaTeX-does-not-know-anything-more-about-this-error,-sorry.
6835   \c_msg_return_text_tl
6836 }
6837 \tl_const:Nn \c_msg_on_line_text_tl { on-line }
6838 \tl_const:Nn \c_msg_return_text_tl
6839 {
6840   \\ \\
6841   Try-typing-<return>-to-proceed.
6842   \\
6843   If-that-doesn't-work,-type-X-<return>-to-quit.
6844 }
6845 \tl_const:Nn \c_msg_trouble_text_tl
6846 {
6847   \\ \\
6848   More-errors-will-almost-certainly-follow: \\
6849   the-LaTeX-run-should-be-aborted.
6850 }

```

`\msg_newline:` New lines are printed in the same way as for low-level file writing.
`\msg_two_newlines:`

```
6851 \cs_new_nopar:Npn \msg_newline: { ^^J }
6852 \cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }
```

(End definition for \msg_newline: and \msg_two_newlines:. These functions are documented on page 163.)

`\msg_line_number:` For writing the line number nicely.
`\msg_line_context:`

```
6853 \cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
6854 \cs_set_nopar:Npn \msg_line_context:
6855 {
6856   \c_msg_on_line_text_tl
6857   \c_space_tl
6858   \msg_line_number:
6859 }
```

(End definition for \msg_line_number:. This function is documented on page 158.)

184.2 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.

```
6860 \char_set_catcode_letter:N \.
6861 \tl_new:N
6862   \c_msg_hide_tl.....
6863 \tl_const:Nn \c_msg_hide_tl
6864   { \c_msg_hide_tl..... }
6865 \char_set_catcode_other:N \.
```

`\msg_interrupt:xxx` 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.

```
6866 \cs_new_protected:Npn \msg_interrupt:xxx #1#2#3
6867 {
6868   \group_begin:
6869     \tl_if_empty:nTF {#3}
6870       { \msg_interrupt_no_details:xx {#1} {#2} }
6871       { \msg_interrupt_details:xxx {#1} {#2} {#3} }
6872   \msg_interrupt_aux:
6873   \group_end:
6874 }
```

```

6875 % Depending on the availability of more information there is a choice of
6876 % how to set up the further help. The extra help text has to be set
6877 % before the message itself can be issued. Everything is done using
6878 % \texttt{x}-type expansion as the new line markers are different for
6879 % the two type of text and need to be correctly set up.
6880 % \begin{macrocode}
6881 \cs_new_protected:Npn \msg_interrupt_no_details:xx #1#2
6882 {
6883   \iow_wrap:xnnnN
6884     { \c_msg_no_info_text_tl }
6885     { |~ } { 2 } { } \msg_interrupt_more_text:n
6886   \iow_wrap:xnnnN { #1 \c_msg_continue_text_tl }
6887     { ! ~ } { 2 } { } \msg_interrupt_text:n
6888 }
6889 \cs_new_protected:Npn \msg_interrupt_details:xxx #1#2#3
6890 {
6891   \iow_wrap:xnnnN
6892     { \c_msg_help_text_tl }
6893     { |~ } { 2 } { } \msg_interrupt_more_text:n
6894   \iow_wrap:xnnnN { #1 \c_msg_help_text_tl }
6895     { ! ~ } { 2 } { } \msg_interrupt_text:n
6896 }
6897 \cs_new_protected:Npn \msg_interrupt_text:n #1
6898 { \tl_set:Nn \l_msg_text_tl {#1} }
6899 \cs_new_protected:Npn \msg_interrupt_more_text:n #1
6900 {
6901   <*initex>
6902     \tl_set:Nx \l_msg_tmp_tl
6903   </initex>
6904   <*package>
6905     \protected@edef \l_msg_tmp_tl
6906   </package>
6907   {
6908     |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
6909     #1
6910     \msg_newline:
6911     |.....
6912   }
6913   \tex_errhelp:D \exp_after:wN { \l_msg_tmp_tl }
6914 }

```

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.

```

6915 \group_begin: % {
6916   \char_set_lccode:nn {'\} } {'\ }
6917   \char_set_lccode:nn {'\& } {'\!}

```

```

6918 \char_set_catcode_active:N \&
6919 \tl_to_lowercase:n
6920 {
6921 \group_end:
6922 \cs_new_protected:Npn \msg_interrupt_aux:
6923 {
6924 \iow_term:x
6925 {
6926 \iow_newline:
6927 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
6928 \iow_newline:
6929 !
6930 }
6931 \tl_put_right:No \l_msg_text_tl { \c_msg_hide_tl }
6932 \cs_set_protected_nopar:Npx &
6933 { \tex_errmessage:D { \exp_not:o { \l_msg_text_tl } } }
6934 &
6935 }
6936 }

```

(End definition for `\msg_interrupt:xxx`. This function is documented on page 163.)

`\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.

```

6937 \cs_new_protected:Npn \msg_log:x #1
6938 {
6939 \iow_log:x { ..... }
6940 \iow_wrap:xnnnN { . ~ #1 } { . ~ } { 2 } { }
6941 \iow_log:x
6942 \iow_log:x { ..... }
6943 }
6944 \cs_new_protected:Npn \msg_term:x #1
6945 {
6946 \iow_term:x { ***** }
6947 \iow_wrap:xnnnN { * ~ #1 } { * ~ } { 2 } { }
6948 \iow_term:x
6949 \iow_term:x { ***** }
6950 }

```

(End definition for `\msg_log:x`. This function is documented on page 164.)

184.3 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled.

```

6951 \int_set:Nn \tex_errorcontextlines:D { -1 }

```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principal vary.
`\msg_critical_text:n`
`\msg_error_text:n`
`\msg_warning_text:n`
`\msg_info_text:n`

```

6952 \cs_new_nopar:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
6953 \cs_new_nopar:Npn \msg_critical_text:n #1 { Critical~#1~error }
6954 \cs_new_nopar:Npn \msg_error_text:n #1 { #1~error }
6955 \cs_new_nopar:Npn \msg_warning_text:n #1 { #1~warning }
6956 \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 160.)

`\msg_see_documentation_text:n` Contextual footer information.

```

6957 \cs_new_nopar:Npn \msg_see_documentation_text:n #1
6958 { \\ \\ 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` For filtering messages, a list of all messages and of those which have to be modified is required.
`\l_msg_redirect_names_prop`

```

6959 \prop_new:N \l_msg_redirect_classes_prop
6960 \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.

```

6961 \cs_new_protected_nopar:Npn \msg_class_set:nn #1#2
6962 {
6963   \prop_clear_new:c { l_msg_redirect_ #1 _prop }
6964   \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
6965   { \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6} }
6966   \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
6967   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
6968   \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
6969   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
6970   \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3
6971   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
6972   \cs_set_protected:cpx { msg_ #1 :nn } ##1##2
6973   { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { } }
6974 }

```

(End definition for `\msg_class_set:nn`. This function is documented on page 160.)

`\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
6975 \prg_set_conditional:Npnn \msg_if_more_text:N #1 { p , T , F , TF }
6976 {
6977   \cs_if_eq:NNTF #1 \msg_no_more_text:xxxx
6978   { \prg_return_false: }
6979   { \prg_return_true: }
6980 }
6981 \cs_new:Npn \msg_no_more_text:xxxx #1#2#3#4 { }

```

```

6982 \cs_generate_variant:Nn \msg_if_more_text_p:N { c }
6983 \cs_generate_variant:Nn \msg_if_more_text:NT { c }
6984 \cs_generate_variant:Nn \msg_if_more_text:NF { c }
6985 \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.

```

\msg_fatal:nnxxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn
6986 \msg_class_set:nn { fatal }
6987 {
6988   \msg_interrupt:xxx
6989   { \msg_fatal_text:n {#1} : ~ "#2" }
6990   {
6991     \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
6992     \msg_see_documentation_text:n {#1}
6993   }
6994   { \c_msg_fatal_text_tl }
6995   \tex_end:D
6996 }

```

(End definition for `\msg_fatal:nnxxxx` and others. These functions are documented on page 161.)

`\msg_critical:nnxxxx` Not quite so bad: just end the current file.

```

\msg_critical:nnxxxx
\msg_critical:nnxxx
\msg_critical:nnxx
\msg_critical:nnx
\msg_critical:nn
6997 \msg_class_set:nn { critical }
6998 {
6999   \msg_interrupt:xxx
7000   { \msg_critical_text:n {#1} : ~ "#2" }
7001   {
7002     \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7003     \msg_see_documentation_text:n {#1}
7004   }
7005   { \c_msg_critical_text_tl }
7006   \tex_endinput:D
7007 }

```

(End definition for `\msg_critical:nnxxxx` and others. These functions are documented on page 161.)

`\msg_error:nnxxxx` For an error, the interrupt routine is called, then any recovery code is tried.

```

\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn
7008 \msg_class_set:nn { error }
7009 {
7010   \msg_if_more_text:cTF { \c_msg_more_text_prefix_tl #1 / #2 }
7011   {
7012     \msg_interrupt:xxx
7013     { \msg_error_text:n {#1} : ~ "#2" }
7014     {
7015       \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}

```

```

7016         \msg_see_documentation_text:n {#1}
7017     }
7018     { \use:c { \c_msg_more_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
7019 }
7020 {
7021     \msg_interrupt:xxx
7022     { \msg_error_text:n {#1} : ~ "#2" }
7023     {
7024         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7025         \msg_see_documentation_text:n {#1}
7026     }
7027     { }
7028 }
7029 }

```

(End definition for `\msg_error:nnxxxx` and others. These functions are documented on page 161.)

`\msg_warning:nnxxxx` Warnings are printed to the terminal.
`\msg_warning:nnxxx`
`\msg_warning:nnxx`
`\msg_warning:nnx`
`\msg_warning:nn`

```

7030 \msg_class_set:nn { warning }
7031 {
7032     \msg_term:x
7033     {
7034         \msg_warning_text:n {#1} : ~ "#2" \\ \\
7035         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7036     }
7037 }

```

(End definition for `\msg_warning:nnxxxx` and others. These functions are documented on page 161.)

`\msg_info:nnxxxx` Information only goes into the log.
`\msg_info:nnxxx`
`\msg_info:nnxx`
`\msg_info:nnx`
`\msg_info:nn`

```

7038 \msg_class_set:nn { info }
7039 {
7040     \msg_log:x
7041     {
7042         \msg_info_text:n {#1} : ~ "#2" \\ \\
7043         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7044     }
7045 }

```

(End definition for `\msg_info:nnxxxx` and others. These functions are documented on page 161.)

`\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`

```

7046 \msg_class_set:nn { log }
7047 {
7048     \msg_log:x
7049     { \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
7050 }

```

(End definition for `\msg_log:nnxxx` and others. These functions are documented on page 162.)

```
\msg_none:nnxxxx
\msg_none:nnxxx
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn
```

The none message type is needed so that input can be gobbled.

```
7051 \msg_class_set:nn { none } { }
```

(End definition for `\msg_none:nnxxx` and others. These functions are documented on page 162.)

```
\l_msg_redirect_classes_seq
\l_msg_class_tl
\l_msg_current_class_tl
\l_msg_current_module_tl
```

Support variables needed for the redirection system.

```
7052 \seq_new:N \l_msg_redirect_classes_seq
7053 \tl_new:N \l_msg_class_tl
7054 \tl_new:N \l_msg_current_class_tl
7055 \tl_new:N \l_msg_current_module_tl
```

```
\msg_use:nnnnxxxx
\msg_use_aux:nnn
\msg_use_aux:nn
\msg_use_loop_check:nn
\msg_use_code:
\msg_use_loop:n
\msg_use_loop:o
```

The main message-using macro creates two auxiliary functions: one containing the code for the message, and the second a loop function. There is then a hand-off to the system for checking if redirection is needed.

```
7056 \cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8
7057 {
7058   \cs_set_protected_nopar:Npx \msg_use_code:
7059   {
7060     \seq_clear:N \exp_not:N \l_msg_redirect_classes_seq
7061     \exp_not:n {#2}
7062   }
7063   \cs_set_protected:Npx \msg_use_loop:n ##1
7064   {
7065     \seq_if_in:NnTF \exp_not:n \l_msg_redirect_classes_seq {#1}
7066     { \msg_kernel_error:nn { msg } { message-loop } {#1} }
7067     {
7068       \seq_put_right:Nn \exp_not:N \l_msg_redirect_classes_seq {#1}
7069       \exp_not:N \cs_if_exist:cTF { msg_ ##1 :nnxxxx }
7070       {
7071         \exp_not:N \use:c { msg_ ##1 :nnxxxx }
7072         \exp_not:n { {#3} {#4} {#5} {#6} {#7} {#8} }
7073       }
7074       {
7075         \msg_kernel_error:nnx { msg } { message-class-unknown } {##1}
7076       }
7077     }
7078   }
7079   \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 }
7080   { \msg_use_aux:nnn {#1} {#3} {#4} }
7081   { \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4} }
7082 }
```

The first auxiliary macro looks for a match by name: the most restrictive check.

```
7083 \cs_new_protected_nopar:Npn \msg_use_aux:nnn #1#2#3
```

```

7084 {
7085     \tl_set:Nn \l_msg_current_class_tl {#1}
7086     \tl_set:Nn \l_msg_current_module_tl {#2}
7087     \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / }
7088     { \msg_use_loop_check:nn { names } { // #2 / #3 / } }
7089     { \msg_use_aux:nn {#1} {#2} }
7090 }

```

The second function checks for general matches by module or for all modules.

```

7091 \cs_new_protected_nopar:Npn \msg_use_aux:nn #1#2
7092 {
7093     \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2}
7094     { \msg_use_loop_check:nn {#1} {#2} }
7095     {
7096         \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } { * }
7097         { \msg_use_loop_check:nn {#1} { * } }
7098         { \msg_use_code: }
7099     }
7100 }

```

When checking whether to loop, the same code is needed in a few places.

```

7101 \cs_new_protected:Npn \msg_use_loop_check:nn #1#2
7102 {
7103     \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
7104     \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl
7105     {
7106         { \msg_use_code: }
7107         { \msg_use_loop:o \l_msg_class_tl }
7108     }
7109 }
7110 \cs_new_protected_nopar:Npn \msg_use_code: { }
7111 \cs_new_protected:Npn \msg_use_loop:n #1 { }
7112 \cs_generate_variant:Nn \msg_use_loop:n { o }

```

(End definition for `\msg_use:nnnnxxx`. This function is documented on page ??.)

`\msg_redirect_class:nn` Converts class one into class two.

```

7113 \cs_new_protected_nopar:Npn \msg_redirect_class:nn #1#2
7114 { \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#2} }

```

(End definition for `\msg_redirect_class:nn`. This function is documented on page 162.)

`\msg_redirect_module:nnn` For when all messages of a class should be altered for a given module.

```

7115 \cs_new_protected_nopar:Npn \msg_redirect_module:nnn #1#2#3
7116 { \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3} }

```

(End definition for `\msg_redirect_module:nnn`. This function is documented on page 162.)

`\msg_redirect_name:nnn` Named message will always use the given class.

```
7117 \cs_new_protected_nopar:Npn \msg_redirect_name:nnn #1#2#3
7118 { \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3} }
```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 163.)

184.4 Kernel-specific functions

`\msg_kernel_new:nnnn` The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

`\msg_kernel_set:nnnn`
`\msg_kernel_set:nnn`

```
7119 \cs_new_protected_nopar:Npn \msg_kernel_new:nnnn #1#2
7120 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
7121 \cs_new_protected_nopar:Npn \msg_kernel_new:nnn #1#2
7122 { \msg_new:nnn { LaTeX } { #1 / #2 } }
7123 \cs_new_protected_nopar:Npn \msg_kernel_set:nnnn #1#2
7124 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
7125 \cs_new_protected_nopar:Npn \msg_kernel_set:nnn #1#2
7126 { \msg_set:nnn { LaTeX } { #1 / #2 } }
```

(End definition for `\msg_kernel_new:nnnn`. This function is documented on page 164.)

`\msg_kernel_fatal:nnxxxx` Fatal kernel errors cannot be re-defined.

`\msg_kernel_fatal:nnxxx`
`\msg_kernel_fatal:nnxx`
`\msg_kernel_fatal:nnx`
`\msg_kernel_fatal:nn`

```
7127 \cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6
7128 {
7129   \msg_interrupt:xxx
7130   { \msg_fatal_text:n { LaTeX } : ~ "#1 / #2" }
7131   {
7132     \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
7133     {#3} {#4} {#5} {#6}
7134     \msg_see_documentation_text:n { LaTeX3 }
7135   }
7136   { \c_msg_fatal_text_tl }
7137   \tex_end:D
7138 }
7139 \cs_new_protected:Npn \msg_kernel_fatal:nnxxx #1#2#3#4#5
7140 { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
7141 \cs_new_protected:Npn \msg_kernel_fatal:nnxx #1#2#3#4
7142 { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} {#4} { } { } }
7143 \cs_new_protected:Npn \msg_kernel_fatal:nnx #1#2#3
7144 { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} { } { } { } }
7145 \cs_new_protected:Npn \msg_kernel_fatal:nn #1#2
7146 { \msg_kernel_fatal:nnxxxx {#1} {#2} { } { } { } { } }
```

(End definition for `\msg_kernel_fatal:nnxxxx`. This function is documented on page 165.)

`\msg_kernel_error:nnxxxx`
`\msg_kernel_error:nnxxx`
`\msg_kernel_error:nnxx`
`\msg_kernel_error:nnx`
`\msg_kernel_error:nn`

Neither can kernel errors.

```

7147 \cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6
7148 {
7149   \msg_if_more_text:cTF { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 }
7150   {
7151     \msg_interrupt:xxx
7152     { \msg_error_text:n { LaTeX } : ~ " #1 / #2 " }
7153     {
7154       \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
7155       {#3} {#4} {#5} {#6}
7156       \msg_see_documentation_text:n { LaTeX3 }
7157     }
7158     {
7159       \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 }
7160       {#3} {#4} {#5} {#6}
7161     }
7162   }
7163   {
7164     \msg_interrupt:xxx
7165     { \msg_error_text:n { LaTeX } : ~ " #1 / #2 " }
7166     {
7167       \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
7168       {#3} {#4} {#5} {#6}
7169       \msg_see_documentation_text:n { LaTeX3 }
7170     }
7171   }
7172 }
7173 }
7174 \cs_new_protected:Npn \msg_kernel_error:nnxxx #1#2#3#4#5
7175 { \msg_kernel_error:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
7176 \cs_set_protected:Npn \msg_kernel_error:nnxx #1#2#3#4
7177 { \msg_kernel_error:nnxxxx {#1} {#2} {#3} {#4} { } { } }
7178 \cs_set_protected:Npn \msg_kernel_error:nnx #1#2#3
7179 { \msg_kernel_error:nnxxxx {#1} {#2} {#3} { } { } { } }
7180 \cs_set_protected:Npn \msg_kernel_error:nn #1#2
7181 { \msg_kernel_error:nnxxxx {#1} {#2} { } { } { } { } }

```

(End definition for `\msg_kernel_error:nnxxxx`. This function is documented on page 165.)

`\msg_kernel_warning:nnxxxx`
`\msg_kernel_warning:nnxxx`
`\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`

Kernel messages which can be redirected.

```

7182 \prop_new:N \l_msg_redirect_kernel_warning_prop
7183 \cs_new_protected:Npn \msg_kernel_warning:nnxxxx #1#2#3#4#5#6
7184 {
7185   \msg_use:nnnnxxxx { warning }
7186   {
7187     \msg_term:x
7188     {
7189       \msg_warning_text:n { LaTeX } : ~ " #1 / #2 " \\ \\

```

```

7190         \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
7191         {#3} {#4} {#5} {#6}
7192     }
7193 }
7194 { LaTeX } { #1 / #2 } {#3} {#4} {#5} {#6}
7195 }
7196 \cs_new_protected:Npn \msg_kernel_warning:nnxxx #1#2#3#4#5
7197 { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
7198 \cs_new_protected:Npn \msg_kernel_warning:nnxx #1#2#3#4
7199 { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} {#4} { } { } }
7200 \cs_new_protected:Npn \msg_kernel_warning:nnx #1#2#3
7201 { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} { } { } { } }
7202 \cs_new_protected:Npn \msg_kernel_warning:nn #1#2
7203 { \msg_kernel_warning:nnxxxx {#1} {#2} { } { } { } { } }
7204 \prop_new:N \l_msg_redirect_kernel_info_prop
7205 \cs_new_protected:Npn \msg_kernel_info:nnxxxx #1#2#3#4#5#6
7206 {
7207     \msg_use:nnnnxxxx { info }
7208     {
7209         \msg_log:x
7210         {
7211             \msg_info_text:n { LaTeX } : ~ " #1 / #2 " \\ \\
7212             \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
7213             {#3} {#4} {#5} {#6}
7214         }
7215     }
7216     { LaTeX } { #1 / #2 } {#3} {#4} {#5} {#6}
7217 }
7218 \cs_new_protected:Npn \msg_kernel_info:nnxxx #1#2#3#4#5
7219 { \msg_kernel_info:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
7220 \cs_new_protected:Npn \msg_kernel_info:nnxx #1#2#3#4
7221 { \msg_kernel_info:nnxxxx {#1} {#2} {#3} {#4} { } { } }
7222 \cs_new_protected:Npn \msg_kernel_info:nnx #1#2#3
7223 { \msg_kernel_info:nnxxxx {#1} {#2} {#3} { } { } { } }
7224 \cs_new_protected:Npn \msg_kernel_info:nn #1#2
7225 { \msg_kernel_info:nnxxxx {#1} {#2} { } { } { } { } }

```

(End definition for `\msg_kernel_warning:nnxxxx`. This function is documented on page 165.)

Error messages needed to actually implement the message system itself.

```

7226 \msg_kernel_new:nnnn { msg } { message-already-defined }
7227 { Message~'#2'~for~module~'#1'~already-defined. }
7228 {
7229     \c_msg_coding_error_text_tl
7230     LaTeX~was~asked~to~define~a~new~message~called~'#2'
7231     by~the~module~'#1'~module:\\
7232     this~message~already~exists.
7233     \c_msg_return_text_tl
7234 }
7235 \msg_kernel_new:nnnn { msg } { message-unknown }

```

```

7236 { Unknown~message~'#2'~for~module~'#1'. }
7237 {
7238   \c_msg_coding_error_text_tl
7239   LaTeX~was~asked~to~display~a~message~called~'#2'\\
7240   by~the~module~'#1'~module:~this~message~does~not~exist.
7241   \c_msg_return_text_tl
7242 }
7243 \msg_kernel_new:nnnn { msg } { message-class-unknown }
7244 { Unknown~message~class~'#1'. }
7245 {
7246   LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
7247   this~was~never~defined.
7248   \c_msg_return_text_tl
7249 }
7250 \msg_kernel_new:nnnn { msg } { redirect-loop }
7251 { Message~redirection~loop~for~message~class~'#1'. }
7252 {
7253   LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.\\
7254   The~original~message~here~has~been~lost.
7255   \c_msg_return_text_tl
7256 }

```

Messages for earlier kernel modules.

```

7257 \msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
7258 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
7259 {
7260   \c_msg_coding_error_text_tl
7261   LaTeX~has~been~asked~to~define~a~function~'#1'~with~
7262   #2~arguments. \\
7263   TeX~allows~between~0~and~9~arguments~for~a~single~function.
7264 }
7265 \msg_kernel_new:nnnn { kernel } { command-already-defined }
7266 { Control~sequence~#1~already~defined. }
7267 {
7268   \c_msg_coding_error_text_tl
7269   LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
7270   but~this~name~has~already~been~used~elsewhere. \\ \\
7271   The~current~meaning~is:\\
7272   \ \ #2
7273 }
7274 \msg_kernel_new:nnnn { kernel } { command-not-defined }
7275 { Control~sequence~#1~undefined. }
7276 {
7277   \c_msg_coding_error_text_tl
7278   LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
7279   been~defined~yet.
7280 }
7281 \msg_kernel_new:nnnn { kernel } { variable-not-defined }
7282 { Variable~#1~undefined. }

```

```

7283 {
7284   \c_msg_coding_error_text_tl
7285   LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
7286   been~defined~yet.
7287 }
7288 \msg_kernel_new:nnnn { seq } { empty-sequence }
7289 { Empty~sequence~#1. }
7290 {
7291   \c_msg_coding_error_text_tl
7292   LaTeX~has~been~asked~to~recover~an~entry~from~a~sequence~that~
7293   has~no~content:~that~cannot~happen!
7294 }
7295 \msg_kernel_new:nnnn { tl } { empty-search-pattern }
7296 { Empty~search~pattern. }
7297 {
7298   \c_msg_coding_error_text_tl
7299   LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'~#1':~that~%
7300   would~lead~to~an~infinite~loop!
7301 }

```

```

\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.

```

7302 \cs_set_protected:Npn \msg_kernel_bug:x #1
7303 {
7304   \msg_interrupt:xxx { \c_msg_kernel_bug_text_tl }
7305   {
7306     #1
7307     \msg_see_documentation_text:n { LaTeX3 }
7308   }
7309   { \c_msg_kernel_bug_more_text_tl }
7310 }
7311 \tl_const:Nn \c_msg_kernel_bug_text_tl
7312 { This~is~a~LaTeX~bug:~check~coding! }
7313 \tl_const:Nn \c_msg_kernel_bug_more_text_tl
7314 {
7315   There~is~a~coding~bug~somewhere~around~here. \\
7316   This~probably~needs~examining~by~an~expert.
7317   \c_msg_return_text_tl
7318 }

```

(End definition for \msg_kernel_bug:x. This function is documented on page ??.)

184.5 Expandable errors

\msg_expandable_error:n

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed T_EX an undefined control sequence, `\LaTeX3 error:.` It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```
<argument> \LaTeX3 error:
          The error message.
```

In other words, \TeX is processing the argument of \use:n , which is \LaTeX3 error: *<error message>*. Then $\text{\msg_expandable_error_aux:w}$ cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until \q_stop . The \c_zero prevents losing braces around the user-inserted text if any, and stops the expansion of $\text{\tex_romannumeral:D}$.

```
7319 \group_begin:
7320 \char_set_catcode_math_superscript:N \^
7321 \char_set_lccode:nn {'~} {'\ }
7322 \char_set_lccode:nn {'L} {'L}
7323 \char_set_lccode:nn {'T} {'T}
7324 \char_set_lccode:nn {'X} {'X}
7325 \tl_to_lowercase:n
7326 {
7327   \cs_new:Npx \msg_expandable_error:n #1
7328   {
7329     \exp_not:n
7330     {
7331       \tex_romannumeral:D
7332       \exp_after:wN \exp_after:wN
7333       \exp_after:wN \msg_expandable_error_aux:w
7334       \exp_after:wN \exp_after:wN
7335       \exp_after:wN \c_zero
7336     }
7337     \exp_not:N \use:n { \exp_not:c { LaTeX3-error: } ^ #1 }
7338     \exp_not:N \q_stop
7339   }
7340   \cs_new:Npn \msg_expandable_error_aux:w #1 ^ #2 \q_stop { #1 }
7341 }
7342 \group_end:
```

(End definition for $\text{\msg_expandable_error:n}$. This function is documented on page 166.)

184.6 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.

```
7343 \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:nn`

```
7344 \cs_new_eq:NN \msg_trace:nnxxxx \msg_log:nnxxxx
7345 \cs_new_eq:NN \msg_trace:nnxxx \msg_log:nnxxx
7346 \cs_new_eq:NN \msg_trace:nnxx \msg_log:nnxx
7347 \cs_new_eq:NN \msg_trace:nnx \msg_log:nnx
7348 \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`

```
7349 \cs_new_protected:Npn \msg_generic_new:nnn #1#2#3 { \deprecated }
7350 \cs_new_protected:Npn \msg_generic_new:nn #1#2 { \deprecated }
7351 \cs_new_protected:Npn \msg_generic_set:nnn #1#2#3 { \deprecated }
7352 \cs_new_protected:Npn \msg_generic_set:nn #1#2 { \deprecated }
7353 \cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 { \deprecated }
7354 \cs_new_protected:Npn \msg_direct_log:xx #1#2 { \deprecated }
7355 \cs_new_protected:Npn \msg_direct_term:xx #1#2 { \deprecated }
```

(End definition for `\msg_generic_new:nnn`. This function is documented on page ??.)

```
7356 </initex | package>
```

185 l3keys Implementation

```
7357 <*initex | package>
7358 <*package>
7359 \ProvidesExplPackage
7360 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
7361 \package_check_loaded_expl:
7362 </package>
```

185.1 Low-level interface

For historical reasons this code uses the ‘keyval’ module prefix.

`\g_keyval_level_int` For nesting purposes an integer is needed for the current level.

```
7363 \int_new:N \g_keyval_level_int
```

`\l_keyval_key_tl` The current key name and value.
`\l_keyval_value_tl`

```
7364 \tl_new:N \l_keyval_key_tl
7365 \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`

```
7366 \tl_new:N \l_keyval_sanitise_tl
7367 \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.

```
7368 \group_begin:
7369   \char_set_catcode_active:n { '\= }
7370   \char_set_catcode_active:n { '\, }
7371   \char_set_lccode:nn { '\8 } { '\= }
7372   \char_set_lccode:nn { '\9 } { '\, }
7373   \tl_to_lowercase:n
7374   {
7375     \group_end:
7376     \cs_new_protected:Npn \keyval_parse:n #1
7377     {
7378       \group_begin:
7379       \tl_clear:N \l_keyval_sanitise_tl
7380       \tl_set:Nn \l_keyval_sanitise_tl {#1}
7381       \tl_replace_all:Nnn \l_keyval_sanitise_tl { = } { 8 }
7382       \tl_replace_all:Nnn \l_keyval_sanitise_tl { , } { 9 }
7383       \tl_clear:N \l_keyval_parse_tl
7384       \exp_after:wN \keyval_parse_elt:w \exp_after:wN
7385       \q_no_value \l_keyval_sanitise_tl 9 \q_nil 9
7386       \exp_after:wN \group_end:
7387       \l_keyval_parse_tl
7388     }
7389   }
```

(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.

```
7390 \cs_new_protected:Npn \keyval_parse_elt:w #1 ,
7391 {
7392   \tl_if_blank:oTF { \use_none:n #1 }
7393   { \keyval_parse_elt:w \q_no_value }
7394   {
7395     \quark_if_nil:oF { \use_ii:nn #1 }
7396     {
7397       \keyval_split_key_value:w #1 = = \q_stop
7398       \keyval_parse_elt:w \q_no_value
7399     }
7400   }
7401 }
```

(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.

`\keyval_split_key_value_aux:wTF`

```

7402 \cs_new_protected:Npn \keyval_split_key_value:w #1 = #2 \q_stop
7403 {
7404   \keyval_split_key:w #1 \q_stop
7405   \str_if_eq:nnTF {#2} { = }
7406   {
7407     \tl_put_right:Nx \l_keyval_parse_tl
7408     {
7409       \exp_not:c
7410       { keyval_key_no_value_elt_ \int_use:N \g_keyval_level_int :n }
7411       { \exp_not:o \l_keyval_key_tl }
7412     }
7413   }
7414   {
7415     \keyval_split_key_value_aux:wTF #2 \q_no_value \q_stop
7416     { \keyval_split_value:w \q_nil #2 }
7417     { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
7418   }
7419 }
7420 \cs_new:Npn \keyval_split_key_value_aux:wTF #1 = #2#3 \q_stop
7421 { \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` 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.
`\keyval_split_key_aux:w`
`\keyval_remove_spaces_aux:w`

```

7422 \group_begin:
7423   \char_set_catcode_math_toggle:n { '\Q }
7424   \cs_new_protected:Npn \keyval_split_key:w #1 \q_stop
7425   {
7426     \exp_args:NNf \tl_set:Nn \l_keyval_key_tl
7427     {
7428       \exp_after:wN \keyval_remove_spaces:w \exp_after:wN
7429       \exp_not:N \use_none:n #1 Q ~ Q
7430     }

```

```

7431 \tl_set:Nx \l_keyval_key_tl
7432 { \exp_after:wN \keyval_split_key_aux:w \l_keyval_key_tl \q_stop }
7433 }
7434 \cs_gset:Npn \keyval_split_key_aux:w #1 \q_stop { \exp_not:n {#1} }
7435 \cs_gset:Npn \keyval_remove_spaces:w #1 ~ Q
7436 { \keyval_remove_spaces_aux:w #1 Q }
7437 \cs_gset:Npn \keyval_remove_spaces_aux:w #1 Q #2 {#1}
7438 \group_end:

```

Fixme: use `\tl_trim_spaces` instead of the aux function above?

(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.

```

7439 \cs_new_protected:Npn \keyval_split_value:w #1 = =
7440 {
7441   \tl_put_right:Nx \l_keyval_parse_tl
7442   {
7443     \exp_not:c
7444     { keyval_key_value_elt_ \int_use:N \g_keyval_level_int :nn }
7445     { \exp_not:o \l_keyval_key_tl }
7446   }
7447   \tl_set:Nx \l_keyval_value_tl
7448   { \exp_not:o { \use_none:nnn #1 \q_nil \q_nil } }
7449   \tl_if_empty:NTF \l_keyval_value_tl
7450   { \tl_put_right:Nn \l_keyval_parse_tl { { } } }
7451   {
7452     \quark_if_nil:NTF \l_keyval_value_tl
7453     {
7454       \tl_put_right:Nx \l_keyval_parse_tl
7455       { { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
7456     }
7457     { \keyval_split_value_aux:w #1 \q_stop }
7458   }
7459 }

```

A similar idea to the key code: remove the spaces from each end and deal with one set of braces.

```

7460 \group_begin:
7461 \char_set_catcode_math_toggle:n { '\Q }
7462 \cs_new_protected:Npn \keyval_split_value_aux:w \q_nil #1 \q_stop
7463 {

```

```

7464 \exp_args:NNf \tl_set:Nn \l_keyval_value_tl
7465 { \keyval_remove_spaces:w \exp_not:N #1 Q ~ Q }
7466 \tl_put_right:Nx \l_keyval_parse_tl
7467 { { \exp_not:o \l_keyval_value_tl } }
7468 }
7469 \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.

```

7470 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
7471 {
7472   \int_gincr:N \g_keyval_level_int
7473   \cs_gset_eq:cN
7474     { keyval_key_no_value_elt_ \int_use:N \g_keyval_level_int :n } #1
7475   \cs_gset_eq:cN
7476     { keyval_key_value_elt_ \int_use:N \g_keyval_level_int :nn } #2
7477   \keyval_parse:n {#3}
7478   \int_gdecr:N \g_keyval_level_int
7479 }

```

(End definition for `\keyval_parse:NNn`. This function is documented on page 178.)

One message for the low level parsing system.

```

7480 \msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
7481 { Misplaced~equals~sign~in~key~value~input~\msg_line_number: }
7482 {
7483   LaTeX-is-attempting-to-parse-some-key-value-input-but-found-
7484   two-equals-signs-not-separated-by-a-comma.
7485 }

```

185.2 Constants and variables

`\c_keys_code_root_tl` The prefixes for the code and variables of the keys themselves.

`\c_keys_vars_root_tl`

```

7486 \tl_const:Nn \c_keys_code_root_tl { key~code~>~ }
7487 \tl_const:Nn \c_keys_vars_root_tl { key~var~>~ }

```

`\c_keys_props_root_tl` The prefix for storing properties.

```

7488 \tl_const:Nn \c_keys_props_root_tl { key~prop~>~ }

```

`\c_keys_value_forbidden_tl` Two marker token lists.

`\c_keys_value_required_tl`

```

7489 \tl_const:Nn \c_keys_value_forbidden_tl { forbidden }
7490 \tl_const:Nn \c_keys_value_required_tl { required }

```

<code>\l_keys_choice_int</code> <code>\l_keys_choices_tl</code>	Publicly accessible data on which choice is being used when several are generated as a set.
	<pre> 7491 \int_new:N \l_keys_choice_int 7492 \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> 7493 \tl_new:N \l_keys_key_tl </pre>
<code>\l_keys_module_tl</code>	The module for an entire set of keys.
	<pre> 7494 \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> 7495 \bool_new:N \l_keys_no_value_bool </pre>
<code>\l_keys_path_tl</code>	The “path” of the current key is stored here: this is available to the programmer and so is public.
	<pre> 7496 \tl_new:N \l_keys_path_tl </pre>
<code>\l_keys_property_tl</code>	The “property” begin set for a key at definition time is stored here.
	<pre> 7497 \tl_new:N \l_keys_property_tl </pre>
<code>\l_keys_unknown_clist</code>	Used when setting only known keys to store those left over.
	<pre> 7498 \tl_new:N \l_keys_unknown_clist </pre>
<code>\l_keys_value_tl</code>	The value given for a key: may be empty if no value was given.
	<pre> 7499 \tl_new:N \l_keys_value_tl </pre>

185.3 The key defining mechanism

<code>\keys_define:nn</code> <code>\keys_define_aux:nnn</code> <code>\keys_define_aux:onn</code>	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.
--	---

```

7500 \cs_new_protected:Npn \keys_define:nn
7501 { \keys_define_aux:onn \l_keys_module_tl }
7502 \cs_new_protected:Npn \keys_define_aux:nnn #1#2#3
7503 {
7504   \tl_set:Nx \l_keys_module_tl { \tl_to_str:n {#2} }
7505   \keyval_parse:NNn \keys_define_elt:n \keys_define_elt:nn {#3}
7506   \tl_set:Nn \l_keys_module_tl {#1}
7507 }
7508 \cs_generate_variant:Nn \keys_define_aux:nnn { o }

```

(End definition for `\keys_define:nn`. This function is documented on page 168.)

`\keys_define_elt:n` The outer functions here record whether a value was given and then converge on a
`\keys_define_elt:nn` common internal mechanism. There is first a search for a property in the current key
`\keys_define_elt_aux:nn` name, then a check to make sure it is known before the code hands off to the next step.

```

7509 \cs_new_protected_nopar:Npn \keys_define_elt:n #1
7510 {
7511   \bool_set_true:N \l_keys_no_value_bool
7512   \keys_define_elt_aux:nn {#1} { }
7513 }
7514 \cs_new_protected:Npn \keys_define_elt:nn #1#2
7515 {
7516   \bool_set_false:N \l_keys_no_value_bool
7517   \keys_define_elt_aux:nn {#1} {#2}
7518 }
7519 \cs_new_protected:Npn \keys_define_elt_aux:nn #1#2 {
7520   \keys_property_find:n {#1}
7521   \cs_if_exist:cTF { \c_keys_props_root_tl \l_keys_property_tl }
7522   { \keys_define_key:n {#2} }
7523   {
7524     \msg_kernel_error:nnxx { keys } { property-unknown }
7525     { \l_keys_property_tl } { \l_keys_path_tl }
7526   }
7527 }

```

(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
`\keys_property_find_aux:w` and after it. Everything is turned into strings, so there is no problem using an x-type
expansion.

```

7528 \cs_new_protected_nopar:Npn \keys_property_find:n #1
7529 {
7530   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
7531   \tl_if_in:nnTF {#1} { . }
7532   { \keys_property_find_aux:w #1 \q_stop }
7533   { \msg_kernel_error:nnx { keys } { key-no-property } {#1} }
7534 }
7535 \cs_new_protected_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop
7536 {
7537   \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
7538   \tl_if_in:nnTF {#2} { . }
7539   {
7540     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
7541     \keys_property_find_aux:w #2 \q_stop
7542   }
7543   { \tl_set:Nn \l_keys_property_tl { . #2 } }
7544 }

```

(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, `\keys_define_key_aux:w` 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.

```

7545 \cs_new_protected:Npn \keys_define_key:n #1
7546 {
7547   \bool_if:NTF \l_keys_no_value_bool
7548   {
7549     \exp_after:wN \keys_define_key_aux:w
7550     \l_keys_property_tl \q_stop
7551     { \use:c { \c_keys_props_root_tl \l_keys_property_tl } }
7552     {
7553       \msg_kernel_error:nxxx { keys }
7554       { property-requires-value } { \l_keys_property_tl }
7555       { \l_keys_path_tl }
7556     }
7557   }
7558   { \use:c { \c_keys_props_root_tl \l_keys_property_tl } {#1} }
7559 }
7560 \cs_new_protected:Npn \keys_define_key_aux:w #1 : #2 \q_stop
7561 { \tl_if_empty:nTF {#2} }

```

(End definition for `\keys_define_key:n`. This function is documented on page ??.)

185.4 Turning properties into actions

`\keys_bool_set:NN` Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

7562 \cs_new_nopar:Npn \keys_bool_set:NN #1#2
7563 {
7564   \cs_if_exist:NF #1 { \bool_new:N #1 }
7565   \keys_choice_make:
7566   \keys_cmd_set:nx { \l_keys_path_tl / true }
7567   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
7568   \keys_cmd_set:nx { \l_keys_path_tl / false }
7569   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
7570   \keys_cmd_set:nn { \l_keys_path_tl / unknown }
7571   {
7572     \msg_kernel_error:nxx { keys } { boolean-values-only }
7573     { \l_keys_key_tl }
7574   }
7575   \keys_default_set:n { true }
7576 }

```

(End definition for `\keys_bool_set:NN`. This function is documented on page ??.)

`\keys_bool_set_inverse:NN` Inverse boolean setting is much the same.

```

7577 \cs_new_nopar:Npn \keys_bool_set_inverse:NN #1#2
7578 {
7579   \cs_if_exist:NF #1 { \bool_new:N #1 }
7580   \keys_choice_make:
7581   \keys_cmd_set:nx { \l_keys_path_tl / true }
7582   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
7583   \keys_cmd_set:nx { \l_keys_path_tl / false }
7584   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
7585   \keys_cmd_set:nn { \l_keys_path_tl / unknown }
7586   {
7587     \msg_kernel_error:nnx { keys } { boolean-values-only }
7588     { \l_keys_key_tl }
7589   }
7590   \keys_default_set:n { true }
7591 }

```

(End definition for \keys_bool_set_inverse: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.

```

7592 \cs_new_protected_nopar:Npn \keys_choice_make:
7593 {
7594   \keys_cmd_set:nn { \l_keys_path_tl }
7595   { \keys_choice_find:n {##1} }
7596   \keys_cmd_set:nn { \l_keys_path_tl / unknown }
7597   {
7598     \msg_kernel_error:nnxx { keys } { choice-unknown }
7599     { \l_keys_path_tl } {##1}
7600   }
7601 }

```

(End definition for \keys_choice_make:. This function is documented on page ??.)

`\keys_choices_make:nn` Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

7602 \cs_new_protected:Npn \keys_choices_make:nn #1#2
7603 {
7604   \keys_choice_make:
7605   \int_zero:N \l_keys_choice_int
7606   \clist_map_inline:nn {#1}
7607   {
7608     \keys_cmd_set:nx { \l_keys_path_tl / ##1 }
7609     {
7610       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
7611       \int_set:Nn \exp_not:N \l_keys_choice_int
7612       { \int_use:N \l_keys_choice_int }
7613       \exp_not:n {#2}

```

```

7614     }
7615     \int_incr:N \l_keys_choice_int
7616   }
7617 }

```

(End definition for \keys_choices_make:nn. 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.

```

7618 \cs_new_protected:Npn \keys_choices_generate:n #1
7619 {
7620   \cs_if_exist:cTF
7621   { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
7622   {
7623     \keys_choice_make:
7624     \int_zero:N \l_keys_choice_int
7625     \clist_map_function:nN {#1} \keys_choices_generate_aux:n
7626   }
7627   {
7628     \msg_kernel_error:nxx { keys }
7629     { generate-choices-before-code } { \l_keys_path_tl }
7630   }
7631 }
7632 \cs_new_protected_nopar:Npn \keys_choices_generate_aux:n #1
7633 {
7634   \keys_cmd_set:nx { \l_keys_path_tl / #1 }
7635   {
7636     \tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
7637     \int_set:Nn \exp_not:N \l_keys_choice_int
7638     { \int_use:N \l_keys_choice_int }
7639     \exp_not:v
7640     { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
7641   }
7642   \int_incr:N \l_keys_choice_int
7643 }

```

(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.

```

7644 \cs_new_protected:Npn \keys_choice_code_store:x #1
7645 {
7646   \cs_if_exist:cF
7647   { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
7648   {
7649     \tl_new:c
7650     { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
7651   }
7652   \tl_set:cx { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }

```

```

7653     {#1}
7654 }

```

(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
`\keys_cmd_set:nx` function which actually does the work.
`\keys_cmd_set_aux:n`

```

7655 \cs_new_protected:Npn \keys_cmd_set:nn #1#2
7656 {
7657   \keys_cmd_set_aux:n {#1}
7658   \cs_set:cpn { \c_keys_code_root_tl #1 } ##1 {#2}
7659 }
7660 \cs_new_protected:Npn \keys_cmd_set:nx #1#2
7661 {
7662   \keys_cmd_set_aux:n {#1}
7663   \cs_set:cpx { \c_keys_code_root_tl #1 } ##1 {#2}
7664 }
7665 \cs_new_protected_nopar:Npn \keys_cmd_set_aux:n #1
7666 {
7667   \tl_clear_new:c { \c_keys_vars_root_tl #1 .default }
7668   \tl_set:cn { \c_keys_vars_root_tl #1 .default } { \q_no_value }
7669   \tl_clear_new:c { \c_keys_vars_root_tl #1 .req }
7670 }

```

(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`

```

7671 \cs_new_protected:Npn \keys_default_set:n #1
7672 { \tl_set:cn { \c_keys_vars_root_tl \l_keys_path_tl .default } {#1} }
7673 \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`

```

7674 \cs_new_protected_nopar:Npn \keys_meta_make:n #1
7675 {
7676   \exp_args:NNo \keys_cmd_set:nn \l_keys_path_tl
7677   { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }
7678 }
7679 \cs_new_protected_nopar:Npn \keys_meta_make:x #1
7680 {
7681   \keys_cmd_set:nx { \l_keys_path_tl }
7682   { \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1} }
7683 }

```

(End definition for `\keys_meta_make:n` and `\keys_meta_make:x`. These functions are documented on page ??.)

`\keys_multichoice_find:n` Choices where several values can be selected are very similar to normal exclusive choices.
`\keys_multichoice_make:` There is just a slight change in implementation to map across a comma-separated list.
`\keys_multichoices_make:nn` This then requires that the appropriate set up takes place elsewhere.

```

7684 \cs_new_nopar:Npn \keys_multichoice_find:n #1
7685 { \clist_map_function:nN {#1} \keys_choice_find:n }
7686 \cs_new_protected_nopar:Npn \keys_multichoice_make:
7687 {
7688   \keys_cmd_set:nn { \l_keys_path_tl }
7689   { \keys_multichoice_find:n {##1} }
7690   \keys_cmd_set:nn { \l_keys_path_tl / unknown }
7691   {
7692     \msg_kernel_error:nnxx { keys } { choice-unknown }
7693     { \l_keys_path_tl } {##1}
7694   }
7695 }
7696 \cs_new_protected:Npn \keys_multichoices_make:nn #1#2
7697 {
7698   \keys_multichoice_make:
7699   \int_zero:N \l_keys_choice_int
7700   \clist_map_inline:nn {#1}
7701   {
7702     \keys_cmd_set:nx { \l_keys_path_tl / ##1 }
7703     {
7704       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
7705       \int_set:Nn \exp_not:N \l_keys_choice_int
7706       { \int_use:N \l_keys_choice_int }
7707       \exp_not:n {#2}
7708     }
7709     \int_incr:N \l_keys_choice_int
7710   }
7711 }

```

(End definition for `\keys_multichoice_find:n`. This function is documented on page ??.)

`\keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```

7712 \cs_new_protected_nopar:Npn \keys_value_requirement:n #1
7713 {
7714   \tl_set_eq:cc
7715   { \c_keys_vars_root_tl \l_keys_path_tl .req }
7716   { c_keys_value_ #1 _tl }
7717 }

```

(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
`\keys_variable_set:cnNN` variable if needed. The three-argument version is set up so that the use of { } as an
`\keys_variable_set:NnN` N-type variable is only done once!
`\keys_variable_set:cnN`

```

7718 \cs_new_protected_nopar:Npn \keys_variable_set:NnNN #1#2#3#4
7719 {
7720   \cs_if_exist:NF #1 { \use:c { #2 _new:N } #1 }
7721   \keys_cmd_set:nx { \l_keys_path_tl }
7722   { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1} }
7723 }
7724 \cs_new_protected_nopar:Npn \keys_variable_set:NnN #1#2#3
7725 { \keys_variable_set:NnNN #1 {#2} { } #3 }
7726 \cs_generate_variant:Nn \keys_variable_set:NnNN { c }
7727 \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 ??.)

185.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

`.bool_set:N` One function for this.
`.bool_gset:N`

```

7728 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .bool_set:N } #1
7729 { \keys_bool_set:NN #1 { } }
7730 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .bool_gset:N } #1
7731 { \keys_bool_set:NN #1 g }

```

(End definition for `.bool_set:N`. This function is documented on page 168.)

`.bool_set_inverse:N` One function for this.
`.bool_gset_inverse:N`

```

7732 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .bool_set_inverse:N } #1
7733 { \keys_bool_set_inverse:NN #1 { } }
7734 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .bool_gset_inverse:N } #1
7735 { \keys_bool_set_inverse:NN #1 g }

```

(End definition for `.bool_set_inverse:N`. This function is documented on page 168.)

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

7736 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .choice: }
7737 { \keys_choice_make: }

```

(End definition for `.choice:.` This function is documented on page 169.)

`.choices:nn` For auto-generation of a series of mutually-exclusive choices. Here, `#1` will consist of two separate arguments, hence the slightly odd-looking implementation.

```

7738 \cs_new_protected:cpn { \c_keys_props_root_tl .choices:nn } #1
7739 { \keys_choices_make:nn #1 }

```

(End definition for `.choices:nn`. This function is documented on page 169.)

`.code:n` Creating code is simply a case of passing through to the underlying `set` function.

`.code:x`

```
7740 \cs_new_protected:cpn { \c_keys_props_root_tl .code:n } #1
7741 { \keys_cmd_set:nn { \l_keys_path_tl } {#1} }
7742 \cs_new_protected:cpn { \c_keys_props_root_tl .code:x } #1
7743 { \keys_cmd_set:nx { \l_keys_path_tl } {#1} }
```

(End definition for `.code:n` and `.code:x`. These functions are documented on page 169.)

`.choice_code:n` Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

`.choice_code:x`

```
7744 \cs_new_protected:cpn { \c_keys_props_root_tl .choice_code:n } #1
7745 { \keys_choice_code_store:x { \exp_not:n {#1} } }
7746 \cs_new_protected:cpn { \c_keys_props_root_tl .choice_code:x } #1
7747 { \keys_choice_code_store:x {#1} }
```

(End definition for `.choice_code:n` and `.choice_code:x`. These functions are documented on page 169.)

`.default:n` Expansion is left to the internal functions.

`.default:V`

```
7748 \cs_new_protected:cpn { \c_keys_props_root_tl .default:n } #1
7749 { \keys_default_set:n {#1} }
7750 \cs_new_protected:cpn { \c_keys_props_root_tl .default:V } #1
7751 { \keys_default_set:V #1 }
```

(End definition for `.default:n` and `.default:V`. These functions are documented on page 169.)

`.dim_set:N` Setting a variable is very easy: just pass the data along.

`.dim_set:c`

`.dim_gset:N`

`.dim_gset:c`

```
7752 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_set:N } #1
7753 { \keys_variable_set:NnN #1 { dim } n }
7754 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_set:c } #1
7755 { \keys_variable_set:cnN {#1} { dim } n }
7756 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_gset:N } #1
7757 { \keys_variable_set:NnNN #1 { dim } g n }
7758 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .dim_gset:c } #1
7759 { \keys_variable_set:cnNN {#1} { dim } g n }
```

(End definition for `.dim_set:N` and `.dim_set:c`. These functions are documented on page 170.)

`.fp_set:N` Setting a variable is very easy: just pass the data along.

`.fp_set:c`

`.fp_gset:N`

`.fp_gset:c`

```
7760 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_set:N } #1
7761 { \keys_variable_set:NnN #1 { fp } n }
7762 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_set:c } #1
7763 { \keys_variable_set:cnN {#1} { fp } n }
7764 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_gset:N } #1
7765 { \keys_variable_set:NnNN #1 { fp } g n }
7766 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .fp_gset:c } #1
7767 { \keys_variable_set:cnNN {#1} { fp } g n }
```

(End definition for `.fp_set:N` and `.fp_set:c`. These functions are documented on page 170.)

`.generate_choices:n` Making choices is easy.

```
7768 \cs_new_protected:cpn { \c_keys_props_root_tl .generate_choices:n } #1
7769 { \keys_choices_generate:n {#1} }
```

(End definition for `.generate_choices:n`. This function is documented on page 170.)

`.int_set:N` Setting a variable is very easy: just pass the data along.

```
.int_set:c
.int_gset:N
.int_gset:c
7770 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_set:N } #1
7771 { \keys_variable_set:NnN #1 { int } n }
7772 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_set:c } #1
7773 { \keys_variable_set:cnN {#1} { int } n }
7774 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_gset:N } #1
7775 { \keys_variable_set:NnNN #1 { int } g n }
7776 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .int_gset:c } #1
7777 { \keys_variable_set:cnNN {#1} { int } g n }
```

(End definition for `.int_set:N` and `.int_set:c`. These functions are documented on page 171.)

`.meta:n` Making a meta is handled internally.

```
.meta:x
7778 \cs_new_protected:cpn { \c_keys_props_root_tl .meta:n } #1
7779 { \keys_meta_make:n {#1} }
7780 \cs_new_protected:cpn { \c_keys_props_root_tl .meta:x } #1
7781 { \keys_meta_make:x {#1} }
```

(End definition for `.meta:n` and `.meta:x`. These functions are documented on page 171.)

`.multichoice:` The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.
`.multichoices:nn`

```
7782 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .multichoice: }
7783 { \keys_multichoice_make: }
7784 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoices:nn } #1
7785 { \keys_multichoices_make:nn #1 }
```

(End definition for `.multichoice:.` This function is documented on page ??.)

`.skip_set:N` Setting a variable is very easy: just pass the data along.

```
.skip_set:c
.skip_gset:N
.skip_gset:c
7786 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_set:N } #1
7787 { \keys_variable_set:NnN #1 { skip } n }
7788 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_set:c } #1
7789 { \keys_variable_set:cnN {#1} { skip } n }
7790 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_gset:N } #1
7791 { \keys_variable_set:NnNN #1 { skip } g n }
7792 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .skip_gset:c } #1
7793 { \keys_variable_set:cnNN {#1} { skip } g n }
```

(End definition for `.skip_set:N` and `.skip_set:c`. These functions are documented on page 171.)

```
.tl_set:N Setting a variable is very easy: just pass the data along.
.tl_set:c
.tl_gset:N 7794 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set:N } #1
.tl_gset:c 7795 { \keys_variable_set:NnN #1 { tl } n }
.tl_set_x:N 7796 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set:c } #1
.tl_set_x:c 7797 { \keys_variable_set:cnN {#1} { tl } n }
.tl_gset_x:N 7798 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set_x:N } #1
.tl_gset_x:c 7799 { \keys_variable_set:NnN #1 { tl } x }
7800 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_set_x:c } #1
7801 { \keys_variable_set:cnN {#1} { tl } x }
7802 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset:N } #1
7803 { \keys_variable_set:NnN #1 { tl } g n }
7804 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset:c } #1
7805 { \keys_variable_set:cnN {#1} { tl } g n }
7806 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset_x:N } #1
7807 { \keys_variable_set:NnN #1 { tl } g x }
7808 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .tl_gset_x:c } #1
7809 { \keys_variable_set:cnN {#1} { tl } g x }
```

(End definition for `.tl_set:N` and `.tl_set:c`. These functions are documented on page 172.)

```
.value_forbidden: These are very similar, so both call the same function.
.value_required: 7810 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .value_forbidden: }
7811 { \keys_value_requirement:n { forbidden } }
7812 \cs_new_protected_nopar:cpn { \c_keys_props_root_tl .value_required: }
7813 { \keys_value_requirement:n { required } }
```

(End definition for `.value_forbidden:`. This function is documented on page 172.)

185.6 Setting keys

```
\keys_set:nn A simple wrapper again.
\keys_set:nV
\keys_set:nv
\keys_set:no
\keys_set_aux:nnn 7814 \cs_new_protected:Npn \keys_set:nn
\keys_set_aux:onn 7815 { \keys_set_aux:onn { \l_keys_module_tl } }
7816 \cs_new_protected:Npn \keys_set_aux:nnn #1#2#3
7817 {
7818   \tl_set:Nn \l_keys_module_tl {#2}
7819   \keyval_parse:NNn \keys_set_elt:n \keys_set_elt:nn {#3}
7820   \tl_set:Nn \l_keys_module_tl {#1}
7821 }
7822 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
7823 \cs_generate_variant:Nn \keys_set_aux:nnn { o }
```

(End definition for `\keys_set:nn` and others. These functions are documented on page 175.)

```

\keys_set_known:nnN
\keys_set_known:nVN
\keys_set_known:nvN
\keys_set_known:noN
\keys_set_known_aux:nnnN
\keys_set_known_aux:onnN
7824 \cs_new_protected:Npn \keys_set_known:nnN
7825 { \keys_set_known_aux:onnN { \l_keys_module_tl } }
7826 \cs_new_protected:Npn \keys_set_known_aux:nnnN #1#2#3#4
7827 {
7828   \tl_set:Nn \l_keys_module_tl {#2}
7829   \clist_clear:N \l_keys_unknown_clist
7830   \cs_set_eq:NN \keys_execute_unknown: \keys_execute_unknown_alt:
7831   \keyval_parse:NNn \keys_set_elt:n \keys_set_elt:nn {#3}
7832   \cs_set_eq:NN \keys_execute_unknown: \keys_execute_unknown_std:
7833   \tl_set:Nn \l_keys_module_tl {#1}
7834   \clist_set_eq:NN #4 \l_keys_unknown_clist
7835 }
7836 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
7837 \cs_generate_variant:Nn \keys_set_known_aux:nnnN { o }

```

(End definition for `\keys_set_known:nnN` 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.

```

7838 \cs_new_protected_nopar:Npn \keys_set_elt:n #1
7839 {
7840   \bool_set_true:N \l_keys_no_value_bool
7841   \keys_set_elt_aux:nn {#1} { }
7842 }
7843 \cs_new_protected:Npn \keys_set_elt:nn #1#2
7844 {
7845   \bool_set_false:N \l_keys_no_value_bool
7846   \keys_set_elt_aux:nn {#1} {#2}
7847 }
7848 \cs_new_protected:Npn \keys_set_elt_aux:nn #1#2
7849 {
7850   \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
7851   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
7852   \keys_value_or_default:n {#2}
7853   \bool_if:nTF
7854   {
7855     \keys_if_value_p:n { required } &&
7856     \l_keys_no_value_bool
7857   }
7858   {
7859     \msg_kernel_error:nnx { keys } { value-required }
7860     { \l_keys_path_tl }
7861   }
7862   {
7863     \bool_if:nTF
7864     {

```

```

7865         \keys_if_value_p:n { forbidden } &&
7866         ! \l_keys_no_value_bool
7867     }
7868     {
7869         \msg_kernel_error:nxxx { keys } { value-forbidden }
7870         { \l_keys_path_tl } { \l_keys_value_tl }
7871     }
7872     { \keys_execute: }
7873 }
7874 }

```

(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.

```

7875 \cs_new_protected:Npn \keys_value_or_default:n #1
7876 {
7877     \tl_set:Nn \l_keys_value_tl {#1}
7878     \bool_if:NT \l_keys_no_value_bool
7879     {
7880         \quark_if_no_value:cF { \c_keys_vars_root_tl \l_keys_path_tl .default }
7881         {
7882             \cs_if_exist:cT { \c_keys_vars_root_tl \l_keys_path_tl .default }
7883             {
7884                 \tl_set_eq:Nc \l_keys_value_tl
7885                 { \c_keys_vars_root_tl \l_keys_path_tl .default }
7886             }
7887         }
7888     }
7889 }

```

(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.

```

7890 \prg_new_conditional:Npnn \keys_if_value:n #1 { p }
7891 {
7892     \tl_if_eq:ccTF { c_keys_value_ #1 _tl }
7893     { \c_keys_vars_root_tl \l_keys_path_tl .req }
7894     { \prg_return_true: }
7895     { \prg_return_false: }
7896 }

```

(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_unknown_std:`

`\keys_execute_unknown_alt:`

`\keys_execute:nn`

```

7897 \cs_new_nopar:Npn \keys_execute:
7898   { \keys_execute:nn { \l_keys_path_tl } { \keys_execute_unknown: } }
7899 \cs_new_nopar:Npn \keys_execute_unknown:
7900   {
7901     \keys_execute:nn { \l_keys_module_tl / unknown }
7902     {
7903       \msg_kernel_error:nnxx { keys } { key-unknown }
7904       { \l_keys_path_tl } { \l_keys_module_tl }
7905     }
7906   }
7907 \cs_new_eq:NN \keys_execute_unknown_std: \keys_execute_unknown:
7908 \cs_new_nopar:Npn \keys_execute_unknown_alt:
7909   {
7910     \clist_put_right:Nx \l_keys_unknown_clist
7911     {
7912       \exp_not:o \l_keys_key_tl
7913       \bool_if:NF \l_keys_no_value_bool
7914       { = { \exp_not:o \l_keys_value_tl } }
7915     }
7916   }
7917 \cs_new_nopar:Npn \keys_execute:nn #1#2
7918   {
7919     \cs_if_exist:cTF { \c_keys_code_root_tl #1 }
7920     {
7921       \exp_args:Nno \use:c { \c_keys_code_root_tl #1 }
7922       \l_keys_value_tl
7923     }
7924     {#2}
7925   }

```

(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.

```

7926 \cs_new_nopar:Npn \keys_choice_find:n #1
7927   {
7928     \keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
7929     { \keys_execute:nn { \l_keys_path_tl / unknown } { } }
7930   }

```

(End definition for `\keys_choice_find:n`. This function is documented on page ??.)

185.7 Utilities

`\keys_if_exist_p:nn` A utility for others to see if a key exists.

`\keys_if_exist:nnTF`

```

7931 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }

```

```

7932 {
7933   \cs_if_exist:cTF { \c_keys_code_root_tl #1 / #2 }
7934   { \prg_return_true: }
7935   { \prg_return_false: }
7936 }

```

(End definition for `\keys_if_exist:nn`. These functions are documented on page 176.)

`\keys_if_choice_exist_p:nnn` Just an alternative view on `\keys_if_exist:nn(TF)`.

`\keys_if_choice_exist:nnnTF`

```

7937 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3 { p , T , F , TF }
7938 {
7939   \cs_if_exist:cTF { \c_keys_code_root_tl #1 / #2 / #3 }
7940   { \prg_return_true: }
7941   { \prg_return_false: }
7942 }

```

(End definition for `\keys_if_choice_exist:nnn`. These functions are documented on page ??.)

`\keys_show:nn` Showing a key is just a question of using the correct name.

```

7943 \cs_new_nopar:Npn \keys_show:nn #1#2
7944 { \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.)

185.8 Messages

For when there is a need to complain.

```

7945 \msg_kernel_new:nnnn { keys } { boolean-values-only }
7946 { Key~'#1'~accepts~boolean-values-only. }
7947 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
7948 \msg_kernel_new:nnnn { keys } { choice-unknown }
7949 { Choice~'#2'~unknown~for~key~'#1'. }
7950 {
7951   The~key~'#1'~takes~a~limited~number~of~values.\\
7952   The~input~given,~'#2',~is~not~on~the~list~accepted.
7953 }
7954 \msg_kernel_new:nnnn { keys } { generate-choices-before-code }
7955 { No~code~available~to~generate~choices~for~key~'#1'. }
7956 {
7957   \c_msg_coding_error_text_tl
7958   Before~using~.generate_choices:n~the~code~should~be~defined~
7959   with~'.choice_code:n'~or~'.choice_code:x'.
7960 }
7961 \msg_kernel_new:nnnn { keys } { key-no-property }
7962 { No~property~given~in~definition~of~key~'#1'. }
7963 {

```

```

7964 \c_msg_coding_error_text_tl
7965 Inside~\keys_define:nn each~key~name
7966 needs~a~property: \\
7967 ~ ~ #1 .<property> \\
7968 LaTeX~did~not~find~a~'. '~to~indicate~the~start~of~a~property.
7969 }
7970 \msg_kernel_new:nnnn { keys } { key-unknown }
7971 { The~key~'#1'~is~unknown~and~is~being~ignored. }
7972 {
7973   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
7974   Check~that~you~have~spelled~the~key~name~correctly.
7975 }
7976 \msg_kernel_new:nnnn { keys } { option-unknown }
7977 { Unknown~option~'#1'~for~package~#2. }
7978 {
7979   LaTeX~has~been~asked~to~set~an~option~called~'#1'~
7980   but~the~#2~package~has~not~created~an~option~with~this~name.
7981 }
7982 \msg_kernel_new:nnnn { keys } { property-requires-value }
7983 { The~property~'#1'~requires~a~value. }
7984 {
7985   \c_msg_coding_error_text_tl
7986   LaTeX~was~asked~to~set~property~'#2'~for~key~'#1'.\\
7987   No~value~was~given~for~the~property,~and~one~is~required.
7988 }
7989 \msg_kernel_new:nnnn { keys } { property-unknown }
7990 { The~key~property~'#1'~is~unknown. }
7991 {
7992   \c_msg_coding_error_text_tl
7993   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
7994   this~property~is~not~defined.
7995 }
7996 \msg_kernel_new:nnnn { keys } { value-forbidden }
7997 { The~key~'#1'~does~not~taken~a~value. }
7998 {
7999   The~key~'#1'~should~be~given~without~a~value.\\
8000   LaTeX~will~ignore~the~given~value~'#2'.
8001 }
8002 \msg_kernel_new:nnnn { keys } { value-required }
8003 { The~key~'#1'~requires~a~value. }
8004 {
8005   The~key~'#1'~must~have~a~value.\\
8006   No~value~was~present:~the~key~will~be~ignored.
8007 }

```

185.9 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```

\KV_process_space_removal_sanitize:NNn
\KV_process_space_removal_no_sanitize:NNn
\KV_process_no_space_removal_no_sanitize:NNn
8008 \cs_new_eq:NN \KV_process_space_removal_sanitize:NNn \keyval_parse:NNn
8009 \cs_new_eq:NN \KV_process_space_removal_no_sanitize:NNn \keyval_parse:NNn
8010 \cs_new_eq:NN \KV_process_no_space_removal_no_sanitize:NNn \keyval_parse:NNn

(End definition for \KV_process_space_removal_sanitize:NNn. This function is documented on page
??.)

8011 </initex | package>

```

186 l3file implementation

The following test files are used for this code: *m3file001*.

```

8012 <*initex | package>

8013 <*package>
8014 \ProvidesExplPackage
8015 { \ExplFileName } { \ExplFileDate } { \ExplFileVersion } { \ExplFileDescription }
8016 \package_check_loaded_expl:
8017 </package>

```

\g_file_current_name_tl The name of the current file should be available at all times.

```

8018 \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.

```

8019 <*initex>
8020 \tex_everyjob:D \exp_after:wN
8021 {
8022   \tex_the:D \tex_everyjob:D
8023   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
8024 }
8025 </initex>
8026 <*package>
8027 \tl_gset_eq:NN \g_file_current_name_tl \@currname
8028 </package>

```

\g_file_stack_seq The input list of files is stored as a sequence stack.

```

8029 \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.

```

8030 \seq_new:N \g_file_record_seq

```

The current file name should be included in the file list!

```

8031 <*initex>
8032 \tex_everyjob:D \exp_after:wN
8033 {
8034   \tex_the:D \tex_everyjob:D
8035   \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
8036 }
8037 </initex>

```

\l_file_name_tl Used to return the fully-qualified name of a file.

```

8038 \tl_new:N \l_file_name_tl

```

\l_file_search_path_seq The current search path.

```

8039 \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.

```

8040 <*package>
8041 \seq_new:N \l_file_search_path_saved_seq
8042 </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 report end-of-file. For files which are in the current directory, this is straight-forward.
\g_file_test_stream For other locations, a search has to be made looking at each potential path in turn. The first location is of course treated as the correct one. If nothing is found, #2 is returned empty.
\file_add_path_search:nN

```

8043 \cs_new_protected_nopar:Npn \file_add_path:nN #1#2
8044 {
8045   \ior_open:Nn \g_file_test_stream {#1}
8046   \ior_if_eof:NTF \g_file_test_stream
8047   { \file_add_path_search:nN {#1} #2 }
8048   {
8049     \ior_close:N \g_file_test_stream
8050     \tl_set:Nx #2 {#1}
8051   }
8052 }
8053 \cs_new_protected_nopar:Npn \file_add_path_search:nN #1#2
8054 {
8055   \tl_clear:N #2
8056   <*package>
8057   \cs_if_exist:NT \input@path
8058   {
8059     \seq_set_eq:NN \l_file_search_path_saved_seq \l_file_search_path_seq
8060     \clist_map_inline:Nn \input@path
8061     { \seq_put_right:Nn \l_file_search_path_seq {##1} }

```

```

8062     }
8063 </package>
8064     \seq_map_inline:Nn \l_file_search_path_seq
8065     {
8066         \ior_open:Nn \g_file_test_stream { ##1 #1 }
8067         \ior_if_eof:NF \g_file_test_stream
8068         {
8069             \tl_set:Nx #2 { ##1 #1 }
8070             \seq_map_break:
8071         }
8072     }
8073 <*package>
8074     \cs_if_exist:NT \input@path
8075     { \seq_set_eq:NN \l_file_search_path_seq \l_file_search_path_saved_seq }
8076 </package>
8077     \ior_close:N \g_file_test_stream
8078 }

```

(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.

```

8079 \prg_new_protected_conditional:Nnn \file_if_exist:n { T , F , TF }
8080 {
8081     \file_add_path:nN {#1} \l_file_name_tl
8082     \tl_if_empty:NTF \l_file_name_tl
8083     { \prg_return_false: }
8084     { \prg_return_true: }
8085 }

```

(End definition for `\file_if_exist:n`. This function is documented on page 179.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does.

```

8086 \cs_new_protected_nopar:Npn \file_input:n #1
8087 {
8088     \file_add_path:nN {#1} \l_file_name_tl
8089     \tl_if_empty:NF \l_file_name_tl
8090     {
8091 <*initex>
8092         \seq_gput_right:Nx \g_file_record_seq {#1}
8093 </initex>
8094 <*package>
8095         \@addtofilelist {#1}
8096 </package>
8097         \seq_gpush:NV \g_file_stack_seq \g_file_current_name_tl
8098         \tl_gset:Nn \g_file_current_name_tl {#1}

```

```

8099         \exp_after:wN \tex_input:D \l_file_name_tl ~
8100         \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
8101     }
8102 }

```

(End definition for `\file_input:n`. This function is documented on page 179.)

`\file_path_include:n` Wrapper functions to manage the search path.

`\file_path_remove:n`

```

8103 \cs_new_protected_nopar:Npn \file_path_include:n #1
8104 {
8105     \seq_if_in:NnF \l_file_search_path_seq {#1}
8106     { \seq_put_right:Nn \l_file_search_path_seq {#1} }
8107 }
8108 \cs_new_protected_nopar:Npn \file_path_remove:n #1
8109 { \seq_remove_all:Nn \l_file_search_path_seq {#1} }

```

(End definition for `\file_path_include:n`. This function is documented on page 179.)

