

# The L<sup>A</sup>T<sub>E</sub>X3 Sources

The L<sup>A</sup>T<sub>E</sub>X3 Project\*

December 18, 2010

## Abstract

This is the reference documentation for the `expl3` programming environment. The `expl3` modules set up an experimental naming scheme for L<sup>A</sup>T<sub>E</sub>X commands, which allow the L<sup>A</sup>T<sub>E</sub>X programmer to systematically name functions and variables, and specify the argument types of functions.

The T<sub>E</sub>X and  $\varepsilon$ -T<sub>E</sub>X 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<sup>A</sup>T<sub>E</sub>X3 programming language.

At present, the `expl3` modules are designed to be loaded on top of L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$ . In time, a L<sup>A</sup>T<sub>E</sub>X3 format will be produced based on this code. This allows the code to be used in L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$  packages *now* while a stand-alone L<sup>A</sup>T<sub>E</sub>X3 is developed.

**While `expl3` is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`.**

**New modules will be added to the distributed version of `expl3` as they reach maturity.**

---

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

# Contents

<b>I</b>	<b>Introduction to <code>expl3</code> and this document</b>	<b>1</b>
<b>1</b>	<b>Naming functions and variables</b>	<b>1</b>
1.0.1	Terminological inexactitude . . . . .	3
<b>2</b>	<b>Documentation conventions</b>	<b>3</b>
<b>II</b>	<b>The <code>l3names</code> package: A systematic naming scheme for <code>TeX</code></b>	<b>5</b>
<b>3</b>	<b>Setting up the <code>LaTeX3</code> programming language</b>	<b>5</b>
<b>4</b>	<b>Using the modules</b>	<b>5</b>
<b>III</b>	<b>The <code>l3basics</code> package: Basic Definitions</b>	<b>6</b>
<b>5</b>	<b>Predicates and conditionals</b>	<b>7</b>
5.1	Primitive conditionals . . . . .	8
5.2	Non-primitive conditionals . . . . .	10
<b>6</b>	<b>Control sequences</b>	<b>12</b>
<b>7</b>	<b>Selecting and discarding tokens from the input stream</b>	<b>12</b>
7.1	Extending the interface . . . . .	14
7.2	Selecting tokens from delimited arguments . . . . .	14
<b>8</b>	<b>That which belongs in other modules but needs to be defined earlier</b>	<b>15</b>
<b>9</b>	<b>Defining functions</b>	<b>16</b>
9.1	Defining new functions using primitive parameter text . . . . .	17
9.2	Defining new functions using the signature . . . . .	18
9.3	Defining functions using primitive parameter text . . . . .	19
9.4	Defining functions using the signature (no checks) . . . . .	20

9.5	Undefining functions . . . . .	22
9.6	Copying function definitions . . . . .	22
9.7	Internal functions . . . . .	23
<b>10</b>	<b>The innards of a function</b>	<b>23</b>
<b>11</b>	<b>Grouping and scanning</b>	<b>24</b>
<b>12</b>	<b>Checking the engine</b>	<b>25</b>
<b>IV</b>	<b>The l3expan package: Controlling Expansion of Function Arguments</b>	<b>25</b>
<b>13</b>	<b>Brief overview</b>	<b>25</b>
<b>14</b>	<b>Defining new variants</b>	<b>26</b>
14.1	Methods for defining variants . . . . .	26
<b>15</b>	<b>Introducing the variants</b>	<b>27</b>
<b>16</b>	<b>Manipulating the first argument</b>	<b>28</b>
<b>17</b>	<b>Manipulating two arguments</b>	<b>29</b>
<b>18</b>	<b>Manipulating three arguments</b>	<b>30</b>
<b>19</b>	<b>Preventing expansion</b>	<b>31</b>
<b>20</b>	<b>Unbraced expansion</b>	<b>32</b>
<b>V</b>	<b>The l3prg package: Program control structures</b>	<b>32</b>
<b>21</b>	<b>Conditionals and logical operations</b>	<b>32</b>
<b>22</b>	<b>Defining a set of conditional functions</b>	<b>33</b>
<b>23</b>	<b>The boolean data type</b>	<b>34</b>

<b>24 Boolean expressions</b>	<b>36</b>
<b>25 Case switches</b>	<b>37</b>
<b>26 Generic loops</b>	<b>38</b>
<b>27 Choosing modes</b>	<b>38</b>
<b>28 Alignment safe grouping and scanning</b>	<b>39</b>
<b>29 Producing <math>n</math> copies</b>	<b>39</b>
<b>30 Sorting</b>	<b>40</b>
30.1 Variable type and scope . . . . .	41
30.2 Mapping to variables . . . . .	41
<b>VI The l3quark package: “Quarks”</b>	<b>42</b>
<b>31 Functions</b>	<b>42</b>
<b>32 Recursion</b>	<b>43</b>
<b>33 Constants</b>	<b>44</b>
<b>VII The l3token package: A token of my appreciation...</b>	<b>45</b>
<b>34 Character tokens</b>	<b>45</b>
<b>35 Generic tokens</b>	<b>48</b>
35.1 Useless code: because we can! . . . . .	52
<b>36 Peeking ahead at the next token</b>	<b>52</b>
<b>VIII The l3int package: Integers/counters</b>	<b>54</b>

<b>37 Integer values</b>	<b>54</b>
37.1 Integer expressions . . . . .	54
37.2 Integer variables . . . . .	55
37.3 Comparing integer expressions . . . . .	57
37.4 Formatting integers . . . . .	60
37.5 Converting from other formats . . . . .	61
37.6 Low-level conversion functions . . . . .	62
<b>38 Variables and constants</b>	<b>63</b>
38.1 Internal functions . . . . .	64
 <b>IX The l3skip package: Dimension and skip registers</b>	 <b>66</b>
<b>39 Skip registers</b>	<b>66</b>
39.1 Functions . . . . .	66
39.2 Formatting a skip register value . . . . .	68
39.3 Variable and constants . . . . .	68
<b>40 Dim registers</b>	<b>69</b>
40.1 Functions . . . . .	69
40.2 Variable and constants . . . . .	72
<b>41 Muskip</b>	<b>73</b>
 <b>X The l3tl package: Token Lists</b>	 <b>73</b>
<b>42 Functions</b>	<b>74</b>
<b>43 Predicates and conditionals</b>	<b>78</b>
<b>44 Working with the contents of token lists</b>	<b>79</b>
<b>45 Variables and constants</b>	<b>80</b>
<b>46 Searching for and replacing tokens</b>	<b>81</b>

47 Heads or tails?	82
<b>XI The l3toks package: Token Registers</b>	<b>84</b>
48 Allocation and use	84
49 Adding to the contents of token registers	86
50 Predicates and conditionals	87
51 Variable and constants	88
<b>XII The l3seq package: Sequences</b>	<b>88</b>
52 Functions for creating/initialising sequences	89
53 Adding data to sequences	90
54 Working with sequences	91
55 Predicates and conditionals	92
56 Internal functions	93
57 Functions for ‘Sequence Stacks’	93
<b>XIII The l3clist package: Comma separated lists</b>	<b>94</b>
58 Functions for creating/initialising comma-lists	95
59 Putting data in	96
60 Getting data out	97
61 Mapping functions	97
62 Predicates and conditionals	99

<b>63 Higher level functions</b>	<b>99</b>
<b>64 Functions for ‘comma-list stacks’</b>	<b>100</b>
<b>65 Internal functions</b>	<b>101</b>
 <b>XIV The <code>l3prop</code> package: Property Lists</b>	 <b>102</b>
<b>66 Functions</b>	<b>102</b>
<b>67 Predicates and conditionals</b>	<b>104</b>
<b>68 Internal functions</b>	<b>105</b>
 <b>XV The <code>l3io</code> package: Low-level file i/o</b>	 <b>106</b>
<b>69 Opening and closing streams</b>	<b>106</b>
69.1 Writing to files . . . . .	107
69.2 Reading from files . . . . .	109
<b>70 Internal functions</b>	<b>109</b>
<b>71 Variables and constants</b>	<b>110</b>
 <b>XVI The <code>l3msg</code> package: Communicating with the user</b>	 <b>110</b>
<b>72 Creating new messages</b>	<b>111</b>
<b>73 Message classes</b>	<b>111</b>
<b>74 Redirecting messages</b>	<b>113</b>
<b>75 Support functions for output</b>	<b>114</b>
<b>76 Low-level functions</b>	<b>114</b>
<b>77 Kernel-specific functions</b>	<b>115</b>

78 Variables and constants	116
<b>XVII The l3box package: Boxes</b>	<b>117</b>
79 Generic functions	118
80 Horizontal mode	121
81 Vertical mode	122
<b>XVIII The l3xref package: Cross references</b>	<b>124</b>
<b>XIX The l3keyval package: Key-value parsing</b>	<b>124</b>
82 Features of l3keyval	125
83 Functions for keyval processing	125
84 Internal functions	126
85 Variables and constants	127
<b>XX The l3keys package: Key-value support</b>	<b>127</b>
86 Creating keys	129
87 Sub-dividing keys	132
87.1 Multiple choices . . . . .	133
88 Setting keys	134
88.1 Examining keys: internal representation . . . . .	135
89 Internal functions	135
90 Variables and constants	137

<b>XXI</b>	<b>The <code>l3file</code> package: File Loading</b>	<b>138</b>
<b>91</b>	<b>Loading files</b>	<b>138</b>
<b>XXII</b>	<b>The <code>l3fp</code> package: Floating point arithmetic</b>	<b>139</b>
<b>92</b>	<b>Floating point numbers</b>	<b>139</b>
92.1	Constants . . . . .	140
92.2	Floating-point variables . . . . .	140
92.3	Conversion to other formats . . . . .	142
92.4	Rounding floating point values . . . . .	143
92.5	Tests on floating-point values . . . . .	144
92.6	Unary operations . . . . .	144
92.7	Arithmetic operations . . . . .	145
92.8	Power operations . . . . .	146
92.9	Exponential and logarithm functions . . . . .	146
92.10	Trigonometric functions . . . . .	147
92.11	Notes on the floating point unit . . . . .	148
<b>XXIII</b>	<b>The <code>l3luatex</code> package: Lua<sub>TEX</sub>-specific functions</b>	<b>148</b>
<b>93</b>	<b>Breaking out to Lua</b>	<b>148</b>
<b>94</b>	<b>Category code tables</b>	<b>149</b>
<b>XXIV</b>	<b>Implementation</b>	<b>150</b>
<b>95</b>	<b><code>l3names</code> implementation</b>	<b>150</b>
95.1	Internal functions . . . . .	150
95.2	Bootstrap code . . . . .	151
95.3	Requirements . . . . .	152
95.4	Catcode assignments . . . . .	153

95.5	Setting up primitive names	154
95.6	Reassignment of primitives	155
95.7	<code>expl3</code> code switches	165
95.8	Package loading	167
95.9	Finishing up	171
95.10	Showing memory usage	173
<b>96</b>	<b><code>l3basics</code> implementation</b>	<b>173</b>
96.1	Renaming some $\text{\TeX}$ primitives (again)	173
96.2	Defining functions	175
96.3	Selecting tokens	176
96.4	Gobbling tokens from input	177
96.5	Expansion control from <code>l3expan</code>	178
96.6	Conditional processing and definitions	178
96.7	Dissecting a control sequence	183
96.8	Exist or free	185
96.9	Defining and checking (new) functions	186
96.10	More new definitions	189
96.11	Copying definitions	191
96.12	Undefining functions	192
96.13	Diagnostic wrapper functions	192
96.14	Engine specific definitions	193
96.15	Scratch functions	193
96.16	Defining functions from a given number of arguments	193
96.17	Using the signature to define functions	195
<b>97</b>	<b><code>l3expan</code> implementation</b>	<b>198</b>
97.1	Internal functions and variables	198
97.2	Module code	199
97.3	General expansion	199
97.4	Hand-tuned definitions	203
97.5	Definitions with the ‘general’ technique	204

97.6 Preventing expansion . . . . .	205
97.7 Defining function variants . . . . .	205
97.8 Last-unbraced versions . . . . .	208
97.9 Items held from earlier . . . . .	209
<b>98 l3prg implementation</b>	<b>210</b>
98.1 Variables . . . . .	210
98.2 Module code . . . . .	210
98.3 Choosing modes . . . . .	210
98.4 Producing $n$ copies . . . . .	212
98.5 Booleans . . . . .	215
98.6 Parsing boolean expressions . . . . .	217
98.7 Case switch . . . . .	223
98.8 Sorting . . . . .	225
98.9 Variable type and scope . . . . .	227
98.10 Mapping to variables . . . . .	228
<b>99 l3quark implementation</b>	<b>230</b>
<b>100 l3token implementation</b>	<b>233</b>
100.1 Documentation of internal functions . . . . .	233
100.2 Module code . . . . .	234
100.3 Character tokens . . . . .	234
100.4 Generic tokens . . . . .	236
100.5 Peeking ahead at the next token . . . . .	244
<b>101 l3int implementation</b>	<b>250</b>
101.1 Internal functions and variables . . . . .	250
101.2 Module loading and primitives definitions . . . . .	251
101.3 Allocation and setting . . . . .	251
101.4 Scanning and conversion . . . . .	259
101.5 Defining constants . . . . .	269
101.6 Backwards compatibility . . . . .	271

<b>1023skip implementation</b>	<b>272</b>
102.1Skip registers . . . . .	272
102.2Dimen registers . . . . .	276
102.3Muskip . . . . .	280
<b>1033tl implementation</b>	<b>281</b>
103.1Functions . . . . .	281
103.2Variables and constants . . . . .	286
103.3Predicates and conditionals . . . . .	287
103.4Working with the contents of token lists . . . . .	290
103.5Checking for and replacing tokens . . . . .	294
103.6Heads or tails? . . . . .	297
<b>1043toks implementation</b>	<b>301</b>
104.1Allocation and use . . . . .	301
104.2Adding to token registers' contents . . . . .	303
104.3Predicates and conditionals . . . . .	305
104.4Variables and constants . . . . .	305
<b>1053seq implementation</b>	<b>306</b>
105.1Allocating and initialisation . . . . .	306
105.2Predicates and conditionals . . . . .	308
105.3Getting data out . . . . .	308
105.4Putting data in . . . . .	309
105.5Mapping . . . . .	311
105.6Manipulation . . . . .	311
105.7Sequence stacks . . . . .	312
<b>1063clist implementation</b>	<b>313</b>
106.1Allocation and initialisation . . . . .	313
106.2Predicates and conditionals . . . . .	314
106.3Retrieving data . . . . .	315
106.4Storing data . . . . .	316
106.5Mapping . . . . .	317
106.6Higher level functions . . . . .	318
106.7Stack operations . . . . .	319

<b>107</b>	<b>prop implementation</b>	<b>320</b>
107.1	Functions . . . . .	320
107.2	Predicates and conditionals . . . . .	324
107.3	Mapping functions . . . . .	324
<b>108</b>	<b>io implementation</b>	<b>326</b>
108.1	Variables and constants . . . . .	326
108.2	Stream management . . . . .	327
108.3	Immediate writing . . . . .	332
108.4	Deferred writing . . . . .	333
<b>109</b>	<b>Special characters for writing</b>	<b>333</b>
109.1	Reading input . . . . .	334
<b>110</b>	<b>msg implementation</b>	<b>334</b>
110.1	Variables and constants . . . . .	335
110.2	Output helper functions . . . . .	336
110.3	Generic functions . . . . .	337
110.4	General functions . . . . .	339
110.5	Redirection functions . . . . .	343
110.6	Kernel-specific functions . . . . .	343
<b>111</b>	<b>box implementation</b>	<b>346</b>
111.1	Generic boxes . . . . .	346
111.2	Vertical boxes . . . . .	349
111.3	Horizontal boxes . . . . .	350
<b>112</b>	<b>xref implementation</b>	<b>352</b>
112.1	Internal functions and variables . . . . .	352
112.2	Module code . . . . .	352
<b>113</b>	<b>xref test file</b>	<b>355</b>

<b>1143</b>	<b>keyval implementation</b>	<b>356</b>
114.1	Module code . . . . .	357
114.1.1	Variables and constants . . . . .	365
114.1.2	Internal functions . . . . .	366
114.1.3	Properties . . . . .	373
114.1.4	Messages . . . . .	376
<b>1153</b>	<b>file implementation</b>	<b>377</b>
<b>116</b>	<b>Implementation</b>	<b>381</b>
116.1	Constants . . . . .	381
116.2	Variables . . . . .	382
116.3	Parsing numbers . . . . .	384
116.4	Internal utilities . . . . .	388
116.5	Operations for <b>fp</b> variables . . . . .	389
116.6	Transferring to other types . . . . .	394
116.7	Rounding numbers . . . . .	400
116.8	Unary functions . . . . .	403
116.9	Basic arithmetic . . . . .	405
116.10	Arithmetic for internal use . . . . .	414
116.11	Trigonometric functions . . . . .	420
116.12	Exponent and logarithm functions . . . . .	432
116.13	Tests for special values . . . . .	455
116.14	Floating-point conditionals . . . . .	456
116.15	Messages . . . . .	460
<b>117</b>	<b>Implementation</b>	<b>461</b>
117.1	Category code tables . . . . .	462

## Part I

# Introduction to expl3 and this document

This document is intended to act as a comprehensive reference manual for the `expl3` language. A general guide to the `LATEX3` programming language is found in [expl3.pdf](#).

## 1 Naming functions and variables

`LATEX3` does not use `@` as a “letter” for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all `expl3` function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

- D** The **D** specifier means *do not use*. All of the `TEX` primitives are initially `\let` to a **D** name, and some are then given a second name. Only the kernel team should use anything with a **D** specifier!
- N and n** These mean *no manipulation*, of a single token for **N** and of a set of tokens given in braces for **n**. Both pass the argument though exactly as given. Usually, if you use a single token for an **n** argument, all will be well.
- c** This means *csname*, and indicates that the argument will be turned into a *csname* before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The **V** and **v** specifiers are used to get the content of a variable without needing to worry about the underlying `TEX` structure containing the data. A **V** argument will be a single token (similar to **N**), for example `\foo:V \MyVariable`; on the other hand, using **v** a *csname* is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

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

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

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

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

Each variable name is then build up in a similar way to that of a function, typically starting with the module<sup>1</sup> name and then a descriptive part. Variables end with a short identifier to show the variable type:

**bool** Either true or false.

**box** Box register.

**clist** Comma separated list.

**dim** ‘Rigid’ lengths.

**int** Integer-valued count register.

---

<sup>1</sup>The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

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

**prop** Property list.

**skip** ‘Rubber’ lengths.

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

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

**t1** Token list variables: placeholder for a token list.

**toks** Token register.

### 1.0.1 Terminological inexactitude

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

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

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

## 2 Documentation conventions

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

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

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

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>	<code>\seq_new:N &lt;sequence&gt;</code>
--	--

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  $\text{T}_{\text{E}}\text{X}$  terms, inside an `\edef`). These fully expandable functions are indicated in the documentation by a star:

<code>\cs_to_str:N *</code>	<code>\cs_to_str:N &lt;cs&gt;</code>
-----------------------------	--------------------------------------

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a `<cs>`, shorthand for a `<control sequence>`.

Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different ‘true’/‘false’ branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

<code>\xetex_if_engine:TF *</code>	<code>\xetex_if_engine:TF &lt;true code&gt; &lt;&gt;false code&gt;</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<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> or plain T<sub>E</sub>X. In these cases, the text will include an extra ‘**T<sub>E</sub>Xhackers note**’ section:

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

The normal description text.

**T<sub>E</sub>Xhackers note:** Detail for the experienced T<sub>E</sub>X or L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> programmer. In this case, it would point out that this function is the T<sub>E</sub>X primitive `\string`.

## Part II

# The l3names package

## A systematic naming scheme for T<sub>E</sub>X

### 3 Setting up the L<sup>A</sup>T<sub>E</sub>X3 programming language

This module is at the core of the L<sup>A</sup>T<sub>E</sub>X3 programming language. It performs the following tasks:

- defines new names for all T<sub>E</sub>X primitives;
- defines catcode regimes for programming;
- provides settings for when the code is used in a format;
- provides tools for when the code is used as a package within a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> context.

### 4 Using the modules

The modules documented in `source3` are designed to be used on top of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> 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<sup>A</sup>T<sub>E</sub>X3 format, but work in this area is incomplete and not included in this documentation.

As the modules use a coding syntax different from standard L<sup>A</sup>T<sub>E</sub>X it provides a few functions for setting it up.

<code>\ExplSyntaxOn</code> <code>\ExplSyntaxOff</code>	<code>\ExplSyntaxOn &lt;code&gt; \ExplSyntaxOff</code>
---	--

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

<code>\ExplSyntaxNamesOn</code> <code>\ExplSyntaxNamesOff</code>	<code>\ExplSyntaxNamesOn &lt;code&gt; \ExplSyntaxNamesOff</code>
---	--

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

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

The package `l3names` (this module) provides `\ProvidesExplPackage` which is a wrapper for `\ProvidesPackage` and sets up the  $\text{\LaTeX}$ 3 catcode settings for programming automatically. Similar for the relationship between `\ProvidesExplClass` and `\ProvidesClass`. Spaces are not ignored in the arguments of these commands.

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

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

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

```
\ProvidesExplPackage{\filename}{\filedate}{\fileversion}{\filedescription}
```

## Part III

# The l3basics package

## Basic Definitions

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

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

## 5 Predicates and conditionals

L<sup>A</sup>T<sub>E</sub>X3 has three concepts for conditional flow processing:

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

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

a function that will turn the first argument into a control sequence (since it's marked as c) then checks whether this control sequence is still free and then depending on the result carry out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as 'conditionals'; whenever a TF function is defined it will usually be accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions will always provide all three versions.

Important to note is that these branching conditionals with  $\langle true\ code \rangle$  and/or  $\langle false\ code \rangle$  are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they will be accompanied by a 'predicate' for the same test as described below.

**Predicates** 'Predicates' are functions that return a special type of boolean value which can be tested by the function `\if_predicate:w` or in the boolean expression parser. All functions of this type are expandable and have names that end with `_p` in the description part. For example,

`\cs_if_free_p:N`

would be a predicate function for the same type of test as the conditional described above. It would return ‘true’ if its argument (a single token denoted by `N`) is still free for definition. It would be used in constructions like

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

or in expressions utilizing the boolean logic parser:

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

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

For each predicate defined, a ‘predicate conditional’ will also exist that behaves like a conditional described above.

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

## 5.1 Primitive conditionals

The `ε-TEX` engine itself provides many different conditionals. Some expand whatever comes after them and others don’t. Hence the names for these underlying functions will often contain a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_num:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We will prefix primitive conditionals with `\if_`.

<code>\if_true:</code>	★	
<code>\if_false:</code>	★	
<code>\or:</code>	★	
<code>\else:</code>	★	
<code>\fi:</code>	★	<code>\if_true: &lt;true code&gt; \else: &lt;false code&gt; \fi:</code>
<code>\reverse_if:N</code>	★	<code>\if_false: &lt;true code&gt; \else: &lt;false code&gt; \fi:</code>
		<code>\reverse_if:N &lt;primitive conditional&gt;</code>

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

<sup>2</sup>If defined using the interface provided.

`\reverse_if:N` reverses any two-way primitive conditional. `\else:` and `\fi:` delimit the branches of the conditional. `\or:` is used in case switches, see `l3intexpr` for more.

**T<sub>E</sub>Xhackers note:** These are equivalent to their corresponding T<sub>E</sub>X primitive conditionals; `\reverse_if:N` is  $\varepsilon$ -T<sub>E</sub>X's `\unless`.

<code>\if_meaning:w</code> *	<code>\if_meaning:w</code> $\langle arg_1 \rangle$ $\langle arg_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
------------------------------	---

`\if_meaning:w` executes  $\langle true\ code \rangle$  when  $\langle arg_1 \rangle$  and  $\langle arg_2 \rangle$  are the same, otherwise it executes  $\langle false\ code \rangle$ .  $\langle arg_1 \rangle$  and  $\langle arg_2 \rangle$  could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

**T<sub>E</sub>Xhackers note:** This is T<sub>E</sub>X's `\ifx`.

<code>\if:w</code> *	<code>\if:w</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_charcode:w</code> *	
<code>\if_catcode:w</code> *	

`\if_catcode:w`  $\langle token_1 \rangle$   $\langle token_2 \rangle$   $\langle true\ code \rangle$  `\else:`  $\langle false\ code \rangle$  `\fi:`

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

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

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

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

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

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

`\if_cs_exist:w`  $\langle tokens \rangle$  `\cs_end:`  $\langle true\ code \rangle$  `\else:`  $\langle false\ code \rangle$  `\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 `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

<code>\if_mode_horizontal:</code>	<code>*</code>
<code>\if_mode_vertical:</code>	<code>*</code>
<code>\if_mode_math:</code>	<code>*</code>
<code>\if_mode_inner:</code>	<code>*</code>

`\if_mode_horizontal: <true code> \else: <false code> \fi:`

Execute *<true code>* if currently in horizontal mode, otherwise execute *<false code>*. Similar for the other functions.

## 5.2 Non-primitive conditionals

<code>\cs_if_eq_name_p:NN</code>
----------------------------------

`\cs_if_eq_name_p:NN <cs1> <cs2>`

Returns ‘true’ if *<cs<sub>1</sub>>* and *<cs<sub>2</sub>>* are textually the same, i.e. have the same name, otherwise it returns ‘false’.

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

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

These functions check if *<cs<sub>1</sub>>* and *<cs<sub>2</sub>>* have same meaning.

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

`\cs_if_free_p:N <cs>`  
`\cs_if_free:NNTF <cs> {\<true code>} {\<false code>}`

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

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

`\cs_if_exist_p:N <cs>`  
`\cs_if_exist:NNTF <cs> {\<true code>} {\<false code>}`

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

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

`\cs_if_do_not_use_p:N <cs>`

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

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

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

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

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

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

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

<code>\str_if_eq_p:nn *</code>
<code>\str_if_eq_p:Vn *</code>
<code>\str_if_eq_p:on *</code>
<code>\str_if_eq_p:no *</code>
<code>\str_if_eq_p:nV *</code>
<code>\str_if_eq_p:VV *</code>
<code>\str_if_eq_p:xx *</code>
<code>\str_if_eq:nnTF *</code>
<code>\str_if_eq:VnTF *</code>
<code>\str_if_eq:onTF *</code>
<code>\str_if_eq:noTF *</code>
<code>\str_if_eq:nVTF *</code>
<code>\str_if_eq:VVTF *</code>
<code>\str_if_eq:xxTF *</code>

`\str_if_eq_p:nn { $\langle tl_1 \rangle$ } { $\langle tl_2 \rangle$ }`  
`\str_if_eq:nnTF { $\langle tl_1 \rangle$ } { $\langle tl_2 \rangle$ } { $\langle true code \rangle$ } { $\langle false code \rangle$ }`

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

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

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

<code>\c_true_bool</code>
<code>\c_false_bool</code>

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

## 6 Control sequences

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

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

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

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

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

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

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

**$\text{\TeX}$ hackers note:** This is  $\text{\TeX}$ 's `\show` and associated csname version of it.

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

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

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

**$\text{\TeX}$ hackers note:** This is  $\text{\TeX}$ 's `\meaning` and associated csname version of it.

## 7 Selecting and discarding tokens from the input stream

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

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

`\use:n {\<arg>}`

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

**T<sub>E</sub>Xhackers note:** `\use:n` is L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@firstofone/\@iden`.

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

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

**T<sub>E</sub>Xhackers note:** `\use:c` is L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@nameuse`.

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

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

**T<sub>E</sub>Xhackers note:** LuaT<sub>E</sub>X provides `\expanded` which performs this operation in an expandable manner, but we cannot assume this behaviour on all platforms yet.

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

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

**T<sub>E</sub>Xhackers note:** `\use_none:n`, `\use_none:nn`, `\use_none:nnnn` are L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@gobble`, `\@gobbletwo`, and `\@gobblefour`.

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

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

**T<sub>E</sub>Xhackers note:** These are L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@firstoftwo` and `\@secondoftwo`, respectively.

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

`\use_i:nnn`  $\{\langle arg_1 \rangle\}$   $\{\langle arg_2 \rangle\}$   $\{\langle arg_3 \rangle\}$

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

**TeXhackers note:** L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> has only `\@thirdofthree`.

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

`\use_i:nnnn`  $\{\langle arg_1 \rangle\}$   $\{\langle arg_2 \rangle\}$   $\{\langle arg_3 \rangle\}$   $\{\langle arg_4 \rangle\}$

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

## 7.1 Extending the interface

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

`\use_i_ii:nnn`  $\{\langle arg_1 \rangle\}$   $\{\langle arg_2 \rangle\}$   $\{\langle arg_3 \rangle\}$

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

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

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

## 7.2 Selecting tokens from delimited arguments

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

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

`\use_none_delimit_by_q_nil:w`  $\langle balanced\ text \rangle$  `\q_nil`

Gobbles  $\langle balanced\ text \rangle$ . Useful in gobbling the remainder in a list structure or terminating recursion.

<code>\use_i_delimit_by_q_nil:nw</code>	<code>*</code>	
<code>\use_i_delimit_by_q_stop:nw</code>	<code>*</code>	
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	<code>*</code>	<code>\use_i_delimit_by_q_nil:nw {\langle arg \rangle} \langle balanced text \rangle \q_</code>

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

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

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

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

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

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

**T<sub>E</sub>Xhackers note:** This is T<sub>E</sub>X's `\expandafter`.

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

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

**T<sub>E</sub>Xhackers note:** These are T<sub>E</sub>X's `\noexpand` and  $\varepsilon$ -T<sub>E</sub>X's `\unexpanded`, respectively.

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

**T<sub>E</sub>Xhackers note:** Definition as in L<sup>A</sup>T<sub>E</sub>X's `\empty` but not used for the same thing.

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

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

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

Internal function for calling errors in our code.

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

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

## 9 Defining functions

There are two types of function definitions in L<sup>A</sup>T<sub>E</sub>X3: versions that check if the function name is still unused, and versions that simply make the definition. The latter are used for internal scratch functions that get new meanings all over the place.

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

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

**T<sub>E</sub>Xhackers note:** While T<sub>E</sub>X makes all definition functions directly available to the user L<sup>A</sup>T<sub>E</sub>X3 hides them very carefully to avoid the problems with definitions that are overwritten accidentally. Many functions that are in T<sub>E</sub>X a combination of prefixes and definition functions are provided as individual functions.

A slew of functions are defined in the following sections for defining new functions.

Here's a quick summary to get an idea of what's available:

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

That stands for, respectively, the following variations:

**g** Global or local;

**new/set** Define a new function or re-define an existing one;

**protected** Prevent expansion of the function in **x** arguments;

**nopar** Restrict the argument(s) from containing `\par`;

**N/c** Either a control sequence or a ‘csname’;

**p** Either the a primitive T<sub>E</sub>X argument or the number of arguments is detected from the argument signature, i.e., `\foo:nnn` is assumed to have three arguments `#1#2#3`;

**n/x** Either an unexpanded or an expanded definition.

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

## 9.1 Defining new functions using primitive parameter text

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

`\cs_new:Npn <cs> <parms> {<code>}`

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

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

`\cs_new_nopar:Npn <cs> <parms> {<code>}`

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

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

`\cs_new_protected:Npn <cs> <parms> {<code>}`

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

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

`\cs_new_protected_nopar:Npn <cs> <parms> {<code>}`

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

## 9.2 Defining new functions using the signature

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

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

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

**T<sub>E</sub>Xhackers note:** Internally, these use T<sub>E</sub>X's `\long`. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn't have been expected.

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

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

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

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

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

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

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

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

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

### 9.3 Defining functions using primitive parameter text

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

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

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

**T<sub>E</sub>Xhackers note:** These are equivalent to T<sub>E</sub>X's `\long\def` and so on. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn't have been expected.