`\file_list:` A function to list all files used to the log.

```

8110 \cs_new_protected_nopar:Npn \file_list:
8111 {
8112     \seq_remove_duplicates:N \g_file_record_seq
8113     \iow_log:n { *~File~List~* }
8114     \seq_map_inline:Nn \g_file_record_seq { \iow_log:n {##1} }
8115     \iow_log:n { ***** }
8116 }

```

(End definition for `\file_list:`. This function is documented on page 180.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```

8117 <*package>
8118 \AtBeginDocument
8119 {
8120     \clist_map_inline:Nn \@filelist
8121     { \seq_put_right:Nn \g_file_record_seq {#1} }
8122 }
8123 </package>
8124 </initex | package>

```

187 l3fp Implementation

The following test files are used for this code: `m3fp003.lvt`.

```

8125 <*initex | package>

```

```

8126 <*package>
8127 \ProvidesExplPackage
8128   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
8129 \package_check_loaded_expl:
8130 </package>

```

187.1 Constants

`\c_forty_four` There is some speed to gain by moving numbers into fixed positions.

```

\c_one_million
\c_one_hundred_million
\c_five_hundred_million
\c_one_thousand_million
8131 \int_const:Nn \c_forty_four { 44 }
8132 \int_const:Nn \c_one_million { 1 000 000 }
8133 \int_const:Nn \c_one_hundred_million { 100 000 000 }
8134 \int_const:Nn \c_five_hundred_million { 500 000 000 }
8135 \int_const:Nn \c_one_thousand_million { 1 000 000 000 }

```

`\c_fp_pi_by_four_decimal_int` Parts of π for trigonometric range reduction, implemented as `int` variables for speed.

```

\c_fp_pi_by_four_extended_int
\c_fp_pi_decimal_int
\c_fp_pi_extended_int
\c_fp_two_pi_decimal_int
\c_fp_two_pi_extended_int
8136 \int_new:N \c_fp_pi_by_four_decimal_int
8137 \int_set:Nn \c_fp_pi_by_four_decimal_int { 785 398 158 }
8138 \int_new:N \c_fp_pi_by_four_extended_int
8139 \int_set:Nn \c_fp_pi_by_four_extended_int { 897 448 310 }
8140 \int_new:N \c_fp_pi_decimal_int
8141 \int_set:Nn \c_fp_pi_decimal_int { 141 592 653 }
8142 \int_new:N \c_fp_pi_extended_int
8143 \int_set:Nn \c_fp_pi_extended_int { 589 793 238 }
8144 \int_new:N \c_fp_two_pi_decimal_int
8145 \int_set:Nn \c_fp_two_pi_decimal_int { 283 185 307 }
8146 \int_new:N \c_fp_two_pi_extended_int
8147 \int_set:Nn \c_fp_two_pi_extended_int { 179 586 477 }

```

`\c_e_fp` The value e as a “machine number”.

```

8148 \tl_const:Nn \c_e_fp { + 2.718281828 e 0 }

```

`\c_one_fp` The constant value 1: used for fast comparisons.

```

8149 \tl_const:Nn \c_one_fp { + 1.000000000 e 0 }

```

`\c_pi_fp` The value π as a “machine number”.

```

8150 \tl_const:Nn \c_pi_fp { + 3.141592654 e 0 }

```

`\c_undefined_fp` A marker for undefined values.

```

8151 \tl_const:Nn \c_undefined_fp { X 0.000000000 e 0 }

```

`\c_zero_fp` The constant zero value.

```

8152 \tl_const:Nn \c_zero_fp { + 0.000000000 e 0 }

```

187.2 Variables

`\l_fp_arg_tl` A token list to store the formalised representation of the input for transcendental functions.

8153 `\tl_new:N \l_fp_arg_tl`

`\l_fp_count_int` A counter for things like the number of divisions possible.

8154 `\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.

8155 `\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`

8156 `\int_new:N \l_fp_exp_integer_int`

8157 `\int_new:N \l_fp_exp_decimal_int`

8158 `\int_new:N \l_fp_exp_extended_int`

8159 `\int_new:N \l_fp_exp_exponent_int`

`\l_fp_input_a_sign_int` Storage for the input: two storage areas as there are at most two inputs.

`\l_fp_input_a_integer_int`

`\l_fp_input_a_decimal_int`

`\l_fp_input_a_exponent_int`

`\l_fp_input_b_sign_int`

`\l_fp_input_b_integer_int`

`\l_fp_input_b_decimal_int`

`\l_fp_input_b_exponent_int`

8160 `\int_new:N \l_fp_input_a_sign_int`

8161 `\int_new:N \l_fp_input_a_integer_int`

8162 `\int_new:N \l_fp_input_a_decimal_int`

8163 `\int_new:N \l_fp_input_a_exponent_int`

8164 `\int_new:N \l_fp_input_b_sign_int`

8165 `\int_new:N \l_fp_input_b_integer_int`

8166 `\int_new:N \l_fp_input_b_decimal_int`

8167 `\int_new:N \l_fp_input_b_exponent_int`

`\l_fp_input_a_extended_int` For internal use, “extended” floating point numbers are needed.

`\l_fp_input_b_extended_int`

8168 `\int_new:N \l_fp_input_a_extended_int`

8169 `\int_new:N \l_fp_input_b_extended_int`

`\l_fp_mul_a_i_int` Multiplication requires that the decimal part is split into parts so that there are no overflows.

`\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`

8170 `\int_new:N \l_fp_mul_a_i_int`

8171 `\int_new:N \l_fp_mul_a_ii_int`

8172 `\int_new:N \l_fp_mul_a_iii_int`

8173 `\int_new:N \l_fp_mul_a_iv_int`

8174 `\int_new:N \l_fp_mul_a_v_int`

8175 `\int_new:N \l_fp_mul_a_vi_int`

```

8176 \int_new:N \l_fp_mul_b_i_int
8177 \int_new:N \l_fp_mul_b_ii_int
8178 \int_new:N \l_fp_mul_b_iii_int
8179 \int_new:N \l_fp_mul_b_iv_int
8180 \int_new:N \l_fp_mul_b_v_int
8181 \int_new:N \l_fp_mul_b_vi_int

```

\l_fp_mul_output_int Space for multiplication results.
\l_fp_mul_output_tl

```

8182 \int_new:N \l_fp_mul_output_int
8183 \tl_new:N \l_fp_mul_output_tl

```

\l_fp_output_sign_int Output is stored in the same way as input.

\l_fp_output_integer_int
\l_fp_output_decimal_int
\l_fp_output_exponent_int

```

8184 \int_new:N \l_fp_output_sign_int
8185 \int_new:N \l_fp_output_integer_int
8186 \int_new:N \l_fp_output_decimal_int
8187 \int_new:N \l_fp_output_exponent_int

```

\l_fp_output_extended_int Again, for calculations an extended part.

```

8188 \int_new:N \l_fp_output_extended_int

```

\l_fp_round_carry_bool To indicate that a digit needs to be carried forward.

```

8189 \bool_new:N \l_fp_round_carry_bool

```

\l_fp_round_decimal_tl A temporary store when rounding, to build up the decimal part without needing to do any maths.

```

8190 \tl_new:N \l_fp_round_decimal_tl

```

\l_fp_round_position_int Used to check the position for rounding.

\l_fp_round_target_int

```

8191 \int_new:N \l_fp_round_position_int
8192 \int_new:N \l_fp_round_target_int

```

\l_fp_sign_tl There are places where the sign needs to be set up “early”, so that the registers can be re-used.

```

8193 \tl_new:N \l_fp_sign_tl

```

\l_fp_split_sign_int When splitting the input it is fastest to use a fixed name for the sign part, and to transfer it after the split is complete.

```

8194 \int_new:N \l_fp_split_sign_int

```

`\l_fp_tmp_int` A scratch int: used only where the value is not carried forward.

```
8195 \int_new:N \l_fp_tmp_int
```

`\l_fp_tmp_tl` A scratch token list variable for expanding material.

```
8196 \tl_new:N \l_fp_tmp_tl
```

`\l_fp_trig_octant_int` To track which octant the trigonometric input is in.

```
8197 \int_new:N \l_fp_trig_octant_int
```

`\l_fp_trig_sign_int` Used for the calculation of trigonometric values.

`\l_fp_trig_decimal_int`
`\l_fp_trig_extended_int`

```
8198 \int_new:N \l_fp_trig_sign_int
```

```
8199 \int_new:N \l_fp_trig_decimal_int
```

```
8200 \int_new:N \l_fp_trig_extended_int
```

187.3 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`

```
8201 \cs_new_protected_nopar:Npn \fp_read:N #1
8202 { \exp_after:wN \fp_read_aux:w #1 \q_stop }
8203 \cs_new_protected_nopar:Npn \fp_read_aux:w #1#2 . #3 e #4 \q_stop
8204 {
8205   \if:w #1 -
8206     \l_fp_input_a_sign_int \c_minus_one
8207   \else:
8208     \l_fp_input_a_sign_int \c_one
8209   \fi:
8210   \l_fp_input_a_integer_int #2 \scan_stop:
8211   \l_fp_input_a_decimal_int #3 \scan_stop:
8212   \l_fp_input_a_exponent_int #4 \scan_stop:
8213 }
```

(End definition for `\fp_read:N`. This function is documented on page ??.)

`\fp_split:Nn` The aim here is to use as much of $\text{T}_{\text{E}}\text{X}$'s mechanism as possible to pick up the numerical

`\fp_split_sign:` input without any mistakes. In particular, negative numbers have to be filtered out first

`\fp_split_exponent:` in case the integer part is 0 (in which case $\text{T}_{\text{E}}\text{X}$ would drop the $-$ sign). That process

`\fp_split_aux_i:w` has to be done in a loop for cases where the sign is repeated. Finding an exponent is

`\fp_split_aux_ii:w` relatively easy, after which the next phase is to find the integer part, which will terminate

`\fp_split_aux_iii:w` with a $.$, and trigger the decimal-finding code. The later will allow the decimal to be too

`\fp_split_decimal:w` long, truncating the result.

`\fp_split_decimal_aux:w`

```
8214 \cs_new_protected_nopar:Npn \fp_split:Nn #1#2
```

```

8215 {
8216   \tl_set:Nx \l_fp_tmp_tl {#2}
8217   \tl_set_rescan:Nno \l_fp_tmp_tl { \char_set_catcode_ignore:n { 32 } }
8218   { \l_fp_tmp_tl }
8219   \l_fp_split_sign_int \c_one
8220   \fp_split_sign:
8221   \use:c { l_fp_input_ #1 _sign_int } \l_fp_split_sign_int
8222   \exp_after:wN \fp_split_exponent:w \l_fp_tmp_tl e e \q_stop #1
8223 }
8224 \cs_new_protected_nopar:Npn \fp_split_sign:
8225 {
8226   \if_int_compare:w \pdfTeX_strcmp:D
8227   { \exp_after:wN \tl_head:w \l_fp_tmp_tl ? \q_stop } { - }
8228   = \c_zero
8229   \tl_set:Nx \l_fp_tmp_tl
8230   {
8231     \exp_after:wN
8232     \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop
8233   }
8234   \l_fp_split_sign_int -\l_fp_split_sign_int
8235   \exp_after:wN \fp_split_sign:
8236   \else:
8237     \if_int_compare:w \pdfTeX_strcmp:D
8238     { \exp_after:wN \tl_head:w \l_fp_tmp_tl ? \q_stop } { + }
8239     = \c_zero
8240     \tl_set:Nx \l_fp_tmp_tl
8241     {
8242       \exp_after:wN
8243       \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop
8244     }
8245     \exp_after:wN \exp_after:wN \exp_after:wN \fp_split_sign:
8246     \fi:
8247   \fi:
8248 }
8249 \cs_new_protected_nopar:Npn \fp_split_exponent:w #1 e #2 e #3 \q_stop #4
8250 {
8251   \use:c { l_fp_input_ #4 _exponent_int }
8252   \int_eval:w 0 #2 \scan_stop:
8253   \tex_afterassignment:D \fp_split_aux_i:w
8254   \use:c { l_fp_input_ #4 _integer_int }
8255   \int_eval:w 0 #1 . . \q_stop #4
8256 }
8257 \cs_new_protected_nopar:Npn \fp_split_aux_i:w #1 . #2 . #3 \q_stop
8258 { \fp_split_aux_ii:w #2 000000000 \q_stop }
8259 \cs_new_protected_nopar:Npn \fp_split_aux_ii:w #1#2#3#4#5#6#7#8#9
8260 { \fp_split_aux_iii:w {#1#2#3#4#5#6#7#8#9} }
8261 \cs_new_protected_nopar:Npn \fp_split_aux_iii:w #1#2 \q_stop
8262 {
8263   \l_fp_tmp_int 1 #1 \scan_stop:
8264   \exp_after:wN \fp_split_decimal:w

```

```

8265     \int_use:N \l_fp_tmp_int 00000000 \q_stop
8266   }
8267   \cs_new_protected_nopar:Npn \fp_split_decimal:w #1#2#3#4#5#6#7#8#9
8268   { \fp_split_decimal_aux:w {#2#3#4#5#6#7#8#9} }
8269   \cs_new_protected_nopar:Npn \fp_split_decimal_aux:w #1#2#3 \q_stop #4
8270   {
8271     \use:c { l_fp_input_ #4 _decimal_int } #1#2 \scan_stop:
8272     \if_int_compare:w
8273       \int_eval:w
8274         \use:c { l_fp_input_ #4 _integer_int } +
8275         \use:c { l_fp_input_ #4 _decimal_int }
8276       \scan_stop:
8277       = \c_zero
8278     \use:c { l_fp_input_ #4 _sign_int } \c_one
8279   \fi:
8280   \if_int_compare:w
8281     \use:c { l_fp_input_ #4 _integer_int } < \c_one_thousand_million
8282   \else:
8283     \exp_after:wN \fp_overflow_msg:
8284   \fi:
8285 }

```

(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.

```

8286   \cs_new_protected_nopar:Npn \fp_standardise:NNNN #1#2#3#4
8287   {
8288     \if_int_compare:w
8289       \int_eval:w #2 + #3 = \c_zero
8290       #1 \c_one
8291       #4 \c_zero
8292     \exp_after:wN \use_none:nnnn
8293   \else:
8294     \exp_after:wN \fp_standardise_aux:NNNN
8295   \fi:
8296   #1#2#3#4
8297 }
8298   \cs_new_protected_nopar:Npn \fp_standardise_aux:NNNN #1#2#3#4
8299   {
8300     \cs_set_protected_nopar:Npn \fp_standardise_aux:
8301     {
8302       \if_int_compare:w #2 = \c_zero
8303         \tex_advance:D #3 \c_one_thousand_million
8304         \exp_after:wN \fp_standardise_aux:w
8305         \int_use:N #3 \q_stop
8306         \exp_after:wN \fp_standardise_aux:
8307       \fi:
8308     }

```

```

8309 \cs_set_protected_nopar:Npn
8310 \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9 \q_stop
8311 {
8312   #2 ##2 \scan_stop:
8313   #3 ##3##4##5##6##7##8##9 0 \scan_stop:
8314   \tex_advance:D #4 \c_minus_one
8315 }
8316 \fp_standardise_aux:
8317 \cs_set_protected_nopar:Npn \fp_standardise_aux:
8318 {
8319   \if_int_compare:w #2 > \c_nine
8320     \tex_advance:D #2 \c_one_thousand_million
8321     \exp_after:wN \use_i:nn \exp_after:wN
8322     \fp_standardise_aux:w \int_use:N #2
8323     \exp_after:wN \fp_standardise_aux:
8324   \fi:
8325 }
8326 \cs_set_protected_nopar:Npn
8327 \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9
8328 {
8329   #2 ##1##2##3##4##5##6##7##8 \scan_stop:
8330   \tex_advance:D #3 \c_one_thousand_million
8331   \tex_divide:D #3 \c_ten
8332   \tl_set:Nx \l_fp_tmp_tl
8333   {
8334     ##9
8335     \exp_after:wN \use_none:n \int_use:N #3
8336   }
8337   #3 \l_fp_tmp_tl \scan_stop:
8338   \tex_advance:D #4 \c_one
8339 }
8340 \fp_standardise_aux:
8341 \if_int_compare:w #4 < \c_one_hundred
8342 \if_int_compare:w #4 > -\c_one_hundred
8343 \else:
8344   #1 \c_one
8345   #2 \c_zero
8346   #3 \c_zero
8347   #4 \c_zero
8348 \fi:
8349 \else:
8350   \exp_after:wN \fp_overflow_msg:
8351 \fi:
8352 }
8353 \cs_new_protected_nopar:Npn \fp_standardise_aux: { }
8354 \cs_new_protected_nopar:Npn \fp_standardise_aux:w { }

```

(End definition for `\fp_standardise:NNMN`. This function is documented on page ??.)

187.4 Internal utilities

`\fp_level_input_exponents:` The routines here are similar to those used to standardise the exponent. However, the aim here is different: the two exponents need to end up the same.

`\fp_level_input_exponents_a:`

`\fp_level_input_exponents_a:NNNNNNNNN`

`\fp_level_input_exponents_b:`

`\fp_level_input_exponents_b:NNNNNNNNN`

```

8355 \cs_new_protected_nopar:Npn \fp_level_input_exponents:
8356 {
8357   \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
8358     \exp_after:wN \fp_level_input_exponents_a:
8359   \else:
8360     \exp_after:wN \fp_level_input_exponents_b:
8361   \fi:
8362 }
8363 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a:
8364 {
8365   \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
8366     \tex_advance:D \l_fp_input_b_integer_int \c_one_thousand_million
8367     \exp_after:wN \use_i:nn \exp_after:wN
8368       \fp_level_input_exponents_a:NNNNNNNNN
8369     \int_use:N \l_fp_input_b_integer_int
8370     \exp_after:wN \fp_level_input_exponents_a:
8371   \fi:
8372 }
8373 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a:NNNNNNNNN
8374   #1#2#3#4#5#6#7#8#9
8375 {
8376   \l_fp_input_b_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
8377   \tex_advance:D \l_fp_input_b_decimal_int \c_one_thousand_million
8378   \tex_divide:D \l_fp_input_b_decimal_int \c_ten
8379   \tl_set:Nx \l_fp_tmp_tl
8380     {
8381       #9
8382       \exp_after:wN \use_none:n
8383       \int_use:N \l_fp_input_b_decimal_int
8384     }
8385   \l_fp_input_b_decimal_int \l_fp_tmp_tl \scan_stop:
8386   \tex_advance:D \l_fp_input_b_exponent_int \c_one
8387 }
8388 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b:
8389 {
8390   \if_int_compare:w \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
8391     \tex_advance:D \l_fp_input_a_integer_int \c_one_thousand_million
8392     \exp_after:wN \use_i:nn \exp_after:wN
8393       \fp_level_input_exponents_b:NNNNNNNNN
8394     \int_use:N \l_fp_input_a_integer_int
8395     \exp_after:wN \fp_level_input_exponents_b:
8396   \fi:
8397 }
8398 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b:NNNNNNNNN
8399   #1#2#3#4#5#6#7#8#9

```

```

8400 {
8401   \l_fp_input_a_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
8402   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8403   \tex_divide:D \l_fp_input_a_decimal_int \c_ten
8404   \tl_set:Nx \l_fp_tmp_tl
8405   {
8406     #9
8407     \exp_after:wN \use_none:n
8408     \int_use:N \l_fp_input_a_decimal_int
8409   }
8410   \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
8411   \tex_advance:D \l_fp_input_a_exponent_int \c_one
8412 }

```

(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.

```

8413 \cs_new_protected_nopar:Npn \fp_tmp:w #1#2 { }

```

(End definition for `\fp_tmp:w`.)

187.5 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`

```

8414 \cs_new_protected_nopar:Npn \fp_new:N #1
8415 {
8416   \tl_new:N #1
8417   \tl_gset_eq:NN #1 \c_zero_fp
8418 }
8419 \cs_generate_variant:Nn \fp_new:N { c }

```

(End definition for `\fp_new:N` and `\fp_new:c`. These functions are documented on page 181.)

`\fp_const:Nn` A simple wrapper.

`\fp_const:cn`

```

8420 \cs_new_protected_nopar:Npn \fp_const:Nn #1#2
8421 {
8422   \fp_new:N #1
8423   \fp_gset:Nn #1 {#2}
8424 }
8425 \cs_generate_variant:Nn \fp_const:Nn { c }

```

(End definition for `\fp_const:Nn` and `\fp_const:cn`. These functions are documented on page 181.)

`\fp_zero:N` Zeroing fixed-points is pretty obvious.
`\fp_zero:c`
`\fp_gzero:N`
`\fp_gzero:c`

```

8426 \cs_new_protected_nopar:Npn \fp_zero:N #1
8427 { \tl_set_eq:NN #1 \c_zero_fp }
8428 \cs_new_protected_nopar:Npn \fp_gzero:N #1
8429 { \tl_gset_eq:NN #1 \c_zero_fp }
8430 \cs_generate_variant:Nn \fp_zero:N { c }
8431 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for `\fp_zero:N` and `\fp_zero:c`. These functions are documented on page 182.)

`\fp_set:Nn` To trap any input errors, a very simple version of the parser is run here. This will pick
`\fp_set:cn` up any invalid characters at this stage, saving issues later. The splitting approach is the
`\fp_gset:Nn` same as the more advanced function later.
`\fp_gset:cn`

```

8432 \cs_new_protected_nopar:Npn \fp_set:Nn { \fp_set_aux:NNn \tl_set:Nn }
8433 \cs_new_protected_nopar:Npn \fp_gset:Nn { \fp_set_aux:NNn \tl_gset:Nn }
8434 \cs_new_protected_nopar:Npn \fp_set_aux:NNn #1#2#3
8435 {
8436   \group_begin:
8437   \fp_split:Nn a {#3}
8438   \fp_standardise:NNNN
8439   \l_fp_input_a_sign_int
8440   \l_fp_input_a_integer_int
8441   \l_fp_input_a_decimal_int
8442   \l_fp_input_a_exponent_int
8443   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8444   \cs_set_protected_nopar:Npx \fp_tmp:w
8445   {
8446     \group_end:
8447     #1 \exp_not:N #2
8448     {
8449       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8450       -
8451       \else:
8452       +
8453       \fi:
8454       \int_use:N \l_fp_input_a_integer_int
8455       .
8456       \exp_after:wN \use_none:n
8457       \int_use:N \l_fp_input_a_decimal_int
8458       e
8459       \int_use:N \l_fp_input_a_exponent_int
8460     }
8461   }
8462   \fp_tmp:w
8463 }
8464 \cs_generate_variant:Nn \fp_set:Nn { c }
8465 \cs_generate_variant:Nn \fp_gset:Nn { c }
```

(End definition for `\fp_set:Nn` and `\fp_set:cn`. These functions are documented on page 182.)

```

\fp_set_from_dim:Nn Here, dimensions are converted to fixed-points via a temporary variable. This ensures
\fp_set_from_dim:cn that they always convert as points. The code is then essentially the same as for \fp_
\fp_gset_from_dim:Nn set:Nn, but with the dimension passed so that it will be striped of the pt on the way
\fp_gset_from_dim:cn through. The passage through a skip is used to remove any rubber part.

\fp_set_from_dim_aux:NNn
\fp_set_from_dim_aux:w
  \l_fp_tmp_dim
  \l_fp_tmp_skip
8466 \cs_new_protected_nopar:Npn \fp_set_from_dim:Nn
8467 { \fp_set_from_dim_aux:NNn \tl_set:Nx }
8468 \cs_new_protected_nopar:Npn \fp_gset_from_dim:Nn
8469 { \fp_set_from_dim_aux:NNn \tl_gset:Nx }
8470 \cs_new_protected_nopar:Npn \fp_set_from_dim_aux:NNn #1#2#3
8471 {
8472   \group_begin:
8473   \l_fp_tmp_skip \etex_glueexpr:D #3 \scan_stop:
8474   \l_fp_tmp_dim \l_fp_tmp_skip
8475   \fp_split:Nn a
8476   {
8477     \exp_after:wN \fp_set_from_dim_aux:w
8478     \dim_use:N \l_fp_tmp_dim
8479   }
8480   \fp_standardise:NNNN
8481   \l_fp_input_a_sign_int
8482   \l_fp_input_a_integer_int
8483   \l_fp_input_a_decimal_int
8484   \l_fp_input_a_exponent_int
8485   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8486   \cs_set_protected_nopar:Npx \fp_tmp:w
8487   {
8488     \group_end:
8489     #1 \exp_not:N #2
8490     {
8491       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8492       -
8493       \else:
8494       +
8495       \fi:
8496       \int_use:N \l_fp_input_a_integer_int
8497       .
8498       \exp_after:wN \use_none:n
8499       \int_use:N \l_fp_input_a_decimal_int
8500       e
8501       \int_use:N \l_fp_input_a_exponent_int
8502     }
8503   }
8504   \fp_tmp:w
8505 }
8506 \cs_set_protected_nopar:Npx \fp_set_from_dim_aux:w
8507 {
8508   \cs_set_nopar:Npn \exp_not:N \fp_set_from_dim_aux:w

```

```

8509     ##1 \tl_to_str:n { pt } {##1}
8510   }
8511   \fp_set_from_dim_aux:w
8512   \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
8513   \cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
8514   \dim_new:N \l_fp_tmp_dim
8515   \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
8516 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
8517 \cs_new_eq:NN \fp_set_eq:cN \tl_set_eq:cN
8518 \cs_new_eq:NN \fp_set_eq:Nc \tl_set_eq:Nc
8519 \cs_new_eq:NN \fp_set_eq:cc \tl_set_eq:cc
8520 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
8521 \cs_new_eq:NN \fp_gset_eq:cN \tl_gset_eq:cN
8522 \cs_new_eq:NN \fp_gset_eq:Nc \tl_gset_eq:Nc
8523 \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 181.)

`\fp_show:N` Simple showing of the underlying variable.

```

\fp_show:c
8524 \cs_new_eq:NN \fp_show:N \tl_show:N
8525 \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 183.)

`\fp_use:N` The idea of the `\fp_use:N` function to convert the stored value into something suitable 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:c
\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
8526 \cs_new_nopar:Npn \fp_use:N #1
8527 { \exp_after:wN \fp_use_aux:w #1 \q_stop }
8528 \cs_generate_variant:Nn \fp_use:N { c }
8529 \cs_new_nopar:Npn \fp_use_aux:w #1#2 e #3 \q_stop
8530 {
8531   \if:w #1 -
8532     -
8533   \fi:
8534   \if_int_compare:w #3 > \c_zero
8535     \exp_after:wN \fp_use_large:w
8536   \else:
8537     \if_int_compare:w #3 < \c_zero
8538       \exp_after:wN \exp_after:wN \exp_after:wN
8539       \fp_use_small:w
8540     \else:

```

```

8541         \exp_after:wN \exp_after:wN \exp_after:wN \fp_use_none:w
8542         \fi:
8543     \fi:
8544     #2 e #3 \q_stop
8545 }

```

When the exponent is zero, the input is simply returned as output.

```

8546 \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.

```

8547 \cs_new_nopar:Npn \fp_use_small:w #1 . #2 e #3 \q_stop
8548 {
8549     0 .
8550     \prg_replicate:nn { -#3 - 1 } { 0 }
8551     #1#2
8552 }

```

Life is more complex for large numbers. The decimal point needs to be shuffled, with potentially some zero-filling for very large values.

```

8553 \cs_new_nopar:Npn \fp_use_large:w #1 . #2 e #3 \q_stop
8554 {
8555     \if_int_compare:w #3 < \c_ten
8556         \exp_after:wN \fp_use_large_aux_i:w
8557     \else:
8558         \exp_after:wN \fp_use_large_aux_ii:w
8559     \fi:
8560     #1#2 e #3 \q_stop
8561 }
8562 \cs_new_nopar:Npn \fp_use_large_aux_i:w #1#2 e #3 \q_stop
8563 {
8564     #1
8565     \use:c { fp_use_large_aux_ #3 :w } #2 \q_stop
8566 }
8567 \cs_new_nopar:cpn { fp_use_large_aux_1:w } #1#2 \q_stop { #1 . #2 }
8568 \cs_new_nopar:cpn { fp_use_large_aux_2:w } #1#2#3 \q_stop
8569 { #1#2 . #3 }
8570 \cs_new_nopar:cpn { fp_use_large_aux_3:w } #1#2#3#4 \q_stop
8571 { #1#2#3 . #4 }
8572 \cs_new_nopar:cpn { fp_use_large_aux_4:w } #1#2#3#4#5 \q_stop
8573 { #1#2#3#4 . #5 }
8574 \cs_new_nopar:cpn { fp_use_large_aux_5:w } #1#2#3#4#5#6 \q_stop
8575 { #1#2#3#4#5 . #6 }
8576 \cs_new_nopar:cpn { fp_use_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop
8577 { #1#2#3#4#5#6 . #7 }
8578 \cs_new_nopar:cpn { fp_use_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop
8579 { #1#2#3#4#6#7 . #8 }
8580 \cs_new_nopar:cpn { fp_use_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop

```

```

8581 { #1#2#3#4#5#6#7#8 . #9 }
8582 \cs_new_nopar:cpn { fp_use_large_aux_9:w } #1 \q_stop { #1 . }
8583 \cs_new_nopar:Npn \fp_use_large_aux_ii:w #1 e #2 \q_stop
8584 {
8585   #1
8586   \prg_replicate:nn { #2 - 9 } { 0 }
8587   .
8588 }

```

(End definition for `\fp_use:N` and `\fp_use:c`. These functions are documented on page 182.)

187.6 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`

```

8589 \cs_new_nopar:Npn \fp_to_dim:N #1 { \fp_use:N #1 pt }
8590 \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 183.)

`\fp_to_int:N` Converting to integers in an expandable manner is very similar to simply using floating
`\fp_to_int:c` point variables, particularly in the lead-off.

```

\fp_to_int_aux:w
\fp_to_int_none:w
\fp_to_int_small:w
\fp_to_int_large:w
\fp_to_int_large_aux_i:w
\fp_to_int_large_aux_1:w
\fp_to_int_large_aux_2:w
\fp_to_int_large_aux_3:w
\fp_to_int_large_aux_4:w
\fp_to_int_large_aux_5:w
\fp_to_int_large_aux_6:w
\fp_to_int_large_aux_7:w
\fp_to_int_large_aux_8:w
\fp_to_int_large_aux_i:w
\fp_to_int_large_aux:nnn
\fp_to_int_large_aux_ii:w
8591 \cs_new_nopar:Npn \fp_to_int:N #1
8592 { \exp_after:wN \fp_to_int_aux:w #1 \q_stop }
8593 \cs_generate_variant:Nn \fp_to_int:N { c }
8594 \cs_new_nopar:Npn \fp_to_int_aux:w #1#2 e #3 \q_stop
8595 {
8596   \if:w #1 -
8597   -
8598   \fi:
8599   \if_int_compare:w #3 < \c_zero
8600     \exp_after:wN \fp_to_int_small:w
8601   \else:
8602     \exp_after:wN \fp_to_int_large:w
8603   \fi:
8604   #2 e #3 \q_stop
8605 }

```

For small numbers, if the decimal part is greater than a half then there is rounding up to do.

```

8606 \cs_new_nopar:Npn \fp_to_int_small:w #1 . #2 e #3 \q_stop
8607 {
8608   \if_int_compare:w #3 > \c_one
8609   \else:

```

```

8610     \if_int_compare:w #1 < \c_five
8611     0
8612     \else:
8613     1
8614     \fi:
8615 \fi:
8616 }

```

For large numbers, the idea is to split off the part for rounding, do the rounding and fill if needed.

```

8617 \cs_new_nopar:Npn \fp_to_int_large:w #1 . #2 e #3 \q_stop
8618 {
8619     \if_int_compare:w #3 < \c_ten
8620     \exp_after:wN \fp_to_int_large_aux_i:w
8621     \else:
8622     \exp_after:wN \fp_to_int_large_aux_ii:w
8623     \fi:
8624     #1#2 e #3 \q_stop
8625 }
8626 \cs_new_nopar:Npn \fp_to_int_large_aux_i:w #1#2 e #3 \q_stop
8627 { \use:c { fp_to_int_large_aux_#3 :w } #2 \q_stop {#1} }
8628 \cs_new_nopar:cpn { fp_to_int_large_aux_1:w } #1#2 \q_stop
8629 { \fp_to_int_large_aux:nnn { #2 0 } {#1} }
8630 \cs_new_nopar:cpn { fp_to_int_large_aux_2:w } #1#2#3 \q_stop
8631 { \fp_to_int_large_aux:nnn { #3 00 } {#1#2} }
8632 \cs_new_nopar:cpn { fp_to_int_large_aux_3:w } #1#2#3#4 \q_stop
8633 { \fp_to_int_large_aux:nnn { #4 000 } {#1#2#3} }
8634 \cs_new_nopar:cpn { fp_to_int_large_aux_4:w } #1#2#3#4#5 \q_stop
8635 { \fp_to_int_large_aux:nnn { #5 0000 } {#1#2#3#4} }
8636 \cs_new_nopar:cpn { fp_to_int_large_aux_5:w } #1#2#3#4#5#6 \q_stop
8637 { \fp_to_int_large_aux:nnn { #6 00000 } {#1#2#3#4#5} }
8638 \cs_new_nopar:cpn { fp_to_int_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop
8639 { \fp_to_int_large_aux:nnn { #7 000000 } {#1#2#3#4#5#6} }
8640 \cs_new_nopar:cpn { fp_to_int_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop
8641 { \fp_to_int_large_aux:nnn { #8 0000000 } {#1#2#3#4#5#6#7} }
8642 \cs_new_nopar:cpn { fp_to_int_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop
8643 { \fp_to_int_large_aux:nnn { #9 00000000 } {#1#2#3#4#5#6#7#8} }
8644 \cs_new_nopar:cpn { fp_to_int_large_aux_9:w } #1 \q_stop {#1}
8645 \cs_new_nopar:Npn \fp_to_int_large_aux:nnn #1#2#3
8646 {
8647     \if_int_compare:w #1 < \c_five_hundred_million
8648     #3#2
8649     \else:
8650     \int_value:w \int_eval:w #3#2 + 1 \int_eval_end:
8651     \fi:
8652 }
8653 \cs_new_nopar:Npn \fp_to_int_large_aux_ii:w #1 e #2 \q_stop
8654 {
8655     #1

```

```

8656 \prg_replicate:nn { #2 - 9 } { 0 }
8657 }

```

(End definition for `\fp_to_int:N` and `\fp_to_int:c`. These functions are documented on page 183.)

`\fp_to_tl:N` Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

`\fp_to_tl:c`

```

\fp_to_tl_aux:w
\fp_to_tl_large:w
\fp_to_tl_large_aux_i:w
\fp_to_tl_large_aux_ii:w
\fp_to_tl_large_0:w
\fp_to_tl_large_1:w
\fp_to_tl_large_2:w
\fp_to_tl_large_3:w
\fp_to_tl_large_4:w
\fp_to_tl_large_5:w
\fp_to_tl_large_6:w
\fp_to_tl_large_7:w
\fp_to_tl_large_8:w
\fp_to_tl_large_8_aux:w
\fp_to_tl_large_9:w
\fp_to_tl_small:w
\fp_to_tl_small_one:w
\fp_to_tl_small_two:w
\fp_to_tl_small_aux:w
\fp_to_tl_large_zeros:NNNNNNNNN
\fp_to_tl_small_zeros:NNNNNNNNN
\fp_use_iix_ix:NNNNNNNNN
\fp_use_ix:NNNNNNNNN
\fp_use_i_to_vii:NNNNNNNNN
\fp_use_i_to_iix:NNNNNNNNN
8658 \cs_new_nopar:Npn \fp_to_tl:N #1
8659 { \exp_after:wN \fp_to_tl_aux:w #1 \q_stop }
8660 \cs_generate_variant:Nn \fp_to_tl:N { c }
8661 \cs_new_nopar:Npn \fp_to_tl_aux:w #1#2 e #3 \q_stop
8662 {
8663   \if:w #1 -
8664     -
8665   \fi:
8666   \if_int_compare:w #3 < \c_zero
8667     \exp_after:wN \fp_to_tl_small:w
8668   \else:
8669     \exp_after:wN \fp_to_tl_large:w
8670   \fi:
8671   #2 e #3 \q_stop
8672 }
8673 \cs_new_nopar:Npn \fp_to_tl_large:w #1 e #2 \q_stop
8674 {
8675   \if_int_compare:w #2 < \c_ten
8676     \exp_after:wN \fp_to_tl_large_aux_i:w
8677   \else:
8678     \exp_after:wN \fp_to_tl_large_aux_ii:w
8679   \fi:
8680   #1 e #2 \q_stop
8681 }
8682 \cs_new_nopar:Npn \fp_to_tl_large_aux_i:w #1 e #2 \q_stop
8683 { \use:c { fp_to_tl_large_ #2 :w } #1 \q_stop }
8684 \cs_new_nopar:Npn \fp_to_tl_large_aux_ii:w #1 . #2 e #3 \q_stop
8685 {
8686   #1
8687   \fp_to_tl_large_zeros:NNNNNNNNN #2
8688   e #3
8689 }
8690 \cs_new_nopar:cpn { fp_to_tl_large_0:w } #1 . #2 \q_stop
8691 {
8692   #1
8693   \fp_to_tl_large_zeros:NNNNNNNNN #2
8694 }

```

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.

```

8695 \cs_new_nopar:cpn { fp_to_tl_large_1:w } #1 . #2#3 \q_stop
8696 {
8697     #1#2
8698     \fp_to_tl_large_zeros:NNNNNNNN #3 0
8699 }
8700 \cs_new_nopar:cpn { fp_to_tl_large_2:w } #1 . #2#3#4 \q_stop
8701 {
8702     #1#2#3
8703     \fp_to_tl_large_zeros:NNNNNNNN #4 00
8704 }
8705 \cs_new_nopar:cpn { fp_to_tl_large_3:w } #1 . #2#3#4#5 \q_stop
8706 {
8707     #1#2#3#4
8708     \fp_to_tl_large_zeros:NNNNNNNN #5 000
8709 }
8710 \cs_new_nopar:cpn { fp_to_tl_large_4:w } #1 . #2#3#4#5#6 \q_stop
8711 {
8712     #1#2#3#4#5
8713     \fp_to_tl_large_zeros:NNNNNNNN #6 0000
8714 }
8715 \cs_new_nopar:cpn { fp_to_tl_large_5:w } #1 . #2#3#4#5#6#7 \q_stop
8716 {
8717     #1#2#3#4#5#6
8718     \fp_to_tl_large_zeros:NNNNNNNN #7 00000
8719 }
8720 \cs_new_nopar:cpn { fp_to_tl_large_6:w } #1 . #2#3#4#5#6#7#8 \q_stop
8721 {
8722     #1#2#3#4#5#6#7
8723     \fp_to_tl_large_zeros:NNNNNNNN #8 000000
8724 }
8725 \cs_new_nopar:cpn { fp_to_tl_large_7:w } #1 . #2#3#4#5#6#7#8#9 \q_stop
8726 {
8727     #1#2#3#4#5#6#7#8
8728     \fp_to_tl_large_zeros:NNNNNNNN #9 0000000
8729 }
8730 \cs_new_nopar:cpn { fp_to_tl_large_8:w } #1 .
8731 {
8732     #1
8733     \use:c { fp_to_tl_large_8_aux:w }
8734 }
8735 \cs_new_nopar:cpn { fp_to_tl_large_8_aux:w } #1#2#3#4#5#6#7#8#9 \q_stop
8736 {
8737     #1#2#3#4#5#6#7#8
8738     \fp_to_tl_large_zeros:NNNNNNNN #9 00000000
8739 }
8740 \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.

```

8741 \cs_new_nopar:Npn \fp_to_tl_small:w #1 e #2 \q_stop
8742 {
8743   \if_int_compare:w #2 = \c_minus_one
8744     \exp_after:wN \fp_to_tl_small_one:w
8745   \else:
8746     \if_int_compare:w #2 = -\c_two
8747       \exp_after:wN \exp_after:wN \exp_after:wN \fp_to_tl_small_two:w
8748     \else:
8749       \exp_after:wN \exp_after:wN \exp_after:wN \fp_to_tl_small_aux:w
8750     \fi:
8751   \fi:
8752   #1 e #2 \q_stop
8753 }
8754 \cs_new_nopar:Npn \fp_to_tl_small_one:w #1 . #2 e #3 \q_stop
8755 {
8756   \if_int_compare:w \fp_use_ix:NNNNNNNN #2 > \c_four
8757     \if_int_compare:w
8758       \int_eval:w #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
8759       < \c_one_thousand_million
8760       0.
8761     \exp_after:wN \fp_to_tl_small_zeros:NNNNNNNN
8762     \int_value:w \int_eval:w
8763       #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
8764     \int_eval_end:
8765   \else:
8766     1
8767   \fi:
8768   \else:
8769     0. #1
8770   \fp_to_tl_small_zeros:NNNNNNNN #2
8771   \fi:
8772 }
8773 \cs_new_nopar:Npn \fp_to_tl_small_two:w #1 . #2 e #3 \q_stop
8774 {
8775   \if_int_compare:w \fp_use_iix_ix:NNNNNNNN #2 > \c_forty_four
8776     \if_int_compare:w
8777       \int_eval:w #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
8778       < \c_one_thousand_million
8779       0.0
8780     \exp_after:wN \fp_to_tl_small_zeros:NNNNNNNN
8781     \int_value:w \int_eval:w
8782       #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
8783     \int_eval_end:
8784   \else:
8785     0.1
8786   \fi:
8787   \else:
8788     0.0

```

```

8789         #1
8790         \fp_to_tl_small_zeros:NNNNNNNN #2
8791     \fi:
8792 }
8793 \cs_new_nopar:Npn \fp_to_tl_small_aux:w #1 . #2 e #3 \q_stop
8794 {
8795     #1
8796     \fp_to_tl_large_zeros:NNNNNNNN #2
8797     e #3
8798 }

```

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.

```

8799 \cs_new_nopar:Npn \fp_to_tl_large_zeros:NNNNNNNN #1#2#3#4#5#6#7#8#9
8800 {
8801     \if_int_compare:w #9 = \c_zero
8802     \if_int_compare:w #8 = \c_zero
8803     \if_int_compare:w #7 = \c_zero
8804     \if_int_compare:w #6 = \c_zero
8805     \if_int_compare:w #5 = \c_zero
8806     \if_int_compare:w #4 = \c_zero
8807     \if_int_compare:w #3 = \c_zero
8808     \if_int_compare:w #2 = \c_zero
8809     \if_int_compare:w #1 = \c_zero
8810     \else:
8811         . #1
8812     \fi:
8813     \else:
8814         . #1#2
8815     \fi:
8816     \else:
8817         . #1#2#3
8818     \fi:
8819     \else:
8820         . #1#2#3#4
8821     \fi:
8822     \else:
8823         . #1#2#3#4#5
8824     \fi:
8825     \else:
8826         . #1#2#3#4#5#6
8827     \fi:
8828     \else:
8829         . #1#2#3#4#5#6#7
8830     \fi:
8831     \else:
8832         . #1#2#3#4#5#6#7#8
8833     \fi:
8834     \else:

```

```

8835     . #1#2#3#4#5#6#7#8#9
8836     \fi:
8837 }
8838 \cs_new_nopar:Npn \fp_to_tl_small_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9
8839 {
8840     \if_int_compare:w #9 = \c_zero
8841     \if_int_compare:w #8 = \c_zero
8842     \if_int_compare:w #7 = \c_zero
8843     \if_int_compare:w #6 = \c_zero
8844     \if_int_compare:w #5 = \c_zero
8845     \if_int_compare:w #4 = \c_zero
8846     \if_int_compare:w #3 = \c_zero
8847     \if_int_compare:w #2 = \c_zero
8848     \if_int_compare:w #1 = \c_zero
8849     \else:
8850         #1
8851         \fi:
8852     \else:
8853         #1#2
8854         \fi:
8855     \else:
8856         #1#2#3
8857         \fi:
8858     \else:
8859         #1#2#3#4
8860         \fi:
8861     \else:
8862         #1#2#3#4#5
8863         \fi:
8864     \else:
8865         #1#2#3#4#5#6
8866         \fi:
8867     \else:
8868         #1#2#3#4#5#6#7
8869         \fi:
8870     \else:
8871         #1#2#3#4#5#6#7#8
8872         \fi:
8873     \else:
8874         #1#2#3#4#5#6#7#8#9
8875     \fi:
8876 }

```

Some quick “return a few” functions.

```

8877 \cs_new_nopar:Npn \fp_use_iix_ix:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {#8#9}
8878 \cs_new_nopar:Npn \fp_use_ix:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {#9}
8879 \cs_new_nopar:Npn \fp_use_i_to_vii:NNNNNNNNN #1#2#3#4#5#6#7#8#9
8880     {#1#2#3#4#5#6#7}
8881 \cs_new_nopar:Npn \fp_use_i_to_iix:NNNNNNNNN #1#2#3#4#5#6#7#8#9

```

```
8882 {#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 183.)

187.7 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

8883 \cs_new_protected_nopar:Npn \fp_round_figures:Nn
8884 { \fp_round_figures_aux:NNn \tl_set:Nn }
8885 \cs_generate_variant:Nn \fp_round_figures:Nn { c }
8886 \cs_new_protected_nopar:Npn \fp_ground_figures:Nn
8887 { \fp_round_figures_aux:NNn \tl_gset:Nn }
8888 \cs_generate_variant:Nn \fp_ground_figures:Nn { c }
8889 \cs_new_protected_nopar:Npn \fp_round_figures_aux:NNn #1#2#3
8890 {
8891   \group_begin:
8892   \fp_read:N #2
8893   \int_set:Nn \l_fp_round_target_int { #3 - 1 }
8894   \if_int_compare:w \l_fp_round_target_int < \c_ten
8895     \exp_after:wN \fp_round:
8896   \fi:
8897   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8898   \cs_set_protected_nopar:Npx \fp_tmp:w
8899   {
8900     \group_end:
8901     #1 \exp_not:N #2
8902     {
8903       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8904         -
8905       \else:
8906         +
8907       \fi:
8908       \int_use:N \l_fp_input_a_integer_int
8909       .
8910       \exp_after:wN \use_none:n
8911       \int_use:N \l_fp_input_a_decimal_int
8912       e
8913       \int_use:N \l_fp_input_a_exponent_int
8914     }
8915   }
8916   \fp_tmp:w
8917 }
```

(End definition for `\fp_round_figures:Nn` and `\fp_round_figures:cn`. These functions are documented on page 184.)

`\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.

```

8918 \cs_new_protected_nopar:Npn \fp_round_places:Nn
8919   { \fp_round_places_aux:NNn \tl_set:Nn }
8920 \cs_generate_variant:Nn \fp_round_places:Nn { c }
8921 \cs_new_protected_nopar:Npn \fp_ground_places:Nn
8922   { \fp_round_places_aux:NNn \tl_gset:Nn }
8923 \cs_generate_variant:Nn \fp_ground_places:Nn { c }
8924 \cs_new_protected_nopar:Npn \fp_round_places_aux:NNn #1#2#3
8925   {
8926     \group_begin:
8927     \fp_read:N #2
8928     \int_set:Nn \l_fp_round_target_int
8929       { #3 + \l_fp_input_a_exponent_int }
8930     \if_int_compare:w \l_fp_round_target_int < \c_ten
8931       \exp_after:wN \fp_round:
8932     \fi:
8933     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8934     \cs_set_protected_nopar:Npx \fp_tmp:w
8935     {
8936       \group_end:
8937       #1 \exp_not:N #2
8938       {
8939         \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
8940           -
8941         \else:
8942           +
8943         \fi:
8944         \int_use:N \l_fp_input_a_integer_int
8945         .
8946         \exp_after:wN \use_none:n
8947         \int_use:N \l_fp_input_a_decimal_int
8948         e
8949         \int_use:N \l_fp_input_a_exponent_int
8950       }
8951     }
8952     \fp_tmp:w
8953   }

```

(End definition for `\fp_round_places:Nn` and `\fp_round_places:cn`. These functions are documented on page 184.)

`\fp_round:`
`\fp_round_aux:NNNNNNNN`
`\fp_round_loop:N`

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.

```

8954 \cs_new_protected_nopar:Npn \fp_round:
8955   {

```

```

8956 \bool_set_false:N \l_fp_round_carry_bool
8957 \l_fp_round_position_int \c_eight
8958 \tl_clear:N \l_fp_round_decimal_tl
8959 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8960 \exp_after:wN \use_i:nn \exp_after:wN
8961 \fp_round_aux:NNNNNNNN \int_use:N \l_fp_input_a_decimal_int
8962 }
8963 \cs_new_protected_nopar:Npn \fp_round_aux:NNNNNNNN #1#2#3#4#5#6#7#8#9
8964 {
8965 \fp_round_loop:N #9#8#7#6#5#4#3#2#1
8966 \bool_if:NT \l_fp_round_carry_bool
8967 { \tex_advance:D \l_fp_input_a_integer_int \c_one }
8968 \l_fp_input_a_decimal_int \l_fp_round_decimal_tl \scan_stop:
8969 \if_int_compare:w \l_fp_input_a_integer_int < \c_ten
8970 \else:
8971 \l_fp_input_a_integer_int \c_one
8972 \tex_divide:D \l_fp_input_a_decimal_int \c_ten
8973 \tex_advance:D \l_fp_input_a_exponent_int \c_one
8974 \fi:
8975 }
8976 \cs_new_protected_nopar:Npn \fp_round_loop:N #1
8977 {
8978 \if_int_compare:w \l_fp_round_position_int < \l_fp_round_target_int
8979 \bool_if:NTF \l_fp_round_carry_bool
8980 { \l_fp_tmp_int \int_eval:w #1 + \c_one \scan_stop: }
8981 { \l_fp_tmp_int \int_eval:w #1 \scan_stop: }
8982 \if_int_compare:w \l_fp_tmp_int = \c_ten
8983 \l_fp_tmp_int \c_zero
8984 \else:
8985 \bool_set_false:N \l_fp_round_carry_bool
8986 \fi:
8987 \tl_set:Nx \l_fp_round_decimal_tl
8988 { \int_use:N \l_fp_tmp_int \l_fp_round_decimal_tl }
8989 \else:
8990 \tl_set:Nx \l_fp_round_decimal_tl { 0 \l_fp_round_decimal_tl }
8991 \if_int_compare:w \l_fp_round_position_int = \l_fp_round_target_int
8992 \if_int_compare:w #1 > \c_four
8993 \bool_set_true:N \l_fp_round_carry_bool
8994 \fi:
8995 \fi:
8996 \fi:
8997 \tex_advance:D \l_fp_round_position_int \c_minus_one
8998 \if_int_compare:w \l_fp_round_position_int > \c_minus_one
8999 \exp_after:wN \fp_round_loop:N
9000 \fi:
9001 }

```

(End definition for `\fp_round`:. This function is documented on page ??.)

187.8 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`

```

9002 \cs_new_protected_nopar:Npn \fp_abs:N { \fp_abs_aux:NN \tl_set:Nn }
9003 \cs_new_protected_nopar:Npn \fp_gabs:N { \fp_abs_aux:NN \tl_gset:Nn }
9004 \cs_generate_variant:Nn \fp_abs:N { c }
9005 \cs_generate_variant:Nn \fp_gabs:N { c }
9006 \cs_new_protected_nopar:Npn \fp_abs_aux:NN #1#2
9007 {
9008   \group_begin:
9009   \fp_read:N #2
9010   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9011   \cs_set_protected_nopar:Npx \fp_tmp:w
9012   {
9013     \group_end:
9014     #1 \exp_not:N #2
9015     {
9016       +
9017       \int_use:N \l_fp_input_a_integer_int
9018       .
9019       \exp_after:wN \use_none:n
9020       \int_use:N \l_fp_input_a_decimal_int
9021       e
9022       \int_use:N \l_fp_input_a_exponent_int
9023     }
9024   }
9025   \fp_tmp:w
9026 }

```

(End definition for `\fp_abs:N` and `\fp_abs:c`. These functions are documented on page 186.)

`\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`

```

9027 \cs_new_protected_nopar:Npn \fp_neg:N { \fp_neg_aux:NN \tl_set:Nn }
9028 \cs_new_protected_nopar:Npn \fp_gneg:N { \fp_neg_aux:NN \tl_gset:Nn }
9029 \cs_generate_variant:Nn \fp_neg:N { c }
9030 \cs_generate_variant:Nn \fp_gneg:N { c }
9031 \cs_new_protected_nopar:Npn \fp_neg_aux:NN #1#2
9032 {
9033   \group_begin:
9034   \fp_read:N #2
9035   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9036   \tl_set:Nx \l_fp_tmp_tl
9037   {
9038     \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9039     +
9040     \else:
9041     -
9042     \fi:

```

```

9043         \int_use:N \l_fp_input_a_integer_int
9044         .
9045         \exp_after:wN \use_none:n
9046         \int_use:N \l_fp_input_a_decimal_int
9047         e
9048         \int_use:N \l_fp_input_a_exponent_int
9049     }
9050     \exp_after:wN \group_end: \exp_after:wN
9051     #1 \exp_after:wN #2 \exp_after:wN { \l_fp_tmp_tl }
9052 }

```

(End definition for `\fp_neg:N` and `\fp_neg:c`. These functions are documented on page 186.)

187.9 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.

```

9053 \cs_new_protected_nopar:Npn \fp_add:Nn { \fp_add_aux:NNn \tl_set:Nn }
9054 \cs_new_protected_nopar:Npn \fp_gadd:Nn { \fp_add_aux:NNn \tl_gset:Nn }
9055 \cs_generate_variant:Nn \fp_add:Nn { c }
9056 \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.

```

9057 \cs_new_protected_nopar:Npn \fp_add_aux:NNn #1#2#3
9058 {
9059     \group_begin:
9060     \fp_read:N #2
9061     \fp_split:Nn b {#3}
9062     \fp_standardise:NNNN
9063     \l_fp_input_b_sign_int
9064     \l_fp_input_b_integer_int
9065     \l_fp_input_b_decimal_int
9066     \l_fp_input_b_exponent_int
9067     \fp_add_core:
9068     \fp_tmp:w #1#2
9069 }
9070 \cs_new_protected_nopar:Npn \fp_add_core:
9071 {
9072     \fp_level_input_exponents:
9073     \if_int_compare:w
9074     \int_eval:w
9075     \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
9076     > \c_zero
9077     \exp_after:wN \fp_add_sum:
9078     \else:

```

```

9079     \exp_after:wN \fp_add_difference:
9080 \fi:
9081 \l_fp_output_exponent_int \l_fp_input_a_exponent_int
9082 \fp_standardise:NNNN
9083   \l_fp_output_sign_int
9084   \l_fp_output_integer_int
9085   \l_fp_output_decimal_int
9086   \l_fp_output_exponent_int
9087 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9088 {
9089   \group_end:
9090   ##1 ##2
9091   {
9092     \if_int_compare:w \l_fp_output_sign_int < \c_zero
9093     -
9094     \else:
9095     +
9096     \fi:
9097     \int_use:N \l_fp_output_integer_int
9098     .
9099     \exp_after:wN \use_none:n
9100     \int_value:w \int_eval:w
9101       \l_fp_output_decimal_int + \c_one_thousand_million
9102     e
9103     \int_use:N \l_fp_output_exponent_int
9104   }
9105 }
9106 }

```

Finding the sum of two numbers is trivially easy.

```

9107 \cs_new_protected_nopar:Npn \fp_add_sum:
9108 {
9109   \l_fp_output_sign_int \l_fp_input_a_sign_int
9110   \l_fp_output_integer_int
9111   \int_eval:w
9112     \l_fp_input_a_integer_int + \l_fp_input_b_integer_int
9113   \scan_stop:
9114   \l_fp_output_decimal_int
9115   \int_eval:w
9116     \l_fp_input_a_decimal_int + \l_fp_input_b_decimal_int
9117   \scan_stop:
9118   \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
9119   \else:
9120     \tex_advance:D \l_fp_output_integer_int \c_one
9121     \tex_advance:D \l_fp_output_decimal_int -\c_one_thousand_million
9122   \fi:
9123 }

```

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.

```

9124 \cs_new_protected_nopar:Npn \fp_add_difference:
9125 {
9126   \l_fp_output_integer_int
9127   \int_eval:w
9128     \l_fp_input_a_integer_int - \l_fp_input_b_integer_int
9129   \scan_stop:
9130   \l_fp_output_decimal_int
9131   \int_eval:w
9132     \l_fp_input_a_decimal_int - \l_fp_input_b_decimal_int
9133   \scan_stop:
9134   \if_int_compare:w \l_fp_output_decimal_int < \c_zero
9135     \tex_advance:D \l_fp_output_integer_int \c_minus_one
9136     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9137   \fi:
9138   \if_int_compare:w \l_fp_output_integer_int < \c_zero
9139     \l_fp_output_sign_int \l_fp_input_b_sign_int
9140     \if_int_compare:w \l_fp_output_decimal_int = \c_zero
9141       \l_fp_output_integer_int -\l_fp_output_integer_int
9142     \else:
9143       \l_fp_output_decimal_int
9144       \int_eval:w
9145         \c_one_thousand_million - \l_fp_output_decimal_int
9146       \scan_stop:
9147       \l_fp_output_integer_int
9148       \int_eval:w
9149         - \l_fp_output_integer_int - \c_one
9150       \scan_stop:
9151   \fi:
9152   \else:
9153     \l_fp_output_sign_int \l_fp_input_a_sign_int
9154   \fi:
9155 }

```

(End definition for `\fp_add:Nn` and `\fp_add:cn`. These functions are documented on page 186.)

`\fp_sub:Nn` Subtraction is essentially the same as addition, but with the sign of the second component
`\fp_sub:cn` reversed. Thus the core of the two function groups is the same, with just a little set up
`\fp_gsub:Nn` here.
`\fp_gsub:cn`

```

\fp_sub_aux:NNn 9156 \cs_new_protected_nopar:Npn \fp_sub:Nn { \fp_sub_aux:NNn \tl_set:Nn }
9157 \cs_new_protected_nopar:Npn \fp_gsub:Nn { \fp_sub_aux:NNn \tl_gset:Nn }
9158 \cs_generate_variant:Nn \fp_sub:Nn { c }
9159 \cs_generate_variant:Nn \fp_gsub:Nn { c }
9160 \cs_new_protected_nopar:Npn \fp_sub_aux:NNn #1#2#3
9161 {
9162   \group_begin:
9163     \fp_read:N #2

```

```

9164 \fp_split:Nn b {#3}
9165 \fp_standardise:NNNN
9166 \l_fp_input_b_sign_int
9167 \l_fp_input_b_integer_int
9168 \l_fp_input_b_decimal_int
9169 \l_fp_input_b_exponent_int
9170 \tex_multiply:D \l_fp_input_b_sign_int \c_minus_one
9171 \fp_add_core:
9172 \fp_tmp:w #1#2
9173 }

```

(End definition for `\fp_sub:Nn` and `\fp_sub:cn`. These functions are documented on page 187.)

`\fp_mul:Nn`

The pattern is much the same for multiplication.

`\fp_mul:cn`

`\fp_gmul:Nn`

`\fp_gmul:cn`

`\fp_mul_aux:NNn`

`\fp_mul_internal:`

`\fp_mul_split:NNNN`

`\fp_mul_split:w`

`\fp_mul_end_level:`

`\fp_mul_end_level:NNNNNNNN`

```

9174 \cs_new_protected_nopar:Npn \fp_mul:Nn { \fp_mul_aux:NNn \tl_set:Nn }
9175 \cs_new_protected_nopar:Npn \fp_gmul:Nn { \fp_mul_aux:NNn \tl_gset:Nn }
9176 \cs_generate_variant:Nn \fp_mul:Nn { c }
9177 \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

```

9178 \cs_new_protected_nopar:Npn \fp_mul_aux:NNn #1#2#3
9179 {
9180   \group_begin:
9181   \fp_read:N #2
9182   \fp_split:Nn b {#3}
9183   \fp_standardise:NNNN
9184   \l_fp_input_b_sign_int
9185   \l_fp_input_b_integer_int
9186   \l_fp_input_b_decimal_int
9187   \l_fp_input_b_exponent_int
9188   \fp_mul_internal:
9189   \l_fp_output_exponent_int
9190   \int_eval:w
9191     \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
9192   \scan_stop:
9193   \fp_standardise:NNNN
9194   \l_fp_output_sign_int
9195   \l_fp_output_integer_int
9196   \l_fp_output_decimal_int
9197   \l_fp_output_exponent_int
9198   \cs_set_protected_nopar:Npx \fp_tmp:w
9199   {
9200     \group_end:
9201     #1 \exp_not:N #2
9202     {

```

```

9203         \if_int_compare:w
9204         \int_eval:w
9205         \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
9206         < \c_zero
9207         \if_int_compare:w
9208         \int_eval:w
9209         \l_fp_output_integer_int + \l_fp_output_decimal_int
9210         = \c_zero
9211         +
9212         \else:
9213         -
9214         \fi:
9215         \else:
9216         +
9217         \fi:
9218         \int_use:N \l_fp_output_integer_int
9219         .
9220         \exp_after:wN \use_none:n
9221         \int_value:w \int_eval:w
9222         \l_fp_output_decimal_int + \c_one_thousand_million
9223         e
9224         \int_use:N \l_fp_output_exponent_int
9225     }
9226 }
9227 \fp_tmp:w
9228 }

```

Done separately so that the internal use is a bit easier.

```

9229 \cs_new_protected_nopar:Npn \fp_mul_internal:
9230 {
9231     \fp_mul_split:NNNN \l_fp_input_a_decimal_int
9232     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
9233     \fp_mul_split:NNNN \l_fp_input_b_decimal_int
9234     \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
9235     \l_fp_mul_output_int \c_zero
9236     \tl_clear:N \l_fp_mul_output_tl
9237     \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
9238     \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
9239     \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
9240     \tex_divide:D \l_fp_mul_output_int \c_one_thousand
9241     \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_iii_int
9242     \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
9243     \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
9244     \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_input_b_integer_int
9245     \fp_mul_end_level:
9246     \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_ii_int
9247     \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
9248     \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_input_b_integer_int
9249     \fp_mul_end_level:

```

```

9250 \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_i_int
9251 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_input_b_integer_int
9252 \fp_mul_end_level:
9253 \l_fp_output_decimal_int 0 \l_fp_mul_output_tl \scan_stop:
9254 \tl_clear:N \l_fp_mul_output_tl
9255 \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_input_b_integer_int
9256 \fp_mul_end_level:
9257 \l_fp_output_integer_int 0 \l_fp_mul_output_tl \scan_stop:
9258 }

```

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.

```

9259 \cs_new_protected_nopar:Npn \fp_mul_split:NNNN #1#2#3#4
9260 {
9261   \tex_advance:D #1 \c_one_thousand_million
9262   \cs_set_protected_nopar:Npn \fp_mul_split_aux:w
9263     ##1##2##3##4##5##6##7##8##9 \q_stop {
9264     #2 ##2##3##4 \scan_stop:
9265     #3 ##5##6##7 \scan_stop:
9266     #4 ##8##9 \scan_stop:
9267   }
9268   \exp_after:wN \fp_mul_split_aux:w \int_use:N #1 \q_stop
9269   \tex_advance:D #1 -\c_one_thousand_million
9270 }
9271 \cs_new_protected_nopar:Npn \fp_mul_product:NN #1#2
9272 {
9273   \l_fp_mul_output_int
9274   \int_eval:w \l_fp_mul_output_int + #1 * #2 \scan_stop:
9275 }

```

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.

```

9276 \cs_new_protected_nopar:Npn \fp_mul_end_level:
9277 {
9278   \tex_advance:D \l_fp_mul_output_int \c_one_thousand_million
9279   \exp_after:wN \use_i:nn \exp_after:wN
9280   \fp_mul_end_level:NNNNNNNN \int_use:N \l_fp_mul_output_int
9281 }
9282 \cs_new_protected_nopar:Npn \fp_mul_end_level:NNNNNNNN #1#2#3#4#5#6#7#8#9
9283 {
9284   \tl_set:Nx \l_fp_mul_output_tl { #7#8#9 \l_fp_mul_output_tl }
9285   \l_fp_mul_output_int #1#2#3#4#5#6 \scan_stop:
9286 }

```

(End definition for `\fp_mul:Nn` and `\fp_mul:cn`. These functions are documented on page 187.)

<pre> \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: </pre>	<p>The pattern is much the same for multiplication.</p> <pre> 9287 \cs_new_protected_nopar:Npn \fp_div:Nn { \fp_div_aux:NNn \tl_set:Nn } 9288 \cs_new_protected_nopar:Npn \fp_gdiv:Nn { \fp_div_aux:NNn \tl_gset:Nn } 9289 \cs_generate_variant:Nn \fp_div:Nn { c } 9290 \cs_generate_variant:Nn \fp_gdiv:Nn { c } </pre> <p>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.</p> <pre> 9291 \cs_new_protected_nopar:Npn \fp_div_aux:NNn #1#2#3 9292 { 9293 \group_begin: 9294 \fp_read:N #2 9295 \fp_split:Nn b {#3} 9296 \fp_standardise:NNNN 9297 \l_fp_input_b_sign_int 9298 \l_fp_input_b_integer_int 9299 \l_fp_input_b_decimal_int 9300 \l_fp_input_b_exponent_int 9301 \if_int_compare:w 9302 \int_eval:w 9303 \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int 9304 = \c_zero 9305 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2 9306 { 9307 \group_end: 9308 #1 \exp_not:N #2 { \c_undefined_fp } 9309 } 9310 \else: 9311 \if_int_compare:w 9312 \int_eval:w 9313 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int 9314 = \c_zero 9315 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2 9316 { 9317 \group_end: 9318 #1 \exp_not:N #2 { \c_zero_fp } 9319 } 9320 \else: 9321 \exp_after:wN \exp_after:wN \exp_after:wN \fp_div_internal: 9322 \fi: 9323 \fi: 9324 \fp_tmp:w #1#2 9325 } </pre>
---	---

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.

```

9326 \cs_new_protected_nopar:Npn \fp_div_internal: {
9327   \l_fp_output_integer_int \c_zero
9328   \l_fp_output_decimal_int \c_zero
9329   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
9330   \l_fp_div_offset_int \c_one_hundred_million
9331   \fp_div_loop:
9332   \l_fp_output_exponent_int
9333   \int_eval:w
9334     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
9335   \scan_stop:
9336   \fp_standardise:NNNN
9337   \l_fp_output_sign_int
9338   \l_fp_output_integer_int
9339   \l_fp_output_decimal_int
9340   \l_fp_output_exponent_int
9341   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9342   {
9343     \group_end:
9344     ##1 ##2
9345     {
9346       \if_int_compare:w
9347         \int_eval:w
9348           \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
9349         < \c_zero
9350       \if_int_compare:w
9351         \int_eval:w
9352           \l_fp_output_integer_int + \l_fp_output_decimal_int
9353         = \c_zero
9354       +
9355       \else:
9356       -
9357       \fi:
9358     \else:
9359     +
9360     \fi:
9361     \int_use:N \l_fp_output_integer_int
9362     .
9363     \exp_after:wN \use_none:n
9364     \int_value:w \int_eval:w
9365       \l_fp_output_decimal_int + \c_one_thousand_million
9366     \int_eval_end:
9367     e
9368     \int_use:N \l_fp_output_exponent_int
9369   }
9370 }
9371 }
```

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.

```

9372 \cs_new_protected_nopar:Npn \fp_div_loop:
9373 {
9374   \l_fp_count_int \c_zero
9375   \fp_div_divide:
9376   \fp_div_store:
9377   \tex_multiply:D \l_fp_input_a_integer_int \c_ten
9378   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9379   \exp_after:wN \fp_div_loop_step:w
9380   \int_use:N \l_fp_input_a_decimal_int \q_stop
9381   \if_int_compare:w
9382     \int_eval:w \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
9383     > \c_zero
9384     \if_int_compare:w \l_fp_div_offset_int > \c_zero
9385       \exp_after:wN \exp_after:wN \exp_after:wN
9386       \fp_div_loop:
9387     \fi:
9388   \fi:
9389 }

```

Checking to see if the numerator can be divided needs quite an involved check. Either the integer part has to be bigger for the numerator or, if it is not smaller then the decimal part of the numerator must not be smaller than that of the denominator. Once the test is right the rest is much as elsewhere.

```

9390 \cs_new_protected_nopar:Npn \fp_div_divide:
9391 {
9392   \if_int_compare:w \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
9393     \exp_after:wN \fp_div_divide_aux:
9394   \else:
9395     \if_int_compare:w \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
9396     \else:
9397       \if_int_compare:w
9398         \l_fp_input_a_decimal_int < \l_fp_input_b_decimal_int
9399       \else:
9400         \exp_after:wN \exp_after:wN \exp_after:wN
9401         \exp_after:wN \exp_after:wN \exp_after:wN
9402         \exp_after:wN \fp_div_divide_aux:
9403       \fi:
9404     \fi:
9405   \fi:
9406 }
9407 \cs_new_protected_nopar:Npn \fp_div_divide_aux:
9408 {
9409   \tex_advance:D \l_fp_count_int \c_one
9410   \tex_advance:D \l_fp_input_a_integer_int -\l_fp_input_b_integer_int
9411   \tex_advance:D \l_fp_input_a_decimal_int -\l_fp_input_b_decimal_int
9412   \if_int_compare:w \l_fp_input_a_decimal_int < \c_zero

```

```

9413      \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
9414      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9415      \fi:
9416      \fp_div_divide:
9417  }

```

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.

```

9418  \cs_new_protected_nopar:Npn \fp_div_store: { }
9419  \cs_new_protected_nopar:Npn \fp_div_store_integer:
9420  {
9421      \l_fp_output_integer_int \l_fp_count_int
9422      \cs_set_eq:NN \fp_div_store: \fp_div_store_decimal:
9423  }
9424  \cs_new_protected_nopar:Npn \fp_div_store_decimal:
9425  {
9426      \l_fp_output_decimal_int
9427      \int_eval:w
9428          \l_fp_output_decimal_int +
9429          \l_fp_count_int * \l_fp_div_offset_int
9430      \int_eval_end:
9431      \tex_divide:D \l_fp_div_offset_int \c_ten
9432  }
9433  \cs_new_protected_nopar:Npn \fp_div_loop_step:w #1#2#3#4#5#6#7#8#9 \q_stop
9434  {
9435      \l_fp_input_a_integer_int
9436      \int_eval:w #2 + \l_fp_input_a_integer_int \int_eval_end:
9437      \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
9438  }

```

(End definition for `\fp_div:Nn` and `\fp_div:cn`. These functions are documented on page ??.)

187.10 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 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.

```

9439 \cs_new_protected_nopar:Npn \fp_add:NNNNNNNNN #1#2#3#4#5#6#7#8#9
9440 {
9441   #7 \int_eval:w #1 + #4 \int_eval_end:
9442   #8 \int_eval:w #2 + #5 \int_eval_end:
9443   #9 \int_eval:w #3 + #6 \int_eval_end:
9444   \if_int_compare:w #9 < \c_one_thousand_million
9445   \else:
9446     \tex_advance:D #8 \c_one
9447     \tex_advance:D #9 -\c_one_thousand_million
9448   \fi:
9449   \if_int_compare:w #8 < \c_one_thousand_million
9450   \else:
9451     \tex_advance:D #7 \c_one
9452     \tex_advance:D #8 -\c_one_thousand_million
9453   \fi:
9454 }
```

(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.

```

9455 \cs_new_protected_nopar:Npn \fp_sub:NNNNNNNNN #1#2#3#4#5#6#7#8#9
9456 {
9457   #7 \int_eval:w #1 - #4 \int_eval_end:
9458   #8 \int_eval:w #2 - #5 \int_eval_end:
9459   #9 \int_eval:w #3 - #6 \int_eval_end:
9460   \if_int_compare:w #9 < \c_zero
9461     \tex_advance:D #8 \c_minus_one
9462     \tex_advance:D #9 \c_one_thousand_million
9463   \fi:
9464   \if_int_compare:w #8 < \c_zero
9465     \tex_advance:D #7 \c_minus_one
9466     \tex_advance:D #8 \c_one_thousand_million
```

```

9467 \fi:
9468 \if_int_compare:w #7 < \c_zero
9469 \if_int_compare:w \int_eval:w #8 + #9 = \c_zero
9470 #7 -#7
9471 \else:
9472 \tex_advance:D #7 \c_one
9473 #8 \int_eval:w \c_one_thousand_million - #8 \int_eval_end:
9474 #9 \int_eval:w \c_one_thousand_million - #9 \int_eval_end:
9475 \fi:
9476 \fi:
9477 }

```

(End definition for `\fp_sub:NNNNNNNN`. This function is documented on page ??.)