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

Global variant of `\cs_set:Npn`.

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

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

**T<sub>E</sub>Xhackers note:** `\cs_set_nopar:Npn` is the L<sup>A</sup>T<sub>E</sub>X3 name for T<sub>E</sub>X's `\def` and `\cs_set_nopar:Npx` corresponds to the primitive `\edef`. The `\cs_set_nopar:cpn` function was known in L<sup>A</sup>T<sub>E</sub>X2 as `\@namedef`. `\cs_set_nopar:cpx` has no equivalent.

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

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

**T<sub>E</sub>Xhackers note:** `\cs_gset_nopar:Npn` and `\cs_gset_nopar:Npx` are T<sub>E</sub>X's `\gdef` and `\xdef`.

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

`\cs_set_protected:Npn <cs> <parms> {<code>}`

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

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

`\cs_gset_protected:Npn <cs> <parms> {<code>}`

Global versions of the above functions.

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

`\cs_set_protected_nopar:Npn <cs> <parms> {<code>}`

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

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

`\cs_gset_protected_nopar:Npn <cs> <parms> {<code>}`

Global versions of the above functions.

## 9.4 Defining functions using the signature (no checks)

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

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

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

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

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

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

Global variant of `\cs_set:Nn`.

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

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

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

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

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

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

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

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

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

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

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

Global versions of the above functions.

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

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

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

<code>\cs_gset_protected_nopar:Nn</code> <code>\cs_gset_protected_nopar:cn</code> <code>\cs_gset_protected_nopar:Nx</code> <code>\cs_gset_protected_nopar:cx</code>	<code>\cs_gset_protected_nopar:Nn &lt;cs&gt; {&lt;code&gt;}</code>
--	--

Global versions of the above functions.

## 9.5 Undefining functions

<code>\cs_undefine:N</code> <code>\cs_undefine:c</code> <code>\cs_gundefine:N</code> <code>\cs_gundefine:c</code>	<code>\cs_gundefine:N &lt;cs&gt;</code>
--	---

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

## 9.6 Copying function definitions

<code>\cs_new_eq:NN</code> <code>\cs_new_eq:cN</code> <code>\cs_new_eq:Nc</code> <code>\cs_new_eq:cc</code>	<code>\cs_new_eq:NN &lt;cs<sub>1</sub>&gt; &lt;cs<sub>2</sub>&gt;</code>
--	--

Gives the function  $\langle cs_1 \rangle$  locally or globally the current meaning of  $\langle cs_2 \rangle$ . If  $\langle cs_1 \rangle$  already exists then an error is called.

<code>\cs_set_eq:NN</code> <code>\cs_set_eq:cN</code> <code>\cs_set_eq:Nc</code> <code>\cs_set_eq:cc</code> <code>\cs_gset_eq:NN</code> <code>\cs_gset_eq:cN</code> <code>\cs_gset_eq:Nc</code> <code>\cs_gset_eq:cc</code>	<code>\cs_set_eq:cN &lt;cs<sub>1</sub>&gt; &lt;cs<sub>2</sub>&gt;</code>
--	--

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

<code>\cs_set_eq:NwN</code>	<code>\cs_set_eq:NwN &lt;cs<sub>1</sub>&gt; &lt;cs<sub>2</sub>&gt;</code> <code>\cs_set_eq:NwN &lt;cs<sub>1</sub>&gt; = &lt;cs<sub>2</sub>&gt;</code>
-----------------------------	--

These functions assign the meaning of  $\langle cs_2 \rangle$  locally or globally to the function  $\langle cs_1 \rangle$ .

Because the  $\text{\TeX}$  primitive operation is being used which may have an equal sign and (a certain number of) spaces between  $\langle cs_1 \rangle$  and  $\langle cs_2 \rangle$  the name contains a `w`. (Not happy about this convention!).

**$\text{\TeX}$ hackers note:** `\cs_set_eq:NwN` is the  $\text{\LaTeX}$ 3 name for  $\text{\TeX}$ 's `\let`.

## 9.7 Internal functions

<code>\pref_global:D</code> <code>\pref_long:D</code> <code>\pref_protected:D</code>	<code>\pref_global:D \cs_set_nopar:Npn</code>
--	---

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

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

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

## 10 The innards of a function

<code>\cs_to_str:N *</code>	<code>\cs_to_str:N \langle cs \rangle</code>
-----------------------------	--

This function returns the name of  $\langle cs \rangle$  as a sequence of letters with the escape character removed.

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

This function return the name of  $\langle arg \rangle$  as a sequence of letters including the escape character.

**$\text{\TeX}$ hackers note:** This is  $\text{\TeX}$ 's `\string`.

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

`\token_to_meaning:N <arg>`

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

**T<sub>E</sub>Xhackers note:** This is T<sub>E</sub>X's `\meaning`.

<code>\cs_get_function_name:N *</code>
<code>\cs_get_function_signature:N *</code>

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

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

<code>\cs_split_function:NN *</code>
--------------------------------------

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

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

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

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

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

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

## 11 Grouping and scanning

<code>\scan_stop:</code>
--------------------------

`\scan_stop:`

This function stops T<sub>E</sub>X's scanning ahead when ending a number.

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\relax` renamed.

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

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

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

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\begingroup` and `\endgroup` renamed.

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

`\group_execute_after:N` *<token>*

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

**T<sub>E</sub>Xhackers note:** This is T<sub>E</sub>X's `\aftergroup`.

## 12 Checking the engine

<code>\xetex_if_engine:TF</code> $\star$
--

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

This function detects if we're running a XeT<sub>E</sub>X-based format.

<code>\luatex_if_engine:TF</code> $\star$
---

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

This function detects if we're running a LuaT<sub>E</sub>X-based format.

<code>\c_xetex_is_engine_bool</code> <code>\c_luatex_is_engine_bool</code>
---

 Boolean variables used for the above functions.

## Part IV

# The l3expan package

## Controlling Expansion of Function Arguments

## 13 Brief overview

The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the L<sup>A</sup>T<sub>E</sub>X3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

## 14 Defining new variants

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

Internally preprocessing of arguments is done with functions from the `\exp_` module. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens the third argument gets expanded once. If `\seq_gpush:No` wouldn't be defined the example above could be coded in the following way:

```
\exp_args:NNo\seq_gpush:Nn
  \g_file_name_stack
  \l_tmpa_tl
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, e.g.

```
\cs_new_nopar:Npn\seq_gpush:No{\exp_args:NNo\seq_gpush:Nn}
```

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

### 14.1 Methods for defining variants

<code>\cs_generate_variant:Nn</code>	<code>\cs_generate_variant:Nn</code> <i>&lt;parent control sequence&gt;</i> { <i>&lt;variant argument specifier&gt;</i> }
--------------------------------------	--

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

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

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

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

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

### Internal functions

```
\cs_generate_internal_variant:n \cs_generate_internal_variant:n {<args>}
```

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

## 15 Introducing the variants

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

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

The `V` type returns the value of a register, which can be one of `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in `TEX` registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by `(cs)name`, the `v` specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `f` type is so special that it deserves an example. Let's pretend we want to set `\aaa` equal to the control sequence stemming from turning `b \l_tmpa_tl b` into a control sequence. Furthermore we want to store the execution of it in a `<toks>` register. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straight forward approach is

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

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

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

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

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

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

## 16 Manipulating the first argument

```
\exp_args:No ★ \exp_args:No <funct> <arg1> <arg2> ...
```

The first argument of `<funct>` (i.e., `<arg1>`) is expanded once, the result is surrounded by braces and passed to `<funct>`. `<funct>` may have more than one argument—all others are passed unchanged.

<code>\exp_args:Nc</code> ★ <code>\exp_args:cc</code> ★
--

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

The first argument of  $\langle funct \rangle$  (i.e.,  $\langle arg_1 \rangle$ ) is expanded until only characters remain. (An internal error occurs if something else is the result of this expansion.) Then the result is turned into a control sequence and passed to  $\langle funct \rangle$  as the first argument.  $\langle funct \rangle$  may have more than one argument—all others are passed unchanged.

In the `:cc` variant, the  $\langle funct \rangle$  control sequence itself is constructed (with the same process as described above) before  $\langle arg_1 \rangle$  is turned into a control sequence and passed as its argument.

<code>\exp_args:Nv</code> ★
-----------------------------

`\exp_args:Nv`  $\langle funct \rangle$   $\langle register \rangle$

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

<code>\exp_args:Nv</code> ★
-----------------------------

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

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

<code>\exp_args:Nx</code>
---------------------------

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

The first argument of  $\langle funct \rangle$  (i.e.,  $\langle arg_1 \rangle$ ) is fully expanded until only unexpandable tokens remain, the result is surrounded by braces and passed to  $\langle funct \rangle$ .  $\langle funct \rangle$  may have more than one argument—all others are passed unchanged. As mentioned before, this type of function is relatively slow.

<code>\exp_args:Nf</code> ★
-----------------------------

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

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

## 17 Manipulating two arguments

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

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

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

and non-expandable.

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

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

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

## 18 Manipulating three arguments

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

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

`\exp_args:Nnnx`  $\langle funct \rangle$   $\langle arg_1 \rangle$   $\langle arg_2 \rangle$   $\langle arg_3 \rangle$  ...

All the above functions are non-expandable.

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

`\exp_args:NNoo`  $\langle funct \rangle$   $\langle arg_1 \rangle$   $\langle arg_2 \rangle$   $\langle arg_3 \rangle$  ...

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

## 19 Preventing expansion

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

`\exp_not:N`  $\langle token \rangle$   
`\exp_not:n`  $\{ \langle token list \rangle \}$

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

**T<sub>E</sub>Xhackers note:** `\exp_not:N` is the primitive `\noexpand` renamed and `\exp_not:n` is the  $\varepsilon$ -T<sub>E</sub>X primitive `\unexpanded`.

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

`\exp_not:o`  $\{ \langle token list \rangle \}$

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

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

`\exp_not:V`  $\langle register \rangle$   
`\exp_not:v`  $\{ \langle token list \rangle \}$

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

<code>\exp_stop_f:</code>
---------------------------

$\langle f expansion \rangle$  ... `\exp_stop_f:`

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

```

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

```

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

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

## 20 Unbraced expansion

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

`\exp_last_unbraced:NV <token> <variable name>`

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

## Part V

# The l3prg package

## Program control structures

## 21 Conditionals and logical operations

Conditional processing in L<sup>A</sup>T<sub>E</sub>X3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead

in the input stream. After processing the input, a *state* is returned. The typical states returned are  $\langle true \rangle$  and  $\langle false \rangle$  but other states are possible, say an  $\langle error \rangle$  state for erroneous input, e.g., text as input in a function comparing integers.

L<sup>A</sup>T<sub>E</sub>X3 has two primary forms of conditional flow processing based on these states. One type is predicate functions that turn the returned state into a boolean  $\langle true \rangle$  or  $\langle false \rangle$ . For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean  $\langle true \rangle$  or  $\langle false \rangle$  values to be used in testing with `\if_predicate:w` or in functions to be described below. The other type is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the N) and then executes either  $\langle true \rangle$  or  $\langle false \rangle$  depending on the result. Important to note here is that the arguments are executed after exiting the underlying `\if...\fi:` structure

## 22 Defining a set of conditional functions

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

These functions exit conditional processing when used in conjunction with the generating functions listed below.

<code>\prg_set_conditional:Nnn</code> <code>\prg_set_conditional:Npnn</code> <code>\prg_new_conditional:Nnn</code> <code>\prg_new_conditional:Npnn</code> <code>\prg_set_protected_conditional:Nnn</code> <code>\prg_set_protected_conditional:Npnn</code> <code>\prg_new_protected_conditional:Nnn</code> <code>\prg_new_protected_conditional:Npnn</code> <code>\prg_set_eq_conditional:NNn</code> <code>\prg_new_eq_conditional:NNn</code>	<code>\prg_set_conditional:Nnn &lt;test&gt; &lt;conds&gt; &lt;code&gt;</code> <code>\prg_set_conditional:Npnn &lt;test&gt; &lt;param&gt; &lt;conds&gt; &lt;code&gt;</code>
--	---

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

An example can easily clarify matters here:

```
\prg_set_conditional:Nnn \foo_if_bar:NN {p,TF,T} {
```

```

\if_meaning:w \l_tmpa_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #2
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
\fi:
}

```

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

## 23 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (e.g., draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, etc. which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions are expandable and expect the input to also be fully expandable (which will generally mean being constructed from predicate functions, possibly nested).

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

`\bool_new:N <bool>`

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

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

`\bool_gset_false:N <bool>`

Set  $\langle bool \rangle$  either  $\langle true \rangle$  or  $\langle false \rangle$ . We can also do this globally.

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

`\bool_set_eq:NN <bool1> <bool2>`

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

<code>\bool_if_p:N *</code>
<code>\bool_if:NTF *</code>
<code>\bool_if_p:c *</code>
<code>\bool_if:cTF *</code>

`\bool_if:NTF <bool> {<true>} {<false>}`  
`\bool_if_p:N <bool>`

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

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

`\bool_while_do:Nn <bool> {<code>}`  
`\bool_until_do:Nn <bool> {<code>}`

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

## 24 Boolean expressions

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

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

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

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

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

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

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

from above returns  $\langle true \rangle$ .

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

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

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

`\bool_xor_p:nn *` `\bool_xor_p:nn {<boolean expression>} {<boolean expression>}`

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

`\bool_set:Nn`  
`\bool_set:cn`  
`\bool_gset:Nn`  
`\bool_gset:cn` `\bool_set:Nn <bool> {<boolean expression>}`

Sets `<bool>` to the logical outcome of evaluating `<boolean expression>`.

## 25 Case switches

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

This function evaluates the first `<integer expr>` and then compares it to the values found in the list. Thus the expression

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

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

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

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

This function works just like `\prg_case_int:nnn` except it works for `<dim>` registers.

```

\prg_case_str:nnn {\string} {
  {\string_1} {\code_1}
  {\string_2} {\code_2}
  ...
  {\string_n} {\code_n}
\prg_case_str:nnn * } {\else case}

```

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

The function is expandable<sup>3</sup> and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

```

\prg_case_tl:Nnn <tl var.> {
  <tl var._1> {\code_1} <tl var._2> {\code_2} ... <tl var._n>
  {\code_n}
\prg_case_tl:Nnn * } {\else case}

```

This function works just like `\prg_case_int:nnn` except it compares token list variables.

The function is expandable<sup>4</sup> and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

## 26 Generic loops

```

\bool_while_do:nn
\bool_until_do:nn
\bool_do_while:nn
\bool_do_until:nn
\bool_while_do:nn {\boolean expression} {\code}
\bool_until_do:nn {\boolean expression} {\code}

```

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

## 27 Choosing modes

```

\mode_if_vertical_p: *
\mode_if_vertical:TF * \mode_if_vertical:TF {\true code} {\false code}

```

Determines if  $\text{\TeX}$  is in vertical mode or not and executes either *<true code>* or *<>false code>* accordingly.

<sup>3</sup>Provided you use pdfTeX v1.30 or later

<sup>4</sup>Provided you use pdfTeX v1.30 or later

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

Determines if T<sub>E</sub>X is in horizontal mode or not and executes either *⟨true code⟩* or *⟨false code⟩* accordingly.

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

Determines if T<sub>E</sub>X is in inner mode or not and executes either *⟨true code⟩* or *⟨false code⟩* accordingly.

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

Determines if T<sub>E</sub>X is in math mode or not and executes either *⟨true code⟩* or *⟨false code⟩* accordingly.

**T<sub>E</sub>Xhackers note:** This version will choose the right branch even at the beginning of an alignment cell.

## 28 Alignment safe grouping and scanning

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

This function gets T<sub>E</sub>X on the right track inside an alignment cell but without destroying any kerning.

<code>\group_align_safe_begin:</code> <code>\group_align_safe_end:</code>	<code>\group_align_safe_begin: ⟨...⟩ \group_align_safe_end:</code>
--	--

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

## 29 Producing *n* copies

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

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

`\prg_replicate:nn {<number>} {<arg>}`  
 Creates *<number>* copies of *<arg>*. Note that it is expandable.

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

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

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

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

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

Same as `\prg_stepwise_function:nnnN` except here *<action>* is performed each time with `##1` as a placeholder for the number currently being tested. This function is not expandable and it is nestable.

<code>\prg_stepwise_variable:nnnNn</code>
---

`\prg_stepwise_variable:nnnn {<start>} {<step>} {<end>} {<temp-var>} {<action>}`

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

## 30 Sorting

<code>\prg_quicksort:n</code>
-------------------------------

`\prg_quicksort:n { {<item1>} {<item2>} ... {<itemn>} }`  
 Performs a Quicksort on the token list. The comparisons are performed by the function `\prg_quicksort_compare:nnTF` which is up to the programmer to define. When the sorting process is over, all items are given as argument to the function `\prg_quicksort_function:n` which the programmer also controls.

<code>\prg_quicksort_function:n</code> <code>\prg_quicksort_compare:nnTF</code>	<code>\prg_quicksort_function:n {&lt;element&gt;}</code> <code>\prg_quicksort_compare:nnTF {&lt;element<sub>1</sub>&gt;} {&lt;element<sub>2</sub>&gt;}</code>
--	--

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

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

Then the function call

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

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

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

### 30.1 Variable type and scope

```
\prg_variable_get_scope:N * \prg_variable_get_scope:N <variable>
```

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

```
\prg_variable_get_type:N * \prg_variable_get_type:N <variable>
```

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

### 30.2 Mapping to variables

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

Creates a family of mapping functions which can be applied to a token list, dividing the list up at each occurrence of the  $\langle token \rangle$ . The functions defined will be

- $\langle name \rangle\_map\_function:NN$
- $\langle name \rangle\_map\_function:nN$
- $\langle name \rangle\_map\_inline:Nn$
- $\langle name \rangle\_map\_inline:nn$
- $\langle name \rangle\_map\_break:$

Of these, the `inline` functions are not expandable but the other functions can be used in expansion contexts. The use of each function is best illustrated by the `\clist_map...` family defined by L<sup>A</sup>T<sub>E</sub>X3 itself for mapping to comma-separated lists. An error will be raised if the  $\langle name \rangle$  has already been used to generate a family of mapping functions. All of the definitions are created globally.

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

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

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

Of these, the `inline` functions are not expandable but the other functions can be used in expansion contexts. The use of each function is best illustrated by the `\clist_map_...` family defined by L<sup>A</sup>T<sub>E</sub>X3 itself for mapping to comma-separated lists. Any existing definitions for the `<name>` will be overwritten. All of the definitions are created globally.

## Part VI

# The l3quark package

## “Quarks”

A special type of constants in L<sup>A</sup>T<sub>E</sub>X3 are ‘quarks’. These are control sequences that expand to themselves and should therefore NEVER be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter is weird functions (for example as the stop token (i.e., `\q_stop`). They also permit the following ingenious trick: when you pick up a token in a temporary, and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary to the quark using `\if_meaning:w`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

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

The documentation needs some updating.

## 31 Functions

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

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

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

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

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

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

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

This tests whether or not *<token>* is equal to the quark `\q_nil`.

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

<code>\quark_if_nil_p:n *</code>	
<code>\quark_if_nil_p:V *</code>	
<code>\quark_if_nil_p:o *</code>	
<code>\quark_if_nil:nTF *</code>	
<code>\quark_if_nil:VTF *</code>	
<code>\quark_if_nil:oTF *</code>	<code>\quark_if_nil:NTF {&lt;tokens&gt;} {&lt;true code&gt;} {&lt;&gt;false code&gt;}</code>

This tests whether or not *<tokens>* is equal to the quark `\q_nil`.

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

## 32 Recursion

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

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

---

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

`\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.

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

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

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

`\tl_set:Nn \l_tmpa_tl { ⟨list element⟩ }`

<code>\quark_if_recursion_tail_stop_do:Nn *</code> <code>\quark_if_recursion_tail_stop_do:nn *</code> <code>\quark_if_recursion_tail_stop_do:on *</code>	<code>\quark_if_recursion_tail_stop_do:nn</code> <code>{⟨list element⟩} {⟨post action⟩}</code> <code>\quark_if_recursion_tail_stop_do:Nn</code> <code>⟨list element⟩ {⟨post action⟩}</code>
--	--

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

### 33 Constants

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

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

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

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

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

## Part VII

# The l3token package

## A token of my appreciation...

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in T<sub>E</sub>X, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such will have two primary function categories: `\token` for anything that deals with tokens and `\peek` for looking ahead in the token stream.

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

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

### 34 Character tokens

<code>\char_set_catcode:nn</code>	
<code>\char_set_catcode:w</code>	
<code>\char_value_catcode:n</code>	
<code>\char_value_catcode:w</code>	
<code>\char_show_value_catcode:n</code>	
<code>\char_show_value_catcode:w</code>	

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

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

`\char_set_catcode` is more usefully abstracted below.

**T<sub>E</sub>Xhackers note:** `\char_set_catcode:w` is the T<sub>E</sub>X primitive `\catcode` renamed.

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

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

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

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

**T<sub>E</sub>Xhackers note:** `\char_make_other:N` is L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@makeother`.

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

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

**T<sub>E</sub>Xhackers note:**  $\backslash char\_set\_lccode:w$  is the T<sub>E</sub>X primitive  $\backslash lccode$  renamed.

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

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

**T<sub>E</sub>Xhackers note:**  $\backslash char\_set\_uccode:w$  is the T<sub>E</sub>X primitive  $\backslash uccode$  renamed.

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

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

**T<sub>E</sub>Xhackers note:**  $\backslash char\_set\_sfcode:w$  is the T<sub>E</sub>X primitive  $\backslash sfcode$  renamed.

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

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

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

## 35 Generic tokens

<code>\token_new:Nn</code>	<code>\token_new:Nn &lt;token<sub>1</sub>&gt; {&lt;token<sub>2</sub>&gt;}</code>
----------------------------	--

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

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

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

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

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

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

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

$\backslash token\_if\_math\_shift\_p:N \star$ $\backslash token\_if\_math\_shift:N\mathit{TF} \star$	$\backslash token\_if\_math\_shift:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

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

$\backslash token\_if\_alignment\_tab\_p:N \star$ $\backslash token\_if\_alignment\_tab:N\mathit{TF} \star$	$\backslash token\_if\_alignment\_tab:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	--

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

$\backslash token\_if\_parameter\_p:N \star$ $\backslash token\_if\_parameter:N\mathit{TF} \star$	$\backslash token\_if\_parameter:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

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

$\backslash token\_if\_math\_superscript\_p:N \star$ $\backslash token\_if\_math\_superscript:N\mathit{TF} \star$	$\backslash token\_if\_math\_superscript:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

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

$\backslash token\_if\_math\_subscript\_p:N \star$ $\backslash token\_if\_math\_subscript:N\mathit{TF} \star$	$\backslash token\_if\_math\_subscript:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

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

$\backslash token\_if\_space\_p:N \star$ $\backslash token\_if\_space:N\mathit{TF} \star$	$\backslash token\_if\_space:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

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

$\backslash token\_if\_letter\_p:N \star$ $\backslash token\_if\_letter:N\mathit{TF} \star$	$\backslash token\_if\_letter:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	--

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

$\backslash token\_if\_other\_char\_p:N \star$ $\backslash token\_if\_other\_char:N\mathit{TF} \star$	$\backslash token\_if\_other\_char:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

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

<code>\token_if_active_char_p:N *</code> <code>\token_if_active_char:NTF *</code>	<code>\token_if_active_char:NTF &lt;token&gt; {\true} {\false}</code>
--	---

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

<code>\token_if_eq_meaning_p:NN *</code> <code>\token_if_eq_meaning:NNTF *</code>	<code>\token_if_eq_meaning:NNTF &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {\true} {\false}</code>
--	---

Check if the meaning of two tokens are identical.

<code>\token_if_eq_catcode_p:NN *</code> <code>\token_if_eq_catcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {\true} {\false}</code>
--	---

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

<code>\token_if_eq_charcode_p:NN *</code> <code>\token_if_eq_charcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {\true} {\false}</code>
--	---

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

<code>\token_if_macro_p:N *</code> <code>\token_if_macro:NTF *</code>	<code>\token_if_macro:NTF &lt;token&gt; {\true} {\false}</code>
--	---

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

<code>\token_if_cs_p:N *</code> <code>\token_if_cs:NTF *</code>	<code>\token_if_cs:NTF &lt;token&gt; {\true} {\false}</code>
--	--

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

<code>\token_if_expandable_p:N *</code> <code>\token_if_expandable:NTF *</code>	<code>\token_if_expandable:NTF &lt;token&gt; {\true} {\false}</code>
--	--

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

The next set of functions here are for picking apart control sequences. Sometimes it is useful to know if a control sequence has arguments and if so, how many. Similarly its status with respect to `\long` or `\protected` is good to have. Finally it can be very useful

to know if a control sequence is of a certain type: Is this  $\langle toks \rangle$  register we’re trying to to something with really a  $\langle toks \rangle$  register at all?

$\backslash token\_if\_long\_macro\_p:N \star$ $\backslash token\_if\_long\_macro:N\mathit{TF} \star$	$\backslash token\_if\_long\_macro:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

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

$\backslash token\_if\_protected\_macro\_p:N \star$ $\backslash token\_if\_protected\_macro:N\mathit{TF} \star$	$\backslash token\_if\_long\_macro:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

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

$\backslash token\_if\_protected\_long\_macro\_p:N \star$ $\backslash token\_if\_protected\_long\_macro:N\mathit{TF} \star$	$\backslash token\_if\_protected\_long\_macro:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

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

$\backslash token\_if\_chardef\_p:N \star$ $\backslash token\_if\_chardef:N\mathit{TF} \star$	$\backslash token\_if\_chardef:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

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

$\backslash token\_if\_mathchardef\_p:N \star$ $\backslash token\_if\_mathchardef:N\mathit{TF} \star$	$\backslash token\_if\_mathchardef:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

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

$\backslash token\_if\_int\_register\_p:N \star$ $\backslash token\_if\_int\_register:N\mathit{TF} \star$	$\backslash token\_if\_int\_register:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

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

$\backslash token\_if\_dim\_register\_p:N \star$ $\backslash token\_if\_dim\_register:N\mathit{TF} \star$	$\backslash token\_if\_dim\_register:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

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

<code>\token_if_skip_register_p:N *</code> <code>\token_if_skip_register:NTF *</code>	<code>\token_if_skip_register:NTF &lt;token&gt; {\true} {\false}</code>
--	---

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

<code>\token_if_toks_register_p:N *</code> <code>\token_if_toks_register:NTF *</code>	<code>\token_if_toks_register:NTF &lt;token&gt; {\true} {\false}</code>
--	---

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

<code>\token_get_prefix_spec:N *</code> <code>\token_get_arg_spec:N *</code> <code>\token_get_replacement_spec:N *</code>	<code>\token_get_arg_spec:N &lt;token&gt;</code>
---	--

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

If the `arg_spec` contains the string `->`, then the `spec` function will produce incorrect results.

### 35.1 Useless code: because we can!

<code>\token_if_primitive_p:N *</code> <code>\token_if_primitive:NTF *</code>	<code>\token_if_primitive:NTF &lt;token&gt; {\true} {\false}</code>
--	---

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

## 36 Peeking ahead at the next token

<code>\l_peek_token</code> <code>\g_peek_token</code> <code>\l_peek_search_token</code>
---

Some useful variables. Initially they are set to ?.

<code>\peek_after:NN</code> <code>\peek_gafter:NN</code>	<code>\peek_after:NN &lt;function&gt;&lt;token&gt;</code>
---	---

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

sort of test on this token. Leaves  $\langle token \rangle$  in the input stream.  $\backslash peek\_gafter:NN$  does this globally to the token  $\backslash g\_peek\_token$ .

**T<sub>E</sub>Xhackers note:** This is the primitive  $\backslash futurelet$  turned into a function.

$\backslash peek\_meaning:N\overline{T}\overline{F}$ $\backslash peek\_meaning\_ignore\_spaces:N\overline{T}\overline{F}$ $\backslash peek\_meaning\_remove:N\overline{T}\overline{F}$ $\backslash peek\_meaning\_remove\_ignore\_spaces:N\overline{T}\overline{F}$	$\backslash peek\_meaning:N\overline{T}\overline{F} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

$\backslash peek\_meaning:N\overline{T}\overline{F}$  checks (by using  $\backslash if\_meaning:w$ ) if  $\langle token \rangle$  equals the next token in the input stream and executes either  $\langle true\ code \rangle$  or  $\langle false\ code \rangle$  accordingly.  $\backslash peek\_meaning\_remove:N\overline{T}\overline{F}$  does the same but additionally removes the token if found. The  $ignore\_spaces$  versions skips blank spaces before making the decision.

**T<sub>E</sub>Xhackers note:** This is equivalent to L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s  $\backslash @ifnextchar$ .

$\backslash peek\_charcode:N\overline{T}\overline{F}$ $\backslash peek\_charcode\_ignore\_spaces:N\overline{T}\overline{F}$ $\backslash peek\_charcode\_remove:N\overline{T}\overline{F}$ $\backslash peek\_charcode\_remove\_ignore\_spaces:N\overline{T}\overline{F}$	$\backslash peek\_charcode:N\overline{T}\overline{F} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

Same as for the  $\backslash peek\_meaning:N\overline{T}\overline{F}$  functions above but these use  $\backslash if\_charcode:w$  to compare the tokens.

$\backslash peek\_catcode:N\overline{T}\overline{F}$ $\backslash peek\_catcode\_ignore\_spaces:N\overline{T}\overline{F}$ $\backslash peek\_catcode\_remove:N\overline{T}\overline{F}$ $\backslash peek\_catcode\_remove\_ignore\_spaces:N\overline{T}\overline{F}$	$\backslash peek\_catcode:N\overline{T}\overline{F} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Same as for the  $\backslash peek\_meaning:N\overline{T}\overline{F}$  functions above but these use  $\backslash if\_catcode:w$  to compare the tokens.

$\backslash peek\_token\_generic:NN\overline{T}\overline{F}$ $\backslash peek\_token\_remove\_generic:NN\overline{T}\overline{F}$	$\backslash peek\_token\_generic:NN\overline{T}\overline{F} \langle token \rangle \langle function \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

$\backslash peek\_token\_generic:NN\overline{T}\overline{F}$  looks ahead and checks if the next token in the input stream is equal to  $\langle token \rangle$ . It uses  $\langle function \rangle$  to make that decision.  $\backslash peek\_token\_remove\_generic:NN\overline{T}\overline{F}$  does the same thing but additionally removes  $\langle token \rangle$  from the input stream if it is found. This also works if  $\langle token \rangle$  is either  $\backslash c\_group\_begin\_token$  or  $\backslash c\_group\_end\_token$ .

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

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

## Part VIII

# The l3int package

## Integers/counters

### 37 Integer values

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

#### 37.1 Integer expressions

<code>\int_eval:n *</code>	<code>\int_eval:n {⟨integer expression⟩}</code>
----------------------------	---

Evaluates the *⟨integer expression⟩*, expanding any integer and token list variables within the *⟨expression⟩* to their content (without requiring `\int_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a number. For example both

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

and

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

both evaluate to  $-6$ . The  $\{\langle integer\ expression\rangle\}$  may contain the operators  $+$ ,  $-$ ,  $*$  and  $/$ , along with parenthesis  $($  and  $)$ .

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

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

`\int_div_round:nn *` `\int_div_round:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}`

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

`\int_div_truncate:nn *` `\int_div_truncate:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}`

Evaluates the two  $\langle integer\ expressions\rangle$  as described earlier, then calculates the result of dividing the first value by the second, truncating any remainder. Note that division using  $/$  rounds the result.

`\int_max:nn *` `\int_max:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}`  
`\int_min:nn *` `\int_min:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}`

Evaluates the  $\langle integer\ expressions\rangle$  as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream, depending on the function name.

`\int_mod:nn *` `\int_mod:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}`

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

## 37.2 Integer variables

`\int_new:N`  
`\int_new:c` `\int_new:N \langle integer\rangle`

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

`\int_set_eq:NN`  
`\int_set_eq:cN`  
`\int_set_eq:Nc`  
`\int_set_eq:cc` `\int_set_eq:NN \langle integer1\rangle \langle integer2\rangle`

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

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

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

Sets the content of  $\langle integer1 \rangle$  equal to that of  $\langle integer2 \rangle$ . This assignment is global and so is not limited by the current T<sub>E</sub>X group level.

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

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

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

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

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

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

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

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

Decreases the value stored in  $\langle integer \rangle$  by 1 within the scope of the current T<sub>E</sub>X group.

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

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

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

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

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

Increases the value stored in  $\langle integer \rangle$  by 1 within the scope of the current T<sub>E</sub>X group.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`\int_zero:N <integer>`

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

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

`\int_gzero:N <integer>`

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

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

`\int_show:N <integer>`

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

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

`\int_use:N <integer>`

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

### 37.3 Comparing integer expressions