`\fp_mul:NNNNNN` Decimal-part only multiplication but with higher accuracy than the user version.

```

9478 \cs_new_protected_nopar:Npn \fp_mul:NNNNNN #1#2#3#4#5#6
9479 {
9480 \fp_mul_split:NNNN #1
9481 \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
9482 \fp_mul_split:NNNN #2
9483 \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
9484 \fp_mul_split:NNNN #3
9485 \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
9486 \fp_mul_split:NNNN #4
9487 \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
9488 \l_fp_mul_output_int \c_zero
9489 \tl_clear:N \l_fp_mul_output_tl
9490 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
9491 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
9492 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
9493 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
9494 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
9495 \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
9496 \tex_divide:D \l_fp_mul_output_int \c_one_thousand
9497 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
9498 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
9499 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
9500 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
9501 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
9502 \fp_mul_end_level:
9503 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
9504 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
9505 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
9506 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
9507 \fp_mul_end_level:
9508 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
9509 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
9510 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
9511 \fp_mul_end_level:

```

```

9512 #6 0 \l_fp_mul_output_tl \scan_stop:
9513 \tl_clear:N \l_fp_mul_output_tl
9514 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
9515 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
9516 \fp_mul_end_level:
9517 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
9518 \fp_mul_end_level:
9519 \fp_mul_end_level:
9520 #5 0 \l_fp_mul_output_tl \scan_stop:
9521 }

```

(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.

```

9522 \cs_new_protected_nopar:Npn \fp_mul:NNNNNNNN #1#2#3#4#5#6#7#8#9
9523 {
9524   \fp_mul_split:NNNN #2
9525   \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
9526   \fp_mul_split:NNNN #3
9527   \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
9528   \fp_mul_split:NNNN #5
9529   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
9530   \fp_mul_split:NNNN #6
9531   \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
9532   \l_fp_mul_output_int \c_zero
9533   \tl_clear:N \l_fp_mul_output_tl
9534   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
9535   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
9536   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
9537   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
9538   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
9539   \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
9540   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
9541   \fp_mul_product:NN #1 \l_fp_mul_b_vi_int
9542   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
9543   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
9544   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
9545   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
9546   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
9547   \fp_mul_product:NN \l_fp_mul_a_vi_int #4
9548   \fp_mul_end_level:
9549   \fp_mul_product:NN #1 \l_fp_mul_b_v_int
9550   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
9551   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
9552   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
9553   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
9554   \fp_mul_product:NN \l_fp_mul_a_v_int #4
9555   \fp_mul_end_level:

```

```

9556 \fp_mul_product:NN #1 \l_fp_mul_b_iv_int
9557 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
9558 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
9559 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
9560 \fp_mul_product:NN \l_fp_mul_a_iv_int #4
9561 \fp_mul_end_level:
9562 #9 0 \l_fp_mul_output_tl \scan_stop:
9563 \tl_clear:N \l_fp_mul_output_tl
9564 \fp_mul_product:NN #1 \l_fp_mul_b_iii_int
9565 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
9566 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
9567 \fp_mul_product:NN \l_fp_mul_a_iii_int #4
9568 \fp_mul_end_level:
9569 \fp_mul_product:NN #1 \l_fp_mul_b_ii_int
9570 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
9571 \fp_mul_product:NN \l_fp_mul_a_ii_int #4
9572 \fp_mul_end_level:
9573 \fp_mul_product:NN #1 \l_fp_mul_b_i_int
9574 \fp_mul_product:NN \l_fp_mul_a_i_int #4
9575 \fp_mul_end_level:
9576 #8 0 \l_fp_mul_output_tl \scan_stop:
9577 \tl_clear:N \l_fp_mul_output_tl
9578 \fp_mul_product:NN #1 #4
9579 \fp_mul_end_level:
9580 #7 0 \l_fp_mul_output_tl \scan_stop:
9581 }

```

(End definition for `\fp_mul:NNNNNNNN`. This function is documented on page ??.)

`\fp_div_integer:NNNN` 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.

```

9582 \cs_new_protected_nopar:Npn \fp_div_integer:NNNN #1#2#3#4#5
9583 {
9584   \l_fp_tmp_int #1
9585   \tex_divide:D \l_fp_tmp_int #3
9586   \l_fp_tmp_int \int_eval:w #1 - \l_fp_tmp_int * #3 \int_eval_end:
9587   #4 #1
9588   \tex_divide:D #4 #3
9589   #5 #2
9590   \tex_divide:D #5 #3
9591   \tex_multiply:D \l_fp_tmp_int \c_one_thousand
9592   \tex_divide:D \l_fp_tmp_int #3
9593   #5 \int_eval:w #5 + \l_fp_tmp_int * \c_one_million \int_eval_end:
9594   \if_int_compare:w #5 > \c_one_thousand_million
9595     \tex_advance:D #4 \c_one
9596     \tex_advance:D #5 -\c_one_thousand_million
9597   \fi:

```

9598 }

(End definition for `\fp_div_integer:NNNNN`. This function is documented on page ??.)

`\fp_extended_normalise:` The “extended” integers for internal use are mainly used in fixed-point mode. This comes up in a few places, so a generalised utility is made available to carry out the change. This function simply calls the two loops to shift the input to the point of having a zero exponent.

```

\fp_extended_normalise_aux_i:
\fp_extended_normalise_aux_i:w
\fp_extended_normalise_aux_ii:w
\fp_extended_normalise_aux_ii:
\fp_extended_normalise_aux:NNNNNNNNN
9599 \cs_new_protected_nopar:Npn \fp_extended_normalise:
9600 {
9601   \fp_extended_normalise_aux_i:
9602   \fp_extended_normalise_aux_ii:
9603 }
9604 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i:
9605 {
9606   \if_int_compare:w \l_fp_input_a_exponent_int > \c_zero
9607     \tex_multiply:D \l_fp_input_a_integer_int \c_ten
9608     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9609     \exp_after:wN \fp_extended_normalise_aux_i:w
9610     \int_use:N \l_fp_input_a_decimal_int \q_stop
9611     \exp_after:wN \fp_extended_normalise_aux_i:
9612   \fi:
9613 }
9614 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii:w
9615   #1#2#3#4#5#6#7#8#9 \q_stop
9616 {
9617   \l_fp_input_a_integer_int
9618   \int_eval:w \l_fp_input_a_integer_int + #2 \scan_stop:
9619   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
9620   \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9621   \exp_after:wN \fp_extended_normalise_aux_ii:w
9622   \int_use:N \l_fp_input_a_extended_int \q_stop
9623 }
9624 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii:w
9625   #1#2#3#4#5#6#7#8#9 \q_stop
9626 {
9627   \l_fp_input_a_decimal_int
9628   \int_eval:w \l_fp_input_a_decimal_int + #2 \scan_stop:
9629   \l_fp_input_a_extended_int #3#4#5#6#7#8#9 0 \scan_stop:
9630   \tex_advance:D \l_fp_input_a_exponent_int \c_minus_one
9631 }
9632 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii:
9633 {
9634   \if_int_compare:w \l_fp_input_a_exponent_int < \c_zero
9635     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9636     \exp_after:wN \use_i:nn \exp_after:wN
9637     \fp_extended_normalise_ii_aux:NNNNNNNNN
9638     \int_use:N \l_fp_input_a_decimal_int
9639     \exp_after:wN \fp_extended_normalise_aux_ii:

```

```

9640     \fi:
9641   }
9642   \cs_new_protected_nopar:Npn \fp_extended_normalise_ii_aux:NNNNNNNNN
9643     #1#2#3#4#5#6#7#8#9
9644   {
9645     \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
9646       \l_fp_input_a_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9647     \else:
9648       \tl_set:Nx \l_fp_tmp_tl
9649       {
9650         \int_use:N \l_fp_input_a_integer_int
9651         #1#2#3#4#5#6#7#8
9652       }
9653       \l_fp_input_a_integer_int \c_zero
9654       \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
9655     \fi:
9656     \tex_divide:D \l_fp_input_a_extended_int \c_ten
9657     \tl_set:Nx \l_fp_tmp_tl
9658     {
9659       #9
9660       \int_use:N \l_fp_input_a_extended_int
9661     }
9662     \l_fp_input_a_extended_int \l_fp_tmp_tl \scan_stop:
9663     \tex_advance:D \l_fp_input_a_exponent_int \c_one
9664   }

```

(End definition for `\fp_extended_normalise:`. This function is documented on page ??.)

`\fp_extended_normalise_output:` At some stages in working out extended output, it is possible for the value to need shifting to keep the integer part in range. This only ever happens such that the integer needs to be made smaller.

`\fp_extended_normalise_output_aux_i:NNNNNNNNN`
`\fp_extended_normalise_output_aux_ii:NNNNNNNNN`
`\fp_extended_normalise_output_aux:N`

```

9665   \cs_new_protected_nopar:Npn \fp_extended_normalise_output:
9666   {
9667     \if_int_compare:w \l_fp_output_integer_int > \c_nine
9668       \tex_advance:D \l_fp_output_integer_int \c_one_thousand_million
9669       \exp_after:wN \use_i:nn \exp_after:wN
9670       \fp_extended_normalise_output_aux_i:NNNNNNNNN
9671       \int_use:N \l_fp_output_integer_int
9672       \exp_after:wN \fp_extended_normalise_output:
9673     \fi:
9674   }
9675   \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux_i:NNNNNNNNN
9676     #1#2#3#4#5#6#7#8#9
9677   {
9678     \l_fp_output_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
9679     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9680     \tl_set:Nx \l_fp_tmp_tl
9681     {
9682       #9

```

```

9683         \exp_after:wN \use_none:n
9684         \int_use:N \l_fp_output_decimal_int
9685     }
9686     \exp_after:wN \fp_extended_normalise_output_aux_ii:NNNNNNNNN
9687     \l_fp_tmp_tl
9688 }
9689 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux_ii:NNNNNNNNN
9690 #1#2#3#4#5#6#7#8#9
9691 {
9692     \l_fp_output_decimal_int #1#2#3#4#5#6#7#8#9 \scan_stop:
9693     \fp_extended_normalise_output_aux:N
9694 }
9695 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux:N #1
9696 {
9697     \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
9698     \tex_divide:D \l_fp_output_extended_int \c_ten
9699     \tl_set:Nx \l_fp_tmp_tl
9700     {
9701         #1
9702         \exp_after:wN \use_none:n
9703         \int_use:N \l_fp_output_extended_int
9704     }
9705     \l_fp_output_extended_int \l_fp_tmp_tl \scan_stop:
9706     \tex_advance:D \l_fp_output_exponent_int \c_one
9707 }

```

(End definition for `\fp_extended_normalise_output`:. This function is documented on page ??.)

187.11 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`

```

9708 \cs_new_protected_nopar:Npn \fp_trig_normalise:
9709 {
9710     \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
9711     \l_fp_input_a_extended_int \c_zero
9712     \fp_extended_normalise:
9713     \fp_trig_normalise_aux:
9714     \if_int_compare:w \l_fp_input_a_integer_int < \c_zero
9715     \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
9716     \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
9717     \fi:
9718     \exp_after:wN \fp_trig_octant:
9719 }
9720 \l_fp_input_a_sign_int \c_one
9721 \l_fp_output_integer_int \c_zero
9722 \l_fp_output_decimal_int \c_zero

```

```

9723     \l_fp_output_exponent_int \c_zero
9724     \exp_after:wN \fp_trig_overflow_msg:
9725     \fi:
9726   }
9727 \cs_new_protected_nopar:Npn \fp_trig_normalise_aux:
9728 {
9729   \if_int_compare:w \l_fp_input_a_integer_int > \c_three
9730     \fp_trig_sub:NNN
9731     \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
9732     \exp_after:wN \fp_trig_normalise_aux:
9733   \else:
9734     \if_int_compare:w \l_fp_input_a_integer_int > \c_two
9735       \if_int_compare:w \l_fp_input_a_decimal_int > \c_fp_pi_decimal_int
9736         \fp_trig_sub:NNN
9737         \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
9738         \exp_after:wN \exp_after:wN \exp_after:wN
9739         \exp_after:wN \exp_after:wN \exp_after:wN
9740         \exp_after:wN \fp_trig_normalise_aux:
9741       \fi:
9742     \fi:
9743   \fi:
9744 }

```

Here, there may be a sign change but there will never be any variation in the input. So a dedicated function can be used.

```

9745 \cs_new_protected_nopar:Npn \fp_trig_sub:NNN #1#2#3
9746 {
9747   \l_fp_input_a_integer_int
9748   \int_eval:w \l_fp_input_a_integer_int - #1 \int_eval_end:
9749   \l_fp_input_a_decimal_int
9750   \int_eval:w \l_fp_input_a_decimal_int - #2 \int_eval_end:
9751   \l_fp_input_a_extended_int
9752   \int_eval:w \l_fp_input_a_extended_int - #3 \int_eval_end:
9753   \if_int_compare:w \l_fp_input_a_extended_int < \c_zero
9754     \tex_advance:D \l_fp_input_a_decimal_int \c_minus_one
9755     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9756   \fi:
9757   \if_int_compare:w \l_fp_input_a_decimal_int < \c_zero
9758     \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
9759     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9760   \fi:
9761   \if_int_compare:w \l_fp_input_a_integer_int < \c_zero
9762     \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
9763   \if_int_compare:w
9764     \int_eval:w
9765     \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
9766     = \c_zero
9767     \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
9768   \else:

```

```

9769         \l_fp_input_a_integer_int
9770         \int_eval:w
9771         - \l_fp_input_a_integer_int - \c_one
9772         \int_eval_end:
9773         \l_fp_input_a_decimal_int
9774         \int_eval:w
9775         \c_one_thousand_million - \l_fp_input_a_decimal_int
9776         \int_eval_end:
9777         \l_fp_input_a_extended_int
9778         \int_eval:w
9779         \c_one_thousand_million - \l_fp_input_a_extended_int
9780         \int_eval_end:
9781     \fi:
9782 \fi:
9783 }

```

(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 $\pi/2$.

```

9784 \cs_new_protected_nopar:Npn \fp_trig_octant:
9785 {
9786     \l_fp_trig_octant_int \c_one
9787     \fp_trig_octant_aux:
9788     \if_int_compare:w \l_fp_input_a_decimal_int < \c_ten
9789         \l_fp_input_a_decimal_int \c_zero
9790         \l_fp_input_a_extended_int \c_zero
9791     \fi:
9792     \if_int_odd:w \l_fp_trig_octant_int
9793     \else:
9794         \fp_sub:NNNNNNNNN
9795         \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
9796         \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9797         \l_fp_input_a_extended_int
9798         \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9799         \l_fp_input_a_extended_int
9800     \fi:
9801 }
9802 \cs_new_protected_nopar:Npn \fp_trig_octant_aux:
9803 {
9804     \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
9805         \fp_sub:NNNNNNNNN
9806         \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9807         \l_fp_input_a_extended_int
9808         \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
9809         \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9810         \l_fp_input_a_extended_int

```

```

9811     \tex_advance:D \l_fp_trig_octant_int \c_one
9812     \exp_after:wN \fp_trig_octant_aux:
9813   \else:
9814     \if_int_compare:w
9815       \l_fp_input_a_decimal_int > \c_fp_pi_by_four_decimal_int
9816     \fp_sub:NNNNNNNNN
9817       \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9818       \l_fp_input_a_extended_int
9819       \c_zero \c_fp_pi_by_four_decimal_int
9820       \c_fp_pi_by_four_extended_int
9821       \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
9822       \l_fp_input_a_extended_int
9823     \tex_advance:D \l_fp_trig_octant_int \c_one
9824     \exp_after:wN \exp_after:wN \exp_after:wN
9825       \fp_trig_octant_aux:
9826   \fi:
9827 \fi:
9828 }

```

(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`

```

9829 \cs_new_protected_nopar:Npn \fp_sin:Nn { \fp_sin_aux:NNn \tl_set:Nn }
9830 \cs_new_protected_nopar:Npn \fp_gsin:Nn { \fp_sin_aux:NNn \tl_gset:Nn }
9831 \cs_generate_variant:Nn \fp_sin:Nn { c }
9832 \cs_generate_variant:Nn \fp_gsin:Nn { c }

```

`\fp_sin_aux:NNn`
`\fp_sin_aux_i:`
`\fp_sin_aux_ii:`

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} .

```

9833 \cs_new_protected_nopar:Npn \fp_sin_aux:NNn #1#2#3
9834 {
9835   \group_begin:
9836     \fp_split:Nn a {#3}
9837     \fp_standardise:NNNN
9838     \l_fp_input_a_sign_int
9839     \l_fp_input_a_integer_int
9840     \l_fp_input_a_decimal_int
9841     \l_fp_input_a_exponent_int
9842     \tl_set:Nx \l_fp_arg_tl
9843     {
9844       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9845       -
9846     \else:
9847       +
9848     \fi:
9849     \int_use:N \l_fp_input_a_integer_int

```

```

9850      .
9851      \exp_after:wN \use_none:n
9852      \int_value:w \int_eval:w
9853      \l_fp_input_a_decimal_int + \c_one_thousand_million
9854      e
9855      \int_use:N \l_fp_input_a_exponent_int
9856    }
9857    \if_int_compare:w \l_fp_input_a_exponent_int < -\c_five
9858      \cs_set_protected_nopar:Npx \fp_tmp:w
9859      {
9860        \group_end:
9861        #1 \exp_not:N #2 { \l_fp_arg_tl }
9862      }
9863    \else:
9864      \if_cs_exist:w
9865        c_fp_sin ( \l_fp_arg_tl ) _fp
9866      \cs_end:
9867    \else:
9868      \exp_after:wN \exp_after:wN \exp_after:wN
9869      \fp_sin_aux_i:
9870    \fi:
9871    \cs_set_protected_nopar:Npx \fp_tmp:w
9872    {
9873      \group_end:
9874      #1 \exp_not:N #2
9875      { \use:c { c_fp_sin ( \l_fp_arg_tl ) _fp } }
9876    }
9877    \fi:
9878    \fp_tmp:w
9879  }

```

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.

```

9880 \cs_new_protected_nopar:Npn \fp_sin_aux_i:
9881 {
9882   \fp_trig_normalise:
9883   \fp_sin_aux_ii:
9884   \if_int_compare:w \l_fp_output_integer_int = \c_one
9885     \l_fp_output_exponent_int \c_zero
9886   \else:
9887     \l_fp_output_integer_int \l_fp_output_decimal_int
9888     \l_fp_output_decimal_int \l_fp_output_extended_int
9889     \l_fp_output_exponent_int -\c_nine
9890   \fi:
9891   \fp_standardise:NNNN
9892   \l_fp_input_a_sign_int
9893   \l_fp_output_integer_int
9894   \l_fp_output_decimal_int

```

```

9895     \l_fp_output_exponent_int
9896     \tl_new:c { c_fp_sin ( \l_fp_arg_tl ) _fp }
9897     \tl_gset:cx { c_fp_sin ( \l_fp_arg_tl ) _fp }
9898     {
9899         \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9900         +
9901         \else:
9902         -
9903         \fi:
9904         \int_use:N \l_fp_output_integer_int
9905         .
9906         \exp_after:wN \use_none:n
9907         \int_value:w \int_eval:w
9908         \l_fp_output_decimal_int + \c_one_thousand_million
9909         e
9910         \int_use:N \l_fp_output_exponent_int
9911     }
9912 }
9913 \cs_new_protected_nopar:Npn \fp_sin_aux_ii:
9914 {
9915     \if_case:w \l_fp_trig_octant_int
9916     \or:
9917         \exp_after:wN \fp_trig_calc_sin:
9918     \or:
9919         \exp_after:wN \fp_trig_calc_cos:
9920     \or:
9921         \exp_after:wN \fp_trig_calc_cos:
9922     \or:
9923         \exp_after:wN \fp_trig_calc_sin:
9924     \fi:
9925 }

```

(End definition for `\fp_sin:Nn` and `\fp_sin:cn`. These functions are documented on page 188.)

```

\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:
9926 \cs_new_protected_nopar:Npn \fp_cos:Nn { \fp_cos_aux:NNn \tl_set:Nn }
9927 \cs_new_protected_nopar:Npn \fp_gcos:Nn { \fp_cos_aux:NNn \tl_gset:Nn }
9928 \cs_generate_variant:Nn \fp_cos:Nn { c }
9929 \cs_generate_variant:Nn \fp_gcos:Nn { c }
9930 \cs_new_protected_nopar:Npn \fp_cos_aux:NNn #1#2#3
9931 {
9932     \group_begin:
9933     \fp_split:Nn a {#3}
9934     \fp_standardise:NNNN
9935     \l_fp_input_a_sign_int
9936     \l_fp_input_a_integer_int
9937     \l_fp_input_a_decimal_int
9938     \l_fp_input_a_exponent_int
9939     \tl_set:Nx \l_fp_arg_tl

```

```

9940     {
9941         \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
9942         -
9943         \else:
9944         +
9945         \fi:
9946         \int_use:N \l_fp_input_a_integer_int
9947         .
9948         \exp_after:wN \use_none:n
9949         \int_value:w \int_eval:w
9950         \l_fp_input_a_decimal_int + \c_one_thousand_million
9951         e
9952         \int_use:N \l_fp_input_a_exponent_int
9953     }
9954     \if_cs_exist:w c_fp_cos ( \l_fp_arg_tl ) _fp \cs_end:
9955     \else:
9956         \exp_after:wN \fp_cos_aux_i:
9957     \fi:
9958     \cs_set_protected_nopar:Npx \fp_tmp:w
9959     {
9960         \group_end:
9961         #1 \exp_not:N #2
9962         { \use:c { c_fp_cos ( \l_fp_arg_tl ) _fp } }
9963     }
9964     \fp_tmp:w
9965 }

```

Almost the same as for sine: just a bit of correction for the sign of the output.

```

9966 \cs_new_protected_nopar:Npn \fp_cos_aux_i:
9967 {
9968     \fp_trig_normalise:
9969     \fp_cos_aux_ii:
9970     \if_int_compare:w \l_fp_output_integer_int = \c_one
9971     \l_fp_output_exponent_int \c_zero
9972     \else:
9973         \l_fp_output_integer_int \l_fp_output_decimal_int
9974         \l_fp_output_decimal_int \l_fp_output_extended_int
9975         \l_fp_output_exponent_int -\c_nine
9976     \fi:
9977     \fp_standardise:NNNN
9978     \l_fp_input_a_sign_int
9979     \l_fp_output_integer_int
9980     \l_fp_output_decimal_int
9981     \l_fp_output_exponent_int
9982     \tl_new:c { c_fp_cos ( \l_fp_arg_tl ) _fp }
9983     \tl_gset:cx { c_fp_cos ( \l_fp_arg_tl ) _fp }
9984     {
9985         \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
9986         +

```

```

9987         \else:
9988             -
9989         \fi:
9990         \int_use:N \l_fp_output_integer_int
9991         .
9992         \exp_after:wN \use_none:n
9993         \int_value:w \int_eval:w
9994         \l_fp_output_decimal_int + \c_one_thousand_million
9995         e
9996         \int_use:N \l_fp_output_exponent_int
9997     }
9998 }
9999 \cs_new_protected_nopar:Npn \fp_cos_aux_ii:
10000 {
10001     \if_case:w \l_fp_trig_octant_int
10002     \or:
10003         \exp_after:wN \fp_trig_calc_cos:
10004     \or:
10005         \exp_after:wN \fp_trig_calc_sin:
10006     \or:
10007         \exp_after:wN \fp_trig_calc_sin:
10008     \or:
10009         \exp_after:wN \fp_trig_calc_cos:
10010     \fi:
10011     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10012         \if_int_compare:w \l_fp_trig_octant_int > \c_two
10013             \l_fp_input_a_sign_int \c_minus_one
10014         \fi:
10015     \else:
10016         \if_int_compare:w \l_fp_trig_octant_int > \c_two
10017         \else:
10018             \l_fp_input_a_sign_int \c_one
10019         \fi:
10020     \fi:
10021 }

```

(End definition for `\fp_cos:Nn` and `\fp_cos:cn`. These functions are documented on page 189.)

`\fp_trig_calc_cos:` These functions actually do the calculation for sine and cosine.
`\fp_trig_calc_sin:`
`\fp_trig_calc_Taylor:`

```

10022 \cs_new_protected_nopar:Npn \fp_trig_calc_cos:
10023 {
10024     \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
10025         \l_fp_output_integer_int \c_one
10026         \l_fp_output_decimal_int \c_zero
10027     \else:
10028         \l_fp_trig_sign_int \c_minus_one
10029         \fp_mul:NNNNNN
10030         \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10031         \l_fp_input_a_decimal_int \l_fp_input_a_extended_int

```

```

10032         \l_fp_trig_decimal_int \l_fp_trig_extended_int
10033     \fp_div_integer:NNNNN
10034         \l_fp_trig_decimal_int \l_fp_trig_extended_int
10035         \c_two
10036         \l_fp_trig_decimal_int \l_fp_trig_extended_int
10037     \l_fp_count_int \c_three
10038     \if_int_compare:w \l_fp_trig_extended_int = \c_zero
10039         \if_int_compare:w \l_fp_trig_decimal_int = \c_zero
10040             \l_fp_output_integer_int \c_one
10041             \l_fp_output_decimal_int \c_zero
10042             \l_fp_output_extended_int \c_zero
10043         \else:
10044             \l_fp_output_integer_int \c_zero
10045             \l_fp_output_decimal_int \c_one_thousand_million
10046             \l_fp_output_extended_int \c_zero
10047         \fi:
10048     \else:
10049         \l_fp_output_integer_int \c_zero
10050         \l_fp_output_decimal_int 999999999 \scan_stop:
10051         \l_fp_output_extended_int \c_one_thousand_million
10052     \fi:
10053     \tex_advance:D \l_fp_output_extended_int -\l_fp_trig_extended_int
10054     \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
10055     \exp_after:wN \fp_trig_calc_Taylor:
10056 \fi:
10057 }
10058 \cs_new_protected_nopar:Npn \fp_trig_calc_sin:
10059 {
10060     \l_fp_output_integer_int \c_zero
10061     \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
10062         \l_fp_output_decimal_int \c_zero
10063     \else:
10064         \l_fp_output_decimal_int \l_fp_input_a_decimal_int
10065         \l_fp_output_extended_int \l_fp_input_a_extended_int
10066         \l_fp_trig_sign_int \c_one
10067         \l_fp_trig_decimal_int \l_fp_input_a_decimal_int
10068         \l_fp_trig_extended_int \l_fp_input_a_extended_int
10069         \l_fp_count_int \c_two
10070         \exp_after:wN \fp_trig_calc_Taylor:
10071     \fi:
10072 }

```

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.

```

10073 \cs_new_protected_nopar:Npn \fp_trig_calc_Taylor:
10074 {
10075     \l_fp_trig_sign_int -\l_fp_trig_sign_int
10076     \fp_mul:NNNNNN
10077     \l_fp_trig_decimal_int \l_fp_trig_extended_int

```

```

10078     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10079     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10080 \fp_mul:NNNNNN
10081     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10082     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10083     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10084 \fp_div_integer:NNNNN
10085     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10086     \l_fp_count_int
10087     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10088 \tex_advance:D \l_fp_count_int \c_one
10089 \fp_div_integer:NNNNN
10090     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10091     \l_fp_count_int
10092     \l_fp_trig_decimal_int \l_fp_trig_extended_int
10093 \tex_advance:D \l_fp_count_int \c_one
10094 \if_int_compare:w \l_fp_trig_decimal_int > \c_zero
10095     \if_int_compare:w \l_fp_trig_sign_int > \c_zero
10096         \tex_advance:D \l_fp_output_decimal_int \l_fp_trig_decimal_int
10097         \tex_advance:D \l_fp_output_extended_int
10098             \l_fp_trig_extended_int
10099         \if_int_compare:w \l_fp_output_extended_int < \c_one_thousand_million
10100         \else:
10101             \tex_advance:D \l_fp_output_decimal_int \c_one
10102             \tex_advance:D \l_fp_output_extended_int
10103                 -\c_one_thousand_million
10104         \fi:
10105     \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
10106     \else:
10107         \tex_advance:D \l_fp_output_integer_int \c_one
10108         \tex_advance:D \l_fp_output_decimal_int
10109             -\c_one_thousand_million
10110     \fi:
10111 \else:
10112     \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
10113     \tex_advance:D \l_fp_output_extended_int
10114         -\l_fp_input_a_extended_int
10115     \if_int_compare:w \l_fp_output_extended_int < \c_zero
10116         \tex_advance:D \l_fp_output_decimal_int \c_minus_one
10117         \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
10118     \fi:
10119     \if_int_compare:w \l_fp_output_decimal_int < \c_zero
10120         \tex_advance:D \l_fp_output_integer_int \c_minus_one
10121         \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
10122     \fi:
10123     \fi:
10124 \exp_after:wN \fp_trig_calc_Taylor:
10125 \fi:
10126 }

```

(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:
\fp_tan_aux_ii:
\fp_tan_aux_iii:
\fp_tan_aux_iv:
10127 \cs_new_protected_nopar:Npn \fp_tan:Nn { \fp_tan_aux:NNn \tl_set:Nn }
10128 \cs_new_protected_nopar:Npn \fp_gtan:Nn { \fp_tan_aux:NNn \tl_gset:Nn }
10129 \cs_generate_variant:Nn \fp_tan:Nn { c }
10130 \cs_generate_variant:Nn \fp_gtan:Nn { c }
10131 \cs_new_protected_nopar:Npn \fp_tan_aux:NNn #1#2#3
10132 {
10133   \group_begin:
10134     \fp_split:Nn a {#3}
10135     \fp_standardise:NNNN
10136     \l_fp_input_a_sign_int
10137     \l_fp_input_a_integer_int
10138     \l_fp_input_a_decimal_int
10139     \l_fp_input_a_exponent_int
10140     \tl_set:Nx \l_fp_arg_tl
10141     {
10142       \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
10143       -
10144       \else:
10145       +
10146       \fi:
10147       \int_use:N \l_fp_input_a_integer_int
10148       .
10149       \exp_after:wN \use_none:n
10150       \int_value:w \int_eval:w
10151       \l_fp_input_a_decimal_int + \c_one_thousand_million
10152       e
10153       \int_use:N \l_fp_input_a_exponent_int
10154     }
10155     \if_int_compare:w \l_fp_input_a_exponent_int < -\c_five
10156     \cs_set_protected_nopar:Npx \fp_tmp:w
10157     {
10158       \group_end:
10159       #1 \exp_not:N #2 { \l_fp_arg_tl }
10160     }
10161   \else:
10162     \if_cs_exist:w
10163       c_fp_tan ( \l_fp_arg_tl ) _fp
10164     \cs_end:
10165   \else:
10166     \exp_after:wN \exp_after:wN \exp_after:wN
10167     \fp_tan_aux_i:
10168   \fi:
10169   \cs_set_protected_nopar:Npx \fp_tmp:w

```

```

10170      {
10171      \group_end:
10172      #1 \exp_not:N #2
10173      { \use:c { c_fp_tan ( \l_fp_arg_tl ) _fp } }
10174      }
10175      \fi:
10176      \fp_tmp:w
10177      }

```

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.

```

10178 \cs_new_protected_nopar:Npn \fp_tan_aux_i:
10179 {
10180   \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
10181   \exp_after:wN \fp_tan_aux_ii:
10182   \else:
10183     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
10184     \c_zero_fp
10185     \exp_after:wN \fp_trig_overflow_msg:
10186     \fi:
10187   }
10188 \cs_new_protected_nopar:Npn \fp_tan_aux_ii:
10189 {
10190   \fp_trig_normalise:
10191   \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10192   \if_int_compare:w \l_fp_trig_octant_int > \c_two
10193     \l_fp_output_sign_int \c_minus_one
10194   \else:
10195     \l_fp_output_sign_int \c_one
10196   \fi:
10197   \else:
10198     \if_int_compare:w \l_fp_trig_octant_int > \c_two
10199     \l_fp_output_sign_int \c_one
10200   \else:
10201     \l_fp_output_sign_int \c_minus_one
10202   \fi:
10203   \fi:
10204   \fp_cos_aux_ii:
10205   \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
10206   \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
10207     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
10208     \c_undefined_fp
10209   \else:
10210     \exp_after:wN \exp_after:wN \exp_after:wN
10211     \fp_tan_aux_iii:
10212     \fi:
10213   \else:

```

```

10214     \exp_after:wN \fp_tan_aux_iii:
10215     \fi:
10216 }

```

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.

```

10217 \cs_new_protected_nopar:Npn \fp_tan_aux_iii:
10218 {
10219   \l_fp_input_b_integer_int \l_fp_output_decimal_int
10220   \l_fp_input_b_decimal_int \l_fp_output_extended_int
10221   \l_fp_input_b_exponent_int -\c_nine
10222   \fp_standardise:NNNN
10223   \l_fp_input_b_sign_int
10224   \l_fp_input_b_integer_int
10225   \l_fp_input_b_decimal_int
10226   \l_fp_input_b_exponent_int
10227   \fp_sin_aux_ii:
10228   \l_fp_input_a_integer_int \l_fp_output_decimal_int
10229   \l_fp_input_a_decimal_int \l_fp_output_extended_int
10230   \l_fp_input_a_exponent_int -\c_nine
10231   \fp_standardise:NNNN
10232   \l_fp_input_a_sign_int
10233   \l_fp_input_a_integer_int
10234   \l_fp_input_a_decimal_int
10235   \l_fp_input_a_exponent_int
10236   \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
10237   \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
10238     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
10239     \c_zero_fp
10240   \else:
10241     \exp_after:wN \exp_after:wN \exp_after:wN \fp_tan_aux_iv:
10242     \fi:
10243   \else:
10244     \exp_after:wN \fp_tan_aux_iv:
10245     \fi:
10246 }
10247 \cs_new_protected_nopar:Npn \fp_tan_aux_iv:
10248 {
10249   \l_fp_output_integer_int \c_zero
10250   \l_fp_output_decimal_int \c_zero
10251   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
10252   \l_fp_div_offset_int \c_one_hundred_million
10253   \fp_div_loop:
10254   \l_fp_output_exponent_int
10255   \int_eval:w
10256     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
10257   \int_eval_end:
10258   \fp_standardise:NNNN
10259   \l_fp_output_sign_int

```

```

10260     \l_fp_output_integer_int
10261     \l_fp_output_decimal_int
10262     \l_fp_output_exponent_int
10263 \tl_new:c { c_fp_tan ( \l_fp_arg_tl ) _fp }
10264 \tl_gset:cx { c_fp_tan ( \l_fp_arg_tl ) _fp }
10265 {
10266     \if_int_compare:w \l_fp_output_sign_int > \c_zero
10267     +
10268     \else:
10269     -
10270     \fi:
10271     \int_use:N \l_fp_output_integer_int
10272     .
10273     \exp_after:wN \use_none:n
10274     \int_value:w \int_eval:w
10275         \l_fp_output_decimal_int + \c_one_thousand_million
10276     e
10277     \int_use:N \l_fp_output_exponent_int
10278 }
10279 }

```

(End definition for `\fp_tan:Nn` and `\fp_tan:cn`. These functions are documented on page 189.)

187.12 Exponent and logarithm functions

`\c_fp_exp_1_tl` Calculation of exponentials requires a number of precomputed values: first the positive integers.

<code>\c_fp_exp_2_tl</code>	10280	<code>\tl_const:cn { c_fp_exp_1_tl }</code>	{ { 2 } { 718281828 } { 459045235 } { 0 } }
<code>\c_fp_exp_3_tl</code>	10281	<code>\tl_const:cn { c_fp_exp_2_tl }</code>	{ { 7 } { 389056098 } { 930650227 } { 0 } }
<code>\c_fp_exp_4_tl</code>	10282	<code>\tl_const:cn { c_fp_exp_3_tl }</code>	{ { 2 } { 008553692 } { 318766774 } { 1 } }
<code>\c_fp_exp_5_tl</code>	10283	<code>\tl_const:cn { c_fp_exp_4_tl }</code>	{ { 5 } { 459815003 } { 314423908 } { 1 } }
<code>\c_fp_exp_6_tl</code>	10284	<code>\tl_const:cn { c_fp_exp_5_tl }</code>	{ { 1 } { 484131591 } { 025766034 } { 2 } }
<code>\c_fp_exp_7_tl</code>	10285	<code>\tl_const:cn { c_fp_exp_6_tl }</code>	{ { 4 } { 034287934 } { 927351226 } { 2 } }
<code>\c_fp_exp_8_tl</code>	10286	<code>\tl_const:cn { c_fp_exp_7_tl }</code>	{ { 1 } { 096633158 } { 428458599 } { 3 } }
<code>\c_fp_exp_9_tl</code>	10287	<code>\tl_const:cn { c_fp_exp_8_tl }</code>	{ { 2 } { 980957987 } { 041728275 } { 3 } }
<code>\c_fp_exp_10_tl</code>	10288	<code>\tl_const:cn { c_fp_exp_9_tl }</code>	{ { 8 } { 103083927 } { 575384008 } { 3 } }
<code>\c_fp_exp_20_tl</code>	10289	<code>\tl_const:cn { c_fp_exp_10_tl }</code>	{ { 2 } { 202646579 } { 480671652 } { 4 } }
<code>\c_fp_exp_30_tl</code>	10290	<code>\tl_const:cn { c_fp_exp_20_tl }</code>	{ { 4 } { 851651954 } { 097902280 } { 8 } }
<code>\c_fp_exp_40_tl</code>	10291	<code>\tl_const:cn { c_fp_exp_30_tl }</code>	{ { 1 } { 068647458 } { 152446215 } { 13 } }
<code>\c_fp_exp_50_tl</code>	10292	<code>\tl_const:cn { c_fp_exp_40_tl }</code>	{ { 2 } { 353852668 } { 370199854 } { 17 } }
<code>\c_fp_exp_60_tl</code>	10293	<code>\tl_const:cn { c_fp_exp_50_tl }</code>	{ { 5 } { 184705528 } { 587072464 } { 21 } }
<code>\c_fp_exp_70_tl</code>	10294	<code>\tl_const:cn { c_fp_exp_60_tl }</code>	{ { 1 } { 142007389 } { 815684284 } { 26 } }
<code>\c_fp_exp_80_tl</code>	10295	<code>\tl_const:cn { c_fp_exp_70_tl }</code>	{ { 2 } { 515438670 } { 919167006 } { 30 } }
<code>\c_fp_exp_90_tl</code>	10296	<code>\tl_const:cn { c_fp_exp_80_tl }</code>	{ { 5 } { 540622384 } { 393510053 } { 34 } }
<code>\c_fp_exp_100_tl</code>	10297	<code>\tl_const:cn { c_fp_exp_90_tl }</code>	{ { 1 } { 220403294 } { 317840802 } { 39 } }
<code>\c_fp_exp_200_tl</code>	10298	<code>\tl_const:cn { c_fp_exp_100_tl }</code>	{ { 2 } { 688117141 } { 816135448 } { 43 } }
	10299	<code>\tl_const:cn { c_fp_exp_200_tl }</code>	{ { 7 } { 225973768 } { 125749258 } { 86 } }

\c_fp_exp-1_tl Now the negative integers.

\c_fp_exp-2_tl	10300	\tl_const:cn { c_fp_exp-1_tl }	{ { 3 } { 678794411 } { 71442322 } { -1 } }
\c_fp_exp-3_tl	10301	\tl_const:cn { c_fp_exp-2_tl }	{ { 1 } { 353352832 } { 366132692 } { -1 } }
\c_fp_exp-4_tl	10302	\tl_const:cn { c_fp_exp-3_tl }	{ { 4 } { 978706836 } { 786394298 } { -2 } }
\c_fp_exp-5_tl	10303	\tl_const:cn { c_fp_exp-4_tl }	{ { 1 } { 831563888 } { 873418029 } { -2 } }
\c_fp_exp-6_tl	10304	\tl_const:cn { c_fp_exp-5_tl }	{ { 6 } { 737946999 } { 085467097 } { -3 } }
\c_fp_exp-7_tl	10305	\tl_const:cn { c_fp_exp-6_tl }	{ { 2 } { 478752176 } { 666358423 } { -3 } }
\c_fp_exp-8_tl	10306	\tl_const:cn { c_fp_exp-7_tl }	{ { 9 } { 118819655 } { 545162080 } { -4 } }
\c_fp_exp-9_tl	10307	\tl_const:cn { c_fp_exp-8_tl }	{ { 3 } { 354626279 } { 025118388 } { -4 } }
\c_fp_exp-10_tl	10308	\tl_const:cn { c_fp_exp-9_tl }	{ { 1 } { 234098040 } { 866795495 } { -4 } }
\c_fp_exp-20_tl	10309	\tl_const:cn { c_fp_exp-10_tl }	{ { 4 } { 539992976 } { 248451536 } { -5 } }
\c_fp_exp-30_tl	10310	\tl_const:cn { c_fp_exp-20_tl }	{ { 2 } { 061153622 } { 438557828 } { -9 } }
\c_fp_exp-40_tl	10311	\tl_const:cn { c_fp_exp-30_tl }	{ { 9 } { 357622968 } { 840174605 } { -14 } }
\c_fp_exp-50_tl	10312	\tl_const:cn { c_fp_exp-40_tl }	{ { 4 } { 248354255 } { 291588995 } { -18 } }
\c_fp_exp-60_tl	10313	\tl_const:cn { c_fp_exp-50_tl }	{ { 1 } { 928749847 } { 963917783 } { -22 } }
\c_fp_exp-70_tl	10314	\tl_const:cn { c_fp_exp-60_tl }	{ { 8 } { 756510762 } { 696520338 } { -27 } }
\c_fp_exp-80_tl	10315	\tl_const:cn { c_fp_exp-70_tl }	{ { 3 } { 975449735 } { 908646808 } { -31 } }
\c_fp_exp-90_tl	10316	\tl_const:cn { c_fp_exp-80_tl }	{ { 1 } { 804851387 } { 845415172 } { -35 } }
\c_fp_exp-100_tl	10317	\tl_const:cn { c_fp_exp-90_tl }	{ { 8 } { 194012623 } { 990515430 } { -40 } }
\c_fp_exp-200_tl	10318	\tl_const:cn { c_fp_exp-100_tl }	{ { 3 } { 720075976 } { 020835963 } { -44 } }
	10319	\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	10320	\cs_new_protected_nopar:Npn \fp_exp:Nn { \fp_exp_aux:NNn \tl_set:Nn }
\fp_exp_internal:	10321	\cs_new_protected_nopar:Npn \fp_gexp:Nn { \fp_exp_aux:NNn \tl_gset:Nn }
\fp_exp_aux:	10322	\cs_generate_variant:Nn \fp_exp:Nn { c }
\fp_exp_integer:	10323	\cs_generate_variant:Nn \fp_gexp:Nn { c }
\fp_exp_integer_tens:	10324	\cs_new_protected_nopar:Npn \fp_exp_aux:NNn #1#2#3
\fp_exp_integer_units:	10325	{
\fp_exp_integer_const:n	10326	\group_begin:
\fp_exp_integer_const:nnnn	10327	\fp_split:Nn a {#3}
\fp_exp_decimal:	10328	\fp_standardise:NNNN
\fp_exp_Taylor:	10329	\l_fp_input_a_sign_int
\fp_exp_const:Nx	10330	\l_fp_input_a_integer_int
\fp_exp_const:cx	10331	\l_fp_input_a_decimal_int
	10332	\l_fp_input_a_exponent_int
	10333	\l_fp_input_a_extended_int \c_zero
	10334	\tl_set:Nx \l_fp_arg_tl
	10335	{
	10336	\if_int_compare:w \l_fp_input_a_sign_int < \c_zero
	10337	-
	10338	\else:
	10339	+
	10340	\fi:
	10341	\int_use:N \l_fp_input_a_integer_int

```

10342 .
10343 \exp_after:wN \use_none:n
10344 \int_value:w \int_eval:w
10345 \l_fp_input_a_decimal_int + \c_one_thousand_million
10346 e
10347 \int_use:N \l_fp_input_a_exponent_int
10348 }
10349 \if_cs_exist:w c_fp_exp ( \l_fp_arg_tl ) _fp \cs_end:
10350 \else:
10351 \exp_after:wN \fp_exp_internal:
10352 \fi:
10353 \cs_set_protected_nopar:Npx \fp_tmp:w
10354 {
10355 \group_end:
10356 #1 \exp_not:N #2
10357 {
10358 \if_cs_exist:w c_fp_exp ( \l_fp_arg_tl ) _fp
10359 \cs_end:
10360 \use:c { c_fp_exp ( \l_fp_arg_tl ) _fp }
10361 \else:
10362 \c_zero_fp
10363 \fi:
10364 }
10365 }
10366 \fp_tmp:w
10367 }

```

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

```

10368 \cs_new_protected_nopar:Npn \fp_exp_internal:
10369 {
10370 \if_int_compare:w \l_fp_input_a_exponent_int < \c_three
10371 \fp_extended_normalise:
10372 \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10373 \if_int_compare:w \l_fp_input_a_integer_int < 230 \scan_stop:
10374 \exp_after:wN \exp_after:wN \exp_after:wN
10375 \exp_after:wN \exp_after:wN \exp_after:wN
10376 \exp_after:wN \fp_exp_aux:
10377 \else:
10378 \exp_after:wN \exp_after:wN \exp_after:wN
10379 \exp_after:wN \exp_after:wN \exp_after:wN
10380 \exp_after:wN \fp_exp_overflow_msg:
10381 \fi:
10382 \else:
10383 \if_int_compare:w \l_fp_input_a_integer_int < 230 \scan_stop:
10384 \exp_after:wN \exp_after:wN \exp_after:wN
10385 \exp_after:wN \exp_after:wN \exp_after:wN
10386 \exp_after:wN \fp_exp_aux:

```

```

10387         \else:
10388             \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
10389             { \c_zero_fp }
10390         \fi:
10391     \fi:
10392 \else:
10393     \exp_after:wN \fp_exp_overflow_msg:
10394 \fi:
10395 }

```

The main algorithm makes use of the fact that

$$e^{nmp.q} = e^n e^m e^{p.0.q}$$

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.

```

10396 \cs_new_protected_nopar:Npn \fp_exp_aux:
10397 {
10398     \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
10399     \exp_after:wN \fp_exp_integer:
10400 \else:
10401     \l_fp_output_integer_int \c_one
10402     \l_fp_output_decimal_int \c_zero
10403     \l_fp_output_extended_int \c_zero
10404     \l_fp_output_exponent_int \c_zero
10405     \exp_after:wN \fp_exp_decimal:
10406 \fi:
10407 }

```

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).

```

10408 \cs_new_protected_nopar:Npn \fp_exp_integer:
10409 {
10410     \if_int_compare:w \l_fp_input_a_integer_int < \c_one_hundred
10411     \l_fp_exp_integer_int \c_one
10412     \l_fp_exp_decimal_int \c_zero
10413     \l_fp_exp_extended_int \c_zero
10414     \l_fp_exp_exponent_int \c_zero
10415     \exp_after:wN \fp_exp_integer_tens:
10416 \else:
10417     \tl_set:Nx \l_fp_tmp_tl
10418     {

```

```

10419         \exp_after:wN \use_i:nnn
10420         \int_use:N \l_fp_input_a_integer_int
10421     }
10422 \l_fp_input_a_integer_int
10423 \int_eval:w
10424     \l_fp_input_a_integer_int - \l_fp_tmp_tl 00
10425 \int_eval_end:
10426 \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
10427     \if_int_compare:w \l_fp_output_integer_int > 200 \scan_stop:
10428     \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
10429     { \c_zero_fp }
10430 \else:
10431     \fp_exp_integer_const:n { - \l_fp_tmp_tl 00 }
10432     \exp_after:wN \exp_after:wN \exp_after:wN
10433     \exp_after:wN \exp_after:wN \exp_after:wN
10434     \exp_after:wN \fp_exp_integer_tens:
10435 \fi:
10436 \else:
10437     \fp_exp_integer_const:n { \l_fp_tmp_tl 00 }
10438     \exp_after:wN \exp_after:wN \exp_after:wN
10439     \exp_after:wN \fp_exp_integer_tens:
10440 \fi:
10441 \fi:
10442 }

```

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.

```

10443 \cs_new_protected_nopar:Npn \fp_exp_integer_tens:
10444 {
10445     \l_fp_output_integer_int \l_fp_exp_integer_int
10446     \l_fp_output_decimal_int \l_fp_exp_decimal_int
10447     \l_fp_output_extended_int \l_fp_exp_extended_int
10448     \l_fp_output_exponent_int \l_fp_exp_exponent_int
10449     \if_int_compare:w \l_fp_input_a_integer_int > \c_nine
10450     \tl_set:Nx \l_fp_tmp_tl
10451     {
10452         \exp_after:wN \use_i:nn
10453         \int_use:N \l_fp_input_a_integer_int
10454     }
10455     \l_fp_input_a_integer_int
10456     \int_eval:w
10457     \l_fp_input_a_integer_int - \l_fp_tmp_tl 0
10458     \int_eval_end:
10459     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10460     \fp_exp_integer_const:n { \l_fp_tmp_tl 0 }
10461 \else:
10462     \fp_exp_integer_const:n { - \l_fp_tmp_tl 0 }
10463 \fi:

```

```

10464 \fp_mul:NNNNNNNNN
10465 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10466 \l_fp_output_integer_int \l_fp_output_decimal_int
10467 \l_fp_output_extended_int
10468 \l_fp_output_integer_int \l_fp_output_decimal_int
10469 \l_fp_output_extended_int
10470 \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
10471 \fp_extended_normalise_output:
10472 \fi:
10473 \fp_exp_integer_units:
10474 }
10475 \cs_new_protected_nopar:Npn \fp_exp_integer_units:
10476 {
10477 \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
10478 \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10479 \fp_exp_integer_const:n { \int_use:N \l_fp_input_a_integer_int }
10480 \else:
10481 \fp_exp_integer_const:n
10482 { - \int_use:N \l_fp_input_a_integer_int }
10483 \fi:
10484 \fp_mul:NNNNNNNNN
10485 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10486 \l_fp_output_integer_int \l_fp_output_decimal_int
10487 \l_fp_output_extended_int
10488 \l_fp_output_integer_int \l_fp_output_decimal_int
10489 \l_fp_output_extended_int
10490 \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
10491 \fp_extended_normalise_output:
10492 \fi:
10493 \fp_exp_decimal:
10494 }

```

Recovery of the stored constant values into the separate registers is done with a simple expansion then assignment.

```

10495 \cs_new_protected_nopar:Npn \fp_exp_integer_const:n #1
10496 {
10497 \exp_after:wN \exp_after:wN \exp_after:wN
10498 \fp_exp_integer_const:nnnn
10499 \cs:w c_fp_exp_ #1 _tl \cs_end:
10500 }
10501 \cs_new_protected_nopar:Npn \fp_exp_integer_const:nnnn #1#2#3#4
10502 {
10503 \l_fp_exp_integer_int #1 \scan_stop:
10504 \l_fp_exp_decimal_int #2 \scan_stop:
10505 \l_fp_exp_extended_int #3 \scan_stop:
10506 \l_fp_exp_exponent_int #4 \scan_stop:
10507 }

```

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.

```

10508 \cs_new_protected_nopar:Npn \fp_exp_decimal:
10509 {
10510   \if_int_compare:w \l_fp_input_a_decimal_int > \c_zero
10511     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10512       \l_fp_exp_integer_int \c_one
10513       \l_fp_exp_decimal_int \l_fp_input_a_decimal_int
10514       \l_fp_exp_extended_int \l_fp_input_a_extended_int
10515     \else:
10516       \l_fp_exp_integer_int \c_zero
10517       \if_int_compare:w \l_fp_exp_extended_int = \c_zero
10518         \l_fp_exp_decimal_int
10519         \int_eval:w
10520           \c_one_thousand_million - \l_fp_input_a_decimal_int
10521         \int_eval_end:
10522       \l_fp_exp_extended_int \c_zero
10523     \else:
10524       \l_fp_exp_decimal_int
10525       \int_eval:w
10526         999999999 - \l_fp_input_a_decimal_int
10527       \scan_stop:
10528       \l_fp_exp_extended_int
10529       \int_eval:w
10530         \c_one_thousand_million - \l_fp_input_a_extended_int
10531       \int_eval_end:
10532     \fi:
10533   \fi:
10534   \l_fp_input_b_sign_int \l_fp_input_a_sign_int
10535   \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
10536   \l_fp_input_b_extended_int \l_fp_input_a_extended_int
10537   \l_fp_count_int \c_one
10538   \fp_exp_Taylor:
10539   \fp_mul:NNNNNNNNN
10540     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10541     \l_fp_output_integer_int \l_fp_output_decimal_int
10542     \l_fp_output_extended_int
10543     \l_fp_output_integer_int \l_fp_output_decimal_int
10544     \l_fp_output_extended_int
10545   \fi:
10546   \if_int_compare:w \l_fp_output_extended_int < \c_five_hundred_million
10547   \else:
10548     \tex_advance:D \l_fp_output_decimal_int \c_one
10549     \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
10550     \else:
10551       \l_fp_output_decimal_int \c_zero
10552       \tex_advance:D \l_fp_output_integer_int \c_one
10553     \fi:
10554   \fi:

```

```

10555 \fp_standardise:NNNN
10556 \l_fp_output_sign_int
10557 \l_fp_output_integer_int
10558 \l_fp_output_decimal_int
10559 \l_fp_output_exponent_int
10560 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
10561 {
10562   +
10563   \int_use:N \l_fp_output_integer_int
10564   .
10565   \exp_after:wN \use_none:n
10566   \int_value:w \int_eval:w
10567   \l_fp_output_decimal_int + \c_one_thousand_million
10568   e
10569   \int_use:N \l_fp_output_exponent_int
10570 }
10571 }

```

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`.

```

10572 \cs_new_protected_nopar:Npn \fp_exp_Taylor:
10573 {
10574   \tex_advance:D \l_fp_count_int \c_one
10575   \tex_multiply:D \l_fp_input_b_sign_int \l_fp_input_a_sign_int
10576   \fp_mul:NNNNNN
10577   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10578   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10579   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10580   \fp_div_integer:NNNNN
10581   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10582   \l_fp_count_int
10583   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10584   \if_int_compare:w
10585     \int_eval:w
10586     \l_fp_input_b_decimal_int + \l_fp_input_b_extended_int
10587     > \c_zero
10588   \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
10589     \tex_advance:D \l_fp_exp_decimal_int \l_fp_input_b_decimal_int
10590     \tex_advance:D \l_fp_exp_extended_int
10591     \l_fp_input_b_extended_int
10592     \if_int_compare:w \l_fp_exp_extended_int < \c_one_thousand_million
10593   \else:

```

```

10594         \tex_advance:D \l_fp_exp_decimal_int \c_one
10595         \tex_advance:D \l_fp_exp_extended_int
10596             -\c_one_thousand_million
10597     \fi:
10598     \if_int_compare:w \l_fp_exp_decimal_int < \c_one_thousand_million
10599     \else:
10600         \tex_advance:D \l_fp_exp_integer_int \c_one
10601         \tex_advance:D \l_fp_exp_decimal_int
10602             -\c_one_thousand_million
10603     \fi:
10604 \else:
10605     \tex_advance:D \l_fp_exp_decimal_int -\l_fp_input_b_decimal_int
10606     \tex_advance:D \l_fp_exp_extended_int
10607         -\l_fp_input_a_extended_int
10608     \if_int_compare:w \l_fp_exp_extended_int < \c_zero
10609         \tex_advance:D \l_fp_exp_decimal_int \c_minus_one
10610         \tex_advance:D \l_fp_exp_extended_int \c_one_thousand_million
10611     \fi:
10612     \if_int_compare:w \l_fp_exp_decimal_int < \c_zero
10613         \tex_advance:D \l_fp_exp_integer_int \c_minus_one
10614         \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
10615     \fi:
10616 \fi:
10617 \exp_after:wN \fp_exp_Taylor:
10618 \fi:
10619 }

```

This is set up as a function so that the power code can redirect the effect.

```

10620 \cs_new_protected_nopar:Npn \fp_exp_const:Nx #1#2
10621 {
10622     \tl_new:N #1
10623     \tl_gset:Nx #1 {#2}
10624 }
10625 \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 188.)

```

\c_fp_ln_10_1_tl  Constants for working out logarithms: first those for the powers of ten.
\c_fp_ln_10_2_tl
\c_fp_ln_10_3_tl  10626 \tl_const:cn { c_fp_ln_10_1_tl } { { 2 } { 302585092 } { 994045684 } { 0 } }
\c_fp_ln_10_4_tl  10627 \tl_const:cn { c_fp_ln_10_2_tl } { { 4 } { 605170185 } { 988091368 } { 0 } }
\c_fp_ln_10_5_tl  10628 \tl_const:cn { c_fp_ln_10_3_tl } { { 6 } { 907755278 } { 982137052 } { 0 } }
\c_fp_ln_10_6_tl  10629 \tl_const:cn { c_fp_ln_10_4_tl } { { 9 } { 210340371 } { 976182736 } { 0 } }
\c_fp_ln_10_7_tl  10630 \tl_const:cn { c_fp_ln_10_5_tl } { { 1 } { 151292546 } { 497022842 } { 1 } }
\c_fp_ln_10_8_tl  10631 \tl_const:cn { c_fp_ln_10_6_tl } { { 1 } { 381551055 } { 796427410 } { 1 } }
\c_fp_ln_10_9_tl  10632 \tl_const:cn { c_fp_ln_10_7_tl } { { 1 } { 611809565 } { 095831979 } { 1 } }
                  10633 \tl_const:cn { c_fp_ln_10_8_tl } { { 1 } { 842068074 } { 395226547 } { 1 } }
                  10634 \tl_const:cn { c_fp_ln_10_9_tl } { { 2 } { 072326583 } { 694641116 } { 1 } }

```

`\c_fp_ln_2_1_tl` The smaller set for powers of two.

```

\c_fp_ln_2_2_tl
\c_fp_ln_2_3_tl
10635 \tl_const:cn { c_fp_ln_2_1_tl } { { 0 } { 693147180 } { 559945309 } { 0 } }
10636 \tl_const:cn { c_fp_ln_2_2_tl } { { 1 } { 386294361 } { 119890618 } { 0 } }
10637 \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:
\fp_ln_exponent:
\fp_ln_internal:
\fp_ln_exponent_tens:
\fp_ln_exponent_units:
\fp_ln_normalise:
\fp_ln_normalise_aux:NNNNNNNNN
\fp_ln_mantissa:
\fp_ln_mantissa_aux:
\fp_ln_mantissa_divide_two:
\fp_ln_integer_const:nn
\fp_ln_Taylor:
\fp_ln_fixed:
\fp_ln_fixed_aux:NNNNNNNNN
\fp_ln_Taylor_aux:
10638 \cs_new_protected_nopar:Npn \fp_ln:Nn { \fp_ln_aux:NNn \tl_set:Nn }
10639 \cs_new_protected_nopar:Npn \fp_gln:Nn { \fp_ln_aux:NNn \tl_gset:Nn }
10640 \cs_generate_variant:Nn \fp_ln:Nn { c }
10641 \cs_generate_variant:Nn \fp_gln:Nn { c }
10642 \cs_new_protected_nopar:Npn \fp_ln_aux:NNn #1#2#3
10643 {
10644   \group_begin:
10645     \fp_split:Nn a {#3}
10646     \fp_standardise:NNNN
10647     \l_fp_input_a_sign_int
10648     \l_fp_input_a_integer_int
10649     \l_fp_input_a_decimal_int
10650     \l_fp_input_a_exponent_int
10651     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
10652       \if_int_compare:w
10653         \int_eval:w
10654           \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10655           > \c_zero
10656         \exp_after:wN \exp_after:wN \exp_after:wN \fp_ln_aux:
10657       \else:
10658         \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10659         {
10660           \group_end:
10661           ##1 \exp_not:N ##2 { \c_zero_fp }
10662         }
10663         \exp_after:wN \exp_after:wN \exp_after:wN \fp_ln_error_msg:
10664       \fi:
10665     \else:
10666       \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10667       {
10668         \group_end:
10669         ##1 \exp_not:N ##2 { \c_zero_fp }
10670       }
10671       \exp_after:wN \fp_ln_error_msg:
10672     \fi:
10673     \fp_tmp:w #1 #2
10674   }

```

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.

```

10675 \cs_new_protected_nopar:Npn \fp_ln_aux:
10676 {
10677   \tl_set:Nx \l_fp_arg_tl
10678   {
10679     +
10680     \int_use:N \l_fp_input_a_integer_int
10681     .
10682     \exp_after:wN \use_none:n
10683     \int_value:w \int_eval:w
10684     \l_fp_input_a_decimal_int + \c_one_thousand_million
10685     e
10686     \int_use:N \l_fp_input_a_exponent_int
10687   }
10688   \if_cs_exist:w c_fp_ln ( \l_fp_arg_tl ) _fp \cs_end:
10689   \else:
10690     \exp_after:wN \fp_ln_exponent:
10691     \fi:
10692     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10693     {
10694       \group_end:
10695       ##1 \exp_not:N ##2
10696       { \use:c { c_fp_ln ( \l_fp_arg_tl ) _fp } }
10697     }
10698   }

```

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.

```

10699 \cs_new_protected_nopar:Npn \fp_ln_exponent:
10700 {
10701   \fp_ln_internal:
10702   \if_int_compare:w \l_fp_output_extended_int < \c_five_hundred_million
10703   \else:
10704     \tex_advance:D \l_fp_output_decimal_int \c_one
10705     \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million

```

```

10706         \else:
10707             \l_fp_output_decimal_int \c_zero
10708             \tex_advance:D \l_fp_output_integer_int \c_one
10709         \fi:
10710     \fi:
10711     \fp_standardise:NNNN
10712     \l_fp_output_sign_int
10713     \l_fp_output_integer_int
10714     \l_fp_output_decimal_int
10715     \l_fp_output_exponent_int
10716     \tl_const:cx { c_fp_ln ( \l_fp_arg_tl ) _fp }
10717     {
10718         \if_int_compare:w \l_fp_output_sign_int > \c_zero
10719             +
10720         \else:
10721             -
10722         \fi:
10723         \int_use:N \l_fp_output_integer_int
10724         .
10725         \exp_after:wN \use_none:n
10726         \int_value:w \int_eval:w
10727             \l_fp_output_decimal_int + \c_one_thousand_million
10728         \scan_stop:
10729         e
10730         \int_use:N \l_fp_output_exponent_int
10731     }
10732 }
10733 \cs_new_protected_nopar:Npn \fp_ln_internal:
10734 {
10735     \if_int_compare:w \l_fp_input_a_exponent_int < \c_zero
10736         \l_fp_input_a_exponent_int -\l_fp_input_a_exponent_int
10737         \l_fp_output_sign_int \c_minus_one
10738     \else:
10739         \l_fp_output_sign_int \c_one
10740     \fi:
10741     \if_int_compare:w \l_fp_input_a_exponent_int > \c_nine
10742         \exp_after:wN \fp_ln_exponent_tens:NN
10743         \int_use:N \l_fp_input_a_exponent_int
10744     \else:
10745         \l_fp_output_integer_int \c_zero
10746         \l_fp_output_decimal_int \c_zero
10747         \l_fp_output_extended_int \c_zero
10748         \l_fp_output_exponent_int \c_zero
10749     \fi:
10750     \fp_ln_exponent_units:
10751 }
10752 \cs_new_protected_nopar:Npn \fp_ln_exponent_tens:NN #1 #2
10753 {
10754     \l_fp_input_a_exponent_int #2 \scan_stop:
10755     \fp_ln_const:nn { 10 } { #1 }

```

```

10756 \tex_advance:D \l_fp_exp_exponent_int \c_one
10757 \l_fp_output_integer_int \l_fp_exp_integer_int
10758 \l_fp_output_decimal_int \l_fp_exp_decimal_int
10759 \l_fp_output_extended_int \l_fp_exp_extended_int
10760 \l_fp_output_exponent_int \l_fp_exp_exponent_int
10761 }

```

Next the smaller powers of ten, which will need to be combined with the above: always an additive process.

```

10762 \cs_new_protected_nopar:Npn \fp_ln_exponent_units:
10763 {
10764   \if_int_compare:w \l_fp_input_a_exponent_int > \c_zero
10765     \fp_ln_const:nn { 10 } { \int_use:N \l_fp_input_a_exponent_int }
10766     \fp_ln_normalise:
10767     \fp_add:NNNNNNNNN
10768     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10769     \l_fp_output_integer_int \l_fp_output_decimal_int
10770     \l_fp_output_extended_int
10771     \l_fp_output_integer_int \l_fp_output_decimal_int
10772     \l_fp_output_extended_int
10773   \fi:
10774   \fp_ln_mantissa:
10775 }

```

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.

```

10776 \cs_new_protected_nopar:Npn \fp_ln_normalise:
10777 {
10778   \if_int_compare:w \l_fp_exp_exponent_int < \l_fp_output_exponent_int
10779     \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
10780     \exp_after:wN \use_i:nn \exp_after:wN
10781     \fp_ln_normalise_aux:NNNNNNNNN
10782     \int_use:N \l_fp_exp_decimal_int
10783     \exp_after:wN \fp_ln_normalise:
10784   \else:
10785     \l_fp_output_exponent_int \l_fp_exp_exponent_int
10786   \fi:
10787 }
10788 \cs_new_protected_nopar:Npn \fp_ln_normalise_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9
10789 {
10790   \if_int_compare:w \l_fp_exp_integer_int = \c_zero
10791     \l_fp_exp_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
10792   \else:
10793     \tl_set:Nx \l_fp_tmp_tl
10794     {
10795       \int_use:N \l_fp_exp_integer_int

```

```

10796         #1#2#3#4#5#6#7#8
10797     }
10798     \l_fp_exp_integer_int \c_zero
10799     \l_fp_exp_decimal_int \l_fp_tmp_tl \scan_stop:
10800 \fi:
10801 \tex_divide:D \l_fp_exp_extended_int \c_ten
10802 \tl_set:Nx \l_fp_tmp_tl
10803 {
10804     #9
10805     \int_use:N \l_fp_exp_extended_int
10806 }
10807 \l_fp_exp_extended_int \l_fp_tmp_tl \scan_stop:
10808 \tex_advance:D \l_fp_exp_exponent_int \c_one
10809 }

```

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*.

```

10810 \cs_new_protected_nopar:Npn \fp_ln_mantissa:
10811 {
10812     \l_fp_count_int \c_zero
10813     \l_fp_input_a_extended_int \c_zero
10814     \fp_ln_mantissa_aux:
10815     \if_int_compare:w \l_fp_count_int > \c_zero
10816         \fp_ln_const:nn { 2 } { \int_use:N \l_fp_count_int }
10817         \fp_ln_normalise:
10818         \if_int_compare:w \l_fp_output_sign_int > \c_zero
10819             \exp_after:wN \fp_add:NNNNNNNNN
10820         \else:
10821             \exp_after:wN \fp_sub:NNNNNNNNN
10822         \fi:
10823         \l_fp_output_integer_int \l_fp_output_decimal_int
10824         \l_fp_output_extended_int
10825         \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
10826         \l_fp_output_integer_int \l_fp_output_decimal_int
10827         \l_fp_output_extended_int
10828     \fi:
10829     \if_int_compare:w
10830         \int_eval:w
10831         \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int > \c_one
10832     \exp_after:wN \fp_ln_Taylor:
10833     \fi:
10834 }
10835 \cs_new_protected_nopar:Npn \fp_ln_mantissa_aux:
10836 {
10837     \if_int_compare:w \l_fp_input_a_integer_int > \c_one
10838         \tex_advance:D \l_fp_count_int \c_one
10839         \fp_ln_mantissa_divide_two:
10840         \exp_after:wN \fp_ln_mantissa_aux:

```

```

10841     \fi:
10842   }

```

A fast one-shot division by two.

```

10843 \cs_new_protected_nopar:Npn \fp_ln_mantissa_divide_two:
10844 {
10845   \if_int_odd:w \l_fp_input_a_decimal_int
10846     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
10847     \fi:
10848   \if_int_odd:w \l_fp_input_a_integer_int
10849     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
10850     \fi:
10851   \tex_divide:D \l_fp_input_a_integer_int \c_two
10852   \tex_divide:D \l_fp_input_a_decimal_int \c_two
10853   \tex_divide:D \l_fp_input_a_extended_int \c_two
10854 }

```

Recovering constants makes use of the same auxiliary code as for exponents.

```

10855 \cs_new_protected_nopar:Npn \fp_ln_const:nn #1#2
10856 {
10857   \exp_after:wN \exp_after:wN \exp_after:wN
10858     \fp_exp_integer_const:nnnn
10859   \cs:w c_fp_ln_ #1 _ #2 _tl \cs_end:
10860 }

```

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”.

```

10861 \cs_new_protected_nopar:Npn \fp_ln_Taylor:
10862 {
10863   \group_begin:
10864     \l_fp_input_a_integer_int \c_zero
10865     \l_fp_input_a_exponent_int \c_zero
10866     \l_fp_input_b_integer_int \c_two
10867     \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
10868     \l_fp_input_b_exponent_int \c_zero
10869     \fp_div_internal:
10870     \fp_ln_fixed:
10871     \l_fp_input_a_integer_int \l_fp_output_integer_int
10872     \l_fp_input_a_decimal_int \l_fp_output_decimal_int
10873     \l_fp_input_a_extended_int \c_zero
10874     \l_fp_input_a_exponent_int \l_fp_output_exponent_int
10875     \l_fp_output_decimal_int \c_zero %^^A Bug?
10876     \l_fp_output_decimal_int \l_fp_input_a_decimal_int
10877     \l_fp_output_extended_int \l_fp_input_a_extended_int
10878     \fp_mul:NNNNNN
10879     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10880     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10881     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10882     \l_fp_count_int \c_one
10883     \fp_ln_Taylor_aux:
10884     \cs_set_protected_nopar:Npx \fp_tmp:w
10885     {
10886       \group_end:
10887       \l_fp_exp_integer_int \c_zero
10888       \exp_not:N \l_fp_exp_decimal_int
10889       \int_use:N \l_fp_output_decimal_int \scan_stop:
10890       \exp_not:N \l_fp_exp_extended_int
10891       \int_use:N \l_fp_output_extended_int \scan_stop:
10892       \exp_not:N \l_fp_exp_exponent_int
10893       \int_use:N \l_fp_output_exponent_int \scan_stop:
10894     }
10895     \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.

```

10896   \tex_advance:D \l_fp_exp_decimal_int \l_fp_exp_decimal_int
10897   \if_int_compare:w \l_fp_exp_extended_int < \c_five_hundred_million
10898   \else:
10899     \tex_advance:D \l_fp_exp_extended_int -\c_five_hundred_million
10900     \tex_advance:D \l_fp_exp_decimal_int \c_one
10901   \fi:
10902   \tex_advance:D \l_fp_exp_extended_int \l_fp_exp_extended_int

```

```

10903 \fp_ln_normalise:
10904 \if_int_compare:w \l_fp_output_sign_int > \c_zero
10905   \exp_after:wN \fp_add:NNNNNNNNN
10906 \else:
10907   \exp_after:wN \fp_sub:NNNNNNNNN
10908 \fi:
10909 \l_fp_output_integer_int \l_fp_output_decimal_int
10910   \l_fp_output_extended_int
10911 \c_zero \l_fp_exp_decimal_int \l_fp_exp_extended_int
10912 \l_fp_output_integer_int \l_fp_output_decimal_int
10913   \l_fp_output_extended_int
10914 }

```

The usual shifts to move to fixed-point working. This is done using the output registers as this saves a reassignment here.