```

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

```

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

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

Equal	=
Greater than	>
Less than	<

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

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

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

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

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

<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</code> { $\langle integer\ expression \rangle$ }
<code>\int_if_odd:nTF *</code>	<code>\int_if_odd:nTF</code> { $\langle integer\ expression \rangle$ }
	{ $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

This function first evaluates the  $\langle integer\ expression \rangle$  as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate. The branching versions then leave either  $\langle true\ code \rangle$  or  $\langle false\ code \rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

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

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

After the  $\langle code \rangle$  has been processed by T<sub>E</sub>X 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 {\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: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<sub>E</sub>X 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 {\langle intexpr1 \rangle} \langle relation \rangle {\langle intexpr2 \rangle} {\langle code \rangle}</code>
-----------------------------------	---

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

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

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

<code>\int_do_while:nn *</code>	<code>\int_do_while:nn { \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<sub>E</sub>X 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<sub>E</sub>X 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<sub>E</sub>X 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</code>	<code>{</code>	$\langle intexpr1 \rangle$	<code>&lt;</code>	$\langle relation \rangle$	<code>&gt;</code>	$\langle intexpr2 \rangle$	<code>}</code>
	<code>{\langle code \rangle}</code>							

Places the  $\langle code \rangle$  in the input stream for T<sub>E</sub>X 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`.

### 37.4 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</code>	<code>{\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</code>	<code>{\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.

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

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

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

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

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

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

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

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

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

Calculates the value of the *⟨integer expression⟩* and places the symbol representation of the result in the input stream. The list of symbols used is equivalent to L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@fnsymbol` set.

## 37.5 Converting from other formats

`\int_from_alpha:n *` `\int_from_alpha: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_alpha: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  $\langle hexadecimal\ number \rangle$  into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the  $\langle hexadecimal\ number \rangle$  by upper or lower case letters.

`\int_from_octal:n *` `\int_from_octal:n { $\langle octal\ number \rangle$ }`

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

`\int_from_roman:n *` `\int_from_roman:n { $\langle roman\ numeral \rangle$ }`

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

## 37.6 Low-level conversion functions

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

`\int_convert_from_base_ten:nn *` `\int_convert_from_base_ten:nn { $\langle integer\ expression \rangle$ }`  
`{ $\langle base \rangle$ }`

Calculates the value of the  $\langle integer\ expression \rangle$  and converts it into the appropriate representation in the  $\langle base \rangle$ ; the later may be given as an integer expression. For bases greater than 10 the higher ‘digits’ are represented by the upper case letters from the English alphabet (with normal category codes). The maximum  $\langle base \rangle$  value is 36.

`\int_convert_to_base_ten:nn *` `\int_convert_to_base_ten:nn { $\langle number \rangle$ }`  
`{ $\langle base \rangle$ }`

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

`\int_convert_to_symbols:nnn *` `\int_convert_to_symbols:nnn`  
`{ $\langle integer\ expression \rangle$ }` `{ $\langle total\ symbols \rangle$ }`  
 `$\langle value\ to\ symbol\ mapping \rangle$`

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

```

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

```

## 38 Variables and constants

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

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

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

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

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

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

```
\c_minus_one
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_hundred_one
\c_twohundred_fifty_five
\c_twohundred_fifty_six
\c_thousand
\c_ten_thousand
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three
\c_ten_thousand_four
\c_twenty_thousand
```

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

`\c_max_register_int` Maximum number of registers.

## 38.1 Internal functions

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

Converts *<integer>* to its lowercase roman representation. Note that it produces a string of letters with catcode 12.

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\romannumeral` renamed.

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

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

<code>\int_convert_number_with_rule:nnN</code>	
<code>\int_symbol_math_conversion_rule:n</code>	<code>\int_convert_number_with_rule:nnN {&lt;int<sub>1</sub>&gt;} {&lt;int<sub>2</sub>&gt;}</code>
<code>\int_symbol_text_conversion_rule:n</code>	<code>&lt;function&gt;</code>

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

<code>\if_num:w</code>	<code>*</code>	<code>\if_num:w &lt;number<sub>1</sub>&gt; &lt;rel&gt; &lt;number<sub>2</sub>&gt; &lt;true&gt; \else: &lt;false&gt;</code>
<code>\if_int_compare:w</code>	<code>*</code>	<code>\fi:</code>

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

**T<sub>E</sub>Xhackers note:** These are both names for the T<sub>E</sub>X primitive `\ifnum`.

<code>\if_case:w</code>	<code>*</code>	<code>\if_case:w &lt;number&gt; &lt;case<sub>0</sub>&gt; \or: &lt;case<sub>1</sub>&gt; \or: ... \else:</code>
<code>\or:</code>	<code>*</code>	<code>&lt;default&gt; \fi:</code>

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

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\ifcase` and `\or`.

<code>\int_value:w</code>	<code>*</code>	<code>\int_value:w &lt;integer&gt;</code>
		<code>\int_value:w &lt;tokens&gt; &lt;optional space&gt;</code>

Expands `<tokens>` until an `<integer>` is formed. One space may be gobbled in the process.

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\number`.

<code>\int_eval:w</code>	<code>*</code>	
<code>\int_eval_end:</code>		<code>\int_eval:w &lt;int expr&gt; \int_eval_end:</code>

Evaluates `<integer expression>` as described for `\int_eval:n`. The evaluation stops when

an unexpandable token with category code other than 12 is read or when `\int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

**T<sub>E</sub>Xhackers note:** This is the  $\varepsilon$ -T<sub>E</sub>X primitive `\numexpr`.

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

`\if_int_odd:w <tokens> <true> \else: <false> \fi:  
\if_int_odd:w <number> <true> \else: <false> \fi:`

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

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\ifodd`.

## Part IX

# The l3skip package

## Dimension and skip registers

L<sup>A</sup>T<sub>E</sub>X3 knows about two types of length registers for internal use: rubber lengths (**skips**) and rigid lengths (**dims**).

## 39 Skip registers

### 39.1 Functions

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

`\skip_new:N <skip>`

Defines `<skip>` to be a new variable of type **skip**.

**T<sub>E</sub>Xhackers note:** `\skip_new:N` is the equivalent to plain T<sub>E</sub>X's `\newskip`.

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

`\skip_zero:N <skip>`

Locally or globally reset `<skip>` to zero. For global variables the global versions should be used.

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

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

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

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

`\skip_add:Nn    <skip> {<length>}`

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

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

`\skip_gsub:Nn    <skip> {<length>}`

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

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

`\skip_use:N    <skip>`

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

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

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

`\skip_show:N    <skip>`

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

**T<sub>E</sub>Xhackers note:** The function `\skip_show:N` could be implemented directly as the T<sub>E</sub>X primitive `\tex_showthe:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

<code>\skip_horizontal:N</code>	
<code>\skip_horizontal:c</code>	
<code>\skip_horizontal:n</code>	
<code>\skip_vertical:N</code>	
<code>\skip_vertical:c</code>	<code>\skip_horizontal:N</code> $\langle skip \rangle$
<code>\skip_vertical:n</code>	<code>\skip_horizontal:n</code> $\{\langle length \rangle\}$

The `hor` functions insert  $\langle skip \rangle$  or  $\langle length \rangle$  with the  $\text{\TeX}$  primitive `\hskip`. The `vertical` variants do the same with `\vskip`. The `n` versions evaluate  $\langle length \rangle$  with `\skip_eval:n`.

<code>\skip_if_infinite_glue_p:n</code>	
<code>\skip_if_infinite_glue:nTF</code>	<code>\skip_if_infinite_glue:nTF</code> $\{\langle skip \rangle\}$ $\{\langle true \rangle\}$ $\{\langle false \rangle\}$

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

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN</code> $\{\langle skip \rangle\}$ $\{\langle action \rangle\}$ $\langle dimen_1 \rangle$ $\langle dimen_2 \rangle$
--	---

Checks if  $\langle skip \rangle$  contains finite glue. If it does then it assigns  $\langle dimen_1 \rangle$  the stretch component and  $\langle dimen_2 \rangle$  the shrink component. If it contains infinite glue set  $\langle dimen_1 \rangle$  and  $\langle dimen_2 \rangle$  to zero and execute #2 which is usually an error or warning message of some sort.

<code>\skip_eval:n *</code>	<code>\skip_eval:n</code> $\{\langle skip\ expr \rangle\}$
-----------------------------	--

Evaluates the value of  $\langle skip\ expr \rangle$  so that `\skip_eval:n {5pt plus 3fil + 3pt minus 1fil}` puts `8.0pt plus 3.0fil minus 1.0fil` back into the input stream. Expandable.

**$\text{\TeX}$ hackers note:** This is the  $\varepsilon$ - $\text{\TeX}$  primitive `\glueexpr` turned into a function taking an argument.

## 39.2 Formatting a skip register value

## 39.3 Variable and constants

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

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

```

\l_tmpa_skip
\l_tmpb_skip
\l_tmpc_skip
\g_tmpa_skip
\g_tmpb_skip

```

Scratch register for immediate use.

## 40 Dim registers

### 40.1 Functions

```

\dim_new:N
\dim_new:c

```

```
\dim_new:N <dim>
```

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

**T<sub>E</sub>Xhackers note:** `\dim_new:N` is the equivalent to plain T<sub>E</sub>X's `\newdimen`.

```

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

```

```
\dim_zero:N <dim>
```

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

```

\dim_set:Nn
\dim_set:Nc
\dim_set:cn
\dim_gset:Nn
\dim_gset:Nc
\dim_gset:cn
\dim_gset:cc

```

```
\dim_set:Nn <dim> {<dim value>}
```

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

```

\dim_set_max:Nn
\dim_set_max:cn

```

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

Compares the current value of the  $\langle dimension \rangle$  with that of the  $\langle dimension expression \rangle$ , and sets the  $\langle dimension \rangle$  to the larger of these two value. This assignment is local to the current T<sub>E</sub>X group.

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

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

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

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

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

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

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

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

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

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

`\dim_add:Nn <dim> {<length>}`

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

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

`\dim_gsub:Nn <dim> {<length>}`

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

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

`\dim_use:N <dim>`

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

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

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

`\dim_show:N <dim>`

This function pauses the compilation and displays the length value kept in `<skip>` in the console output and log file.

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

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

`\dim_eval:n {<dim expr>}`

Evaluates the value of a dimension expression so that `\dim_eval:n {5pt+3pt}` puts 8pt back into the input stream. Expandable.

**T<sub>E</sub>Xhackers note:** This is the  $\varepsilon$ -T<sub>E</sub>X primitive `\dimexpr` turned into a function taking an argument.

<code>\if_dim:w</code>
------------------------

`\if_dim:w <dimen12`

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

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\ifdim`.

<code>\dim_compare_p:n *</code>	<code>\dim_compare_p:n {&lt;&lt;dim expr.<sub>12</sub></code>
<code>\dim_compare:nTF *</code>	<code>\dim_compare:nTF {&lt;&lt;dim expr.<sub>12</sub></code>
	<code>&lt;true code&gt; &lt;false code&gt;</code>

Evaluates `<dim expr.1 and <dim expr.2 and then carries out a comparison of the resulting lengths using C-like operators:`

Less than	<code>&lt;</code>	Less than or equal	<code>&lt;=</code>
Greater than	<code>&gt;</code>	Greater than or equal	<code>&gt;=</code>
Equal	<code>==</code> or <code>=</code>	Not equal	<code>!=</code>

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

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

A single equals sign is available as comparator (in addition to those familiar to C users) as standard T<sub>E</sub>X practice is to compare values using `=`.

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

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

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

you can safely omit the braces.

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

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\ifdim` turned into a function.

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

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

## 40.2 Variable and constants

<code>\c_max_dim</code>	Constant that denotes the maximum value which can be stored in a <code>\dim</code> register.
-------------------------	--

<code>\c_zero_dim</code>	Set of constants denoting useful values.
--------------------------	--

<code>\l_tmpa_dim</code> <code>\l_tmpp_dim</code> <code>\l_tmppc_dim</code> <code>\l_tmppd_dim</code> <code>\g_tmpa_dim</code> <code>\g_tmpp_dim</code>	Scratch register for immediate use.
--	-------------------------------------

## 41 Muskips

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

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

**T<sub>E</sub>Xhackers note:** Defines  $\langle muskip \rangle$  to be a new variable of type `muskip`. `\muskip_new:N` is the equivalent to plain T<sub>E</sub>X's `\newmuskip`.

<code>\muskip_set:Nn</code> <code>\muskip_gset:Nn</code>
---

`\muskip_set:Nn`     $\langle muskip \rangle$  { $\langle muskip \text{ value} \rangle$ }

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

<code>\muskip_add:Nn</code> <code>\muskip_gadd:Nn</code>
---

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

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

<code>\muskip_sub:Nn</code> <code>\muskip_gsub:Nn</code>
---

`\muskip_gsub:Nn`     $\langle muskip \rangle$  { $\langle length \rangle$ }

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

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

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

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

**T<sub>E</sub>Xhackers note:** See note for `\dim_use:N`.

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

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

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

## Part X

# The l3tl package

# Token Lists

L<sup>A</sup>T<sub>E</sub>X3 stores token lists in variables also called ‘token lists’. Variables of this type get the suffix `tl` and functions of this type have the prefix `tl`. To use a token list variable you simply call the corresponding variable.

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

## 42 Functions

<code>\tl_new:N</code> <code>\tl_new:c</code> <code>\tl_new:Nn</code> <code>\tl_new:cn</code> <code>\tl_new:Nx</code>	<code>\tl_new:Nn &lt;tl var.&gt; {&lt;initial token list&gt;}</code>
---	--

Defines `<tl var.>` globally to be a new variable to store a token list. `<initial token list>` is the initial value of `<tl var.>`. This makes it possible to assign values to a constant token list variable.

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

<code>\tl_const:Nn</code>	<code>\tl_const:Nn &lt;tl var.&gt; {&lt;token list&gt;}</code>
---------------------------	--

Defines `<tl var.>` as a global constant expanding to `<token list>`. The name of the constant must be free when the constant is created.

<code>\tl_use:N</code> <code>\tl_use:c</code>	<code>\tl_use:N &lt;tl var.&gt;</code>
--	--

Function that inserts the `<tl var.>` into the processing stream. Instead of `\tl_use:N` simply placing the `<tl var.>` into the input stream is also supported. `\tl_use:c` will complain if the `<tl var.>` hasn’t been declared previously!

<code>\tl_show:N</code> <code>\tl_show:c</code> <code>\tl_show:n</code>	<code>\tl_show:N &lt;tl var.&gt;</code> <code>\tl_show:n {&lt;token list&gt;}</code>
---	---

Function that pauses the compilation and displays the `<tl var.>` or `<token list>` on the console output and in the log file.

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

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

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

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

`\tl_clear:N <tl var.>`

The  $\langle tl\ var.\rangle$  is locally or globally cleared. The `c` variants will generate a control sequence name which is then interpreted as  $\langle tl\ var.\rangle$  before clearing.

<code>\tl_clear_new:N</code>
<code>\tl_clear_new:c</code>
<code>\tl_gclear_new:N</code>
<code>\tl_gclear_new:c</code>

`\tl_clear_new:N <tl var.>`

These functions check if  $\langle tl\ var.\rangle$  exists. If it does it will be cleared; if it doesn't it will be allocated.

<code>\tl_put_left:Nn</code>
<code>\tl_put_left:NV</code>
<code>\tl_put_left:No</code>
<code>\tl_put_left:Nx</code>
<code>\tl_put_left:cn</code>
<code>\tl_put_left:cV</code>
<code>\tl_put_left:co</code>

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

These functions will append *<token list>* to the left of *<tl var.>*. *<token list>* might be subject to expansion before assignment.

<code>\tl_put_right:Nn</code>
<code>\tl_put_right:NV</code>
<code>\tl_put_right:No</code>
<code>\tl_put_right:Nx</code>
<code>\tl_put_right:cn</code>
<code>\tl_put_right:cV</code>
<code>\tl_put_right:co</code>

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

These functions append *<token list>* to the right of *<tl var.>*.

<code>\tl_gput_left:Nn</code>
<code>\tl_gput_left:No</code>
<code>\tl_gput_left:NV</code>
<code>\tl_gput_left:Nx</code>
<code>\tl_gput_left:cn</code>
<code>\tl_gput_left:co</code>
<code>\tl_gput_left:cV</code>

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

These functions will append *<token list>* globally to the left of *<tl var.>*.

<code>\tl_gput_right:Nn</code>
<code>\tl_gput_right:No</code>
<code>\tl_gput_right:NV</code>
<code>\tl_gput_right:Nx</code>
<code>\tl_gput_right:cn</code>
<code>\tl_gput_right:co</code>
<code>\tl_gput_right:cV</code>

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

These functions will globally append *<token list>* to the right of *<tl var.>*.

A word of warning is appropriate here: Token list variables are implemented as macros and as such currently inherit some of the peculiarities of how T<sub>E</sub>X handles #s in the argument of macros. In particular, the following actions are legal

```

\tl_set:Nn \l_tmpa_tl{##1}
\tl_put_right:Nn \l_tmpa_tl{##2}
\tl_set:No \l_tmpb_tl{\l_tmpa_tl ##3}

```

x type expansions where macros being expanded contain #s do not work and will not work until there is an `\expanded` primitive in the engine. If you want them to work you must double #s another level.

<code>\tl_set_eq:NN</code>
<code>\tl_set_eq:Nc</code>
<code>\tl_set_eq:cN</code>
<code>\tl_set_eq:cc</code>
<code>\tl_gset_eq:NN</code>
<code>\tl_gset_eq:Nc</code>
<code>\tl_gset_eq:cN</code>
<code>\tl_gset_eq:cc</code>

`\tl_set_eq:NN <tl var. 1> <tl var. 2>`

Fast form for `\tl_set:No <tl var. 1> {\<tl var. 2>}`  
 when `<tl var. 2>` is known to be a variable of type `tl`.

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

`\tl_to_str:N <tl var. >`

This function returns the token list kept in `<tl var. >` as a string list with all characters catcoded to ‘other’.

<code>\tl_to_str:n</code>
---------------------------

`\tl_to_str:n {\<token list>}`

This function turns its argument into a string where all characters have catcode ‘other’.

**TeXhackers note:** This is the  $\varepsilon$ -TeX primitive `\detokenize`.

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

`\tl_rescan:nn {\<catcode setup>} {\<token list>}`

Returns the result of re-tokenising `<token list>` with the catcode setup (and whatever other redefinitions) specified. This is useful because the catcodes of characters are ‘frozen’ when first tokenised; this allows their meaning to be changed even after they’ve been read as an argument. Also see `\tl_set_rescan:Nnn` below.

**TeXhackers note:** This is a wrapper around  $\varepsilon$ -TeX’s `\scantokens`.

<code>\tl_set_rescan:Nnn</code>
<code>\tl_set_rescan:Nno</code>
<code>\tl_set_rescan:Nnx</code>
<code>\tl_gset_rescan:Nnn</code>
<code>\tl_gset_rescan:Nno</code>
<code>\tl_gset_rescan:Nnx</code>

`\tl_set_rescan:Nnn <tl var. > {\<catcode setup>} {\<token list>}`

Sets `<tl var. >` to the result of re-tokenising `<token list>` with the catcode setup (and whatever other redefinitions) specified.

**T<sub>E</sub>Xhackers note:** This is a wrapper around  $\varepsilon$ -T<sub>E</sub>X’s `\scantokens`.

## 43 Predicates and conditionals

<code>\tl_if_empty_p:N *</code> <code>\tl_if_empty_p:c *</code>	<code>\tl_if_empty_p:N &lt;tl var.&gt;</code>
--	---

This predicate returns ‘true’ if  $\langle tl\ var.\rangle$  is ‘empty’ i.e., doesn’t contain any tokens.

<code>\tl_if_empty:NTF *</code> <code>\tl_if_empty:cTF *</code>	<code>\tl_if_empty:NTF &lt;tl var.&gt; {\langle true code\rangle} {\langle false code\rangle}</code>
--	--

Execute  $\langle true\ code\rangle$  if  $\langle tl\ var.\rangle$  is empty and  $\langle false\ code\rangle$  if it contains any tokens.

<code>\tl_if_eq_p:NN *</code> <code>\tl_if_eq_p:cN *</code> <code>\tl_if_eq_p:Nc *</code> <code>\tl_if_eq_p:cc *</code>	<code>\tl_if_eq_p:NN &lt;tl var.<sub>1</sub>&gt; &lt;tl var.<sub>2</sub>&gt;</code>
--	---

Predicate function which returns ‘true’ if the two token list variables are identical and ‘false’ otherwise.

<code>\tl_if_eq:NNTF *</code> <code>\tl_if_eq:cNTF *</code> <code>\tl_if_eq:NcTF *</code> <code>\tl_if_eq:ccTF *</code>	<code>\tl_if_eq:NNTF &lt;tl var.<sub>1</sub>&gt; &lt;tl var.<sub>2</sub>&gt; {\langle true code\rangle} {\langle false code\rangle}</code>
--	--

Execute  $\langle true\ code\rangle$  if  $\langle tl\ var.<sub>1</sub>\rangle$  holds the same token list as  $\langle tl\ var.<sub>2</sub>\rangle$  and  $\langle false\ code\rangle$  otherwise.

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

Execute  $\langle true\ code\rangle$  if  $\langle token\ list\rangle$  doesn’t contain any tokens and  $\langle false\ code\rangle$  otherwise.

<code>\tl_if_eq:nnTF</code>	<code>\tl_if_eq:nnTF &lt;token list1&gt; {\langle token list2\rangle} {\langle true code\rangle} {\langle false code\rangle}</code>
-----------------------------	---

Tests if  $\langle token\ list1\rangle$  and  $\langle token\ list2\rangle$  both in respect of character codes and category codes. Either the  $\langle true\ code\rangle$  or  $\langle false\ code\rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

<code>\tl_if_blank_p:n *</code>
<code>\tl_if_blank:nTF *</code>
<code>\tl_if_blank_p:V *</code>
<code>\tl_if_blank_p:o *</code>
<code>\tl_if_blank:VTF *</code>
<code>\tl_if_blank:oTF *</code>

`\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}`

Execute *<true code>* if *<token list>* is blank meaning that it is either empty or contains only blank spaces.

<code>\tl_if_single_p:n *</code>
<code>\tl_if_single:nTF *</code>
<code>\tl_if_single_p:N *</code>
<code>\tl_if_single:NTF *</code>

`\tl_if_single:NTF {<tl var.>} {<true code>} {<false code>}`

`\tl_if_single:nTF {<token list>} {<true code>} {<false code>}`

Conditional returning true if the token list or the contents of the *tl var.* consists of a single token only.

Note that an input of ‘space’<sup>6</sup> returns *<true>* from this function.

<code>\tl_to_lowercase:n</code>
<code>\tl_to_uppercase:n</code>

`\tl_to_lowercase:n {<token list>}`

`\tl_to_lowercase:n` converts all tokens in *<token list>* to their lower case representation. Similar for `\tl_to_uppercase:n`.

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\lowercase` and `\uppercase` renamed.

## 44 Working with the contents of token lists

<code>\tl_map_function:nN *</code>
<code>\tl_map_function:NN</code>
<code>\tl_map_function:cN</code>

`\tl_map_function:nN {<token list>} <function>`

`\tl_map_function:NN <tl var.> <function>`

Runs through all elements in a *<token list>* from left to right and places *<function>* in front of each element. As this function will also pick up elements in brace groups, the element is returned with braces and hence *<function>* should be a function with a `:n` suffix even though it may very well only deal with a single token.

This function uses a purely expandable loop function and will stay so as long as *<function>* is expandable too.

---

<sup>6</sup>But remember any number of consecutive spaces are read as a single space by T<sub>E</sub>X.

<code>\tl_map_inline:nn</code>
<code>\tl_map_inline:Nn</code>
<code>\tl_map_inline:cn</code>

<code>\tl_map_inline:nn</code> $\{\langle token\ list\rangle\}$ $\{\langle inline\ function\rangle\}$
<code>\tl_map_inline:Nn</code> $\langle tl\ var.\rangle$ $\{\langle inline\ function\rangle\}$

Allows a syntax like `\tl_map_inline:nn`  $\{\langle token\ list\rangle\}$   $\{\backslash token\_to\_str:N\ \#\#1\}$ . This renders it non-expandable though. Remember to double the #s for each level.

<code>\tl_map_variable:nNn</code>
<code>\tl_map_variable:NNn</code>
<code>\tl_map_variable:cNn</code>

<code>\tl_map_variable:nNn</code> $\{\langle token\ list\rangle\}$ $\langle temp\rangle$ $\{\langle action\rangle\}$
<code>\tl_map_variable:NNn</code> $\langle tl\ var.\rangle$ $\langle temp\rangle$ $\{\langle action\rangle\}$

Assigns  $\langle temp\rangle$  to each element on  $\langle token\ list\rangle$  and executes  $\langle action\rangle$ . As there is an assignment in this process it is not expandable.

**T<sub>E</sub>Xhackers note:** This is the L<sup>A</sup>T<sub>E</sub>X2 function `\@tfor` but with a more sane syntax. Also it works by tail recursion and so is faster as lists grow longer.

<code>\tl_map_break:</code>
-----------------------------

`\tl_map_break:`

For breaking out of a loop. Must not be nested inside a primitive `\if` structure.

<code>\tl_reverse:n</code>
<code>\tl_reverse:V</code>
<code>\tl_reverse:o</code>
<code>\tl_reverse:N</code>

<code>\tl_reverse:n</code> $\{\langle token_1\rangle\langle token_2\rangle\ldots\langle token_n\rangle\}$
<code>\tl_reverse:N</code> $\langle tl\ var.\rangle$

Reverse the token list (or the token list in the  $\langle tl\ var.\rangle$ ) to result in  $\langle token_n\rangle\ldots\langle token_2\rangle\langle token_1\rangle$ . Note that spaces in this token list are gobbled in the process.

Note also that braces are lost in the process of reversing a  $\langle tl\ var.\rangle$ . That is, `\tl_set:Nn \l_tmpa_tl {a{bcd}e} \tl_reverse:N \l_tmpa_tl` will result in `ebcda`. This behaviour is probably more of a bug than a feature.

<code>\tl_elt_count:n *</code>
<code>\tl_elt_count:V *</code>
<code>\tl_elt_count:o *</code>
<code>\tl_elt_count:N *</code>

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

Returns the number of elements in the token list. Brace groups encountered count as one element. Note that spaces in this token list are gobbled in the process.

## 45 Variables and constants

<code>\c_job_name_tl</code>
-----------------------------

Constant that gets the ‘job name’ assigned when T<sub>E</sub>X starts.

**T<sub>E</sub>Xhackers note:** This is the new name for the primitive `\jobname`. It is a constant that is set by T<sub>E</sub>X and should not be overwritten by the package.

<code>\c_empty_tl</code>	Constant that is always empty.
--------------------------	--------------------------------

**T<sub>E</sub>Xhackers note:** This was named `\@empty` in L<sup>A</sup>T<sub>E</sub>X2 and `\empty` in plain T<sub>E</sub>X.

<code>\c_space_tl</code>	A space token contained in a token list (compare this with <code>\char_space_token</code> ). For use where an explicit space is required.
--------------------------	---

<code>\l_tmpa_tl</code> <code>\l_tmpb_tl</code> <code>\g_tmpa_tl</code> <code>\g_tmpb_tl</code>	Scratch register for immediate use. They are not used by conditionals or predicate functions. However, it is important to note that you should never rely on such scratch variables unless you fully control the code used between setting them and retrieving their value. Calling code from other modules, or worse allowing arbitrary user input to interfere might result in them not containing what you expect. In that is the case you better define your own scratch variables that are tight to your code by giving them suitable names.
--	---

<code>\l_tl_replace_tl</code>	Internal register used in the replace functions.
-------------------------------	--

<code>\l_kernel_testa_tl</code> <code>\l_kernel_testb_tl</code>	Registers used for conditional processing if the engine doesn't support arbitrary string comparison. Not for use outside the kernel code!
--	---

<code>\l_kernel_tmpa_tl</code> <code>\l_kernel_tmpb_tl</code>	Scratch registers reserved for other places in kernel code. Not for use outside the kernel code!
--	--

<code>\g_tl_inline_level_int</code>	Internal register used in the inline map functions.
-------------------------------------	---

## 46 Searching for and replacing tokens

<code>\tl_if_in:NnTF</code> <code>\tl_if_in:cnTF</code> <code>\tl_if_in:nnTF</code> <code>\tl_if_in:VnTF</code> <code>\tl_if_in:onTF</code>	<code>\tl_if_in:NnTF</code> <i>&lt;tl var.&gt;</i> <i>{&lt;item&gt;}</i> <i>{&lt;true code&gt;}</i> <i>{&lt;false code&gt;}</i>
---	---

Function that tests if  $\langle item \rangle$  is in  $\langle tl\ var. \rangle$ . Depending on the result either  $\langle true\ code \rangle$  or  $\langle false\ code \rangle$  is executed. Note that  $\langle item \rangle$  cannot contain brace groups nor  $\#_6$  tokens.

<code>\tl_replace_in:Nnn</code>
<code>\tl_replace_in:cnn</code>
<code>\tl_greplace_in:Nnn</code>
<code>\tl_greplace_in:cnn</code>

`\tl_replace_in:Nnn  $\langle tl\ var. \rangle$   $\{\langle item_1 \rangle\}$   $\{\langle item_2 \rangle\}$` 

Replaces the leftmost occurrence of  $\langle item_1 \rangle$  in  $\langle tl\ var. \rangle$  with  $\langle item_2 \rangle$  if present, otherwise the  $\langle tl\ var. \rangle$  is left untouched. Note that  $\langle item_1 \rangle$  cannot contain brace groups nor  $\#_6$  tokens, and  $\langle item_2 \rangle$  cannot contain  $\#_6$  tokens.

<code>\tl_replace_all_in:Nnn</code>
<code>\tl_replace_all_in:cnn</code>
<code>\tl_greplace_all_in:Nnn</code>
<code>\tl_greplace_all_in:cnn</code>

`\tl_replace_all_in:Nnn  $\langle tl\ var. \rangle$   $\{\langle item_1 \rangle\}$   $\{\langle item_2 \rangle\}$` 

Replaces *all* occurrences of  $\langle item_1 \rangle$  in  $\langle tl\ var. \rangle$  with  $\langle item_2 \rangle$ . Note that  $\langle item_1 \rangle$  cannot contain brace groups nor  $\#_6$  tokens, and  $\langle item_2 \rangle$  cannot contain  $\#_6$  tokens.

<code>\tl_remove_in:Nn</code>
<code>\tl_remove_in:cn</code>
<code>\tl_gremove_in:Nn</code>
<code>\tl_gremove_in:cn</code>

`\tl_remove_in:Nn  $\langle tl\ var. \rangle$   $\{\langle item \rangle\}$` 

Removes the leftmost occurrence of  $\langle item \rangle$  from  $\langle tl\ var. \rangle$  if present. Note that  $\langle item \rangle$  cannot contain brace groups nor  $\#_6$  tokens.

<code>\tl_remove_all_in:Nn</code>
<code>\tl_remove_all_in:cn</code>
<code>\tl_gremove_all_in:Nn</code>
<code>\tl_gremove_all_in:cn</code>

`\tl_remove_all_in:Nn  $\langle tl\ var. \rangle$   $\{\langle item \rangle\}$` 

Removes *all* occurrences of  $\langle item \rangle$  from  $\langle tl\ var. \rangle$ . Note that  $\langle item \rangle$  cannot contain brace groups nor  $\#_6$  tokens.

## 47 Heads or tails?

Here are some functions for grabbing either the head or tail of a list and perform some tests on it.

<code>\tl_head:n</code>	<code>*</code>	
<code>\tl_head:V</code>	<code>*</code>	
<code>\tl_head:v</code>	<code>*</code>	
<code>\tl_tail:n</code>	<code>*</code>	
<code>\tl_tail:V</code>	<code>*</code>	
<code>\tl_tail:v</code>	<code>*</code>	
<code>\tl_tail:f</code>	<code>*</code>	
<code>\tl_head_i:n</code>	<code>*</code>	
<code>\tl_head_iii:n</code>	<code>*</code>	
<code>\tl_head_iii:f</code>	<code>*</code>	
<code>\tl_head:w</code>	<code>*</code>	
<code>\tl_tail:w</code>	<code>*</code>	
<code>\tl_head_i:w</code>	<code>*</code>	<code>\tl_head:n { &lt;token<sub>1</sub>&gt;&lt;token<sub>2</sub>&gt;...&lt;token<sub>k</sub>&gt; }</code>
<code>\tl_head_iii:w</code>	<code>*</code>	<code>\tl_tail:n { &lt;token<sub>1</sub>&gt;&lt;token<sub>2</sub>&gt;...&lt;token<sub>k</sub>&gt; }</code>
		<code>\tl_head:w &lt;token<sub>1</sub>&gt;&lt;token<sub>2</sub>&gt;...&lt;token<sub>k</sub>&gt; \q_stop</code>

These functions return either the head or the tail from a list of tokens, thus in the above example `\tl_head:n` would return `<token1>` and `\tl_tail:n` would return `<token2>...<tokenk>`. `\tl_head_iii:n` returns the first three tokens. The `:w` versions require some care as they expect the token list to be delimited by `\q_stop`.

**TeXhackers note:** These are the Lisp functions `car` and `cdr` but with L<sup>A</sup>T<sub>E</sub>X3 names.

<code>\tl_if_head_eq_meaning_p:nN</code>	<code>*</code>	
<code>\tl_if_head_eq_meaning:nNTF</code>	<code>*</code>	<code>\tl_if_head_eq_meaning:nNTF {&lt;token list&gt;} &lt;token&gt;</code>
		<code>{&lt;true&gt;} {&lt;false&gt;}</code>

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. The `meaning` version compares the two tokens with `\if_meaning:w`.

<code>\tl_if_head_eq_charcode_p:nN</code>	<code>*</code>	
<code>\tl_if_head_eq_charcode_p:fN</code>	<code>*</code>	
<code>\tl_if_head_eq_charcode:nNTF</code>	<code>*</code>	
<code>\tl_if_head_eq_charcode:fNTF</code>	<code>*</code>	<code>\tl_if_head_eq_charcode:nNTF {&lt;token list&gt;} &lt;token&gt;</code>
		<code>{&lt;true&gt;} {&lt;false&gt;}</code>

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. The `meaning` version compares the two tokens with `\if_charcode:w` but it prevents expansion of them. If you want them to expand, you can use an `f` type expansion first (define `\tl_if_head_eq_charcode:fNTF` or similar).

<code>\tl_if_head_eq_catcode_p:nN</code>	<code>*</code>	
<code>\tl_if_head_eq_catcode:nNTF</code>	<code>*</code>	<code>\tl_if_head_eq_catcode:nNTF {&lt;token list&gt;} &lt;token&gt;</code>
		<code>{&lt;true&gt;} {&lt;false&gt;}</code>

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. This version uses `\if_catcode:w` for the test but is otherwise identical to the `charcode` version.

## Part XI

# The l3toks package

## Token Registers

There is a second form beside token list variables in which L<sup>A</sup>T<sub>E</sub>X3 stores token lists, namely the internal T<sub>E</sub>X token registers. Functions dealing with these registers got the prefix `\toks_`. Unlike token list variables we have an accessing function as one can see below.

The main difference between  $\langle toks \rangle$  (token registers) and  $\langle tl var. \rangle$  (token list variable) is their behavior regarding expansion. While  $\langle tl vars \rangle$  expand fully (i.e., until only unexpandable tokens are left) inside an argument that is subject to expansion (i.e., denoted by  $\mathbf{x}$ )  $\langle toks \rangle$ 's expand always only up to one level, i.e., passing their contents without further expansion.

There are fewer restrictions on the contents of a token register over a token list variable. So while  $\langle token list \rangle$  is used to describe the contents of both of these, bear in mind that slightly different lists of tokens are allowed in each case. The best (only?) example is that a  $\langle toks \rangle$  can contain the `#` character (i.e., characters of catcode 6), whereas a  $\langle tl var. \rangle$  will require its input to be sanitised before that is possible.

If you're not sure which to use between a  $\langle tl var. \rangle$  or a  $\langle toks \rangle$ , consider what data you're trying to hold. If you're dealing with function parameters involving `#`, or building some sort of data structure then you probably want a  $\langle toks \rangle$  (e.g., `l3prop` uses  $\langle toks \rangle$  to store its property lists).

If you're storing ad-hoc data for later use (possibly from direct user input) then usually a  $\langle tl var. \rangle$  will be what you want.

## 48 Allocation and use

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

`\toks_new:N \langle toks \rangle`

Defines  $\langle toks \rangle$  to be a new token list register.

**T<sub>E</sub>Xhackers note:** This is the L<sup>A</sup>T<sub>E</sub>X3 allocation for what was called `\newtoks` in plain T<sub>E</sub>X.

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

`\toks_use:N \langle toks \rangle`

Accesses the contents of  $\langle toks \rangle$ . Contrary to token list variables  $\langle toks \rangle$  can't be access simply by calling them directly.

**T<sub>E</sub>Xhackers note:** Something like `\the <toks>`.

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

`\toks_set:Nn <toks> {\<token list>}`

Defines `<toks>` to hold the token list `<token list>`.

**T<sub>E</sub>Xhackers note:** `\toks_set:Nn` could have been specified in plain T<sub>E</sub>X by `<toks> = {\<token list>}` but all other functions have no counterpart in plain T<sub>E</sub>X.

<code>\toks_gset:Nn</code>
<code>\toks_gset:NV</code>
<code>\toks_gset:No</code>
<code>\toks_gset:Nx</code>
<code>\toks_gset:cn</code>
<code>\toks_gset:cV</code>
<code>\toks_gset:co</code>
<code>\toks_gset:cx</code>

`\toks_gset:Nn <toks> {\<token list>}`

Defines `<toks>` to globally hold the token list `<token list>`.

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

`\toks_set_eq:NN <toks12`

Set `<toks1>` to the value of `<toks2>`. Don't try to use `\toks_set:Nn` for this purpose if the second argument is also a token register.

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

`\toks_gset_eq:NN <toks1> <toks2>`

The `<toks1>` globally set to the value of `<toks2>`. Don't try to use `\toks_gset:Nn` for this purpose if the second argument is also a token register.

<code>\toks_clear:N</code>
<code>\toks_clear:c</code>
<code>\toks_gclear:N</code>
<code>\toks_gclear:c</code>

`\toks_clear:N <toks>`

The  $\langle toks \rangle$  is locally or globally cleared.

<code>\toks_use_clear:N</code>
<code>\toks_use_clear:c</code>
<code>\toks_use_gclear:N</code>
<code>\toks_use_gclear:c</code>

`\toks_use_clear:N <toks>`

Accesses the contents of  $\langle toks \rangle$  and clears (locally or globally) it afterwards. Actually the clearing operation is done in a way that does not prohibit the access of the following tokens in the input stream with functions stored in the token register. In other words this function is not exactly the same as calling `\toks_use:N <toks> \toks_clear:N <toks>` in sequence.

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

`\toks_show:N <toks>`

Displays the contents of  $\langle toks \rangle$  in the terminal output and log file. # signs in the  $\langle toks \rangle$  will be shown doubled.

**TpXhackers note:** Something like `\showthe <toks>`.

## 49 Adding to the contents of token registers

<code>\toks_put_left:Nn</code>
<code>\toks_put_left:NV</code>
<code>\toks_put_left:No</code>
<code>\toks_put_left:Nx</code>
<code>\toks_put_left:cn</code>
<code>\toks_put_left:cV</code>
<code>\toks_put_left:co</code>

`\toks_put_left:Nn <toks> {<token list>}`

These functions will append  $\langle token list \rangle$  to the left of  $\langle toks \rangle$ . Assignment is done locally. If possible append to the right since this operation is faster.

<code>\toks_gput_left:Nn</code>
<code>\toks_gput_left:NV</code>
<code>\toks_gput_left:No</code>
<code>\toks_gput_left:Nx</code>
<code>\toks_gput_left:cn</code>
<code>\toks_gput_left:cV</code>
<code>\toks_gput_left:co</code>

`\toks_gput_left:Nn <toks> {<token list>}`

These functions will append *<token list>* to the left of *<toks>*. Assignment is done globally. If possible append to the right since this operation is faster.

<code>\toks_put_right:Nn</code>
<code>\toks_put_right:NV</code>
<code>\toks_put_right:No</code>
<code>\toks_put_right:Nx</code>
<code>\toks_put_right:cV</code>
<code>\toks_put_right:cn</code>
<code>\toks_put_right:co</code>

`\toks_put_right:Nn <toks> {<token list>}`

These functions will append *<token list>* to the right of *<toks>*. Assignment is done locally.

<code>\toks_put_right:Nf</code>
---------------------------------

`\toks_put_right:Nf <toks> {<token list>}`

Variant of the above. `:Nf` is used by `template.dtx` and will perhaps be moved to that package.

<code>\toks_gput_right:Nn</code>
<code>\toks_gput_right:NV</code>
<code>\toks_gput_right:No</code>
<code>\toks_gput_right:Nx</code>
<code>\toks_gput_right:cn</code>
<code>\toks_gput_right:cV</code>
<code>\toks_gput_right:co</code>

`\toks_gput_right:Nn <toks> {<token list>}`

These functions will append *<token list>* to the right of *<toks>*. Assignment is done globally.

## 50 Predicates and conditionals

<code>\toks_if_empty_p:N *</code>
<code>\toks_if_empty:N<math>\overline{TF}</math> *</code>
<code>\toks_if_empty_p:c *</code>
<code>\toks_if_empty:c<math>\overline{TF}</math> *</code>

`\toks_if_empty:N $\overline{TF}$  <toks> {<true code>} {<false code>}`

Expandable test for whether *<toks>* is empty.

<code>\toks_if_eq:NNTF</code>	<code>*</code>
<code>\toks_if_eq:NcTF</code>	<code>*</code>
<code>\toks_if_eq:cNTF</code>	<code>*</code>
<code>\toks_if_eq:ccTF</code>	<code>*</code>
<code>\toks_if_eq_p:NN</code>	<code>*</code>
<code>\toks_if_eq_p:cN</code>	<code>*</code>
<code>\toks_if_eq_p:Nc</code>	<code>*</code>
<code>\toks_if_eq_p:cc</code>	<code>*</code>

`\toks_if_eq:NNTF <toks1> <toks2> {<true code>} {<false code>}`

Expandably tests if  $\langle toks_1 \rangle$  and  $\langle toks_2 \rangle$  are equal.

## 51 Variable and constants

<code>\c_empty_toks</code>
----------------------------

Constant that is always empty.

<code>\l_tmpa_toks</code>
<code>\l_tmpb_toks</code>
<code>\l_tmpc_toks</code>
<code>\g_tmpa_toks</code>
<code>\g_tmpb_toks</code>
<code>\g_tmpc_toks</code>

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

<code>\l_tl_replace_toks</code>
---------------------------------

A placeholder for contents of functions replacing contents of strings.

## Part XII

# The l3seq package

## Sequences

L<sup>A</sup>T<sub>E</sub>X3 implements a data type called ‘sequences’. These are special token lists that can be accessed via special function on the ‘left’. Appending tokens is possible at both ends. Appended token lists can be accessed only as a union. The token lists that form the individual items of a sequence might contain any tokens except two internal functions that are used to structure sequences (see section internal functions below). It is also possible to map functions on such sequences so that they are executed for every item on the sequence.

All functions that return items from a sequence in some  $\langle tl\ var.\rangle$  assume that the  $\langle tl\ var.\rangle$  is local. See remarks below if you need a global returned value.

The defined functions are not orthogonal in the sense that every possible variation possible is actually available. If you need a new variant use the expansion functions described in the package `l3expan` to build it.

Adding items to the left of a sequence can currently be done with either something like `\seq_put_left:Nn` or with a “stack” function like `\seq_push:Nn` which has the same effect. Maybe one should therefore remove the “left” functions totally.

## 52 Functions for creating/initialising sequences

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

Defines  $\langle sequence \rangle$  to be a variable of type `seq`.

<code>\seq_clear:N</code> <code>\seq_clear:c</code> <code>\seq_gclear:N</code> <code>\seq_gclear:c</code>	<code>\seq_clear:N</code> $\langle sequence \rangle$
--	--

These functions locally or globally clear  $\langle sequence \rangle$ .

<code>\seq_clear_new:N</code> <code>\seq_clear_new:c</code> <code>\seq_gclear_new:N</code> <code>\seq_gclear_new:c</code>	<code>\seq_clear_new:N</code> $\langle sequence \rangle$
--	--

These functions locally or globally clear  $\langle sequence \rangle$  if it exists or otherwise allocates it.

<code>\seq_set_eq:NN</code> <code>\seq_set_eq:cN</code> <code>\seq_set_eq:Nc</code> <code>\seq_set_eq:cc</code>	<code>\seq_set_eq:NN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$
--	---

Function that locally makes  $\langle seq_1 \rangle$  identical to  $\langle seq_2 \rangle$ .

<code>\seq_gset_eq:NN</code> <code>\seq_gset_eq:cN</code> <code>\seq_gset_eq:Nc</code> <code>\seq_gset_eq:cc</code>	<code>\seq_gset_eq:NN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$
--	--

Function that globally makes  $\langle seq_1 \rangle$  identical to  $\langle seq_2 \rangle$ .

<code>\seq_concat:NNN</code>	
<code>\seq_concat:ccc</code>	
<code>\seq_gconcat:NNN</code>	<code>\seq_concat:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle seq_3 \rangle$
<code>\seq_gconcat:ccc</code>	<code>\seq_gconcat:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle seq_3 \rangle$

Function that concatenates  $\langle seq_2 \rangle$  and  $\langle seq_3 \rangle$  and locally or globally assigns the result to  $\langle seq_1 \rangle$ .

## 53 Adding data to sequences

<code>\seq_put_left:Nn</code>	
<code>\seq_put_left:NV</code>	
<code>\seq_put_left:No</code>	
<code>\seq_put_left:Nx</code>	
<code>\seq_put_left:cn</code>	
<code>\seq_put_left:cV</code>	
<code>\seq_put_left:co</code>	<code>\seq_put_left:Nn</code> $\langle sequence \rangle$ $\langle token list \rangle$

Locally appends  $\langle token list \rangle$  as a single item to the left of  $\langle sequence \rangle$ .  $\langle token list \rangle$  might get expanded before appending according to the variant.

<code>\seq_put_right:Nn</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:co</code>	<code>\seq_put_right:Nn</code> $\langle sequence \rangle$ $\langle token list \rangle$

Locally appends  $\langle token list \rangle$  as a single item to the right of  $\langle sequence \rangle$ .  $\langle token list \rangle$  might get expanded before appending according to the variant.

<code>\seq_gput_left:Nn</code>	
<code>\seq_gput_left:NV</code>	
<code>\seq_gput_left:No</code>	
<code>\seq_gput_left:Nx</code>	
<code>\seq_gput_left:cn</code>	
<code>\seq_gput_left:cV</code>	
<code>\seq_gput_left:co</code>	<code>\seq_gput_left:Nn</code> $\langle sequence \rangle$ $\langle token list \rangle$

Globally appends  $\langle token list \rangle$  as a single item to the left of  $\langle sequence \rangle$ .

<code>\seq_gput_right:Nn</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:co</code>

`\seq_gput_right:Nn`  $\langle sequence \rangle$   $\langle token list \rangle$

Globally appends  $\langle token list \rangle$  as a single item to the right of  $\langle sequence \rangle$ .

## 54 Working with sequences

<code>\seq_get:NN</code>
<code>\seq_get:cN</code>

`\seq_get:NN`  $\langle sequence \rangle$   $\langle tl var. \rangle$

Functions that locally assign the left-most item of  $\langle sequence \rangle$  to the token list variable  $\langle tl var. \rangle$ . Item is not removed from  $\langle sequence \rangle$ ! If you need a global return value you need to code something like this:

```
\seq_get:NN  $\langle sequence \rangle$  \l_tmpa_tl
\tl_gset_eq:NN  $\langle global tl var. \rangle$  \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

<code>\seq_map_variable:NNn</code>
<code>\seq_map_variable:cNn</code>

`\seq_map_variable:NNn`  $\langle sequence \rangle$   $\langle tl var. \rangle$  { $\langle code using tl var. \rangle$ }

Every element in  $\langle sequence \rangle$  is assigned to  $\langle tl var. \rangle$  and then  $\langle code using tl var. \rangle$  is executed. The operation is not expandable which means that it can't be used within write operations etc. However, this function can be nested which the others can't.

<code>\seq_map_function:NN</code>
<code>\seq_map_function:cN</code>

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

This function applies  $\langle function \rangle$  (which must be a function with one argument) to every item of  $\langle sequence \rangle$ .  $\langle function \rangle$  is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc.

In the current implementation the next functions are more efficient and should be preferred.

<code>\seq_map_inline:Nn</code> <code>\seq_map_inline:cn</code>
--

`\seq_map_inline:Nn`  $\langle sequence \rangle$   $\{\langle inline function \rangle\}$

Applies  $\langle inline function \rangle$  (which should be the direct coding for a function with one argument (i.e. use #1 as the place holder for this argument)) to every item of  $\langle sequence \rangle$ .  $\langle inline function \rangle$  is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc.

<code>\seq_map_break:</code> <code>\seq_map_break:n</code>
---

These functions are used to break out of a mapping function at the point of execution. (Please do not put ‘\q\_stop’ inside a  $\langle seq \rangle$  that uses these functions.)

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

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

Function that pauses the compilation and displays  $\langle seq \rangle$  in the terminal output and in the log file. (Usually used for diagnostic purposes.)

<code>\seq_display:N</code> <code>\seq_display:c</code>
--

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

As with `\seq_show:N` but pretty prints the output one line per element.

<code>\seq_remove_duplicates:N</code> <code>\seq_gremove_duplicates:N</code>
---

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

Function that removes any duplicate entries in  $\langle seq \rangle$ .

## 55 Predicates and conditionals

<code>\seq_if_empty_p:N *</code> <code>\seq_if_empty_p:c *</code>
--

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

This predicate returns ‘true’ if  $\langle sequence \rangle$  is ‘empty’ i.e., doesn’t contain any items. Note that this is ‘false’ even if the  $\langle sequence \rangle$  only contains a single empty item.

<code>\seq_if_empty:NTF</code> <code>\seq_if_empty:cTF</code>
--

`\seq_if_empty:NTF`  $\langle sequence \rangle$   $\{\langle true code \rangle\}$   $\{\langle false code \rangle\}$

Set of conditionals that test whether or not a particular  $\langle sequence \rangle$  is empty and if so executes either  $\langle true code \rangle$  or  $\langle false code \rangle$ .

<code>\seq_if_in:NnTF</code>
<code>\seq_if_in:NVTF</code>
<code>\seq_if_in:cnTF</code>
<code>\seq_if_in:cVTF</code>
<code>\seq_if_in:coTF</code>
<code>\seq_if_in:cxTF</code>

`\seq_if_in:NnTF <sequence> {<item>} {<true code>} {<false code>}`

Functions that test if  $\langle item \rangle$  is in  $\langle sequence \rangle$ . Depending on the result either  $\langle true code \rangle$  or  $\langle false code \rangle$  is executed.

## 56 Internal functions

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

`\seq_if_empty_err:N <sequence>`

Signals an L<sup>A</sup>T<sub>E</sub>X3 error if  $\langle sequence \rangle$  is empty.

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

`\seq_pop_aux:nnNN <assign1> <assign2> <sequence> <tl var.>`

Function that assigns the left-most item of  $\langle sequence \rangle$  to  $\langle tl var. \rangle$  using  $\langle assign_1 \rangle$  and assigns the tail to  $\langle sequence \rangle$  using  $\langle assign_2 \rangle$ . This function could be used to implement a global return function.

<code>\seq_get_aux:w</code>
<code>\seq_pop_aux:w</code>
<code>\seq_put_aux:Nnn</code>
<code>\seq_put_aux:w</code>

Functions used to implement put and get operations. They are not for meant for direct use.

<code>\seq_elt:w</code>
<code>\seq_elt_end:</code>

Functions (usually used as constants) that separates items within a sequence. They might get special meaning during mapping operations and are not supposed to show up as tokens within an item appended to a sequence.

## 57 Functions for ‘Sequence Stacks’

Special sequences in L<sup>A</sup>T<sub>E</sub>X3 are ‘stacks’ with their usual operations of ‘push’, ‘pop’, and ‘top’. They are internally implemented as sequences and share some of the functions (like `\seq_new:N` etc.)

```

\seq_push:Nn
\seq_push:NV
\seq_push:No
\seq_push:cn
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:No
\seq_gpush:Nv
\seq_gpush:cn

```

```
\seq_push:Nn <stack> {<token list>}
```

Locally or globally pushes *<token list>* as a single item onto the *<stack>*.

```

\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN

```

```
\seq_pop:NN <stack> <tl var.>
```

Functions that assign the top item of *<stack>* to *<tl var.>* and removes it from *<stack>*!

```

\seq_top:NN
\seq_top:cN

```

```
\seq_top:NN <stack> <tl var.>
```

Functions that locally assign the top item of *<stack>* to the *<tl var.>*. Item is *not* removed from *<stack>*!

## Part XIII

# The l3clist package

## Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the sequence. This gives an ordered list which can then be utilised with the `\clist_map_function:NN` function. Comma lists cannot contain empty items, thus

```

\clist_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { }
\clist_if_empty:NTF \l_my_clist { true } { false }

```

will leave `true` in the input stream.

## 58 Functions for creating/initialising comma-lists

<code>\clist_new:N</code> <code>\clist_new:c</code>	<code>\clist_new:N &lt;comma list&gt;</code>
--	--

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

<code>\clist_set_eq:NN</code> <code>\clist_set_eq:cN</code> <code>\clist_set_eq:Nc</code> <code>\clist_set_eq:cc</code>	<code>\clist_set_eq:NN &lt;comma list1&gt; &lt;comma list2&gt;</code>
--	---

Sets the content of *<comma list1>* equal to that of *<comma list2>*. This assignment is restricted to the current  $\text{\TeX}$  group level.

<code>\clist_gset_eq:NN</code> <code>\clist_gset_eq:cN</code> <code>\clist_gset_eq:Nc</code> <code>\clist_gset_eq:cc</code>	<code>\clist_gset_eq:NN &lt;comma list1&gt; &lt;comma list2&gt;</code>
--	--

Sets the content of *<comma list1>* equal to that of *<comma list2>*. This assignment is global and so is not limited by the current  $\text{\TeX}$  group level.

<code>\clist_clear:N</code> <code>\clist_clear:c</code>	<code>\clist_clear:N &lt;comma list&gt;</code>
--	--

Clears all entries from the *<comma list>* within the scope of the current  $\text{\TeX}$  group.

<code>\clist_gclear:N</code> <code>\clist_gclear:c</code>	<code>\clist_gclear:N &lt;comma list&gt;</code>
--	---

Clears all entries from the *<comma list>* globally.

<code>\clist_clear_new:N</code> <code>\clist_clear_new:c</code> <code>\clist_gclear_new:N</code> <code>\clist_gclear_new:c</code>	<code>\clist_clear_new:N &lt;comma-list&gt;</code>
--	--

These functions locally or globally clear *<comma-list>* if it exists or otherwise allocates it.

## 59 Putting data in

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn &lt;comma list&gt; {\&lt;entry&gt;}</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>	

Adds  $\langle entry \rangle$  onto the left of the  $\langle comma list \rangle$ . The assignment is restricted to the current  $\text{T}_{\text{E}}\text{X}$  group.

<code>\clist_gput_left:Nn</code>	<code>\clist_gput_left:Nn &lt;comma list&gt; {\&lt;entry&gt;}</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>	

Adds  $\langle entry \rangle$  onto the left of the  $\langle comma list \rangle$ . The assignment is global.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn &lt;comma list&gt; {\&lt;entry&gt;}</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>	

Adds  $\langle entry \rangle$  onto the right of the  $\langle comma list \rangle$ . The assignment is restricted to the current  $\text{T}_{\text{E}}\text{X}$  group.

<code>\clist_gput_right:Nn</code>	<code>\clist_gput_right:Nn &lt;comma list&gt; {\&lt;entry&gt;}</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>	

Adds  $\langle entry \rangle$  onto the right of the  $\langle comma list \rangle$ . The assignment is global.

## 60 Getting data out

<code>\clist_use:N *</code> <code>\clist_use:c *</code>	<code>\clist_use:N &lt;comma list&gt;</code>
--	--

Recovers the content of a *<comma list>* and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. This function is intended mainly for use when a *<comma list>* is being saved to an auxiliary file.

<code>\clist_show:N</code> <code>\clist_show:c</code>	<code>\clist_show:N &lt;clist&gt;</code>
--	--

Function that pauses the compilation and displays *<clist>* in the terminal output and in the log file. (Usually used for diagnostic purposes.)

<code>\clist_display:N</code> <code>\clist_display:c</code>	<code>\clist_display:N &lt;comma list&gt;</code>
--	--

Displays the value of the *<comma list>* on the terminal.

<code>\clist_get:NN</code> <code>\clist_get:cN</code>	<code>\clist_get:NN &lt;comma-list&gt; &lt;tl var.&gt;</code>
--	---