```

10915 \cs_new_protected_nopar:Npn \fp_ln_fixed:
10916 {
10917   \if_int_compare:w \l_fp_output_exponent_int < \c_zero
10918     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
10919     \exp_after:wN \use_i:nn \exp_after:wN
10920       \fp_ln_fixed_aux:NNNNNNNNN
10921     \int_use:N \l_fp_output_decimal_int
10922     \exp_after:wN \fp_ln_fixed:
10923   \fi:
10924 }
10925 \cs_new_protected_nopar:Npn \fp_ln_fixed_aux:NNNNNNNN #1#2#3#4#5#6#7#8#9
10926 {
10927   \if_int_compare:w \l_fp_output_integer_int = \c_zero
10928     \l_fp_output_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
10929   \else:
10930     \tl_set:Nx \l_fp_tmp_tl
10931     {
10932       \int_use:N \l_fp_output_integer_int
10933       #1#2#3#4#5#6#7#8
10934     }
10935     \l_fp_output_integer_int \c_zero
10936     \l_fp_output_decimal_int \l_fp_tmp_tl \scan_stop:
10937   \fi:
10938   \tex_advance:D \l_fp_output_exponent_int \c_one
10939 }

```

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.

```

10940 \cs_new_protected_nopar:Npn \fp_ln_Taylor_aux:
10941 {
10942   \tex_advance:D \l_fp_count_int \c_two
10943   \fp_mul:NNNNNN

```

```

10944 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10945 \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10946 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10947 \if_int_compare:w
10948 \int_eval:w
10949 \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10950 > \c_zero
10951 \fp_div_integer:NNNNN
10952 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10953 \l_fp_count_int
10954 \l_fp_exp_decimal_int \l_fp_exp_extended_int
10955 \tex_advance:D \l_fp_output_decimal_int \l_fp_exp_decimal_int
10956 \tex_advance:D \l_fp_output_extended_int \l_fp_exp_extended_int
10957 \if_int_compare:w \l_fp_output_extended_int < \c_one_thousand_million
10958 \else:
10959 \tex_advance:D \l_fp_output_decimal_int \c_one
10960 \tex_advance:D \l_fp_output_extended_int
10961 -\c_one_thousand_million
10962 \fi:
10963 \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
10964 \else:
10965 \tex_advance:D \l_fp_output_integer_int \c_one
10966 \tex_advance:D \l_fp_output_decimal_int
10967 -\c_one_thousand_million
10968 \fi:
10969 \exp_after:wN \fp_ln_Taylor_aux:
10970 \fi:
10971 }

```

(End definition for `\fp_ln:Nn` and `\fp_ln:cn`. These functions are documented on page 188.)

`\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:
10972 \cs_new_protected_nopar:Npn \fp_pow:Nn { \fp_pow_aux:NNn \tl_set:Nn }
10973 \cs_new_protected_nopar:Npn \fp_gpow:Nn { \fp_pow_aux:NNn \tl_gset:Nn }
10974 \cs_generate_variant:Nn \fp_pow:Nn { c }
10975 \cs_generate_variant:Nn \fp_gpow:Nn { c }
10976 \cs_new_protected_nopar:Npn \fp_pow_aux:NNn #1#2#3
10977 {
10978 \group_begin:
10979 \fp_read:N #2
10980 \l_fp_input_b_sign_int \l_fp_input_a_sign_int
10981 \l_fp_input_b_integer_int \l_fp_input_a_integer_int
10982 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
10983 \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
10984 \fp_split:Nn a {#3}
10985 \fp_standardise:NNNN

```

```

10986     \l_fp_input_a_sign_int
10987     \l_fp_input_a_integer_int
10988     \l_fp_input_a_decimal_int
10989     \l_fp_input_a_exponent_int
10990 \if_int_compare:w
10991     \int_eval:w
10992     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10993     = \c_zero
10994 \if_int_compare:w
10995     \int_eval:w
10996     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10997     = \c_zero
10998     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10999     {
11000         \group_end:
11001         ##1 ##2 { \c_undefined_fp }
11002     }
11003 \else:
11004     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
11005     {
11006         \group_end:
11007         ##1 ##2 { \c_zero_fp }
11008     }
11009 \fi:
11010 \else:
11011     \if_int_compare:w
11012     \int_eval:w
11013     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
11014     = \c_zero
11015     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
11016     {
11017         \group_end:
11018         ##1 ##2 { \c_one_fp }
11019     }
11020 \else:
11021     \exp_after:wN \exp_after:wN \exp_after:wN
11022     \fp_pow_aux_i:
11023 \fi:
11024 \fi:
11025 \fp_tmp:w #1 #2
11026 }

```

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.

```

11027 \cs_new_protected_nopar:Npn \fp_pow_aux_i:
11028 {
11029     \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
11030     \tl_set:Nn \l_fp_sign_tl { + }

```

```

11031     \exp_after:wN \fp_pow_aux_ii:
11032 \else:
11033     \l_fp_input_a_extended_int \c_zero
11034     \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
11035         \group_begin:
11036         \fp_extended_normalise:
11037         \if_int_compare:w
11038             \int_eval:w
11039             \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
11040             = \c_zero
11041         \group_end:
11042         \tl_set:Nn \l_fp_sign_tl { - }
11043         \exp_after:wN \exp_after:wN \exp_after:wN
11044         \exp_after:wN \exp_after:wN \exp_after:wN
11045         \exp_after:wN \fp_pow_aux_ii:
11046     \else:
11047         \group_end:
11048         \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
11049         {
11050             \group_end:
11051             ##1 ##2 { \c_undefined_fp }
11052         }
11053     \fi:
11054 \else:
11055     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
11056     {
11057         \group_end:
11058         ##1 ##2 { \c_undefined_fp }
11059     }
11060 \fi:
11061 \fi:
11062 }

```

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.

```

11063 \cs_new_protected_nopar:Npn \fp_pow_aux_ii:
11064 {
11065     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
11066         \exp_after:wN \fp_pow_aux_iv:
11067     \else:
11068         \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
11069             \group_begin:
11070             \l_fp_input_a_extended_int \c_zero
11071             \fp_extended_normalise:
11072             \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
11073                 \if_int_compare:w \l_fp_input_a_integer_int > \c_ten

```

```

11074         \group_end:
11075         \exp_after:wN \exp_after:wN \exp_after:wN
11076         \exp_after:wN \exp_after:wN \exp_after:wN
11077         \exp_after:wN \exp_after:wN \exp_after:wN
11078         \exp_after:wN \exp_after:wN \exp_after:wN
11079         \exp_after:wN \exp_after:wN \exp_after:wN
11080         \fp_pow_aux_iv:
11081     \else:
11082         \group_end:
11083         \exp_after:wN \exp_after:wN \exp_after:wN
11084         \exp_after:wN \exp_after:wN \exp_after:wN
11085         \exp_after:wN \exp_after:wN \exp_after:wN
11086         \exp_after:wN \exp_after:wN \exp_after:wN
11087         \exp_after:wN \exp_after:wN \exp_after:wN
11088         \exp_after:wN \fp_pow_aux_iii:
11089     \fi:
11090 \else:
11091     \group_end:
11092     \exp_after:wN \exp_after:wN \exp_after:wN
11093     \exp_after:wN \exp_after:wN \exp_after:wN
11094     \exp_after:wN \fp_pow_aux_iv:
11095 \fi:
11096 \else:
11097     \exp_after:wN \exp_after:wN \exp_after:wN
11098     \fp_pow_aux_iv:
11099 \fi:
11100 \fi:
11101 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
11102 {
11103     \group_end:
11104     ##1 ##2
11105     {
11106         \l_fp_sign_tl
11107         \int_use:N \l_fp_output_integer_int
11108         .
11109         \exp_after:wN \use_none:n
11110         \int_value:w \int_eval:w
11111         \l_fp_output_decimal_int + \c_one_thousand_million
11112         e
11113         \int_use:N \l_fp_output_exponent_int
11114     }
11115 }
11116 }

```

For the small negative integer powers, the calculation is done for the positive power and the reciprocal is then taken.

```

11117 \cs_new_protected_nopar:Npn \fp_pow_aux_iii:
11118 {
11119     \l_fp_input_a_sign_int \c_one

```

```

11120 \fp_pow_aux_iv:
11121 \l_fp_input_a_integer_int \c_one
11122 \l_fp_input_a_decimal_int \c_zero
11123 \l_fp_input_a_exponent_int \c_zero
11124 \l_fp_input_b_integer_int \l_fp_output_integer_int
11125 \l_fp_input_b_decimal_int \l_fp_output_decimal_int
11126 \l_fp_input_b_exponent_int \l_fp_output_exponent_int
11127 \fp_div_internal:
11128 }

```

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.

```

11129 \cs_new_protected_nopar:Npn \fp_pow_aux_iv:
11130 {
11131   \group_begin:
11132     \l_fp_input_a_integer_int \l_fp_input_b_integer_int
11133     \l_fp_input_a_decimal_int \l_fp_input_b_decimal_int
11134     \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
11135     \fp_ln_internal:
11136     \cs_set_protected_nopar:Npx \fp_tmp:w
11137     {
11138       \group_end:
11139       \exp_not:N \l_fp_input_b_sign_int
11140       \int_use:N \l_fp_output_sign_int \scan_stop:
11141       \exp_not:N \l_fp_input_b_integer_int
11142       \int_use:N \l_fp_output_integer_int \scan_stop:
11143       \exp_not:N \l_fp_input_b_decimal_int
11144       \int_use:N \l_fp_output_decimal_int \scan_stop:
11145       \exp_not:N \l_fp_input_b_extended_int
11146       \int_use:N \l_fp_output_extended_int \scan_stop:
11147       \exp_not:N \l_fp_input_b_exponent_int
11148       \int_use:N \l_fp_output_exponent_int \scan_stop:
11149     }
11150     \fp_tmp:w
11151     \l_fp_input_a_extended_int \c_zero
11152     \fp_mul:NNNNNNNNN
11153     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
11154     \l_fp_input_a_extended_int
11155     \l_fp_input_b_integer_int \l_fp_input_b_decimal_int
11156     \l_fp_input_b_extended_int
11157     \l_fp_output_integer_int \l_fp_output_decimal_int
11158     \l_fp_output_extended_int
11159     \l_fp_output_exponent_int
11160     \int_eval:w

```

```

11161         \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
11162     \scan_stop:
11163     \fp_extended_normalise_output:
11164     \tex_multiply:D \l_fp_input_a_sign_int \l_fp_input_b_sign_int
11165     \l_fp_input_a_integer_int \l_fp_output_integer_int
11166     \l_fp_input_a_decimal_int \l_fp_output_decimal_int
11167     \l_fp_input_a_extended_int \l_fp_output_extended_int
11168     \l_fp_input_a_exponent_int \l_fp_output_exponent_int
11169     \l_fp_output_integer_int \c_zero
11170     \l_fp_output_decimal_int \c_zero
11171     \l_fp_output_extended_int \c_zero
11172     \l_fp_output_exponent_int \c_zero
11173     \cs_set_eq:NN \fp_exp_const:Nx \use_none:nn
11174     \fp_exp_internal:
11175 }

```

(End definition for `\fp_pow:Nn` and `\fp_pow:cn`. These functions are documented on page 187.)

187.13 Tests for special values

`\fp_if_undefined_p:N`
`\fp_if_undefined:NTF`

Testing for an undefined value is easy.

```

11176 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { p , T , F , TF }
11177 {
11178     \if_meaning:w #1 \c_undefined_fp
11179     \prg_return_true:
11180     \else:
11181         \prg_return_false:
11182     \fi:
11183 }

```

(End definition for `\fp_if_undefined:N`. These functions are documented on page 184.)

`\fp_if_zero_p:N`
`\fp_if_zero:NTF`

Testing for a zero fixed-point is also easy.

```

11184 \prg_new_conditional:Npnn \fp_if_zero:N #1 { p , T , F , TF }
11185 {
11186     \if_meaning:w #1 \c_zero_fp
11187     \prg_return_true:
11188     \else:
11189         \prg_return_false:
11190     \fi:
11191 }

```

(End definition for `\fp_if_zero:N`. These functions are documented on page 185.)

187.14 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.

```

11192 \prg_new_protected_conditional:Npnn \fp_compare:nNn #1#2#3 { T , F , TF }
11193 {
11194   \group_begin:
11195     \fp_split:Nn a {#1}
11196     \fp_standardise:NNNN
11197     \l_fp_input_a_sign_int
11198     \l_fp_input_a_integer_int
11199     \l_fp_input_a_decimal_int
11200     \l_fp_input_a_exponent_int
11201     \fp_split:Nn b {#3}
11202     \fp_standardise:NNNN
11203     \l_fp_input_b_sign_int
11204     \l_fp_input_b_integer_int
11205     \l_fp_input_b_decimal_int
11206     \l_fp_input_b_exponent_int
11207     \fp_compare_aux:N #2
11208   }
11209 \prg_new_protected_conditional:Npnn \fp_compare:NNN #1#2#3 { T , F , TF }
11210 {
11211   \group_begin:
11212     \fp_read:N #3
11213     \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
11214     \l_fp_input_b_integer_int   \l_fp_input_a_integer_int
11215     \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
11216     \l_fp_input_b_exponent_int  \l_fp_input_a_exponent_int
11217     \fp_read:N #1
11218     \fp_compare_aux:N #2
11219   }
11220 \cs_new_protected_nopar:Npn \fp_compare_aux:N #1
11221 {
11222   \cs_if_exist:cTF { fp_compare_#1: }
11223   { \use:c { fp_compare_#1: } }
11224   {
11225     \group_end:
11226     \prg_return_false:
11227   }
11228 }

```

For equality, the test is pretty easy as things are either equal or they are not.

```

11229 \cs_new_protected_nopar:cpn { fp_compare_=: }
11230 {
11231   \if_int_compare:w \l_fp_input_a_sign_int = \l_fp_input_b_sign_int
11232   \if_int_compare:w \l_fp_input_a_integer_int = \l_fp_input_b_integer_int
11233   \if_int_compare:w \l_fp_input_a_decimal_int = \l_fp_input_b_decimal_int
11234   \if_int_compare:w
11235     \l_fp_input_a_exponent_int = \l_fp_input_b_exponent_int
11236   \group_end:
11237   \prg_return_true:

```

```

11238         \else:
11239             \group_end:
11240             \prg_return_false:
11241         \fi:
11242     \else:
11243         \group_end:
11244         \prg_return_false:
11245     \fi:
11246 \else:
11247     \group_end:
11248     \prg_return_false:
11249 \fi:
11250 \else:
11251     \group_end:
11252     \prg_return_false:
11253 \fi:
11254 }

```

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.

```

11255 \cs_new_protected_nopar:cpn { fp_compare_>: }
11256 {
11257     \if_int_compare:w \int_eval:w
11258         \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
11259         = \c_zero
11260     \if_int_compare:w \int_eval:w
11261         \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
11262         = \c_zero
11263     \group_end:
11264     \prg_return_false:
11265 \else:
11266     \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
11267     \group_end:
11268     \prg_return_false:
11269 \else:
11270     \group_end:
11271     \prg_return_true:
11272 \fi:
11273 \fi:
11274 \else:
11275     \if_int_compare:w \int_eval:w
11276         \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
11277         = \c_zero
11278     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
11279     \group_end:
11280     \prg_return_true:
11281 \else:
11282     \group_end:
11283     \prg_return_false:

```

```

11284         \fi:
11285     \else:
11286         \use:c { fp_compare_>_aux: }
11287     \fi:
11288 \fi:
11289 }

```

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.

```

11290 \cs_new_protected_nopar:cpn { fp_compare_>_aux: }
11291 {
11292     \if_int_compare:w \l_fp_input_a_sign_int > \l_fp_input_b_sign_int
11293     \group_end:
11294     \prg_return_true:
11295 \else:
11296     \if_int_compare:w \l_fp_input_a_sign_int < \l_fp_input_b_sign_int
11297     \group_end:
11298     \prg_return_false:
11299 \else:
11300     \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
11301     \use:c { fp_compare_absolute_a>b: }
11302 \else:
11303     \use:c { fp_compare_absolute_a<b: }
11304 \fi:
11305 \fi:
11306 \fi:
11307 }

```

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.

```

11308 \cs_new_protected_nopar:cpn { fp_compare_absolute_a>b: }
11309 {
11310     \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
11311     \group_end:
11312     \prg_return_true:
11313 \else:
11314     \if_int_compare:w \l_fp_input_a_exponent_int < \l_fp_input_b_exponent_int
11315     \group_end:
11316     \prg_return_false:
11317 \else:
11318     \if_int_compare:w \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
11319     \group_end:
11320     \prg_return_true:
11321 \else:
11322     \if_int_compare:w
11323         \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
11324     \group_end:

```

```

11325         \prg_return_false:
11326     \else:
11327         \if_int_compare:w
11328             \l_fp_input_a_decimal_int > \l_fp_input_b_decimal_int
11329         \group_end:
11330         \prg_return_true:
11331     \else:
11332         \group_end:
11333         \prg_return_false:
11334     \fi:
11335 \fi:
11336 \fi:
11337 \fi:
11338 \fi:
11339 }
11340 \cs_new_protected_nopar:cpn { fp_compare_absolute_a<b: }
11341 {
11342     \if_int_compare:w \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
11343     \group_end:
11344     \prg_return_true:
11345 \else:
11346     \if_int_compare:w \l_fp_input_b_exponent_int < \l_fp_input_a_exponent_int
11347     \group_end:
11348     \prg_return_false:
11349 \else:
11350     \if_int_compare:w \l_fp_input_b_integer_int > \l_fp_input_a_integer_int
11351     \group_end:
11352     \prg_return_true:
11353 \else:
11354     \if_int_compare:w
11355         \l_fp_input_b_integer_int < \l_fp_input_a_integer_int
11356     \group_end:
11357     \prg_return_false:
11358 \else:
11359     \if_int_compare:w
11360         \l_fp_input_b_decimal_int > \l_fp_input_a_decimal_int
11361     \group_end:
11362     \prg_return_true:
11363 \else:
11364     \group_end:
11365     \prg_return_false:
11366 \fi:
11367 \fi:
11368 \fi:
11369 \fi:
11370 \fi:
11371 }

```

This is just a case of reversing the two input values and then running the tests already defined.

```

11372 \cs_new_protected_nopar:cpn { fp_compare_<: }
11373 {
11374   \tl_set:Nx \l_fp_tmp_tl
11375   {
11376     \int_set:Nn \exp_not:N \l_fp_input_a_sign_int
11377     { \int_use:N \l_fp_input_b_sign_int }
11378     \int_set:Nn \exp_not:N \l_fp_input_a_integer_int
11379     { \int_use:N \l_fp_input_b_integer_int }
11380     \int_set:Nn \exp_not:N \l_fp_input_a_decimal_int
11381     { \int_use:N \l_fp_input_b_decimal_int }
11382     \int_set:Nn \exp_not:N \l_fp_input_a_exponent_int
11383     { \int_use:N \l_fp_input_b_exponent_int }
11384     \int_set:Nn \exp_not:N \l_fp_input_b_sign_int
11385     { \int_use:N \l_fp_input_a_sign_int }
11386     \int_set:Nn \exp_not:N \l_fp_input_b_integer_int
11387     { \int_use:N \l_fp_input_a_integer_int }
11388     \int_set:Nn \exp_not:N \l_fp_input_b_decimal_int
11389     { \int_use:N \l_fp_input_a_decimal_int }
11390     \int_set:Nn \exp_not:N \l_fp_input_b_exponent_int
11391     { \int_use:N \l_fp_input_a_exponent_int }
11392   }
11393   \l_fp_tmp_tl
11394   \use:c { fp_compare_>: }
11395 }

```

(End definition for `\fp_compare:nNn`. This function is documented on page ??.)

`\fp_compare:nTF`

As TeX 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
11396 \prg_new_protected_conditional:Npnn \fp_compare:n #1 { T , F , TF }
11397 {
11398   \group_begin:
11399   \tl_set:Nx \l_fp_tmp_tl
11400   {
11401     \group_end:
11402     \fp_compare_aux_i:w #1 \exp_not:n { == \q_nil == \q_stop }
11403   }
11404   \l_fp_tmp_tl
11405 }
11406 \cs_new_protected_nopar:Npn \fp_compare_aux_i:w #1 == #2 == #3 \q_stop
11407 {
11408   \quark_if_nil:nTF {#2}
11409   { \fp_compare_aux_ii:w #1 != \q_nil != \q_stop }
11410   { \fp_compare:nNnTF {#1} = {#2} \prg_return_true: \prg_return_false: }
11411 }

```

```

11412 \cs_new_protected_nopar:Npn \fp_compare_aux_ii:w #1 != #2 != #3 \q_stop
11413 {
11414   \quark_if_nil:nTF {#2}
11415   { \fp_compare_aux_iii:w #1 <= \q_nil <= \q_stop }
11416   { \fp_compare:nNnTF {#1} = {#2} \prg_return_false: \prg_return_true: }
11417 }
11418 \cs_new_protected_nopar:Npn \fp_compare_aux_iii:w #1 <= #2 <= #3 \q_stop
11419 {
11420   \quark_if_nil:nTF {#2}
11421   { \fp_compare_aux_iv:w #1 >= \q_nil >= \q_stop }
11422   { \fp_compare:nNnTF {#1} > {#2} \prg_return_false: \prg_return_true: }
11423 }
11424 \cs_new_protected_nopar:Npn \fp_compare_aux_iv:w #1 >= #2 >= #3 \q_stop
11425 {
11426   \quark_if_nil:nTF {#2}
11427   { \fp_compare_aux_v:w #1 = \q_nil \q_stop }
11428   { \fp_compare:nNnTF {#1} < {#2} \prg_return_false: \prg_return_true: }
11429 }
11430 \cs_new_protected_nopar:Npn \fp_compare_aux_v:w #1 = #2 = #3 \q_stop
11431 {
11432   \quark_if_nil:nTF {#2}
11433   { \fp_compare_aux_vi:w #1 < \q_nil < \q_stop }
11434   { \fp_compare:nNnTF {#1} = {#2} \prg_return_true: \prg_return_false: }
11435 }
11436 \cs_new_protected_nopar:Npn \fp_compare_aux_vi:w #1 < #2 < #3 \q_stop
11437 {
11438   \quark_if_nil:nTF {#2}
11439   { \fp_compare_aux_vii:w #1 > \q_nil > \q_stop }
11440   { \fp_compare:nNnTF {#1} < {#2} \prg_return_true: \prg_return_false: }
11441 }
11442 \cs_new_protected_nopar:Npn \fp_compare_aux_vii:w #1 > #2 > #3 \q_stop
11443 {
11444   \quark_if_nil:nTF {#2}
11445   { \prg_return_false: }
11446   { \fp_compare:nNnTF {#1} > {#2} \prg_return_true: \prg_return_false: }
11447 }

```

(End definition for `\fp_compare:n`. This function is documented on page 185.)

187.15 Messages

`\fp_overflow_msg:` A generic overflow message, used whenever there is a possible overflow.

```

11448 \msg_kernel_new:nnnn { fpu } { overflow }
11449 { Number~too~big. }
11450 {
11451   The~input~given~is~too~big~for~the~LaTeX~floating~point~unit. \\
11452   Further~errors~may~well~occur!
11453 }

```

```

11454 \cs_new_protected_nopar:Npn \fp_overflow_msg:
11455 { \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.

```

11456 \msg_kernel_new:nnnn { fpu } { exponent-overflow }
11457 { Number~too-big-for-exponent-unit. }
11458 {
11459   The~exponent~of~the~input~given~is~too~big~for~the~floating~point~
11460   unit:~the~maximum~input~value~for~an~exponent~is~230.
11461 }
11462 \cs_new_protected_nopar:Npn \fp_exp_overflow_msg:
11463 { \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

```

11464 \msg_kernel_new:nnnn { fpu } { logarithm-input-error }
11465 { Invalid~input~to~ln~function. }
11466 { Logarithms~can~only~be~calculated~for~positive~numbers. }
11467 \cs_new_protected_nopar:Npn \fp_ln_error_msg: {
11468   \msg_kernel_error:nn { fpu } { logarithm-input-error }
11469 }

```

(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.

```

11470 \msg_kernel_new:nnnn { fpu } { trigonometric-overflow }
11471 { Number~too-big-for-trigonometry-unit. }
11472 {
11473   The~trigonometry~code~can~only~work~with~numbers~smaller~
11474   than~1000000000.
11475 }
11476 \cs_new_protected_nopar:Npn \fp_trig_overflow_msg:
11477 { \msg_kernel_error:nn { fpu } { trigonometric-overflow } }

```

(End definition for `\fp_trig_overflow_msg:`. This function is documented on page ??.)

```

11478 </initex | package>

```

188 l3luatex implementation

```

11479 <*initex | package>

```

Announce and ensure that the required packages are loaded.

```

11480 <*package>
11481 \ProvidesExplPackage
11482   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
11483 \package_check_loaded_expl:
11484 </package>

```

`\lua_now:n` When LuaTeX is in use, this is all a question of primitives with new names. On the other hand, for pdfTeX and XeTeX the argument should be removed from the input stream before issuing an error. This is expandable, using `\msg_expandable_error:n` as for V-type expansion.

```

\lua_now:x
\lua_shipout_x:n
\lua_shipout_x:x
\lua_shipout:n
\lua_shipout:x
11485 \luatex_if_engine:TF
11486 {
11487   \cs_new_eq:NN \lua_now:x      \luatex_directlua:D
11488   \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
11489 }
11490 {
11491   \cs_new:Npn \lua_now:x #1
11492   {
11493     \msg_expandable_error:n
11494     { LuaTeX~ engine~ not~ in~ use!~ Ignoring~ \lua_now:x. }
11495   }
11496   \cs_new_protected:Npn \lua_shipout_x:n #1
11497   {
11498     \msg_expandable_error:n
11499     { LuaTeX~ engine~ not~ in~ use!~ Ignoring~ \lua_shipout_x:n. }
11500   }
11501 }
11502 \cs_new:Npn \lua_now:n #1
11503 { \lua_now:x { \exp_not:n {#1} } }
11504 \cs_generate_variant:Nn \lua_shipout_x:n { x }
11505 \cs_new_protected:Npn \lua_shipout:n #1
11506 { \lua_shipout_x:n { \exp_not:n {#1} } }
11507 \cs_generate_variant:Nn \lua_shipout:n { x }

```

(End definition for `\lua_now:n` and `\lua_now:x`. These functions are documented on page 191.)

188.1 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`

```

11508 \int_new:N \g_cctab_allocate_int
11509 \int_set:Nn \g_cctab_allocate_int { -1 }
11510 \int_new:N \g_cctab_stack_int
11511 \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.

```

11512 \cs_new_protected_nopar:Npn \cctab_new:N #1
11513 {
11514   \cs_if_free:NTF #1
11515   {
11516     \int_gadd:Nn \g_cctab_allocate_int { 2 }
11517     \int_compare:nNnTF
11518       { \g_cctab_allocate_int } < { \c_max_register_int + 1 }
11519       {
11520         \pref_global:D \tex_mathchardef:D #1 \g_cctab_allocate_int
11521         \luatex_initcatcodetable:D #1
11522       }
11523       { \msg_kernel_fatal:nxx { alloc } { out-of-registers } { cctab } }
11524   }
11525   {
11526     \msg_kernel_error:nxx { code } { variable-already-defined }
11527     { \token_to_str:N #1 }
11528   }
11529 }
11530 \luatex_if_engine:F
11531 { \cs_set_protected_nopar:Npn \cctab_new:N #1 { \lua_wrong_engine: } }
11532 <*package>
11533 \luatex_if_engine:T
11534 {
11535   \cs_set_protected_nopar:Npn \cctab_new:N #1
11536   {
11537     \newcatcodetable #1
11538     \luatex_initcatcodetable:D #1
11539   }
11540 }
11541 </package>

```

(End definition for `\cctab_new:N`. This function is documented on page 192.)

`\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`

```

11542 \cs_new_protected_nopar:Npn \cctab_begin:N #1
11543 {
11544   \seq_gpush:Nx \g_cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
11545   \luatex_catcodetable:D #1
11546   \int_gadd:Nn \g_cctab_stack_int { 2 }
11547   \int_compare:nNnT { \g_cctab_stack_int } > { 268 435 453 }
11548     { \msg_kernel_error:nn { code } { cctab-stack-full } }
11549   \luatex_savecatcodetable:D \g_cctab_stack_int
11550   \luatex_catcodetable:D \g_cctab_stack_int
11551 }

```

```

11552 \cs_new_protected_nopar:Npn \cctab_end:
11553 {
11554   \int_gsub:Nn \g_cctab_stack_int { 2 }
11555   \seq_gpop:NN \g_cctab_stack_seq \l_cctab_tmp_tl
11556   \quark_if_no_value:NT \l_cctab_tmp_tl
11557   { \tl_set:Nn \l_cctab_tmp_tl { 0 } }
11558   \luatex_catcodetable:D \l_cctab_tmp_tl \scan_stop:
11559 }
11560 \luatex_if_engine:F
11561 {
11562   \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \lua_wrong_engine: }
11563   \cs_set_protected_nopar:Npn \cctab_end: { \lua_wrong_engine: }
11564 }
11565 <*package>
11566 \luatex_if_engine:T
11567 {
11568   \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \BeginCatcodeRegime #1 }
11569   \cs_set_protected_nopar:Npn \cctab_end: { \EndCatcodeRegime }
11570 }
11571 </package>
11572 \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.

```

11573 \cs_new_protected:Npn \cctab_gset:Nn #1#2
11574 {
11575   \group_begin:
11576   #2
11577   \luatex_savecatcodetable:D #1
11578   \group_end:
11579 }
11580 \luatex_if_engine:F
11581 { \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 192.)

`\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`

```

11582 \luatex_if_engine:T
11583 {
11584   \cctab_new:N \c_code_cctab
11585   \cctab_gset:Nn \c_code_cctab { }
11586 }