Functions that locally assign the left-most item of *<comma-list>* to the token list variable *<tl var.>*. Item is not removed from *<comma-list>*! If you need a global return value you need to code something like this:

```
\clist_get:NN <comma-list> \l_tmpa_tl
\tl_gset_eq:NN <global tl var.> \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

## 61 Mapping functions

We provide three types of mapping functions, each with their own strengths. The `\clist_map_function:NN` is expandable whereas `\clist_map_inline:Nn` type uses `##1` as a placeholder for the current item in *<clist>*. Finally we have the `\clist_map_variable:NNn` type which uses a user-defined variable as placeholder. Both the `_inline` and `_variable` versions are nestable.

\clist_map_function:NN *	\clist_map_function:NN <comma list> <function>
\clist_map_function:Nc *	
\clist_map_function:cN *	
\clist_map_function:cc *	
\clist_map_function:nN *	
\clist_map_function:nc *	

Applies *<function>* to every *<entry>* stored in the *<comma list>*. The *<function>* will receive one argument for each iteration. The *<entries>* in the *<comma list>* are supplied to the *<function>* reading from the left to the right. These function may be nested.

\clist_map_inline:Nn	\clist_map_inline:Nn <comma list> {<inline function>}
\clist_map_inline:cn	
\clist_map_inline:nn	

Applies *<inline function>* to every *<entry>* stored within the *<comma list>*. The *<inline function>* should consist of code which will receive the *<entry>* as #1. One inline mapping can be nested inside another. The *<entries>* in the *<comma list>* are supplied to the *<function>* reading from the left to the right.

\clist_map_variable:NNn	\clist_map_variable:NNn <comma-list> <temp-var> {<action>}
\clist_map_variable:cNn	
\clist_map_variable:nNn	

Assigns *<temp-var>* to each element in *<clist>* and then executes *<action>* which should contain *<temp-var>*. As the operation performs an assignment, it is not expandable.

**T<sub>E</sub>Xhackers note:** These functions resemble the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> function `\@for` but does not borrow the somewhat strange syntax.

\clist_map_break: *	\clist_map_break:
---------------------	-------------------

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 low level T<sub>E</sub>X errors.

## 62 Predicates and conditionals

<code>\clist_if_empty_p:N *</code> <code>\clist_if_empty_p:c *</code> <code>\clist_if_empty:NTF *</code> <code>\clist_if_empty:cTF *</code>	<code>\clist_if_empty_p:N &lt;clist&gt;</code> <code>\clist_if_empty:NNTF &lt;clist&gt; {\true code} {\false code}</code>
--	--

Tests if the  $\langle comma list \rangle$  is empty (containing no items). The branching versions then leave either  $\langle true code \rangle$  or  $\langle false code \rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\clist_if_eq_p:NN *</code> <code>\clist_if_eq_p:Nc *</code> <code>\clist_if_eq_p:cN *</code> <code>\clist_if_eq_p:cc *</code> <code>\clist_if_eq:NNTF *</code> <code>\clist_if_eq:NcTF *</code> <code>\clist_if_eq:cNTF *</code> <code>\clist_if_eq:ccTF *</code>	<code>\clist_if_eq_p:NN {\clist_1} {\clist_2}</code> <code>\clist_if_eq:NNTF {\clist_1} {\clist_2} {\true code}</code> <code>\false code}</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. The branching versions then leave either  $\langle true code \rangle$  or  $\langle false code \rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\clist_if_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 &lt;clist&gt; {\entry} {\true code}</code> <code>\false code}</code>
--	---

Tests if the  $\langle entry \rangle$  is present in the  $\langle comma list \rangle$  as a discrete entry. The  $\langle entry \rangle$  cannot contain the tokens  $\{, \}$  or  $\#$  (assuming the usual  $\text{\TeX}$  category codes apply). Either the  $\langle true code \rangle$  or  $\langle false code \rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

## 63 Higher level functions

<code>\clist_concat:NNN</code> <code>\clist_concat:ccc</code>	<code>\clist_concat:NNN &lt;clist1&gt; &lt;clist2&gt; &lt;clist3&gt;</code>
--	---

Concatenates the content of  $\langle comma list2 \rangle$  and  $\langle comma list3 \rangle$  together and saves the

result in  $\langle comma list1 \rangle$ .  $\langle comma list2 \rangle$  will be placed at the left side of the new comma list. This operation is local to the current  $\text{T}_{\text{E}}\text{X}$  group and will remove any existing content in  $\langle comma list1 \rangle$ .

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

`\clist_gconcat:NNN  $\langle clist1 \rangle$   $\langle clist2 \rangle$   $\langle clist3 \rangle$` 

Concatenates the content of  $\langle comma list2 \rangle$  and  $\langle comma list3 \rangle$  together and saves the result in  $\langle comma list1 \rangle$ .  $\langle comma list2 \rangle$  will be placed at the left side of the new comma list. This operation is global and will remove any existing content in  $\langle comma list1 \rangle$ .

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

`\clist_remove_duplicates:N  $\langle comma list \rangle$` 

Removes duplicate entries from the  $\langle comma list \rangle$ , leaving left most entry in the  $\langle comma list \rangle$ . The removal is local to the current  $\text{T}_{\text{E}}\text{X}$  group.

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

`\clist_gremove_duplicates:N  $\langle comma list \rangle$` 

Removes duplicate entries from the  $\langle comma list \rangle$ , leaving left most entry in the  $\langle comma list \rangle$ . The removal is applied globally.

<code>\clist_remove_element:Nn</code>
---------------------------------------

`\clist_remove_element:Nn  $\langle comma list \rangle$   $\{ \langle entry \rangle \}$` 

Removes each occurrence of  $\langle entry \rangle$  from the  $\langle comma list \rangle$ , where  $\langle entry \rangle$  cannot contain the tokens  $\{, \}$  or  $\#$  (assuming normal  $\text{T}_{\text{E}}\text{X}$  category codes). The removal is local to the current  $\text{T}_{\text{E}}\text{X}$  group.

<code>\clist_gremove_element:Nn</code>
--

`\clist_gremove_element:Nn  $\langle comma list \rangle$   $\{ \langle entry \rangle \}$` 

Removes each occurrence of  $\langle entry \rangle$  from the  $\langle comma list \rangle$ , where  $\langle entry \rangle$  cannot contain the tokens  $\{, \}$  or  $\#$  (assuming normal  $\text{T}_{\text{E}}\text{X}$  category codes). The removal applied globally.

## 64 Functions for ‘comma-list stacks’

Special comma-lists in  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}3$  are ‘stacks’ with their usual operations of ‘push’, ‘pop’, and ‘top’. They are internally implemented as comma-lists and share some of the functions (like `\clist_new:N` etc.)

<code>\clist_push:Nn</code>
<code>\clist_push:NV</code>
<code>\clist_push:No</code>
<code>\clist_push:cn</code>
<code>\clist_gpush:Nn</code>
<code>\clist_gpush:NV</code>
<code>\clist_gpush:No</code>
<code>\clist_gpush:cn</code>

`\clist_push:Nn <stack> {<token list>}`

Locally or globally pushes  $\langle token\ list \rangle$  as a single item onto the  $\langle stack \rangle$ .  $\langle token\ list \rangle$  might get expanded before the operation.

<code>\clist_pop:NN</code>
<code>\clist_pop:cN</code>
<code>\clist_gpop:NN</code>
<code>\clist_gpop:cN</code>

`\clist_pop:NN <stack> <tl var.>`

Functions that assign the top item of  $\langle stack \rangle$  to the token list variable  $\langle tl\ var. \rangle$  and removes it from  $\langle stack \rangle$ !

<code>\clist_top:NN</code>
<code>\clist_top:cN</code>

`\clist_top:NN <stack> <tl var.>`

Functions that locally assign the top item of  $\langle stack \rangle$  to the token list variable  $\langle tl\ var. \rangle$ . Item is not removed from  $\langle stack \rangle$ !

## 65 Internal functions

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

`\clist_if_empty_err:N <comma-list>`

Signals an L<sup>A</sup>T<sub>E</sub>X3 error if  $\langle comma-list \rangle$  is empty.

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

`\clist_pop_aux:nnNN <assign1> <assign2> <comma-list> <tl var.>`

Function that assigns the left-most item of  $\langle comma-list \rangle$  to  $\langle tl\ var. \rangle$  using  $\langle assign_1 \rangle$  and assigns the tail to  $\langle comma-list \rangle$  using  $\langle assign_2 \rangle$ . This function could be used to implement a global return function.

<code>\clist_get_aux:w</code>
<code>\clist_pop_aux:w</code>
<code>\clist_pop_auxi:w</code>
<code>\clist_put_aux:NNnnNn</code>

Functions used to implement put and get operations. They are not for meant for direct use.

## Part XIV

# The l3prop package

## Property Lists

L<sup>A</sup>T<sub>E</sub>X3 implements a data structure called a ‘property list’ which allows arbitrary information to be stored and accessed using keywords rather than numerical indexing.

A property list might contain a set of keys such as `name`, `age`, and `ID`, which each have individual values that can be saved and retrieved.

## 66 Functions

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

`\prop_new:N <prop>`  
Defines `<prop>` to be a variable of type `<prop>`.

<code>\prop_clear:N</code>
<code>\prop_clear:c</code>
<code>\prop_gclear:N</code>
<code>\prop_gclear:c</code>

`\prop_clear:N <prop>`  
These functions locally or globally clear `<prop>`.

<code>\prop_put:Nnn</code>
<code>\prop_put:Nno</code>
<code>\prop_put:NnV</code>
<code>\prop_put:NVn</code>
<code>\prop_put:NVV</code>
<code>\prop_put:Nnx</code>
<code>\prop_put:cnn</code>
<code>\prop_put:cnx</code>
<code>\prop_gput:Nnn</code>
<code>\prop_gput:NVn</code>
<code>\prop_gput:Nno</code>
<code>\prop_gput:NnV</code>
<code>\prop_gput:Nnx</code>
<code>\prop_gput:cnn</code>
<code>\prop_gput:ccx</code>

`\prop_put:Nnn <prop> {<key>} {<token list>}`

Locally or globally associates `<token list>` with `<key>` in the `<prop>` `<prop>`. If `<key>` has already a meaning within `<prop>` this value is overwritten.

The  $\langle key \rangle$  must not contain unescaped # tokens but the  $\langle token list \rangle$  may.

<code>\prop_gput_if_new:Nnn</code>
------------------------------------

`\prop_gput_if_new:Nnn <prop> {<key>} {<token list>}`

Globally associates  $\langle token list \rangle$  with  $\langle key \rangle$  in the  $\langle prop \rangle$   $\langle prop \rangle$  but only if  $\langle key \rangle$  has so far no meaning within  $\langle prop \rangle$ . Silently ignored if  $\langle key \rangle$  is already set in the  $\langle prop \rangle$ .

<code>\prop_get:NnN</code>
<code>\prop_get:NVN</code>
<code>\prop_get:cnN</code>
<code>\prop_get:cVN</code>
<code>\prop_gget:NnN</code>
<code>\prop_gget:NVN</code>
<code>\prop_gget:cnN</code>
<code>\prop_gget:cVN</code>

`\prop_get:NnN <prop> {<key>} <tl var.>`

If  $\langle info \rangle$  is the information associated with  $\langle key \rangle$  in the  $\langle prop \rangle$   $\langle prop \rangle$  then the token list variable  $\langle tl var. \rangle$  gets  $\langle info \rangle$  assigned. Otherwise its value is the special quark `\q_no_value`. The assignment is done either locally or globally.

<code>\prop_set_eq:NN</code>
<code>\prop_set_eq:cN</code>
<code>\prop_set_eq:Nc</code>
<code>\prop_set_eq:cc</code>
<code>\prop_gset_eq:NN</code>
<code>\prop_gset_eq:cN</code>
<code>\prop_gset_eq:Nc</code>
<code>\prop_gset_eq:cc</code>

`\prop_set_eq:NN <prop1> <prop2>`

A fast assignment of  $\langle prop \rangle$ s.

<code>\prop_get_gdel:NnN</code>
---------------------------------

`\prop_get_gdel:NnN <prop> {<key>} <tl var.>`

Like `\prop_get:NnN` but additionally removes  $\langle key \rangle$  (and its  $\langle info \rangle$ ) from  $\langle prop \rangle$ .

<code>\prop_del:Nn</code>
<code>\prop_del:NV</code>
<code>\prop_gdel:Nn</code>
<code>\prop_gdel:NV</code>

`\prop_del:Nn <prop> {<key>}`

Locally or globally deletes  $\langle key \rangle$  and its  $\langle info \rangle$  from  $\langle prop \rangle$  if found. Otherwise does nothing.

<code>\prop_map_function:NN *</code>
<code>\prop_map_function:cN *</code>
<code>\prop_map_function:Nc *</code>
<code>\prop_map_function:cc *</code>

`\prop_map_function:NN <prop> <function>`

Maps  $\langle function \rangle$  which should be a function with two arguments ( $\langle key \rangle$  and  $\langle info \rangle$ ) over every  $\langle key \rangle$   $\langle info \rangle$  pair of  $\langle prop \rangle$ . Property lists do not have any intrinsic “order” when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

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

`\prop_map_inline:Nn  $\langle prop \rangle$  { $\langle inline function \rangle$ }`

Just like `\prop_map_function:NN` but with the function of two arguments supplied as inline code. Within  $\langle inline function \rangle$  refer to the arguments via #1 ( $\langle key \rangle$ ) and #2 ( $\langle info \rangle$ ). Nestable. Property lists do not have any intrinsic “order” when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

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

`\prop_map_inline:Nn  $\langle prop \rangle$  {  
...  $\langle break test \rangle$ :T {\prop_map_break:} }`

For breaking out of a loop. To be used inside TF-type functions as shown in the example above.

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

`\prop_show:N  $\langle prop \rangle$` 

Pauses the compilation and shows  $\langle prop \rangle$  on the terminal output and in the log file.

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

`\prop_display:N  $\langle prop \rangle$` 

As with `\prop_show:N` but pretty prints the output one line per property pair.

## 67 Predicates and conditionals

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

`\prop_if_empty_p:N  $\langle prop \rangle$  { $\langle true code \rangle$ } { $\langle false code \rangle$ }`

Predicates to test whether or not a particular  $\langle prop \rangle$  is empty.

<code>\prop_if_empty:N<math>\underline{TF}</math> *</code>
<code>\prop_if_empty:c<math>\underline{TF}</math> *</code>

`\prop_if_empty:N $\underline{TF}$   $\langle prop \rangle$  { $\langle true code \rangle$ } { $\langle false code \rangle$ }`

Set of conditionals that test whether or not a particular  $\langle prop \rangle$  is empty.

```

\prop_if_eq_p:NN *
\prop_if_eq_p:cN *
\prop_if_eq_p:Nc *
\prop_if_eq_p:cc *
\prop_if_eq:NN $\overline{TF}$  *
\prop_if_eq:cN $\overline{TF}$  *
\prop_if_eq:Nc $\overline{TF}$  *
\prop_if_eq:cc $\overline{TF}$  *

```

`\prop_if_eq:NMF  $\langle prop_1 \rangle$   $\langle prop_2 \rangle$  { $\langle false code \rangle$ }`

Execute  $\langle false code \rangle$  if  $\langle prop_1 \rangle$  doesn't hold the same token list as  $\langle prop_2 \rangle$ . Only expandable for new versions of pdfTeX.

```

\prop_if_in:Nn $\overline{TF}$ 
\prop_if_in:NV $\overline{TF}$ 
\prop_if_in:No $\overline{TF}$ 
\prop_if_in:cn $\overline{TF}$ 
\prop_if_in:cc $\overline{TF}$ 

```

`\prop_if_in:NnTF  $\langle prop \rangle$  { $\langle key \rangle$ } { $\langle true code \rangle$ } { $\langle false code \rangle$ }`

Tests if  $\langle key \rangle$  is used in  $\langle prop \rangle$  and then either executes  $\langle true code \rangle$  or  $\langle false code \rangle$ .

## 68 Internal functions

`\q_prop`

Quark used to delimit property lists internally.

```

\prop_put_aux:w
\prop_put_if_new_aux:w

```

Internal functions implementing the put operations.

```

\prop_get_aux:w
\prop_gget_aux:w
\prop_get_del_aux:w
\prop_del_aux:w

```

Internal functions implementing the get and delete operations.

`\prop_if_in_aux:w`

Internal function implementing the key test operation.

`\prop_map_function_aux:w`

Internal function implementing the map operations.

`\g_prop_inline_level_int`

Integer used in internal name for function used inside `\prop_map_inline:NN`.

`\prop_split_aux:Nnn`

`\prop_split_aux:Nnn  $\langle prop \rangle$   $\langle key \rangle$   $\langle cmd \rangle$`

Internal function that invokes  $\langle cmd \rangle$  with 3 arguments: 1st is the beginning of  $\langle prop \rangle$

before  $\langle key \rangle$ , 2nd is the value associated with  $\langle key \rangle$ , 3rd is the rest of  $\langle prop \rangle$  after  $\langle key \rangle$ . If there is no key  $\langle key \rangle$  in  $\langle prop \rangle$ , then the 2 arg is `\q_no_value` and the 3rd arg is empty; otherwise the 3rd argument has the two extra tokens  $\langle key \rangle$  `\q_no_value` at the end.

This function is used to implement various get operations.

## Part XV

# The l3io package

## Low-level file i/o

Reading and writing from file streams is handled in L<sup>A</sup>T<sub>E</sub>X3 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<sub>E</sub>X is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in L<sup>A</sup>T<sub>E</sub>X3. 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 c<sub>s</sub>name 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).

## 69 Opening and closing streams

<code>\iow_new:N</code> <code>\iow_new:c</code> <code>\ior_new:N</code> <code>\ior_new:c</code>	<code>\iow_new:N <math>\langle stream \rangle</math></code>
--	---

Reserves the name  $\langle stream \rangle$  for use in accessing a file stream. This operation does not open a raw T<sub>E</sub>X stream, which is handled internally using a pool and is should not be accessed directly by the programmer.

<code>\iow_open:Nn</code> <code>\iow_open:cn</code> <code>\ior_open:Nn</code> <code>\ior_open:cn</code>	<code>\iow_open:Nn <math>\langle stream \rangle</math> {<math>\langle file name \rangle</math>}</code> <code>\iow_open:cn <math>\langle stream \rangle</math> {<math>\langle file name \rangle</math>}</code>
--	--

Opens  $\langle file name \rangle$  for writing (`\iow_...`) or reading (`\ior_...`) using  $\langle stream \rangle$  as the

cname by which the file is accessed. If  $\langle stream \rangle$  was already open (for either writing or reading) it is closed before the new operation begins. The  $\langle stream \rangle$  is available for access immediately after issuing an `open` instruction. The  $\langle stream \rangle$  will remain allocated to  $\langle file name \rangle$  until a `close` instruction is given or at the end of the  $\text{\TeX}$  run.

Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive). As the total number of writing streams is limited, it may well be best to save material to be written to an intermediate storage format (for example a token list or toks), and to write the material in one ‘shot’ from this variable. In this way the file stream is only required for a limited time.

<code>\iow_close:N</code>	
<code>\iow_close:c</code>	
<code>\ior_close:N</code>	<code>\iow_close:N <math>\langle stream \rangle</math></code>
<code>\ior_close:c</code>	<code>\ior_close:N <math>\langle stream \rangle</math></code>

Closes  $\langle stream \rangle$ , freeing up one of the underlying  $\text{\TeX}$  streams for reuse. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers (the resources here are limited). The name of the  $\langle stream \rangle$  will be freed at this stage, to ensure that any further attempts to write to it result in an error.

<code>\iow_open_streams:</code>	
<code>\iow_open_streams:</code>	<code>\iow_open_streams:</code>

Displays a list of the file names associated with each open stream: intended for tracking down problems.

## 69.1 Writing to files

<code>\iow_now:Nx</code>	
<code>\iow_now:Nn</code>	<code>\iow_now:Nx <math>\langle stream \rangle</math> <math>\{\langle tokens \rangle\}</math></code>

`\iow_now:Nx` immediately writes the expansion of  $\langle tokens \rangle$  to the output  $\langle stream \rangle$ . If the  $\langle stream \rangle$  is not open output goes to the terminal. The variant `\iow_now:Nn` writes out  $\langle tokens \rangle$  without any further expansion.

**$\text{\TeX}$ hackers note:** These are the equivalent of  $\text{\TeX}$ ’s `\immediate\write` with and without expansion control.

<code>\iow_log:n</code>	
<code>\iow_log:x</code>	
<code>\iow_term:n</code>	
<code>\iow_term:x</code>	<code>\iow_log:x <math>\{\langle tokens \rangle\}</math></code>

These are dedicated functions which write to the log (transcript) file and the terminal,

respectively. They are equivalent to using `\iow_now:N(n/x)` to the streams `\c_iow_log_stream` and `\c_iow_term_stream`. The writing takes place immediately.

<code>\iow_now_buffer_safe:Nn</code>	<code>\iow_now_buffer_safe:Nn &lt;stream&gt; {&lt;tokens&gt;}</code>
<code>\iow_now_buffer_safe:Nx</code>	

Immediately write `<tokens>` expanded to `<stream>`, with every space converted into a newline. This means that the file can be read back without the danger that very long lines overflow T<sub>E</sub>X's buffer.

<code>\iow_now_when_avail:Nn</code>	<code>\iow_now_when_avail:Nn &lt;stream&gt; {&lt;tokens&gt;}</code>
<code>\iow_now_when_avail:cn</code>	
<code>\iow_now_when_avail:Nx</code>	
<code>\iow_now_when_avail:cx</code>	

If `<stream>` is open, writes the `<tokens>` to the `<stream>` in the same manner as `\iow_now:N(n/x)`. If the `<stream>` is not open, the `<tokens>` are simply thrown away.

<code>\iow_shipout:Nx</code>	<code>\iow_shipout:Nx &lt;stream&gt; {&lt;tokens&gt;}</code>
<code>\iow_shipout:Nn</code>	

Write `<tokens>` to `<stream>` at the point at which the current page is finished. The `<tokens>` are either written unexpanded (`\iow_shipout:Nn`) or expanded only at the point that the function is used (`\iow_shipout:Nx`), *i.e.* no expansion takes place when writing to the file.

<code>\iow_shipout_x:Nx</code>	<code>\iow_shipout_x:Nx &lt;stream&gt; {&lt;tokens&gt;}</code>
<code>\iow_shipout_x:Nn</code>	

Write `<tokens>` to `<stream>` at the point at which the current page is finished. The `<tokens>` are expanded at the time of writing in addition to any expansion at the time of use of the function. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

**T<sub>E</sub>Xhackers note:** These are the equivalent of T<sub>E</sub>X's `\write` with and without expansion control at point of use.

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

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 a `\iow_now:Nn` call).

<code>\iow_char:N *</code>	<code>\iow_char:N \&lt;char&gt;</code>
<code>\iow_char:N *</code>	<code>\iow_char:N \%</code>

Inserts `<char>` into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_stream { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in a `\iow_now:Nn` call).

## 69.2 Reading from files

<code>\ior_to:NN</code> <code>\ior_gto:NN</code>	<code>\ior_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 stream  $\langle stream \rangle$  and places the result locally or globally into the  $\langle token\ list\ variable \rangle$ . If  $\langle stream \rangle$  is not open, input is requested from the terminal.

<code>\ior_if_eof_p:N *</code> <code>\ior_if_eof:NTF *</code>	<code>\ior_if_eof:NTF</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.

## 70 Internal functions

<code>\iow_raw_new:N</code> <code>\iow_raw_new:c</code> <code>\ior_raw_new:N</code> <code>\ior_raw_new:c</code>	<code>\iow_raw_new:N</code> $\langle stream \rangle$
--	--

Creates a new low-level  $\langle stream \rangle$  for use in subsequent functions. As allocations are made using a pool *do not use this function!*

**T<sub>E</sub>Xhackers note:** This is L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\newwrite`.

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

Tests if the end of  $\langle stream \rangle$  has been reached during a reading operation.

**T<sub>E</sub>Xhackers note:** This is the primitive `\ifeof`.

## 71 Variables and constants

<code>\c_io_streams_tl</code>	A list of the positions available for stream allocation (numbers 0 to 15).
-------------------------------	--

<code>\c_iow_term_stream</code> <code>\c_ior_term_stream</code> <code>\c_iow_log_stream</code> <code>\c_ior_log_stream</code>	Fixed stream numbers for accessing to the log and the terminal. The reading and writing values are the same but are provided so that the meaning is clear.
--	--

<code>\g_iow_streams_prop</code> <code>\g_ior_streams_prop</code>	Allocation records for streams, linking the stream number to the current name being used for that stream.
--	---

<code>\g_iow_tmp_stream</code> <code>\g_ior_tmp_stream</code>	Used when creating new streams at the $\text{\TeX}$ level.
--	--

<code>\l_iow_stream_int</code> <code>\l_ior_stream_int</code>	Number of stream currently being allocated.
--	---

## Part XVI

# The `l3msg` package

## Communicating with the user

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example **error**, **warning** or **info**.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the

code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

## 72 Creating new messages

All messages have to be created before they can be used. Inside the message text, spaces are *not* ignored. A space where T<sub>E</sub>X would normally gobble one can be created using `\` , and a new line with `\\`. New lines may have ‘continuation’ text added by the output system.

<pre>\msg_new:nnnn \msg_new:nnn \msg_set:nnnn \msg_set:nnn</pre>	<pre>\msg_new:nnnn {&lt;module&gt;} {&lt;message&gt;} {&lt;text&gt;} {&lt;more text&gt;}</pre>
--	--

Creates new `<message>` for `<module>` to produce `<text>` initially and `<more text>` if requested by the user. `<text>` and `<more text>` can use up to four macro parameters (`#1` to `#4`), which are supplied by the message system. At the point where `<message>` is printed, the material supplied for `#1` to `#4` will be subject to an x-type expansion.

An error will be raised by the `new` functions if the message already exists: the `set` functions do not carry any checking. For messages defined using `\msg_new:nnn` or `\msg_set:nnn` L<sup>A</sup>T<sub>E</sub>X3 will supply a standard `<more text>` at the point the message is used, if this is required.

## 73 Message classes

Creating message output requires the message to be given a class.

<pre>\msg_class_new:nn \msg_class_set:nn</pre>	<pre>\msg_class_new:nn {&lt;class&gt;} {&lt;code&gt;}</pre>
--	---

Creates new `<class>` to output a message, using `<code>` to process the message text. The `<class>` should be a text value, while the `<code>` may be any arbitrary material.

The module defines several common message classes. The following describes the standard behaviour of each class if no redirection of the class or message is active. In all cases, the message may be issued supplying 0 to 4 arguments. The code will ensure that there are no errors if the number of arguments supplied here does not match the number in the definition of the message (although of course the sense of the message may be impaired).

```

\msg_fatal:nnxxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn

```

```

\msg_fatal:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Issues  $\langle module \rangle$  error message  $\langle name \rangle$ , passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions. After issuing a fatal error the T<sub>E</sub>X run will halt.

```

\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn

```

```

\msg_error:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Issues  $\langle module \rangle$  error message  $\langle name \rangle$ , passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions.

**T<sub>E</sub>Xhackers note:** The standard output here is similar to `\PackageError`.

```

\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn

```

```

\msg_warning:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Prints  $\langle module \rangle$  message  $\langle name \rangle$  to the terminal, passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions.

**T<sub>E</sub>Xhackers note:** The standard output here is similar to `\PackageWarningNoLine`.

```

\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn

```

```

\msg_info:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Prints  $\langle module \rangle$  message  $\langle name \rangle$  to the log, passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions.

**T<sub>E</sub>Xhackers note:** The standard output here is similar to `\PackageInfoNoLine`.

<code>\msg_log:nnxxxx</code>	
<code>\msg_log:nnxxx</code>	
<code>\msg_log:nnxx</code>	
<code>\msg_log:nnx</code>	
<code>\msg_log:nn</code>	<code>\msg_log:nnxxxx {&lt;module&gt;} {&lt;name&gt;} {&lt;arg one&gt;} {&lt;arg two&gt;} {&lt;arg three&gt;} {&lt;arg four&gt;}</code>

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-creating functions. No continuation text is added.

<code>\msg_trace:nnxxxx</code>	
<code>\msg_trace:nnxxx</code>	
<code>\msg_trace:nnxx</code>	
<code>\msg_trace:nnx</code>	
<code>\msg_trace:nn</code>	<code>\msg_trace:nnxxxx {&lt;module&gt;} {&lt;name&gt;} {&lt;arg one&gt;} {&lt;arg two&gt;} {&lt;arg three&gt;} {&lt;arg four&gt;}</code>

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-creating functions. No continuation text is added.

<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	<code>\msg_none:nnxxxx {&lt;module&gt;} {&lt;name&gt;} {&lt;arg one&gt;} {&lt;arg two&gt;} {&lt;arg three&gt;} {&lt;arg four&gt;}</code>

Does nothing: used for redirecting other message classes. Gobbles arguments given.

## 74 Redirecting messages

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {&lt;class one&gt;} {&lt;class two&gt;}</code>
-------------------------------------	---

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

<code>\msg_redirect_module:nnn</code>	<code>\msg_redirect_module:nnn {&lt;module&gt;} {&lt;class one&gt;} {&lt;class two&gt;}</code>
---------------------------------------	--

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection.

**TeXhackers note:** This function can be used to make some messages ‘silent’ by default. For example, all of the `trace` messages of *<module>* could be turned off with:

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

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

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages.

**TeXhackers note:** This function can be used to make a selected message ‘silent’ without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

## 75 Support functions for output

```
\msg_line_context: \msg_line_context:
```

Prints the text specified in `\c_msg_on_line_tl` followed by the current line in the current input file.

**TeXhackers note:** This is similar to the text added to messages by L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>’s `\PackageWarning` and `\PackageInfo`.

```
\msg_line_number: \msg_line_number:
```

Prints the current line number in the current input file.

```
\msg_newline:
\msg_two_newlines: \msg_newline:
```

Print one or two newlines with no continuation information.

## 76 Low-level functions

The low-level functions do not make assumptions about module names. The output functions here produce messages directly, and do not respond to redirection.

<code>\msg_generic_new:nnn</code> <code>\msg_generic_new:nn</code> <code>\msg_generic_set:nnn</code> <code>\msg_generic_set:nn</code>	<code>\msg_generic_new:nnn {&lt;name&gt;} {&lt;text&gt;} {&lt;more text&gt;}</code>
--	---

Creates new message  $\langle name \rangle$  to produce  $\langle text \rangle$  initially and  $\langle more text \rangle$  if requested by the user.  $\langle text \rangle$  and  $\langle more text \rangle$  can use up to four macro parameters (#1 to #4), which are supplied by the message system. Inside  $\langle text \rangle$  and  $\langle more text \rangle$  spaces are not ignored.

<code>\msg_direct_interrupt:xxxxx</code>	<code>\msg_direct_interrupt:xxxxx {&lt;first line&gt;} {&lt;text&gt;}</code> <code>{&lt;continuation&gt;} {&lt;last line&gt;} {&lt;more text&gt;}</code>
--	---

Executes a T<sub>E</sub>X error, interrupting compilation. The  $\langle first line \rangle$  is displayed followed by  $\langle text \rangle$  and the input prompt.  $\langle more text \rangle$  is displays if requested by the user. If  $\langle more text \rangle$  is blank a default is supplied. Each line of  $\langle text \rangle$  (broken with `\\`) begins with  $\langle continuation \rangle$  and finishes off with  $\langle last line \rangle$ .  $\langle last line \rangle$  has a period appended to it; do not add one yourself.

<code>\msg_direct_log:xx</code> <code>\msg_direct_term:xx</code>	<code>\msg_direct_log:xx {&lt;text&gt;} {&lt;continuation&gt;}</code>
---	---

Prints  $\langle text \rangle$  to either the log or terminal. New lines (broken with `\\`) start with  $\langle continuation \rangle$ .

## 77 Kernel-specific functions

<code>\msg_kernel_new:nnnn</code> <code>\msg_kernel_new:nnn</code> <code>\msg_kernel_set:nnnn</code> <code>\msg_kernel_set:nnn</code>	<code>\msg_kernel_new:nnnn {&lt;division&gt;} {&lt;name&gt;} {&lt;text&gt;}</code> <code>{&lt;more text&gt;}</code>
--	--

Creates new kernel message  $\langle name \rangle$  to produce  $\langle text \rangle$  initially and  $\langle more text \rangle$  if requested by the user.  $\langle text \rangle$  and  $\langle more text \rangle$  can use up to four macro parameters (#1 to #4), which are supplied by the message system. Kernel messages are divided into  $\langle divisions \rangle$ , roughly equivalent to the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> package names used.

<code>\msg_kernel_fatal:nnxxxx</code> <code>\msg_kernel_fatal:nnxxx</code> <code>\msg_kernel_fatal:nnxx</code> <code>\msg_kernel_fatal:nnx</code> <code>\msg_kernel_fatal:nn</code>	<code>\msg_kernel_fatal:nnxx {&lt;division&gt;} {&lt;name&gt;} {&lt;arg one&gt;}</code> <code>{&lt;arg two&gt;} {&lt;arg three&gt;} {&lt;arg four&gt;}</code>
---	--

Issues kernel error message  $\langle name \rangle$  for  $\langle division \rangle$ , passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions. The TeX run then halts. Cannot be redirected.

<pre>\msg_kernel_error:nnxxxx \msg_kernel_error:nnxxx \msg_kernel_error:nnxx \msg_kernel_error:nnx \msg_kernel_error:nn</pre>	<pre>\msg_kernel_error:nnxx {\langle division \rangle} {\langle name \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle}</pre>
---	--

Issues kernel error message  $\langle name \rangle$  for  $\langle division \rangle$ , passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions. Cannot be redirected.

<pre>\msg_kernel_warning:nnxxxx \msg_kernel_warning:nnxxx \msg_kernel_warning:nnxx \msg_kernel_warning:nnx \msg_kernel_warning:nn</pre>	<pre>\msg_kernel_warning:nnxx {\langle division \rangle} {\langle name \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle}</pre>
---	--

Prints kernel message  $\langle name \rangle$  for  $\langle division \rangle$  to the terminal, passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions.

<pre>\msg_kernel_info:nnxxxx \msg_kernel_info:nnxxx \msg_kernel_info:nnxx \msg_kernel_info:nnx \msg_kernel_info:nn</pre>	<pre>\msg_kernel_info:nnxx {\langle division \rangle} {\langle name \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle}</pre>
--	---

Prints kernel message  $\langle name \rangle$  for  $\langle division \rangle$  to the log, passing  $\langle arg one \rangle$  to  $\langle arg four \rangle$  to the text-creating functions.

<pre>\msg_kernel_bug:x</pre>	<pre>\msg_kernel_bug:x {\langle text \rangle}</pre>
------------------------------	---

Short-cut for ‘This is a LaTeX bug: check coding’ errors.

<pre>\msg_fatal_text:n</pre>	<pre>\msg_fatal_text:n {\langle package \rangle}</pre>
------------------------------	--

Prints ‘Fatal  $\langle package \rangle$  error’ for use in error messages.

## 78 Variables and constants

<pre>\c_msg_error_tl \c_msg_warning_tl \c_msg_info_tl</pre>
---

Simple headers for errors. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\c_msg_coding_error_text_tl
\c_msg_fatal_text_tl
\c_msg_help_text_tl
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl
\c_msg_no_info_text_tl
\c_msg_return_text_tl
```

Various pieces of text for use in messages, which are not changed by the code here although they could be to alter the language. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\c_msg_on_line_tl
```

The ‘on line’ phrase for line numbers. Although marked as a constant, they could be changed for printing errors in a different language.

```
\c_msg_text_prefix_tl
\c_msg_more_text_prefix_tl
```

Header information for storing the ‘paths’ to parts of a message. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\l_msg_class_tl
\l_msg_current_class_tl
\l_msg_current_module_tl
```

Information about message method, used for filtering.

```
\l_msg_names_clist
```

List of all of the message names defined.

```
\l_msg_redirect_classes_prop
\l_msg_redirect_names_prop
```

Re-direction lists containing the class of message to convert an different one.

```
\l_msg_redirect_classes_clist
```

List so that filtering does not loop.

## Part XVII

# 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_`.

## 79 Generic functions

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

`\box_new:N    <box>`

Defines `<box>` to be a new variable of type `box`.

**T<sub>E</sub>Xhackers note:** `\box_new:N` is the equivalent of plain T<sub>E</sub>X's `\newbox`.

<code>\if_hbox:N</code>
<code>\if_vbox:N</code>
<code>\if_box_empty:N</code>

`\if_hbox:N <box> <true code>\else: <false code>\fi:`  
`\if_box_empty:N <box> <true code>\else: <false code>\fi:`

`\if_hbox:N` and `\if_vbox:N` check if `<box>` is an horizontal or vertical box resp. `\if_box_empty:N` tests if `<box>` is empty (void) and executes `code` according to the test outcome.

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\ifhbox`, `\ifvbox` and `\ifvoid`.

<code>\box_if_horizontal_p:N</code>
<code>\box_if_horizontal_p:c</code>
<code>\box_if_horizontal:NTF</code>
<code>\box_if_horizontal:cTF</code>

`\box_if_horizontal:NTF    <box> {\true code} {\false code}`

Tests if `<box>` is an horizontal box and executes `<code>` accordingly.

<code>\box_if_vertical_p:N</code>
<code>\box_if_vertical_p:c</code>
<code>\box_if_vertical:NTF</code>
<code>\box_if_vertical:cTF</code>

`\box_if_vertical:NTF    <box> {\true code} {\false code}`

Tests if `<box>` is a vertical box and executes `<code>` accordingly.

<code>\box_if_empty_p:N</code>
<code>\box_if_empty_p:c</code>
<code>\box_if_empty:NTF</code>
<code>\box_if_empty:cTF</code>

`\box_if_empty:NTF    <box> {\true code} {\false code}`

Tests if `<box>` is empty (void) and executes `code` according to the test outcome.

**T<sub>E</sub>Xhackers note:** `\box_if_empty:NTF` is the L<sup>A</sup>T<sub>E</sub>X3 function name for `\ifvoid`.

<code>\box_set_eq:NN</code>
<code>\box_set_eq:cN</code>
<code>\box_set_eq:Nc</code>
<code>\box_set_eq:cc</code>
<code>\box_set_eq_clear:NN</code>
<code>\box_set_eq_clear:cN</code>
<code>\box_set_eq_clear:Nc</code>
<code>\box_set_eq_clear:cc</code>

`\box_set_eq:NN     $\langle box_1 \rangle$   $\langle box_2 \rangle$` 

Sets  $\langle box_1 \rangle$  equal to  $\langle box_2 \rangle$ . The `_clear` versions eradicate the contents of  $\langle box_2 \rangle$  afterwards.

<code>\box_gset_eq:NN</code>
<code>\box_gset_eq:cN</code>
<code>\box_gset_eq:Nc</code>
<code>\box_gset_eq:cc</code>
<code>\box_gset_eq_clear:NN</code>
<code>\box_gset_eq_clear:cN</code>
<code>\box_gset_eq_clear:Nc</code>
<code>\box_gset_eq_clear:cc</code>

`\box_gset_eq:NN     $\langle box_1 \rangle$   $\langle box_2 \rangle$` 

Globally sets  $\langle box_1 \rangle$  equal to  $\langle box_2 \rangle$ . The `_clear` versions eradicate the contents of  $\langle box_2 \rangle$  afterwards.

<code>\box_set_to_last:N</code>
<code>\box_set_to_last:c</code>
<code>\box_gset_to_last:N</code>
<code>\box_gset_to_last:c</code>

`\box_set_to_last:N     $\langle box \rangle$` 

Sets  $\langle box \rangle$  equal to the previous box `\l_last_box` and removes `\l_last_box` from the current list (unless in outer vertical or math mode).

<code>\box_move_right:nn</code>
<code>\box_move_left:nn</code>
<code>\box_move_up:nn</code>
<code>\box_move_down:nn</code>

`\box_move_left:nn    { $\langle dimen \rangle$ } { $\langle box function \rangle$ }`

Moves  $\langle box function \rangle$   $\langle dimen \rangle$  in the direction specified.  $\langle box function \rangle$  is either an operation on a box such as `\box_use:N` or a “raw” box specification like `\vbox:n{xyz}`.

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

`\box_clear:N     $\langle box \rangle$` 

Clears  $\langle box \rangle$  by setting it to the constant `\c_void_box`. `\box_gclear:N` does it globally.

<code>\box_use:N</code>	
<code>\box_use:c</code>	
<code>\box_use_clear:N</code>	<code>\box_use:N</code> $\langle box \rangle$
<code>\box_use_clear:c</code>	<code>\box_use_clear:N</code> $\langle box \rangle$

`\box_use:N` puts a copy of  $\langle box \rangle$  on the current list while `\box_use_clear:N` puts the box on the current list and then eradicates the contents of it.

**T<sub>E</sub>Xhackers note:** `\box_use:N` and `\box_use_clear:N` are the T<sub>E</sub>X primitives `\copy` and `\box` with new (descriptive) names.

<code>\box_ht:N</code>	
<code>\box_ht:c</code>	
<code>\box_dp:N</code>	
<code>\box_dp:c</code>	
<code>\box_wd:N</code>	
<code>\box_wd:c</code>	<code>\box_ht:N</code> $\langle box \rangle$

Returns the height, depth, and width of  $\langle box \rangle$  for use in dimension settings.

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\ht`, `\dp` and `\wd`.

<code>\box_set_dp:Nn</code>	
<code>\box_set_dp:cn</code>	<code>\box_set_dp:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$

Set the depth(below the baseline) of the  $\langle box \rangle$  to the value of the  $\{ \langle dimension expression \rangle \}$ . This is a local assignment.

<code>\box_set_ht:Nn</code>	
<code>\box_set_ht:cn</code>	<code>\box_set_ht:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$

Set the height(above the baseline) of the  $\langle box \rangle$  to the value of the  $\{ \langle dimension expression \rangle \}$ . This is a local assignment.

<code>\box_set_wd:Nn</code>	
<code>\box_set_wd:cn</code>	<code>\box_set_wd:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$

Set the width of the  $\langle box \rangle$  to the value of the  $\{ \langle dimension expression \rangle \}$ . This is a local assignment.

<code>\box_show:N</code>	
<code>\box_show:c</code>	<code>\box_show:N</code> $\langle box \rangle$

Writes the contents of  $\langle box \rangle$  to the log file.

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\showbox`.

<code>\c_empty_box</code> <code>\l_tmpa_box</code> <code>\l_tmpb_box</code>
---

`\c_empty_box` is the constantly empty box. The others are scratch boxes.

<code>\l_last_box</code>
--------------------------

`\l_last_box` is more or less a read-only box register managed by the engine. It denotes the last box on the current list if there is one, otherwise it is void. You can set other boxes to this box, with the result that the last box on the current list is removed at the same time (so it is with variable with side-effects).

## 80 Horizontal mode

<code>\hbox:n</code>
----------------------

`\hbox:n {<contents>}`

Places a `hbox` of natural size.

<code>\hbox_set:Nn</code> <code>\hbox_set:cn</code> <code>\hbox_gset:Nn</code> <code>\hbox_gset:cn</code>
--

`\hbox_set:Nn <box> {<contents>}`

Sets `<box>` to be a horizontal mode box containing `<contents>`. It has its natural size. `\hbox_gset:Nn` does it globally.

<code>\hbox_set_to_wd:Nnn</code> <code>\hbox_set_to_wd:cnn</code> <code>\hbox_gset_to_wd:Nnn</code> <code>\hbox_gset_to_wd:cnn</code>
--

`\hbox_set_to_wd:Nnn <box> {<dimen>} {<contents>}`

Sets `<box>` to contain `<contents>` and have width `<dimen>`. `\hbox_gset_to_wd:Nn` does it globally.

<code>\hbox_to_wd:nn</code> <code>\hbox_to_zero:n</code>
---

`\hbox_to_wd:nn {<dimen>} <contents>`  
`\hbox_to_zero:n <contents>`

Places a `<box>` of width `<dimen>` containing `<contents>`. `\hbox_to_zero:n` is a shorthand for a width of zero.

<code>\hbox_overlap_left:n</code> <code>\hbox_overlap_right:n</code>
---

`\hbox_overlap_left:n <contents>`

Places a  $\langle box \rangle$  of width zero containing  $\langle contents \rangle$  in a way the it overlaps with surrounding material (sticking out to the left or right).

<code>\hbox_set_inline_begin:N</code> <code>\hbox_set_inline_begin:c</code> <code>\hbox_set_inline_end:</code> <code>\hbox_gset_inline_begin:N</code> <code>\hbox_gset_inline_begin:c</code> <code>\hbox_gset_inline_end:</code>	<code>\hbox_set_inline_begin:N</code> $\langle box \rangle$ $\langle contents \rangle$ <code>\hbox_set_inline_end:</code>
---	--

Sets  $\langle box \rangle$  to contain  $\langle contents \rangle$ . This type is useful for use in environment definitions.

<code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <code>\hbox_unpack_clear:N</code> <code>\hbox_unpack_clear:c</code>	<code>\hbox_unpack:N</code> $\langle box \rangle$
--	---

`\hbox_unpack:N` unpacks the contents of the  $\langle box \rangle$  register and `\hbox_unpack_clear:N` also clears the  $\langle box \rangle$  after unpacking it.

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\unhcopy` and `\unhbox`.

## 81 Vertical mode

<code>\vbox:n</code>	<code>\vbox:n</code> $\{\langle contents \rangle\}$
----------------------	---

Places a `vbox` of natural size with baseline equal to the baseline of the last object in the box, i.e., if the last object is a line of text the box has the same depth as that line; otherwise the depth will be zero.

<code>\vbox_top:n</code>	<code>\vbox_top:n</code> $\{\langle contents \rangle\}$
--------------------------	---

Same as `\vbox:n` except that the reference point will be at the baseline of the first object in the box not the last.

<code>\vbox_set:Nn</code> <code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code>	<code>\vbox_set:Nn</code> $\langle box \rangle$ $\{\langle contents \rangle\}$
--	--

Sets  $\langle box \rangle$  to be a vertical mode box containing  $\langle contents \rangle$ . It has its natural size and the reference point will be at the baseline of the last object in the box. `\vbox_gset:Nn` does it globally.

<code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code>	<code>\vbox_set_top:Nn</code> $\langle box \rangle$ $\{\langle contents \rangle\}$
--	--

Sets  $\langle box \rangle$  to be a vertical mode box containing  $\langle contents \rangle$ . It has its natural size (usually a small height and a larger depth) and the reference point will be at the baseline of the first object in the box. `\vbox_gset_top:Nn` does it globally.

<code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code> <code>\vbox_gset_to_ht:ccn</code>	<code>\vbox_set_to_ht:Nnn</code> $\langle box \rangle$ $\{\langle dimen \rangle\}$ $\{\langle contents \rangle\}$
---	---

Sets  $\langle box \rangle$  to contain  $\langle contents \rangle$  and have total height  $\langle dimen \rangle$ . `\vbox_gset_to_ht:Nn` does it globally.

<code>\vbox_set_inline_begin:N</code> <code>\vbox_set_inline_end:</code> <code>\vbox_gset_inline_begin:N</code> <code>\vbox_gset_inline_end:</code>	<code>\vbox_set_inline_begin:N</code> $\langle box \rangle$ $\langle contents \rangle$ <code>\vbox_set_inline_end:</code>
--	--

Sets  $\langle box \rangle$  to contain  $\langle contents \rangle$ . This type is useful for use in environment definitions.

<code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$ $\{\langle dimen \rangle\}$
--	--

Sets  $\langle box_1 \rangle$  to contain the top  $\langle dimen \rangle$  part of  $\langle box_2 \rangle$ .

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\vsplit`.

<code>\vbox_to_ht:nn</code> <code>\vbox_to_zero:n</code>	<code>\vbox_to_ht:nn</code> $\{\langle dimen \rangle\}$ $\langle contents \rangle$ <code>\vbox_to_zero:n</code> $\langle contents \rangle$
---	---

Places a  $\langle box \rangle$  of size  $\langle dimen \rangle$  containing  $\langle contents \rangle$ .

<code>\vbox_unpack:N</code> <code>\vbox_unpack:c</code> <code>\vbox_unpack_clear:N</code> <code>\vbox_unpack_clear:c</code>	<code>\vbox_unpack:N</code> $\langle box \rangle$
--	---

`\vbox_unpack:N` unpacks the contents of the  $\langle box \rangle$  register and `\vbox_unpack_clear:N` also clears the  $\langle box \rangle$  after unpacking it.

**T<sub>E</sub>Xhackers note:** These are the T<sub>E</sub>X primitives `\unvcopy` and `\unvbox`.

## Part XVIII

# The l3xref package

## Cross references

`\xref_set_label:n` `\xref_set_label:n {<name>}`

Sets a label in the text. Note that this function does not do anything else than setting the correct labels. In particular, it does not try to fix any spacing around the write node; this is a task for the `galley2` module.

`\xref_new:nn` `\xref_new:nn {<type>} {<value>}`

Defines a new cross reference type `<type>`. This defines the token list variable `\l_xref_curr_<type>_tl` with default value `<value>` which gets written fully expanded when `\xref_set_label:n` is called.

`\xref_deferred_new:nn` `\xref_deferred_new:nn {<type>} {<value>}`

Same as `\xref_new:n` except for this one, the value written happens when T<sub>E</sub>X ships out the page. Page numbers use this one obviously.

`\xref_get_value:nn *` `\xref_get_value:nn {<type>} {<name>}`

Extracts the cross reference information of type `<type>` for the label `<name>`. This operation is expandable.

## Part XIX

# The l3keyval package

## Key-value parsing

A key-value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree      ,
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

This module provides the low-level machinery for processing arbitrary key–value lists. The `l3keys` module provides a higher-level interface for managing run-time settings using key–value input, while other parts of L<sup>A</sup>T<sub>E</sub>X3 also use key–value input based on `l3keyval` (for example the `xtemplate` module).

## 82 Features of `l3keyval`

As `l3keyval` is a low-level module, its functions are restricted to converting a *keyval list* into keys and values for further processing. Each key and value (or key alone) has to be processed further by a function provided when `l3keyval` is called. Typically, this will be *via* one of the `\KV_process...` functions:

```
\KV_process_space_removal_sanitiz:NNn
\my_processor_function_one:n
\my_processor_function_two:nn
{ <keyval list> }
```

The two processor functions here handle the cases where there is only a key, and where there is both a key and value, respectively.

`l3keyval` parses key–value lists in a manner that does not double `#` tokens or expand any input. The module has processor functions which will sanitize the category codes of `=` and `,` tokens (for use in the document body) as well as faster versions which do not do this (for use inside code blocks). Spaces can be removed from each end of the key and value (again for the document body), again with faster code to be used where this is not necessary. Values which are wrapped in braces will have exactly one set removed, meaning that

```
key = {value here},
```

and

```
key = value here,
```

are treated as identical (assuming that space removal is in force). `l3keyval`

## 83 Functions for keyval processing

The `l3keyval` module should be accessed *via* a small set of external functions. These correctly set up the module internals for use by other parts of L<sup>A</sup>T<sub>E</sub>X3.

In all cases, two functions have to be supplied by the programmer to apply to the items from the `<keyval list>` after `!3keyval` has separated out the entries. The first function should take one argument, and will receive the names of keys for which no value was supplied. The second function should take two arguments: a key name and the associated value.

<code>\KV_process_space_removal_sanitiz:NNn</code>	<code>\KV_process_space_removal_sanitiz:NNn</code> $\langle function_1 \rangle \langle function_2 \rangle \{ \langle keyval list \rangle \}$
--	---

Parses the `<keyval list>` splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, and the category codes of non-braced `=` and `,` tokens are normalised so that parsing is ‘category code safe’. After parsing is completed, `<function1>` is used to process keys without values and `<function2>` deals with keys which have associated values.

<code>\KV_process_space_removal_no_sanitiz:NNn</code>	<code>\KV_process_space_removal_no_sanitiz:NNn</code> $\langle function_1 \rangle \langle function_2 \rangle \{ \langle keyval list \rangle \}$
---	--

Parses the `<keyval list>` splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, but category codes are not normalised. After parsing is completed, `<function1>` is used to process keys without values and `<function2>` deals with keys which have associated values.

<code>\KV_process_no_space_removal_no_sanitiz:NNn</code>	<code>\KV_process_no_space_removal_no_sanitiz:NNn</code> $\langle function_1 \rangle \langle function_2 \rangle \{ \langle keyval list \rangle \}$
--	---

Parses the `<keyval list>` splitting it into keys and associated values. Spaces are *not* removed from the ends of the key and value, and category codes are *not* normalised. After parsing is completed, `<function1>` is used to process keys without values and `<function2>` deals with keys which have associated values.

<code>\l_KV_remove_one_level_of_braces_bool</code>	This boolean controls whether or not one level of braces is stripped from the key and value. The default value for this boolean is <code>true</code> so that exactly one level of braces is stripped. For certain applications it is desirable to keep the braces in which case the programmer just has to set the boolean false temporarily. Only applicable when spaces are being removed.
--	--

## 84 Internal functions

The remaining functions provided by `!3keyval` do not have any protection for nesting of one call to the module inside another. They should therefore not be called directly by other modules.

<code>\KV_parse_no_space_removal_no_sanitize:n</code>	<code>\KV_parse_no_space_removal_no_sanitize:n</code> $\{\langle keyval list \rangle\}$
---	---

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are not removed in the parsing process and the category codes of = and , are not normalised.

<code>\KV_parse_space_removal_no_sanitize:n</code>	<code>\KV_parse_space_removal_no_sanitize:n</code> $\{\langle keyval list \rangle\}$
--	--

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value, but the category codes of = and , are not normalised.

<code>\KV_parse_space_removal_sanitize:n</code>	<code>\KV_parse_space_removal_sanitize:n</code> $\{\langle keyval list \rangle\}$
---	---

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value and the category codes of = and , are normalised at the outer level (*i.e.* only unbraced tokens are affected).

<code>\KV_key_no_value_elt:n</code>	<code>\KV_key_no_value_elt:n</code> $\{\langle key \rangle\}$
<code>\KV_key_value_elt:nn</code>	<code>\KV_key_value_elt:n</code> $\{\langle key \rangle\} \{\langle value \rangle\}$

Used by `\KV_parse...` functions to further process keys with no values and keys with values, respectively. The standard definitions are error functions: the programmer should provide appropriate definitions for both at point of use.

## 85 Variables and constants

<code>\c_KV_single_equal_sign_tl</code>	Constant token list to make finding = faster.
---	---

<code>\l_KV_tmpa_tl</code> <code>\l_KV_tmpb_tl</code>	Scratch token lists.
--	----------------------

<code>\l_KV_parse_tl</code> <code>\l_KV_currkey_tl</code> <code>\l_KV_currval_tl</code>	Token list variables for various parts of the parsed input.
---	---

## Part XX

# The l3keys package

## Key–value support

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. For the user, the system normally results in input of the form

```
\PackageControlMacro{
  key-one = value one,
  key-two = value two
}
```

or

```
\PackageMacro[
  key-one = value one,
  key-two = value two
]{argument}.
```

For the programmer, the original keyval package gives only the most basic interface for this work. All key macros have to be created one at a time, and as a result the kvoptions and xkeyval packages have been written to extend the ease of creating keys. A very different approach has been provided by the pgfkeys package, which uses a key–value list to generate keys.

The l3keys package is aimed at creating a programming interface for key–value controls in L<sup>A</sup>T<sub>E</sub>X3. Keys are created using a key–value interface, in a similar manner to pgfkeys. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { module }
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_module_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { module }
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` is used within a document function. For L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub>, a generic set up function could be created with

```

\newcommand*\SomePackageSetup[1]{%
  \@nameuse{keys_set:nn}{module}{#1}%
}

```

or to use key–value input as the optional argument for a macro:

```

\newcommand*\SomePackageMacro[2][]{%
  \begingroup
    \@nameuse{keys_set:nn}{module}{#1}%
    % Main code for \SomePackageMacro
  \endgroup
}

```

The same concepts using xparse for L<sup>A</sup>T<sub>E</sub>X3 use:

```

\DeclareDocumentCommand \SomePackageSetup { m } {
  \keys_set:nn { module } { #1 }
}
\DeclareDocumentCommand \SomePackageMacro { o m } {
  \group_begin:
    \keys_set:nn { module } { #1 }
    % Main code for \SomePackageMacro
  \group_end:
}

```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 87, it is suggested that the character ‘/’ is reserved for sub-division of keys into logical groups. Macros are *not* expanded when creating key names, and so

```

\tl_set:Nn \l_module_tmp_tl { key }
\keys_define:nn { module } {
  \l_module_tmp_tl .code:n = code
}

```

will create a key called `\l_module_tmp_tl`, and not one called `key`.

## 86 Creating keys

`\keys_define:nn` `\keys_define:nn {<module>} {<keyval list>}`

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name should be a text value, but there are no restrictions on the nature of the text. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The  $\langle keyval list \rangle$  should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule } {
  keyname .code:n = Some-code-using-#1,
  keyname .value_required:
}
```

where the properties of the key begin from the `.` after the key name.

The `\keys_define:nn` function does not skip spaces in the input, and does not check the category codes for `,` and `=` tokens. This means that it is intended for use with code blocks and other environments where spaces are ignored.

<code>.bool_set:N</code>	$\langle key \rangle$ <code>.bool_set:N = <math>\langle bool \rangle</math></code>
<code>.bool_gset:N</code>	

Defines  $\langle key \rangle$  to set  $\langle bool \rangle$  to  $\langle value \rangle$  (which must be either `true` or `false`). Here,  $\langle bool \rangle$  is a L<sup>A</sup>T<sub>E</sub>X3 boolean variable (*i.e.* created using `\bool_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.choice:</code>	$\langle key \rangle$ <code>.choice:</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 [87.1](#).

<code>.choice_code:n</code>	$\langle key \rangle$ <code>.choice_code:n = <math>\langle code \rangle</math></code>
<code>.choice_code:x</code>	

Stores  $\langle code \rangle$  for use when `.generate_choices:n` creates one or more choice sub-keys of the current key. Inside  $\langle code \rangle$ , `\l_keys_choice_tl` contains the name of the choice made, and `\l_keys_choice_int` is the position of the choice in the list given to `.generate_choices:n`. Choices are discussed in detail in section [87.1](#).

<code>.code:n</code>	$\langle key \rangle$ <code>.code:n = <math>\langle code \rangle</math></code>
<code>.code:x</code>	

Stores the  $\langle code \rangle$  for execution when  $\langle key \rangle$  is called. The  $\langle code \rangle$  can include one parameter (`#1`), which will be the  $\langle value \rangle$  given for the  $\langle key \rangle$ . The `.code:x` variant will expand  $\langle code \rangle$  at the point where the  $\langle key \rangle$  is created.

<code>.default:n</code>	$\langle key \rangle$ <code>.default:n = <math>\langle default \rangle</math></code>
<code>.default:V</code>	

Creates a  $\langle default \rangle$  value for  $\langle key \rangle$ , which is used if no value is given. This will be used if only the key name is given, but not if a blank  $\langle value \rangle$  is given:

```

\keys_define:nn { module } {
  key .code:n      = Hello #1,
  key .default:n   = World
}
\keys_set:nn { module} {
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,   % Prints 'Hello '
}

```

**T<sub>E</sub>Xhackers note:** The  $\langle default \rangle$  is stored as a token list variable, and therefore should not contain unescaped # tokens.

<code>.dim_set:N</code>
<code>.dim_set:c</code>
<code>.dim_gset:N</code>
<code>.dim_gset:c</code>

 $\langle key \rangle .dim\_set:N = \langle dimension \rangle$ 

Sets  $\langle key \rangle$  to store the value it is given in  $\langle dimension \rangle$ . Here,  $\langle dimension \rangle$  is a L<sup>A</sup>T<sub>E</sub>X3 dim variable (*i.e.* created using `\dim_new:N`) or a L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> `dimen` (*i.e.* created using `\newdimen`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.fp_set:N</code>
<code>.fp_set:c</code>
<code>.fp_gset:N</code>
<code>.fp_gset:c</code>

 $\langle key \rangle .fp\_set:N = \langle floating\ point \rangle$ 

Sets  $\langle key \rangle$  to store the value it is given in  $\langle floating\ point \rangle$ . Here,  $\langle floating\ point \rangle$  is a L<sup>A</sup>T<sub>E</sub>X3 fp variable (*i.e.* created using `\fp_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.generate_choices:n</code>
----------------------------------

 $\langle key \rangle .generate\_choices:n = \langle comma\ list \rangle$ 

Makes  $\langle key \rangle$  a multiple choice key, accepting the choices specified in  $\langle comma\ list \rangle$ . Each choice will execute code which should previously have been defined using `.choice_code:n` or `.choice_code:x`. Choices are discussed in detail in section 87.1.

<code>.int_set:N</code>
<code>.int_set:c</code>
<code>.int_gset:N</code>
<code>.int_gset:c</code>

 $\langle key \rangle .int\_set:N = \langle integer \rangle$ 

Sets  $\langle key \rangle$  to store the value it is given in  $\langle integer \rangle$ . Here,  $\langle integer \rangle$  is a L<sup>A</sup>T<sub>E</sub>X3 int variable (*i.e.* created using `\int_new:N`) or a L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> `count` (*i.e.* created using `\newcount`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.meta:n</code> <code>.meta:x</code>	$\langle key \rangle .meta:n = \langle multiple\ keys \rangle$
--	--

Makes  $\langle key \rangle$  a meta-key, which will set  $\langle multiple\ keys \rangle$  in one go. If  $\langle key \rangle$  is given with a value at the time the key is used, then the value will be passed through to the subsidiary  $\langle keys \rangle$  for processing (as #1).

<code>.skip_set:N</code> <code>.skip_set:c</code> <code>.skip_gset:N</code> <code>.skip_gset:c</code>	$\langle key \rangle .skip\_set:N = \langle skip \rangle$
--	---

Sets  $\langle key \rangle$  to store the value it is given in  $\langle skip \rangle$ , which is created if it does not already exist. Here,  $\langle skip \rangle$  is a L<sup>A</sup>T<sub>E</sub>X3 `skip` variable (*i.e.* created using `\skip_new:N`) or a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> `skip` (*i.e.* created using `\newskip`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.tl_set:N</code> <code>.tl_set:c</code> <code>.tl_set_x:N</code> <code>.tl_set_x:c</code> <code>.tl_gset:N</code> <code>.tl_gset:c</code> <code>.tl_gset_x:N</code> <code>.tl_gset_x:c</code>	$\langle key \rangle .tl\_set:N = \langle token\ list\ variable \rangle$
--	--

Sets  $\langle key \rangle$  to store the value it is given in  $\langle token\ list\ variable \rangle$ , which is created if it does not already exist. Here,  $\langle token\ list\ variable \rangle$  is a L<sup>A</sup>T<sub>E</sub>X3 `tl` variable (*i.e.* created using `\tl_new:N`) or a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> macro with no arguments (*i.e.* created using `\newcommand` or `\def`). If the variable does not exist, it will be created at the point that the key is set up. The `x` variants perform an `x` expansion at the time the  $\langle value \rangle$  passed to the  $\langle key \rangle$  is saved to the  $\langle token\ list\ variable \rangle$ .

<code>.value_forbidden:</code> <code>.value_required:</code>	$\langle key \rangle .value\_forbidden:$
---	--

Flags for forbidding and requiring a  $\langle value \rangle$  for  $\langle key \rangle$ . Giving a  $\langle value \rangle$  for a  $\langle key \rangle$  which has the `.value_forbidden:` property set will result in an error. In the same way, if a  $\langle key \rangle$  has the `.value_required:` property set then a  $\langle value \rangle$  must be given when the  $\langle key \rangle$  is used.

## 87 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { module / subgroup } {
  key .code:n = code
}
```

or to the key name:

```
\keys_define:nn { module } {
  subgroup / key .code:n = code
}
```

As illustrated, the best choice of token for sub-dividing keys in this way is ‘/’. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `module/subgroup/key`.

As will be illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

## 87.1 Multiple choices

Multiple choices are created by setting the `.choice:` property:

```
\keys_define:nn { module } {
  key .choice:
}
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of choices. Here, the keys can share the same code, and can be rapidly created using the `.choice_code:n` and `.generate_choices:n` properties:

```
\keys_define:nn { module } {
  key .choice_code:n = {
    You~gave~choice~‘‘\int_use:N \l_keys_choice_tl’’ ,~
    which~is~in~position~
    \int_use:N\l_keys_choice_int\space
    in~the~list.
  },
  key .generate_choices:n = {
    choice-a, choice-b, choice-c
  }
}
```

Following common computing practice, `\l_keys_choice_int` is indexed from 0 (as an offset), so that the value of `\l_keys_choice_int` for the first choice in a list will be zero. This means that `\l_keys_choice_int` can be used directly with `\if_case:w` and so on.

```
\l_keys_choice_int
\l_keys_choice_tl
```

Inside the code block for a choice generated using `.generate_choice:`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 0.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { module } {
  key .choice:n,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.generate_choices:n` (*i.e.* anything might happen!).

## 88 Setting keys

```
\keys_set:nn
\keys_set:nV
\keys_set:nv
```

`\keys_set:nn {<module>} {<keyval list>}`

Parses the `<keyval list>`, and sets those keys which are defined for `<module>`. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this will be illustrated later. In contrast to `\keys_define:nn`, this function does check category codes and ignore spaces, and is therefore suitable for user input.

If a key is not known, `\keys_set:nn` will look for a special `unknown` key for the same module. This mechanism can be used to create new keys from user input.