```

```

11587 <*package>
11588 \luatex_if_engine:T
11589 {
11590   \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
11591   \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
11592   \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
11593   \cs_new_eq:NN \c_string_cctab \CatcodeTableString
11594 }
11595 </package>
11596 <*initex>
11597 \luatex_if_engine:T
11598 {
11599   \cctab_new:N \c_document_cctab
11600   \cctab_new:N \c_other_cctab
11601   \cctab_new:N \c_string_cctab
11602   \cctab_gset:Nn \c_document_cctab
11603   {
11604     \char_set_catcode_space:n { 9 }
11605     \char_set_catcode_space:n { 32 }
11606     \char_set_catcode_other:n { 58 }
11607     \char_set_catcode_math_subscript:n { 95 }
11608     \char_set_catcode_active:n { 126 }
11609   }
11610   \cctab_gset:Nn \c_other_cctab
11611   {
11612     \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
11613     { \char_set_catcode_other:n {#1} }
11614   }
11615   \cctab_gset:Nn \c_string_cctab
11616   {
11617     \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
11618     { \char_set_catcode_other:n {#1} }
11619     \char_set_catcode_space:n { 32 }
11620   }
11621 }
11622 </initex>
11623 </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	
\!	6917
\#	5
\	2265, 2266, 2279, 2280, 6917, 6918
*	2588, 2590, 2594, 2601
\,	7370, 7372
\-	348
\.	6860, 6865
\.bool_gset:N	7728
\.bool_gset_inverse:N	7732
\.bool_set:N	7728
\.bool_set_inverse:N	7732
\.choice:	7736
\.choice_code:n	7744
\.choice_code:x	7744
\.choices:nn	7738
\.code:n	7740
\.code:x	7740
\.default:V	7748
\.default:n	7748
\.dim_gset:N	7752
\.dim_gset:c	7752
\.dim_set:N	7752
\.dim_set:c	7752
\.fp_gset:N	7760
\.fp_gset:c	7760
\.fp_set:N	7760
\.fp_set:c	7760
\.generate_choices:n	7768
\.int_gset:N	7770
\.int_gset:c	7770
\.int_set:N	7770
\.int_set:c	7770
\.meta:n	7778
\.meta:x	7778
\.multichoice:	7782
\.multichoice:nn	7782
\.skip_gset:N	7786
\.skip_gset:c	7786
\.skip_set:N	7786
\.skip_set:c	7786
\.tl_gset:N	7794
\.tl_gset:c	7794
\.tl_gset_x:N	7794
\.tl_gset_x:c	7794
\.tl_set:N	7794
\.tl_set:c	7794
\.tl_set_x:N	7794
\.tl_set_x:c	7794
\.value_forbidden:	7810
\.value_required:	7810
\/	347
\:	1073, 2680, 2813
\::	36,
	1525, 1526, 1527, 1527–1530, 1532,
	1534, 1541, 1546, 1552, 1673–1699,
	1701, 1706, 1708, 1713, 1739, 1740
\::N	36, 1529,
	1529, 1682, 1688, 1689, 1693, 1694
\::V	36, 1546, 1546, 1679
\::V_unbraced	1700, 1708
\::c	36, 1530, 1530,
	1674, 1680, 1683, 1690, 1697, 1698
\::f	36, 1534, 1534, 1675–1677, 1740
\::f_unbraced	1700, 1701
\::n	36, 1528,
	1528, 1674, 1677–1679, 1684, 1688,
	1690, 1691, 1693, 1695, 1696, 1698
\::o	36, 1532, 1532, 1675, 1678,
	1680, 1681, 1685, 1686, 1688, 1689,
	1691, 1692, 1694, 1696, 1699, 1739
\::o_unbraced	1700, 1706, 1739, 1740
\::v	36, 1546, 1552
\::v_unbraced	1700, 1713
\::x	36, 1541,
	1541, 1673, 1682–1687, 1693–1699
\;	2680, 2812, 2813
\=	7369, 7371
\?	1814, 2720
\@	1073, 1074, 4296, 4297
\@end	763
\@hyph	766
\@input	767
\@italiccorr	768
\@underline	769
\@addtofilelist	8095

\@currname	8027	\badness	617
\@filelist	8120	\baselineskip	545
\@ifpackageloaded	216	\batchmode	438
\@l@expl@log@functions@bool	1201	\begin	6880
\@namedef	198	\BeginCatcodeRegime	11568
\@nil	193, 201	\begingroup	12, 61, 100, 228, 232, 338, 376
\@popfilename	163, 181, 183	\beginL	730
\@pushfilename	163, 164, 179	\beginR	732
\@tempa	54, 56, 64	\belowdisplayskip	482
\@tempboxa	6295	\belowdisplayskip	483
\[3761, 6860	\binoppenalty	506
\\	1814, 2720,	\bool(:w	1912
	6609, 6616, 6655, 6664, 6712, 6719,	\bool_)_0:w	1912
	6822, 6840, 6842, 6847, 6848, 6884,	\bool_)_1:w	1912
	6886, 6892, 6894, 6958, 7034, 7042,	\bool_8_0:w	1912
	7189, 7211, 7231, 7239, 7246, 7253,	\bool_8_1:w	1912
	7262, 7270, 7271, 7315, 7951, 7966,	\bool_:w	1912
	7967, 7973, 7986, 7999, 8005, 11451	\bool_choose:NN	1912, 2001, 2009
\{	3, 5299, 5813, 6120, 6122	\bool_cleanup:N	1912, 1994, 1997, 1999
\}	4, 5299, 5813, 6120, 6122, 6916	\bool_do_until:cn	2057
\^	6, 9, 249, 3075, 3082, 7320	\bool_do_until:Nn	2057, 2059, 2060, 2062
_	2279	\bool_do_until:nn	2063, 2084, 2087
\	3867	\bool_do_while:cn	2057
		\bool_do_while:Nn	2057, 2057, 2058, 2061
		\bool_do_while:nn	2063, 2071, 2074
		\bool_eval_skip_to_end:Nw	1912,
			2021, 2022, 2024, 2026, 2028, 2042
		\bool_eval_skip_to_end_aux:Nw	1912, 2030, 2034
			1912, 2038, 2040
		\bool_get_next:N	1912, 1923, 1925,
			1945, 1974, 1994, 1996, 2011–2014
		\bool_get_next:NN	1945, 1953
		\bool_get_not_next:N	1935, 1946, 1973
		\bool_get_not_next:NN	1946, 1966
		\bool_gset:cn	41, 1892
		.bool_gset:N	168
		\bool_gset:Nn	41, 1892, 1894, 1897
		\bool_gset_eq:cc	40, 1884, 1891
		\bool_gset_eq:cN	40, 1884, 1890
		\bool_gset_eq:Nc	40, 1884, 1889
		\bool_gset_eq:NN	40, 1884, 1888
		\bool_gset_false:c	40, 1872
		\bool_gset_false:N	40, 1872, 1878, 1883
		.bool_gset_inverse:N	168
		\bool_gset_true:c	40, 1872
		\bool_gset_true:N	40, 1872, 1876, 1882
		\bool_I_0:w	1912
		\bool_I_1:w	1912
Numbers			
\0	5866, 5877		
\8	7371		
\9	7372		
_	346, 1065, 6656, 6916, 7272, 7321		
A			
\A	2679, 2721, 4296, 4298		
\above	467		
\abovedisplayshortskip	480		
\abovedisplayskip	481		
\abovewithdelims	468		
\accent	518		
\adjdemerits	555		
\advance	362		
\afterassignment	372		
\aftergroup	373		
\alloc_reg:nNN	6413, 6416		
\alloc_setup_type:nnn	6411, 6414		
\AtBeginDocument	8118		
\atop	469		
\atopwithdelims	470		
B			
\B	4297, 4299		

\bool_if:cTF	41, 1898	\botmark	453
\bool_if:N	1898	\botmarks	679
\bool_if:n	1912	\box	661
\bool_if:NF	294, 1908, 2054, 2060, 3698, 7913	\box_clear:c	142, 6201
\bool_if:nF	2078, 2087	\box_clear:N	142, 6201, 6201, 6205
\bool_if:NT	1907, 2052, 2058, 3698, 3699, 7878, 8966	\box_clear_new:c	142, 6207
\bool_if:nT	2065, 2074	\box_clear_new:N	142, 6207, 6207, 6219
\bool_if:NTF	41, 1898, 1909, 3684, 6698, 7547, 8979	\box_dp:c	144, 6233
\bool_if:nTF	42, 1912, 2945, 3095, 4102, 7853, 7863	\box_dp:N	144, 6233, 6234, 6237, 6240
\bool_if_p:c	41, 1898	\box_gclear:c	142, 6201
\bool_if_p:N	41, 1898, 1906	\box_gclear:N	142, 6201, 6203, 6206
\bool_if_p:n	42, 1893, 1895, 1912, 1914, 1920, 2044, 2047	\box_gclear_new:c	142, 6207
\bool_new:c	40, 1870	\box_gclear_new:N	142, 6207, 6213, 6220
\bool_new:N	40, 1870, 1870, 1871, 1910, 1911, 6646, 7495, 7564, 7579, 8189	\box_gset_eq:cc	143, 6221
\bool_Not:N	1955, 1975	\box_gset_eq:cN	143, 6221
\bool_Not:w	1912, 1950, 1973	\box_gset_eq:Nc	143, 6221
\bool_not_choose:NN	2006, 2010	\box_gset_eq:NN	143, 6204, 6216, 6221, 6223, 6226
\bool_not_cleanup:N	1996, 1998, 2004	\box_gset_eq_clear:cc	143, 6227
\bool_not_Not:N	1968, 1984	\box_gset_eq_clear:cN	143, 6227
\bool_not_Not:w	1963, 1974	\box_gset_eq_clear:Nc	143, 6227
\bool_not_p:n	42, 2044, 2044	\box_gset_eq_clear:NN	143, 6227, 6229, 6232
\bool_p:w	1912, 1977, 1986	\box_gset_to_last:c	6282
\bool_S_0:w	1912	\box_gset_to_last:N	6282, 6284, 6287
\bool_S_1:w	1912	\box_ht:c	144, 6233
\bool_set:cn	41, 1892	\box_ht:N	144, 6233, 6233, 6236, 6242
.bool_set:N	168	\box_if_empty:cTF	145, 6275
\bool_set:Nn	41, 1892, 1892, 1896	\box_if_empty:N	6275
\bool_set_eq:cc	40, 1884, 1887	\box_if_empty:NF	6279
\bool_set_eq:cN	40, 1884, 1886	\box_if_empty:NT	6278
\bool_set_eq:Nc	40, 1884, 1885	\box_if_empty:NTF	145, 6275, 6280
\bool_set_eq:NN	40, 1884, 1884	\box_if_empty_p:c	145, 6275
\bool_set_false:c	40, 1872	\box_if_empty_p:N	145, 6275, 6277
\bool_set_false:N	40, 309, 1872, 1874, 1881, 6700, 7516, 7845, 8956, 8985	\box_if_horizontal:cTF	145, 6263
.bool_set_inverse:N	168	\box_if_horizontal:N	6263
\bool_set_true:c	40, 1872	\box_if_horizontal:NF	6269
\bool_set_true:N	40, 323, 1872, 1872, 1880, 6654, 6722, 7511, 7840, 8993	\box_if_horizontal:NT	6268
\bool_until_do:cn	43, 2051	\box_if_horizontal:NTF	145, 6263, 6270
\bool_until_do:Nn	43, 2051, 2053, 2054, 2056	\box_if_horizontal_p:c	145, 6263
\bool_until_do:nn	43, 2063, 2076, 2081	\box_if_horizontal_p:N	145, 6263, 6267
\bool_while_do:cn	43, 2051	\box_if_vertical:cTF	145, 6263
\bool_while_do:Nn	43, 2051, 2051, 2052, 2055	\box_if_vertical:N	6265
\bool_while_do:nn	43, 2063, 2063, 2068	\box_if_vertical:NF	6273
\bool_xor_p:nn	42, 2045, 2045	\box_if_vertical:NT	6272
		\box_if_vertical:NTF	145, 6263, 6274
		\box_if_vertical_p:c	145, 6263
		\box_if_vertical_p:N	145, 6263, 6271
		\box_move_down:nn	144, 6252, 6258
		\box_move_left:nn	144, 6252, 6252
		\box_move_right:nn	144, 6252, 6254

- \box_move_up:nn [144](#), [6252](#), [6256](#)
 - \box_new:c [142](#), [6193](#)
 - \box_new:N [142](#), [6193](#), [6194](#),
[6200](#), [6211](#), [6217](#), [6292](#), [6298](#), [6300](#)
 - \box_set_dp:cn [145](#), [6239](#)
 - \box_set_dp:Nn [145](#), [6239](#), [6239](#), [6246](#)
 - \box_set_eq:cc [142](#), [6221](#)
 - \box_set_eq:cN [142](#), [6221](#)
 - \box_set_eq:Nc [142](#), [6221](#)
 - \box_set_eq:NN
[142](#), [6202](#), [6210](#), [6221](#), [6221](#), [6224](#), [6225](#)
 - \box_set_eq_clear:cc [143](#), [6227](#)
 - \box_set_eq_clear:cN [143](#), [6227](#)
 - \box_set_eq_clear:Nc [143](#), [6227](#)
 - \box_set_eq_clear:NN
[143](#), [6227](#), [6227](#), [6230](#), [6231](#)
 - \box_set_ht:cn [145](#), [6239](#)
 - \box_set_ht:Nn [145](#), [6239](#), [6241](#), [6245](#)
 - \box_set_to_last:c [6282](#)
 - \box_set_to_last:N [6282](#), [6282](#), [6285](#), [6286](#)
 - \box_set_wd:cn [145](#), [6239](#)
 - \box_set_wd:Nn [145](#), [6239](#), [6243](#), [6247](#)
 - \box_show:c [146](#), [6301](#)
 - \box_show:N [146](#), [6301](#), [6301](#), [6302](#)
 - \box_use:c [143](#), [6248](#)
 - \box_use:N [143](#), [6248](#), [6249](#), [6251](#)
 - \box_use_clear:c [143](#), [6248](#)
 - \box_use_clear:N .. [143](#), [6248](#), [6248](#), [6250](#)
 - \box_wd:c [144](#), [6233](#)
 - \box_wd:N [144](#), [6233](#), [6235](#), [6238](#), [6244](#)
 - \boxmaxdepth [623](#)
 - \brokenpenalty [580](#)
- C**
- \C [1820](#), [2721](#)
 - \c_active_char_token [3136](#), [3136](#)
 - \c_alignment_tab_token [3132](#), [3132](#)
 - \c_alignment_token
[56](#), [2585](#), [2591](#), [2621](#), [3132](#)
 - \c_catcode_active_tl
[57](#), [2600](#), [2602](#), [2659](#), [3136](#)
 - \c_catcode_letter_token
[56](#), [2585](#), [2597](#), [2649](#), [3134](#)
 - \c_catcode_other_token
[56](#), [2585](#), [2598](#), [2654](#), [3135](#)
 - \c_code_cctab ... [192](#), [11582](#), [11584](#), [11585](#)
 - \c_document_cctab
[192](#), [11582](#), [11590](#), [11599](#), [11602](#)
 - \c_e_fp [189](#), [8148](#), [8148](#)
 - \c_eight [76](#), [2513](#),
[2545](#), [3760](#), [3823](#), [3828](#), [6390](#), [8957](#)
 - \c_eleven [76](#), [2519](#), [2551](#), [3823](#), [3831](#), [6393](#)
 - \c_empty_box [146](#), [6202](#),
[6204](#), [6210](#), [6216](#), [6288](#), [6289](#), [6292](#)
 - \c_empty_prop . [141](#), [5899](#), [5899-5905](#), [6018](#)
 - \c_empty_tl [108](#), [3573](#), [4194](#), [4210](#),
[4212](#), [4436](#), [4782](#), [4782](#), [5191](#), [5305](#)
 - \c_false_bool [23](#), [992](#),
[1021](#), [1061](#), [1062](#), [1090](#), [1457](#), [1459](#),
[1468](#), [1480](#), [1870](#), [1875](#), [1879](#), [1979](#),
[1990](#), [2015](#), [2018](#), [2019](#), [2021](#), [2024](#),
[2048](#), [2765](#), [3673](#), [3678](#), [3687](#), [4741](#)
 - \c_fifteen [76](#), [2527](#), [2559](#), [3823](#), [3834](#), [6397](#)
 - \c_five [76](#), [2507](#), [2539](#),
[3823](#), [3827](#), [6387](#), [8610](#), [9857](#), [10155](#)
 - \c_five_hundred_million [8131](#),
[8134](#), [8647](#), [10546](#), [10702](#), [10897](#), [10899](#)
 - \c_forty_four [8131](#), [8131](#), [8775](#)
 - \c_four [76](#), [2505](#),
[2537](#), [3823](#), [3826](#), [6386](#), [8756](#), [8992](#)
 - \c_fourteen [76](#), [2525](#), [2557](#), [3823](#), [3833](#), [6396](#)
 - \c_fp_exp-100_tl [10300](#)
 - \c_fp_exp-10_tl [10300](#)
 - \c_fp_exp-1_tl [10300](#)
 - \c_fp_exp-200_tl [10300](#)
 - \c_fp_exp-20_tl [10300](#)
 - \c_fp_exp-2_tl [10300](#)
 - \c_fp_exp-30_tl [10300](#)
 - \c_fp_exp-3_tl [10300](#)
 - \c_fp_exp-40_tl [10300](#)
 - \c_fp_exp-4_tl [10300](#)
 - \c_fp_exp-50_tl [10300](#)
 - \c_fp_exp-5_tl [10300](#)
 - \c_fp_exp-60_tl [10300](#)
 - \c_fp_exp-6_tl [10300](#)
 - \c_fp_exp-70_tl [10300](#)
 - \c_fp_exp-7_tl [10300](#)
 - \c_fp_exp-80_tl [10300](#)
 - \c_fp_exp-8_tl [10300](#)
 - \c_fp_exp-90_tl [10300](#)
 - \c_fp_exp-9_tl [10300](#)
 - \c_fp_exp_100_tl [10280](#)
 - \c_fp_exp_10_tl [10280](#)
 - \c_fp_exp_1_tl [10280](#)
 - \c_fp_exp_200_tl [10280](#)
 - \c_fp_exp_20_tl [10280](#)
 - \c_fp_exp_2_tl [10280](#)
 - \c_fp_exp_30_tl [10280](#)
 - \c_fp_exp_3_tl [10280](#)

- \c_fp_exp_40_tl 10280
- \c_fp_exp_4_tl 10280
- \c_fp_exp_50_tl 10280
- \c_fp_exp_5_tl 10280
- \c_fp_exp_60_tl 10280
- \c_fp_exp_6_tl 10280
- \c_fp_exp_70_tl 10280
- \c_fp_exp_7_tl 10280
- \c_fp_exp_80_tl 10280
- \c_fp_exp_8_tl 10280
- \c_fp_exp_90_tl 10280
- \c_fp_exp_9_tl 10280
- \c_fp_ln_10_1_tl 10626
- \c_fp_ln_10_2_tl 10626
- \c_fp_ln_10_3_tl 10626
- \c_fp_ln_10_4_tl 10626
- \c_fp_ln_10_5_tl 10626
- \c_fp_ln_10_6_tl 10626
- \c_fp_ln_10_7_tl 10626
- \c_fp_ln_10_8_tl 10626
- \c_fp_ln_10_9_tl 10626
- \c_fp_ln_2_1_tl 10635
- \c_fp_ln_2_2_tl 10635
- \c_fp_ln_2_3_tl 10635
- \c_fp_pi_by_four_decimal_int . 8136,
8136, 8137, 9795, 9808, 9815, 9819
- \c_fp_pi_by_four_extended_int
.. 8136, 8138, 8139, 9795, 9808, 9820
- \c_fp_pi_decimal_int 8136, 8140, 8141, 9735
- \c_fp_pi_extended_int .. 8136, 8142, 8143
- \c_fp_two_pi_decimal_int
..... 8136, 8144, 8145, 9731, 9737
- \c_fp_two_pi_extended_int
..... 8136, 8146, 8147, 9731, 9737
- \c_group_begin_token
..... 56, 2585, 2585, 2606,
2947, 3097, 4689, 4723, 6315, 6355
- \c_group_end_token 56, 2585, 2586, 2611,
2948, 3098, 6320, 6321, 6360, 6361
- \c_initex_cctab 192, 11582, 11591
- \c_ior_log_stream 6376, 6379
- \c_ior_streams_tl 6380, 6399, 6428
- \c_ior_term_stream 6376, 6377
- \c_iow_log_stream 6376, 6378, 6628, 6629
- \c_iow_streams_tl 6380, 6380, 6399, 6441
- \c_iow_term_stream 6376, 6376, 6630, 6631
- \c_job_name_tl 107, 4770, 4781
- \c_keys_code_root_tl 7486, 7486, 7658,
7663, 7919, 7921, 7933, 7939, 7944
- \c_keys_props_root_tl
7488, 7488, 7521, 7551, 7558, 7728,
7730, 7732, 7734, 7736, 7738, 7740,
7742, 7744, 7746, 7748, 7750, 7752,
7754, 7756, 7758, 7760, 7762, 7764,
7766, 7768, 7770, 7772, 7774, 7776,
7778, 7780, 7782, 7784, 7786, 7788,
7790, 7792, 7794, 7796, 7798, 7800,
7802, 7804, 7806, 7808, 7810, 7812
- \c_keys_value_forbidden_tl .. 7489, 7489
- \c_keys_value_required_tl ... 7489, 7490
- \c_keys_vars_root_tl 7486, 7487, 7621,
7640, 7647, 7650, 7652, 7667–7669,
7672, 7715, 7880, 7882, 7885, 7893
- \c_letter_token 3132, 3134
- \c_luatex_is_engine_bool
..... 24, 1448, 1457, 1479
- \c_math_shift_token 3132, 3133
- \c_math_subscript_token
..... 56, 2585, 2595, 2639
- \c_math_superscript_token
..... 56, 2585, 2593, 2634
- \c_math_toggle_token
..... 56, 2585, 2589, 2616, 3133
- \c_max_dim 84, 4043, 4045, 4046, 4050, 4125
- \c_max_int 76, 3841, 3841
- \c_max_register_int
..... 76, 1168, 1168, 3282, 11518
- \c_max_skip 88, 4124, 4125
- \c_minus_one 76,
1156, 1157, 1160, 1161, 1170, 3280,
3324, 3823, 4314, 4315, 4341, 4342,
4354, 4355, 6378, 6379, 6546, 6558,
8206, 8314, 8743, 8997, 8998, 9135,
9170, 9413, 9461, 9465, 9630, 9754,
9758, 10013, 10028, 10116, 10120,
10193, 10201, 10609, 10613, 10737
- \c_msg_coding_error_text_tl
..... 6819, 6819,
7229, 7238, 7260, 7268, 7277, 7284,
7291, 7298, 7957, 7964, 7985, 7992
- \c_msg_continue_text_tl 6819, 6824, 6886
- \c_msg_critical_text_tl 6819, 6826, 7005
- \c_msg_fatal_text_tl 6819, 6828, 6994, 7136
- \c_msg_help_text_tl 6819, 6830, 6894
- \c_msg_hide_tl 6860, 6862–6864, 6931
- \c_msg_kernel_bug_more_text_tl
..... 7302, 7309, 7313
- \c_msg_kernel_bug_text_tl 7302, 7304, 7311

- \c_msg_more_text_prefix_tl [6788](#), [6789](#),
[6805](#), [6814](#), [7010](#), [7018](#), [7149](#), [7159](#)
- \c_msg_no_info_text_tl . [6819](#), [6832](#), [6884](#)
- \c_msg_on_line_text_tl [6837](#), [6856](#)
- \c_msg_on_line_tl [6819](#)
- \c_msg_return_text_tl [159](#), [6819](#), [6835](#),
[6838](#), [7233](#), [7241](#), [7248](#), [7255](#), [7317](#)
- \c_msg_text_prefix_tl
[6788](#), [6788](#), [6792](#), [6803](#), [6812](#), [6991](#),
[7002](#), [7015](#), [7024](#), [7035](#), [7043](#), [7049](#),
[7079](#), [7132](#), [7154](#), [7167](#), [7190](#), [7212](#)
- \c_msg_trouble_text_tl . [159](#), [6819](#), [6845](#)
- \c_nine [76](#), [2515](#),
[2547](#), [3823](#), [3829](#), [6391](#), [8319](#), [9667](#),
[9889](#), [9975](#), [10221](#), [10230](#), [10449](#), [10741](#)
- \c_one [76](#), [2499](#), [2531](#), [3322](#), [3823](#),
[3823](#), [6383](#), [8208](#), [8219](#), [8278](#), [8290](#),
[8338](#), [8344](#), [8386](#), [8411](#), [8608](#), [8967](#),
[8971](#), [8973](#), [8980](#), [9120](#), [9149](#), [9409](#),
[9446](#), [9451](#), [9472](#), [9595](#), [9663](#), [9706](#),
[9720](#), [9771](#), [9786](#), [9811](#), [9823](#), [9884](#),
[9970](#), [10018](#), [10025](#), [10040](#), [10066](#),
[10088](#), [10093](#), [10101](#), [10107](#), [10195](#),
[10199](#), [10401](#), [10411](#), [10512](#), [10537](#),
[10548](#), [10552](#), [10574](#), [10594](#), [10600](#),
[10704](#), [10708](#), [10739](#), [10756](#), [10808](#),
[10831](#), [10837](#), [10838](#), [10882](#), [10900](#),
[10938](#), [10959](#), [10965](#), [11119](#), [11121](#)
- \c_one_fp [189](#), [8149](#), [8149](#), [11018](#)
- \c_one_hundred
. [76](#), [3838](#), [3838](#), [8341](#), [8342](#), [10410](#)
- \c_one_hundred_million
. [8131](#), [8133](#), [9330](#), [10252](#)
- \c_one_million [8131](#), [8132](#), [9593](#)
- \c_one_thousand
[76](#), [3838](#), [3839](#), [9240](#), [9496](#), [9540](#), [9591](#)
- \c_one_thousand_million
. [8131](#), [8135](#), [8281](#), [8303](#), [8320](#),
[8330](#), [8366](#), [8377](#), [8391](#), [8402](#), [8443](#),
[8485](#), [8759](#), [8778](#), [8897](#), [8933](#), [8959](#),
[9010](#), [9035](#), [9101](#), [9118](#), [9121](#), [9136](#),
[9145](#), [9222](#), [9261](#), [9269](#), [9278](#), [9365](#),
[9378](#), [9414](#), [9444](#), [9447](#), [9449](#), [9452](#),
[9462](#), [9466](#), [9473](#), [9474](#), [9594](#), [9596](#),
[9608](#), [9620](#), [9635](#), [9668](#), [9679](#), [9697](#),
[9755](#), [9759](#), [9775](#), [9779](#), [9853](#), [9908](#),
[9950](#), [9994](#), [10045](#), [10051](#), [10099](#),
[10103](#), [10105](#), [10109](#), [10117](#), [10121](#),
[10151](#), [10275](#), [10345](#), [10520](#), [10530](#),
[10549](#), [10567](#), [10592](#), [10596](#), [10598](#),
[10602](#), [10610](#), [10614](#), [10684](#), [10705](#),
[10727](#), [10779](#), [10846](#), [10849](#), [10918](#),
[10957](#), [10961](#), [10963](#), [10967](#), [11111](#)
- \c_other_cctab
. [192](#), [11582](#), [11592](#), [11600](#), [11610](#)
- \c_other_char_token [3132](#), [3135](#)
- \c_parameter_token
. [56](#), [2585](#), [2592](#), [2625](#), [2628](#)
- \c_pdftex_is_engine_bool
. [24](#), [1448](#), [1458](#), [1468](#), [1480](#)
- \c_pi_fp [189](#), [8150](#), [8150](#)
- \c_seven [76](#), [1156](#),
[1166](#), [2261](#), [2511](#), [2543](#), [3823](#), [6389](#)
- \c_six [76](#), [1156](#), [1165](#), [2258](#),
[2509](#), [2541](#), [3823](#), [6388](#), [9731](#), [9737](#)
- \c_sixteen
. [76](#), [1156](#), [1163](#), [1172](#), [3758](#), [3823](#),
[6376](#), [6377](#), [6411](#), [6414](#), [6427](#), [6429](#),
[6440](#), [6442](#), [6475](#), [6494](#), [6512](#), [6531](#)
- \c_space_tl [108](#), [4783](#), [4783](#),
[4845](#), [5281](#), [5287](#), [5298](#), [5795](#), [5801](#),
[5812](#), [6102](#), [6108](#), [6119](#), [6121](#), [6585](#),
[6587](#), [6655](#), [6656](#), [6668](#), [6669](#), [6857](#)
- \c_space_token [56](#), [2585](#), [2596](#),
[2644](#), [2949](#), [2968](#), [3099](#), [4690](#), [4724](#)
- \c_string_cctab
. [192](#), [11582](#), [11593](#), [11601](#), [11615](#)
- \c_ten [76](#), [2517](#), [2549](#),
[3598](#), [3823](#), [3830](#), [6392](#), [8331](#), [8378](#),
[8403](#), [8555](#), [8619](#), [8675](#), [8777](#), [8782](#),
[8894](#), [8930](#), [8969](#), [8972](#), [8982](#), [9377](#),
[9431](#), [9607](#), [9656](#), [9698](#), [9710](#), [9788](#),
[10180](#), [10801](#), [11034](#), [11068](#), [11073](#)
- \c_ten_thousand [76](#), [3838](#), [3840](#)
- \c_thirteen [76](#), [2523](#), [2555](#), [3823](#), [3832](#), [6395](#)
- \c_thirty_two [76](#), [3835](#), [3835](#)
- \c_three [76](#), [2503](#), [2535](#),
[3823](#), [3825](#), [6385](#), [9729](#), [10037](#), [10370](#)
- \c_tl_act_lowercase_tl . [4910](#), [4915](#), [4923](#)
- \c_tl_act_uppercase_tl . [4910](#), [4910](#), [4921](#)
- \c_tl_rescan_marker_tl
. [4295](#), [4303](#), [4313](#), [4325](#), [4353](#)
- \c_token_A_int [2810](#), [2845](#)
- \c_true_bool [23](#),
[992](#), [1021](#), [1061](#), [1061](#), [1094](#), [1458](#),
[1469](#), [1479](#), [1873](#), [1877](#), [1978](#), [1981](#),
[1987](#), [1988](#), [2016](#), [2017](#), [2020](#), [2022](#),
[2026](#), [2049](#), [3673](#), [3678](#), [3691](#), [4743](#)
- \c_twelve [76](#), [1156](#), [1167](#), [2266](#),
[2280](#), [2521](#), [2553](#), [2722](#), [3823](#), [6394](#)

- `\c_two` 76, 2501, 2533,
3756, [3823](#), 3824, 6384, 8746, 9734,
10012, 10016, 10035, 10069, 10192,
10198, 10851–10853, 10866, 10942
- `\c_two_hundred_fifty_five` 76, [3836](#), 3836
- `\c_two_hundred_fifty_six` . 76, [3836](#), 3837
- `\c_undefined:D` 1297, 1299
- `\c_undefined_fp` [189](#), [8151](#), 8151, 9308,
10208, 11001, 11051, 11058, 11178
- `\c_xetex_is_engine_bool`
..... [24](#), [1448](#), 1459, 1469
- `\c_zero` 76, 991, 999, 1007,
1014, 1020, 1028, 1036, 1043, [1156](#),
1164, 1486, 1491, 1567, 1576, 2165–
2175, 2253, 2255, 2260, 2462, 2471,
2497, 2529, 2693, 3211, 3241, 3245,
3246, 3252, 3299, 3300, 3571, [3823](#),
4094, 4104, 4105, 4755, 4792, 4837,
5387, 5400, 5839, 5849, 6382, 6411,
6414, 6771, 6773, 7335, 8228, 8239,
8277, 8289, 8291, 8302, 8345–8347,
8449, 8491, 8534, 8537, 8599, 8666,
8801–8809, 8840–8848, 8903, 8939,
8983, 9038, 9076, 9092, 9134, 9138,
9140, 9206, 9210, 9235, 9304, 9314,
9327, 9328, 9349, 9353, 9374, 9383,
9384, 9412, 9460, 9464, 9468, 9469,
9488, 9532, 9606, 9634, 9645, 9653,
9711, 9714, 9721–9723, 9753, 9757,
9761, 9766, 9789, 9790, 9795, 9804,
9808, 9819, 9844, 9885, 9899, 9941,
9971, 9985, 10011, 10024, 10026,
10038, 10039, 10041, 10042, 10044,
10046, 10049, 10060–10062, 10094,
10095, 10115, 10119, 10142, 10191,
10205, 10206, 10236, 10237, 10249,
10250, 10266, 10333, 10336, 10372,
10398, 10402–10404, 10412–10414,
10426, 10459, 10477, 10478, 10510,
10511, 10516, 10517, 10522, 10551,
10587, 10588, 10608, 10612, 10651,
10655, 10707, 10718, 10735, 10745–
10748, 10764, 10790, 10798, 10812,
10813, 10815, 10818, 10864, 10865,
10868, 10873, 10875, 10887, 10904,
10911, 10917, 10927, 10935, 10950,
10993, 10997, 11014, 11029, 11033,
11040, 11065, 11070, 11072, 11122,
11123, 11151, 11169–11172, 11259,
11262, 11266, 11277, 11278, 11300
- `\c_zero_dim`
84, 3906, [4043](#), 4044, 4049, 4124, 6337
- `\c_zero_fp`
189, [8152](#), 8152, 8417, 8427, 8429,
9318, 10184, 10239, 10362, 10389,
10429, 10661, 10669, 11007, 11186
- `\c_zero_muskip` 4140
- `\c_zero_skip`
88, 4065, [4124](#), 4124, 4175, 4176, 6324
- `\catcode` 3–
6, 9, 70–78, 84–91, 101, 281–288, 665
- `\catcodetable` 755
- `\CatcodeTableIniTeX` 11591
- `\CatcodeTableLaTeX` 11590
- `\CatcodeTableOther` 11592
- `\CatcodeTableString` 11593
- `\cctab_begin:N`
.... [192](#), [11542](#), 11542, 11562, 11568
- `\cctab_end:` [192](#), [11542](#), 11552, 11563, 11569
- `\cctab_gset:Nn` [192](#), [11573](#), 11573,
11581, 11585, 11602, 11610, 11615
- `\cctab_new:N` [192](#), [11512](#), 11512,
11531, 11535, 11584, 11599–11601
- `\char` 519
- `\char_active_gset:Npn` 65, 3088
- `\char_active_gset:Npx` 65, 3089
- `\char_active_gset:Nn` . 66, [3074](#), 3091
- `\char_active_set:Npn` 65, [3074](#), 3086
- `\char_active_set:Npx` 65, [3074](#), 3087
- `\char_active_set_eq:Nn` .. 65, [3074](#), 3090
- `\char_make_active:N` [3137](#), 3152
- `\char_make_active:n` [3137](#), 3170
- `\char_make_alignment:N` [3137](#)
- `\char_make_alignment:n` [3137](#)
- `\char_make_alignment_tab:N` 3141
- `\char_make_alignment_tab:n` 3159
- `\char_make_begin_group:N` 3138
- `\char_make_begin_group:n` 3156
- `\char_make_comment:N` [3137](#), 3153
- `\char_make_comment:n` [3137](#), 3171
- `\char_make_end_group:N` 3139
- `\char_make_end_group:n` 3157
- `\char_make_end_line:N` [3137](#), 3142
- `\char_make_end_line:n` [3137](#), 3160
- `\char_make_escape:N` [3137](#), 3137
- `\char_make_escape:n` [3137](#), 3155
- `\char_make_group_begin:N` [3137](#)
- `\char_make_group_begin:n` [3137](#)
- `\char_make_group_end:N` [3137](#)
- `\char_make_group_end:n` [3137](#)

- \char_make_ignore:N 3137, 3148
- \char_make_ignore:n 3137, 3166
- \char_make_invalid:N 3137, 3154
- \char_make_invalid:n 3137, 3172
- \char_make_letter:N 3137, 3150
- \char_make_letter:n 3137, 3168
- \char_make_math_shift:N 3140
- \char_make_math_shift:n 3158
- \char_make_math_subscript:N . 3137, 3146
- \char_make_math_subscript:n . 3137, 3164
- \char_make_math_superscript:N 3137, 3144
- \char_make_math_superscript:n 3137, 3162
- \char_make_math_toggle:N 3137
- \char_make_math_toggle:n 3137
- \char_make_other:N 3137, 3151
- \char_make_other:n 3137, 3169
- \char_make_parameter:N 3137, 3143
- \char_make_parameter:n 3137, 3161
- \char_make_space:N 3137, 3149
- \char_make_space:n 3137, 3167
- \char_set_catcode:nn 54, 298–306, 2490, 2490, 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, 2559, 2722
- \char_set_catcode:w 3110, 3110, 3115, 3117
- \char_set_catcode_active:N 53, 2496, 2522, 2601, 3075, 3152, 6918
- \char_set_catcode_active:n . 53, 2528, 2554, 3080, 3170, 7369, 7370, 11608
- \char_set_catcode_alignment:N 53, 2496, 2504, 2590, 3141
- \char_set_catcode_alignment:n 53, 316, 2528, 2536, 3159
- \char_set_catcode_comment:N 53, 2496, 2524, 3153
- \char_set_catcode_comment:n 53, 2528, 2556, 3171
- \char_set_catcode_end_line:N 53, 2496, 2506, 3142
- \char_set_catcode_end_line:n 53, 2528, 2538, 3160
- \char_set_catcode_escape:N 53, 2496, 2496, 3137
- \char_set_catcode_escape:n 53, 2528, 2528, 3155
- \char_set_catcode_group_begin:N 53, 2496, 2498, 3138
- \char_set_catcode_group_begin:n 53, 2528, 2530, 3156
- \char_set_catcode_group_end:N 53, 2496, 2500, 3139
- \char_set_catcode_group_end:n 53, 2528, 2532, 3157
- \char_set_catcode_ignore:N 53, 2496, 2514, 3148
- \char_set_catcode_ignore:n 53, 313, 314, 2528, 2546, 3166, 8217
- \char_set_catcode_invalid:N 53, 2496, 2526, 3154
- \char_set_catcode_invalid:n 53, 2528, 2558, 3172
- \char_set_catcode_letter:N 53, 2496, 2518, 3150, 6860
- \char_set_catcode_letter:n 53, 317, 319, 2528, 2550, 3168
- \char_set_catcode_math_subscript:N . 53, 2496, 2512, 2594, 3147
- \char_set_catcode_math_subscript:n . 53, 2528, 2544, 3165, 11607
- \char_set_catcode_math_superscript:N 53, 2496, 2510, 3145, 7320
- \char_set_catcode_math_superscript:n 53, 318, 2528, 2542, 3163
- \char_set_catcode_math_toggle:N 53, 2496, 2502, 2588, 3140
- \char_set_catcode_math_toggle:n 53, 2528, 2534, 3158, 7423, 7461
- \char_set_catcode_other:N .. 53, 2496, 2520, 2678, 2679, 2812, 3151, 6865
- \char_set_catcode_other:n 53, 315, 320, 2528, 2552, 3169, 11606, 11613, 11618
- \char_set_catcode_parameter:N 53, 2496, 2508, 3143
- \char_set_catcode_parameter:n 53, 2528, 2540, 3161
- \char_set_catcode_space:N 53, 2496, 2516, 3149
- \char_set_catcode_space:n .. 53, 321, 2528, 2548, 3167, 11604, 11605, 11619
- \char_set_lccode:nn 54, 2560, 2566, 2680–2682, 2715–2720, 2813–2815, 3082, 6916, 6917, 7321–7324, 7371, 7372
- \char_set_lccode:w 3110, 3112, 3121, 3123
- \char_set_mathcode:nn ... 55, 2560, 2560
- \char_set_mathcode:w 3110, 3111, 3118, 3120
- \char_set_sfcode:nn 56, 2560, 2578

`\char_set_sfcode:w` [3110](#), [3114](#), [3127](#), [3129](#)
`\char_set_uccode:nn` [55](#), [2560](#), [2572](#)
`\char_set_uccode:w` [3110](#), [3113](#), [3124](#), [3126](#)
`\char_show_value_catcode:n` [54](#), [2490](#), [2494](#)
`\char_show_value_catcode:w` [3115](#), [3116](#)
`\char_show_value_lccode:n` [54](#), [2560](#), [2570](#)
`\char_show_value_lccode:w` [3115](#), [3122](#)
`\char_show_value_mathcode:n` [55](#), [2560](#), [2564](#)
`\char_show_value_mathcode:w` [3115](#), [3119](#)
`\char_show_value_sfcode:n` [56](#), [2560](#), [2582](#)
`\char_show_value_sfcode:w` [3115](#), [3128](#)
`\char_show_value_uccode:n` [55](#), [2560](#), [2576](#)
`\char_show_value_uccode:w` [3115](#), [3125](#)
`\char_tmp:NN` [3076](#), [3086](#)–[3091](#)
`\char_value_catcode:n` [54](#), [298](#)–[306](#), [2490](#), [2492](#)
`\char_value_catcode:w` [3115](#), [3115](#)
`\char_value_lccode:n` [54](#), [2560](#), [2568](#)
`\char_value_lccode:w` [3115](#), [3121](#)
`\char_value_mathcode:n` [55](#), [2560](#), [2562](#)
`\char_value_mathcode:w` [3115](#), [3118](#)
`\char_value_sfcode:n` [56](#), [2560](#), [2580](#)
`\char_value_sfcode:w` [3115](#), [3127](#)
`\char_value_uccode:n` [55](#), [2560](#), [2574](#)
`\char_value_uccode:w` [3115](#), [3124](#)
`\chardef` [80](#), [93](#), [96](#), [328](#), [354](#)
`\chk_if_exist_cs:c` [26](#), [1215](#), [1223](#)
`\chk_if_exist_cs:N` [26](#), [1215](#), [1215](#), [1224](#), [1770](#)
`\chk_if_free_cs:c` [26](#), [1192](#), [1213](#)
`\chk_if_free_cs:N` [26](#), [1192](#), [1192](#), [1202](#),
[1214](#), [1229](#), [1286](#), [3273](#), [3288](#), [3901](#),
[4060](#), [4134](#), [4193](#), [4199](#), [4204](#), [6196](#)
`.choice:` [169](#)
`.choice_code:n` [169](#)
`.choice_code:x` [169](#)
`.choices:nn` [169](#)
`\cleaders` [537](#)
`\clist_clear:c` [123](#), [5526](#), [5527](#)
`\clist_clear:N` [123](#), [5526](#), [5526](#), [5631](#), [5649](#), [5862](#), [7829](#)
`\clist_clear_new:c` [123](#), [5530](#), [5531](#)
`\clist_clear_new:N` [123](#), [5530](#), [5530](#)
`\clist_concat:ccc` [123](#), [5542](#)
`\clist_concat:NNN` [123](#), [5542](#), [5542](#), [5562](#)
`\clist_concat_aux:NNNN` [5542](#), [5543](#), [5545](#), [5546](#)
`\clist_display:c` [5889](#), [5890](#)
`\clist_display:N` [5889](#), [5889](#)
`\clist_gclear:c` [123](#), [5526](#), [5529](#)
`\clist_gclear:N` [123](#), [5526](#), [5528](#), [5873](#)
`\clist_gclear_new:c` [123](#), [5530](#), [5533](#)
`\clist_gclear_new:N` [123](#), [5530](#), [5532](#)
`\clist_gconcat:ccc` [124](#), [5542](#)
`\clist_gconcat:NNN` [124](#), [5542](#), [5544](#), [5563](#)
`\clist_get:cN` [125](#), [126](#), [131](#), [5586](#), [5886](#)
`\clist_get:NN` [125](#), [126](#), [131](#), [5586](#), [5586](#), [5590](#), [5885](#)
`\clist_get_aux:wN` [5586](#), [5587](#), [5588](#)
`\clist_gpop:cN` [126](#), [131](#), [5591](#)
`\clist_gpop:NN` [126](#), [131](#), [5591](#), [5593](#), [5605](#)
`\clist_gpush:cn` [126](#), [132](#), [5606](#), [5618](#)
`\clist_gpush:co` [126](#), [132](#), [5606](#), [5620](#)
`\clist_gpush:cV` [126](#), [132](#), [5606](#), [5619](#)
`\clist_gpush:cx` [126](#), [132](#), [5606](#), [5621](#)
`\clist_gpush:Nn` [126](#), [132](#), [5606](#), [5614](#)
`\clist_gpush:No` [126](#), [132](#), [5606](#), [5616](#)
`\clist_gpush:NV` [126](#), [132](#), [5606](#), [5615](#)
`\clist_gpush:Nx` [126](#), [132](#), [5606](#), [5617](#)
`\clist_gput_left:cn` [125](#), [5564](#), [5618](#)
`\clist_gput_left:co` [125](#), [5564](#), [5620](#)
`\clist_gput_left:cV` [125](#), [5564](#), [5619](#)
`\clist_gput_left:cx` [125](#), [5564](#), [5621](#)
`\clist_gput_left:Nn` [125](#), [5564](#), [5566](#), [5576](#), [5577](#), [5614](#)
`\clist_gput_left:No` [125](#), [5564](#), [5616](#)
`\clist_gput_left:NV` [125](#), [5564](#), [5615](#)
`\clist_gput_left:Nx` [125](#), [5564](#), [5617](#)
`\clist_gput_right:cn` [125](#), [5578](#)
`\clist_gput_right:co` [125](#), [5578](#)
`\clist_gput_right:cV` [125](#), [5578](#)
`\clist_gput_right:cx` [125](#), [5578](#)
`\clist_gput_right:Nn` [125](#), [5578](#), [5580](#), [5584](#), [5585](#)
`\clist_gput_right:No` [125](#), [5578](#)
`\clist_gput_right:NV` [125](#), [5578](#)
`\clist_gput_right:Nx` [125](#), [5578](#)
`\clist_gremove_all:cn` [128](#), [5641](#)
`\clist_gremove_all:Nn` [128](#), [5641](#), [5643](#), [5670](#), [5888](#)
`\clist_gremove_duplicates:c` [127](#), [5625](#)
`\clist_gremove_duplicates:N` [127](#), [5625](#), [5627](#), [5640](#)
`\clist_gremove_element:Nn` [5887](#), [5888](#)
`\clist_gset_eq:cc` [123](#), [5534](#), [5541](#)
`\clist_gset_eq:cN` [123](#), [5534](#), [5540](#)
`\clist_gset_eq:Nc` [123](#), [5534](#), [5539](#)
`\clist_gset_eq:NN` [123](#), [5534](#), [5538](#), [5628](#), [5644](#)

\clist_gset_from_seq:cc [133](#), [5859](#)
\clist_gset_from_seq:cN [133](#), [5859](#)
\clist_gset_from_seq:Nc [133](#), [5859](#)
\clist_gset_from_seq:NN
..... [133](#), [5859](#), [5870](#), [5883](#), [5884](#)
\clist_gtrim_spaces:c [128](#), [5671](#)
\clist_gtrim_spaces:N [128](#), [5671](#), [5681](#), [5684](#)
\clist_if_empty:c [5687](#)
\clist_if_empty:cTF [128](#), [5686](#)
\clist_if_empty:N [5686](#)
\clist_if_empty:NF [5554](#), [5647](#), [5729](#)
\clist_if_empty:NTF
..... [128](#), [5550](#), [5570](#), [5686](#), [5793](#)
\clist_if_empty_p:c [128](#), [5686](#)
\clist_if_empty_p:N [128](#), [5686](#)
\clist_if_eq:cc [5691](#)
\clist_if_eq:ccTF [129](#), [5688](#)
\clist_if_eq:cN [5690](#)
\clist_if_eq:cNTF [129](#), [5688](#)
\clist_if_eq:Nc [5689](#)
\clist_if_eq:NcTF [129](#), [5688](#)
\clist_if_eq:NN [5688](#)
\clist_if_eq:NNTF [129](#), [5688](#)
\clist_if_eq_p:cc [129](#), [5688](#)
\clist_if_eq_p:cN [129](#), [5688](#)
\clist_if_eq_p:Nc [129](#), [5688](#)
\clist_if_eq_p:NN [129](#), [5688](#)
\clist_if_in:cnTF [129](#), [5692](#)
\clist_if_in:coTF [129](#), [5692](#)
\clist_if_in:cVTF [129](#), [5692](#)
\clist_if_in:Nn [5692](#)
\clist_if_in:nn [5705](#)
\clist_if_in:NnF [5634](#), [5720](#), [5721](#)
\clist_if_in:nnF [5725](#)
\clist_if_in:NnT [5718](#), [5719](#)
\clist_if_in:nnT [5724](#)
\clist_if_in:NnTF . [129](#), [5692](#), [5722](#), [5723](#)
\clist_if_in:nnTF [129](#), [5726](#)
\clist_if_in:NoTF [129](#), [5692](#)
\clist_if_in:noTF [129](#)
\clist_if_in:NVTF [129](#), [5692](#)
\clist_if_in:nVTF [129](#)
\clist_item:cn [133](#), [5835](#)
\clist_item:Nn [133](#), [5835](#), [5835](#), [5858](#)
\clist_item:nn [133](#), [5835](#), [5836](#), [5837](#)
\clist_item_aux:nnn [5835](#)
\clist_item_aux:nw [5841](#), [5845](#), [5847](#), [5853](#)
\clist_length:c [132](#), [5817](#)
\clist_length:N ... [132](#), [5817](#), [5817](#), [5834](#)
\clist_length:n ... [132](#), [5817](#), [5825](#), [5842](#)
\clist_length_aux:n [5817](#), [5822](#), [5830](#), [5833](#)
\clist_map_break: [130](#), [5789](#), [5789](#)
\clist_map_break:n [130](#), [5789](#), [5790](#)
\clist_map_function:cN [129](#), [5727](#)
\clist_map_function:NN [129](#), [5436](#), [5446](#),
[5727](#), [5727](#), [5749](#), [5757](#), [5805](#), [5822](#)
\clist_map_function:nN
..... [129](#), [5441](#), [5451](#), [5674](#),
[5727](#), [5735](#), [5767](#), [5830](#), [7625](#), [7685](#)
\clist_map_function_aux:Nw
..... [5727](#), [5731](#), [5739](#), [5743](#), [5747](#)
\clist_map_inline:cn [129](#), [5751](#)
\clist_map_inline:Nn [129](#), [5632](#),
[5751](#), [5751](#), [5771](#), [5774](#), [8060](#), [8120](#)
\clist_map_inline:nn
... [129](#), [5751](#), [5761](#), [5782](#), [7606](#), [7700](#)
\clist_map_variable:cNn [130](#), [5772](#)
\clist_map_variable:NNn
..... [130](#), [5772](#), [5772](#), [5788](#)
\clist_map_variable:nNn [130](#), [5780](#)
\clist_new:c [122](#), [5524](#), [5525](#)
\clist_new:N [122](#), [5524](#), [5524](#), [5624](#)
\clist_pop:cN [126](#), [131](#), [5591](#)
\clist_pop:NN .. [126](#), [131](#), [5591](#), [5591](#), [5604](#)
\clist_pop_aux:NNN [5591](#), [5592](#), [5594](#), [5595](#)
\clist_pop_aux:wNNN [5591](#), [5596](#), [5597](#)
\clist_push:cn [126](#), [132](#), [5606](#), [5610](#)
\clist_push:co [126](#), [132](#), [5606](#), [5612](#)
\clist_push:cV [126](#), [132](#), [5606](#), [5611](#)
\clist_push:cx [126](#), [132](#), [5606](#), [5613](#)
\clist_push:Nn [126](#), [132](#), [5606](#), [5606](#)
\clist_push:No [126](#), [132](#), [5606](#), [5608](#)
\clist_push:NV [126](#), [132](#), [5606](#), [5607](#)
\clist_push:Nx [126](#), [132](#), [5606](#), [5609](#)
\clist_put_aux:NNnnNn
..... [5565](#), [5567](#), [5568](#), [5579](#), [5581](#)
\clist_put_left:cn [124](#), [5564](#), [5610](#)
\clist_put_left:co [124](#), [5564](#), [5612](#)
\clist_put_left:cV [124](#), [5564](#), [5611](#)
\clist_put_left:cx [124](#), [5564](#), [5613](#)
\clist_put_left:Nn
... [124](#), [5564](#), [5564](#), [5574](#), [5575](#), [5606](#)
\clist_put_left:No [124](#), [5564](#), [5608](#)
\clist_put_left:NV [124](#), [5564](#), [5607](#)
\clist_put_left:Nx [124](#), [5564](#), [5609](#)
\clist_put_right:cn [125](#), [5578](#)
\clist_put_right:co [125](#), [5578](#)
\clist_put_right:cV [125](#), [5578](#)
\clist_put_right:cx [125](#), [5578](#)

\clist_put_right:Nncode:n	169
... 125, 5578, 5578, 5582, 5583, 5635	.code:x	169
\clist_put_right:No	\copy	605
\clist_put_right:Nv	\count	656
\clist_put_right:Nx 125, 5578, 7910	\countdef	355
\clist_remove_all:cn	\cr	380
\clist_remove_all:Nn	\crr	381
... 127, 5641, 5641, 5669, 5887	\cs:w	17, 805,
\clist_remove_all_aux:NNn	807, 820, 852, 1120, 1148, 1361,	
... 5641, 5642, 5644, 5645	1393, 1531, 1570, 1584, 1586, 1588,	
\clist_remove_all_aux:w	1592–1594, 1629, 1635, 1655, 1657,	
... 5641, 5650, 5655, 5657, 5668	1662, 1669, 1670, 1728, 1767, 2144,	
\clist_remove_duplicates:c ... 127, 5625	2146, 3339, 4570, 4955, 10499, 10859	
\clist_remove_duplicates:N	\cs_end:	17, 805,
... 127, 5625, 5625, 5639	808, 820, 852, 1114, 1120, 1142,	
\clist_remove_duplicates_aux:NN	1148, 1361, 1393, 1531, 1570, 1584,	
... 5625, 5626, 5628, 5629	1586, 1588, 1592–1594, 1629, 1635,	
\clist_remove_element:Nn 5887, 5887	1655, 1657, 1662, 1669, 1670, 1728,	
\clist_set_eq:cc	1767, 2141, 2147–2150, 2152, 2154,	
\clist_set_eq:cN	2156, 2158, 2160, 2162, 2164, 3339,	
\clist_set_eq:Nc	4569, 4570, 4955, 9866, 9954, 10164,	
\clist_set_eq:NN	10349, 10359, 10499, 10688, 10859	
... 123, 5534, 5534, 5626, 5642, 7834	\cs_generate_from_arg_count:cNnn ...	
\clist_set_from_seq:cc	15, 1018, 1026, 1034, 1042, 1349, 1392	
\clist_set_from_seq:cN	\cs_generate_from_arg_count:NNnn ...	
\clist_set_from_seq:Nc 15, 1318, 1318, 1350, 1360	
\clist_set_from_seq:NN	\cs_generate_from_arg_count_error_msg:Nn	
... 133, 5859, 5859, 5881, 5882	... 1318, 1343, 1351	
\clist_show:c	\cs_generate_internal_variant:n	
\clist_show:N . 132, 5791, 5791, 5816, 5889	... 36, 1793, 1839, 1839	
\clist_show_aux:n	\cs_generate_internal_variant_aux:N	
\clist_show_aux:w 1839, 1844, 1847, 1853	
\clist_tmp:w	\cs_generate_variant:Nn	
... 5685, 5685, 5695, 5703, 5708, 5716	... 28, 1768, 1768, 1855–1862,	
\clist_top:cN	1871, 1880–1883, 1896, 1897, 1906–	
\clist_top:NN	1909, 2055, 2056, 2061, 2062, 2118,	
\clist_trim_spaces:c	2135, 2437, 2438, 2455–2458, 2477–	
\clist_trim_spaces:N 128, 5671, 5679, 5683	2480, 3277, 3298, 3301, 3302, 3304,	
\clist_trim_spaces:n	3305, 3307, 3308, 3317–3320, 3329–	
... 128, 5671, 5671, 5680, 5682	3332, 3336, 3337, 3703, 3905, 3908,	
\clist_trim_spaces_aux_i:n	3909, 3913, 3914, 3916, 3917, 3919,	
... 5671, 5673, 5676	3920, 3929–3932, 3936, 3937, 3941,	
\clist_trim_spaces_aux_ii:n	3942, 4040, 4042, 4064, 4067, 4068,	
... 5671, 5674, 5677	4072, 4073, 4075, 4076, 4078, 4079,	
\clist_use:c	4083, 4084, 4088, 4089, 4113, 4120,	
\clist_use:N	4121, 4123, 4138, 4142, 4143, 4147,	
\closein	4148, 4150, 4151, 4153, 4154, 4158,	
\closeout	4159, 4163, 4164, 4168, 4170, 4196,	
\clubpenalties	4207, 4208, 4213, 4214, 4219, 4220,	
\clubpenalty	4241–4246, 4263–4270, 4287–4294,	

- 4329–4332, 4348, 4349, 4372–4375,
 4416, 4417, 4422, 4423, 4426–4433,
 4442–4445, 4454–4457, 4477–4480,
 4499–4501, 4508–4516, 4529, 4550,
 4561, 4565, 4586, 4587, 4639, 4640,
 4648, 4649, 4677–4680, 4768, 4890,
 4893, 4950, 4951, 4980, 5018, 5019,
 5024–5027, 5032–5035, 5051, 5052,
 5077, 5078, 5100–5105, 5114, 5130,
 5131, 5155, 5182, 5183, 5208, 5237,
 5248, 5249, 5302, 5325–5330, 5359–
 5370, 5380, 5404, 5406, 5431, 5432,
 5454–5459, 5562, 5563, 5574–5577,
 5582–5585, 5590, 5604, 5605, 5639,
 5640, 5669, 5670, 5683, 5684, 5718–
 5726, 5749, 5771, 5788, 5816, 5834,
 5858, 5881–5884, 5908, 5944–5947,
 5956, 5957, 5975–5978, 5993, 5995,
 5997, 5999, 6014, 6015, 6024–6027,
 6047–6054, 6066–6068, 6082, 6083,
 6094, 6125, 6144–6149, 6163, 6168,
 6175, 6176, 6178–6181, 6200, 6205,
 6206, 6219, 6220, 6225, 6226, 6231,
 6232, 6236–6238, 6245–6247, 6250,
 6251, 6267–6274, 6277–6280, 6286,
 6287, 6302, 6306, 6307, 6312, 6313,
 6318, 6319, 6331, 6332, 6340, 6341,
 6346, 6347, 6352, 6353, 6358, 6359,
 6364, 6365, 6422, 6423, 6450, 6451,
 6566, 6567, 6621, 6624, 6982–6985,
 7112, 7508, 7673, 7726, 7727, 7822,
 7823, 7836, 7837, 8419, 8425, 8430,
 8431, 8464, 8465, 8512, 8513, 8528,
 8590, 8593, 8660, 8885, 8888, 8920,
 8923, 9004, 9005, 9029, 9030, 9055,
 9056, 9158, 9159, 9176, 9177, 9289,
 9290, 9831, 9832, 9928, 9929, 10129,
 10130, 10322, 10323, 10625, 10640,
 10641, 10974, 10975, 11504, 11507
 \cs_generate_variant_aux:N
 1768, 1783, 1804, 1810
 \cs_generate_variant_aux:NNn
 1768, 1781, 1787
 \cs_generate_variant_aux:nnNNn
 1768, 1771, 1774
 \cs_generate_variant_aux:Nnnw
 1768, 1775, 1776, 1785
 \cs_generate_variant_aux:NNpx
 1791, 1812, 1825
 \cs_generate_variant_aux:w
 1812, 1827, 1830
 \cs_get_arg_count_from_signature:c
 1316, 1394
 \cs_get_arg_count_from_signature:N
 21, 937,
 946, 953, 963, 1300, 1300, 1317, 1362
 \cs_get_arg_count_from_signature_aux:nnN
 1300, 1301, 1302
 \cs_get_arg_count_from_signature_auxii:w
 1300, 1310, 1315
 \cs_get_function_name:N 21, 1096, 1096
 \cs_get_function_signature:N 1096, 1098
 \cs_gnew:cpn 1503
 \cs_gnew:cpx 1507
 \cs_gnew:Npn 1495
 \cs_gnew:Npx 1499
 \cs_gnew_eq:cc 1513
 \cs_gnew_eq:cN 1511
 \cs_gnew_eq:Nc 1512
 \cs_gnew_eq:NN 1510
 \cs_gnew_nopar:cpn 1502
 \cs_gnew_nopar:cpx 1506
 \cs_gnew_nopar:Npn 1494
 \cs_gnew_nopar:Npx 1498
 \cs_gnew_protected:cpn 1505
 \cs_gnew_protected:cpx 1509
 \cs_gnew_protected:Npn 1497
 \cs_gnew_protected:Npx 1501
 \cs_gnew_protected_nopar:cpn 1504
 \cs_gnew_protected_nopar:cpx 1508
 \cs_gnew_protected_nopar:Npn 1496
 \cs_gnew_protected_nopar:Npx 1500
 \cs_gset:cn 14, 1397
 \cs_gset:cpn 11, 1249, 1251, 4533,
 4543, 5754, 5764, 6088, 6812, 6814
 \cs_gset:cpx 11, 1249, 1252
 \cs_gset:cx 14, 1397
 \cs_gset:Nn 14, 1356
 \cs_gset:Npn 11, 838, 840, 1235,
 1251, 3088, 5212, 7434, 7435, 7437
 \cs_gset:Npx
 11, 838, 842, 1236, 1252, 3089, 5217
 \cs_gset:Nx 14, 1356
 \cs_gset_eq:cc 15, 1292, 1295, 1891, 4228
 \cs_gset_eq:cN
 15, 1292, 1294, 1299, 1890, 4226,
 5221, 5905, 6479, 6516, 7473, 7475
 \cs_gset_eq:Nc 15, 1292, 1293, 1889,
 4227, 5228, 6471, 6487, 6508, 6524

- \cs_gset_eq:NN
 . 15, 1292, 1292–1295, 1297, 1877,
 1879, 1888, 3091, 4194, 4225, 5904
- \cs_gset_nopar:cn 14, 1397
- \cs_gset_nopar:cpn .. 11, 1241, 1245, 2205
- \cs_gset_nopar:cpx .. 11, 1241, 1246, 2986
- \cs_gset_nopar:cx 14, 1397
- \cs_gset_nopar:Nn 14, 1356
- \cs_gset_nopar:Npn 11,
 838, 838, 841, 845, 849, 1233, 1245
- \cs_gset_nopar:Npx 11, 838, 839,
 843, 847, 851, 1234, 1246, 4200,
 4205, 4236, 4238, 4240, 4256, 4258,
 4260, 4262, 4280, 4282, 4284, 4286
- \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, 6977
- \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 .. 3791, 3798, 3800, 7646
- \cs_if_exist:cT 6792, 7882
- \cs_if_exist:cTF 23, 1100,
 6470, 6496, 6507, 6533, 7069, 7079,
 7521, 7620, 7919, 7933, 7939, 11222
- \cs_if_exist:N 1100
- \cs_if_exist:NF .. 1217, 7564, 7579, 7720
- \cs_if_exist:NT
 .. 1460, 1471, 6544, 6556, 8057, 8074
- \cs_if_exist:NTF
 23, 1100, 1440, 2707, 4216,
 4218, 5907, 5910, 6209, 6215, 6752
- \cs_if_exist_p:c 23, 1100
- \cs_if_exist_p:N 23, 1100
- \cs_if_free:c 1140
- \cs_if_free:cT 1841
- \cs_if_free:cTF 23, 1128
- \cs_if_free:N 1128
- \cs_if_free:NF 1194, 1204
- \cs_if_free:NTF
 ... 23, 1128, 1789, 6633, 6635, 11514
- \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, 1947,
 1960, 1993, 1995, 1997, 1998, 2148–
 2151, 2153, 2155, 2157, 2159, 2161,
 2163, 2165–2175, 3354, 3362, 3370,
 3378, 3386, 3394, 3402, 3974–3980
- \cs_new:cpx 9, 1249, 1254, 1507, 1843
- \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,
 1571, 1578, 1579, 1581, 1583, 1585,
 1587, 1589, 1596, 1598, 1603, 1608,
 1614, 1620, 1626, 1632, 1645, 1652,
 1659, 1666, 1700, 1701, 1706, 1708,
 1713, 1718, 1720, 1722, 1723, 1725,

- 1731, 1737, 1741, 1748, 1750, 1752,
 1754, 1755, 1757, 1762, 1767, 1774,
 1776, 1787, 1804, 1830, 1847, 1892,
 1894, 1920, 1925, 1935, 1945, 1946,
 1973–1975, 1984, 1999, 2004, 2028,
 2034, 2040, 2044, 2045, 2051, 2053,
 2057, 2059, 2063, 2071, 2076, 2084,
 2090, 2092, 2094, 2100, 2102, 2104,
 2110, 2112, 2119, 2121, 2127, 2129,
 2176, 2183, 2192, 2390, 2396, 2405,
 2418, 2434, 2835, 2861, 2876, 2957,
 3047, 3056, 3065, 3078, 3206, 3208,
 3216, 3227, 3238, 3263, 3264, 3342,
 3352, 3434, 3442, 3450, 3456, 3462,
 3470, 3478, 3484, 3503, 3535, 3567,
 3569, 3575, 3587, 3595, 3628, 3630,
 3632, 3670, 3675, 3680, 3704, 3712,
 3714, 3723, 3725, 3734, 3736, 3746,
 3755, 3757, 3759, 3858, 3873, 3962,
 4399, 4400, 4410, 4458, 4517, 4524,
 4575, 4585, 4588, 4590, 4598, 4612,
 4620, 4625, 4631, 4641–4643, 4645,
 4647, 4650, 4658, 4704, 4712, 4761,
 4788, 4798–4801, 4810, 4811, 4817,
 4828, 4837, 4838, 4845, 4851, 4853,
 4855, 4860, 4869, 4871, 4873, 4879,
 4888, 4894, 4906–4908, 4920, 4922,
 4924, 4931, 4932, 4940, 4979, 4981,
 4989, 5150, 5184, 5185, 5196, 5202,
 5296, 5301, 5371, 5379, 5424, 5453,
 5473, 5503, 5509, 5671, 5676, 5677,
 5735, 5743, 5810, 5815, 5817, 5825,
 5833, 5930, 6034, 6074, 6117, 6124,
 6150, 6155, 6164, 6166, 6583, 6732,
 6739, 6981, 7340, 7420, 11491, 11502
 \cs_new:Npx 9, 1225, 1236, 1254, 1499, 7327
 \cs_new:Nx 12, 1381
 \cs_new_eq:cc .. 15, 968, 1284, 1291, 1513
 \cs_new_eq:cN 15, 1284,
 1289, 1511, 5901, 10183, 10207, 10238
 \cs_new_eq:Nc 15, 1284, 1290, 1512
 \cs_new_eq:NN 15, 1284, 1284, 1289–1291,
 1448–1459, 1494–1516, 1540, 1870,
 1884–1891, 2089, 2584–2586, 2869–
 2871, 3110–3114, 3130–3144, 3146,
 3148–3162, 3164, 3166–3188, 3196–
 3201, 3338, 3821, 3822, 3849–3851,
 3895–3897, 4039, 4041, 4049, 4050,
 4112, 4114, 4117, 4122, 4124, 4125,
 4167, 4169, 4221–4228, 4361, 4362,
 4562, 4563, 4566, 4769, 4956–4978,
 4996–5013, 5186–5188, 5250–5275,
 5511–5514, 5524–5541, 5606–5623,
 5789, 5790, 5885–5890, 5900, 5911–
 5919, 6095, 6096, 6169, 6170, 6177,
 6233–6235, 6248, 6249, 6260–6262,
 6281, 6289, 6295, 6301, 6320, 6321,
 6329, 6330, 6360–6363, 6375–6379,
 6399, 6409, 6605, 6620, 6749, 6774–
 6779, 7343–7348, 7907, 8008–8010,
 8516–8525, 11487, 11488, 11590–11593
 \cs_new_nopar:cn 12, 1413
 \cs_new_nopar:cpn 9, 1241, 1247, 1267–
 1274, 1276, 1278, 1502, 2011–2023,
 2025, 8567, 8568, 8570, 8572, 8574,
 8576, 8578, 8580, 8582, 8628, 8630,
 8632, 8634, 8636, 8638, 8640, 8642,
 8644, 8690, 8695, 8700, 8705, 8710,
 8715, 8720, 8725, 8730, 8735, 8740
 \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, 1638, 1674–
 1681, 1688–1692, 1739, 1740, 1825,
 2009, 2010, 2136, 2143, 2145, 2147,
 2252, 2254, 2256, 2270, 2275, 2284,
 2289, 2492, 2494, 2562, 2564, 2568,
 2570, 2574, 2576, 2580, 2582, 2602,
 2691, 2731, 2738, 2749, 2760, 2771,
 2782, 2790, 2798, 2806, 2827, 2834,
 2843, 2852, 2872–2875, 2926, 2935,
 2943, 3044, 3115, 3116, 3118, 3119,
 3121, 3122, 3124, 3125, 3127, 3128,
 3311, 3339, 3490, 3491, 3634, 3639,
 3644, 3649, 3654–3669, 3775, 3784,
 3818, 3820, 3852, 3943, 3945, 4037,
 4110, 4115, 4118, 4165, 4171, 4405,
 4519, 4564, 4567, 4580, 4656, 4664,
 5099, 5303, 5381, 5397, 5405, 5407,
 5416, 5727, 6069, 6354, 6730, 6748,
 6851–6853, 6952–6957, 7562, 7577,
 7684, 7897, 7899, 7908, 7917, 7926,
 7943, 8526, 8529, 8546, 8547, 8553,
 8562, 8583, 8589, 8591, 8594, 8606,
 8617, 8626, 8645, 8653, 8658, 8661,
 8673, 8682, 8684, 8741, 8754, 8773,
 8793, 8799, 8838, 8877–8879, 8881
 \cs_new_nopar:Npx

- 9, 1225, 1234, 1248, 1498, 1835, 4322
 \cs_new_nopar:Nx 12, 1381
 \cs_new_protected:cn 12, 1413
 \cs_new_protected:cpn . 9, 1261, 1265,
 1505, 7738, 7740, 7742, 7744, 7746,
 7748, 7750, 7768, 7778, 7780, 7784
 \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, 1768,
 1839, 2202, 2215, 2224, 2233, 2374,
 2375, 2881, 2887, 2904, 2906, 2908,
 2922, 2924, 4197, 4202, 4229, 4231,
 4233, 4235, 4237, 4239, 4247, 4249,
 4251, 4253, 4255, 4257, 4259, 4261,
 4271, 4273, 4275, 4277, 4279, 4281,
 4283, 4285, 4310, 4350, 4376, 4530,
 4540, 4551, 4555, 4635, 4637, 4767,
 4945, 5020, 5022, 5028, 5030, 5037,
 5039, 5041, 5053, 5055, 5057, 5112,
 5125, 5209, 5214, 5219, 5231, 5238,
 5433, 5438, 5443, 5448, 5568, 5578,
 5588, 5597, 5625, 5627, 5629, 5641,
 5643, 5645, 5668, 5679, 5681, 5685,
 5751, 5761, 5772, 5780, 5859, 5870,
 5900–5906, 5909, 5922, 5931, 5938,
 5940, 5942, 5948, 5954, 5958, 5964,
 5970, 5979–5981, 5985, 6005, 6061,
 6085, 6138, 6171, 6173, 6194, 6252,
 6254, 6256, 6258, 6304, 6308, 6322,
 6324, 6325, 6327, 6335, 6337, 6338,
 6342, 6348, 6647, 6674, 6770, 6772,
 6790, 6799, 6801, 6808, 6810, 6817,
 6866, 6881, 6889, 6897, 6899, 6922,
 6937, 6944, 7056, 7101, 7111, 7127,
 7139, 7141, 7143, 7145, 7147, 7174,
 7183, 7196, 7198, 7200, 7202, 7205,
 7218, 7220, 7222, 7224, 7349–7355,
 7376, 7390, 7402, 7424, 7439, 7462,
 7470, 7500, 7502, 7514, 7519, 7545,
 7560, 7602, 7618, 7644, 7655, 7660,
 7671, 7696, 7814, 7816, 7824, 7826,
 7843, 7848, 7875, 11496, 11505, 11573
 \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, 7728, 7730, 7732,
 7734, 7736, 7752, 7754, 7756, 7758,
 7760, 7762, 7764, 7766, 7770, 7772,
 7774, 7776, 7782, 7786, 7788, 7790,
 7792, 7794, 7796, 7798, 7800, 7802,
 7804, 7806, 7808, 7810, 7812, 11229,
 11255, 11290, 11308, 11340, 11372
 \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,
 1673, 1682–1687, 1693–1699, 1870,
 1872, 1874, 1876, 1878, 2296, 2383,
 2490, 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, 2558, 2560, 2566, 2572,
 2578, 2584, 2877, 2879, 2966, 2975,
 3093, 3104, 3106, 3108, 3271, 3278,
 3299, 3300, 3303, 3306, 3309, 3313,
 3315, 3321, 3323, 3325, 3327, 3333,
 3335, 3899, 3906, 3907, 3910, 3912,
 3915, 3918, 3921, 3923, 3925, 3927,
 3933, 3935, 3938, 3940, 4058, 4065,
 4066, 4069, 4071, 4074, 4077, 4080,
 4082, 4085, 4087, 4132, 4139, 4141,
 4144, 4146, 4149, 4152, 4155, 4157,
 4160, 4162, 4191, 4209, 4211, 4215,
 4217, 4306, 4308, 4333, 4335, 4337,
 4364, 4366, 4368, 4370, 4412, 4414,
 4418, 4420, 4496–4498, 4553, 4891,
 4952, 4954, 5014, 5016, 5106, 5115,
 5117, 5119, 5132, 5138, 5156, 5158,
 5160, 5166, 5189, 5225, 5277, 5460,
 5462, 5464, 5466, 5479, 5481, 5483,
 5542, 5544, 5546, 5564, 5566, 5580,
 5586, 5591, 5593, 5595, 5791, 6001,
 6003, 6098, 6201, 6203, 6207, 6213,
 6221, 6223, 6227, 6229, 6239, 6241,
 6243, 6282, 6284, 6303, 6305, 6310,
 6314, 6316, 6333, 6334, 6339, 6344,
 6350, 6356, 6366, 6412, 6415, 6424,
 6437, 6452, 6460, 6468, 6491, 6505,
 6528, 6542, 6554, 6568, 6590, 6622,
 6625, 6626, 6629, 6631, 6632, 6634,

- 6685, 6696, 6709, 6716, 6725, 6762,
6764, 6766, 6768, 6961, 7083, 7091,
7110, 7113, 7115, 7117, 7119, 7121,
7123, 7125, 7509, 7528, 7535, 7592,
7632, 7665, 7674, 7679, 7686, 7712,
7718, 7724, 7838, 8043, 8053, 8086,
8103, 8108, 8110, 8201, 8203, 8214,
8224, 8249, 8257, 8259, 8261, 8267,
8269, 8286, 8298, 8353–8355, 8363,
8373, 8388, 8398, 8413, 8414, 8420,
8426, 8428, 8432–8434, 8466, 8468,
8470, 8883, 8886, 8889, 8918, 8921,
8924, 8954, 8963, 8976, 9002, 9003,
9006, 9027, 9028, 9031, 9053, 9054,
9057, 9070, 9107, 9124, 9156, 9157,
9160, 9174, 9175, 9178, 9229, 9259,
9271, 9276, 9282, 9287, 9288, 9291,
9326, 9372, 9390, 9407, 9418, 9419,
9424, 9433, 9439, 9455, 9478, 9522,
9582, 9599, 9604, 9614, 9624, 9632,
9642, 9665, 9675, 9689, 9695, 9708,
9727, 9745, 9784, 9802, 9829, 9830,
9833, 9880, 9913, 9926, 9927, 9930,
9966, 9999, 10022, 10058, 10073,
10127, 10128, 10131, 10178, 10188,
10217, 10247, 10320, 10321, 10324,
10368, 10396, 10408, 10443, 10475,
10495, 10501, 10508, 10572, 10620,
10638, 10639, 10642, 10675, 10699,
10733, 10752, 10762, 10776, 10788,
10810, 10835, 10843, 10855, 10861,
10915, 10925, 10940, 10972, 10973,
10976, 11027, 11063, 11117, 11129,
11220, 11406, 11412, 11418, 11424,
11430, 11436, 11442, 11454, 11462,
11467, 11476, 11512, 11542, 11552
\cs_new_protected_nopar:Npx
..... 9, 1225, 1238, 1260, 1500, 1833
\cs_new_protected_nopar:Nx 12, 1381
\cs_record_meaning:N 1191, 1191
\cs_set:cn 13, 1397
\cs_set:cpn 10, 1249, 1249, 6803, 6805, 7658
\cs_set:cpx 10, 1249, 1250,
2297, 2301, 2305, 2309, 2313, 2322,
2331, 2340, 2349, 2351, 2353, 2355,
2357, 2359, 2361, 2363, 2365, 7663
\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,
2372, 2373, 3076, 3086, 3203, 3981,
3989, 3997, 4003, 4009, 4017, 4025,
4031, 4504, 4596, 5837, 5847, 6664
\cs_set:Npx 10,
824, 828, 856, 1250, 3087, 4384, 6655
\cs_set:Nx 13, 1356
\cs_set_eq:cc 15, 966, 1280, 1283, 1887, 4224
\cs_set_eq:cN
.... 15, 1280, 1281, 1886, 4222, 5903
\cs_set_eq:Nc .. 15, 1280, 1282, 1885, 4223
\cs_set_eq:NN 15, 1280, 1280–1283, 1287,
1292, 1462–1469, 1473–1480, 1873,
1875, 1884, 2625, 2885, 2889, 2910,
2912, 2971, 2989, 3090, 4221, 5468,
5469, 5471, 5902, 6419, 6420, 6656,
7830, 7832, 9329, 9422, 10251, 11173
\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, 2978, 2984, 5835, 6854, 8508
\cs_set_nopar:Npx 10, 824, 825, 829, 833,
837, 1244, 1543, 2891, 2896, 2913,
2914, 4230, 4232, 4234, 4248, 4250,
4252, 4254, 4272, 4274, 4276, 4278
\cs_set_nopar:Nx 13, 1356
\cs_set_protected:cn 13, 1397
\cs_set_protected:cpn 10, 1261, 1261, 6964
\cs_set_protected:cpx 10, 1261, 1262,
1358, 1391, 6966, 6968, 6970, 6972
\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,
4324, 5088, 5650, 5695, 5708, 5920,
5924, 5932, 7176, 7178, 7180, 7302
\cs_set_protected:Npx
..... 10, 824, 836, 1262, 7063
\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](#), [6628](#), [6630](#), [8300](#), [8309](#),
 [8317](#), [8326](#), [9262](#), [11531](#), [11535](#),
 [11562](#), [11563](#), [11568](#), [11569](#), [11581](#)
`\cs_set_protected_nopar:Npx` [10](#), [296](#),
 [824](#), [832](#), [1256](#), [6932](#), [7058](#), [8444](#),
 [8486](#), [8506](#), [8898](#), [8934](#), [9011](#), [9087](#),
 [9198](#), [9305](#), [9315](#), [9341](#), [9858](#), [9871](#),
 [9958](#), [10156](#), [10169](#), [10353](#), [10658](#),
 [10666](#), [10692](#), [10884](#), [10998](#), [11004](#),
 [11015](#), [11048](#), [11055](#), [11101](#), [11136](#)
`\cs_set_protected_nopar:Nx` [13](#), [1356](#)
`\cs_show:c` [16](#), [821](#), [823](#), [7944](#)
`\cs_show:N` [16](#), [805](#), [810](#), [823](#), [4767](#)
`\cs_split_function:NN` . [903](#), [911](#), [918](#),
 [926](#), [934](#), [943](#), [951](#), [960](#), [1053](#), [1054](#),
 [1072](#), [1078](#), [1097](#), [1099](#), [1301](#), [1771](#)
`\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](#), [2273](#), [6749](#)
`\cs_to_str_aux:w` [1063](#), [1066](#), [1070](#)
`\cs_undefine:c` [16](#), [1296](#), [1298](#), [1515](#)
`\cs_undefine:N`
 [16](#), [1296](#), [1296](#), [1514](#), [6550](#), [6562](#)
`\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](#)
`\currentgroupstype` [696](#)
`\currentifbranch` [692](#)
`\currentiflevel` [691](#)
`\currentifttype` [693](#)

D

`\d` [1813](#), [2719](#)
`\dagger` [3863](#), [3869](#)
`\day` [651](#)
`\ddagger` [3864](#), [3870](#)
`\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` [169](#)
 `.default:V` [169](#)
`\defaultshyphenchar` [635](#)
`\defaultskewchar` [636](#)
`\delcode` [666](#)
`\delimiter` [460](#)
`\delimiterfactor` [509](#)
`\delimitershortfall` [508](#)
`\deprecated` [2374](#), [2375](#), [7349–7355](#)
`\detokenize` [32](#), [35](#), [80](#), [93](#), [96](#),
 [167](#), [170](#), [176](#), [185](#), [190](#), [192](#), [198](#),
 [201](#), [204](#), [214](#), [269](#), [271](#), [276](#), [278](#), [683](#)
`\dim_add:cn` [79](#), [3933](#)
`\dim_add:Nn` [79](#), [3933](#), [3933](#), [3935](#), [3936](#)
`\dim_compare:n` [3952](#)
`\dim_compare:nF` [3991](#), [4006](#)
`\dim_compare:nNn` [3947](#)
`\dim_compare:nNnF` [4019](#), [4034](#)
`\dim_compare:nNnT`
 [3922](#), [3924](#), [3926](#), [3928](#), [4011](#), [4028](#)
`\dim_compare:nNnTF` [82](#), [2106](#), [3947](#)
`\dim_compare:nT` [3983](#), [4000](#)
`\dim_compare:nTF` [82](#), [3952](#)
`\dim_compare:<:nw` [3952](#)
`\dim_compare=:nw` [3952](#)
`\dim_compare:>:nw` [3952](#)
`\dim_compare_aux:wNN` [3952](#), [3954](#), [3962](#)
`\dim_compare_p:n` [82](#), [3952](#)
`\dim_compare_p:nNn` [82](#), [3947](#)
`\dim_do_until:nn` [83](#), [3981](#), [4003](#), [4007](#)
`\dim_do_until:nNnn` [82](#), [4009](#), [4031](#), [4035](#)
`\dim_do_while:nn` [83](#), [3981](#), [3997](#)
`\dim_do_while:nNnn`
 [82](#), [4001](#), [4009](#), [4025](#), [4029](#)
`\dim_eval:n` [84](#), [2101](#), [4037](#), [4037](#)
`\dim_eval:w`
 [91](#), [3895](#), [3896](#), [3911](#), [3934](#), [3939](#),
 [3946](#), [3949](#), [3954](#), [3974–3980](#), [4038](#),
 [6240](#), [6242](#), [6244](#), [6253](#), [6255](#), [6257](#),
 [6259](#), [6309](#), [6323](#), [6336](#), [6349](#), [6367](#)

- \dim_eval_end: .. [91](#), [3895](#), [3897](#), [3911](#),
[3934](#), [3939](#), [3946](#), [3949](#), [3955](#), [4038](#),
[6240](#), [6242](#), [6244](#), [6253](#), [6255](#), [6257](#),
[6259](#), [6309](#), [6323](#), [6336](#), [6349](#), [6367](#)
 - \dim_gadd:cn [79](#), [3933](#)
 - \dim_gadd:Nn [79](#), [3933](#), [3935](#), [3937](#)
 - .dim_gset:c [170](#)
 - \dim_gset:cn [80](#), [3910](#)
 - .dim_gset:N [170](#)
 - \dim_gset:Nn [80](#), [3910](#), [3912](#), [3914](#), [3924](#), [3928](#)
 - \dim_gset_eq:cc [80](#), [3915](#)
 - \dim_gset_eq:cN [80](#), [3915](#)
 - \dim_gset_eq:Nc [80](#), [3915](#)
 - \dim_gset_eq:NN [80](#), [3915](#), [3918](#)–[3920](#)
 - \dim_gset_max:cn [80](#), [3921](#)
 - \dim_gset_max:Nn [80](#), [3921](#), [3923](#), [3930](#)
 - \dim_gset_min:cn [81](#), [3921](#)
 - \dim_gset_min:Nn [81](#), [3921](#), [3927](#), [3932](#)
 - \dim_gsub:cn [81](#), [3933](#)
 - \dim_gsub:Nn [81](#), [3933](#), [3940](#), [3942](#)
 - \dim_gzero:c [79](#), [3906](#)
 - \dim_gzero:N [79](#), [3906](#), [3907](#), [3909](#)
 - \dim_new:c [79](#), [3898](#)
 - \dim_new:N [79](#), [3898](#), [3899](#),
[3905](#), [4044](#), [4045](#), [4052](#)–[4056](#), [8514](#)
 - \dim_ratio:nn [81](#), [3943](#), [3943](#)
 - \dim_ratio_aux:n [3943](#), [3944](#), [3945](#)
 - .dim_set:c [170](#)
 - \dim_set:cn [80](#), [3910](#)
 - .dim_set:N [170](#)
 - \dim_set:Nn [80](#), [3910](#),
[3910](#), [3912](#), [3913](#), [3922](#), [3926](#), [4046](#)
 - \dim_set_eq:cc [80](#), [3915](#)
 - \dim_set_eq:cN [80](#), [3915](#)
 - \dim_set_eq:Nc [80](#), [3915](#)
 - \dim_set_eq:NN [80](#), [3915](#), [3915](#)–[3917](#)
 - \dim_set_max:cn [80](#), [3921](#)
 - \dim_set_max:Nn [80](#), [3921](#), [3921](#), [3929](#)
 - \dim_set_min:cn [80](#), [3921](#)
 - \dim_set_min:Nn [80](#), [3921](#), [3925](#), [3931](#)
 - \dim_show:c [84](#), [4041](#)
 - \dim_show:N [84](#), [4041](#), [4041](#), [4042](#)
 - \dim_sub:cn [81](#), [3933](#)
 - \dim_sub:Nn [81](#), [3933](#), [3938](#), [3940](#), [3941](#)
 - \dim_until_do:nn [83](#), [3981](#), [3989](#), [3994](#)
 - \dim_until_do:nNnn .. [83](#), [4009](#), [4017](#), [4022](#)
 - \dim_use:c [84](#), [4039](#)
 - \dim_use:N
[84](#), [3954](#), [4038](#), [4039](#), [4039](#), [4040](#), [8478](#)
 - \dim_while_do:nn [83](#), [3981](#), [3981](#), [3986](#)
 - \dim_while_do:nNnn .. [83](#), [4009](#), [4009](#), [4014](#)
 - \dim_zero:c [79](#), [3906](#)
 - \dim_zero:N [79](#), [3906](#), [3906](#)–[3908](#)
 - \dimen [657](#)
 - \dimendef [356](#)
 - \dimexpr [710](#)
 - \directlua [15](#), [756](#)
 - \discretionary [520](#)
 - \displayindent [485](#)
 - \displaylimits [495](#)
 - \displaystyle [473](#)
 - \displaywidowpenalties [723](#)
 - \displaywidowpenalty [484](#)
 - \displaywidth [486](#)
 - \divide [363](#)
 - \doublehyphenemerits [553](#)
 - \dp [664](#)
 - \dump [647](#)
- E**
- \E [1819](#), [2721](#)
 - \edef [33](#),
[68](#), [82](#), [164](#), [166](#), [181](#), [201](#), [266](#), [273](#), [351](#)
 - \else .. [14](#), [63](#), [117](#), [137](#), [172](#), [187](#), [207](#), [404](#)
 - \else: [25](#), [784](#), [787](#), [883](#), [1087](#), [1104](#), [1107](#),
[1116](#), [1122](#), [1132](#), [1135](#), [1144](#), [1150](#),
[1306](#), [1340](#), [1424](#), [1487](#), [1492](#), [1564](#),
[1834](#), [1902](#), [1916](#), [1930](#), [1940](#), [1951](#),
[1954](#), [1964](#), [1967](#), [1980](#), [1989](#), [2242](#),
[2244](#), [2246](#), [2250](#), [2400](#), [2429](#), [2443](#),
[2451](#), [2464](#), [2473](#), [2607](#), [2612](#), [2617](#),
[2622](#), [2629](#), [2635](#), [2640](#), [2645](#), [2650](#),
[2655](#), [2660](#), [2665](#), [2670](#), [2675](#), [2695](#),
[2703](#), [2710](#), [2744](#), [2755](#), [2766](#), [2777](#),
[2839](#), [2848](#), [2856](#), [2865](#), [2931](#), [2939](#),
[2961](#), [3222](#), [3233](#), [3243](#), [3248](#), [3251](#),
[3254](#), [3347](#), [3358](#), [3366](#), [3374](#), [3382](#),
[3390](#), [3398](#), [3406](#), [3414](#), [3422](#), [3430](#),
[3625](#), [3950](#), [3957](#), [4096](#), [4438](#), [4450](#),
[4463](#), [4473](#), [4489](#), [4571](#), [4673](#), [4693](#),
[4708](#), [4716](#), [4726](#), [4745](#), [4757](#), [4794](#),
[5699](#), [5712](#), [6020](#), [6264](#), [6266](#), [6276](#),
[6743](#), [6756](#), [8207](#), [8236](#), [8282](#), [8293](#),
[8343](#), [8349](#), [8359](#), [8451](#), [8493](#), [8536](#),
[8540](#), [8557](#), [8601](#), [8609](#), [8612](#), [8621](#),
[8649](#), [8668](#), [8677](#), [8745](#), [8748](#), [8765](#),
[8768](#), [8784](#), [8787](#), [8810](#), [8813](#), [8816](#),
[8819](#), [8822](#), [8825](#), [8828](#), [8831](#), [8834](#),
[8849](#), [8852](#), [8855](#), [8858](#), [8861](#), [8864](#),
[8867](#), [8870](#), [8873](#), [8905](#), [8941](#), [8970](#),

9884, 9898, 9040, 9078, 9094, 9119,	\etex_currentiflevel:D	691
9142, 9152, 9212, 9215, 9310, 9320,	\etex_currentifttype:D	693
9355, 9358, 9394, 9396, 9399, 9445,	\etex_detokenize:D	
9450, 9471, 9647, 9719, 9733, 9768,	... 683, 1832, 2409, 2422, 4563, 4564	
9793, 9813, 9846, 9863, 9867, 9886,	\etex_dimexpr:D	710, 3896
9901, 9943, 9955, 9972, 9987, 10015,	\etex_displaywidowpenalties:D	723
10017, 10027, 10043, 10048, 10063,	\etex_endL:D	731
10100, 10106, 10111, 10144, 10161,	\etex_endR:D	733
10165, 10182, 10194, 10197, 10200,	\etex_eTeXrevision:D	674
10209, 10213, 10240, 10243, 10268,	\etex_eTeXversion:D	674
10338, 10350, 10361, 10377, 10382,	\etex_everyeof:D	735, 4313, 4340, 4353
10387, 10392, 10400, 10416, 10430,	\etex_firstmarks:D	678
10436, 10461, 10480, 10515, 10523,	\etex_fontchardp:D	703
10547, 10550, 10593, 10599, 10604,	\etex_fontcharht:D	702
10657, 10665, 10689, 10703, 10706,	\etex_fontcharic:D	705
10720, 10738, 10744, 10784, 10792,	\etex_fontcharwd:D	704
10820, 10898, 10906, 10929, 10958,	\etex_glueexpr:D	711, 4070,
10964, 11003, 11010, 11020, 11032,	4081, 4086, 4111, 4116, 4119, 8473	
11046, 11054, 11067, 11081, 11090,	\etex_glueshrink:D	714, 4181
11096, 11180, 11188, 11238, 11242,	\etex_glueshrinkorder:D	716, 4105
11246, 11250, 11265, 11269, 11274,	\etex_gluestretch:D	713, 4180
11281, 11285, 11295, 11299, 11302,	\etex_gluestretchorder:D	715, 4104
11313, 11317, 11321, 11326, 11331,	\etex_gluetomu:D	717