```
\keys_define:nn { module } {
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_path_tl'~to~'#1'
}
```

```
\l_keys_key_tl
```

When processing an unknown key, the name of the key is available as `\l_keys_key_tl`. Note that this will have been processed using `\tl_to_str:N`. The value passed to the key (if any) is available as the macro parameter `#1`.

## 88.1 Examining keys: internal representation

`\keys_if_exist:nnTF` `\keys_if_exist:nnTF {<module>} {<key>} {<true code>}`  
`{<false code>}`

Tests if  $\langle key \rangle$  exists for  $\langle module \rangle$ , i.e. if any code has been defined for  $\langle key \rangle$ .

**TeXhackers note:** The function works by testing for the existence of the internal function `\keys > <module>/<key>.cmd:n`.

`\keys_show:nn` `\keys_show:nn {<module>} {<key>}`

Shows the internal representation of a  $\langle key \rangle$ .

**TeXhackers note:** Keys are stored as functions with names of the format `\keys > <module>/<key>.cmd:n`.

## 89 Internal functions

`\keys_bool_set:Nn` `\keys_bool_set:Nn <bool> {<scope>}`

Creates code to set  $\langle bool \rangle$  when  $\langle key \rangle$  is given, with setting using  $\langle scope \rangle$  (empty or `g` for local or global, respectively).  $\langle bool \rangle$  should be a L<sup>A</sup>T<sub>E</sub>X3 boolean variable.

`\keys_choice_code_store:x` `\keys_choice_code_store:x <code>`

Stores  $\langle code \rangle$  for later use by `.generate_code:n`.

`\keys_choice_make:` `\keys_choice_make:`

Makes  $\langle key \rangle$  a choice key.

`\keys_choices_generate:n` `\keys_choices_generate:n {<comma list>}`

Makes  $\langle comma list \rangle$  choices for  $\langle key \rangle$ .

`\keys_choice_find:n` `\keys_choice_find:n {<choice>}`

Searches for  $\langle choice \rangle$  as a sub-key of  $\langle key \rangle$ .

<code>\keys_cmd_set:nn</code>
<code>\keys_cmd_set:nx</code>

`\keys_cmd_set:nn {⟨path⟩} {⟨code⟩}`  
 Creates a function for `⟨path⟩` using `⟨code⟩`.

<code>\keys_default_set:n</code>
<code>\keys_default_set:V</code>

`\keys_default_set:n {⟨default⟩}`  
 Sets `⟨default⟩` for `⟨key⟩`.

<code>\keys_define_elt:n</code>
<code>\keys_define_elt:nn</code>

`\keys_define_elt:nn {⟨key⟩} {⟨value⟩}`  
 Processing functions for key–value pairs when defining keys.

<code>\keys_define_key:n</code>
---------------------------------

`\keys_define_key:n {⟨key⟩}`  
 Defines `⟨key⟩`.

<code>\keys_execute:</code>
-----------------------------

`\keys_execute:`  
 Executes `⟨key⟩` (where the name of the `⟨key⟩` will be stored internally).

<code>\keys_execute_unknown:</code>
-------------------------------------

`\keys_execute_unknown:`

Handles unknown `⟨key⟩` names.

<code>\keys_if_value_requirement:nTF</code>	<code>\keys_if_value_requirement:nTF {⟨requirement⟩}</code>
<code>★</code>	<code>{⟨true code⟩} {⟨false code⟩}</code>

Check if `⟨requirement⟩` applies to `⟨key⟩`.

<code>\keys_meta_make:n</code>
<code>\keys_meta_make:x</code>

`\keys_meta_make:n {⟨keys⟩}`  
 Makes `⟨key⟩` a meta-key to set `⟨keys⟩`.

<code>\keys_property_find:n</code>
------------------------------------

`\keys_property_find:n {⟨key⟩}`

Separates `⟨key⟩` from `⟨property⟩`.

<code>\keys_property_new:nn</code>
<code>\keys_property_new_arg:nn</code>

`\keys_property_new:nn {⟨property⟩} {⟨code⟩}`

Makes a new  $\langle property \rangle$  expanding to  $\langle code \rangle$ . The `arg` version makes properties with one argument.

`\keys_property_undefine:n` `\keys_property_undefine:n { $\langle property \rangle$ }`

Deletes  $\langle property \rangle$  of  $\langle key \rangle$ .

`\keys_set_elt:n`  
`\keys_set_elt:nn` `\keys_set_elt:nn { $\langle key \rangle$ } { $\langle value \rangle$ }`

Processing functions for key–value pairs when setting keys.

`\keys_tmp:w` `\keys_tmp:w  $\langle args \rangle$`

Used to store  $\langle code \rangle$  to execute a  $\langle key \rangle$ .

`\keys_value_or_default:n` `\keys_value_or_default:n { $\langle value \rangle$ }`

Sets `\l_keys_value_tl` to  $\langle value \rangle$ , or  $\langle default \rangle$  if  $\langle value \rangle$  was not given and if  $\langle default \rangle$  is available.

`\keys_value_requirement:n` `\keys_value_requirement:n { $\langle requirement \rangle$ }`

Sets  $\langle key \rangle$  to have  $\langle requirement \rangle$  concerning  $\langle value \rangle$ .

`\keys_variable_set:NnNN`  
`\keys_variable_set:cnNN` `\keys_variable_set:NnNN  $\langle var \rangle$   $\langle type \rangle$   $\langle scope \rangle$   $\langle expansion \rangle$`

Sets  $\langle key \rangle$  to assign  $\langle value \rangle$  to  $\langle variable \rangle$ . The  $\langle scope \rangle$  (blank for local, `g` for global) and  $\langle type \rangle$  (`tl`, `int`, etc.) are given explicitly.

## 90 Variables and constants

`\c_keys_properties_root_tl`  
`\c_keys_root_tl`

The root paths for keys and properties, used to generate the names of the functions which store these items.

`\c_keys_value_forbidden_tl`  
`\c_keys_value_required_tl`

Marker text containers: by storing the values the code can make comparisons slightly faster.

`\l_keys_choice_code_tl` Used to transfer code from storage when making multiple choices.

`\l_keys_module_tl`  
`\l_keys_path_tl`  
`\l_keys_property_tl` Various key paths need to be stored. These are flexible items that are set during the key reading process.

`\l_keys_no_value_bool` A marker for ‘no value’ as key input.

`\l_keys_value_tl` Holds the currently supplied value, in a token register as there may be # tokens.

## Part XXI

# The l3file package

## File Loading

### 91 Loading files

In contrast to the l3io module, which deals with the lowest level of file management, the l3file module provides a higher level interface for handling file contents. This involves providing convenient wrappers around many of the functions in l3io to make them more generally accessible.

It is important to remember that T<sub>E</sub>X will attempt to locate files using both the operating system path and entries in the T<sub>E</sub>X file database (most T<sub>E</sub>X systems use such a database). Thus the ‘current path’ for T<sub>E</sub>X is somewhat broader than that for other programs.

`\g_file_current_name_tl` Contains the name of the current LaTeX file. This variable should not be modified: it is intended for information only. It will be equal to `\c_job_name_tl` at the start of a L<sup>A</sup>T<sub>E</sub>X run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`  
`\file_if_exist:VTF` `\file_if_exist:nTF {<file name>} {<true code>} {<false code>}`  
 Searches for <file name> using the current T<sub>E</sub>X search path and the additional paths controlled by `\file_path_include:n`. The branching versions then leave either <true

*code*) or *(false code)* in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

<code>\file_input:n</code>	<code>\file_input:n {&lt;file name&gt;}</code>
<code>\file_input:V</code>	

Searches for *<file name>* in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L<sup>A</sup>T<sub>E</sub>X source. All files read are recorded for information and the file name stack is updated by this function.

<code>\file_path_include:n</code>	<code>\file_path_include:n {&lt;path&gt;}</code>
-----------------------------------	--

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.

<code>\file_path_remove:n</code>	<code>\file_path_remove:n {&lt;path&gt;}</code>
----------------------------------	---

Removes *<path>* from the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

<code>\file_list:</code>	<code>\file_list:</code>
--------------------------	--------------------------

This function will list all files loaded using `\file_input:n` in the log file.

## Part XXII

# The l3fp package

## Floating point arithmetic

### 92 Floating point numbers

A floating point number is one which is stored as a mantissa and a separate exponent. This module implements arithmetic using radix 10 floating point numbers. This means that the mantissa should be a real number in the range  $1 \leq |x| < 10$ , with the exponent given as an integer between  $-99$  and  $99$ . In the input, the exponent part is represented starting with an `e`. As this is a low-level module, error-checking is minimal. Numbers which are too large for the floating point unit to handle will result in errors, either from T<sub>E</sub>X or from L<sup>A</sup>T<sub>E</sub>X. The L<sup>A</sup>T<sub>E</sub>X code does not check that the input will not overflow, hence the possibility of a T<sub>E</sub>X 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.

## 92.1 Constants

`\c_e_fp` The value of the base of natural numbers,  $e$ .

`\c_one_fp` A floating point variable with permanent value 1: used for speeding up some comparisons.

`\c_pi_fp` The value of  $\pi$ .

`\c_undefined_fp` A special marker floating point variable representing the result of an operation which does not give a defined result (such as division by 0).

`\c_zero_fp` A permanently zero floating point variable.

## 92.2 Floating-point variables

`\fp_new:N`  
`\fp_new:c` `\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).

`\fp_const:Nn`  
`\fp_const:cn` `\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`  $\langle fp\ var1 \rangle$   $\langle fp\ var2 \rangle$

Sets the value of  $\langle floating\ point\ variable1 \rangle$  equal to that of  $\langle floating\ point\ variable2 \rangle$ . This assignment is restricted to the current  $\text{\TeX}$  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`  $\langle fp\ var1 \rangle$   $\langle fp\ var2 \rangle$

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  $\text{\TeX}$  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  $+0.000000000\text{e}0$  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  $+0.000000000\text{e}0$  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  $\text{\TeX}$  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 *floating point variable* to the distance represented by the *dimension expression* in the units points. This means that distances given in other units are first converted to points before being assigned to the *floating point variable*. The assignment is global.

\fp_use:N ★	
\fp_use:c ★	

\fp\_use:N *floating point variable*

Inserts the value of the *floating point variable* into the input stream. The value will be given as a real number without any exponent part, and will always include a decimal point. For example,

```
\fp_new:Nn \test
\fp_set:Nn \test { 1.234 e 5 }
\fp_use:N \test
```

will insert ‘12345.00000’ into the input stream. As illustrated, a floating point will always be inserted with ten significant digits given. Very large and very small values will include additional zeros for place value.

\fp_show:N	
\fp_show:c	

\fp\_show:N *floating point variable*

Displays the content of the *floating point variable* on the terminal.

## 92.3 Conversion to other formats

It is useful to be able to convert floating point variables to other forms. These functions are expandable, so that the material can be used in a variety of contexts. The `\fp_use:N` function should also be consulted in this context, as it will insert the value of the floating point variable as a real number.

\fp_to_dim:N ★	
\fp_to_dim:c ★	

\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.

\fp_to_int:N ★	
\fp_to_int:c ★	

\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.

\fp_to_tl:N ★	
\fp_to_tl:c ★	

\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.

## 92.4 Rounding floating point values

The module can round floating point values to either decimal places or significant figures using the usual method in which exact halves are rounded up.

<code>\fp_round_figures:Nn</code> <code>\fp_round_figures:cn</code>
--

`\fp_round_figures:Nn <floating point variable> {\<target>}`

Rounds the *<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>
--

`\fp_ground_figures:Nn <floating point variable> {\<target>}`

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>
--

`\fp_round_places:Nn <floating point variable> {\<target>}`

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>
--

`\fp_ground_places:Nn <floating point variable> {\<target>}`

Rounds the *<floating point variable>* to the *<target>* number of decimal places (an integer expression). The rounding is carried out globally.

## 92.5 Tests on floating-point values

$\backslash\text{fp\_if\_undefined\_p:N} \star$	$\backslash\text{fp\_if\_undefined\_p:N} \langle\text{fixed-point}\rangle$
$\backslash\text{fp\_if\_undefined:N}\underline{\text{TF}} \star$	$\backslash\text{fp\_if\_undefined:N}\underline{\text{TF}} \langle\text{fixed-point}\rangle$
	$\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$

Tests if  $\langle\text{floating point}\rangle$  is undefined (*i.e.* equal to the special  $\backslash\text{c\_undefined\_fp}$  variable). The branching versions then leave either  $\langle\text{true code}\rangle$  or  $\langle\text{false code}\rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

$\backslash\text{fp\_if\_zero\_p:N} \star$	$\backslash\text{fp\_if\_zero\_p:N} \langle\text{fixed-point}\rangle$
$\backslash\text{fp\_if\_zero:N}\underline{\text{TF}} \star$	$\backslash\text{fp\_if\_zero:N}\underline{\text{TF}} \langle\text{fixed-point}\rangle \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$

Tests if  $\langle\text{floating point}\rangle$  is equal to zero (*i.e.* equal to the special  $\backslash\text{c\_zero\_fp}$  variable). The branching versions then leave either  $\langle\text{true code}\rangle$  or  $\langle\text{false code}\rangle$  in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

	$\backslash\text{fp\_compare:nNnTF}$
$\backslash\text{fp\_compare:nNn}\underline{\text{TF}}$	$\{\langle\text{floating point}_1\rangle\} \langle\text{relation}\rangle \{\langle\text{floating point}_2\rangle\}$
	$\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$

This function compared the two  $\langle\text{floating point}\rangle$  values, which may be stored as **fp** variables, using the  $\langle\text{relation}\rangle$ :

Equal	=
Greater than	>
Less than	<

Either  $\langle\text{true code}\rangle$  or  $\langle\text{false code}\rangle$  is then left in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

## 92.6 Unary operations

The unary operations alter the value stored within an **fp** variable.

$\backslash\text{fp\_abs:N}$	
$\backslash\text{fp\_abs:c}$	$\backslash\text{fp\_abs:N} \langle\text{floating point variable}\rangle$

Converts the  $\langle\text{floating point variable}\rangle$  to its absolute value, assigning the result within the current  $\text{T}_{\text{E}}\text{X}$  group.

$\backslash\text{fp\_gabs:N}$	
$\backslash\text{fp\_gabs:c}$	$\backslash\text{fp\_gabs:N} \langle\text{floating point variable}\rangle$

Converts the  $\langle\text{floating point variable}\rangle$  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 TeX 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.

## 92.7 Arithmetic operations

Binary arithmetic operations act on the value stored in an `fp`, so for example

```
\fp_set:Nn \l_my_fp { 1.234 }
\fp_sub:Nn \l_my_fp { 5.678 }
```

sets `\l_my_fp` to the result of  $1.234 - 5.678$  (*i.e.*  $-4.444$ ).

<code>\fp_add:Nn</code> <code>\fp_add:cn</code>
--

`\fp_add:Nn` *<floating point>* {*<value>*}

Adds the *<value>* to the *<floating point>*, making the assignment within the current TeX 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` *<floating point>* {*<value>*}

Subtracts the *<value>* from the *<floating point>*, making the assignment globally.

<code>\fp_mul:Nn</code> <code>\fp_mul:cn</code>
--

`\fp_mul:Nn` *<floating point>* {*<value>*}

Multiplies the *<floating point>* by the *<value>*, making the assignment within the current TeX group level.

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

`\fp_gmul:Nn` *<floating point>* {*<value>*}

Multiples the *<floating point>* by the *<value>*, making the assignment globally.

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

`\fp_div:Nn` *<floating point>* {*<value>*}

Divides the *<floating point>* by the *<value>*, making the assignment within the current TeX group level. If the *<value>* is zero, the *<floating point>* will be set to `\c_undefined_fp`. The assignment is local.

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

`\fp_gdiv:Nn` *<floating point>* {*<value>*}

Divides the *<floating point>* by the *<value>*, making the assignment globally. If the *<value>* is zero, the *<floating point>* will be set to `\c_undefined_fp`. The assignment is global.

## 92.8 Power operations

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

`\fp_pow:Nn` *<floating point>* {*<value>*}

Raises the *<floating point>* to the given *<value>*, which should be a positive real number or a negative integer. Mathematically invalid operations such as  $0^0$  will give set the *<floating point>* to `\c_undefined_fp`. The assignment is local.

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

`\fp_gpow:Nn` *<floating point>* {*<value>*}

Raises the *<floating point>* to the given *<value>*, which should be a positive real number or a negative integer. Mathematically invalid operations such as  $0^0$  will give set the *<floating point>* to `\c_undefined_fp`. The assignment is global.

## 92.9 Exponential and logarithm functions

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

`\fp_exp:Nn` *<floating point>* {*<value>*}

Calculates the exponential of the *<value>* and assigns this to the *<floating point>*. The assignment is local.

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

`\fp_gexp:Nn` *<floating point>* {*<value>*}

Calculates the exponential of the *<value>* and assigns this to the *<floating point>*. The assignment is global.

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

`\fp_ln:Nn` *<floating point>* {*<value>*}

Calculates the natural logarithm of the *<value>* and assigns this to the *<floating point>*. The assignment is local.

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

`\fp_gln:Nn` *<floating point>* {*<value>*}

Calculates the natural logarithm of the *<value>* and assigns this to the *<floating point>*. The assignment is global.

## 92.10 Trigonometric functions

The trigonometric functions all work in radians. They accept a maximum input value of 100 000 000, as there are issues with range reduction and very large input values.

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

`\fp_sin:Nn` *<floating point>* {*<value>*}

Assigns the sine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is local.

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

`\fp_gsin:Nn` *<floating point>* {*<value>*}

Assigns the sine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is global.

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

`\fp_cos:Nn` *<floating point>* {*<value>*}

Assigns the cosine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is local.

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

`\fp_gcos:Nn` *<floating point>* {*<value>*}

Assigns the cosine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is global.

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

`\fp_tan:Nn` *<floating point>* {*<value>*}

Assigns the tangent of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is local.

<code>\fp_gtan:Nn</code> <code>\fp_gtan:cn</code>
--

`\fp_gtan:Nn`  $\langle floating\ point \rangle$   $\{ \langle value \rangle \}$

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.

## 92.11 Notes on the floating point unit

As calculation of the elemental transcendental functions is computationally expensive compared to storage of results, after calculating a trigonometric function, exponent, *etc.* the module stored the result for reuse. Thus the performance of the module for repeated operations, most probably trigonometric functions, should be much higher than if the values were re-calculated every time they were needed.

Anyone with experience of programming floating point calculations will know that this is a complex area. The aim of the unit is to be accurate enough for the likely applications in a typesetting context. The arithmetic operations are therefore intended to provide ten digit accuracy with the last digit accurate to  $\pm 1$ . 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  $\text{\TeX}$  system. This means that the format is somewhat different from that used in, for example, computer floating point units. Programming in  $\text{\TeX}$  makes it most convenient to use a radix 10 system, using  $\text{\TeX}$  `count` registers for storage and taking advantage where possible of delimited arguments.

## Part XXIII

# The l3 $\text{\LaTeX}$ package Lua $\text{\TeX}$ -specific functions

## 93 Breaking out to Lua

The Lua $\text{\TeX}$  engine provides access to the Lua programming language, and with it access to the 'internals' of  $\text{\TeX}$ . In order to use this within the framework provided here, two functions are available. When used with pdf $\text{\TeX}$  or Xe $\text{\TeX}$  these will raise an error: use `\engine_if_lua $\text{\TeX}$ :T` to avoid this. Details of coding the Lua $\text{\TeX}$  engine are detailed in the Lua $\text{\TeX}$  manual.

`\lua_now:x *` `\lua_now:x {<token list>}`

The *<token list>* is fully expandable using the current applicable T<sub>E</sub>X category codes: this will include converting line ends to spaces in the usual T<sub>E</sub>X manner. The resulting *<Lua input>* is passed to the Lua interpreter for processing. Each `\lua_now:x` block is treated by Lua as a separate chunk. The Lua interpreter will execute the *<Lua input>* immediately, and in an expandable manner.

`\lua_shipout:x *` `\lua_shipout:x {<token list>}`

The *<token list>* is fully expandable using the current applicable T<sub>E</sub>X category codes: this will include converting line ends to spaces in the usual T<sub>E</sub>X manner. The resulting *<Lua input>* is passed to the Lua interpreter for processing. Each `\lua_now:x` block is treated by Lua as a separate chunk. The Lua interpreter will execute the *<Lua input>* during the page-building routine. (At a T<sub>E</sub>X level, the *<Lua input>* is stored as a 'whatsit'.)

## 94 Category code tables

As well as providing methods to break out into Lua, there are places where additional L<sup>A</sup>T<sub>E</sub>X3 functions are provided by the LuaT<sub>E</sub>X engine. In particular, LuaT<sub>E</sub>X provides category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by `\ExplSyntaxOn` and `\ExplSyntaxOff` when using the LuaT<sub>E</sub>X engine.

`\cctab_new:N` `\cctab_new:N <category code table>`

Creates a new category code table, initially with the codes as used by IniT<sub>E</sub>X.

`\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.

<code>\c_code_cctab</code>	Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOn</code> .
----------------------------	--

<code>\c_document_cctab</code>	Category code table for a standard $\text{\LaTeX}$ document. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOff</code> .
--------------------------------	--

<code>\c_initex_cctab</code>	Category code table as set up by <code>Init<math>\text{\TeX}</math></code> .
------------------------------	--

<code>\c_other_cctab</code>	Category code table where all characters have category code 12 (other).
-----------------------------	---

<code>\c_string_cctab</code>	Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).
------------------------------	---

## Part XXIV

# Implementation

## 95 `l3names` implementation

This is the base part of  $\text{\LaTeX}3$  defining things like `catcodes` and redefining the  $\text{\TeX}$  primitives, as well as setting up the code to load `expl3` modules in  $\text{\LaTeX}2_{\epsilon}$ .

### 95.1 Internal functions

<code>\ExplSyntaxStatus</code> <code>\ExplSyntaxPopStack</code> <code>\ExplSyntaxStack</code>	Functions used to track the state of the catcode regime.
---	--

<code>\@pushfilename</code> <code>\@popfilename</code>	Re-definitions of $\text{\LaTeX}$ 's file-loading functions to support <code>\ExplSyntax</code> .
---	---

## 95.2 Bootstrap code

The very first thing to do is to bootstrap the  $\text{\LaTeX}$  system so that everything else will actually work.  $\text{\TeX}$  does not start with some pretty basic character codes set up.

```
1 <!*package>
2 \catcode '\{ = 1 \relax
3 \catcode '\} = 2 \relax
4 \catcode '\# = 6 \relax
5 \catcode '\^ = 7 \relax
6 </!package>
```

Tab characters should not show up in the code, but to be on the safe side.

```
7 <!*package>
8 \catcode '\^^I = 10 \relax
9 </!package>
```

For  $\text{\LuaTeX}$  the extra primitives need to be enabled before they can be use. No  $\text{\ifdefined}$  yet, so do it the old-fashioned way. The primitive  $\text{\strcmp}$  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  $\text{\csname}$  business is needed so that the later deletion code will work.

```
10 <!*package>
11 \begingroup\expandafter\expandafter\expandafter\endgroup
12 \expandafter\ifx\csname directlua\endcsname\relax
13 \else
14 \directlua
15 {
16   tex.enableprimitives('',tex.extraprimitives ())
17   lua.bytecode[1] = function ()
18     function strcmp (A, B)
19       if A == B then
20         tex.write("0")
21       elseif A < B then
22         tex.write("-1")
23       else
24         tex.write("1")
25       end
26     end
27   end
28   lua.bytecode[1]()
29 }
30 \everyjob\expandafter
31 {\csname tex_directlua:D\endcsname{lua.bytecode[1]()}}
32 \long\edef\pdfstrcmp#1#2%
33 {%
34   \expandafter\noexpand\csname tex_directlua:D\endcsname
35   {%
```

```

36         strcmp(%
37             "\noexpand\luaescapestring{#1}",%
38             "\noexpand\luaescapestring{#2}"%
39         )%
40     }%
41 }
42 \fi
43 </!package>

```

When loaded as a package this can all be handed off to other L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> code.

```

44 <*package>
45 \def\@tempa{%
46     \def\@tempa{%
47         \RequirePackage{luatex}%
48         \RequirePackage{pdfsync}%
49         \let\pdfstrcmp\pdf@strcmp
50     }
51 \begingroup\expandafter\expandafter\expandafter\endgroup
52 \expandafter\ifx\csname directlua\endcsname\relax
53 \else
54     \expandafter\@tempa
55 \fi
56 </package>

```

X<sub>ε</sub>T<sub>E</sub>X calls the primitive `\strcmp`, so there needs to be a check for that too.

```

57 \begingroup\expandafter\expandafter\expandafter\endgroup
58 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
59     \let\pdfstrcmp\strcmp
60 \fi

```

### 95.3 Requirements

Currently, the code requires the  $\varepsilon$ -T<sub>E</sub>X primitives and functionality equivalent to `\pdfstrcmp`. Any package which provides the later will provide the former, so the test can be done only for `\pdfstrcmp`.

```

61 \begingroup\expandafter\expandafter\expandafter\endgroup
62 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
63 <*package>
64 \PackageError{!names}{Required primitive not found: \protect\pdfstrcmp}
65 {%
66     LaTeX3 requires the e-TeX primitives and
67     \string\pdfstrcmp.\MessageBreak
68     These are available in engine versions: \MessageBreak
69     - pdfTeX 1.30 \MessageBreak
70     - XeTeX 0.9994 \MessageBreak
71     - LuaTeX 0.60 \MessageBreak

```

```

72         or later. \MessageBreak
73         \MessageBreak
74         Loading of l3names will abort!
75     }
76 </package>
77 <!*package>
78     \newlinechar'\^^J\relax
79     \errhelp{%
80         LaTeX3 requires the e-TeX primitives and
81         \string\pdfstrcmp. ^^J
82         These are available in engine versions: ^^J
83         - pdfTeX 1.30 ^^J
84         - XeTeX 0.9994 ^^J
85         - LuaTeX 0.60 ^^J
86         or later. ^^J
87         For pdfTeX and XeTeX the '-etex' command-line switch is also
88         needed.^^J
89         ^^J
90         Format building will abort!
91     }
92 </!package>
93     \expandafter\endinput
94 \fi

```

## 95.4 Catcode assignments

Catcodes for `begingroup`, `endgroup`, macro parameter, superscript, and `tab`, are all assigned before the start of the documented code. (See the beginning of `l3names.dtx`.)

Reason for `\endlinechar=32` is that a line ending with a backslash will be interpreted as the token `\_` which seems most natural and since spaces are ignored it works as we intend elsewhere.

Before we do this we must however record the settings for the catcode regime as it was when we start changing it.

```

95 <*initex | package>
96 \protected\edef\ExplSyntaxOff{
97     \unexpanded{\ifodd \ExplSyntaxStatus\relax
98     \def\ExplSyntaxStatus{0}
99     }
100     \catcode 126=\the \catcode 126 \relax
101     \catcode 32=\the \catcode 32 \relax
102     \catcode 9=\the \catcode 9 \relax
103     \endlinechar =\the \endlinechar \relax
104     \catcode 95=\the \catcode 95 \relax
105     \catcode 58=\the \catcode 58 \relax
106     \catcode 124=\the \catcode 124 \relax
107     \catcode 38=\the \catcode 38 \relax

```

```

108 \catcode 94=\the \catcode 94 \relax
109 \catcode 34=\the \catcode 34 \relax
110 \noexpand\fi
111 }
112 \catcode126=10\relax % tilde is a space char.
113 \catcode32=9\relax % space is ignored
114 \catcode9=9\relax % tab also ignored
115 \endlinechar=32\relax % newline is space
116 \catcode95=11\relax % underscore letter
117 \catcode58=11\relax % colon letter
118 \catcode124=12\relax % vert bar, other
119 \catcode38=4\relax % ampersand, alignment token
120 \catcode34=12\relax % doublequote, other
121 \catcode94=7\relax % caret, math superscript

```

## 95.5 Setting up primitive names

Here is the function that renames T<sub>E</sub>X's primitives.

Normally the old name is left untouched, but the possibility of undefining the original names is made available by docstrip and package options. If nothing else, this gives a way of checking what ‘old code’ a package depends on...

If the package option ‘removeoldnames’ is used then some trick code is run after the end of this file, to skip past the code which has been inserted by L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> to manage the file name stack, this code would break if run once the T<sub>E</sub>X primitives have been undefined. (What a surprise!) **The option has been temporarily disabled.**

To get things started, give a new name for \let.

```

122 \let \tex_let:D \let
123 </initex | package>

```

and now an internal function to possibly remove the old name: for the moment.

```

124 <*initex>
125 \long \def \name_undefine:N #1 {
126   \tex_let:D #1 \c_undefined
127 }
128 </initex>

129 <*package>
130 \DeclareOption{removeoldnames}{
131   \long\def\name_undefine:N#1{
132     \tex_let:D#1\c_undefined}}

133 \DeclareOption{keepoldnames}{
134   \long\def\name_undefine:N#1{}}

135 \ExecuteOptions{keepoldnames}

```

```

136 \ProcessOptions
137 \</package>

```

The internal function to give the new name and possibly undefine the old name.

```

138 \<*initex | package>
139 \long \def \name_primitive:NN #1#2 {
140   \tex_let:D #2 #1
141   \name_undefine:N #1
142 }

```

## 95.6 Reassignment of primitives

In the current incarnation of this package, all T<sub>E</sub>X primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```

143 \name_primitive:NN \                \tex_space:D
144 \name_primitive:NN \/              \tex_italiccor:D
145 \name_primitive:NN \-              \tex_hyphen:D

```

Now all the other primitives.

```

146 \name_primitive:NN \let              \tex_let:D
147 \name_primitive:NN \def              \tex_def:D
148 \name_primitive:NN \edef            \tex_edef:D
149 \name_primitive:NN \gdef            \tex_gdef:D
150 \name_primitive:NN \xdef            \tex_xdef:D
151 \name_primitive:NN \chardef          \tex_chardef:D
152 \name_primitive:NN \countdef         \tex_countdef:D
153 \name_primitive:NN \dimendef         \tex_dimendef:D
154 \name_primitive:NN \skipdef          \tex_skipdef:D
155 \name_primitive:NN \mskipdef         \tex_mskipdef:D
156 \name_primitive:NN \mathchardef      \tex_mathchardef:D
157 \name_primitive:NN \toksdef          \tex_toksdef:D
158 \name_primitive:NN \futurelet        \tex_futurelet:D
159 \name_primitive:NN \advance          \tex_advance:D
160 \name_primitive:NN \divide           \tex_divide:D
161 \name_primitive:NN \multiply         \tex_multiply:D
162 \name_primitive:NN \font             \tex_font:D
163 \name_primitive:NN \fam              \tex_fam:D
164 \name_primitive:NN \global           \tex_global:D
165 \name_primitive:NN \long             \tex_long:D
166 \name_primitive:NN \outer            \tex_outer:D
167 \name_primitive:NN \setlanguage      \tex_setlanguage:D
168 \name_primitive:NN \globaldefs       \tex_globaldefs:D
169 \name_primitive:NN \afterassignment  \tex_afterassignment:D
170 \name_primitive:NN \aftergroup       \tex_aftergroup:D
171 \name_primitive:NN \expandafter      \tex_expandafter:D

```

172	\name_primitive:NN \noexpand	\tex_noexpand:D
173	\name_primitive:NN \begingroup	\tex_begingroup:D
174	\name_primitive:NN \endgroup	\tex_endgroup:D
175	\name_primitive:NN \halign	\tex_halign:D
176	\name_primitive:NN \valign	\tex_valign:D
177	\name_primitive:NN \cr	\tex_cr:D
178	\name_primitive:NN \crr	\tex_crr:D
179	\name_primitive:NN \noalign	\tex_noalign:D
180	\name_primitive:NN \omit	\tex_omit:D
181	\name_primitive:NN \span	\tex_span:D
182	\name_primitive:NN \tabskip	\tex_tabskip:D
183	\name_primitive:NN \everycr	\tex_everycr:D
184	\name_primitive:NN \if	\tex_if:D
185	\name_primitive:NN \ifcase	\tex_ifcase:D
186	\name_primitive:NN \ifcat	\tex_ifcat:D
187	\name_primitive:NN \ifnum	\tex_ifnum:D
188	\name_primitive:NN \ifodd	\tex_ifodd:D
189	\name_primitive:NN \ifdim	\tex_ifdim:D
190	\name_primitive:NN \ifeof	\tex_ifeof:D
191	\name_primitive:NN \ifhbox	\tex_ifhbox:D
192	\name_primitive:NN \ifvbox	\tex_ifvbox:D
193	\name_primitive:NN \ifvoid	\tex_ifvoid:D
194	\name_primitive:NN \ifx	\tex_ifx:D
195	\name_primitive:NN \iffalse	\tex_iffalse:D
196	\name_primitive:NN \iftrue	\tex_iftrue:D
197	\name_primitive:NN \ifhmode	\tex_ifhmode:D
198	\name_primitive:NN \ifmmode	\tex_ifmmode:D
199	\name_primitive:NN \ifvmode	\tex_ifvmode:D
200	\name_primitive:NN \ifinner	\tex_ifinner:D
201	\name_primitive:NN \else	\tex_else:D
202	\name_primitive:NN \fi	\tex_fi:D
203	\name_primitive:NN \or	\tex_or:D
204	\name_primitive:NN \immediate	\tex_immediate:D
205	\name_primitive:NN \closeout	\tex_closeout:D
206	\name_primitive:NN \openin	\tex_openin:D
207	\name_primitive:NN \openout	\tex_openout:D
208	\name_primitive:NN \read	\tex_read:D
209	\name_primitive:NN \write	\tex_write:D
210	\name_primitive:NN \closein	\tex_closein:D
211	\name_primitive:NN \newlinechar	\tex_newlinechar:D
212	\name_primitive:NN \input	\tex_input:D
213	\name_primitive:NN \endinput	\tex_endinput:D
214	\name_primitive:NN \inputlineno	\tex_inputlineno:D
215	\name_primitive:NN \errmessage	\tex_errmessage:D
216	\name_primitive:NN \message	\tex_message:D
217	\name_primitive:NN \show	\tex_show:D
218	\name_primitive:NN \showthe	\tex_showthe:D
219	\name_primitive:NN \showbox	\tex_showbox:D
220	\name_primitive:NN \showlists	\tex_showlists:D
221	\name_primitive:NN \errhelp	\tex_errhelp:D