11345, 11349, 11353, 11358, 11363	\etex_ifcsname:D	672, 801
\emergencystretch	\etex_ifdefined:D	671, 800
\end	\etex_iffontchar:D	701
\EndCatcodeRegime	\etex_interactionmode:D	699
\endcsname 13, 32, 35, 62, 80, 93, 96, 167,	\etex_interlinepenalties:D	720
170, 176, 185, 190, 192, 201, 204,	\etex_lastlinefit:D	719
214, 229, 233, 269, 271, 276, 278, 444	\etex_lastnodetype:D	700, 2260, 2261
\endgroup .. 12, 61, 111, 120, 228, 232, 377	\etex_marks:D	676
\endinput	\etex_middle:D	724
\endL	\etex_muexpr:D	712, 4145, 4156, 4161, 4166
\endlinechar	\etex_mutogluue:D	718
\endR	\etex_numexpr:D	709, 3197
seq_push:cn	\etex_pagediscards:D	727
\eqno	\etex_parshapedimen:D	708
\errhelp	\etex_parshapeindent:D	706
\errmessage	\etex_parshapelength:D	707
\ERROR	\etex_predisplaydirection:D	734
\errorcontextlines	\etex_protected:D	736, 819
\errorstopmode	\etex_readline:D	686, 6767, 6769
\escapechar	\etex_savinghyphcodes:D	725
\etex_beginL:D	\etex_savingvdiscards:D	726
\etex_beginR:D	\etex_scantokens:D	684, 4318, 4344, 4357
\etex_botmarks:D	\etex_showgroups:D	697
\etex_clubpenalties:D	\etex_showifs:D	698
\etex_currentgrouplevel:D	\etex_showtokens:D	
\etex_currentgroupstype:D	... 685, 4769, 5292, 5806, 6579, 6601	
\etex_currentifbranch:D	\etex_splitbotmarks:D	681

<code>\etex_splitdiscards:D</code>	728	2940, 2953, 2960, 2962, 3051, 3060,
<code>\etex_splitfirstmarks:D</code>	680	3069, 3341, 3344, 3597, 3636, 3646,
<code>\etex_TeXXETstate:D</code>	729	3779, 3946, 3954, 4318, 4357, 4394,
<code>\etex_topmarks:D</code>	677	4402, 4407, 4448, 4460, 4461, 4521,
<code>\etex_tracingassigns:D</code>	687	4522, 4564, 4570, 4632, 4636, 4638,
<code>\etex_tracinggroups:D</code>	694	4652, 4660, 4670, 4686, 4706, 4715,
<code>\etex_tracingifs:D</code>	690	4718, 4736–4738, 4765, 4771, 4831,
<code>\etex_tracingnesting:D</code>	689	4858, 4942, 4943, 5092, 5109, 5122,
<code>\etex_tracingscantokens:D</code>	688	5141, 5142, 5170, 5171, 5193, 5198,
<code>\etex_unexpanded:D</code>	682,	5292, 5293, 5307, 5313, 5334, 5341,
	804, 1754, 1756, 1759, 1764, 4600, 4892	5409–5411, 5418, 5419, 5587, 5596,
<code>\etex_unless:D</code>	673, 789	5657, 5658, 5661–5663, 5806, 5807,
<code>\etex_widowpenalties:D</code>	722	5866, 5877, 5927, 6038, 6077, 6113,
<code>\eTeXrevision</code>	675	6114, 6158, 6579, 6580, 6601, 6602,
<code>\eTeXversion</code>	674	6735, 6742, 6745, 6913, 7332–7335,
<code>\everycr</code>	386	7384, 7386, 7428, 7432, 7549, 7677,
<code>\everydisplay</code>	487	8020, 8032, 8099, 8202, 8222, 8227,
<code>\everyeof</code>	735	8231, 8235, 8238, 8242, 8245, 8264,
<code>\everyhbox</code>	626	8283, 8292, 8294, 8304, 8306, 8321,
<code>\everyjob</code>	31, 655	8323, 8335, 8350, 8358, 8360, 8367,
<code>\everymath</code>	511	8370, 8382, 8392, 8395, 8407, 8456,
<code>\everypar</code>	574	8477, 8498, 8527, 8535, 8538, 8541,
<code>\everyvbox</code>	627	8556, 8558, 8592, 8600, 8602, 8620,
<code>\exhyphenpenalty</code>	550	8622, 8659, 8667, 8669, 8676, 8678,
<code>[EXP]\cs_get_function_signature:N</code> ..	21	8744, 8747, 8749, 8761, 8780, 8895,
<code>\exp_after:wN</code>	33, 802,	8910, 8931, 8946, 8960, 8999, 9019,
	802, 820, 857, 896, 898, 982, 1050,	9045, 9050, 9051, 9077, 9079, 9099,
	1068, 1080, 1086, 1088, 1115, 1117,	9220, 9268, 9279, 9321, 9363, 9379,
	1120, 1143, 1145, 1148, 1305, 1307,	9385, 9393, 9400–9402, 9609, 9611,
	1310, 1361, 1393, 1524, 1531, 1533,	9621, 9636, 9639, 9669, 9672, 9683,
	1536, 1537, 1544, 1548, 1549, 1554,	9686, 9702, 9718, 9724, 9732, 9738–
	1555, 1560, 1565, 1567, 1570, 1578,	9740, 9812, 9824, 9851, 9868, 9906,
	1580, 1582, 1584, 1586, 1588, 1591–	9917, 9919, 9921, 9923, 9948, 9956,
	1593, 1597, 1600, 1605, 1610–1612,	9992, 10003, 10005, 10007, 10009,
	1616–1618, 1622–1624, 1628–1630,	10055, 10070, 10124, 10149, 10166,
	1634–1636, 1640–1643, 1647–1650,	10181, 10185, 10210, 10214, 10241,
	1654–1656, 1661–1664, 1668–1671,	10244, 10273, 10343, 10351, 10374–
	1703, 1704, 1707, 1710, 1711, 1715,	10376, 10378–10380, 10384–10386,
	1716, 1719, 1721, 1722, 1724, 1727,	10393, 10399, 10405, 10415, 10419,
	1728, 1733, 1734, 1738, 1743–1745,	10432–10434, 10438, 10439, 10452,
	1749, 1751, 1753, 1754, 1756, 1759,	10497, 10565, 10617, 10656, 10663,
	1764, 1767, 1779, 1807, 1827, 1832,	10671, 10682, 10690, 10725, 10742,
	1833, 1835, 1851, 1950, 1953, 1955,	10780, 10783, 10819, 10821, 10832,
	1963, 1966, 1968, 1973, 1974, 1977,	10840, 10857, 10905, 10907, 10919,
	1986, 1994, 1996–1998, 2001, 2006,	10922, 10969, 11021, 11031, 11043–
	2139, 2286, 2393, 2399, 2401, 2409,	11045, 11066, 11075–11079, 11083–
	2415, 2422, 2428, 2430, 2688, 2709,	11088, 11092–11094, 11097, 11109
	2728, 2735, 2745, 2756, 2767, 2778,	
	2787, 2795, 2803, 2823, 2846, 2847,	<code>\exp_arg_last_unbraced:nn</code>
	2849, 2855, 2858, 2930, 2932, 2938,	.. 1700, 1700, 1703, 1707, 1710, 1715
		<code>\exp_arg_next:nnn</code>
		1525,

- 1525, 1533, 1536, 1544, 1548, 1554
 \exp_arg_next_nobrace:nnn 1525, 1526, 1531
 \exp_args:cc 30, 1583, 1583
 \exp_args:Nc 30, 820,
 820–823, 989, 997, 1005, 1013, 1214,
 1224, 1242, 1281, 1289, 1294, 1317,
 1350, 1426–1429, 1447, 1583, 4535
 \exp_args:Ncc 31, 1283,
 1291, 1295, 1434–1437, 1583, 1587
 \exp_args:Nccc 32, 1583, 1589
 \exp_args:Ncco 32, 1645, 1666
 \exp_args:Nccx 33, 1688, 1697
 \exp_args:Ncf . . 31, 1608, 1632, 2209, 2210
 \exp_args:NcNc 32, 1645, 1652
 \exp_args:NcNo 32, 1645, 1659
 \exp_args:Ncnx 33, 1688, 1698
 \exp_args:Nco 31, 1608, 1626
 \exp_args:Ncx 32, 1674, 1683
 \exp_args:Nf 30, 1596, 1596,
 2091, 2101, 2179, 2180, 2188, 2197,
 3495, 3497, 3501, 3568, 3580, 3589,
 3682, 3695, 3709, 3719, 3730, 3741,
 4857, 4934, 5402, 5673, 5841, 5853
 \exp_args:Nff 31, 1674, 1676
 \exp_args:Nfo 31, 1674, 1675
 \exp_args:NNc
 . 31, 1053, 1054, 1282, 1290, 1293,
 1430–1433, 1583, 1585, 1781, 5757
 \exp_args:Nnc 31, 1674, 1674, 5767
 \exp_args:NNf
 31, 1608, 1608, 2220, 2229, 7426, 7464
 \exp_args:Nnf
 . . . 31, 932, 941, 950, 958, 1674, 1677
 \exp_args:Nnnc 32, 1688, 1690
 \exp_args:NNNo . 32, 1578, 1581, 4319, 4345
 \exp_args:NNno 32, 1688, 1688
 \exp_args:Nnno 32, 1688, 1691
 \exp_args:NNNV 32, 1645, 1645
 \exp_args:NNnx 33, 1688, 1693
 \exp_args:Nnnx 33, 1688, 1695
 \exp_args:NNo
 31, 1578, 1579, 4936, 5921, 6671, 7676
 \exp_args:Nno
 31, 1674, 1678, 3043, 3961, 5936, 7921
 \exp_args:NNoo 32, 1688, 1689
 \exp_args:NNox 33, 1688, 1694
 \exp_args:Nnox 33, 1688, 1696
 \exp_args:NNV 31, 1608, 1620
 \exp_args:NNv 31, 1608, 1614
 \exp_args:NnV 31, 1674, 1679
 \exp_args:NNx 32, 1674, 1682
 \exp_args:Nnx 32, 1674, 1684
 \exp_args:No 30, 1578, 1578, 3572, 4313,
 4353, 4358, 4496–4498, 4554, 4798–
 4801, 5506, 5680, 5682, 5836, 6731
 \exp_args:Noc 31, 1674, 1680
 \exp_args:Noo 31, 1674, 1681
 \exp_args:Nooo 32, 1688, 1692
 \exp_args:Noox 33, 1688, 1699
 \exp_args:Nox 32, 1674, 1685
 \exp_args:NV 30, 1596, 1603
 \exp_args:Nv 30, 1596, 1598
 \exp_args:NVV 31, 1608, 1638
 \exp_args:Nx 31, 1596, 1673, 1673
 \exp_args:Nxo 32, 1674, 1686
 \exp_args:Nxx 32, 1674, 1687
 \exp_eval_error_msg:w . . 1558, 1562, 1571
 \exp_eval_register:c . 35, 1555, 1558,
 1569, 1601, 1618, 1716, 1721, 1765
 \exp_eval_register:N
 35, 1549, 1558, 1558,
 1570, 1606, 1624, 1642, 1643, 1650,
 1711, 1719, 1729, 1735, 1746, 1760
 \exp_last_two_unbraced:Noo 33, 1750, 1750
 \exp_last_two_unbraced_aux:nnN
 1751, 1752
 \exp_last_unbraced:NcV 33, 1718, 1725, 4545
 \exp_last_unbraced:Nf
 33, 1718, 1723, 2272, 3578
 \exp_last_unbraced:Nfo 33, 1718, 1740, 5383
 \exp_last_unbraced:NNNo . 33, 1718, 1748
 \exp_last_unbraced:NNNV . 33, 1718, 1741
 \exp_last_unbraced:NNo
 33, 1718, 1737, 4844, 5703, 5731, 6071
 \exp_last_unbraced:Nno 6152
 \exp_last_unbraced:NNV . . 33, 1718, 1731
 \exp_last_unbraced:No . . . 33, 1718, 1722
 \exp_last_unbraced:Noo . 1718, 1739, 6030
 \exp_last_unbraced:NV . . . 33, 1718, 1718
 \exp_last_unbraced:Nv . . . 33, 1718, 1720
 \exp_not:c 34, 1767,
 1767, 1792, 1849, 2298, 2302, 2307,
 2311, 2314, 2316–2318, 2323, 2325–
 2327, 2332, 2334–2336, 2341, 2343–
 2345, 2350, 2352, 2354, 2356, 2358,
 2360, 2362, 2364, 2366–2368, 2990,
 6967, 6969, 6971, 6973, 7337, 7409,
 7443, 7567, 7569, 7582, 7584, 7722
 \exp_not:f 35, 1754, 1755

- \exp_not:N 34, 802,
 803, 1360–1362, 1392–1394, 1524,
 1560, 1767, 1792, 2303, 2306, 2310,
 2314, 2323, 2332, 2341, 2462, 2471,
 2602, 2606, 2611, 2616, 2621, 2628,
 2634, 2639, 2644, 2649, 2654, 2659,
 2669, 2674, 2702, 2709, 2893, 2898,
 2916, 2929, 2959, 4324, 4326, 4340,
 4387, 4388, 4668, 4670, 4684, 4686,
 4714, 4721, 5242, 5505, 5507, 6667,
 6669, 7060, 7068, 7069, 7071, 7337,
 7338, 7429, 7465, 7567, 7569, 7582,
 7584, 7610, 7611, 7636, 7637, 7682,
 7704, 7705, 7722, 8447, 8489, 8508,
 8901, 8937, 9014, 9201, 9308, 9318,
 9861, 9874, 9961, 10159, 10172,
 10356, 10661, 10669, 10695, 10888,
 10890, 10892, 11139, 11141, 11143,
 11145, 11147, 11376, 11378, 11380,
 11382, 11384, 11386, 11388, 11390
 \exp_not:n 34, 802, 804, 1485,
 1524, 2299, 2462, 2471, 2894, 2899,
 2913, 2917, 2989, 2991, 4200, 4230,
 4236, 4248, 4256, 4272, 4280, 4389,
 4791, 5071, 5168, 5243, 5299, 5453,
 5477, 5510, 5813, 5864, 5875, 5989,
 5990, 6011, 6122, 6588, 6623, 6627,
 6771, 7061, 7065, 7072, 7329, 7434,
 7613, 7707, 7745, 11402, 11503, 11506
 \exp_not:o 35, 1754, 1754,
 4232, 4238, 4248, 4250, 4252, 4254,
 4256, 4258, 4260, 4262, 4272, 4274,
 4276, 4278, 4280, 4282, 4284, 4286,
 4399, 4411, 4763, 4791, 5015, 5017,
 5067, 5551, 5553, 5557, 6933, 7411,
 7445, 7448, 7455, 7467, 7912, 7914
 \exp_not:V
 34, 1754, 1757, 4250, 4258, 4274, 4282
 \exp_not:v 34, 1754, 1762, 7639
 \exp_stop_f: ... 35, 1534, 1540, 2273, 4937
 \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, 1867, 2380, 2487, 3193,
 3892, 4188, 4986, 5519, 5895, 6190,
 6372, 6784, 7360, 8015, 8128, 11482
 \ExplFileDescription .. 113, 130, 334,
 779, 1521, 1867, 2380, 2487, 3193,
 3892, 4188, 4986, 5519, 5895, 6190,
 6372, 6784, 7360, 8015, 8128, 11482
 \ExplFileName 114, 128, 334,
 779, 1521, 1867, 2380, 2487, 3193,
 3892, 4188, 4986, 5519, 5895, 6190,
 6372, 6784, 7360, 8015, 8128, 11482
 \ExplFileVersion .. 49, 115, 129, 334,
 779, 1521, 1867, 2380, 2487, 3193,
 3892, 4188, 4986, 5519, 5895, 6190,
 6372, 6784, 7360, 8015, 8128, 11482
 \ExplSyntaxNamesOff 6, 266, 273
 \ExplSyntaxNamesOn 6, 266, 266
 \ExplSyntaxOff 4, 6,
 67, 68, 178, 186, 208, 291, 296, 310, 325
 \ExplSyntaxOn 4,
 6, 67, 82, 150, 155, 160, 206, 291, 292
- F**
- \F 2682, 2716, 2815
 \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,
 1573, 1574, 1780, 1808, 1836, 1852,
 1904, 1918, 1930, 1940, 1956, 1957,
 1969, 1970, 1982, 1991, 2242, 2244,
 2246, 2250, 2253, 2255, 2394, 2402,
 2416, 2431, 2445, 2453, 2466, 2475,
 2607, 2612, 2617, 2622, 2629, 2635,
 2640, 2645, 2650, 2655, 2660, 2665,
 2670, 2675, 2697, 2703, 2710, 2747,
 2758, 2769, 2780, 2841, 2850, 2859,
 2867, 2933, 2941, 2963, 3213, 3224,
 3235, 3250, 3256, 3257, 3259, 3349,
 3353, 3360, 3368, 3376, 3384, 3392,
 3400, 3408, 3416, 3424, 3432, 3626,
 3950, 3959, 3969, 4098, 4440, 4452,
 4465, 4475, 4492, 4573, 4664, 4675,
 4695, 4710, 4719, 4728, 4747, 4759,
 4764, 4765, 4796, 4832, 5063, 5066,
 5093, 5169, 5173, 5194, 5308, 5701,
 5714, 6022, 6039, 6078, 6159, 6264,
 6266, 6276, 6746, 6758, 8209, 8246,
 8247, 8279, 8284, 8295, 8307, 8324,

8348, 8351, 8361, 8371, 8396, 8453,	\floatingpenalty	599
8495, 8533, 8542, 8543, 8559, 8598,	\font	365
8603, 8614, 8615, 8623, 8651, 8665,	\fontchardp	703
8670, 8679, 8750, 8751, 8767, 8771,	\fontcharht	702
8786, 8791, 8812, 8815, 8818, 8821,	\fontcharic	705
8824, 8827, 8830, 8833, 8836, 8851,	\fontcharwd	704
8854, 8857, 8860, 8863, 8866, 8869,	\fontdimen	632
8872, 8875, 8896, 8907, 8932, 8943,	\fontname	456
8974, 8986, 8994–8996, 9000, 9042,	\fp_abs:c	<u>185</u> , <u>9002</u>
9080, 9096, 9122, 9137, 9151, 9154,	\fp_abs:N	<u>185</u> , <u>9002</u> , 9002, 9004
9214, 9217, 9322, 9323, 9357, 9360,	\fp_abs_aux:NN	<u>9002</u> , 9002, 9003, 9006
9387, 9388, 9403–9405, 9415, 9448,	\fp_add:cn	<u>186</u> , <u>9053</u>
9453, 9463, 9467, 9475, 9476, 9597,	\fp_add:Nn	<u>186</u> , <u>9053</u> , 9053, 9055
9612, 9640, 9655, 9673, 9717, 9725,	\fp_add:NNNNNNNN	
9741–9743, 9756, 9760, 9781, 9782,	<u>9439</u> , <u>9439</u> , 10767, 10819, 10905
9791, 9800, 9826, 9827, 9848, 9870,	\fp_add_aux:NNn	<u>9053</u> , 9053, 9054, 9057
9877, 9890, 9903, 9924, 9945, 9957,	\fp_add_core:	<u>9053</u> , 9067, 9070, 9171
9976, 9989, 10010, 10014, 10019,	\fp_add_difference:	<u>9053</u> , 9079, 9124
10020, 10047, 10052, 10056, 10071,	\fp_add_sum:	<u>9053</u> , 9077, 9107
10104, 10110, 10118, 10122, 10123,	\fp_compare:n	11396
10125, 10146, 10168, 10175, 10186,	\fp_compare:NNN	11209
10196, 10202, 10203, 10212, 10215,	\fp_compare:nNn	11192
10242, 10245, 10270, 10340, 10352,	\fp_compare:NNTF	<u>11192</u>
10363, 10381, 10390, 10391, 10394,	\fp_compare:nNnTF	
10406, 10435, 10440, 10441, 10463,	<u>185</u> , <u>11192</u> , 11410, 11416,
10472, 10483, 10492, 10532, 10533,	11422, 11428, 11434, 11440, 11446
10545, 10553, 10554, 10597, 10603,	\fp_compare:nTF	<u>185</u> , <u>11396</u>
10611, 10615, 10616, 10618, 10664,	\fp_compare_<:	<u>11192</u>
10672, 10691, 10709, 10710, 10722,	\fp_compare_<_aux:	<u>11192</u>
10740, 10749, 10773, 10786, 10800,	\fp_compare_>:	<u>11192</u>
10822, 10828, 10833, 10841, 10847,	\fp_compare_absolute_a<b:	<u>11192</u>
10850, 10901, 10908, 10923, 10937,	\fp_compare_absolute_a>b:	<u>11192</u>
10962, 10968, 10970, 11009, 11023,	\fp_compare_aux:N	
11024, 11053, 11060, 11061, 11089,	<u>11192</u> , 11207, 11218, 11220
11095, 11099, 11100, 11182, 11190,	\fp_compare_aux_i:w	<u>11396</u> , 11402, 11406
11241, 11245, 11249, 11253, 11272,	\fp_compare_aux_ii:w	<u>11396</u> , 11409, 11412
11273, 11284, 11287, 11288, 11304–	\fp_compare_aux_iii:w	<u>11396</u> , 11415, 11418
11306, 11334–11338, 11366–11370	\fp_compare_aux_iv:w	<u>11396</u> , 11421, 11424
\file_add_path:nN	\fp_compare_aux_v:w	<u>11396</u> , 11427, 11430
.	\fp_compare_aux_vi:w	<u>11396</u> , 11433, 11436
\file_add_path_search:nN <u>8043</u> , 8047, 8053	\fp_compare_aux_vii:w	<u>11396</u> , 11439, 11442
\file_if_exist:n	\fp_const:cn	<u>181</u> , <u>8420</u>
\file_if_exist:nTF	\fp_const:Nn	<u>181</u> , <u>8420</u> , 8420, 8425
\file_input:n	\fp_cos:cn	<u>189</u> , <u>9926</u>
\file_list:	\fp_cos:Nn	<u>189</u> , <u>9926</u> , 9926, 9928
\file_path_include:n	\fp_cos_aux:NNn	<u>9926</u> , 9926, 9927, 9930
\file_path_remove:n	\fp_cos_aux_i:	<u>9926</u> , 9956, 9966
\finalhyphdemerits	\fp_cos_aux_ii:	<u>9926</u> , 9969, 9999, 10204
\firstmark	\fp_div:cn	<u>187</u> , <u>9287</u>
\firstmarks	\fp_div:Nn	<u>187</u> , <u>9287</u> , 9287, 9289

\fp_div_aux:NNn .. [9287](#), [9287](#), [9288](#), [9291](#)
\fp_div_divide: .. [9287](#), [9375](#), [9390](#), [9416](#)
\fp_div_divide_aux: [9287](#), [9393](#), [9402](#), [9407](#)
\fp_div_integer:NNNN .. [9582](#), [9582](#),
[10033](#), [10084](#), [10089](#), [10580](#), [10951](#)
\fp_div_internal:
..... [9287](#), [9321](#), [9326](#), [10869](#), [11127](#)
\fp_div_loop: [9287](#), [9331](#), [9372](#), [9386](#), [10253](#)
\fp_div_loop_step:w [9379](#), [9433](#)
\fp_div_store:
. [9287](#), [9329](#), [9376](#), [9418](#), [9422](#), [10251](#)
\fp_div_store_decimal: . [9287](#), [9422](#), [9424](#)
\fp_div_store_integer:
..... [9287](#), [9329](#), [9419](#), [10251](#)
\fp_exp:cn [188](#), [10320](#)
\fp_exp:Nn [188](#), [10320](#), [10320](#), [10322](#)
\fp_exp_aux: .. [10320](#), [10376](#), [10386](#), [10396](#)
\fp_exp_aux:NNn [10320](#), [10320](#), [10321](#), [10324](#)
\fp_exp_const:cx [10320](#), [10388](#), [10428](#), [10560](#)
\fp_exp_const:Nx [10320](#), [10620](#), [10625](#), [11173](#)
\fp_exp_decimal: [10320](#), [10405](#), [10493](#), [10508](#)
\fp_exp_integer: ... [10320](#), [10399](#), [10408](#)
\fp_exp_integer_const:n
..... [10320](#), [10431](#), [10437](#),
[10460](#), [10462](#), [10479](#), [10481](#), [10495](#)
\fp_exp_integer_const:nnnn
..... [10320](#), [10498](#), [10501](#), [10858](#)
\fp_exp_integer_tens:
.. [10320](#), [10415](#), [10434](#), [10439](#), [10443](#)
\fp_exp_integer_units: [10320](#), [10473](#), [10475](#)
\fp_exp_internal:
..... [10320](#), [10351](#), [10368](#), [11174](#)
\fp_exp_overflow_msg:
..... [10380](#), [10393](#), [11456](#), [11462](#)
\fp_exp_Taylor: [10320](#), [10538](#), [10572](#), [10617](#)
\fp_extended_normalise:
[9599](#), [9599](#), [9712](#), [10371](#), [11036](#), [11071](#)
\fp_extended_normalise_aux:NNNNNNNN
..... [9599](#)
\fp_extended_normalise_aux_i:
..... [9599](#), [9601](#), [9604](#), [9611](#)
\fp_extended_normalise_aux_i:w
..... [9599](#), [9609](#), [9614](#)
\fp_extended_normalise_aux_ii:
..... [9599](#), [9602](#), [9632](#), [9639](#)
\fp_extended_normalise_aux_ii:w
..... [9599](#), [9621](#), [9624](#)
\fp_extended_normalise_ii_aux:NNNNNNNN
..... [9637](#), [9642](#)
\fp_extended_normalise_output:
[9665](#), [9665](#), [9672](#), [10471](#), [10491](#), [11163](#)
\fp_extended_normalise_output_aux:N
..... [9665](#), [9693](#), [9695](#)
\fp_extended_normalise_output_aux_i:NNNNNNNN
..... [9665](#), [9670](#), [9675](#)
\fp_extended_normalise_output_aux_ii:NNNNNNNN
..... [9665](#), [9686](#), [9689](#)
\fp_gabs:c [186](#), [9002](#)
\fp_gabs:N [186](#), [9002](#), [9003](#), [9005](#)
\fp_gadd:cn [186](#), [9053](#)
\fp_gadd:Nn [186](#), [9053](#), [9054](#), [9056](#)
\fp_gcos:cn [189](#), [9926](#)
\fp_gcos:Nn [189](#), [9926](#), [9927](#), [9929](#)
\fp_gdiv:cn [187](#), [9287](#)
\fp_gdiv:Nn [187](#), [9287](#), [9288](#), [9290](#)
\fp_gexp:cn [188](#), [10320](#)
\fp_gexp:Nn [188](#), [10320](#), [10321](#), [10323](#)
\fp_gln:cn [188](#), [10638](#)
\fp_gln:Nn [188](#), [10638](#), [10639](#), [10641](#)
\fp_gmul:cn [187](#), [9174](#)
\fp_gmul:Nn [187](#), [9174](#), [9175](#), [9177](#)
\fp_gneg:c [186](#), [9027](#)
\fp_gneg:N [186](#), [9027](#), [9028](#), [9030](#)
\fp_gpow:cn [187](#), [10972](#)
\fp_gpow:Nn [187](#), [10972](#), [10973](#), [10975](#)
\fp_ground_figures:cn [184](#), [8883](#)
\fp_ground_figures:Nn [184](#), [8883](#), [8886](#), [8888](#)
\fp_ground_places:cn [184](#), [8918](#)
\fp_ground_places:Nn [184](#), [8918](#), [8921](#), [8923](#)
.fgset:c [170](#)
\fp_gset:cn [182](#), [8432](#)
.fgset:N [170](#)
\fp_gset:Nn ... [182](#), [8423](#), [8432](#), [8433](#), [8465](#)
\fp_gset_eq:cc [181](#), [8516](#), [8523](#)
\fp_gset_eq:cN [181](#), [8516](#), [8521](#)
\fp_gset_eq:Nc [181](#), [8516](#), [8522](#)
\fp_gset_eq:NN [181](#), [8516](#), [8520](#)
\fp_gset_from_dim:cn [182](#), [8466](#)
\fp_gset_from_dim:Nn [182](#), [8466](#), [8468](#), [8513](#)
\fp_gsin:cn [188](#), [9829](#)
\fp_gsin:Nn [188](#), [9829](#), [9830](#), [9832](#)
\fp_gsub:cn [187](#), [9156](#)
\fp_gsub:Nn [187](#), [9156](#), [9157](#), [9159](#)
\fp_gtan:cn [189](#), [10127](#)
\fp_gtan:Nn [189](#), [10127](#), [10128](#), [10130](#)
\fp_gzero:c [182](#), [8426](#)
\fp_gzero:N [182](#), [8426](#), [8428](#), [8431](#)
\fp_if_undefined:N [11176](#)
\fp_if_undefined:NTF [184](#), [11176](#)

\fp_if_undefined_p:N [184](#), [11176](#)
 \fp_if_zero:N [185](#), [11184](#)
 \fp_if_zero:NTF [11184](#)
 \fp_if_zero_p:N [11184](#)
 \fp_level_input_exponents:
 [8355](#), [8355](#), [9072](#)
 \fp_level_input_exponents_a:
 [8355](#), [8358](#), [8363](#), [8370](#)
 \fp_level_input_exponents_a:NNNNNNNN
 [8355](#), [8368](#), [8373](#)
 \fp_level_input_exponents_b:
 [8355](#), [8360](#), [8388](#), [8395](#)
 \fp_level_input_exponents_b:NNNNNNNN
 [8355](#), [8393](#), [8398](#)
 \fp_ln:cn [188](#), [10638](#)
 \fp_ln:Nn [188](#), [10638](#), [10638](#), [10640](#)
 \fp_ln_aux: [10638](#), [10656](#), [10675](#)
 \fp_ln_aux:NNn [10638](#), [10638](#), [10639](#), [10642](#)
 \fp_ln_const:nn [10755](#), [10765](#), [10816](#), [10855](#)
 \fp_ln_error_msg:
 [10663](#), [10671](#), [11464](#), [11467](#)
 \fp_ln_exponent: ... [10638](#), [10690](#), [10699](#)
 \fp_ln_exponent_tens: [10638](#)
 \fp_ln_exponent_tens:NN .. [10742](#), [10752](#)
 \fp_ln_exponent_units: [10638](#), [10750](#), [10762](#)
 \fp_ln_fixed: . [10638](#), [10870](#), [10915](#), [10922](#)
 \fp_ln_fixed_aux:NNNNNNNN
 [10638](#), [10920](#), [10925](#)
 \fp_ln_integer_const:nn [10638](#)
 \fp_ln_internal: [10638](#), [10701](#), [10733](#), [11135](#)
 \fp_ln_mantissa: ... [10638](#), [10774](#), [10810](#)
 \fp_ln_mantissa_aux:
 [10638](#), [10814](#), [10835](#), [10840](#)
 \fp_ln_mantissa_divide_two:
 [10638](#), [10839](#), [10843](#)
 \fp_ln_normalise: [10638](#),
 [10766](#), [10776](#), [10783](#), [10817](#), [10903](#)
 \fp_ln_normalise_aux:NNNNNNNN
 [10781](#), [10788](#)
 \fp_ln_nornalise_aux:NNNNNNNN .. [10638](#)
 \fp_ln_Taylor: [10638](#), [10832](#), [10861](#)
 \fp_ln_Taylor_aux:
 [10638](#), [10883](#), [10940](#), [10969](#)
 \fp_mul:cn [187](#), [9174](#)
 \fp_mul:Nn [187](#), [9174](#), [9174](#), [9176](#)
 \fp_mul:NNNNN [9478](#), [9478](#), [10029](#),
 [10076](#), [10080](#), [10576](#), [10878](#), [10943](#)
 \fp_mul:NNNNNNNN
 [9522](#), [9522](#), [10464](#), [10484](#), [10539](#), [11152](#)
 \fp_mul_aux:NNn .. [9174](#), [9174](#), [9175](#), [9178](#)
 \fp_mul_end_level: [9174](#),
 [9245](#), [9249](#), [9252](#), [9256](#), [9276](#), [9502](#),
 [9507](#), [9511](#), [9516](#), [9518](#), [9519](#), [9548](#),
 [9555](#), [9561](#), [9568](#), [9572](#), [9575](#), [9579](#)
 \fp_mul_end_level:NNNNNNNN
 [9174](#), [9280](#), [9282](#)
 \fp_mul_internal: [9174](#), [9188](#), [9229](#)
 \fp_mul_product:NN
 [9237](#)–[9239](#), [9241](#)–[9244](#), [9246](#)–[9248](#),
 [9250](#), [9251](#), [9255](#), [9271](#), [9490](#)–[9495](#),
 [9497](#)–[9501](#), [9503](#)–[9506](#), [9508](#)–[9510](#),
 [9514](#), [9515](#), [9517](#), [9534](#)–[9539](#), [9541](#)–
 [9547](#), [9549](#)–[9554](#), [9556](#)–[9560](#), [9564](#)–
 [9567](#), [9569](#)–[9571](#), [9573](#), [9574](#), [9578](#)
 \fp_mul_split:NNNN
 [9174](#), [9231](#), [9233](#), [9259](#), [9480](#), [9482](#),
 [9484](#), [9486](#), [9524](#), [9526](#), [9528](#), [9530](#)
 \fp_mul_split:w [9174](#)
 \fp_mul_split_aux:w [9262](#), [9268](#)
 \fp_neg:c [186](#), [9027](#)
 \fp_neg:N [186](#), [9027](#), [9027](#), [9029](#)
 \fp_neg:NN [9027](#)
 \fp_neg_aux:NN [9027](#), [9028](#), [9031](#)
 \fp_new:c [181](#), [8414](#)
 \fp_new:N [181](#), [8414](#), [8414](#), [8419](#), [8422](#)
 \fp_overflow_msg: [8283](#), [8350](#), [11448](#), [11454](#)
 \fp_pow:cn [187](#), [10972](#)
 \fp_pow:Nn [187](#), [10972](#), [10972](#), [10974](#)
 \fp_pow_aux:NNn [10972](#), [10972](#), [10973](#), [10976](#)
 \fp_pow_aux_i: [10972](#), [11022](#), [11027](#)
 \fp_pow_aux_ii: [10972](#), [11031](#), [11045](#), [11063](#)
 \fp_pow_aux_iii: ... [10972](#), [11088](#), [11117](#)
 \fp_pow_aux_iv: [10972](#), [11066](#),
 [11080](#), [11094](#), [11098](#), [11120](#), [11129](#)
 \fp_pow_negative: [10972](#)
 \fp_pow_positive: [10972](#)
 \fp_read:N [8201](#),
 [8201](#), [8892](#), [8927](#), [9009](#), [9034](#), [9060](#),
 [9163](#), [9181](#), [9294](#), [10979](#), [11212](#), [11217](#)
 \fp_read_aux:w [8201](#), [8202](#), [8203](#)
 \fp_round: [8895](#), [8931](#), [8954](#), [8954](#)
 \fp_round_aux:NNNNNNNN [8954](#), [8961](#), [8963](#)
 \fp_round_figures:cn [184](#), [8883](#)
 \fp_round_figures:Nn [184](#), [8883](#), [8883](#), [8885](#)
 \fp_round_figures_aux:NNn
 [8883](#), [8884](#), [8887](#), [8889](#)
 \fp_round_loop:N . [8954](#), [8965](#), [8976](#), [8999](#)
 \fp_round_places:cn [184](#), [8918](#)
 \fp_round_places:Nn [184](#), [8918](#), [8918](#), [8920](#)

\fp_round_places_aux:NNn 8918, 8919, 8922, 8924
.fp_set:c 170
\fp_set:cn 182, 8432
.fp_set:N 170
\fp_set:Nn 182, 8432, 8432, 8464
\fp_set_aux:NNn 8432, 8432-8434
\fp_set_eq:cc 181, 8516, 8519
\fp_set_eq:cN 181, 8516, 8517
\fp_set_eq:Nc 181, 8516, 8518
\fp_set_eq:NN 181, 8516, 8516
\fp_set_from_dim:cn 182, 8466
\fp_set_from_dim:Nn 182, 8466, 8466, 8512
\fp_set_from_dim_aux:NNn
..... 8466, 8467, 8469, 8470
\fp_set_from_dim_aux:w
..... 8466, 8477, 8506, 8508, 8511
\fp_show:c 183, 8524, 8525
\fp_show:N 183, 8524, 8524
\fp_sin:cn 188, 9829
\fp_sin:Nn 188, 9829, 9829, 9831
\fp_sin_aux:NNn .. 9829, 9829, 9830, 9833
\fp_sin_aux_i: 9829, 9869, 9880
\fp_sin_aux_ii: .. 9829, 9883, 9913, 10227
\fp_split:Nn
... 8214, 8214, 8437, 8475, 9061,
9164, 9182, 9295, 9836, 9933, 10134,
10327, 10645, 10984, 11195, 11201
\fp_split_aux_i:w 8214, 8253, 8257
\fp_split_aux_ii:w 8214, 8258, 8259
\fp_split_aux_iii:w 8214, 8260, 8261
\fp_split_decimal:w 8214, 8264, 8267
\fp_split_decimal_aux:w 8214, 8268, 8269
\fp_split_exponent: 8214
\fp_split_exponent:w 8222, 8249
\fp_split_sign: 8214, 8220, 8224, 8235, 8245
\fp_standardise:NNNN
..... 8286, 8286, 8438, 8480,
9062, 9082, 9165, 9183, 9193, 9296,
9336, 9837, 9891, 9934, 9977, 10135,
10222, 10231, 10258, 10328, 10555,
10646, 10711, 10985, 11196, 11202
\fp_standardise_aux: 8286, 8300,
8306, 8316, 8317, 8323, 8340, 8353
\fp_standardise_aux:NNNN 8286, 8294, 8298
\fp_standardise_aux:w
.. 8286, 8304, 8310, 8322, 8327, 8354
\fp_sub:cn 186, 9156
\fp_sub:Nn 186, 9156, 9156, 9158
\fp_sub:NNNNNNNN 9455,
9455, 9794, 9805, 9816, 10821, 10907
\fp_sub_aux:NNn .. 9156, 9156, 9157, 9160
\fp_tan:cn 189, 10127
\fp_tan:Nn 189, 10127, 10127, 10129
\fp_tan_aux:NNn 10127, 10127, 10128, 10131
\fp_tan_aux_i: 10127, 10167, 10178
\fp_tan_aux_ii: 10127, 10181, 10188
\fp_tan_aux_iii: 10127, 10211, 10214, 10217
\fp_tan_aux_iv: 10127, 10241, 10244, 10247
\fp_tmp:w 8413, 8413,
8444, 8462, 8486, 8504, 8898, 8916,
8934, 8952, 9011, 9025, 9068, 9087,
9172, 9198, 9227, 9305, 9315, 9324,
9341, 9858, 9871, 9878, 9958, 9964,
10156, 10169, 10176, 10353, 10366,
10658, 10666, 10673, 10692, 10884,
10895, 10998, 11004, 11015, 11025,
11048, 11055, 11101, 11136, 11150
\fp_to_dim:c 183, 8589
\fp_to_dim:N 183, 8589, 8589, 8590
\fp_to_int:c 183, 8591
\fp_to_int:N 183, 8591, 8591, 8593
\fp_to_int_aux:w 8591, 8592, 8594
\fp_to_int_large:w 8591, 8602, 8617
\fp_to_int_large_aux:nnn
..... 8591, 8629, 8631, 8633,
8635, 8637, 8639, 8641, 8643, 8645
\fp_to_int_large_aux_1:w 8591
\fp_to_int_large_aux_2:w 8591
\fp_to_int_large_aux_3:w 8591
\fp_to_int_large_aux_4:w 8591
\fp_to_int_large_aux_5:w 8591
\fp_to_int_large_aux_6:w 8591
\fp_to_int_large_aux_7:w 8591
\fp_to_int_large_aux_8:w 8591
\fp_to_int_large_aux_i:w 8591, 8620, 8626
\fp_to_int_large_aux_ii:w 8591, 8622, 8653
\fp_to_int_none:w 8591
\fp_to_int_small:w 8591, 8600, 8606
\fp_to_tl:c 183, 8658
\fp_to_tl:N 183, 8658, 8658, 8660
\fp_to_tl_aux:w 8658, 8659, 8661
\fp_to_tl_large:w 8658, 8669, 8673
\fp_to_tl_large_0:w 8658
\fp_to_tl_large_1:w 8658
\fp_to_tl_large_2:w 8658
\fp_to_tl_large_3:w 8658
\fp_to_tl_large_4:w 8658
\fp_to_tl_large_5:w 8658

- \fp_to_tl_large_6:w [8658](#)
 - \fp_to_tl_large_7:w [8658](#)
 - \fp_to_tl_large_8:w [8658](#)
 - \fp_to_tl_large_8_aux:w [8658](#)
 - \fp_to_tl_large_9:w [8658](#)
 - \fp_to_tl_large_aux_i:w [8658](#), [8676](#), [8682](#)
 - \fp_to_tl_large_aux_ii:w [8658](#), [8678](#), [8684](#)
 - \fp_to_tl_large_zeros:NNNNNNNN [8658](#),
[8687](#), [8693](#), [8698](#), [8703](#), [8708](#), [8713](#),
[8718](#), [8723](#), [8728](#), [8738](#), [8796](#), [8799](#)
 - \fp_to_tl_small:w [8658](#), [8667](#), [8741](#)
 - \fp_to_tl_small_aux:w .. [8658](#), [8749](#), [8793](#)
 - \fp_to_tl_small_one:w .. [8658](#), [8744](#), [8754](#)
 - \fp_to_tl_small_two:w .. [8658](#), [8747](#), [8773](#)
 - \fp_to_tl_small_zeros:NNNNNNNN
.. [8658](#), [8761](#), [8770](#), [8780](#), [8790](#), [8838](#)
 - \fp_trig_calc_cos:
[9919](#), [9921](#), [10003](#), [10009](#), [10022](#), [10022](#)
 - \fp_trig_calc_sin:
[9917](#), [9923](#), [10005](#), [10007](#), [10022](#), [10058](#)
 - \fp_trig_calc_Taylor:
.. [10022](#), [10055](#), [10070](#), [10073](#), [10124](#)
 - \fp_trig_normalise:
..... [9708](#), [9708](#), [9882](#), [9968](#), [10190](#)
 - \fp_trig_normalise_aux:
..... [9708](#), [9713](#), [9727](#), [9732](#), [9740](#)
 - \fp_trig_octant: [9718](#), [9784](#), [9784](#)
 - \fp_trig_octant_aux:
..... [9784](#), [9787](#), [9802](#), [9812](#), [9825](#)
 - \fp_trig_overflow_msg:
..... [9724](#), [10185](#), [11470](#), [11476](#)
 - \fp_trig_sub:NNN . [9708](#), [9730](#), [9736](#), [9745](#)
 - \fp_use:c [182](#), [8526](#)
 - \fp_use:N [182](#), [8526](#), [8526](#), [8528](#), [8589](#)
 - \fp_use_aux:w [8526](#), [8527](#), [8529](#)
 - \fp_use_i_to_iix:NNNNNNNN
..... [8658](#), [8758](#), [8763](#), [8881](#)
 - \fp_use_i_to_vii:NNNNNNNN
..... [8658](#), [8777](#), [8782](#), [8879](#)
 - \fp_use_iix_ix:NNNNNNNN [8658](#), [8775](#), [8877](#)
 - \fp_use_ix:NNNNNNNN ... [8658](#), [8756](#), [8878](#)
 - \fp_use_large:w [8526](#), [8535](#), [8553](#)
 - \fp_use_large_aux_1:w [8526](#)
 - \fp_use_large_aux_2:w [8526](#)
 - \fp_use_large_aux_3:w [8526](#)
 - \fp_use_large_aux_4:w [8526](#)
 - \fp_use_large_aux_5:w [8526](#)
 - \fp_use_large_aux_6:w [8526](#)
 - \fp_use_large_aux_7:w [8526](#)
 - \fp_use_large_aux_8:w [8526](#)
 - \fp_use_large_aux_i:w .. [8526](#), [8556](#), [8562](#)
 - \fp_use_large_aux_ii:w . [8526](#), [8558](#), [8583](#)
 - \fp_use_none:w [8526](#), [8541](#), [8546](#)
 - \fp_use_small:w [8526](#), [8539](#), [8547](#)
 - \fp_zero:c [182](#), [8426](#)
 - \fp_zero:N [182](#), [8426](#), [8426](#), [8430](#)
 - \frozen@everydisplay [764](#)
 - \frozen@everymath [765](#)
 - \futurelet [361](#)
- G**
- \G [2721](#)
 - \g [2265](#)
 - \g_cctab_allocate_int [11508](#),
[11508](#), [11509](#), [11516](#), [11518](#), [11520](#)
 - \g_cctab_stack_int [11508](#), [11510](#),
[11546](#), [11547](#), [11549](#), [11550](#), [11554](#)
 - \g_cctab_stack_seq
..... [11508](#), [11511](#), [11544](#), [11555](#)
 - \g_clist_map_inline_int
..... [5750](#), [5750](#), [5753](#), [5754](#),
[5758](#), [5759](#), [5763](#), [5764](#), [5768](#), [5769](#)
 - \g_file_current_name_tl
..... [179](#), [8018](#), [8018](#),
[8023](#), [8027](#), [8035](#), [8097](#), [8098](#), [8100](#)
 - \g_file_record_seq [180](#), [8030](#),
[8030](#), [8035](#), [8092](#), [8112](#), [8114](#), [8121](#)
 - \g_file_stack_seq
..... [180](#), [8029](#), [8029](#), [8097](#), [8100](#)
 - \g_file_test_stream [8043](#),
[8045](#), [8046](#), [8049](#), [8066](#), [8067](#), [8077](#)
 - \g_ior_streams_prop [6400](#), [6401](#),
[6406](#), [6433](#), [6535](#), [6549](#), [6570](#), [6578](#)
 - \g_ior_tmp_stream [6468](#), [6514](#), [6515](#), [6518](#)
 - \g_iow_streams_prop
..... [6400](#), [6400](#), [6403](#)–[6405](#), [6446](#),
[6454](#), [6462](#), [6498](#), [6561](#), [6592](#), [6600](#)
 - \g_iow_tmp_stream [6468](#), [6477](#), [6478](#), [6481](#)
 - \g_keyval_level_int [7363](#), [7363](#),
[7410](#), [7444](#), [7472](#), [7474](#), [7476](#), [7478](#)
 - \g_peek_token [62](#), [2869](#), [2870](#), [2880](#)
 - \g_prg_stepwise_level_int
.. [2201](#), [2201](#), [2204](#), [2206](#), [2211](#), [2213](#)
 - \g_prop_map_inline_int
.. [6084](#), [6084](#), [6087](#), [6088](#), [6091](#), [6092](#)
 - \g_seq_nesting_depth_int
.. [3847](#), [5209](#), [5221](#), [5223](#), [5227](#), [5229](#)
 - \g_tl_inline_level_int
..... [3848](#), [4530](#), [4532](#), [4533](#),
[4536](#), [4538](#), [4542](#), [4543](#), [4546](#), [4548](#)

- \g_tmpa_bool [41](#), [1910](#), 1911
 - \g_tmpa_dim [85](#), [4052](#), 4055
 - \g_tmpa_int [77](#), [3842](#), 3845
 - \g_tmpa_skip [88](#), [4126](#), 4129
 - \g_tmpa_tl [108](#), [4784](#), 4784
 - \g_tmpb_dim [85](#), [4052](#), 4056
 - \g_tmpb_int [77](#), [3842](#), 3846
 - \g_tmpb_skip [88](#), [4126](#), 4130
 - \g_tmpb_tl [108](#), [4784](#), 4785
 - \gdef 352
 - .generate_choices:n 170
 - \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:
..... [46](#), [1922](#), [2252](#), 2252, 2901, 2919
 - \group_align_safe_end: [46](#), 2019, 2020,
[2252](#), 2254, 2883, 2893, 2898, 2916
 - \group_begin: [8](#), [811](#), 812, 855,
1072, 1812, 2264, 2278, 2587, 2600,
2624, 2677, 2714, 2811, 2977, 3074,
3081, 4295, 4312, 4339, 4352, 4483,
5086, 6649, 6868, 6915, 7319, 7368,
7378, 7422, 7460, 8436, 8472, 8891,
8926, 9008, 9033, 9059, 9162, 9180,
9293, 9835, 9932, 10133, 10326,
10644, 10863, 10978, 11035, 11069,
11131, 11194, 11211, 11398, 11575
 - \group_end: [8](#), [811](#), 813, 858, 1077,
1824, 2269, 2283, 2599, 2603, 2631,
2685, 2725, 2817, 3042, 3083, 3092,
4302, 4319, 4345, 4358, 4487, 4490,
5097, 6671, 6873, 6921, 7342, 7375,
7386, 7438, 7469, 8446, 8488, 8900,
8936, 9013, 9050, 9089, 9200, 9307,
9317, 9343, 9860, 9873, 9960, 10158,
10171, 10355, 10660, 10668, 10694,
10886, 11000, 11006, 11017, 11041,
11047, 11050, 11057, 11074, 11082,
11091, 11103, 11138, 11225, 11236,
11239, 11243, 11247, 11251, 11263,
11267, 11270, 11279, 11282, 11293,
11297, 11311, 11315, 11319, 11324,
11329, 11332, 11343, 11347, 11351,
11356, 11361, 11364, 11401, 11578
 - \group_execute_after:N 1516
 - \group_insert_after:N . . . [8](#), [816](#), 816, 1516
- ## H
- \H 2721
 - \halign 378
 - \hangafter 556
 - \hangindent 557
 - \hbadness 618
 - \hbox 613
 - \hbox:n [147](#), [6303](#), 6303
 - \hbox_gset:cn [147](#), [6304](#)
 - \hbox_gset:Nn [147](#), [6304](#), 6305, 6307
 - \hbox_gset_inline_begin:c [148](#), [6314](#)
 - \hbox_gset_inline_begin:N
..... [148](#), [6314](#), 6316, 6319
 - \hbox_gset_inline_end: . . [148](#), [6314](#), 6321
 - \hbox_gset_to_wd:cnn [147](#), [6308](#)
 - \hbox_gset_to_wd:Nnn [147](#), [6308](#), 6310, 6313
 - \hbox_overlap_left:n . . . [148](#), [6325](#), 6325
 - \hbox_overlap_right:n . . [147](#), [6325](#), 6327
 - \hbox_set:cn [147](#), [6304](#)
 - \hbox_set:Nn [147](#), [6304](#), 6304–6306
 - \hbox_set_inline_begin:c [148](#), [6314](#)
 - \hbox_set_inline_begin:N
..... [148](#), [6314](#), 6314, 6317, 6318
 - \hbox_set_inline_end: . . [148](#), [6314](#), 6320
 - \hbox_set_to_wd:cnn [147](#), [6308](#)
 - \hbox_set_to_wd:Nnn
..... [147](#), [6308](#), 6308, 6311, 6312
 - \hbox_to_wd:nn [147](#), [6322](#), 6322
 - \hbox_to_zero:n [147](#), [6322](#), 6324, 6326, 6328
 - \hbox_unpack:c [148](#), [6329](#)
 - \hbox_unpack:N [148](#), [6329](#), 6329, 6331
 - \hbox_unpack_clear:c [148](#), [6329](#)
 - \hbox_unpack_clear:N [148](#), [6329](#), 6330, 6332
 - \hfil 521
 - \hfill 523
 - \hfilneg 522
 - \hfuzz 620
 - \hoffset 595
 - \holdinginserts 598
 - \hrule 534

<code>\hsize</code>	559	9412, 9444, 9449, 9460, 9464, 9468,
<code>\hskip</code>	524	9469, 9594, 9606, 9634, 9645, 9667,
<code>\hss</code>	525	9710, 9714, 9729, 9734, 9735, 9753,
<code>\ht</code>	663	9757, 9761, 9763, 9788, 9804, 9814,
<code>\hyphenation</code>	649	9844, 9857, 9884, 9899, 9941, 9970,
<code>\hyphenchar</code>	633	9985, 10011, 10012, 10016, 10024,
<code>\hyphenpenalty</code>	551	10038, 10039, 10061, 10094, 10095,
		10099, 10105, 10115, 10119, 10142,
		10155, 10180, 10191, 10192, 10198,
		10205, 10206, 10236, 10237, 10266,
		10336, 10370, 10372, 10373, 10383,
		10398, 10410, 10426, 10427, 10449,
		10459, 10477, 10478, 10510, 10511,
		10517, 10546, 10549, 10584, 10588,
		10592, 10598, 10608, 10612, 10651,
		10652, 10702, 10705, 10718, 10735,
		10741, 10764, 10778, 10790, 10815,
		10818, 10829, 10837, 10897, 10904,
		10917, 10927, 10947, 10957, 10963,
		10990, 10994, 11011, 11029, 11034,
		11037, 11065, 11068, 11072, 11073,
		11231–11234, 11257, 11260, 11266,
		11275, 11278, 11292, 11296, 11300,
		11310, 11314, 11318, 11322, 11327,
		11342, 11346, 11350, 11354, 11359
<code>\I</code>	2721	
<code>\if</code>	184, 387	
<code>\if:w</code>	25, <u>784</u> , 790,	
	981, 1049, 1065, 1778, 1806, 2854,	
	2959, 3346, 8205, 8531, 8596, 8663	
<code>\if_bool:N</code>	26, <u>784</u> , 791, 1900	
<code>\if_box_empty:N</code> ...	152, <u>6260</u> , 6262, 6276	
<code>\if_case:w</code>	78,	
	1320, <u>3196</u> , 3201, 3598, 9915, 10001	
<code>\if_catcode:w</code>	25, <u>784</u> ,	
	794, 1832, 2407, 2420, 2606, 2611,	
	2616, 2621, 2628, 2634, 2639, 2644,	
	2649, 2654, 2659, 2669, 2702, 2928,	
	4683, 4721, 6036, 6076, 6157, 6741	
<code>\if_charcode:w</code>		
	25, <u>784</u> , 793, 2674, 4661, 4667, 4714	
<code>\if_cs_exist:N</code> <u>26</u> , <u>800</u> , 800, 1105, 1133, 2863		
<code>\if_cs_exist:w</code>		
	26, <u>800</u> , 801, 1114, 1142, 4569,	
	9864, 9954, 10162, 10349, 10358, 10688	
<code>\if_dim:w</code> <u>91</u> , <u>3895</u> , 3895, 3949, 3974–3980		
<code>\if_eof:w</code>	157, <u>6375</u> , 6375, 6754	
<code>\if_false:</code>	25, <u>784</u> , 785, 2253,	
	4764, 4765, 5063, 5066, 5169, 5173	
<code>\if_hbox:N</code>	151, <u>6260</u> , 6260, 6264	
<code>\if_int_compare:w</code>	77,	
	814, 814, 1485, 1491, 2253, 2255,	
	2461, 2470, 2693, <u>3196</u> , 3211, 3219,	
	3230, 3241, 3245, 3246, 3252, 3356,	
	3364, 3372, 3380, 3388, 3396, 3404,	
	3412, 4092, 4751, 4790, 8226, 8237,	
	8272, 8280, 8288, 8302, 8319, 8341,	
	8342, 8357, 8365, 8390, 8449, 8491,	
	8534, 8537, 8555, 8599, 8608, 8610,	
	8619, 8647, 8666, 8675, 8743, 8746,	
	8756, 8757, 8775, 8776, 8801–8809,	
	8840–8848, 8894, 8903, 8930, 8939,	
	8969, 8978, 8982, 8991, 8992, 8998,	
	9038, 9073, 9092, 9118, 9134, 9138,	
	9140, 9203, 9207, 9301, 9311, 9346,	
	9350, 9381, 9384, 9392, 9395, 9397,	
		9412, 9444, 9449, 9460, 9464, 9468,
		9469, 9594, 9606, 9634, 9645, 9667,
		9710, 9714, 9729, 9734, 9735, 9753,
		9757, 9761, 9763, 9788, 9804, 9814,
		9844, 9857, 9884, 9899, 9941, 9970,
		9985, 10011, 10012, 10016, 10024,
		10038, 10039, 10061, 10094, 10095,
		10099, 10105, 10115, 10119, 10142,
		10155, 10180, 10191, 10192, 10198,
		10205, 10206, 10236, 10237, 10266,
		10336, 10370, 10372, 10373, 10383,
		10398, 10410, 10426, 10427, 10449,
		10459, 10477, 10478, 10510, 10511,
		10517, 10546, 10549, 10584, 10588,
		10592, 10598, 10608, 10612, 10651,
		10652, 10702, 10705, 10718, 10735,
		10741, 10764, 10778, 10790, 10815,
		10818, 10829, 10837, 10897, 10904,
		10917, 10927, 10947, 10957, 10963,
		10990, 10994, 11011, 11029, 11034,
		11037, 11065, 11068, 11072, 11073,
		11231–11234, 11257, 11260, 11266,
		11275, 11278, 11292, 11296, 11300,
		11310, 11314, 11318, 11322, 11327,
		11342, 11346, 11350, 11354, 11359
<code>\if_int_odd:w</code>	78, <u>3196</u> ,	
	3200, 3420, 3428, 9792, 10845, 10848	
<code>\if_meaning:w</code>	25,	
	<u>795</u> , 795, 1085, 1102, 1120, 1130,	
	1148, 1423, 1560, 1561, 1850, 1930,	
	1940, 1949, 1952, 1962, 1965, 1978,	
	1987, 2392, 2398, 2441, 2449, 2664,	
	2709, 2742, 2753, 2764, 2775, 2837,	
	2937, 3353, 3969, 4436, 4448, 4460,	
	4471, 4486, 4706, 4830, 5091, 5191,	
	5305, 5697, 5710, 6018, 11178, 11186	
<code>\if_mode_horizontal:</code> ..	<u>26</u> , <u>796</u> , 797, 2244	
<code>\if_mode_inner:</code>	<u>26</u> , <u>796</u> , 799, 2246	
<code>\if_mode_math:</code>	<u>26</u> , <u>796</u> , 796, 2250	
<code>\if_mode_vertical:</code>	<u>26</u> , <u>796</u> , 798, 2242	
<code>\if_num:w</code>	77, 2845, <u>3196</u> , 3199	
<code>\if_predicate:w</code>		
	25, <u>784</u> , 792, 1304, 1914, 4735	
<code>\if_true:</code>	25, <u>784</u> , 784	
<code>\if_vbox:N</code>	152, <u>6260</u> , 6261, 6266	
<code>\ifcase</code>	388	
<code>\ifcat</code>	389	
<code>\ifcsname</code>	672	
<code>\ifdefined</code>	671	
<code>\ifdim</code>	392	

<code>\ifeof</code>	393	<code>\int_convert_from_base_ten:nn</code>	3849, 3849
<code>\iffalse</code>	398	<code>\int_convert_to_base_ten:nn</code> .	3849, 3851
<code>\iffontchar</code>	701	<code>\int_convert_to_symbols:nnn</code> .	3849, 3850
<code>\ifhbox</code>	394	<code>\int_decr:c</code>	69, 3321
<code>\ifhmode</code>	400	<code>\int_decr:N</code>	69, 3321, 3323, 3328, 3330
<code>\ifinner</code>	403	<code>\int_div_round:nn</code>	67, 3238, 3263
<code>\ifmmode</code>	401	<code>\int_div_truncate:nn</code>	67, 3238, 3238, 3267, 3496, 3590
<code>\ifnum</code>	390	<code>\int_do_until:nn</code> ...	72, 3434, 3456, 3460
<code>\ifodd</code>	169, 205, 391	<code>\int_do_until:nNnn</code> ..	71, 3462, 3484, 3488
<code>\iftrue</code>	399	<code>\int_do_while:nn</code>	72, 3434, 3450
<code>\ifvbox</code>	395	<code>\int_do_while:nNnn</code>	71, 3454, 3462, 3478, 3482
<code>\ifvmode</code>	402	<code>\int_eval:n</code>	66, 1354, 2091, 2181, 2189, 2198, 2212, 2221, 2230, 3202, 3203, 3206, 3263, 3490, 3498, 3501, 3568, 3637, 3647, 3706, 3720, 3724, 3727, 3742, 3751, 4577, 4582, 4896, 5373, 5385, 5819, 5827, 5842, 5854
<code>\ifvoid</code>	396	<code>\int_eval:w</code> 78, 1320, 2140, 2491, 2493, 2495, 2561, 2563, 2565, 2567, 2569, 2571, 2573, 2575, 2577, 2579, 2581, 2583, 3196, 3197, 3203, 3206, 3211, 3214, 3218, 3220, 3229, 3231, 3240, 3241, 3245, 3246, 3252, 3266, 3290, 3310, 3312, 3334, 3341, 3356, 3364, 3372, 3380, 3388, 3396, 3404, 3412, 3420, 3428, 3598, 3780, 6734, 8252, 8255, 8273, 8289, 8650, 8758, 8762, 8777, 8781, 8980, 8981, 9074, 9100, 9111, 9115, 9127, 9131, 9144, 9148, 9190, 9204, 9208, 9221, 9274, 9302, 9312, 9333, 9347, 9351, 9364, 9382, 9427, 9436, 9441–9443, 9457–9459, 9469, 9473, 9474, 9586, 9593, 9618, 9628, 9748, 9750, 9752, 9764, 9770, 9774, 9778, 9852, 9907, 9949, 9993, 10150, 10255, 10274, 10344, 10423, 10456, 10519, 10525, 10529, 10566, 10585, 10653, 10683, 10726, 10830, 10948, 10991, 10995, 11012, 11038, 11110, 11160, 11257, 11260, 11275	
<code>\ifx</code>	13, 62, 108, 135, 229, 233, 397	<code>\int_eval_end:</code>	78, 1320, 2140, 2491, 2493, 2495, 2561, 2563, 2565, 2567, 2569, 2571, 2573, 2575, 2577, 2579, 2581, 2583, 3196, 3198, 3203, 3206, 3214, 3220, 3225, 3231, 3236, 3261, 3268, 3290, 3310, 3312, 3334, 3356, 3364, 3372, 3380, 3388, 3396, 3404, 3412,
<code>\ignorespaces</code>	445		
<code>\immediate</code>	407		
<code>\indent</code>	541		
<code>\initcatcodetable</code>	757		
<code>\input</code>	415		
<code>\input@path</code>	8057, 8060, 8074		
<code>\inputlineno</code>	417		
<code>\insert</code>	597		
<code>\insertpenalties</code>	600		
<code>\int_abs:n</code>	67, 3208, 3208		
<code>\int_add:cn</code>	68, 3309		
<code>\int_add:Nn</code> 68, 3309, 3309, 3314, 3317, 6689			
<code>\int_compare:n</code>	3340		
<code>\int_compare:nF</code>	3444, 3459		
<code>\int_compare:nNn</code>	3410		
<code>\int_compare:nNnF</code>	2185, 2194, 2217, 2226, 2260, 2261, 3472, 3487, 6546, 6558		
<code>\int_compare:nNnT</code> ..	2258, 3464, 3481, 5387, 6475, 6494, 6512, 6531, 11547		
<code>\int_compare:nNnTF</code>	70, 2047, 2096, 2178, 2208, 3280, 3282, 3410, 3493, 3571, 3577, 3724, 3748, 3752, 3802, 5400, 5839, 5849, 6429, 6442, 6690, 11517		
<code>\int_compare:nT</code>	3436, 3453		
<code>\int_compare:nTF</code>	70, 3340		
<code>\int_compare_<:w</code>	3340		
<code>\int_compare_=:w</code>	3340		
<code>\int_compare_>:w</code>	3340		
<code>\int_compare_aux:Nw</code>	3340, 3344, 3352		
<code>\int_compare_aux:nw</code>	3340, 3341, 3342		
<code>\int_compare_p:n</code>	70, 3340		
<code>\int_compare_p:nNn</code> ..	70, 3410, 4104, 4105		
<code>\int_const:cn</code>	68, 3278, 3761–3774		
<code>\int_const:Nn</code>	68, 3278, 3278, 3298, 3823–3841, 8131–8135		

- 3420, 3428, 3598, 6737, 8650, 8764,
 8783, 9366, 9430, 9436, 9441–9443,
 9457–9459, 9473, 9474, 9586, 9593,
 9748, 9750, 9752, 9772, 9776, 9780,
 10257, 10425, 10458, 10521, 10531
 \int_from_alph:n 75, 3704, 3704
 \int_from_alph_aux:N ... 3704, 3720, 3723
 \int_from_alph_aux:n ... 3704, 3709, 3712
 \int_from_alph_aux:nn
 3704, 3713, 3714, 3719
 \int_from_base:nn
 75, 3725, 3725, 3756, 3758, 3760, 3851
 \int_from_base_aux:N ... 3725, 3742, 3746
 \int_from_base_aux:nn ... 3725, 3730, 3734
 \int_from_base_aux:nnN
 3725, 3735, 3736, 3741
 \int_from_binary:n 75, 3755, 3755
 \int_from_hexadecimal:n . 75, 3755, 3757
 \int_from_octal:n 75, 3755, 3759
 \int_from_roman:n 75, 3775, 3775
 \int_from_roman_aux:NN
 3775, 3781, 3784, 3809, 3813
 \int_from_roman_clean_up:w
 3775, 3792, 3799, 3801, 3820
 \int_from_roman_end:w .. 3775, 3779, 3818
 \int_gadd:cn 69, 3309
 \int_gadd:Nn
 .. 69, 3309, 3313, 3318, 11516, 11546
 \int_gdecr:c 69, 3321
 \int_gdecr:N
 . 69, 2213, 3321, 3327, 3332, 4538,
 4548, 5227, 5759, 5769, 6092, 7478
 \int_get_digits:n 3670, 3675, 3709, 3731
 \int_get_sign:n .. 3670, 3670, 3708, 3729
 \int_get_sign_and_digits_aux:nnn
 3670, 3672, 3677, 3680, 3703
 \int_get_sign_and_digits_aux:onn
 3670, 3686, 3690, 3696
 \int_gincr:c 69, 3321
 \int_gincr:N
 . 69, 2204, 3321, 3325, 3331, 4532,
 4542, 5223, 5753, 5763, 6087, 7472
 .int_gset:c 171
 \int_gset:cn 69, 3333
 .int_gset:N 171
 \int_gset:Nn 69, 3285, 3295, 3333, 3335, 3337
 \int_gset_eq:cc 68, 3303
 \int_gset_eq:cN 68, 3303
 \int_gset_eq:Nc 68, 3303
 \int_gset_eq:NN 68, 3303, 3306–3308
 \int_gsub:cn 70, 3309
 \int_gsub:Nn .. 70, 3309, 3315, 3320, 11554
 \int_gzero:c 68, 3299
 \int_gzero:N 68, 3299, 3300, 3302
 \int_if_even:n 3426
 \int_if_even:nTF 71, 3418
 \int_if_even_p:n 71, 3418
 \int_if_odd:n 3418
 \int_if_odd:nTF 71, 3418
 \int_if_odd_p:n 71, 3418
 \int_incr:c 69, 3321
 \int_incr:N 69, 3321, 3321, 3326, 3329,
 6493, 6530, 6706, 7615, 7642, 7709
 \int_max:nn 67, 3208, 3216
 \int_min:nn 67, 3208, 3227
 \int_mod:nn 67, 3238, 3264, 3498, 3581
 \int_new:c 68, 3270
 \int_new:N 68, 2201, 3270, 3271, 3277,
 3284, 3294, 3842–3848, 5750, 6084,
 6408, 6636, 6638–6640, 7363, 7491,
 8136, 8138, 8140, 8142, 8144, 8146,
 8154–8182, 8184–8188, 8191, 8192,
 8194, 8195, 8197–8200, 11508, 11510
 .int_set:c 170
 \int_set:cn 69, 3333
 .int_set:N 170
 \int_set:Nn 69,
 3333, 3333, 3335, 3336, 6427, 6440,
 6456, 6464, 6478, 6515, 6637, 6650,
 6687, 6951, 7611, 7637, 7705, 8137,
 8139, 8141, 8143, 8145, 8147, 8893,
 8928, 11376, 11378, 11380, 11382,
 11384, 11386, 11388, 11390, 11509
 \int_set_eq:cc 68, 3303
 \int_set_eq:cN 68, 3303
 \int_set_eq:Nc 68, 3303
 \int_set_eq:NN . 68, 3303, 3303–3305, 6713
 \int_show:c 75, 3821, 3822
 \int_show:N 75, 1447, 3821, 3821
 \int_sub:cn 69, 3309
 \int_sub:Nn 69, 3309, 3311, 3316, 3319
 \int_to_Alph:n 72, 3503, 3535
 \int_to_alph:n 72, 3503, 3503
 \int_to_arabic:n 72, 3490, 3490
 \int_to_base:nn
 74, 3567, 3567, 3629, 3631, 3633, 3849
 \int_to_base_aux_i:nn .. 3567, 3568, 3569
 \int_to_base_aux_ii:nnN
 3567, 3572, 3573, 3575, 3589
 \int_to_base_aux_iii:nnnN 3567, 3580, 3587

- \int_to_binary:n 74, 3628, 3628
- \int_to_hexadecimal:n . . . 74, 3628, 3630
- \int_to_letter:n . . 3567, 3578, 3581, 3595
- \int_to_octal:n 74, 3628, 3632
- \int_to_Roman:n 74, 3634, 3644
- \int_to_roman:n 74, 3634, 3634
- \int_to_roman:w 77, 814, 815,
896, 898, 1065, 1071, 1081, 2002,
2007, 2138, 3196, 3345, 3637, 3647
- \int_to_Roman_aux:N 3646, 3649, 3652
- \int_to_roman_aux:N 3634, 3636, 3639, 3642
- \int_to_Roman_c:w 3634, 3666
- \int_to_roman_c:w 3634, 3658
- \int_to_Roman_d:w 3634, 3667
- \int_to_roman_d:w 3634, 3659
- \int_to_Roman_i:w 3634, 3662
- \int_to_roman_i:w 3634, 3654
- \int_to_Roman_l:w 3634, 3665
- \int_to_roman_l:w 3634, 3657
- \int_to_Roman_m:w 3634, 3668
- \int_to_roman_m:w 3634, 3660
- \int_to_Roman_Q:w 3634, 3669
- \int_to_roman_Q:w 3634, 3661
- \int_to_Roman_v:w 3634, 3663
- \int_to_roman_v:w 3634, 3655
- \int_to_Roman_x:w 3634, 3664
- \int_to_roman_x:w 3634, 3656
- \int_to_symbol:n 3852, 3852
- \int_to_symbol_math:n . . 3852, 3855, 3858
- \int_to_symbol_text:n . . 3852, 3856, 3873
- \int_to_symbols:nnn . . 73, 3491, 3491,
3495, 3505, 3537, 3850, 3860, 3875
- \int_until_do:nn . . . 72, 3434, 3442, 3447
- \int_until_do:nNnn . . 71, 3462, 3470, 3475
- \int_use:c 70, 3338, 3339
- \int_use:N 70, 2206, 2211, 3338,
3338, 3339, 4533, 4536, 4543, 4546,
5221, 5229, 5754, 5758, 5764, 5768,
6088, 6091, 6470, 6471, 6480, 6485,
6487, 6496, 6507, 6508, 6517, 6522,
6524, 6533, 6853, 7410, 7444, 7474,
7476, 7612, 7638, 7706, 8265, 8305,
8322, 8335, 8369, 8383, 8394, 8408,
8454, 8457, 8459, 8496, 8499, 8501,
8908, 8911, 8913, 8944, 8947, 8949,
8961, 8988, 9017, 9020, 9022, 9043,
9046, 9048, 9097, 9103, 9218, 9224,
9268, 9280, 9361, 9368, 9380, 9610,
9622, 9638, 9650, 9660, 9671, 9684,
9703, 9849, 9855, 9904, 9910, 9946,
9952, 9990, 9996, 10147, 10153,
10271, 10277, 10341, 10347, 10420,
10453, 10479, 10482, 10563, 10569,
10680, 10686, 10723, 10730, 10743,
10765, 10782, 10795, 10805, 10816,
10889, 10891, 10893, 10921, 10932,
11107, 11113, 11140, 11142, 11144,
11146, 11148, 11377, 11379, 11381,
11383, 11385, 11387, 11389, 11391
- \int_value:w 78, 1973, 1974,
1994, 1996–1998, 2140, 3196, 3196,
3203, 3206, 3210, 3218, 3229, 3240,
3266, 3341, 3780, 3946, 6734, 8650,
8762, 8781, 9100, 9221, 9364, 9852,
9907, 9949, 9993, 10150, 10274,
10344, 10566, 10683, 10726, 11110
- \int_while_do:nn . . . 72, 3434, 3434, 3439
- \int_while_do:nNnn . . 71, 3462, 3462, 3467
- \int_zero:c 68, 3299
- \int_zero:N 68, 3299, 3299,
3301, 6651, 6720, 7605, 7624, 7699
- \interactionmode 699
- \interlinepenalties 720
- \interlinepenalty 579
- \ior_alloc_read:n 6428, 6452, 6460
- \ior_close:c 153
- \ior_close:N
. . . 153, 6426, 6542, 6566, 8049, 8077
- \ior_gto:NN 156, 6762, 6764
- \ior_if_eof:N 6750
- \ior_if_eof:Nf 8067
- \ior_if_eof:Nf 156, 8046
- \ior_if_eof_p:N 156
- \ior_if_eof_p:Nf 6750
- \ior_if_eof_p_p:N 6750
- \ior_list_streams: . . 153, 6568, 6568, 6778
- \ior_new:c 6774, 6775
- \ior_new:N 6774, 6774
- \ior_open:cn 153, 6424
- \ior_open:Nn
. . . 153, 6424, 6424, 6450, 8045, 8066
- \ior_open_streams: 6778, 6778
- \ior_raw_new:c 157, 6410, 6522
- \ior_raw_new:N
. . . 157, 6410, 6412, 6420, 6422, 6514
- \ior_show_aux:nn . . 6568, 6578, 6583, 6605
- \ior_str_gto:NN 156, 6766, 6768
- \ior_str_to:NN 156, 6766, 6766
- \ior_stream_alloc:N 6432, 6468, 6505

- \ior_stream_alloc_aux: 6468, 6511, 6528, 6536, 6538
 - \ior_to:NN 155, 6762, 6762
 - \iow_alloc_write:n 6441, 6452, 6452
 - \iow_char:N 154, 5299, 5813, 6120, 6122, 6586, 6749, 6749
 - \iow_close:c 153, 6542
 - \iow_close:N .. 153, 6439, 6542, 6554, 6567
 - \iow_list_streams: . 153, 6568, 6590, 6779
 - \iow_log:n ... 154, 6628, 6629, 8113–8115
 - \iow_log:x 154, 1169, 1169, 1209, 1796, 6628, 6628, 6939, 6941, 6942
 - \iow_new:c 6774, 6777
 - \iow_new:N 6774, 6776
 - \iow_newline: 155, 5298, 5812, 6119, 6585, 6655, 6664, 6677, 6748, 6748, 6926, 6928
 - \iow_now:Nn 153, 6626, 6626, 6629, 6631, 6633, 6771, 6773
 - \iow_now:Nx 153, 6625, 6625, 6627, 6628, 6630, 6635
 - \iow_now_buffer_safe:Nn 6770, 6770
 - \iow_now_buffer_safe:Nx 6770, 6772
 - \iow_now_when_avail:Nn . 154, 6632, 6632
 - \iow_now_when_avail:Nx . 154, 6632, 6634
 - \iow_open:cn 153, 6424
 - \iow_open:Nn 153, 6424, 6437, 6451
 - \iow_open_streams: 6778, 6779
 - \iow_raw_new:c 157, 6410, 6485
 - \iow_raw_new:N 157, 6410, 6415, 6419, 6423, 6477
 - \iow_shipout:Nn ... 154, 6622, 6622, 6624
 - \iow_shipout:Nx 154, 6622
 - \iow_shipout_x:Nn 154, 6620, 6620, 6621, 6623, 6625
 - \iow_shipout_x:Nx 154, 6620
 - \iow_show_aux:nn 6568, 6600, 6605
 - \iow_stream_alloc:N 6445, 6468, 6468
 - \iow_stream_alloc_aux: 6468, 6474, 6491, 6499, 6501
 - \iow_term:n 154, 6628, 6631
 - \iow_term:x 154, 1169, 1171, 5281, 5285, 5795, 5799, 6102, 6106, 6572, 6576, 6594, 6598, 6628, 6630, 6924, 6946, 6948, 6949
 - \iow_wrap:xnnnN 155, 6647, 6647, 6771, 6773, 6883, 6886, 6891, 6894, 6940, 6947
 - \iow_wrap_end: 6647, 6681, 6725
 - \iow_wrap_loop:w 6647, 6667, 6674, 6694, 6723
 - \iow_wrap_newline: 6647, 6678, 6716
 - \iow_wrap_word: 6647, 6682, 6685
 - \iow_wrap_word_fits: ... 6647, 6692, 6696
 - \iow_wrap_word_newline: 6647, 6693, 6709
- J**
- \jobname 654
- K**
- \K 2721
 - \kern 532
 - \kernel_register_show:c 1438, 1447, 3822
 - \kernel_register_show:N 1438, 1438, 3821, 4041, 4122, 4169
 - \keys_bool_set:NN 7562, 7562, 7729, 7731
 - \keys_bool_set_inverse:NN 7577, 7577, 7733, 7735
 - \keys_choice_code_store:x 7644, 7644, 7745, 7747
 - \keys_choice_find:n 7595, 7685, 7926, 7926
 - \keys_choice_make: 7565, 7580, 7592, 7592, 7604, 7623, 7737
 - \keys_choices_generate:n 7618, 7618, 7769
 - \keys_choices_generate_aux:n 7618, 7625, 7632
 - \keys_choices_make:nn .. 7602, 7602, 7739
 - \keys_cmd_set:nn 7570, 7585, 7594, 7596, 7655, 7655, 7676, 7688, 7690, 7741
 - \keys_cmd_set:nx 7566, 7568, 7581, 7583, 7608, 7634, 7655, 7660, 7681, 7702, 7721, 7743
 - \keys_cmd_set_aux:n 7655, 7657, 7662, 7665
 - \keys_default_set:n 7575, 7590, 7671, 7671, 7673, 7749
 - \keys_default_set:V 7671, 7751
 - \keys_define:nn ... 168, 7500, 7500, 7965
 - \keys_define_aux:nnn ... 7500, 7502, 7508
 - \keys_define_aux:onnn 7500, 7501
 - \keys_define_elt:n 7505, 7509, 7509
 - \keys_define_elt:nn 7505, 7509, 7514
 - \keys_define_elt_aux:nn 7509, 7512, 7517, 7519
 - \keys_define_key:n 7522, 7545, 7545
 - \keys_define_key_aux:w . 7545, 7549, 7560
 - \keys_execute: 7872, 7897, 7897
 - \keys_execute:nn 7897, 7898, 7901, 7917, 7928, 7929

- \keys_execute_unknown:
 ... [7830](#), [7832](#), [7897](#), [7898](#), [7899](#), [7907](#)
 - \keys_execute_unknown_alt:
 [7830](#), [7897](#), [7908](#)
 - \keys_execute_unknown_std:
 [7832](#), [7897](#), [7907](#)
 - \keys_if_choice_exist:nnn [7937](#)
 - \keys_if_choice_exist:nnnTF [7937](#)
 - \keys_if_choice_exist:nnTF [177](#)
 - \keys_if_choice_exist_p:nn [177](#)
 - \keys_if_choice_exist_p:nnn [7937](#)
 - \keys_if_exist:nn [7931](#)
 - \keys_if_exist:nnTF [176](#), [7931](#)
 - \keys_if_exist_p:nn [176](#), [7931](#)
 - \keys_if_value:n [7890](#)
 - \keys_if_value_p:n [7855](#), [7865](#), [7890](#)
 - \keys_meta_make:n [7674](#), [7674](#), [7779](#)
 - \keys_meta_make:x [7674](#), [7679](#), [7781](#)
 - \keys_multichoice_find:n [7684](#), [7684](#), [7689](#)
 - \keys_multichoice_make:
 [7684](#), [7686](#), [7698](#), [7783](#)
 - \keys_multichoices_make:nn
 [7684](#), [7696](#), [7785](#)
 - \keys_property_find:n .. [7520](#), [7528](#), [7528](#)
 - \keys_property_find_aux:w
 [7528](#), [7532](#), [7535](#), [7541](#)
 - \keys_set:nn
 ... [175](#), [7677](#), [7682](#), [7814](#), [7814](#), [7822](#)
 - \keys_set:no [175](#), [7814](#)
 - \keys_set:nV [175](#), [7814](#)
 - \keys_set:nv [175](#), [7814](#)
 - \keys_set_aux:nnn [7814](#), [7816](#), [7823](#)
 - \keys_set_aux:onn [7814](#), [7815](#)
 - \keys_set_elt:n .. [7819](#), [7831](#), [7838](#), [7838](#)
 - \keys_set_elt:nn . [7819](#), [7831](#), [7838](#), [7843](#)
 - \keys_set_elt_aux:nn [7838](#), [7841](#), [7846](#), [7848](#)
 - \keys_set_known:nnN [176](#), [7824](#), [7824](#), [7836](#)
 - \keys_set_known:noN [176](#), [7824](#)
 - \keys_set_known:nVN [176](#), [7824](#)
 - \keys_set_known:nvN [176](#), [7824](#)
 - \keys_set_known_aux:nnnN [7824](#), [7826](#), [7837](#)
 - \keys_set_known_aux:onnN [7824](#), [7825](#)
 - \keys_show:nn [177](#), [7943](#), [7943](#)
 - \keys_value_or_default:n [7852](#), [7875](#), [7875](#)
 - \keys_value_requirement:n
 [7712](#), [7712](#), [7811](#), [7813](#)
 - \keys_variable_set:cnN [7718](#),
 [7755](#), [7763](#), [7773](#), [7789](#), [7797](#), [7801](#)
 - \keys_variable_set:cnNN [7718](#),
 [7759](#), [7767](#), [7777](#), [7793](#), [7805](#), [7809](#)
 - \keys_variable_set:NnN
 [7718](#), [7724](#), [7727](#),
 [7753](#), [7761](#), [7771](#), [7787](#), [7795](#), [7799](#)
 - \keys_variable_set:NnNN
 [7718](#), [7718](#), [7725](#), [7726](#),
 [7757](#), [7765](#), [7775](#), [7791](#), [7803](#), [7807](#)
 - \keyval_parse:n [7368](#), [7376](#), [7477](#)
 - \keyval_parse:NnN [178](#), [7470](#),
 [7470](#), [7505](#), [7819](#), [7831](#), [8008](#)–[8010](#)
 - \keyval_parse_elt:w
 [7384](#), [7390](#), [7390](#), [7393](#), [7398](#)
 - \keyval_remove_spaces:w
 [7422](#), [7428](#), [7435](#), [7465](#)
 - \keyval_remove_spaces_aux:w
 [7422](#), [7436](#), [7437](#)
 - \keyval_split_key:w [7404](#), [7422](#), [7424](#)
 - \keyval_split_key_aux:w [7422](#), [7432](#), [7434](#)
 - \keyval_split_key_value:w [7397](#), [7402](#), [7402](#)
 - \keyval_split_key_value_aux:wTF
 [7402](#), [7415](#), [7420](#)
 - \keyval_split_value:w .. [7416](#), [7439](#), [7439](#)
 - \keyval_split_value_aux:w ... [7457](#), [7462](#)
 - \KV_process_no_space_removal_no_sanitiz:NNn
 [8008](#), [8010](#)
 - \KV_process_space_removal_no_sanitiz:NNn
 [8008](#), [8009](#)
 - \KV_process_space_removal_sanitiz:NNn
 [8008](#), [8008](#)
- L**
- \L [2721](#)
 - \l_cctab_tmp_tl [11542](#), [11555](#)–[11558](#), [11572](#)
 - \l_clist_remove_clist
 [5624](#), [5624](#), [5631](#), [5634](#), [5635](#), [5637](#),
 [5649](#), [5653](#), [5659](#), [5662](#), [5663](#), [5665](#)
 - \l_clist_show_tl [5804](#), [5807](#)
 - \l_clist_tmpa_tl [5522](#), [5522](#)
 - \l_clist_tmpb_tl [5522](#), [5523](#)
 - \l_exp_tl [35](#), [1524](#), [1524](#), [1543](#), [1544](#)
 - \l_expl_status_bool
 [96](#), [294](#), [309](#), [323](#), [327](#), [328](#)
 - \l_expl_status_stack_tl [197](#)
 - \l_file_name_tl [180](#), [8038](#),
 [8038](#), [8081](#), [8082](#), [8088](#), [8089](#), [8099](#)
 - \l_file_search_path_saved_seq
 [180](#), [8040](#), [8041](#), [8059](#), [8075](#)
 - \l_file_search_path_seq
 [180](#), [8039](#), [8039](#), [8059](#),
 [8061](#), [8064](#), [8075](#), [8105](#), [8106](#), [8109](#)

- \l_fp_arg_t1 [8153](#),
8153, 9842, 9861, 9865, 9875, 9896,
9897, 9939, 9954, 9962, 9982, 9983,
10140, 10159, 10163, 10173, 10183,
10207, 10238, 10263, 10264, 10334,
10349, 10358, 10360, 10388, 10428,
10560, 10677, 10688, 10696, 10716
- \l_fp_count_int [8154](#),
8154, 9374, 9409, 9421, 9429, 10037,
10069, 10086, 10088, 10091, 10093,
10537, 10574, 10582, 10812, 10815,
10816, 10838, 10882, 10942, 10953
- \l_fp_div_offset_int [8155](#),
8155, 9330, 9384, 9429, 9431, 10252
- \l_fp_exp_decimal_int ... [8156](#), 8157,
10412, 10446, 10465, 10485, 10504,
10513, 10518, 10524, 10540, 10589,
10594, 10598, 10601, 10605, 10609,
10612, 10614, 10758, 10768, 10779,
10782, 10791, 10799, 10825, 10888,
10896, 10900, 10911, 10954, 10955
- \l_fp_exp_exponent_int
..... [8156](#), 8159, 10414,
10448, 10470, 10490, 10506, 10756,
10760, 10778, 10785, 10808, 10892
- \l_fp_exp_extended_int [8156](#),
8158, 10413, 10447, 10465, 10485,
10505, 10514, 10517, 10522, 10528,
10540, 10590, 10592, 10595, 10606,
10608, 10610, 10759, 10768, 10801,
10805, 10807, 10825, 10890, 10897,
10899, 10902, 10911, 10954, 10956
- \l_fp_exp_integer_int
..... [8156](#), 8156, 10411, 10445,
10465, 10485, 10503, 10512, 10516,
10540, 10600, 10613, 10757, 10768,
10790, 10795, 10798, 10825, 10887
- \l_fp_input_a_decimal_int
..... [8160](#), 8162, 8211,
8402, 8403, 8408, 8410, 8441, 8443,
8457, 8483, 8485, 8499, 8897, 8911,
8933, 8947, 8959, 8961, 8968, 8972,
9010, 9020, 9035, 9046, 9116, 9132,
9231, 9313, 9378, 9380, 9382, 9398,
9411, 9412, 9414, 9437, 9608, 9610,
9619, 9627, 9628, 9635, 9638, 9646,
9654, 9735, 9749, 9750, 9754, 9757,
9759, 9765, 9773, 9775, 9788, 9789,
9796, 9798, 9806, 9809, 9815, 9817,
9821, 9840, 9853, 9937, 9950, 10024,
10030, 10031, 10061, 10064, 10067,
10078, 10082, 10138, 10151, 10205,
10229, 10234, 10236, 10331, 10345,
10510, 10513, 10520, 10526, 10535,
10577, 10649, 10654, 10684, 10831,
10845, 10849, 10852, 10867, 10872,
10876, 10879, 10880, 10944, 10946,
10949, 10952, 10982, 10988, 10996,
11013, 11039, 11072, 11122, 11133,
11153, 11166, 11199, 11215, 11233,
11258, 11328, 11360, 11380, 11389
- \l_fp_input_a_exponent_int
..... [8160](#), 8163, 8212, 8357,
8365, 8390, 8411, 8442, 8459, 8484,
8501, 8913, 8929, 8949, 8973, 9022,
9048, 9081, 9191, 9334, 9606, 9630,
9634, 9663, 9710, 9841, 9855, 9857,
9938, 9952, 10139, 10153, 10155,
10180, 10230, 10235, 10256, 10332,
10347, 10370, 10650, 10686, 10735,
10736, 10741, 10743, 10754, 10764,
10765, 10865, 10874, 10983, 10989,
11034, 11068, 11123, 11134, 11161,
11168, 11200, 11216, 11235, 11310,
11314, 11342, 11346, 11382, 11391
- \l_fp_input_a_extended_int
..... [8168](#), 8168, 9620, 9622,
9629, 9656, 9660, 9662, 9711, 9751–
9753, 9755, 9765, 9777, 9779, 9790,
9797, 9799, 9807, 9810, 9818, 9822,
10030, 10031, 10065, 10068, 10078,
10082, 10114, 10333, 10514, 10530,
10536, 10577, 10607, 10813, 10846,
10853, 10873, 10877, 10879, 10880,
10944, 10946, 10949, 10952, 11033,
11039, 11070, 11151, 11154, 11167
- \l_fp_input_a_integer_int
..... [8160](#), 8161, 8210, 8391, 8394, 8401,
8440, 8454, 8482, 8496, 8908, 8944,
8967, 8969, 8971, 9017, 9043, 9112,
9128, 9241, 9246, 9250, 9255, 9313,
9377, 9382, 9392, 9395, 9410, 9413,
9435, 9436, 9607, 9617, 9618, 9645,
9650, 9653, 9714, 9716, 9729, 9734,
9747, 9748, 9758, 9761, 9767, 9769,
9771, 9796, 9798, 9804, 9806, 9809,
9817, 9821, 9839, 9849, 9936, 9946,
10137, 10147, 10206, 10228, 10233,
10237, 10330, 10341, 10373, 10383,
10398, 10410, 10420, 10422, 10424,

- 10449, 10453, 10455, 10457, 10477,
 10479, 10482, 10648, 10654, 10680,
 10831, 10837, 10848, 10851, 10864,
 10871, 10981, 10987, 10996, 11013,
 11073, 11121, 11132, 11153, 11165,
 11198, 11214, 11232, 11258, 11318,
 11323, 11350, 11355, 11378, 11387
 \l_fp_input_a_sign_int
 8160, 8160, 8206, 8208, 8439, 8449,
 8481, 8491, 8903, 8939, 9038, 9075,
 9109, 9153, 9205, 9348, 9715, 9720,
 9762, 9838, 9844, 9892, 9899, 9935,
 9941, 9978, 9985, 10011, 10013,
 10018, 10136, 10142, 10191, 10232,
 10329, 10336, 10372, 10426, 10459,
 10478, 10511, 10534, 10575, 10647,
 10651, 10980, 10986, 11065, 11119,
 11164, 11197, 11213, 11231, 11278,
 11292, 11296, 11300, 11376, 11385
 \l_fp_input_b_decimal_int 8160, 8166,
 8377, 8378, 8383, 8385, 9065, 9116,
 9132, 9168, 9186, 9233, 9299, 9303,
 9398, 9411, 10220, 10225, 10535,
 10578, 10579, 10581, 10583, 10586,
 10589, 10605, 10867, 10881, 10945,
 10982, 10992, 11125, 11133, 11143,
 11155, 11205, 11215, 11233, 11261,
 11276, 11328, 11360, 11381, 11388
 \l_fp_input_b_exponent_int
 8160, 8167, 8357, 8365,
 8386, 8390, 9066, 9169, 9187, 9191,
 9300, 9334, 10221, 10226, 10256,
 10868, 10983, 11126, 11134, 11147,
 11161, 11206, 11216, 11235, 11310,
 11314, 11342, 11346, 11383, 11390
 \l_fp_input_b_extended_int
 8168, 8169, 10536,
 10578, 10579, 10581, 10583, 10586,
 10591, 10881, 10945, 11145, 11156
 \l_fp_input_b_integer_int 8160,
 8165, 8366, 8369, 8376, 9064, 9112,
 9128, 9167, 9185, 9244, 9248, 9251,
 9255, 9298, 9303, 9392, 9395, 9410,
 10219, 10224, 10866, 10981, 10992,
 11124, 11132, 11141, 11155, 11204,
 11214, 11232, 11261, 11276, 11318,
 11323, 11350, 11355, 11379, 11386
 \l_fp_input_b_sign_int
 8160, 8164, 9063, 9075, 9139, 9166,
 9170, 9184, 9205, 9297, 9348, 10223,
 10534, 10575, 10588, 10980, 11029,
 11139, 11164, 11203, 11213, 11231,
 11266, 11292, 11296, 11377, 11384
 \l_fp_mul_a_i_int
 8170, 8170, 9232, 9237,
 9242, 9247, 9251, 9481, 9490, 9497,
 9503, 9508, 9514, 9517, 9525, 9534,
 9542, 9550, 9557, 9565, 9570, 9574
 \l_fp_mul_a_ii_int 8170,
 8171, 9232, 9238, 9243, 9248, 9481,
 9491, 9498, 9504, 9509, 9515, 9525,
 9535, 9543, 9551, 9558, 9566, 9571
 \l_fp_mul_a_iii_int
 8170, 8172, 9232, 9239,
 9244, 9481, 9492, 9499, 9505, 9510,
 9525, 9536, 9544, 9552, 9559, 9567
 \l_fp_mul_a_iv_int
 8170, 8173, 9483, 9493, 9500,
 9506, 9527, 9537, 9545, 9553, 9560
 \l_fp_mul_a_v_int .. 8170, 8174, 9483,
 9494, 9501, 9527, 9538, 9546, 9554
 \l_fp_mul_a_vi_int 8170,
 8175, 9483, 9495, 9527, 9539, 9547
 \l_fp_mul_b_i_int
 8170, 8176, 9234, 9239,
 9243, 9247, 9250, 9485, 9495, 9501,
 9506, 9510, 9515, 9517, 9529, 9539,
 9546, 9553, 9559, 9566, 9570, 9573
 \l_fp_mul_b_ii_int 8170,
 8177, 9234, 9238, 9242, 9246, 9485,
 9494, 9500, 9505, 9509, 9514, 9529,
 9538, 9545, 9552, 9558, 9565, 9569
 \l_fp_mul_b_iii_int
 8170, 8178, 9234, 9237,
 9241, 9485, 9493, 9499, 9504, 9508,
 9529, 9537, 9544, 9551, 9557, 9564
 \l_fp_mul_b_iv_int
 8170, 8179, 9487, 9492, 9498,
 9503, 9531, 9536, 9543, 9550, 9556
 \l_fp_mul_b_v_int .. 8170, 8180, 9487,
 9491, 9497, 9531, 9535, 9542, 9549
 \l_fp_mul_b_vi_int 8170,
 8181, 9487, 9490, 9531, 9534, 9541
 \l_fp_mul_output_int 8182,
 8182, 9235, 9240, 9273, 9274, 9278,
 9280, 9285, 9488, 9496, 9532, 9540
 \l_fp_mul_output_tl
 8182, 8183, 9236, 9253, 9254,
 9257, 9284, 9489, 9512, 9513, 9520,
 9533, 9562, 9563, 9576, 9577, 9580

```

\l_fp_output_decimal_int ..... 8184, 8186, 9085, 9101,
9114, 9118, 9121, 9130, 9134, 9136,
9140, 9143, 9145, 9196, 9209, 9222,
9253, 9328, 9339, 9352, 9365, 9426,
9428, 9679, 9684, 9692, 9722, 9887,
9888, 9894, 9908, 9973, 9974, 9980,
9994, 10026, 10041, 10045, 10050,
10054, 10062, 10064, 10096, 10101,
10105, 10108, 10112, 10116, 10119,
10121, 10219, 10228, 10250, 10261,
10275, 10402, 10446, 10466, 10468,
10486, 10488, 10541, 10543, 10548,
10549, 10551, 10558, 10567, 10704,
10705, 10707, 10714, 10727, 10746,
10758, 10769, 10771, 10823, 10826,
10872, 10875, 10876, 10889, 10909,
10912, 10918, 10921, 10928, 10936,
10955, 10959, 10963, 10966, 11111,
11125, 11144, 11157, 11166, 11170
\l_fp_output_exponent_int .....
..... 8184, 8187, 9081,
9086, 9103, 9189, 9197, 9224, 9332,
9340, 9368, 9706, 9723, 9885, 9889,
9895, 9910, 9971, 9975, 9981, 9996,
10254, 10262, 10277, 10404, 10448,
10470, 10490, 10559, 10569, 10715,
10730, 10748, 10760, 10778, 10785,
10874, 10893, 10917, 10938, 11113,
11126, 11148, 11159, 11168, 11172
\l_fp_output_extended_int 8188, 8188,
9697, 9698, 9703, 9705, 9888, 9974,
10042, 10046, 10051, 10053, 10065,
10097, 10099, 10102, 10113, 10115,
10117, 10220, 10229, 10403, 10447,
10467, 10469, 10487, 10489, 10542,
10544, 10546, 10702, 10747, 10759,
10770, 10772, 10824, 10827, 10877,
10891, 10910, 10913, 10956, 10957,
10960, 11146, 11158, 11167, 11171
\l_fp_output_integer_int .....
..... 8184, 8185, 9084,
9097, 9110, 9120, 9126, 9135, 9138,
9141, 9147, 9149, 9195, 9209, 9218,
9257, 9327, 9338, 9352, 9361, 9421,
9667, 9668, 9671, 9678, 9721, 9884,
9887, 9893, 9904, 9970, 9973, 9979,
9990, 10025, 10040, 10044, 10049,
10060, 10107, 10120, 10249, 10260,
10271, 10401, 10427, 10445, 10466,
10468, 10486, 10488, 10541, 10543, 10548,
10549, 10551, 10558, 10567, 10704,
10705, 10707, 10714, 10727, 10746,
10758, 10769, 10771, 10823, 10826,
10872, 10875, 10876, 10889, 10909,
10912, 10918, 10921, 10928, 10936,
10955, 10959, 10963, 10966, 11111,
11125, 11144, 11157, 11166, 11170
\l_fp_output_sign_int ... 8184, 8184,
9083, 9092, 9109, 9139, 9153, 9194,
9337, 10193, 10195, 10199, 10201,
10259, 10266, 10556, 10712, 10718,
10737, 10739, 10818, 10904, 11140
\l_fp_round_carry_bool ..... 8189,
8189, 8956, 8966, 8979, 8985, 8993
\l_fp_round_decimal_tl ..... 8190,
8190, 8958, 8968, 8987, 8988, 8990
\l_fp_round_position_int ..... 8191,
8191, 8957, 8978, 8991, 8997, 8998
\l_fp_round_target_int .. 8191, 8192,
8893, 8894, 8928, 8930, 8978, 8991
\l_fp_sign_tl .....
..... 8193, 8193, 11030, 11042, 11106
\l_fp_split_sign_int .....
..... 8194, 8194, 8219, 8221, 8234
\l_fp_tmp_dim .... 8466, 8474, 8478, 8514
\l_fp_tmp_int .....
... 8195, 8195, 8263, 8265, 8980–
8983, 8988, 9584–9586, 9591–9593
\l_fp_tmp_skip ... 8466, 8473, 8474, 8515
\l_fp_tmp_tl ..... 8196,
8196, 8216–8218, 8222, 8227, 8229,
8232, 8238, 8240, 8243, 8332, 8337,
8379, 8385, 8404, 8410, 9036, 9051,
9648, 9654, 9657, 9662, 9680, 9687,
9699, 9705, 10417, 10424, 10431,
10437, 10450, 10457, 10460, 10462,
10793, 10799, 10802, 10807, 10930,
10936, 11374, 11393, 11399, 11404
\l_fp_trig_decimal_int .....
..... 8198, 8199, 10032, 10034,
10036, 10039, 10054, 10067, 10077,
10079, 10081, 10083, 10085, 10087,
10090, 10092, 10094, 10096, 10112
\l_fp_trig_extended_int . 8198, 8200,
10032, 10034, 10036, 10038, 10053,
10068, 10077, 10079, 10081, 10083,
10085, 10087, 10090, 10092, 10098
\l_fp_trig_octant_int ..... 8197,
8197, 9786, 9792, 9811, 9823, 9915,
10001, 10012, 10016, 10192, 10198

```

\l_fp_trig_sign_int
 8198, 8198, 10028, 10066, 10075, 10095
 \l_ior_stream_int
 6408, 6409, 6427, 6429,
 6433, 6464, 6507, 6508, 6512, 6515,
 6522, 6524, 6530, 6531, 6533, 6535
 \l_iow_current_line_int . 6639, 6639,
 6651, 6689, 6690, 6706, 6713, 6720
 \l_iow_current_line_tl
 6641, 6641, 6652, 6701,
 6704, 6712, 6714, 6719, 6721, 6728
 \l_iow_current_word_int
 6639, 6640, 6687, 6689, 6713
 \l_iow_current_word_tl
 6641, 6642, 6676,
 6677, 6680, 6688, 6701, 6705, 6714
 \l_iow_line_length_int
 155, 6636, 6636, 6637, 6650
 \l_iow_line_start_bool
 .. 6646, 6646, 6654, 6698, 6700, 6722
 \l_iow_stream_int 6408,
 6408, 6409, 6440, 6442, 6446, 6456,
 6470, 6471, 6475, 6478, 6480, 6485,
 6487, 6493, 6494, 6496, 6498, 6517
 \l_iow_target_length_int
 6638, 6638, 6650, 6691
 \l_iow_wrap_tl
 .. 6643, 6643, 6653, 6659, 6662, 6668
 \l_iow_wrapped_tl
 .. 6644, 6644, 6672, 6711, 6718, 6727
 \l_keys_choice_int .. 174, 7491, 7491,
 7605, 7611, 7612, 7615, 7624, 7637,
 7638, 7642, 7699, 7705, 7706, 7709
 \l_keys_choice_tl . 174, 7610, 7636, 7704
 \l_keys_choices_tl 7491, 7492
 \l_keys_key_tl 176, 7493,
 7493, 7573, 7588, 7850, 7851, 7912
 \l_keys_module_tl
 ... 7494, 7494, 7501, 7504, 7506,
 7530, 7677, 7682, 7815, 7818, 7820,
 7825, 7828, 7833, 7851, 7901, 7904
 \l_keys_no_value_bool
 ... 7495, 7495, 7511, 7516, 7547,
 7840, 7845, 7856, 7866, 7878, 7913
 \l_keys_path_tl
 176, 7496, 7496, 7525, 7530,
 7537, 7540, 7555, 7566, 7568, 7570,
 7581, 7583, 7585, 7594, 7596, 7599,
 7608, 7621, 7629, 7634, 7640, 7647,
 7650, 7652, 7672, 7676, 7681, 7688,
 7690, 7693, 7702, 7715, 7721, 7741,
 7743, 7851, 7860, 7870, 7880, 7882,
 7885, 7893, 7898, 7904, 7928, 7929
 \l_keys_property_tl . 7497, 7497, 7521,
 7525, 7543, 7550, 7551, 7554, 7558
 \l_keys_unknown_clist
 7498, 7498, 7829, 7834, 7910
 \l_keys_value_tl 176, 7499,
 7499, 7870, 7877, 7884, 7914, 7922
 \l_keyval_key_tl 7364,
 7364, 7411, 7426, 7431, 7432, 7445
 \l_keyval_parse_tl .. 7366, 7367, 7383,
 7387, 7407, 7441, 7450, 7454, 7466
 \l_keyval_sanitise_tl
 7366, 7366, 7379-7382, 7385
 \l_keyval_value_tl 7364,
 7365, 7447, 7449, 7452, 7464, 7467
 \l_last_box 146, 6281, 6281, 6283
 \l_msg_class_tl 7052, 7053, 7103, 7104, 7107
 \l_msg_current_class_tl
 7052, 7054, 7085, 7104
 \l_msg_current_module_tl 7052, 7055, 7086
 \l_msg_redirect_classes_prop 6959, 6959
 \l_msg_redirect_classes_seq
 7052, 7052, 7060, 7065, 7068
 \l_msg_redirect_kernel_info_prop . 7204
 \l_msg_redirect_kernel_warning_prop
 7182
 \l_msg_redirect_names_prop
 6959, 6960, 7087, 7118
 \l_msg_text_tl 6898, 6931, 6933
 \l_msg_tmp_tl 6787, 6787, 6902, 6905, 6913
 \l_peek_search_tl
 2872, 2872, 2890, 2911, 2954
 \l_peek_search_token
 .. 2871, 2871, 2889, 2910, 2929, 2937
 \l_peek_token 62, 2869, 2869, 2878, 2929,
 2937, 2947-2949, 2968, 3097-3099
 \l_prop_show_tl 6097, 6097,
 6111, 6114, 6577, 6580, 6599, 6602
 \l_seq_remove_seq
 .. 5036, 5036, 5043, 5046, 5047, 5049
 \l_seq_show_tl ... 5276, 5276, 5290, 5293
 \l_seq_tmpa_tl 4994, 4994, 5068, 5073,
 5087, 5091, 5488, 5494, 5499, 5500
 \l_seq_tmpb_tl
 .. 4994, 4995, 5064, 5068, 5090, 5091
 \l_tl_replace_tl 4363, 4363
 \l_tl_rescan_tl 4305, 4305,
 4317, 4320, 4326, 4344, 4346, 4359

\l_tl_tmpa_tl	4481 , 4484 , 4486 , 4494	\luaescapestring	39, 40
\l_tl_tmpb_tl	4481 , 4485 , 4486 , 4495	\luatex_catcodetable:D	
\l_tmpa_bool	41 , 1910 , 1910	\luatex_directlua:D	755 , 770 , 11544 , 11545 , 11550 , 11558
\l_tmpa_box	6294 , 6295 , 6298	\luatex_if_engine:F	756 , 1471 , 11487
\l_tmpa_dim	85 , 4052 , 4052	1449 , 1474 , 11530 , 11560 , 11580	
\l_tmpa_int	76 , 3842 , 3842	\luatex_if_engine:T	.	1448 , 1473 , 4774 ,
\l_tmpa_skip	88 , 4126 , 4126			11533 , 11566 , 11582 , 11588 , 11597
\l_tmpa_tl	5 , 108 , 146 , 4786 , 4786	\luatex_if_engine:TF	
\l_tmpb_box	6294 , 6300	24 , 1448 , 1450 , 1475 , 11485	
\l_tmpb_dim	85 , 4052 , 4053	\luatex_initcatcodetable:D	
\l_tmpb_int	76 , 3842 , 3843	757 , 771 , 11521 , 11538	
\l_tmpb_skip	88 , 4126 , 4127	\luatex_latelua:D	758 , 772 , 11488
\l_tmpb_tl	108 , 146 , 4786 , 4787	\luatex luatexversion:D	759
\l_tmpc_dim	85 , 4052 , 4054	\luatex_savecatcodetable:D	
\l_tmpc_int	76 , 3842 , 3844	760 , 773 , 11549 , 11577	
\l_tmpc_skip	88 , 4126 , 4128	\luatexcacodetable	770
\language	449	\luatexinitcatcodetable	771
\lastbox	606	\luatexlatelua	772
\lastkern	539	\luatexsavecatcodetable	773
\lastlinefit	719	\luatexversion	759
\lastnodetype	700			
\lastpenalty	645	M		
\lastskip	540	\M	2678 , 2721
\latelua	758	\m@ne	1157
\lccode	668	\mag	448
\leaders	536	\mark	450
\left	504	\marks	676
\lefthyphenmin	560	\mathaccent	461
\leftskip	562	\mathbin	491
\leqno	479	\mathchar	462
\let	59 , 230 , 336 , 337 , 349	\mathchardef	359
\limits	496	\mathchoice	459
\linepenalty	552	\mathclose	492
\lineskip	546	\mathcode	670
\lineskiplimit	547	\mathinner	493
\long	33 , 339 , 368	\mathop	494
\looseness	564	\mathopen	498
\lower	601	\mathord	499
\lowercase	640	\mathparagraph	3866
\lua_now:n	190 , 11485 , 11502	\mathpunct	500
\lua_now:x	190 , 4776 ,	\mathrel	501
		11485 , 11487 , 11491 , 11494 , 11503	\mathsection	3865
\lua_shipout:n	. .	191 , 11485 , 11505 , 11507	\mathsurround	512
\lua_shipout:x	191 , 11485	\maxdeadcycles	582
\lua_shipout_x:n	191 , 11485 ,	\maxdepth	583
		11488 , 11496 , 11499 , 11504 , 11506	\maxdimen	4050
\lua_shipout_x:x	191 , 11485	\meaning	642
\lua_wrong_engine:		\medmuskip	513
		11531 , 11562 , 11563 , 11581	\message	419
\luaescapestring	39 , 40	\MessageBreak	222 , 238–244

.meta:n	171	\msg_generic_new:nn	7349, 7350
.meta:x	171	\msg_generic_new:nnn	7349, 7349
\middle	724	\msg_generic_set:nn	7349, 7352
\mkern	466	\msg_generic_set:nnn	7349, 7351
\mode_if_horizontal:	2243	\msg_gset:nnn	6790, 6817
\mode_if_horizontal:TF	46, 2243	\msg_gset:nnnn	6790, 6797, 6810, 6818
\mode_if_horizontal_p:	46, 2243	\msg_if_more_text:cTF	6975, 7010, 7149
\mode_if_inner:	2245	\msg_if_more_text:N	6975
\mode_if_inner:TF	46, 2245	\msg_if_more_text:NF	6984
\mode_if_inner_p:	46, 2245	\msg_if_more_text:NT	6983
\mode_if_math:	2247	\msg_if_more_text:NTF	6975, 6985
\mode_if_math:TF	46, 2247, 3854	\msg_if_more_text_p:c	6975
\mode_if_math_p:	2247	\msg_if_more_text_p:N	6975, 6982
\mode_if_vertical:	2241	\msg_info:nn	161, 7038
\mode_if_vertical:TF	46, 2241	\msg_info:nnx	161, 7038
\mode_if_vertical_p:	46, 2241	\msg_info:nnxx	161, 7038
\month	652	\msg_info:nnxxx	161, 7038
\moveleft	602	\msg_info:nnxxxx	161, 7038
\moveright	603	\msg_info_text:n	160, 6952, 6956, 7042, 7211
\msg_class_new:nn	7343, 7343	\msg_interrupt:xxx	
\msg_class_set:nn		163, 6866, 6866, 6988, 6999,	
160, 6961, 6961, 6986, 6997,		7012, 7021, 7129, 7151, 7164, 7304	
7008, 7030, 7038, 7046, 7051, 7343		\msg_interrupt_aux:	6866, 6872, 6922
\msg_critical:nn	161, 6997	\msg_interrupt_details:xxx	
\msg_critical:nnx	161, 6997	6866, 6871, 6889	
\msg_critical:nnxx	161, 6997	\msg_interrupt_more_text:n	
\msg_critical:nnxxx	161, 6997	6866, 6885, 6893, 6899	
\msg_critical:nnxxxx	161, 6997	\msg_interrupt_no_details:xx	
\msg_critical_text:n	159, 6952, 6953, 7000	6866, 6870, 6881	
\msg_direct_interrupt:xxxxx	7349, 7353	\msg_interrupt_text:n	
\msg_direct_log:xx	7349, 7354	6866, 6887, 6895, 6897	
\msg_direct_term:xx	7349, 7355	\msg_kernel_bug:x	7302, 7302
\msg_error:nn	161, 7008	\msg_kernel_error:nn	
\msg_error:nnx	161, 7008	165, 1173, 1187, 6430,	
\msg_error:nnxx	161, 7008	6443, 6794, 7066, 7147, 7180, 7417,	
\msg_error:nnxxx	161, 7008	11455, 11463, 11468, 11477, 11548	
\msg_error:nnxxxx	161, 7008	\msg_kernel_error:nnx	165, 1173, 1185,
\msg_error_text:n		1443, 4380, 5192, 7075, 7147, 7178,	
159, 6952, 6954, 7013, 7022, 7152, 7165		7533, 7572, 7587, 7628, 7859, 11526	
\msg_expandable_error:n		\msg_kernel_error:nnxx	
166, 1575, 4572,		165, 1173, 1173, 1186, 1188, 1196,	
4991, 5461, 7319, 7327, 11493, 11498		1206, 1219, 1353, 7081, 7147, 7176,	
\msg_expandable_error_aux:w	7333, 7340	7524, 7553, 7598, 7692, 7869, 7903	
\msg_fatal:nn	161, 6986	\msg_kernel_error:nnxxx	165, 7147, 7174
\msg_fatal:nnx	161, 6986	\msg_kernel_error:nnxxxx	
\msg_fatal:nnxx	161, 6986	165, 7147, 7147, 7175, 7177, 7179, 7181	
\msg_fatal:nnxxx	161, 6986	\msg_kernel_fatal:nn	165, 7127, 7145
\msg_fatal:nnxxxx	161, 6986	\msg_kernel_fatal:nnx	
\msg_fatal_text:n		165, 7127, 7143, 11523	
159, 6952, 6952, 6989, 7130		\msg_kernel_fatal:nnxx	165, 7127, 7141

\msg_kernel_fatal:nnxxx 165, 7127, 7139
 \msg_kernel_fatal:nnxxxx 165, 7127, 7127, 7140, 7142, 7144, 7146
 \msg_kernel_info:nn 165, 7182, 7224
 \msg_kernel_info:nnx 165, 7182, 7222
 \msg_kernel_info:nnxx 165, 7182, 7220
 \msg_kernel_info:nnxxx 165, 7182, 7218
 \msg_kernel_info:nnxxxx 165, 7182, 7205, 7219, 7221, 7223, 7225
 \msg_kernel_new:nnn 164, 7119, 7121
 \msg_kernel_new:nnnn 164, 6606, 6613, 7119, 7119, 7226, 7235, 7243, 7250, 7257, 7265, 7274, 7281, 7288, 7295, 7480, 7945, 7948, 7954, 7961, 7970, 7976, 7982, 7989, 7996, 8002, 11448, 11456, 11464, 11470
 \msg_kernel_set:nnn 164, 7119, 7125
 \msg_kernel_set:nnnn 164, 7119, 7123
 \msg_kernel_warning:nn 165, 7182, 7202
 \msg_kernel_warning:nnx 165, 7182, 7200
 \msg_kernel_warning:nnxx 165, 7182, 7198
 \msg_kernel_warning:nnxxx 165, 7182, 7196
 \msg_kernel_warning:nnxxxx 165, 7182, 7183, 7197, 7199, 7201, 7203
 \msg_line_context: 158, 1189, 1189, 1209, 6853, 6854
 \msg_line_number: 159, 6853, 6853, 6858, 7481
 \msg_log:nn 162, 7046, 7348
 \msg_log:nnx 162, 7046, 7347
 \msg_log:nnxx 162, 7046, 7346
 \msg_log:nnxxx 162, 7046, 7345
 \msg_log:nnxxxx 162, 7046, 7344
 \msg_log:x 164, 6937, 6937, 7040, 7048, 7209
 \msg_new:nnn 158, 6790, 6799, 7122
 \msg_new:nnnn 158, 6790, 6790, 6800, 7120
 \msg_newline: 163, 6851, 6851, 6910
 \msg_no_more_text:xxxx 6975, 6977, 6981
 \msg_none:nn 162, 7051
 \msg_none:nnx 162, 7051
 \msg_none:nnxx 162, 7051
 \msg_none:nnxxx 162, 7051
 \msg_none:nnxxxx 162, 7051
 \msg_redirect_class:nn 162, 7113, 7113
 \msg_redirect_module:nnn 162, 7115, 7115
 \msg_redirect_name:nnn 163, 7117, 7117
 \msg_see_documentation_text:n 6957, 6957, 6992, 7003, 7016, 7025, 7134, 7156, 7169, 7307
 \msg_set:nnn 158, 6790, 6808, 7126
 \msg_set:nnnn 158, 6790, 6801, 6809, 7124
 \msg_term:x 164, 6937, 6944, 7032, 7187
 \msg_trace:nn 7344, 7348
 \msg_trace:nnx 7344, 7347
 \msg_trace:nnxx 7344, 7346
 \msg_trace:nnxxx 7344, 7345
 \msg_trace:nnxxxx 7344, 7344
 \msg_two_newlines: 163, 6851, 6852
 \msg_use:nnnnxxxx 6965, 7056, 7056, 7185, 7207
 \msg_use_aux:nn 7056, 7089, 7091
 \msg_use_aux:nnn 7056, 7080, 7083
 \msg_use_code: 7056, 7058, 7098, 7106, 7110
 \msg_use_loop:n 7056, 7063, 7111, 7112
 \msg_use_loop:o 7056, 7107
 \msg_use_loop_check:nn 7056, 7088, 7094, 7097, 7101
 \msg_warning:nn 161, 7030
 \msg_warning:nnx 161, 7030
 \msg_warning:nnxx 161, 7030
 \msg_warning:nnxxx 161, 7030
 \msg_warning:nnxxxx 161, 7030
 \msg_warning_text:n 160, 6952, 6955, 7034, 7189
 \mskip 463
 \muexpr 712
 .multichoice: 171
 .multichoice:nn 171
 \multiply 364
 \muskip 660
 \muskip_add:cn 89, 4155
 \muskip_add:Nn 89, 4155, 4155, 4157, 4158
 \muskip_eval:n 90, 4165, 4165
 \muskip_gadd:cn 89, 4155
 \muskip_gadd:Nn 89, 4155, 4157, 4159
 \muskip_gset:cn 89, 4144
 \muskip_gset:Nn 89, 4144, 4146, 4148
 \muskip_gset_eq:cc 90, 4149
 \muskip_gset_eq:cN 90, 4149
 \muskip_gset_eq:Nc 90, 4149
 \muskip_gset_eq:NN 90, 4149, 4152–4154
 \muskip_gsub:cn 90, 4155
 \muskip_gsub:Nn 90, 4155, 4162, 4164
 \muskip_gzero:c 89, 4139
 \muskip_gzero:N 89, 4139, 4141, 4143
 \muskip_new:c 88, 4131
 \muskip_new:N 88, 4131, 4132, 4138
 \muskip_set:cn 89, 4144
 \muskip_set:Nn 89, 4144, 4144, 4146, 4147
 \muskip_set_eq:cc 89, 4149

\muskip_set_eq:cN	89, 4149	\or:	25, 78, 784, 786, 884, 1322,
\muskip_set_eq:Nc	89, 4149		1324, 1326, 1328, 1330, 1332, 1334,
\muskip_set_eq:NN	89, 4149, 4149–4151		1336, 1338, 3600–3624, 9916, 9918,
\muskip_show:c	91, 4169		9920, 9922, 10002, 10004, 10006, 10008
\muskip_show:N	91, 4169, 4169, 4170	\outer	369
\muskip_sub:cn	90, 4155	\output	584
\muskip_sub:Nn	90, 4155, 4160, 4162, 4163	\outputpenalty	594
\muskip_use:c	90, 4167	\over	471
\muskip_use:N	90, 4166, 4167, 4167, 4168	\overfullrule	622
\muskip_zero:c	89, 4139	\overline	502
\muskip_zero:N	89, 4139, 4139, 4141, 4142	\overwithdelims	472
\muskipdef	358		
\mutogluue	718	P	
N		\P	1815, 2721
\n	2717	\package_check_loaded_expl:	
\name_primitive:NN	339, 339, 346–760		780, 1522, 1868, 2381, 2488, 3194,
\negative_replication	2165		3893, 4189, 4987, 5520, 5896, 6191,
\newbox	6197		6373, 6785, 7361, 8016, 8129, 11483
\newcatcodetable	11537	\PackageError	219, 235
\newcount	3274	\pagedepth	586
\newdimen	3902	\pagediscards	727
\newlinechar	249, 414	\pagefilllstretch	590
\newmuskip	4135	\pagefillstretch	589
\newread	6420	\pagefilstretch	588
\newskip	4061	\pagegoal	592
\newwrite	6419	\pageshrink	591
\noalign	382	\pagestretch	587
\noboundary	517	\pagetotal	593
\noexpand	35, 39, 40, 166, 169, 172, 174, 175, 184, 187–189, 191, 193, 194, 203, 205–209, 268, 270, 275, 277, 375	\par	542
\noindent	543	\parfillskip	573
\nolimits	497	\parindent	566
\nonscript	477	\parshape	558
\nonstopmode	440	\parshapedimen	708
int_get_digits:n	77	\parshapeindent	706
int_get_sign:n	77	\parshapelength	707
int_to_letter:n	77	\parskip	565
\nulldelimiterspace	510	\patterns	648
\nullfont	628	\pausing	435
\number	637	\pdf@strcmp	59
\numexpr	709	\pdfcolorstack	738
O		\pdfcompresslevel	739
\O	1817, 2721	\pdfcreationdate	737
\omit	383	\pdfdecimaldigits	740
\openin	409	\pdfhorigin	741
\openout	410	\pdfinfo	742
\or	406	\pdfliteral	743
		\pdfminorversion	744
		\pdfobjcompresslevel	745
		\pdfoutput	746
		\pdfpkresolution	750

- \pdfrestore 747
- \pdfsave 748
- \pdfsetmatrix 749
- \pdfstrcmp . 33, 59, 230, 235, 238, 252, 753
- \pdftex_if_engine:F 1452, 1463, 1477
- \pdftex_if_engine:T 1451, 1462, 1476
- \pdftex_if_engine:TF
 - 24, 1448, 1453, 1464, 1478
- \pdftex_pdfcolorstack:D 738
- \pdftex_pdfcompresslevel:D 739
- \pdftex_pdfcreationdate:D 737
- \pdftex_pdfdecimaldigits:D 740
- \pdftex_pdfhorigin:D 741
- \pdftex_pdfinfo:D 742
- \pdftex_pdfliteral:D 743
- \pdftex_pdfminorversion:D 744
- \pdftex_pdfobjcompresslevel:D 745
- \pdftex_pdfoutput:D 746
- \pdftex_pdfpkresolution:D 750
- \pdftex_pdfrestore:D 747
- \pdftex_pdfsave:D 748
- \pdftex_pdfsetmatrix:D 749
- \pdftex_pdftextrevision:D 751
- \pdftex_pdfvorigin:D 752
- \pdftex_strcmp:D
 - 753, 1485, 1491, 2461, 2470,
 - 2693, 4093, 4752, 4791, 8226, 8237
- \pdftextrevision 751
- \pdfvorigin 752
- \peek_after:NN 3130, 3130
- \peek_after:Nw
 - 61, 2877, 2877, 2902, 2920, 2976, 3130
- \peek_catcode:NTF 62, 2994
- \peek_catcode_ignore_spaces:NTF 62, 2994
- \peek_catcode_remove:NTF 62, 2994
- \peek_catcode_remove_ignore_spaces:NTF
 - 62, 2994
- \peek_charcode:NTF 62, 3010
- \peek_charcode_ignore_spaces:NTF ...
 - 63, 3010
- \peek_charcode_remove:NTF 63, 3010
- \peek_charcode_remove_ignore_spaces:NTF
 - 63, 3010
- \peek_def:nnnn 2977, 2978,
 - 2994, 2998, 3002, 3006, 3010, 3014,
 - 3018, 3022, 3026, 3030, 3034, 3038
- \peek_def_aux:nnnnn 2977, 2980–2982, 2984
- \peek_execute_branches: 2973, 2989
- \peek_execute_branches_catcode:
 - .. 2926, 2926, 2997, 2999, 3005, 3007
- \peek_execute_branches_charcode: ...
 - .. 2943, 2943, 3013, 3015, 3021, 3023
- \peek_execute_branches_charcode:NN 2943
- \peek_execute_branches_charcode_aux:NN
 - 2953, 2957
- \peek_execute_branches_meaning:
 - .. 2926, 2935, 3029, 3031, 3037, 3039
- \peek_execute_branches_N_type:
 - 3093, 3093, 3105, 3107, 3109
- \peek_false:w 2873, 2875, 2896,
 - 2914, 2932, 2940, 2951, 2962, 3101
- \peek_gafter:NN 3130, 3131
- \peek_gafter:Nw 61, 2879, 3131
- \peek_ignore_spaces_execute_branches:
 - 2966, 2966, 2976,
 - 3001, 3009, 3017, 3025, 3033, 3041
- \peek_ignore_spaces_execute_branches_aux:
 - 2966, 2970, 2975
- \peek_meaning:NTF 63, 3026
- \peek_meaning_ignore_spaces:NTF 63, 3026
- \peek_meaning_remove:NTF 64, 3026
- \peek_meaning_remove_ignore_spaces:NTF
 - 64, 3026
- \peek_N_type:F 3108
- \peek_N_type:T 3106
- \peek_N_type:TF 66, 3093, 3104
- \peek_tmp:w 2873, 2876, 2885, 2971
- \peek_token_generic:NNF 2906, 3109
- \peek_token_generic:NNT 2904, 3107
- \peek_token_generic:NNTF
 - 2887, 2887, 2905, 2907, 3105
- \peek_token_remove_generic:NNF .. 2924
- \peek_token_remove_generic:NNT .. 2922
- \peek_token_remove_generic:NNTF
 - 2908, 2908, 2923, 2925
- \peek_true:w 2873, 2873,
 - 2891, 2912, 2930, 2938, 2960, 3102
- \peek_true_aux:w . 2873, 2874, 2884, 2913
- \peek_true_remove:w 2881, 2881, 2912
- \penalty 643
- \postdisplaypenalty 490
- \predisplaydirection 734
- \predisdisplaypenalty 489
- \predisplaysize 488
- \pref_global:D 26,
 - 817, 817, 1287, 1292, 2880, 3289,
 - 3300, 3306, 3314, 3316, 3326, 3328,
 - 3335, 3907, 3912, 3918, 3935, 3940,
 - 4066, 4071, 4077, 4082, 4087, 4141,
 - 4146, 4152, 4157, 4162, 4953, 6224,

- 6230, 6285, 6305, 6311, 6317, 6339,
6345, 6351, 6357, 6765, 6769, 11520
- \pref_long:D 26, 817, 818,
827, 829, 835, 837, 841, 843, 849, 851
- \pref_protected:D 26, 817, 819, 826, 828,
830–833, 835, 837, 845, 847, 849, 851
- \pretolerance 569
- \prevdepth 616
- \prevgraf 575
- \prg_case_dim:nnn 44, 2100, 2100
- \prg_case_dim_aux:nnn .. 2100, 2101, 2102
- \prg_case_dim_aux:nw 2100, 2103, 2104, 2108
- \prg_case_end:nw 2089,
2089, 2097, 2107, 2115, 2124, 2132
- \prg_case_int:nnn 43, 2090, 2090, 3497, 3501
- \prg_case_int_aux:nnn .. 2090, 2091, 2092
- \prg_case_int_aux:nw 2090, 2093, 2094, 2098
- \prg_case_str:nnn 44, 2110, 2110, 2118, 4936
- \prg_case_str:onn 44, 2110
- \prg_case_str:xxn 44, 2110, 2119
- \prg_case_str_aux:nw 2110, 2111, 2112, 2116
- \prg_case_str_x_aux:nw
..... 2110, 2120, 2121, 2125
- \prg_case_tl:cnn 45, 2127
- \prg_case_tl:Nnn ... 45, 2127, 2127, 2135
- \prg_case_tl_aux:Nw 2127, 2128, 2129, 2133
- \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 2296, 2296, 2371
- \prg_do_nothing: 8, 1482,
1482, 4403, 4408, 4566, 4754, 5470,
5473, 5476, 5490, 5497, 8232, 8243
- \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, 1870, 2439, 2447, 2459, 2468, 6750
- \prg_new_conditional:Npnn
..... 37, 899, 907, 1421,
1483, 1489, 1870, 1898, 1912, 2241,
2243, 2245, 2247, 2604, 2609, 2614,
2619, 2626, 2632, 2637, 2642, 2647,
2652, 2657, 2662, 2667, 2672, 2686,
2700, 2705, 2726, 2733, 2740, 2751,
2762, 2773, 2784, 2793, 2800, 2818,
3340, 3410, 3418, 3426, 3947, 3952,
4090, 4100, 4424, 4446, 4467, 4469,
4665, 4681, 4697, 4731, 4733, 4749,
4802, 4804, 6016, 6028, 6263, 6265,
6275, 7890, 7931, 7937, 11176, 11184
- \prg_new_eq_conditional:NN 39
- \prg_new_eq_conditional:NNn
..... 965, 967, 1870,
5079, 5081, 5686–5691, 6182–6185
- \prg_new_map_functions:Nn ... 2374, 2374
- \prg_new_protected_conditional:Nnn ..
..... 38, 930, 956, 1870, 8079
- \prg_new_protected_conditional:Npnn
..... 38, 899, 922,
1870, 4481, 4502, 5083, 5310, 5318,
5331, 5338, 5345, 5352, 5692, 5705,
6055, 6126, 6132, 11192, 11209, 11396
- \prg_quicksort:n 47, 2371
- \prg_quicksort_compare:nnTF
..... 47, 2372, 2373
- \prg_quicksort_function:n 47, 2372, 2372
- \prg_replicate:nn
..... 45, 2136, 2136, 8550, 8586, 8656
- \prg_replicate_ 2136, 2147
- \prg_replicate_0:n 2136
- \prg_replicate_1:n 2136
- \prg_replicate_2:n 2136
- \prg_replicate_3:n 2136
- \prg_replicate_4:n 2136

<code>\prg_replicate_5:n</code>	2136	2792 , 2799 , 2808 , 2838 , 2864 , 3357 ,
<code>\prg_replicate_6:n</code>	2136	3365 , 3375 , 3381 , 3389 , 3399 , 3407 ,
<code>\prg_replicate_7:n</code>	2136	3413 , 3421 , 3431 , 3950 , 3956 , 4095 ,
<code>\prg_replicate_8:n</code>	2136	4107 , 4437 , 4449 , 4462 , 4472 , 4488 ,
<code>\prg_replicate_9:n</code>	2136	4506 , 4672 , 4692 , 4707 , 4725 , 4746 ,
<code>\prg_replicate_aux:N</code> 2136 , 2143 , 2144 , 2146		4756 , 4793 , 5099 , 5314 , 5322 , 5335 ,
<code>\prg_replicate_first_~:n</code>	2136	5342 , 5349 , 5356 , 5700 , 5713 , 6019 ,
<code>\prg_replicate_first_0:n</code>	2136	6042 , 6064 , 6142 , 6264 , 6266 , 6276 ,
<code>\prg_replicate_first_1:n</code>	2136	6755 , 6760 , 6979 , 7894 , 7934 , 7940 ,
<code>\prg_replicate_first_2:n</code>	2136	8084 , 11179 , 11187 , 11237 , 11271 ,
<code>\prg_replicate_first_3:n</code>	2136	11280 , 11294 , 11312 , 11320 , 11330 ,
<code>\prg_replicate_first_4:n</code>	2136	11344 , 11352 , 11362 , 11410 , 11416 ,
<code>\prg_replicate_first_5:n</code>	2136	11422 , 11428 , 11434 , 11440 , 11446
<code>\prg_replicate_first_6:n</code>	2136	<code>\prg_set_conditional:Nnn</code>
<code>\prg_replicate_first_7:n</code>	2136 37 , 930 , 930 , 1870
<code>\prg_replicate_first_8:n</code>	2136	<code>\prg_set_conditional:Npnn</code>
<code>\prg_replicate_first_9:n</code>	2136 37 , 899 , 899 , 1100 ,
<code>\prg_replicate_first_aux:N</code>		1112 , 1128 , 1140 , 1870 , 4434 , 6975
..... 2136 , 2139 , 2145		<code>\prg_set_eq_conditional:NN</code>
<code>\prg_return_false:</code>	39 ,	39
895 , 897 , 1103 , 1108 , 1121 , 1126 ,		<code>\prg_set_eq_conditional:NNn</code>
1134 , 1151 , 1424 , 1487 , 1492 , 1870 ,	 965 , 965 , 1870
1903 , 1917 , 2242 , 2244 , 2246 , 2250 ,		<code>\prg_set_eq_conditional_aux:NNNn</code> ...
2444 , 2452 , 2465 , 2474 , 2607 , 2612 ,	 966 , 968 , 1045 , 1045
2617 , 2622 , 2629 , 2635 , 2640 , 2645 ,		<code>\prg_set_eq_conditional_aux:NNNw</code> ...
2650 , 2655 , 2660 , 2665 , 2670 , 2675 ,	 1045 , 1046 , 1047 , 1055
2696 , 2703 , 2710 , 2712 , 2732 , 2739 ,		<code>\prg_set_map_functions:Nn</code> ... 2374 , 2375
2743 , 2750 , 2754 , 2761 , 2772 , 2776 ,		<code>\prg_set_protected_conditional:Nnn</code> .
2783 , 2792 , 2799 , 2808 , 2821 , 2840 ,	 38 , 930 , 949 , 1870
2857 , 2866 , 3359 , 3367 , 3373 , 3383 ,		<code>\prg_set_protected_conditional:Npnn</code>
3391 , 3397 , 3405 , 3415 , 3423 , 3429 ,	 38 , 899 , 915 , 1870
3950 , 3958 , 4097 , 4108 , 4439 , 4451 ,		<code>\prg_stepwise_function:nnnN</code>
4464 , 4474 , 4491 , 4506 , 4674 , 4694 ,	 45 , 2176 , 2176
4709 , 4717 , 4727 , 4744 , 4758 , 4795 ,		<code>\prg_stepwise_function_decr:nnnN</code> ...
5096 , 5306 , 5698 , 5711 , 6021 , 6037 ,	 2176 , 2180 , 2192 , 2197
6059 , 6130 , 6136 , 6264 , 6266 , 6276 ,		<code>\prg_stepwise_function_incr:nnnN</code> ...
6757 , 6978 , 7895 , 7935 , 7941 , 8083 ,	 2176 , 2179 , 2183 , 2188
11181 , 11189 , 11226 , 11240 , 11244 ,		<code>\prg_stepwise_inline:nnnn</code>
11248 , 11252 , 11264 , 11268 , 11283 ,		.. 46 , 2202 , 2202 , 2235 , 11612 , 11617
11298 , 11316 , 11325 , 11333 , 11348 ,		<code>\prg_stepwise_inline_decr:Nnnn</code>
11357 , 11365 , 11410 , 11416 , 11422 ,	 2202 , 2210 , 2224 , 2229
11428 , 11434 , 11440 , 11445 , 11446		<code>\prg_stepwise_inline_incr:Nnnn</code>
<code>\prg_return_true:</code> 39 , 895 , 895 , 1106 ,	 2202 , 2209 , 2215 , 2220
1123 , 1131 , 1136 , 1149 , 1154 , 1424 ,		<code>\prg_stepwise_variable:nnnn</code>
1487 , 1492 , 1870 , 1901 , 1915 , 2242 ,		46
2244 , 2246 , 2250 , 2442 , 2450 , 2463 ,		<code>\prg_stepwise_variable:nnnnN</code> 2233 , 2233
2472 , 2607 , 2612 , 2617 , 2622 , 2629 ,		<code>\prg_variable_get_scope:N</code> 47 , 2264 , 2270
2635 , 2640 , 2645 , 2650 , 2655 , 2660 ,		<code>\prg_variable_get_scope_aux:w</code>
2665 , 2670 , 2675 , 2694 , 2703 , 2710 ,	 2264 , 2272 , 2275
2732 , 2739 , 2750 , 2761 , 2772 , 2783 ,		<code>\prg_variable_get_type:N</code> . 47 , 2264 , 2284
		<code>\prg_variable_get_type:w</code>
		2264

\prg_variable_get_type_aux:w	
.....	2286, 2289, 2293	
\prop_clear:c	134, 5902, 5903
\prop_clear:N	134, 5902, 5902, 5907
\prop_clear_new:c	134, 5906, 6963
\prop_clear_new:N	134, 5906, 5906, 5908
\prop_del:cn	137, 5938
\prop_del:cV	137, 5938
\prop_del:Nn	..	137, 5938, 5938, 5944, 5945
\prop_del:NV	137, 5938
\prop_del_aux:Nnnn	5938, 5939, 5941, 5942	
\prop_display:c	6169, 6170
\prop_display:N	6169, 6169
\prop_gclear:c	134, 5902, 5905, 5911
\prop_gclear:N	134, 5902, 5904, 5910
\prop_gclear_new:c	134, 5906, 5911
\prop_gclear_new:N	134, 5906, 5909
\prop_gdel:cn	137, 5938
\prop_gdel:cV	137, 5938
\prop_gdel:Nn	..	137, 5938, 5940, 5946, 5947
\prop_gdel:NV	137, 5938, 6549, 6561
\prop_get:cn	140
\prop_get:cnN	136, 5948, 7103
\prop_get:cnNTF	138, 6055
\prop_get:coN	136
\prop_get:cVN	136, 5948
\prop_get:Nn	140, 6164, 6164, 6168
\prop_get:NnN
...	136, 5948, 5948, 5956, 5957, 6055	
\prop_get:NnNF	6067
\prop_get:NnNT	6066
\prop_get:NnNTF	138, 6055, 6068
\prop_get:NoN	136, 5948
\prop_get:NVN	136, 5948
\prop_get_aux:nnn	6164, 6165, 6166
\prop_get_aux:Nnnn	5948, 5951, 5954
\prop_get_aux_true:Nnnn	6055, 6058, 6061	
\prop_get_gdel:NnN	6177, 6177
\prop_gget:cnN	6171
\prop_gget:cVN	6171
\prop_gget:NnN	...	6171, 6171, 6175, 6176
\prop_gget:NVN	6171
\prop_gget_aux:Nnnn	6171, 6172, 6173
\prop_gpop:cnN	137, 5958
\prop_gpop:cnNTF	140, 6126
\prop_gpop:coN	137, 5958
\prop_gpop:NnN
...	137, 5958, 5964, 5977, 5978, 6132, 6177	
\prop_gpop:NnNF	6148
\prop_gpop:NnNT	6147
\prop_gpop:NnNTF	140, 6149
\prop_gpop:NoN	137, 5958
\prop_gput:ccx	6181
\prop_gput:cn	135, 5979
\prop_gput:cno	135, 5979
\prop_gput:cnV	135, 5979
\prop_gput:cnx	135, 5979
\prop_gput:con	135, 5979
\prop_gput:coo	135, 5979
\prop_gput:cVn	135, 5979
\prop_gput:cVV	135, 5979
\prop_gput:Nnn
...	135, 5979, 5980, 5997, 5999, 6181	
\prop_gput:Nno	135, 5979
\prop_gput:NnV	135, 5979
\prop_gput:Nnx	135, 5979
\prop_gput:Non	135, 5979
\prop_gput:Noo	135, 5979
\prop_gput:NVn	135, 5979, 6433, 6446
\prop_gput:NVV	135, 5979
\prop_gput_if_new:cn	136, 6001
\prop_gput_if_new:Nnn	136, 6001, 6003, 6015	
\prop_gset_eq:cc	134, 5912, 5919
\prop_gset_eq:cN	134, 5912, 5918
\prop_gset_eq:Nc	134, 5912, 5917
\prop_gset_eq:NN	134, 5912, 5916
\prop_if_empty:cTF	137, 6016
\prop_if_empty:N	6016
\prop_if_empty:NF	6027
\prop_if_empty:NT	6026
\prop_if_empty:NTF
...	137, 6016, 6025, 6100, 6570, 6592	
\prop_if_empty_p:c	137, 6016
\prop_if_empty_p:N	137, 6016, 6024
\prop_if_eq:cc	6185
\prop_if_eq:ccTF	6182
\prop_if_eq:cN	6183
\prop_if_eq:cNTF	6182
\prop_if_eq:Nc	6184
\prop_if_eq:NcTF	6182
\prop_if_eq:NN	6182
\prop_if_eq:NNTF	6182
\prop_if_eq_p:cc	6182
\prop_if_eq_p:cN	6182
\prop_if_eq_p:Nc	6182
\prop_if_eq_p:NN	6182
\prop_if_in:ccTF	6178
\prop_if_in:cnTF	..	138, 6028, 7093, 7096
\prop_if_in:coTF	138, 6028
\prop_if_in:cVTF	138, 6028

- \prop_if_in:Nn 6028
- \prop_if_in:NnF 6051, 6052, 6179, 6454, 6462
- \prop_if_in:NnT 6049, 6050, 6178
- \prop_if_in:NnTF
 - ... 138, 6028, 6053, 6054, 6180, 7087
- \prop_if_in:NoTF 138, 6028
- \prop_if_in:NVT 6498, 6535
- \prop_if_in:NVTF 138, 6028
- \prop_if_in_aux:nwn 6030, 6034, 6045
- \prop_if_in_aux:w 6028
- \prop_if_in_p:cn 138, 6028
- \prop_if_in_p:co 138, 6028
- \prop_if_in_p:cV 138, 6028
- \prop_if_in_p:Nn .. 138, 6028, 6047, 6048
- \prop_if_in_p:No 138, 6028
- \prop_if_in_p:NV 138, 6028
- \prop_map_break: 139, 6077, 6095, 6095, 6158
- \prop_map_break:n . 139, 6096, 6096, 6167
- \prop_map_function:cc 6069
- \prop_map_function:cN 139, 6069
- \prop_map_function:Nc 6069, 6090
- \prop_map_function:NN 139, 6069,
 - 6069, 6082, 6083, 6112, 6578, 6600
- \prop_map_function_aux:Nwn
 - 6069, 6071, 6074, 6080
- \prop_map_inline:cn 139, 6085
- \prop_map_inline:Nn 139, 6085, 6085, 6094
- \prop_map_tokens:cn 140, 6150
- \prop_map_tokens:Nn
 - 140, 6150, 6150, 6163, 6165
- \prop_map_tokens_aux:nwn
 - 6150, 6152, 6155, 6161
- \prop_new:c 134, 5900, 5901
- \prop_new:N 134, 5900, 5900, 5907, 5910,
 - 6400, 6401, 6959, 6960, 7182, 7204
- \prop_pop:cnN 136, 5958
- \prop_pop:cnNTF 138, 6126
- \prop_pop:coN 136, 5958
- \prop_pop:NnN
 - 136, 5958, 5958, 5975, 5976, 6126
- \prop_pop:NnNF 6145
- \prop_pop:NnNT 6144
- \prop_pop:NnNTF 138, 6126, 6146
- \prop_pop:NoN 136, 5958
- \prop_pop_aux:NNNnnn 5958, 5961, 5967, 5970
- \prop_pop_aux_true:NNNnnn
 - 6126, 6129, 6135, 6138
- \prop_put:cnn 135, 5979, 7114, 7116
- \prop_put:cno 135, 5979
- \prop_put:cnV 135, 5979
- \prop_put:cnx 135, 5979
- \prop_put:con 135, 5979
- \prop_put:coo 135, 5979
- \prop_put:cVn 135, 5979
- \prop_put:cVV 135, 5979
- \prop_put:Nnn 135, 5979,
 - 5979, 5993, 5995, 6403–6406, 7118
- \prop_put:Nno 135, 5979
- \prop_put:NnV 135, 5979
- \prop_put:Nnx 135, 5979
- \prop_put:Non 135, 5979
- \prop_put:Noo 135, 5979
- \prop_put:NVn 135, 5979
- \prop_put:NVV 135, 5979
- \prop_put_aux:NNnn 5979–5981
- \prop_put_aux:NNnnnnn .. 5979, 5983, 5985
- \prop_put_if_new:cnn 136, 6001
- \prop_put_if_new:Nnn 136, 6001, 6001, 6014
- \prop_put_if_new_aux:NNnn 6002, 6004, 6005
- \prop_set_eq:cc 134, 5912, 5915
- \prop_set_eq:cN 134, 5912, 5914
- \prop_set_eq:Nc 134, 5912, 5913
- \prop_set_eq:NN 134, 5912, 5912
- \prop_show:c 140, 6098, 6170
- \prop_show:N .. 140, 6098, 6098, 6125, 6169
- \prop_show_aux:n 6098
- \prop_show_aux:nn 6112, 6117
- \prop_show_aux:w
 - 6098, 6114, 6124, 6580, 6602
- \prop_split:Nnn 141, 5932, 5932, 5983, 6172
- \prop_split:NnTF 141,
 - 5920, 5920, 5934, 5939, 5941, 5950,
 - 5960, 5966, 6007, 6057, 6128, 6134
- \prop_split_aux:nnnn .. 5920, 5926, 5930
- \prop_split_aux:NnTF .. 5920, 5921, 5922
- \prop_split_aux:w 5920, 5924, 5927, 5931
- \protect 235
- \protected 68,
 - 82, 98, 104, 126, 133, 141, 143, 147,
 - 152, 157, 213, 266, 273, 292, 325, 736
- \protected@edef 6662, 6905
- \ProvidesClass 154
- \ProvidesExplClass 6, 146, 152
- \ProvidesExplFile 6, 146, 157
- \ProvidesExplPackage 6, 146, 147, 333,
 - 778, 1520, 1866, 2379, 2486, 3192,
 - 3891, 4187, 4985, 5518, 5894, 6189,
 - 6371, 6783, 7359, 8014, 8127, 11481
- \ProvidesFile 159
- \ProvidesPackage 47, 149

Q

- `\Q` 7423, 7461
- `\q` 1081, 2002, 2007
- `\q_iow_stop` [6645](#), 6645, 6669, 6680
- `\q_mark` [49](#), [2384](#), 2385, 3350, 3352, 4387, 4395, 4399, 4410, 4602, 4605, 4606, 4612, 4616, 4618, 4620, 4646, 4647, 5925, 5927, 5928
- `\q_nil` [49](#), 876, 879, 2299, 2303, [2384](#), 2384, 2441, 2462, 3713, 3735, 4448, 4460, 4461, 4604, 4608, 4625, 4628, 4631, 4670, 4686, 4706, 5596, 5597, 7385, 7416, 7448, 7455, 7462, 11402, 11409, 11415, 11421, 11427, 11433, 11439
- `\q_no_value` [49](#), 2031, [2384](#), 2386, 2449, 2471, 5658, 5697, 5703, 5710, 5716, 5936, 5952, 5962, 5968, 7385, 7393, 7398, 7415, 7421, 7668
- `\q_prop` [141](#), [5898](#), 5898, 5899, 5925, 5926, 5928, 5990, 6011, 6032, 6034, 6072, 6074, 6153, 6155
- `\q_recursion_stop` [50](#), 878, 881, 977, 1046, 1775, 2093, 2103, 2111, 2120, 2128, [2388](#), 2389, 2411, 2412, 2424, 2425, 2435, 4518, 4522, 4537, 4547, 4552, 4589, 4590, 4593, 5732, 5740, 5843, 5845, 6072, 6153
- `\q_recursion_tail` [50](#), [2388](#), 2388, 2392, 2398, 2412, 2425, 2435, 4518, 4522, 4537, 4547, 4552, 4589, 5732, 5740, 5843, 5845
- `\q_stop` [48](#), 877, 880, 1081, 1083, 1091, 1093, 1311, 1315, 1828, 1830, 2031, 2034, 2273, 2275, 2287, 2289, 2293, 2299, 2303, 2365, [2384](#), 2387, 2412, 2425, 2435, 2689, 2691, 2729, 2731, 2736, 2738, 2746, 2749, 2757, 2760, 2768, 2771, 2779, 2782, 2788, 2791, 2796, 2798, 2804, 2807, 2824, 2827, 2830, 2852, 3045, 3052, 3061, 3070, 3341, 3342, 3350, 3354, 3362, 3370, 3378, 3386, 3394, 3402, 3781, 3818, 4395, 4410, 4610, 4631, 4641, 4642, 4644, 4646, 4647, 4654, 4662, 4664, 4670, 4686, 4706, 4979, 4981, 5109, 5112, 5122, 5125, 5313, 5334, 5341, 5422, 5424, 5429, 5587, 5588, 5596, 5597, 5651, 5655, 5658, 5695, 5703, 5708, 5716, 5925, 5928, 6032, 6736, 7338, 7340, 7397, 7402, 7404, 7415, 7420, 7424, 7432, 7434, 7457, 7462, 7532, 7535, 7541, 7550, 7560, 8202, 8203, 8222, 8227, 8232, 8238, 8243, 8249, 8255, 8257, 8258, 8261, 8265, 8269, 8305, 8310, 8527, 8529, 8544, 8546, 8547, 8553, 8560, 8562, 8565, 8567, 8568, 8570, 8572, 8574, 8576, 8578, 8580, 8582, 8583, 8592, 8594, 8604, 8606, 8617, 8624, 8626–8628, 8630, 8632, 8634, 8636, 8638, 8640, 8642, 8644, 8653, 8659, 8661, 8671, 8673, 8680, 8682–8684, 8690, 8695, 8700, 8705, 8710, 8715, 8720, 8725, 8735, 8740, 8741, 8752, 8754, 8773, 8793, 9263, 9268, 9380, 9433, 9610, 9615, 9622, 9625, 11402, 11406, 11409, 11412, 11415, 11418, 11421, 11424, 11427, 11430, 11433, 11436, 11439, 11442
- `\q_tl_act_mark` [109](#), [2481](#), 2481, [4810](#), 4813, 4830
- `\q_tl_act_stop` [109](#), [2481](#), 2482, [4810](#), 4813, 4817, 4826, 4828, 4834, 4838, 4841, 4845, 4848
- `\quark_if_nil:N` 2439
- `\quark_if_nil:n` 2459
- `\quark_if_nil:nF` 2480
- `\quark_if_nil:nT` 2306, 2310, 2479
- `\quark_if_nil:NTF` [49](#), [2439](#), 3716, 3738, 7452
- `\quark_if_nil:nTF` [49](#), 2314, 2323, 2332, 2341, [2459](#), 2478, 5599, 11408, 11414, 11420, 11426, 11432, 11438, 11444
- `\quark_if_nil:oF` 7395
- `\quark_if_nil:oTF` [49](#), [2459](#)
- `\quark_if_nil:VTF` [49](#), [2459](#)
- `\quark_if_nil_p:N` [49](#), [2439](#)
- `\quark_if_nil_p:n` [49](#), [2459](#), 2477
- `\quark_if_nil_p:o` [49](#), [2459](#)
- `\quark_if_nil_p:V` [49](#), [2459](#)
- `\quark_if_no_value:cF` 7880
- `\quark_if_no_value:cTF` [49](#), [2439](#)
- `\quark_if_no_value:N` 2447
- `\quark_if_no_value:n` 2468
- `\quark_if_no_value:N.TF` [2439](#)
- `\quark_if_no_value:NF` 2457
- `\quark_if_no_value:nF` 5654
- `\quark_if_no_value:NT` [2456](#), 11556
- `\quark_if_no_value:NTF` .. [49](#), 2036, 2458

- `\quark_if_no_value:nTF` 49, 2459
`\quark_if_no_value_p:c` 49, 2439
`\quark_if_no_value_p:N` 49, 2455
`\quark_if_no_value_p:n` 49, 2459
`\quark_if_no_value_p:N.` 2439
`\quark_if_recursion_tail_aux:w`
 2405, 2411, 2424, 2434
`\quark_if_recursion_tail_stop:N`
 50, 2390, 2390, 4558
`\quark_if_recursion_tail_stop:n`
 50, 2405, 2405, 2437, 4526, 5745, 5852
`\quark_if_recursion_tail_stop:o` 50, 2405
`\quark_if_recursion_tail_stop_do:Nn`
 50, 2390, 2396
`\quark_if_recursion_tail_stop_do:nn`
 50, 2405, 2418, 2438, 4592
`\quark_if_recursion_tail_stop_do:on`
 50, 2405
`\quark_new:N` 48, 2383,
 2383–2389, 2481, 2482, 5898, 6645
- ### R
- `\R` 1816, 2721
`\radical` 464
`\raise` 604
`\read` 411
`\readline` 686
`\relax` 3–6, 9, 13,
 62, 70–80, 84–93, 96, 101, 131, 133,
 141, 143, 229, 233, 249, 281–289, 446
`\relpenalty` 507
`\RequirePackage` 57, 58
`\reverse_if:N` 25, 784, 789, 3978–3980, 4661
`\right` 505
`\righthyphenmin` 561
`\rightskip` 563
`\romannumeral` 638
- ### S
- `\S` 2721
`\savecatcodetable` 760
`\savinghyphcodes` 725
`\savingvdiscards` 726
`\scan_align_safe_stop:` 47, 2249, 2256, 2256
`\scan_stop:` 8, 308, 322,
 811, 811, 1049, 1073, 1102, 1120,
 1130, 1148, 1168, 1561, 1813–1821,
 2261, 2265, 2279, 2625, 2702, 3054,
 3063, 3072, 3105, 3107, 3109, 4070,
 4081, 4086, 4111, 4116, 4119, 4145,
 4156, 4161, 4166, 4180, 4181, 4296–
 4299, 4662, 6303, 6333, 6334, 6434,
 6447, 8210–8212, 8252, 8263, 8271,
 8276, 8312, 8313, 8329, 8337, 8376,
 8385, 8401, 8410, 8473, 8968, 8980,
 8981, 9113, 9117, 9129, 9133, 9146,
 9150, 9192, 9253, 9257, 9264–9266,
 9274, 9285, 9335, 9437, 9512, 9520,
 9562, 9576, 9580, 9618, 9619, 9628,
 9629, 9646, 9654, 9662, 9678, 9692,
 9705, 10050, 10373, 10383, 10427,
 10503–10506, 10527, 10728, 10754,
 10791, 10799, 10807, 10889, 10891,
 10893, 10928, 10936, 11140, 11142,
 11144, 11146, 11148, 11162, 11558
`\scantokens` 684
`\scriptfont` 630
`\scriptscriptfont` 631
`\scriptscriptstyle` 476
`\scriptspace` 516
`\scriptstyle` 475
`\scrollmode` 441
`\seq_break:` 121, 5147, 5179, 5184, 5184,
 5186, 5193, 5307, 5315, 5322, 5335,
 5342, 5349, 5356, 5393, 5413, 5421
`\seq_break:n`
 122, 5096, 5099, 5184, 5185, 5187, 5401
`\seq_break_point:n` .. 122, 5097, 5110,
 5123, 5136, 5164, 5184, 5185, 5188,
 5188, 5200, 5235, 5246, 5316, 5323,
 5336, 5343, 5350, 5357, 5395, 5414
`\seq_clear:c` 110, 4998, 4999
`\seq_clear:N` .. 110, 4998, 4998, 5043, 7060
`\seq_clear_new:c` 110, 5002, 5003
`\seq_clear_new:N` 110, 5002, 5002
`\seq_concat:ccc` 110, 5014
`\seq_concat:NNN` ... 110, 5014, 5014, 5018
`\seq_display:c` 5513, 5514
`\seq_display:N` 5513, 5513
`\seq_gclear:c` 110, 4998, 5001
`\seq_gclear:N` 110, 4998, 5000
`\seq_gclear_new:c` 110, 5002, 5005
`\seq_gclear_new:N` 110, 5002, 5004
`\seq_gconcat:ccc` 111, 5014
`\seq_gconcat:NNN` .. 111, 5014, 5016, 5019
`\seq_get:cN` 117, 5270, 5271
`\seq_get:NN` 117, 5270, 5270
`\seq_get_left:cN` .. 112, 5106, 5271, 5512
`\seq_get_left:cNTF` 118, 5310

\seq_get_left:NN	112, 5106, 5106, 5114, 5270, 5310, 5511
\seq_get_left:NNF	5326
\seq_get_left:NNT	5325
\seq_get_left:NNTF	118, 5310, 5327
\seq_get_left_aux:NnwN	5106, 5109, 5112
\seq_get_left_aux:Nw	5313
\seq_get_right:cN	112, 5132
\seq_get_right:cNTF	118, 5310
\seq_get_right:NN	112, 5132, 5132, 5155, 5318
\seq_get_right:NNF	5329
\seq_get_right:NNT	5328
\seq_get_right:NNTF	118, 5310, 5330
\seq_get_right_aux:NN	5132, 5135, 5138, 5321
\seq_get_right_loop:nn	5132, 5141, 5150, 5153, 5170
\seq_gpop:cN	117, 5270, 5275
\seq_gpop:NN	117, 5270, 5274, 8100, 11555
\seq_gpop_left:cN	113, 5115, 5275
\seq_gpop_left:cNTF	119, 5331
\seq_gpop_left:NN	113, 5115, 5117, 5131, 5274, 5338
\seq_gpop_left:NNF	5363
\seq_gpop_left:NNT	5362
\seq_gpop_left:NNTF	119, 5331, 5364
\seq_gpop_right:cN	113, 5156
\seq_gpop_right:cNTF	119, 5331
\seq_gpop_right:NN	113, 5156, 5158, 5183, 5352
\seq_gpop_right:NNF	5369
\seq_gpop_right:NNT	5368
\seq_gpop_right:NNTF	119, 5331, 5370
\seq_gpush:cn	118, 5250, 5265
\seq_gpush:co	118, 5250, 5268
\seq_gpush:cV	118, 5250, 5266
\seq_gpush:cv	118, 5250, 5267
\seq_gpush:cx	118, 5250, 5269
\seq_gpush:Nn	118, 5250, 5260
\seq_gpush:No	118, 5250, 5263
\seq_gpush:Nv	118, 5250, 5261, 8097
\seq_gpush:Nx	118, 5250, 5264, 11544
\seq_gput_left:cn	111, 5028, 5265
\seq_gput_left:co	111, 5028, 5268
\seq_gput_left:cV	111, 5028, 5266
\seq_gput_left:cv	111, 5028, 5267
\seq_gput_left:cx	111, 5028, 5269
\seq_gput_left:Nn	111, 5028, 5028, 5032, 5033, 5260
\seq_gput_left:No	111, 5028, 5263
\seq_gput_left:Nv	111, 5028, 5261
\seq_gput_left:Nv	111, 5028, 5262
\seq_gput_left:Nx	111, 5028, 5264
\seq_gput_right:cn	112, 5028
\seq_gput_right:co	112, 5028
\seq_gput_right:cV	112, 5028
\seq_gput_right:cv	112, 5028
\seq_gput_right:cx	112, 5028
\seq_gput_right:Nn	112, 5028, 5030, 5034, 5035
\seq_gput_right:No	112, 5028
\seq_gput_right:Nv	112, 5028, 8035
\seq_gput_right:Nv	112, 5028
\seq_gput_right:Nx	112, 5028, 8092
\seq_gremove_all:cn	114, 5053
\seq_gremove_all:Nn	114, 5053, 5055, 5078
\seq_gremove_duplicates:c	114, 5037
\seq_gremove_duplicates:N	114, 5037, 5039, 5052
\seq_gset_eq:cc	110, 5006, 5013
\seq_gset_eq:cN	110, 5006, 5012
\seq_gset_eq:Nc	110, 5006, 5011
\seq_gset_eq:NN	110, 5006, 5010, 5040
\seq_gset_from_clist:cc	120, 5433
\seq_gset_from_clist:cN	120, 5433
\seq_gset_from_clist:cn	120, 5433
\seq_gset_from_clist:Nc	120, 5433
\seq_gset_from_clist:NN	120, 5433, 5443, 5457, 5458
\seq_gset_from_clist:Nn	120, 5433, 5448, 5459
\seq_gset_reverse:N	121, 5460, 5464
\seq_gset_split:Nnn	121, 5479, 5481
\seq_if_empty:c	5081
\seq_if_empty:cTF	115, 5079
\seq_if_empty:N	115, 5079
\seq_if_empty:NTF	115, 5079, 5279, 5861, 5872
\seq_if_empty_break_return_false:N	5303, 5303, 5312, 5320, 5333, 5340, 5347, 5354
\seq_if_empty_err_break:N	121, 5108, 5121, 5134, 5162, 5189, 5189
\seq_if_empty_p:c	115, 5079
\seq_if_empty_p:N	115, 5079
\seq_if_in:cnTF	115, 5083
\seq_if_in:coTF	115, 5083

<code>\seq_if_in:cvTF</code>	115, 5083	<code>\seq_pop:NN</code>	117, 5270, 5272
<code>\seq_if_in:cvTF</code>	115, 5083	<code>\seq_pop_item_def:</code>	
<code>\seq_if_in:cxTF</code>	115, 5083	121, 5075, 5146, 5178,
<code>\seq_if_in:Nn</code>	5083	5209, 5225, 5235, 5246, 5867, 5878
<code>\seq_if_in:NnF</code> ...	5046, 5102, 5103, 8105	<code>\seq_pop_left:cN</code>	113, 5115, 5273
<code>\seq_if_in:NnT</code>	5100, 5101	<code>\seq_pop_left:cNTF</code>	119, 5331
<code>\seq_if_in:NnTF</code> 115, 5083, 5104, 5105, 7065		<code>\seq_pop_left:NN</code>	
<code>\seq_if_in:NoTF</code>	115, 5083	... 113, 5115, 5115, 5130, 5272, 5331	
<code>\seq_if_in:NVTF</code>	115, 5083	<code>\seq_pop_left:NNF</code>	5360
<code>\seq_if_in:NvTF</code>	115, 5083	<code>\seq_pop_left:NNT</code>	5359
<code>\seq_if_in:NxTF</code>	115, 5083	<code>\seq_pop_left:NNTF</code>	119, 5331, 5361
<code>\seq_if_in_aux:</code>	5083, 5092, 5099	<code>\seq_pop_left_aux:NNN</code>	
<code>\seq_item:cn</code>	119, 5381	5115, 5116, 5118, 5119
<code>\seq_item:n</code> ...	121, 4989, 4989, 5021,	<code>\seq_pop_left_aux:NnwNNN</code>	
5023, 5029, 5031, 5071, 5088, 5112,		5115, 5122, 5125, 5334, 5341
5125, 5168, 5212, 5217, 5222, 5228,		<code>\seq_pop_right:cN</code>	113, 5156
5453, 5468, 5469, 5471, 5477, 5510		<code>\seq_pop_right:cNTF</code>	119, 5331
<code>\seq_item:Nn</code>	119, 5381, 5381, 5404	<code>\seq_pop_right:NN</code>	
<code>\seq_item_aux:nnn</code> 5381, 5383, 5397, 5402		113, 5156, 5156, 5182, 5345
<code>\seq_length:c</code>	119, 5371	<code>\seq_pop_right:NNF</code>	5366
<code>\seq_length:N</code> . 119, 5371, 5371, 5380, 5388		<code>\seq_pop_right:NNT</code>	5365
<code>\seq_length_aux:n</code>	5371, 5376, 5379	<code>\seq_pop_right:NNTF</code> ...	119, 5331, 5367
<code>\seq_map_break:</code> 116, 5186, 5186, 5199, 8070		<code>\seq_pop_right_aux:NNN</code>	
<code>\seq_map_break:n</code>	116, 5186, 5187	5156, 5157, 5159, 5160
<code>\seq_map_function:cN</code>	115, 5196	<code>\seq_pop_right_aux_ii:NNN</code>	
<code>\seq_map_function:NN</code>	5156, 5163, 5166, 5348, 5355
115, 5196, 5196, 5208, 5291, 5376, 5405		<code>\seq_push:cn</code>	5250, 5255
<code>\seq_map_function_aux:NNn</code>		<code>\seq_push:co</code>	117, 5250, 5258
.....	5196, 5198, 5202, 5206	<code>\seq_push:cV</code>	117, 5250, 5256
<code>\seq_map_inline:cn</code>	115, 5231	<code>\seq_push:cv</code>	117, 5257
<code>\seq_map_inline:Nn</code>		<code>\seq_push:cx</code>	117, 5250, 5259
115, 5044, 5231, 5231, 5237, 8064, 8114		<code>\seq_push:Nn</code>	117, 5250, 5250
<code>\seq_map_variable:ccn</code>	115, 5238	<code>\seq_push:No</code>	117, 5250, 5253
<code>\seq_map_variable:cNn</code>	115, 5238	<code>\seq_push:NV</code>	117, 5250, 5251
<code>\seq_map_variable:Ncn</code>	115, 5238	<code>\seq_push:Nv</code>	117, 5250, 5252
<code>\seq_map_variable:NNn</code>		<code>\seq_push:Nx</code>	117, 5250, 5254
.....	115, 5238, 5238, 5248, 5249	<code>\seq_push_item_def:n</code> 121, 5059, 5140,	
<code>\seq_mapthread_function:ccN</code> ..	120, 5407	5168, 5209, 5209, 5233, 5864, 5875	
<code>\seq_mapthread_function:cNN</code> ..	120, 5407	<code>\seq_push_item_def:x</code> 121, 5209, 5214, 5240	
<code>\seq_mapthread_function:NcN</code> ..	120, 5407	<code>\seq_push_item_def_aux:</code>	
<code>\seq_mapthread_function:NNN</code>	5209, 5211, 5216, 5219
.....	120, 5407, 5407, 5431, 5432	<code>\seq_put_left:cn</code>	111, 5020, 5255
<code>\seq_mapthread_function_aux:NN</code>		<code>\seq_put_left:co</code>	111, 5020, 5258
.....	5407, 5409, 5416	<code>\seq_put_left:cV</code>	111, 5020, 5256
<code>\seq_mapthread_function_aux:Nnnwnn</code> .		<code>\seq_put_left:cv</code>	111, 5020, 5257
.....	5407, 5418, 5424, 5429	<code>\seq_put_left:cx</code>	111, 5020, 5259
<code>\seq_new:c</code>	4, 109, 4996, 4997	<code>\seq_put_left:Nn</code>	
<code>\seq_new:N</code> ..	4, 109, 4996, 4996, 5036,	... 111, 5020, 5020, 5024, 5025, 5250	
7052, 8029, 8030, 8039, 8041, 11511		<code>\seq_put_left:No</code>	111, 5020, 5253
<code>\seq_pop:cN</code>	117, 5270, 5273	<code>\seq_put_left:NV</code>	111, 5020, 5251

<code>\seq_put_left:Nv</code>	111 , 5020 , 5252	<code>\seq_show_aux:w</code>	5277 , 5293 , 5301
<code>\seq_put_left:Nx</code>	111 , 5020 , 5254	<code>\seq_tmp:w</code>	5460 , 5468 , 5471
<code>\seq_put_right:cn</code>	112 , 5020	<code>\seq_top:cN</code>	5511 , 5512
<code>\seq_put_right:co</code>	112 , 5020	<code>\seq_top:NN</code>	5511 , 5511
<code>\seq_put_right:cV</code>	112 , 5020	<code>\seq_use:c</code>	120 , 5405
<code>\seq_put_right:cv</code>	112 , 5020	<code>\seq_use:N</code>	120 , 5405 , 5405 , 5406
<code>\seq_put_right:cx</code>	112 , 5020	<code>\seq_wrap_item:n</code> 5433 , 5436 , 5441 , 5446 , 5451 , 5453
<code>\seq_put_right:Nn</code> 112 , 5020 , 5022 , 5026 , 5027 , 5047 , 7068 , 8061 , 8106 , 8121		<code>\setbox</code>	612
<code>\seq_put_right:No</code>	112 , 5020	<code>\setlanguage</code>	370
<code>\seq_put_right:NV</code>	112 , 5020	<code>\sfcode</code>	667
<code>\seq_put_right:Nv</code>	112 , 5020	<code>\shipout</code>	577
<code>\seq_put_right:Nx</code>	112 , 5020	<code>\show</code>	420
<code>\seq_remove_all:cn</code>	114 , 5053	<code>\showbox</code>	422
<code>\seq_remove_all:Nn</code> 114 , 5053 , 5053 , 5077 , 8109	<code>\showboxbreadth</code>	436
<code>\seq_remove_all_aux:NNn</code> 5053 , 5054 , 5056 , 5057	<code>\showboxdepth</code>	437
<code>\seq_remove_duplicates:c</code>	113 , 5037	<code>\showgroups</code>	697
<code>\seq_remove_duplicates:N</code> 113 , 5037 , 5037 , 5051 , 8112	<code>\showifs</code>	698
<code>\seq_remove_duplicates_aux:NN</code> 5037 , 5038 , 5040 , 5041	<code>\showlists</code>	423
<code>\seq_reverse_aux:NN</code>	5463 , 5465 , 5466	<code>\showthe</code>	421
<code>\seq_reverse_aux_item:w</code>	5469 , 5473	<code>\showtokens</code>	685
<code>\seq_set_eq:cc</code>	110 , 5006 , 5009	<code>\skewchar</code>	634
<code>\seq_set_eq:cN</code>	110 , 5006 , 5008	<code>\skip</code>	658
<code>\seq_set_eq:Nc</code>	110 , 5006 , 5007	<code>\skip_add:cn</code>	85 , 4080
<code>\seq_set_eq:NN</code> 110 , 5006 , 5006 , 5038 , 8059 , 8075	<code>\skip_add:Nn</code>	85 , 4080 , 4080 , 4082 , 4083
<code>\seq_set_from_clist:cc</code>	120 , 5433	<code>\skip_eval:n</code>	87 , 4093 , 4110 , 4110
<code>\seq_set_from_clist:cN</code>	120 , 5433	<code>\skip_gadd:cn</code> 86 , 4080
<code>\seq_set_from_clist:cn</code>	120 , 5433	<code>\skip_gadd:Nn</code>	86 , 4080 , 4082 , 4084
<code>\seq_set_from_clist:Nc</code>	120 , 5433	<code>.skip_gset:c</code>	171
<code>\seq_set_from_clist:NN</code> 120 , 5433 , 5433 , 5454 , 5455	<code>\skip_gset:cn</code>	86 , 4069
<code>\seq_set_from_clist:Nn</code> 120 , 5433 , 5438 , 5456	<code>.skip_gset:N</code>	171
<code>\seq_set_reverse:N</code>	121 , 5460 , 5462	<code>\skip_gset:Nn</code>	86 , 4069 , 4071 , 4073
<code>\seq_set_split:Nnn</code>	121 , 5479 , 5479	<code>\skip_gset_eq:cc</code>	86 , 4074
<code>\seq_set_split_aux:NNnn</code> 5479 , 5480 , 5482 , 5483	<code>\skip_gset_eq:cN</code>	86 , 4074
<code>\seq_set_split_aux_end:</code> 5479 , 5492 , 5496 , 5503 , 5507 , 5509	<code>\skip_gset_eq:Nc</code>	86 , 4074
<code>\seq_set_split_aux_i:w</code> 5479 , 5490 , 5497 , 5503	<code>\skip_gset_eq:NN</code>	86 , 4074 , 4077 – 4079
<code>\seq_set_split_aux_ii:w</code> 5479 , 5505 , 5509		<code>\skip_gsub:cn</code>	86 , 4080
<code>\seq_show:c</code>	118 , 5277 , 5514	<code>\skip_gsub:Nn</code>	86 , 4080 , 4087 , 4089
<code>\seq_show:N</code>	118 , 5277 , 5277 , 5302 , 5513	<code>\skip_gzero:c</code> 85 , 4065
<code>\seq_show_aux:n</code>	5277 , 5291 , 5296	<code>\skip_gzero:N</code>	85 , 4065 , 4066 , 4068
		<code>\skip_horizontal:c</code>	91 , 4114
		<code>\skip_horizontal:N</code> 91 , 4114 , 4114 , 4116 , 4120
		<code>\skip_horizontal:n</code>	91 , 4114 , 4115
		<code>\skip_if_eq:nn</code>	4090
		<code>\skip_if_eq:nnTF</code>	87 , 4090
		<code>\skip_if_eq:p:nn</code>	87 , 4090
		<code>\skip_if_infinite_glue:n</code>	4100
		<code>\skip_if_infinite_glue:nTF</code> 87 , 4100 , 4173	
		<code>\skip_if_infinite_glue_p:n</code>	87 , 4100

<code>\skip_new:c</code>	85 , 4057	<code>\str_if_eq:xxTF</code>	24 , 1483 , 2123
<code>\skip_new:N</code>		<code>\str_if_eq_p:nn</code>	24 , 1483 , 1855 , 1856
85 , 4057 , 4058 , 4064 , 4126–4130 , 8515		<code>\str_if_eq_p:no</code>	24 , 1855
<code>.skip_set:c</code>	171	<code>\str_if_eq_p:nV</code>	24 , 1855
<code>\skip_set:cn</code>	86 , 4069	<code>\str_if_eq_p:on</code>	24 , 1855
<code>.skip_set:N</code>	171	<code>\str_if_eq_p:Vn</code>	24 , 1855
<code>\skip_set:Nn</code> ...	86 , 4069 , 4069 , 4071 , 4072	<code>\str_if_eq_p:VV</code>	24 , 1855
<code>\skip_set_eq:cc</code>	86 , 4074	<code>\str_if_eq_p:xx</code>	24 , 1483
<code>\skip_set_eq:cN</code>	86 , 4074	<code>\str_if_eq_return:on</code>	
<code>\skip_set_eq:Nc</code>	86 , 4074	4732 , 4788 , 4803 , 4807 , 4808
<code>\skip_set_eq:NN</code>	86 , 4074 , 4074–4076	<code>\str_if_eq_return:onTF</code>	4788
<code>\skip_show:c</code>	88 , 4122	<code>\str_if_eq_return_p:on</code>	4788
<code>\skip_show:N</code>	88 , 4122 , 4122 , 4123	<code>\str_length_loop:NNNNNNNN</code>	
<code>\skip_split_finite_else_action:nnNN</code>	92 , 4171 , 4171	6730 , 6735 , 6739 , 6745
<code>\skip_sub:cn</code>	86 , 4080	<code>\str_length_skip_spaces:N</code>	6688 , 6730 , 6730
<code>\skip_sub:Nn</code> ...	86 , 4080 , 4085 , 4087 , 4088	<code>\str_length_skip_spaces:n</code>	6730 , 6731 , 6732
<code>\skip_use:c</code>	87 , 4112	<code>\str_tail:n</code>	105 , 4650 , 4658
<code>\skip_use:N</code>	87 , 4111 , 4112 , 4112 , 4113	<code>\str_tail_aux:w</code>	4660 , 4664
<code>\skip_vertical:c</code>	91 , 4114	<code>\strcmp</code>	230
<code>\skip_vertical:N</code>	91 , 4114 , 4117 , 4119 , 4121	<code>\string</code>	223 , 238 , 252 , 639
<code>\skip_vertical:n</code>	91 , 4114 , 4118		
<code>\skip_zero:c</code>	85 , 4065		
<code>\skip_zero:N</code>	85 , 4065 , 4065–4067		
<code>\skipdef</code>	357		
<code>\space</code>	49 , 205		
<code>\spacefactor</code>	576		
<code>\spaceskip</code>	571		
<code>\span</code>	384		
<code>\special</code>	646		
<code>\splitbotmark</code>	455		
<code>\splitbotmarks</code>	681		
<code>\splitdiscards</code>	728		
<code>\splitfirstmark</code>	454		
<code>\splitfirstmarks</code>	680		
<code>\splitmaxdepth</code>	624		
<code>\splittopskip</code>	625		
<code>\str_head:n</code> ...	105 , 4650 , 4650 , 4671 , 4714		
<code>\str_head_aux:w</code>	4650 , 4652 , 4656		
<code>\str_if_eq:nn</code>	1483		
<code>\str_if_eq:nnF</code>	1859 , 1860		
<code>\str_if_eq:nnT</code>	1857 , 1858 , 5061 , 6040 , 6167		
<code>\str_if_eq:nnTF</code>	24 , 1483 , 1861 , 1862 , 2114 , 3786 , 3789 , 7405		
<code>\str_if_eq:noTF</code>	24 , 1855		
<code>\str_if_eq:nVTF</code>	24 , 1855		
<code>\str_if_eq:onTF</code>	24 , 1855		
<code>\str_if_eq:VnTF</code>	24 , 1855		
<code>\str_if_eq:VVTF</code>	24 , 1855		
<code>\str_if_eq:xx</code>	1489		

10595, 10600, 10601, 10605, 10606, 10609, 10610, 10613, 10614, 10704, 10708, 10756, 10779, 10808, 10838, 10846, 10849, 10896, 10899, 10900, 10902, 10918, 10938, 10942, 10955, 10956, 10959, 10960, 10965, 10966			
<code>\tex_afterassignment:D</code>	372, 2884, 2970, 8253	<code>\tex_displayindent:D</code>	485
<code>\tex_aftergroup:D</code>	373, 816	<code>\tex_displaylimits:D</code>	495
<code>\tex_atop:D</code>	469	<code>\tex_displaystyle:D</code>	473
<code>\tex_atopwithdelims:D</code>	470	<code>\tex_displaywidowpenalty:D</code>	484
<code>\tex_badness:D</code>	617	<code>\tex_displaywidth:D</code>	486
<code>\tex_baselineskip:D</code>	545	<code>\tex_divide:D</code>	
<code>\tex_batchmode:D</code>	438		363, 8331, 8378, 8403, 8972, 9240, 9431, 9496, 9540, 9585, 9588, 9590, 9592, 9656, 9698, 10801, 10851–10853
<code>\tex_begingroup:D</code>	376, 812	<code>\tex_doublehyphendemerits:D</code>	553
<code>\tex_belowdisplayshortskip:D</code>	482	<code>\tex_dp:D</code>	664, 6234
<code>\tex_belowdisplayskip:D</code>	483	<code>\tex_dump:D</code>	647
<code>\tex_binoppenalty:D</code>	506	<code>\tex_edef:D</code>	351, 825
<code>\tex_botmark:D</code>	453	<code>\tex_else:D</code>	404, 787
<code>\tex_box:D</code>	661, 6228, 6248	<code>\tex_emergencystretch:D</code>	568
<code>\tex_boxmaxdepth:D</code>	623	<code>\tex_end:D</code>	442, 763, 1183, 6995, 7137
<code>\tex_brokenpenalty:D</code>	580	<code>\tex_endcsname:D</code>	444, 808
<code>\tex_catcode:D</code>		<code>\tex_endgroup:D</code>	377, 761, 813
	665, 1074, 1815–1821, 2266, 2280, 2491, 2493, 2495, 3110, 4298, 4299	<code>\tex_endinput:D</code>	416, 7006
<code>\tex_char:D</code>	519	<code>\tex_endlinechar:D</code>	
<code>\tex_chardef:D</code> .	354, 1061, 1062, 1163– 1167, 1893, 1895, 2810, 6413, 6416		307, 308, 322, 458, 4314, 4341, 4354
<code>\tex_cleaders:D</code>	537	<code>\tex_eqno:D</code>	478
<code>\tex_closein:D</code>	413, 6548	<code>\tex_errhelp:D</code>	424, 6913
<code>\tex_closeout:D</code>	408, 6560	<code>\tex_errmessage:D</code>	418, 1175, 6933
<code>\tex_clubpenalty:D</code>	548	<code>\tex_errorcontextlines:D</code>	425, 6951
<code>\tex_copy:D</code>	605, 6222, 6249	<code>\tex_errorstopmode:D</code>	439
<code>\tex_count:D</code>	656	<code>\tex_escapechar:D</code>	457
<code>\tex_countdef:D</code>	355, 1160, 2742	<code>\tex_everycr:D</code>	386
<code>\tex_cr:D</code>	380	<code>\tex_everydisplay:D</code>	487, 764
<code>\tex_crcr:D</code>	381	<code>\tex_everyhbox:D</code>	626
<code>\tex_csname:D</code>	443, 807	<code>\tex_everyjob:D</code>	
<code>\tex_day:D</code>	651		655, 4771, 4773, 8020, 8022, 8032, 8034
<code>\tex_deadcycles:D</code>	585	<code>\tex_everymath:D</code>	511, 765
<code>\tex_def:D</code>	350, 820–824	<code>\tex_everypar:D</code>	574
<code>\tex_defaultthyphenchar:D</code>	635	<code>\tex_everyvbox:D</code>	627
<code>\tex_defaultskewchar:D</code>	636	<code>\tex_exhyphenpenalty:D</code>	550
<code>\tex_delcode:D</code>	666	<code>\tex_expandafter:D</code>	374, 802
<code>\tex_delimiter:D</code>	460	<code>\tex_fam:D</code>	366
<code>\tex_delimiterfactor:D</code>	509	<code>\tex_fi:D</code>	405, 788
<code>\tex_delimitershortfall:D</code>	508	<code>\tex_finalhyphendemerits:D</code>	554
<code>\tex_dimen:D</code>	657	<code>\tex_firstmark:D</code>	452
<code>\tex_dimendef:D</code>	356, 2764	<code>\tex_floatingpenalty:D</code>	599
<code>\tex_discretionary:D</code>	520	<code>\tex_font:D</code>	365
		<code>\tex_fontdimen:D</code>	632
		<code>\tex_fontname:D</code>	456
		<code>\tex_futurelet:D</code>	361, 2878, 2880
		<code>\tex_gdef:D</code>	352, 838
		<code>\tex_global:D</code> 336, 341, 343, 367, 817, 1895	
		<code>\tex_globaldefs:D</code>	371
		<code>\tex_halign:D</code>	378

<code>\tex_hangafter:D</code>	556	<code>\tex_language:D</code>	449
<code>\tex_hangindent:D</code>	557	<code>\tex_lastbox:D</code>	606, 6281
<code>\tex_hbadness:D</code>	618	<code>\tex_lastkern:D</code>	539
<code>\tex_hbox:D</code>	613, 6303, 6304, 6309, 6315, 6323, 6324	<code>\tex_lastpenalty:D</code>	645
<code>\tex_hfil:D</code>	521	<code>\tex_lastskip:D</code>	540
<code>\tex_hfill:D</code>	523	<code>\tex_lccode:D</code>	668, 1073, 1813, 1814, 2265, 2279, 2567, 2569, 2571, 3112, 4296, 4297
<code>\tex_hfilneg:D</code>	522	<code>\tex_leaders:D</code>	536
<code>\tex_hfuzz:D</code>	620	<code>\tex_left:D</code>	504
<code>\tex_hoffset:D</code>	595	<code>\tex_lefthyphenmin:D</code>	560
<code>\tex_holdinginserts:D</code>	598	<code>\tex_leftskip:D</code>	562
<code>\tex_hruler:D</code>	534	<code>\tex_leqno:D</code>	479
<code>\tex_hsize:D</code>	559	<code>\tex_let:D</code> 337, 341, 343, 349, 763–773, 783	
<code>\tex_hskip:D</code>	524, 4114	<code>\tex_limits:D</code>	496
<code>\tex_hss:D</code>	525, 6326, 6328	<code>\tex_linepenalty:D</code>	552
<code>\tex_ht:D</code>	663, 6233	<code>\tex_lineskip:D</code>	546
<code>\tex_hyphen:D</code>	348, 766	<code>\tex_lineskiplimit:D</code>	547
<code>\tex_hyphenation:D</code>	649	<code>\tex_long:D</code>	368, 818, 820
<code>\tex_hyphenchar:D</code>	633	<code>\tex_looseness:D</code>	564
<code>\tex_hyphenpenalty:D</code>	551	<code>\tex_lower:D</code>	601, 6259
<code>\tex_if:D</code>	387, 790, 793	<code>\tex_lowercase:D</code> 640, 1075, 1822, 4300, 4361	
<code>\tex_ifcase:D</code>	388, 3201	<code>\tex_mag:D</code>	448
<code>\tex_ifcat:D</code>	389, 794	<code>\tex_mark:D</code>	450
<code>\tex_ifdim:D</code>	392, 3895	<code>\tex_mathaccent:D</code>	461
<code>\tex_ifeof:D</code>	393, 6375	<code>\tex_mathbin:D</code>	491
<code>\tex_iffalse:D</code>	398, 785	<code>\tex_mathchar:D</code>	462
<code>\tex_ifhbox:D</code>	394, 6260	<code>\tex_mathchardef:D</code> 359, 1168, 3289, 11520	
<code>\tex_ifhmode:D</code>	400, 797	<code>\tex_mathchoice:D</code>	459
<code>\tex_ifinner:D</code>	403, 799	<code>\tex_mathclose:D</code>	492
<code>\tex_ifmmode:D</code>	401, 796	<code>\tex_mathcode:D</code> 670, 2561, 2563, 2565, 3111	
<code>\tex_ifnum:D</code>	390, 814, 3199	<code>\tex_mathinner:D</code>	493
<code>\tex_ifodd:D</code> ...	391, 791, 792, 1201, 3200	<code>\tex_mathop:D</code>	494
<code>\tex_iftrue:D</code>	399, 784	<code>\tex_mathopen:D</code>	498
<code>\tex_ifvbox:D</code>	395, 6261	<code>\tex_mathord:D</code>	499
<code>\tex_ifvmode:D</code>	402, 798	<code>\tex_mathpunct:D</code>	500
<code>\tex_ifvoid:D</code>	396, 6262	<code>\tex_mathrel:D</code>	501
<code>\tex_ifx:D</code>	397, 795	<code>\tex_mathsurround:D</code>	512
<code>\tex_ignorespaces:D</code>	445	<code>\tex_maxdeadcycles:D</code>	582
<code>\tex_immediate:D</code>	407, 1170, 1172, 6447, 6560, 6625	<code>\tex_maxdepth:D</code>	583
<code>\tex_indent:D</code>	541	<code>\tex_meaning:D</code>	642, 805, 809
<code>\tex_input:D</code>	415, 767, 8099	<code>\tex_medmuskip:D</code>	513
<code>\tex_inputlineno:D</code> .	417, 1190, 1800, 6853	<code>\tex_message:D</code>	419
<code>\tex_insert:D</code>	597	<code>\tex_mkern:D</code>	466
<code>\tex_insertpenalties:D</code>	600	<code>\tex_month:D</code>	652
<code>\tex_interlinepenalty:D</code>	579	<code>\tex_moveleft:D</code>	602, 6253
<code>\tex_italic_correction:D</code>	768	<code>\tex_moveright:D</code>	603, 6255
<code>\tex_italiccor:D</code>	347	<code>\tex_mskip:D</code>	463
<code>\tex_jobname:D</code>	654, 4781, 8023	<code>\tex_multiply:D</code>	364, 9170, 9377, 9591, 9607, 10575, 11164
<code>\tex_kern:D</code>	532		

<code>\tex_muskip:D</code>	660	<code>\tex_relpentalty:D</code>	507
<code>\tex_muskipdef:D</code>	358	<code>\tex_right:D</code>	505
<code>\tex_newlinechar:D</code> .	414, 4315, 4342, 4355	<code>\tex_righthyphenmin:D</code>	561
<code>\tex_noalign:D</code>	382	<code>\tex_rightskip:D</code>	563
<code>\tex_noboundary:D</code>	517	<code>\tex_romannumeral:D</code>	
<code>\tex_noexpand:D</code>	375, 803	. 638, 815, 1537, 1549, 1555, 1597,	
<code>\tex_noindent:D</code>	543	1601, 1606, 1612, 1618, 1624, 1636,	
<code>\tex_nolimits:D</code>	497	1641, 1643, 1650, 1704, 1711, 1716,	
<code>\tex_nonscript:D</code>	477	1719, 1721, 1724, 1729, 1735, 1746,	
<code>\tex_nonstopmode:D</code>	440	1756, 1760, 1765, 4810, 4862, 4881,	
<code>\tex_nulldelimiterspace:D</code>	510	4921, 4923, 4943, 5866, 5877, 7331	
<code>\tex_nullfont:D</code>	628, 2837	<code>\tex_scriptfont:D</code>	630
<code>\tex_number:D</code>	637, 3196	<code>\tex_scriptscriptfont:D</code>	631
<code>\tex_omit:D</code>	383	<code>\tex_scriptscriptstyle:D</code>	476
<code>\tex_openin:D</code>	409, 6434	<code>\tex_scriptspace:D</code>	516
<code>\tex_openout:D</code>	410, 6447	<code>\tex_scriptstyle:D</code>	475
<code>\tex_or:D</code>	406, 786	<code>\tex_scrollmode:D</code>	441
<code>\tex_outer:D</code>	369	<code>\tex_setbox:D</code>	
<code>\tex_output:D</code>	584	612, 6222, 6228, 6283, 6304, 6309,	
<code>\tex_outputpenalty:D</code>	594	6315, 6338, 6343, 6349, 6355, 6367	
<code>\tex_over:D</code>	471	<code>\tex_setlanguage:D</code>	370
<code>\tex_overfullrule:D</code>	622	<code>\tex_sfcode:D</code> .	667, 2579, 2581, 2583, 3114
<code>\tex_overline:D</code>	502	<code>\tex_shipout:D</code>	577
<code>\tex_overwithdelims:D</code>	472	<code>\tex_show:D</code>	420, 810
<code>\tex_pagedepth:D</code>	586	<code>\tex_showbox:D</code>	422, 6301
<code>\tex_pagefilllstretch:D</code>	590	<code>\tex_showboxbreadth:D</code>	436
<code>\tex_pagefillstretch:D</code>	589	<code>\tex_showboxdepth:D</code>	437
<code>\tex_pagefilstretch:D</code>	588	<code>\tex_showlists:D</code>	423
<code>\tex_pagegoal:D</code>	592	<code>\tex_showthe:D</code>	
<code>\tex_pageshrink:D</code>	591	421, 1441, 2495, 2565, 2571, 2577,	
<code>\tex_pagestretch:D</code>	587	2583, 3117, 3120, 3123, 3126, 3129	
<code>\tex_pagetotal:D</code>	593	<code>\tex_skewchar:D</code>	634
<code>\tex_par:D</code>	542	<code>\tex_skip:D</code>	658
<code>\tex_parfillskip:D</code>	573	<code>\tex_skipdef:D</code>	357, 2753
<code>\tex_parindent:D</code>	566	<code>\tex_space:D</code>	346
<code>\tex_parshape:D</code>	558	<code>\tex_spacefactor:D</code>	576
<code>\tex_parskip:D</code>	565	<code>\tex_spaceskip:D</code>	571
<code>\tex_patterns:D</code>	648	<code>\tex_span:D</code>	384
<code>\tex_pausing:D</code>	435	<code>\tex_special:D</code>	646
<code>\tex_penalty:D</code>	643	<code>\tex_splitbotmark:D</code>	455
<code>\tex_postdisplaypenalty:D</code>	490	<code>\tex_splitfirstmark:D</code>	454
<code>\tex_predisplaypenalty:D</code>	489	<code>\tex_splitmaxdepth:D</code>	624
<code>\tex_predisplaysize:D</code>	488	<code>\tex_splittopskip:D</code>	625
<code>\tex_pretolerance:D</code>	569	<code>\tex_string:D</code>	639, 806
<code>\tex_prevdepth:D</code>	616	<code>\tex_tabskip:D</code>	385
<code>\tex_prevgraf:D</code>	575	<code>\tex_textfont:D</code>	629
<code>\tex_radical:D</code>	464	<code>\tex_textstyle:D</code>	474
<code>\tex_raise:D</code>	604, 6257	<code>\tex_the:D</code>	
<code>\tex_read:D</code>	411, 6763, 6765	. 308, 447, 1190, 1567, 1571, 1800,	
<code>\tex_relax:D</code>	446, 811, 3198, 3897	2493, 2563, 2569, 2575, 2581, 3115,	

- 3118, 3121, 3124, 3127, 3338, 4039,
 4112, 4167, 4773, 8022, 8034, 11544
 \tex_thickmuskip:D 515
 \tex_thinmuskip:D 514
 \tex_time:D 650
 \tex_toks:D 659
 \tex_toksdef:D 360, 2775
 \tex_tolerance:D 570
 \tex_topmark:D 451
 \tex_topskip:D 581
 \tex_tracingcommands:D 426
 \tex_tracinglostchars:D 427
 \tex_tracingmacros:D 428
 \tex_tracingonline:D 429
 \tex_tracingoutput:D 430
 \tex_tracingpages:D 431
 \tex_tracingparagraphs:D 432
 \tex_tracingrestores:D 433
 \tex_tracingstats:D 434
 \tex_uccode:D . 669, 2573, 2575, 2577, 3113
 \tex_uchyph:D 567
 \tex_undefined:D 336, 343
 \tex_underline:D 503, 769
 \tex_unhbox:D 608, 6330
 \tex_unhcopy:D 609, 6329
 \tex_unkern:D 533
 \tex_unpenalty:D 644
 \tex_unskip:D 531
 \tex_unvbox:D 610, 6363
 \tex_unvcopy:D 611, 6362
 \tex_uppercase:D 641, 4362
 \tex_vadjust:D 544
 \tex_valign:D 379
 \tex_vbadness:D 619
 \tex_vbox:D 614, 6333, 6336–6338, 6349, 6355
 \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, 4117
 \tex_vsplit:D 607, 6367
 \tex_vss:D 530
 \tex_vtop:D 615, 6334, 6343
 \tex_wd:D 662, 6235
 \tex_widowpenalty:D 549
 \tex_write:D 412, 1170, 1172, 6620
 \tex_xdef:D 353, 839
 \tex_xleaders:D 538
 \tex_xspaceskip:D 572
 \tex_year:D 653
 \textasteriskcentered 3877, 3883
 \textbardbl 3882
 \textdagger 3878, 3884
 \textdaggerdbl 3879, 3885
 \textfont 629
 \textparagraph 3881
 \textsection 3880
 \textstyle 474
 \texttt 6878
 \TeXETstate 729
 \the 70–79, 447
 \thickmuskip 515
 \thinmuskip 514
 \time 650
 \tl_act:NNNnn 4810, 4810
 \tl_act_aux:NNNnn 4810,
 4810, 4811, 4863, 4882, 4898, 4926
 \tl_act_case_aux:nn 4921, 4923, 4924, 4943
 \tl_act_case_group:nn .. 4920, 4928, 4940
 \tl_act_case_normal:nN . 4920, 4927, 4932
 \tl_act_case_space:n ... 4920, 4929, 4931
 \tl_act_end:w 4810
 \tl_act_end:wn 4831, 4837
 \tl_act_group:nwnNNN ... 4810, 4823, 4838
 \tl_act_group_recurse:Nnn 4810, 4855, 4875
 \tl_act_length_group:nn 4894, 4900, 4908
 \tl_act_length_normal:nN 4894, 4899, 4906
 \tl_act_length_space:n . 4894, 4901, 4907
 \tl_act_loop:w
 .. 4810, 4813, 4817, 4834, 4841, 4848
 \tl_act_normal:NwnNNN .. 4810, 4820, 4828
 \tl_act_output:n
 4810, 4851, 4931, 4934, 4942
 \tl_act_result:n .. 4815, 4837, 4851–4854
 \tl_act_reverse_group:nn 4860, 4865, 4873
 \tl_act_reverse_group_preserve:nn ..
 4884, 4888
 \tl_act_reverse_normal:nN
 4860, 4864, 4871, 4883
 \tl_act_reverse_output:n
 .. 4810, 4853, 4870, 4872, 4876, 4889
 \tl_act_reverse_space:n
 4860, 4866, 4869, 4885
 \tl_act_space:wnnNNN ... 4810, 4824, 4845
 \tl_clear:c 93, 4209, 4999, 5527

- \tl_clear:N 93, [4209](#), 4209, 4213,
4216, 4317, 4998, 5526, 6652, 6653,
6721, 7379, 7383, 8055, 8958, 9236,
9254, 9489, 9513, 9533, 9563, 9577
- \tl_clear_new:c
. 93, [4215](#), 5003, 5531, 7667, 7669
- \tl_clear_new:N
. 93, [4215](#), 4215, 4219, 5002, 5530
- \tl_const:cn
. 93, [4197](#), 10280–10319, 10626–10637
- \tl_const:cx 93, [4197](#), 10716
- \tl_const:Nn
. 93, 2383, [4197](#), 4197, 4207, 4303,
4782, 4783, 4910, 4915, 5899, 6380,
6788, 6789, 6819, 6824, 6826, 6828,
6830, 6832, 6837, 6838, 6845, 6863,
7311, 7313, 7486–7490, 8148–8152
- \tl_const:Nx 93, [4197](#), 4202, 4208, 4781
- \tl_elt_count:c [4972](#), 4976
- \tl_elt_count:N [4972](#), 4975
- \tl_elt_count:n [4972](#), 4972
- \tl_elt_count:o [4972](#), 4974
- \tl_elt_count:V [4972](#), 4973
- \tl_error_message: [4566](#)
- \tl_expandable_lowercase:n
. 109, [4920](#), 4922
- \tl_expandable_uppercase:n
. 109, [4920](#), 4920
- \tl_gclear:c 93, [4209](#), 5001, 5529
- \tl_gclear:N
93, [4209](#), 4211, 4214, 4218, 5000, 5528
- \tl_gclear_new:c 93, [4215](#), 5005, 5533
- \tl_gclear_new:N
. 93, [4215](#), 4217, 4220, 5004, 5532
- \tl_gput_left:cn 95, [4247](#)
- \tl_gput_left:co 95, [4247](#)
- \tl_gput_left:cV 95, [4247](#)
- \tl_gput_left:cx 95, [4247](#)
- \tl_gput_left:Nn
. 95, [4247](#), 4255, 4267, 5029, 5567
- \tl_gput_left:No 95, [4247](#), 4259, 4269
- \tl_gput_left:Nv 95, [4247](#), 4257, 4268
- \tl_gput_left:Nx 95, [4247](#), 4261, 4270
- \tl_gput_right:cn 95, [4271](#)
- \tl_gput_right:co 95, [4271](#)
- \tl_gput_right:cV 95, [4271](#)
- \tl_gput_right:cx 95, [4271](#)
- \tl_gput_right:Nn
. 95, [4271](#), 4279, 4291, 5031, 5581
- \tl_gput_right:No 95, [4271](#), 4283, 4293
- \tl_gput_right:Nv 95, [4271](#), 4281, 4292
- \tl_gput_right:Nx 95, [4271](#), 4285, 4294, 6004
- \tl_gremove_all:cn 97, [4418](#), 4971
- \tl_gremove_all:Nn
. 97, [4418](#), 4420, 4423, 4970
- \tl_gremove_all_in:cn [4964](#), 4971
- \tl_gremove_all_in:Nn [4964](#), 4970
- \tl_gremove_in:cn [4964](#), 4967
- \tl_gremove_in:Nn [4964](#), 4966
- \tl_gremove_once:cn 97, [4412](#), 4967
- \tl_gremove_once:Nn
. 97, [4412](#), 4414, 4417, 4966
- \tl_greplace_all:cnn 96, [4364](#), 4963
- \tl_greplace_all:Nnn
. 96, [4364](#), 4370, 4375, 4421, 4962
- \tl_greplace_all_in:cnn [4956](#), 4963
- \tl_greplace_all_in:Nnn [4956](#), 4962
- \tl_greplace_in:cnn [4956](#), 4959
- \tl_greplace_in:Nnn [4956](#), 4958
- \tl_greplace_once:cnn 96, [4364](#), 4959
- \tl_greplace_once:Nnn
. 96, [4364](#), 4366, 4373, 4415, 4958
- .tl_gset:c 172
- \tl_gset:cf 94, [4229](#)
- \tl_gset:cn 94, [4229](#)
- \tl_gset:co 94, [4229](#)
- \tl_gset:cV 94
- \tl_gset:cv 94
- \tl_gset:cx 94, [4229](#), 9897, 9983, 10264
- .tl_gset:N 172
- \tl_gset:Nc [4952](#), 4952
- \tl_gset:Nf 94, [4229](#), 5682
- \tl_gset:Nn 94, [4229](#), 4235,
4244, 4246, 4309, 4336, 4948, 5118,
5341, 5567, 5581, 5594, 5941, 5967,
6135, 6174, 8098, 8433, 8887, 8922,
9003, 9028, 9054, 9157, 9175, 9288,
9830, 9927, 10128, 10321, 10639, 10973
- \tl_gset:No 94, [4229](#), 4237
- \tl_gset:Nv 94, [4229](#)
- \tl_gset:Nv 94, [4229](#)
- \tl_gset:Nx 94, [4229](#), 4239,
4245, 4367, 4371, 4638, 5017, 5056,
5159, 5355, 5445, 5450, 5465, 5482,
5545, 5876, 5980, 8023, 8469, 10623
- \tl_gset_eq:cc
94, [4221](#), 4228, 5013, 5541, 5919, 8523
- \tl_gset_eq:cN
94, [4221](#), 4226, 5012, 5540, 5918, 8521

- \tl_gset_eq:Nc 94, [4221](#), [4227](#), [5011](#), [5539](#), [5917](#), [8522](#)
- \tl_gset_eq:NN 94, [4212](#), [4221](#), [4225](#), [5010](#), [5538](#), [5916](#), [8027](#), [8417](#), [8429](#), [8520](#)
- \tl_gset_rescan:cnm 98, [4306](#)
- \tl_gset_rescan:cno 98, [4306](#)
- \tl_gset_rescan:cnx 98, [4333](#)
- \tl_gset_rescan:Nnn 98, [4306](#), [4308](#), [4331](#), [4332](#)
- \tl_gset_rescan:Nno 98, [4306](#)
- \tl_gset_rescan:Nnx . 98, [4333](#), [4335](#), [4349](#)
- .tl_gset_x:c 172
- .tl_gset_x:N 172
- \tl_gtrim_spaces:c 104, [4596](#)
- \tl_gtrim_spaces:N . 104, [4596](#), [4637](#), [4640](#)
- \tl_head:f 104, [4641](#)
- \tl_head:n ... 104, [4641](#), [4643](#), [4648](#), [4977](#)
- \tl_head:V 104, [4641](#)
- \tl_head:v 104, [4641](#)
- \tl_head:w 105, [4641](#), [4641](#), [4644](#), [4657](#), [4670](#), [4686](#), [4706](#), [4978](#), [8227](#), [8238](#)
- \tl_head_i:n 4977, [4977](#)
- \tl_head_i:w 4977, [4978](#)
- \tl_head_iii:f 4977
- \tl_head_iii:n 4977, [4979](#), [4980](#)
- \tl_head_iii:w 4977, [4979](#), [4981](#)
- \tl_if_blank:n 4424
- \tl_if_blank:nF 3777, [4428](#), [4432](#)
- \tl_if_blank:nT 4427, [4431](#)
- \tl_if_blank:nTF ... 99, [4424](#), [4429](#), [4433](#)
- \tl_if_blank:oTF 99, [4424](#), [7392](#)
- \tl_if_blank:VTF 99, [4424](#)
- \tl_if_blank_p:n ... 99, [4424](#), [4426](#), [4430](#)
- \tl_if_blank_p:o 99, [4424](#)
- \tl_if_blank_p:V 99, [4424](#)
- \tl_if_blank_p_aux:NNw 4424
- \tl_if_empty:c 5081, [5687](#)
- \tl_if_empty:cTF 99, [4434](#)
- \tl_if_empty:N 4434, [5079](#), [5686](#)
- \tl_if_empty:n 4446
- \tl_if_empty:NF 4444, [5659](#), [8089](#)
- \tl_if_empty:nF .. 2988, [4457](#), [5572](#), [5737](#)
- \tl_if_empty:NT 4443
- \tl_if_empty:nT 4456
- \tl_if_empty:NTF 99, [4434](#), [4445](#), [7449](#), [8082](#)
- \tl_if_empty:nTF 99, [2732](#), [2739](#), [2750](#), [2761](#), [2772](#), [2783](#), [2792](#), [2799](#), [2808](#), [3819](#), [4378](#), [4446](#), [4455](#), [5485](#), [6869](#), [7561](#)
- \tl_if_empty:o 4467
- \tl_if_empty:oTF ... 99, [2829](#), [4458](#), [4505](#)
- \tl_if_empty:VTF 99, [4446](#)
- \tl_if_empty_p:c 99, [4434](#)
- \tl_if_empty_p:N 99, [4434](#), [4442](#)
- \tl_if_empty_p:n 99, [4446](#), [4454](#)
- \tl_if_empty_p:o 99, [4458](#)
- \tl_if_empty_p:V 99, [4446](#)
- \tl_if_empty_return:o 4425, [4458](#), [4458](#), [4468](#)
- \tl_if_eq:cc 5691, [6185](#)
- \tl_if_eq:ccTF 99, [4469](#), [7892](#)
- \tl_if_eq:cN 5690, [6183](#)
- \tl_if_eq:cNTF 99, [4469](#)
- \tl_if_eq:Nc 5689, [6184](#)
- \tl_if_eq:NcTF 99, [4469](#)
- \tl_if_eq:NN 4469, [5688](#), [6182](#)
- \tl_if_eq:nn 4481
- \tl_if_eq:NNF 4480
- \tl_if_eq:NNT 4479, [5068](#)
- \tl_if_eq:NNTF 99, [2131](#), [4469](#), [4478](#), [6677](#), [6680](#), [7104](#)
- \tl_if_eq:nnTF 100, [4481](#)
- \tl_if_eq_p:cc 99, [4469](#)
- \tl_if_eq_p:cN 99, [4469](#)
- \tl_if_eq_p:Nc 99, [4469](#)
- \tl_if_eq_p:NN 99, [4469](#), [4477](#)
- \tl_if_head_eq_catcode:nN 4681
- \tl_if_head_eq_catcode:nNTF .. 106, [4665](#)
- \tl_if_head_eq_catcode_p:nN .. 106, [4665](#)
- \tl_if_head_eq_charcode:fNTF .. 106, [4665](#)
- \tl_if_head_eq_charcode:nN 4665
- \tl_if_head_eq_charcode:nNF 4680
- \tl_if_head_eq_charcode:nNT 4679
- \tl_if_head_eq_charcode:nNTF 106, [3682](#), [3695](#), [4665](#), [4678](#)
- \tl_if_head_eq_charcode_p:fN . 106, [4665](#)
- \tl_if_head_eq_charcode_p:nN 106, [4665](#), [4677](#)
- \tl_if_head_eq_meaning:nN 4697
- \tl_if_head_eq_meaning:nNTF 106, [4665](#), [7421](#)
- \tl_if_head_eq_meaning_aux_normal:nN 4700, [4704](#)
- \tl_if_head_eq_meaning_aux_special:nN 4701, [4712](#)
- \tl_if_head_eq_meaning_p:nN .. 106, [4665](#)
- \tl_if_head_group:n 4733
- \tl_if_head_group:nTF 106, [4688](#), [4722](#), [4733](#), [4822](#)

- \tl_if_head_group_p:n [106](#), [4733](#)
- \tl_if_head_N_type:n [4731](#)
- \tl_if_head_N_type:nTF
[106](#), [4669](#), [4685](#), [4699](#), [4731](#), [4806](#), [4819](#)
- \tl_if_head_N_type_p:n [106](#), [4731](#)
- \tl_if_head_space:n [4749](#)
- \tl_if_head_space:nTF [107](#), [4749](#)
- \tl_if_head_space_aux:w [4749](#), [4754](#), [4761](#)
- \tl_if_head_space_p:n [107](#), [4749](#)
- \tl_if_in:cnTF [100](#), [4496](#)
- \tl_if_in:nn [4502](#)
- \tl_if_in:NnF [4497](#), [4500](#)
- \tl_if_in:nnF [4497](#), [4509](#), [4512](#), [4515](#)
- \tl_if_in:NnT [4496](#), [4499](#)
- \tl_if_in:nnT [4496](#), [4508](#), [4511](#), [4514](#)
- \tl_if_in:NnTF [100](#), [4496](#), [4498](#), [4501](#)
- \tl_if_in:nnTF [100](#), [4498](#),
[4502](#), [4510](#), [4513](#), [4516](#), [7531](#), [7538](#)
- \tl_if_in:noTF [4502](#)
- \tl_if_in:onTF [100](#), [4502](#)
- \tl_if_in:VnTF [100](#), [4502](#)
- \tl_if_single:cTF [100](#)
- \tl_if_single:n [4802](#)
- \tl_if_single:Nf [4800](#)
- \tl_if_single:nF [4800](#)
- \tl_if_single:NT [4799](#)
- \tl_if_single:nT [4799](#)
- \tl_if_single:Nf [100](#), [4798](#), [4801](#)
- \tl_if_single:nTF [100](#), [4801](#), [4802](#)
- \tl_if_single_p:c [100](#)
- \tl_if_single_p:N [100](#), [4798](#), [4798](#)
- \tl_if_single_p:n [100](#), [4798](#), [4802](#)
- \tl_if_single_token:n [4804](#)
- \tl_if_single_token:nTF [100](#), [4804](#)
- \tl_if_single_token_p:n [100](#), [4804](#)
- \tl_length:c [103](#), [4575](#), [4976](#)
- \tl_length:N .. [103](#), [4575](#), [4580](#), [4587](#), [4975](#)
- \tl_length:n .. [103](#), [4575](#), [4575](#), [4586](#), [4972](#)
- \tl_length:o [103](#), [4575](#), [4974](#)
- \tl_length:V [103](#), [4575](#), [4973](#)
- \tl_length_aux:n . [4575](#), [4578](#), [4583](#), [4585](#)
- \tl_length_tokens:n [108](#), [4894](#), [4894](#), [4909](#)
- \tl_map_break: [102](#), [4562](#), [4562](#), [6457](#), [6465](#)
- \tl_map_function:cN [101](#), [4517](#)
- \tl_map_function:NN
[101](#), [4517](#), [4519](#), [4529](#), [4583](#), [6428](#), [6441](#)
- \tl_map_function:nN [101](#), [4517](#), [4517](#), [4578](#)
- \tl_map_function_aux:NN [4517](#)
- \tl_map_function_aux:Nn
... [4518](#), [4521](#), [4524](#), [4527](#), [4535](#), [4545](#)
- \tl_map_inline:cn [101](#), [4530](#)
- \tl_map_inline:Nn . [101](#), [4530](#), [4540](#), [4550](#)
- \tl_map_inline:nn . [101](#), [2721](#), [4530](#), [4530](#)
- \tl_map_inline_aux:n [4530](#)
- \tl_map_variable:cNn [101](#), [4551](#)
- \tl_map_variable:NNn [101](#), [4551](#), [4553](#), [4561](#)
- \tl_map_variable:nNn [101](#), [4551](#), [4551](#), [4554](#)
- \tl_map_variable_aux:NnN [4551](#)
- \tl_map_variable_aux:Nnn [4552](#), [4555](#), [4559](#)
- \tl_new:c [93](#), [4191](#),
[4997](#), [5525](#), [7649](#), [9896](#), [9982](#), [10263](#)
- \tl_new:cn [4945](#)
- \tl_new:N . [93](#), [4191](#), [4191](#), [4196](#), [4216](#),
[4218](#), [4305](#), [4363](#), [4494](#), [4495](#), [4784](#)–
[4787](#), [4947](#), [4994](#)–[4996](#), [5276](#), [5522](#)–
[5524](#), [6097](#), [6641](#)–[6644](#), [6787](#), [6861](#),
[7053](#)–[7055](#), [7364](#)–[7367](#), [7492](#)–[7494](#),
[7496](#)–[7499](#), [8018](#), [8038](#), [8153](#), [8183](#),
[8190](#), [8193](#), [8196](#), [8416](#), [10622](#), [11572](#)
- \tl_new:Nn [4945](#), [4945](#), [4950](#), [4951](#)
- \tl_new:Nx [4945](#)
- \tl_put_left:cn [95](#), [4247](#)
- \tl_put_left:co [95](#), [4247](#)
- \tl_put_left:cV [95](#), [4247](#)
- \tl_put_left:cx [95](#), [4247](#)
- \tl_put_left:Nn
... [95](#), [4247](#), [4247](#), [4263](#), [5021](#), [5565](#)
- \tl_put_left:No [95](#), [4247](#), [4251](#), [4265](#)
- \tl_put_left:NV [95](#), [4247](#), [4249](#), [4264](#)
- \tl_put_left:Nx [95](#), [4247](#), [4253](#), [4266](#)
- \tl_put_right:cn [95](#), [4271](#)
- \tl_put_right:co [95](#), [4271](#)
- \tl_put_right:cV [95](#), [4271](#)
- \tl_put_right:cx [95](#), [4271](#)
- \tl_put_right:Nn [95](#), [4271](#),
[4271](#), [4287](#), [5023](#), [5579](#), [5653](#), [7450](#)
- \tl_put_right:No [95](#), [4271](#), [4275](#), [4289](#), [6931](#)
- \tl_put_right:NV ... [95](#), [4271](#), [4273](#), [4288](#)
- \tl_put_right:Nx [95](#),
[4271](#), [4277](#), [4290](#), [6002](#), [6704](#), [6711](#),
[6718](#), [6727](#), [7407](#), [7441](#), [7454](#), [7466](#)
- \tl_remove_all:cn [97](#), [4418](#), [4969](#)
- \tl_remove_all:Nn [97](#), [4418](#), [4418](#), [4422](#), [4968](#)
- \tl_remove_all_in:cn [4964](#), [4969](#)
- \tl_remove_all_in:Nn [4964](#), [4968](#)
- \tl_remove_in:cn [4964](#), [4965](#)
- \tl_remove_in:Nn [4964](#), [4964](#)
- \tl_remove_once:cn [96](#), [4412](#), [4965](#)
- \tl_remove_once:Nn
... [96](#), [4412](#), [4412](#), [4416](#), [4964](#)

`\tl_replace_all:cnn` 96, [4364](#), 4961
`\tl_replace_all:Nnn` . . . 96, [4364](#), 4368,
4374, 4419, 4960, 5494, 7381, 7382
`\tl_replace_all_aux:`
. [4364](#), 4369, 4371, 4400, 4403
`\tl_replace_all_in:cnn` [4956](#), 4961
`\tl_replace_all_in:Nnn` [4956](#), 4960
`\tl_replace_aux:NNNnn`
. . . [4364](#), 4365, 4367, 4369, 4371, 4376
`\tl_replace_aux_ii:w` [4364](#), 4399, 4402, 4407
`\tl_replace_in:cnn` [4956](#), 4957
`\tl_replace_in:Nnn` [4956](#), 4956
`\tl_replace_once:cnn` 96, [4364](#), 4957
`\tl_replace_once:Nnn`
. 96, [4364](#), 4364, 4372, 4413, 4956
`\tl_replace_once_aux:`
. [4364](#), 4365, 4367, 4405
`\tl_replace_once_aux_end:w`
. [4364](#), 4408, 4410
`\tl_rescan:nn` 98, [4350](#), 4350
`\tl_rescan_aux:w`
. . . [4306](#), 4318, 4322, 4324, 4328, 4357
`\tl_reverse:c` 103, [4891](#)
`\tl_reverse:N` 103, [4891](#), 4891, 4893
`\tl_reverse:n` 103, [4879](#), 4879, 4890
`\tl_reverse:o` 103, [4879](#), 4892
`\tl_reverse:V` 103, [4879](#)
`\tl_reverse_group_preserve:nn` . . . [4879](#)
`\tl_reverse_items:n` 103, [4588](#), 4588
`\tl_reverse_items_aux:nN` [4588](#)
`\tl_reverse_items_aux:nw` [4589](#), 4590, 4593
`\tl_reverse_tokens:n` 108, [4860](#), 4860, 4877
`.tl_set:c` 171
`\tl_set:cf` 94, [4229](#)
`\tl_set:cn` 94, [4229](#), 7668, 7672
`\tl_set:co` 94, [4229](#)
`\tl_set:cx` 94, [4229](#), 7652
`.tl_set:N` 171
`\tl_set:Nc` [4952](#), 4953, 4954
`\tl_set:Nf` 94, [4229](#), 5680
`\tl_set:Nn` 94, 2237,
2890, 2911, [4229](#), 4229, 4241, 4243,
4307, 4326, 4334, 4484, 4485, 4557,
5064, 5073, 5087, 5090, 5113, 5116,
5128, 5143, 5174, 5242, 5334, 5488,
5565, 5579, 5589, 5592, 5602, 5776,
5784, 5939, 5952, 5955, 5961, 5962,
5968, 5972, 6063, 6129, 6140, 6676,
6898, 7085, 7086, 7380, 7426, 7464,
7506, 7543, 7610, 7636, 7704, 7818,
7820, 7828, 7833, 7877, 8432, 8884,
8919, 9002, 9027, 9053, 9156, 9174,
9287, 9829, 9926, 10127, 10320,
10638, 10972, 11030, 11042, 11557
`\tl_set:No` 94, [4229](#), 4231, 4892, 4955, 5661
`\tl_set:Nv` 94, [4229](#)
`\tl_set:Nv` 94, [4229](#)
`\tl_set:Nx`
. . . 94, [4229](#), 4233, 4242, 4344, 4365,
4369, 4636, 5015, 5054, 5157, 5290,
5348, 5435, 5440, 5463, 5480, 5499,
5543, 5804, 5865, 5979, 6111, 6577,
6599, 6659, 6902, 7431, 7447, 7504,
7530, 7537, 7540, 7850, 7851, 8050,
8069, 8216, 8229, 8240, 8332, 8379,
8404, 8467, 8987, 8990, 9036, 9284,
9648, 9657, 9680, 9699, 9842, 9939,
10140, 10334, 10417, 10450, 10677,
10793, 10802, 10930, 11374, 11399
`\tl_set_eq:cc` 93, [4221](#),
4224, 5009, 5537, 5915, 7714, 8519
`\tl_set_eq:cN`
. . . 93, [4221](#), 4222, 5008, 5536, 5914, 8517
`\tl_set_eq:Nc` 93, [4221](#),
4223, 5007, 5535, 5913, 7884, 8518
`\tl_set_eq:NN` 93, 4210, [4221](#), 4221, 5006,
5534, 5912, 6701, 6714, 8427, 8516
`\tl_set_rescan:cnn` 97, [4306](#)
`\tl_set_rescan:cno` 97, [4306](#)
`\tl_set_rescan:cnx` 97, [4333](#)
`\tl_set_rescan:Nnn`
. 97, [4306](#), 4306, 4329, 4330
`\tl_set_rescan:Nno` 97, [4306](#), 8217
`\tl_set_rescan:Nnx` . . . 97, [4333](#), 4333, 4348
`\tl_set_rescan_aux:NNnn`
. [4306](#), 4307, 4309, 4310
`\tl_set_rescan_aux:NNnx`
. [4333](#), 4334, 4336, 4337
`.tl_set_x:c` 172
`.tl_set_x:N` 172
`\tl_show:c` 107, [4767](#), 8525
`\tl_show:N` . . . 107, [4767](#), 4767, 4768, 8524
`\tl_show:n` 107, [4769](#), 4769,
5282, 5796, 6103, 6113, 6573, 6595
`\tl_tail:f` 105, [4641](#)
`\tl_tail:n` 105, [4641](#), 4645, 4649
`\tl_tail:V` 105, [4641](#)
`\tl_tail:v` 105, [4641](#)
`\tl_tail:w` . . . 105, [4641](#), 4642, 8232, 8243
`\tl_tail_aux:w` 4646, 4647

\tl_tmp:w	4384, 4403, 4408, 4504, 4505, 4596, 4634
\tl_to_lowercase:n	98, 2267, 2281, 2683, 2723, 2816, 3083, 4361, 4361, 6919, 7325, 7373
\tl_to_str:c	102, 4564
\tl_to_str:N	102, 4564, 4564, 4565, 6668
\tl_to_str:n	102, 3045, 3963, 4381, 4448, 4461, 4563, 4563, 4653, 4662, 5921, 5990, 6011, 6031, 6735, 7504, 7537, 7850, 7928, 7944, 8509
\tl_to_uppercase:n	98, 4361, 4362
\tl_trim_spaces:c	104, 4596
\tl_trim_spaces:N	104, 4596, 4635, 4639
\tl_trim_spaces:n	104, 4596, 4598, 4636, 4638, 5506, 5678
\tl_trim_spaces_aux_i:w	4596, 4601, 4612, 4615
\tl_trim_spaces_aux_ii:w	4606, 4620
\tl_trim_spaces_aux_ii:w\tl_trim_spaces_aux_iii:w	4596
\tl_trim_spaces_aux_iii:w	4607, 4622, 4625, 4629
\tl_trim_spaces_aux_iv:w	4596, 4609, 4631
\tl_use:c	102, 4566, 4567, 5623
\tl_use:N	102, 4566, 4566, 5622
\token_get_arg_spec:N	64, 3043, 3056
\token_get_prefix_arg_replacement_aux:wN	3043, 3044, 3051, 3060, 3069
\token_get_prefix_spec:N	65, 3043, 3047
\token_get_replacement_spec:N	3043, 3065
\token_get_replacement_text:N	65
\token_if_active:N	2657
\token_if_active:NF	3187
\token_if_active:NT	3186
\token_if_active:NTF	59, 2657, 3188
\token_if_active_char:NF	3187
\token_if_active_char:NT	3186
\token_if_active_char:NTF	3173, 3188
\token_if_active_char_p:N	3173, 3185
\token_if_active_p:N	59, 2657, 3185
\token_if_alignment:N	2619
\token_if_alignment:NF	3175
\token_if_alignment:NT	3174
\token_if_alignment:NTF	58, 2619, 3176
\token_if_alignment_p:N	58, 2619, 3173
\token_if_alignment_tab:NF	3175
\token_if_alignment_tab:NT	3174
\token_if_alignment_tab:NTF	3173, 3176
\token_if_alignment_tab_p:N	3173, 3173
\token_if_chardef:N	2726
\token_if_chardef:NTF	60, 2714
\token_if_chardef_aux:w	2728, 2731
\token_if_chardef_p:N	60, 2714
\token_if_chardef_p_aux:w	2714
\token_if_cs:N	2700
\token_if_cs:NTF	60, 2700
\token_if_cs_p:N	60, 2700
\token_if_dim_register:N	2762
\token_if_dim_register:NTF	60, 2714
\token_if_dim_register_aux:w	2767, 2771
\token_if_dim_register_p:N	60, 2714
\token_if_dim_register_p_aux:w	2714
\token_if_eq_catcode:NN	2667
\token_if_eq_catcode:NNTF	59, 2667
\token_if_eq_catcode_p:NN	59, 2667, 2947, 2948, 3097, 3098
\token_if_eq_charcode:NN	2672
\token_if_eq_charcode:NNTF	59, 2672
\token_if_eq_charcode_p:NN	59, 2672
\token_if_eq_meaning:NN	2662
\token_if_eq_meaning:NNT	2276
\token_if_eq_meaning:NNTF	59, 2291, 2662, 2968
\token_if_eq_meaning_p:NN	59, 2662, 2949, 3099
\token_if_expandable:N	2705
\token_if_expandable:NTF	60, 2705
\token_if_expandable_p:N	60, 2705
\token_if_group_begin:N	2604
\token_if_group_begin:NTF	57, 2604
\token_if_group_begin_p:N	57, 2604
\token_if_group_end:N	2609
\token_if_group_end:NTF	57, 2609
\token_if_group_end_p:N	57, 2609
\token_if_int_register:N	2740
\token_if_int_register:NTF	61, 2714
\token_if_int_register_aux:w	2745, 2749
\token_if_int_register_p:N	61, 2714
\token_if_int_register_p_aux:w	2714
\token_if_letter:N	2647
\token_if_letter:NTF	59, 2647
\token_if_letter_p:N	59, 2647
\token_if_long_macro:N	2793
\token_if_long_macro:NTF	60, 2714
\token_if_long_macro_aux:w	2795, 2798
\token_if_long_macro_p:N	60, 2714
\token_if_long_macro_p_aux:w	2714
\token_if_macro:N	2686

\token_if_macro:NTF	\token_if_primitive_p:N
.... 59, 2677, 2820, 3049, 3058, 3067	61, 2810
\token_if_macro_p:N	\token_if_protected_long_macro:N .
59, 2677	2800
\token_if_macro_p_aux:w 2677, 2688, 2691	\token_if_protected_long_macro:NTF .
\token_if_math_shift:N	60, 2714
3179	\token_if_protected_long_macro_aux:w
\token_if_math_shift:NT	2803, 2806
3178	\token_if_protected_long_macro_p:N .
\token_if_math_shift:NTF 3173, 3180	60, 2714
\token_if_math_shift_p:N 3173, 3177	\token_if_protected_long_macro_p_aux:w
\token_if_math_subscript:N	2714
2637	\token_if_protected_macro:N
\token_if_math_subscript:NTF .. 58, 2637	2784
\token_if_math_subscript_p:N .. 58, 2637	\token_if_protected_macro:NTF . 60, 2714
\token_if_math_superscript:N 2632	\token_if_protected_macro_aux:w
\token_if_math_superscript:NTF 58, 2632	2787, 2790
\token_if_math_superscript_p:N 58, 2632	\token_if_protected_macro_p:N . 60, 2714
\token_if_math_toggle:N	\token_if_protected_macro_p_aux:w 2714
2614	\token_if_skip_register:N
\token_if_math_toggle:NF	2751
3179	\token_if_skip_register:NTF ... 61, 2714
\token_if_math_toggle:NT	\token_if_skip_register_aux:w 2756, 2760
3178	\token_if_skip_register_p:N ... 61, 2714
\token_if_math_toggle:NTF 58, 2614, 3180	\token_if_skip_register_p_aux:w .. 2714
\token_if_math_toggle_p:N 58, 2614, 3177	\token_if_space:N
\token_if_mathchardef:N	2642
2733	\token_if_space:NTF
\token_if_mathchardef:NTF 60, 2714	58, 2642
\token_if_mathchardef_aux:w . 2735, 2738	\token_if_space_p:N
\token_if_mathchardef_p:N 60, 2714	58, 2642
\token_if_mathchardef_p_aux:w ... 2714	\token_if_toks_register:N
\token_if_other:N	2773
2652	\token_if_toks_register:NTF ... 61, 2714
\token_if_other:NF	\token_if_toks_register_aux:w 2778, 2782
3183	\token_if_toks_register_p:N ... 61, 2714
\token_if_other:NT	\token_if_toks_register_p_aux:w .. 2714
3182	\token_new:Nn
\token_if_other:NTF 59, 2652, 3184	56, 2584,
\token_if_other_char:NF	2584, 2589, 2591–2593, 2595–2598
3183	\token_to_meaning:N 57, 805, 805,
\token_if_other_char:NT	1197, 1207, 1220, 1828, 2689, 2729,
3182	2736, 2746, 2757, 2768, 2779, 2788,
\token_if_other_char:NTF 3173, 3184	2796, 2804, 2824, 3052, 3061, 3070
\token_if_other_char_p:N 3173, 3181	\token_to_str:c
\token_if_other_p:N 59, 2652, 3181	57, 821, 822
\token_if_parameter:N	\token_to_str:N
2626 5, 57, 805, 806, 822, 1065,
\token_if_parameter:NTF	1068, 1197, 1207, 1209, 1220, 1354,
58, 2624	1444, 1798, 2287, 4739, 5192, 5281,
\token_if_parameter_p:N	5287, 5795, 5801, 6102, 6108, 11527
58, 2624	\toks
\token_if_primitive:N	659
2818	\toksdef
\token_if_primitive:NTF	360
61, 2810	\tolerance
\token_if_primitive_aux:NNw	570
2810, 2823, 2827	\topmark
\token_if_primitive_aux_loop:N	451
2810, 2830, 2843, 2849	\topmarks
\token_if_primitive_aux_nullfont:N .	677
2810, 2831, 2835	\topskip
\token_if_primitive_aux_space:w	581
2810, 2829, 2834	\tracingassigns
\token_if_primitive_aux_undefined:N	687
2810, 2855, 2861	\tracingcommands
\token_if_primitive_auxii:Nw	426
2810, 2846, 2852	\tracinggroups
	694

<code>\tracingifs</code>	690	<code>\use_4_parameter:</code>	1267
<code>\tracinglostchars</code>	427	<code>\use_5_parameter:</code>	1267
<code>\tracingmacros</code>	428	<code>\use_6_parameter:</code>	1267
<code>\tracingnesting</code>	689	<code>\use_7_parameter:</code>	1267
<code>\tracingonline</code>	429	<code>\use_8_parameter:</code>	1267
<code>\tracingoutput</code>	430	<code>\use_9_parameter:</code>	1267
<code>\tracingpages</code>	431	<code>\use_i:nn</code>	18 , 866 , 866 ,
<code>\tracingparagraphs</code>	432		896 , 1086 , 1115 , 1143 , 1305 , 1453 ,
<code>\tracingrestores</code>	433		1467 , 1475 , 8321 , 8367 , 8392 , 8960 ,
<code>\tracingscantokens</code>	688		9279 , 9636 , 9669 , 10452 , 10780 , 10919
<code>\tracingstats</code>	434	<code>\use_i:nnn</code>	18 , 868 , 868 , 1097 , 3052 , 10419
U			
<code>\U</code>	2721	<code>\use_i:nnnn</code>	19 , 868 , 872
<code>\uccode</code>	669	<code>\use_i_after_else:nw</code>	20 , 882 , 883
<code>\uchyph</code>	567	<code>\use_i_after_fi:nw</code>	20 , 882 , 882 , 1341
<code>\underline</code>	503	<code>\use_i_after_or:nw</code>	20 , 882 , 884
<code>\unexpanded</code>	179 , 183 , 682	<code>\use_i_after_orelse:nw</code>	20 ,
<code>\unhbox</code>	608		882 , 885 , 1321 , 1323 , 1325 , 1327 ,
<code>\unhcopy</code>	609		1329 , 1331 , 1333 , 1335 , 1337 , 1339
<code>\unkern</code>	533	<code>\use_i_delimit_by_q_nil:nw</code> .	20 , 879 , 879
<code>\unless</code>	673	<code>\use_i_delimit_by_q_recursion_stop:nw</code> 20 , 51 , 879 ,
<code>\unpenalty</code>	644		881 , 2089 , 2399 , 2428 , 5790 , 5850 , 6096
<code>\unskip</code>	531	<code>\use_i_delimit_by_q_stop:nw</code>	20 , 879 , 880
<code>\unvbox</code>	610	<code>\use_i_ii:nnn</code>	19 , 868 , 871 , 1565
<code>\unvcopy</code>	611	<code>\use_ii:nn</code>	18 , 866 , 867 ,
<code>\uppercase</code>	641		898 , 1088 , 1117 , 1145 , 1307 , 1450 ,
<code>\use:c</code>	16 , 852 ,		1456 , 1464 , 1478 , 5676 , 5928 , 7395
	852 , 984 , 1783 , 1927 , 1937 , 2009 ,	<code>\use_ii:nnn</code> .	18 , 868 , 869 , 1099 , 3061 , 7455
	2010 , 3353 , 3641 , 3651 , 3794 , 3803 ,	<code>\use_ii:nnnn</code>	19 , 868 , 873
	3805 , 3807 , 3808 , 3812 , 3966 , 6991 ,	<code>\use_iii:nnn</code>	18 , 868 , 870 , 3070
	7002 , 7015 , 7018 , 7024 , 7035 , 7043 ,	<code>\use_iii:nnnn</code>	19 , 868 , 874
	7049 , 7071 , 7132 , 7154 , 7159 , 7167 ,	<code>\use_iv:nnnn</code>	19 , 868 , 875
	7190 , 7212 , 7551 , 7558 , 7720 , 7921 ,	<code>\use_none:n</code>	19 ,
	8221 , 8251 , 8254 , 8271 , 8274 , 8275 ,		886 , 886 , 1000 , 1029 , 1068 , 1070 ,
	8278 , 8281 , 8565 , 8627 , 8683 , 8733 ,		1344 , 1448 , 1452 , 1454 , 1462 , 1466 ,
	9875 , 9962 , 10173 , 10360 , 10696 ,		1474 , 1476 , 1851 , 2401 , 2430 , 2858 ,
	11223 , 11286 , 11301 , 11303 , 11394		3572 , 3687 , 3691 , 3696 , 4425 , 4592 ,
<code>\use:n</code>	18 ,		4632 , 4718 , 4736 , 4765 , 4807 , 4992 ,
	862 , 862 , 1000 , 1029 , 1449 , 1451 ,		5142 , 5171 , 5204 , 5399 , 5426 , 5427 ,
	1455 , 1463 , 1465 , 1473 , 1477 , 4715 ,		5663 , 5866 , 5877 , 6633 , 6635 , 6774 –
	4732 , 5188 , 5405 , 5936 , 6160 , 7337		6777 , 7392 , 7429 , 8335 , 8382 , 8407 ,
<code>\use:nn</code>	18 , 862 , 863 , 1540 , 3043 , 3961		8456 , 8498 , 8910 , 8946 , 9019 , 9045 ,
<code>\use:nnn</code>	18 , 862 , 864		9099 , 9220 , 9363 , 9683 , 9702 , 9851 ,
<code>\use:nnnn</code>	18 , 862 , 865		9906 , 9948 , 9992 , 10149 , 10273 ,
<code>\use:x</code>	19 , 853 , 853 , 6665	<code>\use_none:nn</code>	10343 , 10565 , 10682 , 10725 , 11109
<code>\use_0_parameter:</code>	1267		
<code>\use_1_parameter:</code>	1267		19 , 886 , 887 , 1807 , 4803 , 5069 , 11173
<code>\use_2_parameter:</code>	1267	<code>\use_none:nnn</code> ...	19 , 886 , 888 , 6008 , 7448
<code>\use_3_parameter:</code>	1267	<code>\use_none:nnnn</code>	19 , 886 , 889 , 8292
		<code>\use_none:nnnnn</code>	19 , 886 , 890

<code>\use_none:nnnnnn</code>	19, 886, 891	<code>\vbox_top:n</code>	149, 6333, 6334
<code>\use_none:nnnnnnn</code>	19, 886, 892	<code>\vbox_unpack:c</code>	151, 6362
<code>\use_none:nnnnnnnn</code>	19, 886, 893	<code>\vbox_unpack:N</code>	151, 6362, 6362, 6364
<code>\use_none:nnnnnnnnn</code>	19, 886, 894, 1311	<code>\vbox_unpack_clear:c</code>	151, 6362
<code>\use_none_delimit_by_q_nil:w</code>	20, 876, 876	<code>\vbox_unpack_clear:N</code>	151, 6362, 6363, 6365
<code>\use_none_delimit_by_q_recursion_stop:w</code>	20, 51, 876, 878, 982, 1050, 1779, 2393, 2415, 4562, 5789, 6095	<code>\vcenter</code>	465
<code>\use_none_delimit_by_q_stop:w</code>	20, 876, 877, 2306, 2310, 4388, 6038, 6043, 6742	<code>\vfil</code>	526
<code>\usepackage</code>	223	<code>\vfill</code>	528
V			
<code>\vadjust</code>	544	<code>\vfilneg</code>	527
<code>\valign</code>	379	<code>\vfuzz</code>	621
<code>.value_forbidden:</code>	172	<code>\voffset</code>	596
<code>.value_required:</code>	172	<code>\voidb@x</code>	6289
<code>\vbadness</code>	619	<code>\vrule</code>	535
<code>\vbox</code>	614	<code>\vsize</code>	578
<code>\vbox:n</code>	149, 6333, 6333	<code>\vskip</code>	529
<code>\vbox_gset:cn</code>	149, 6338	<code>\vsplit</code>	607
<code>\vbox_gset:Nn</code>	149, 6338, 6339, 6341	<code>\vss</code>	530
<code>\vbox_gset_inline_begin:c</code>	150, 6354	<code>\vtop</code>	615
<code>\vbox_gset_inline_begin:N</code>	150, 6354, 6356, 6359	W	
<code>\vbox_gset_inline_end:</code>	150, 6354, 6361	<code>\wd</code>	662
<code>\vbox_gset_to_ht:cnn</code>	150, 6348	<code>\widowpenalties</code>	722
<code>\vbox_gset_to_ht:Nnn</code>	150, 6348, 6350, 6353	<code>\widowpenalty</code>	549
<code>\vbox_gset_top:cn</code>	150, 6342	<code>\write</code>	412
<code>\vbox_gset_top:Nn</code>	150, 6342, 6344, 6347	X	
<code>\vbox_set:cn</code>	149, 6338	<code>\X</code>	2717, 2721
<code>\vbox_set:Nn</code>	149, 6338, 6338–6340	<code>\xdef</code>	353
<code>\vbox_set_inline_begin:c</code>	150, 6354	<code>\xetex_if_engine:F</code>	1455, 1466
<code>\vbox_set_inline_begin:N</code>	150, 6354, 6354, 6357, 6358	<code>\xetex_if_engine:T</code>	1454, 1465
<code>\vbox_set_inline_end:</code>	150, 6354, 6360	<code>\xetex_if_engine:TF</code>	4, 24, 1448, 1456, 1467
<code>\vbox_set_split_to_ht:Nnn</code>	151, 6366, 6366	<code>\xetex_if_engine_p:</code>	4
<code>\vbox_set_to_ht:cnn</code>	150, 6348	<code>\xetex_XeTeXversion:D</code>	754, 1460
<code>\vbox_set_to_ht:Nnn</code>	150, 6348, 6348, 6351, 6352	<code>\XeTeXversion</code>	754
<code>\vbox_set_top:cn</code>	150, 6342	<code>\xleaders</code>	538
<code>\vbox_set_top:Nn</code>	150, 6342, 6342, 6345, 6346	<code>\xspaceskip</code>	572
<code>\vbox_to_ht:nn</code>	149, 6335, 6335	Y	
<code>\vbox_to_zero:n</code>	149, 6335, 6337	<code>\Y</code>	2718, 2721
		<code>\year</code>	653
		Z	
		<code>\Z</code>	1813, 1821, 2719, 2721
		<code>\z@</code>	4049