222	\name_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
223	\name_primitive:NN	\tracingcommands	\tex_tracingcommands:D
224	\name_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
225	\name_primitive:NN	\tracingmacros	\tex_tracingmacros:D
226	\name_primitive:NN	\tracingonline	\tex_tracingonline:D
227	\name_primitive:NN	\tracingoutput	\tex_tracingoutput:D
228	\name_primitive:NN	\tracingpages	\tex_tracingpages:D
229	\name_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
230	\name_primitive:NN	\tracingrestores	\tex_tracingrestores:D
231	\name_primitive:NN	\tracingstats	\tex_tracingstats:D
232	\name_primitive:NN	\pausing	\tex_pausing:D
233	\name_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
234	\name_primitive:NN	\showboxdepth	\tex_showboxdepth:D
235	\name_primitive:NN	\batchmode	\tex_batchmode:D
236	\name_primitive:NN	\errorstopmode	\tex_errorstopmode:D
237	\name_primitive:NN	\nonstopmode	\tex_nonstopmode:D
238	\name_primitive:NN	\scrollmode	\tex_scrollmode:D
239	\name_primitive:NN	\end	\tex_end:D
240	\name_primitive:NN	\csname	\tex_csname:D
241	\name_primitive:NN	\endcsname	\tex_endcsname:D
242	\name_primitive:NN	\ignorespaces	\tex_ignorespaces:D
243	\name_primitive:NN	\relax	\tex_relax:D
244	\name_primitive:NN	\the	\tex_the:D
245	\name_primitive:NN	\mag	\tex_mag:D
246	\name_primitive:NN	\language	\tex_language:D
247	\name_primitive:NN	\mark	\tex_mark:D
248	\name_primitive:NN	\topmark	\tex_topmark:D
249	\name_primitive:NN	\firstmark	\tex_firstmark:D
250	\name_primitive:NN	\botmark	\tex_botmark:D
251	\name_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
252	\name_primitive:NN	\splitbotmark	\tex_splitbotmark:D
253	\name_primitive:NN	\fontname	\tex_fontname:D
254	\name_primitive:NN	\escapechar	\tex_escapechar:D
255	\name_primitive:NN	\endlinechar	\tex_endlinechar:D
256	\name_primitive:NN	\mathchoice	\tex_mathchoice:D
257	\name_primitive:NN	\delimiter	\tex_delimiter:D
258	\name_primitive:NN	\mathaccent	\tex_mathaccent:D
259	\name_primitive:NN	\mathchar	\tex_mathchar:D
260	\name_primitive:NN	\mskip	\tex_mskip:D
261	\name_primitive:NN	\radical	\tex_radical:D
262	\name_primitive:NN	\vcenter	\tex_vcenter:D
263	\name_primitive:NN	\mkern	\tex_mkern:D
264	\name_primitive:NN	\above	\tex_above:D
265	\name_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
266	\name_primitive:NN	\atop	\tex_atop:D
267	\name_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
268	\name_primitive:NN	\over	\tex_over:D
269	\name_primitive:NN	\overwithdelims	\tex_overwithdelims:D
270	\name_primitive:NN	\displaystyle	\tex_displaystyle:D
271	\name_primitive:NN	\textstyle	\tex_textstyle:D

272	<code>\name_primitive:NN \scriptstyle</code>	<code>\tex_scriptstyle:D</code>
273	<code>\name_primitive:NN \scriptscriptstyle</code>	<code>\tex_scriptscriptstyle:D</code>
274	<code>\name_primitive:NN \nonscript</code>	<code>\tex_nonscript:D</code>
275	<code>\name_primitive:NN \eqno</code>	<code>\tex_eqno:D</code>
276	<code>\name_primitive:NN \leqno</code>	<code>\tex_leqno:D</code>
277	<code>\name_primitive:NN \abovedisplayskip</code>	<code>\tex_abovedisplayskip:D</code>
278	<code>\name_primitive:NN \abovedisplayskip</code>	<code>\tex_abovedisplayskip:D</code>
279	<code>\name_primitive:NN \belowdisplayskip</code>	<code>\tex_belowdisplayskip:D</code>
280	<code>\name_primitive:NN \belowdisplayskip</code>	<code>\tex_belowdisplayskip:D</code>
281	<code>\name_primitive:NN \displaywidowpenalty</code>	<code>\tex_displaywidowpenalty:D</code>
282	<code>\name_primitive:NN \displayindent</code>	<code>\tex_displayindent:D</code>
283	<code>\name_primitive:NN \displaywidth</code>	<code>\tex_displaywidth:D</code>
284	<code>\name_primitive:NN \everydisplay</code>	<code>\tex_everydisplay:D</code>
285	<code>\name_primitive:NN \predisplaysize</code>	<code>\tex_predisplaysize:D</code>
286	<code>\name_primitive:NN \predisplaypenalty</code>	<code>\tex_predisplaypenalty:D</code>
287	<code>\name_primitive:NN \postdisplaypenalty</code>	<code>\tex_postdisplaypenalty:D</code>
288	<code>\name_primitive:NN \mathbin</code>	<code>\tex_mathbin:D</code>
289	<code>\name_primitive:NN \mathclose</code>	<code>\tex_mathclose:D</code>
290	<code>\name_primitive:NN \mathinner</code>	<code>\tex_mathinner:D</code>
291	<code>\name_primitive:NN \mathop</code>	<code>\tex_mathop:D</code>
292	<code>\name_primitive:NN \displaylimits</code>	<code>\tex_displaylimits:D</code>
293	<code>\name_primitive:NN \limits</code>	<code>\tex_limits:D</code>
294	<code>\name_primitive:NN \nolimits</code>	<code>\tex_nolimits:D</code>
295	<code>\name_primitive:NN \mathopen</code>	<code>\tex_mathopen:D</code>
296	<code>\name_primitive:NN \mathord</code>	<code>\tex_mathord:D</code>
297	<code>\name_primitive:NN \mathpunct</code>	<code>\tex_mathpunct:D</code>
298	<code>\name_primitive:NN \mathrel</code>	<code>\tex_mathrel:D</code>
299	<code>\name_primitive:NN \overline</code>	<code>\tex_overline:D</code>
300	<code>\name_primitive:NN \underline</code>	<code>\tex_underline:D</code>
301	<code>\name_primitive:NN \left</code>	<code>\tex_left:D</code>
302	<code>\name_primitive:NN \right</code>	<code>\tex_right:D</code>
303	<code>\name_primitive:NN \binoppenalty</code>	<code>\tex_binoppenalty:D</code>
304	<code>\name_primitive:NN \relpenalty</code>	<code>\tex_relpenalty:D</code>
305	<code>\name_primitive:NN \delimitershortfall</code>	<code>\tex_delimitershortfall:D</code>
306	<code>\name_primitive:NN \delimiterfactor</code>	<code>\tex_delimiterfactor:D</code>
307	<code>\name_primitive:NN \nulldelimiterspace</code>	<code>\tex_nulldelimiterspace:D</code>
308	<code>\name_primitive:NN \everymath</code>	<code>\tex_everymath:D</code>
309	<code>\name_primitive:NN \mathsurround</code>	<code>\tex_mathsurround:D</code>
310	<code>\name_primitive:NN \medmuskip</code>	<code>\tex_medmuskip:D</code>
311	<code>\name_primitive:NN \thinmuskip</code>	<code>\tex_thinmuskip:D</code>
312	<code>\name_primitive:NN \thickmuskip</code>	<code>\tex_thickmuskip:D</code>
313	<code>\name_primitive:NN \scriptspace</code>	<code>\tex_scriptspace:D</code>
314	<code>\name_primitive:NN \noboundary</code>	<code>\tex_noboundary:D</code>
315	<code>\name_primitive:NN \accent</code>	<code>\tex_accent:D</code>
316	<code>\name_primitive:NN \char</code>	<code>\tex_char:D</code>
317	<code>\name_primitive:NN \discretionary</code>	<code>\tex_discretionary:D</code>
318	<code>\name_primitive:NN \hfil</code>	<code>\tex_hfil:D</code>
319	<code>\name_primitive:NN \hfilneg</code>	<code>\tex_hfilneg:D</code>
320	<code>\name_primitive:NN \hfill</code>	<code>\tex_hfill:D</code>
321	<code>\name_primitive:NN \hskip</code>	<code>\tex_hskip:D</code>

322	\name_primitive:NN	\hss	\tex_hss:D
323	\name_primitive:NN	\vfil	\tex_vfil:D
324	\name_primitive:NN	\vfilneg	\tex_vfilneg:D
325	\name_primitive:NN	\vfill	\tex_vfill:D
326	\name_primitive:NN	\vskip	\tex_vskip:D
327	\name_primitive:NN	\vss	\tex_vss:D
328	\name_primitive:NN	\unskip	\tex_unskip:D
329	\name_primitive:NN	\kern	\tex_kern:D
330	\name_primitive:NN	\unkern	\tex_unkern:D
331	\name_primitive:NN	\hrule	\tex_hrule:D
332	\name_primitive:NN	\vrule	\tex_vrule:D
333	\name_primitive:NN	\leaders	\tex_leaders:D
334	\name_primitive:NN	\cleaders	\tex_cleaders:D
335	\name_primitive:NN	\xleaders	\tex_xleaders:D
336	\name_primitive:NN	\lastkern	\tex_lastkern:D
337	\name_primitive:NN	\lastskip	\tex_lastskip:D
338	\name_primitive:NN	\indent	\tex_indent:D
339	\name_primitive:NN	\par	\tex_par:D
340	\name_primitive:NN	\noindent	\tex_noindent:D
341	\name_primitive:NN	\vadjust	\tex_vadjust:D
342	\name_primitive:NN	\baselineskip	\tex_baselineskip:D
343	\name_primitive:NN	\lineskip	\tex_lineskip:D
344	\name_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
345	\name_primitive:NN	\clubpenalty	\tex_clubpenalty:D
346	\name_primitive:NN	\widowpenalty	\tex_widowpenalty:D
347	\name_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
348	\name_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
349	\name_primitive:NN	\linepenalty	\tex_linepenalty:D
350	\name_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
351	\name_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
352	\name_primitive:NN	\adjdemerits	\tex_adjdemerits:D
353	\name_primitive:NN	\hangafter	\tex_hangafter:D
354	\name_primitive:NN	\hangindent	\tex_hangindent:D
355	\name_primitive:NN	\parshape	\tex_parshape:D
356	\name_primitive:NN	\hsize	\tex_hsize:D
357	\name_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
358	\name_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
359	\name_primitive:NN	\leftskip	\tex_leftskip:D
360	\name_primitive:NN	\rightskip	\tex_rightskip:D
361	\name_primitive:NN	\looseness	\tex_looseness:D
362	\name_primitive:NN	\parskip	\tex_parskip:D
363	\name_primitive:NN	\parindent	\tex_parindent:D
364	\name_primitive:NN	\uchyph	\tex_uchyph:D
365	\name_primitive:NN	\emergencystretch	\tex_emergencystretch:D
366	\name_primitive:NN	\pretolerance	\tex_pretolerance:D
367	\name_primitive:NN	\tolerance	\tex_tolerance:D
368	\name_primitive:NN	\spaceskip	\tex_spaceskip:D
369	\name_primitive:NN	\xspaceskip	\tex_xspaceskip:D
370	\name_primitive:NN	\parfillskip	\tex_parfillskip:D
371	\name_primitive:NN	\everypar	\tex_everypar:D

372	<code>\name_primitive:NN \prevgraf</code>	<code>\tex_prevgraf:D</code>
373	<code>\name_primitive:NN \spacefactor</code>	<code>\tex_spacefactor:D</code>
374	<code>\name_primitive:NN \shipout</code>	<code>\tex_shipout:D</code>
375	<code>\name_primitive:NN \vsize</code>	<code>\tex_vsize:D</code>
376	<code>\name_primitive:NN \interlinepenalty</code>	<code>\tex_interlinepenalty:D</code>
377	<code>\name_primitive:NN \brokenpenalty</code>	<code>\tex_brokenpenalty:D</code>
378	<code>\name_primitive:NN \topskip</code>	<code>\tex_topskip:D</code>
379	<code>\name_primitive:NN \maxdeadcycles</code>	<code>\tex_maxdeadcycles:D</code>
380	<code>\name_primitive:NN \maxdepth</code>	<code>\tex_maxdepth:D</code>
381	<code>\name_primitive:NN \output</code>	<code>\tex_output:D</code>
382	<code>\name_primitive:NN \deadcycles</code>	<code>\tex_deadcycles:D</code>
383	<code>\name_primitive:NN \pagedepth</code>	<code>\tex_pagedepth:D</code>
384	<code>\name_primitive:NN \pagestretch</code>	<code>\tex_pagestretch:D</code>
385	<code>\name_primitive:NN \pagefilstretch</code>	<code>\tex_pagefilstretch:D</code>
386	<code>\name_primitive:NN \pagefillstretch</code>	<code>\tex_pagefillstretch:D</code>
387	<code>\name_primitive:NN \pagefilllstretch</code>	<code>\tex_pagefilllstretch:D</code>
388	<code>\name_primitive:NN \pageshrink</code>	<code>\tex_pageshrink:D</code>
389	<code>\name_primitive:NN \pagegoal</code>	<code>\tex_pagegoal:D</code>
390	<code>\name_primitive:NN \pagetotal</code>	<code>\tex_pagetotal:D</code>
391	<code>\name_primitive:NN \outputpenalty</code>	<code>\tex_outputpenalty:D</code>
392	<code>\name_primitive:NN \hoffset</code>	<code>\tex_hoffset:D</code>
393	<code>\name_primitive:NN \voffset</code>	<code>\tex_voffset:D</code>
394	<code>\name_primitive:NN \insert</code>	<code>\tex_insert:D</code>
395	<code>\name_primitive:NN \holdinginserts</code>	<code>\tex_holdinginserts:D</code>
396	<code>\name_primitive:NN \floatingpenalty</code>	<code>\tex_floatingpenalty:D</code>
397	<code>\name_primitive:NN \insertpenalties</code>	<code>\tex_insertpenalties:D</code>
398	<code>\name_primitive:NN \lower</code>	<code>\tex_lower:D</code>
399	<code>\name_primitive:NN \moveleft</code>	<code>\tex_moveleft:D</code>
400	<code>\name_primitive:NN \moveright</code>	<code>\tex_moveright:D</code>
401	<code>\name_primitive:NN \raise</code>	<code>\tex_raise:D</code>
402	<code>\name_primitive:NN \copy</code>	<code>\tex_copy:D</code>
403	<code>\name_primitive:NN \lastbox</code>	<code>\tex_lastbox:D</code>
404	<code>\name_primitive:NN \vsplit</code>	<code>\tex_vsplit:D</code>
405	<code>\name_primitive:NN \unhbox</code>	<code>\tex_unhbox:D</code>
406	<code>\name_primitive:NN \unhcopy</code>	<code>\tex_unhcopy:D</code>
407	<code>\name_primitive:NN \unvbox</code>	<code>\tex_unvbox:D</code>
408	<code>\name_primitive:NN \unvcopy</code>	<code>\tex_unvcopy:D</code>
409	<code>\name_primitive:NN \setbox</code>	<code>\tex_setbox:D</code>
410	<code>\name_primitive:NN \hbox</code>	<code>\tex_hbox:D</code>
411	<code>\name_primitive:NN \vbox</code>	<code>\tex_vbox:D</code>
412	<code>\name_primitive:NN \vtop</code>	<code>\tex_vtop:D</code>
413	<code>\name_primitive:NN \prevdepth</code>	<code>\tex_prevdepth:D</code>
414	<code>\name_primitive:NN \badness</code>	<code>\tex_badness:D</code>
415	<code>\name_primitive:NN \hbadness</code>	<code>\tex_hbadness:D</code>
416	<code>\name_primitive:NN \vbadness</code>	<code>\tex_vbadness:D</code>
417	<code>\name_primitive:NN \hfuzz</code>	<code>\tex_hfuzz:D</code>
418	<code>\name_primitive:NN \vfuzz</code>	<code>\tex_vfuzz:D</code>
419	<code>\name_primitive:NN \overfullrule</code>	<code>\tex_overfullrule:D</code>
420	<code>\name_primitive:NN \boxmaxdepth</code>	<code>\tex_boxmaxdepth:D</code>
421	<code>\name_primitive:NN \splitmaxdepth</code>	<code>\tex_splitmaxdepth:D</code>

422	<code>\name_primitive:NN \splittopskip</code>	<code>\tex_splittopskip:D</code>
423	<code>\name_primitive:NN \everyhbox</code>	<code>\tex_everyhbox:D</code>
424	<code>\name_primitive:NN \everyvbox</code>	<code>\tex_everyvbox:D</code>
425	<code>\name_primitive:NN \nullfont</code>	<code>\tex_nullfont:D</code>
426	<code>\name_primitive:NN \textfont</code>	<code>\tex_textfont:D</code>
427	<code>\name_primitive:NN \scriptfont</code>	<code>\tex_scriptfont:D</code>
428	<code>\name_primitive:NN \scriptscriptfont</code>	<code>\tex_scriptscriptfont:D</code>
429	<code>\name_primitive:NN \fontdimen</code>	<code>\tex_fontdimen:D</code>
430	<code>\name_primitive:NN \hyphenchar</code>	<code>\tex_hyphenchar:D</code>
431	<code>\name_primitive:NN \skewchar</code>	<code>\tex_skewchar:D</code>
432	<code>\name_primitive:NN \defaultshyphenchar</code>	<code>\tex_defaultshyphenchar:D</code>
433	<code>\name_primitive:NN \defaultskewchar</code>	<code>\tex_defaultskewchar:D</code>
434	<code>\name_primitive:NN \number</code>	<code>\tex_number:D</code>
435	<code>\name_primitive:NN \romannumeral</code>	<code>\tex_romannumeral:D</code>
436	<code>\name_primitive:NN \string</code>	<code>\tex_string:D</code>
437	<code>\name_primitive:NN \lowercase</code>	<code>\tex_lowercase:D</code>
438	<code>\name_primitive:NN \uppercase</code>	<code>\tex_uppercase:D</code>
439	<code>\name_primitive:NN \meaning</code>	<code>\tex_meaning:D</code>
440	<code>\name_primitive:NN \penalty</code>	<code>\tex_penalty:D</code>
441	<code>\name_primitive:NN \unpenalty</code>	<code>\tex_unpenalty:D</code>
442	<code>\name_primitive:NN \lastpenalty</code>	<code>\tex_lastpenalty:D</code>
443	<code>\name_primitive:NN \special</code>	<code>\tex_special:D</code>
444	<code>\name_primitive:NN \dump</code>	<code>\tex_dump:D</code>
445	<code>\name_primitive:NN \patterns</code>	<code>\tex_patterns:D</code>
446	<code>\name_primitive:NN \hyphenation</code>	<code>\tex_hyphenation:D</code>
447	<code>\name_primitive:NN \time</code>	<code>\tex_time:D</code>
448	<code>\name_primitive:NN \day</code>	<code>\tex_day:D</code>
449	<code>\name_primitive:NN \month</code>	<code>\tex_month:D</code>
450	<code>\name_primitive:NN \year</code>	<code>\tex_year:D</code>
451	<code>\name_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
452	<code>\name_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
453	<code>\name_primitive:NN \count</code>	<code>\tex_count:D</code>
454	<code>\name_primitive:NN \dimen</code>	<code>\tex_dimen:D</code>
455	<code>\name_primitive:NN \skip</code>	<code>\tex_skip:D</code>
456	<code>\name_primitive:NN \toks</code>	<code>\tex_toks:D</code>
457	<code>\name_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
458	<code>\name_primitive:NN \box</code>	<code>\tex_box:D</code>
459	<code>\name_primitive:NN \wd</code>	<code>\tex_wd:D</code>
460	<code>\name_primitive:NN \ht</code>	<code>\tex_ht:D</code>
461	<code>\name_primitive:NN \dp</code>	<code>\tex_dp:D</code>
462	<code>\name_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
463	<code>\name_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
464	<code>\name_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
465	<code>\name_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
466	<code>\name_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
467	<code>\name_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

Since L<sup>A</sup>T<sub>E</sub>X3 requires at least the  $\varepsilon$ -T<sub>E</sub>X extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

468	\name_primitive:NN	\ifdefined	\etex_ifdefined:D
469	\name_primitive:NN	\ifcsname	\etex_ifcsname:D
470	\name_primitive:NN	\unless	\etex_unless:D
471	\name_primitive:NN	\eTeXversion	\etex_eTeXversion:D
472	\name_primitive:NN	\eTeXrevision	\etex_eTeXrevision:D
473	\name_primitive:NN	\marks	\etex_marks:D
474	\name_primitive:NN	\topmarks	\etex_topmarks:D
475	\name_primitive:NN	\firstmarks	\etex_firstmarks:D
476	\name_primitive:NN	\botmarks	\etex_botmarks:D
477	\name_primitive:NN	\splitfirstmarks	\etex_splitfirstmarks:D
478	\name_primitive:NN	\splitbotmarks	\etex_splitbotmarks:D
479	\name_primitive:NN	\unexpanded	\etex_unexpanded:D
480	\name_primitive:NN	\detokenize	\etex_detokenize:D
481	\name_primitive:NN	\scantokens	\etex_scantokens:D
482	\name_primitive:NN	\showtokens	\etex_showtokens:D
483	\name_primitive:NN	\readline	\etex_readline:D
484	\name_primitive:NN	\tracingassigns	\etex_tracingassigns:D
485	\name_primitive:NN	\tracingscantokens	\etex_tracingscantokens:D
486	\name_primitive:NN	\tracingnesting	\etex_tracingnesting:D
487	\name_primitive:NN	\tracingifs	\etex_tracingifs:D
488	\name_primitive:NN	\currentiflevel	\etex_currentiflevel:D
489	\name_primitive:NN	\currentifbranch	\etex_currentifbranch:D
490	\name_primitive:NN	\currentifttype	\etex_currentifttype:D
491	\name_primitive:NN	\tracinggroups	\etex_tracinggroups:D
492	\name_primitive:NN	\currentgrouplevel	\etex_currentgrouplevel:D
493	\name_primitive:NN	\currentgroupstype	\etex_currentgroupstype:D
494	\name_primitive:NN	\showgroups	\etex_showgroups:D
495	\name_primitive:NN	\showifs	\etex_showifs:D
496	\name_primitive:NN	\interactionmode	\etex_interactionmode:D
497	\name_primitive:NN	\lastnodetype	\etex_lastnodetype:D
498	\name_primitive:NN	\iffontchar	\etex_iffontchar:D
499	\name_primitive:NN	\fontcharht	\etex_fontcharht:D
500	\name_primitive:NN	\fontchardp	\etex_fontchardp:D
501	\name_primitive:NN	\fontcharwd	\etex_fontcharwd:D
502	\name_primitive:NN	\fontcharic	\etex_fontcharic:D
503	\name_primitive:NN	\parshapeindent	\etex_parshapeindent:D
504	\name_primitive:NN	\parshapelength	\etex_parshapelength:D
505	\name_primitive:NN	\parshapedimen	\etex_parshapedimen:D
506	\name_primitive:NN	\numexpr	\etex_numexpr:D
507	\name_primitive:NN	\dimexpr	\etex_dimexpr:D
508	\name_primitive:NN	\glueexpr	\etex_glueexpr:D
509	\name_primitive:NN	\muexpr	\etex_muexpr:D
510	\name_primitive:NN	\gluestretch	\etex_gluestretch:D
511	\name_primitive:NN	\glueshrink	\etex_glueshrink:D
512	\name_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
513	\name_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
514	\name_primitive:NN	\gluetomu	\etex_gluetomu:D
515	\name_primitive:NN	\mutoglu	\etex_mutoglu:D
516	\name_primitive:NN	\lastlinefit	\etex_lastlinefit:D
517	\name_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D

```

518 \name_primitive:NN \clubpenalties \etex_clubpenalties:D
519 \name_primitive:NN \widowpenalties \etex_widowpenalties:D
520 \name_primitive:NN \displaywidowpenalties \etex_displaywidowpenalties:D
521 \name_primitive:NN \middle \etex_middle:D
522 \name_primitive:NN \savinghyphcodes \etex_savinghyphcodes:D
523 \name_primitive:NN \savingvdiscards \etex_savingvdiscards:D
524 \name_primitive:NN \pagediscards \etex_pagediscards:D
525 \name_primitive:NN \splitdiscards \etex_splitdiscards:D
526 \name_primitive:NN \TeXETstate \etex_TeXXETstate:D
527 \name_primitive:NN \beginL \etex_beginL:D
528 \name_primitive:NN \endL \etex_endL:D
529 \name_primitive:NN \beginR \etex_beginR:D
530 \name_primitive:NN \endR \etex_endR:D
531 \name_primitive:NN \predisplaydirection \etex_predisplaydirection:D
532 \name_primitive:NN \everyeof \etex_everyeof:D
533 \name_primitive:NN \protected \etex_protected:D

```

All major distributions use pdf $\epsilon$ -TeX as engine so we add these names as well. Since the pdfTeX team has been very good at prefixing most primitives with pdf (so far only five do not start with pdf) we do not give then a double pdf prefix. The list below covers pdfTeXv 1.30.4.

```

534 %% integer registers:
535 \name_primitive:NN \pdfoutput \pdf_output:D
536 \name_primitive:NN \pdfminorversion \pdf_minorversion:D
537 \name_primitive:NN \pdfcompresslevel \pdf_compresslevel:D
538 \name_primitive:NN \pdfdecimaldigits \pdf_decimaldigits:D
539 \name_primitive:NN \pdfimageresolution \pdf_imageresolution:D
540 \name_primitive:NN \pdfpkresolution \pdf_pkresolution:D
541 \name_primitive:NN \pdftracingfonts \pdf_tracingfonts:D
542 \name_primitive:NN \pdfuniqueresname \pdf_uniqueresname:D
543 \name_primitive:NN \pdfadjustspacing \pdf_adjustspacing:D
544 \name_primitive:NN \pdfprotrudechars \pdf_protrudechars:D
545 \name_primitive:NN \efcode \pdf_efcode:D
546 \name_primitive:NN \lpcode \pdf_lpcode:D
547 \name_primitive:NN \rprcode \pdf_rprcode:D
548 \name_primitive:NN \pdfforcepagebox \pdf_forcepagebox:D
549 \name_primitive:NN \pdfoptionalwaysusepdfpagebox \pdf_optionalwaysusepdfpagebox:D
550 \name_primitive:NN \pdfinclusionerrorlevel \pdf_inclusionerrorlevel:D
551 \name_primitive:NN \pdfoptionpdfinclusionerrorlevel \pdf_optionpdfinclusionerrorlevel:D
552 \name_primitive:NN \pdfimagehicolor \pdf_imagehicolor:D
553 \name_primitive:NN \pdfimageapplygamma \pdf_imageapplygamma:D
554 \name_primitive:NN \pdfgamma \pdf_gamma:D
555 \name_primitive:NN \pdfimagegamma \pdf_imagegamma:D
556 %% dimen registers:
557 \name_primitive:NN \pdfhorigin \pdf_horigin:D
558 \name_primitive:NN \pdfvorigin \pdf_vorigin:D
559 \name_primitive:NN \pdfpagewidth \pdf_pagewidth:D
560 \name_primitive:NN \pdfpageheight \pdf_pageheight:D
561 \name_primitive:NN \pdflinkmargin \pdf_linkmargin:D

```

562	\name_primitive:NN	\pdfdestmargin	\pdf_destmargin:D
563	\name_primitive:NN	\pdfthreadmargin	\pdf_threadmargin:D
564	%% token registers:		
565	\name_primitive:NN	\pdfpagesattr	\pdf_pagesattr:D
566	\name_primitive:NN	\pdfpageattr	\pdf_pageattr:D
567	\name_primitive:NN	\pdfpageresources	\pdf_pageresources:D
568	\name_primitive:NN	\pdfpkmode	\pdf_pkmode:D
569	%% expandable commands:		
570	\name_primitive:NN	\pdftexrevision	\pdf_texrevision:D
571	\name_primitive:NN	\pdftexbanner	\pdf_texbanner:D
572	\name_primitive:NN	\pdfcreationdate	\pdf_creationdate:D
573	\name_primitive:NN	\pdfpageref	\pdf_pageref:D
574	\name_primitive:NN	\pdfxformname	\pdf_xformname:D
575	\name_primitive:NN	\pdffontname	\pdf_fontname:D
576	\name_primitive:NN	\pdffontobjnum	\pdf_fontobjnum:D
577	\name_primitive:NN	\pdffontsize	\pdf_fontsize:D
578	\name_primitive:NN	\pdfincludechars	\pdf_includechars:D
579	\name_primitive:NN	\leftmarginkern	\pdf_leftmarginkern:D
580	\name_primitive:NN	\rightmarginkern	\pdf_rightmarginkern:D
581	\name_primitive:NN	\pdfescapestring	\pdf_escapestring:D
582	\name_primitive:NN	\pdfescapename	\pdf_escapename:D
583	\name_primitive:NN	\pdfescapehex	\pdf_escapehex:D
584	\name_primitive:NN	\pdfunescapehex	\pdf_unescapehex:D
585	\name_primitive:NN	\pdfstrcmp	\pdf_strcmp:D
586	\name_primitive:NN	\pdfuniformdeviate	\pdf_uniformdeviate:D
587	\name_primitive:NN	\pdfnormaldeviate	\pdf_normaldeviate:D
588	\name_primitive:NN	\pdfmdfivesum	\pdf_mdfivesum:D
589	\name_primitive:NN	\pdffilemoddate	\pdf_filemoddate:D
590	\name_primitive:NN	\pdffilesize	\pdf_filesize:D
591	\name_primitive:NN	\pdffiledump	\pdf_filedump:D
592	%% read-only integers:		
593	\name_primitive:NN	\pdftexversion	\pdf_texversion:D
594	\name_primitive:NN	\pdflastobj	\pdf_lastobj:D
595	\name_primitive:NN	\pdflastxform	\pdf_lastxform:D
596	\name_primitive:NN	\pdflastximage	\pdf_lastximage:D
597	\name_primitive:NN	\pdflastximagepages	\pdf_lastximagepages:D
598	\name_primitive:NN	\pdflastannot	\pdf_lastannot:D
599	\name_primitive:NN	\pdflastxpos	\pdf_lastxpos:D
600	\name_primitive:NN	\pdflastypos	\pdf_lastypos:D
601	\name_primitive:NN	\pdflastdemerits	\pdf_lastdemerits:D
602	\name_primitive:NN	\pdfelapsedtime	\pdf_elapsedtime:D
603	\name_primitive:NN	\pdfrandomseed	\pdf_randomseed:D
604	\name_primitive:NN	\pdfshellescape	\pdf_shellescape:D
605	%% general commands:		
606	\name_primitive:NN	\pdfobj	\pdf_obj:D
607	\name_primitive:NN	\pdfrefobj	\pdf_refobj:D
608	\name_primitive:NN	\pdfxform	\pdf_xform:D
609	\name_primitive:NN	\pdfrefxform	\pdf_refxform:D
610	\name_primitive:NN	\pdfximage	\pdf_ximage:D
611	\name_primitive:NN	\pdfrefximage	\pdf_refximage:D

```

612 \name_primitive:NN \pdfannot \pdf_annot:D
613 \name_primitive:NN \pdfstartlink \pdf_startlink:D
614 \name_primitive:NN \pdfendlink \pdf_endlink:D
615 \name_primitive:NN \pdfoutline \pdf_outline:D
616 \name_primitive:NN \pdfdest \pdf_dest:D
617 \name_primitive:NN \pdfthread \pdf_thread:D
618 \name_primitive:NN \pdfstartthread \pdf_startthread:D
619 \name_primitive:NN \pdfendthread \pdf_endthread:D
620 \name_primitive:NN \pdfsavepos \pdf_savepos:D
621 \name_primitive:NN \pdfinfo \pdf_info:D
622 \name_primitive:NN \pdfcatalog \pdf_catalog:D
623 \name_primitive:NN \pdfnames \pdf_names:D
624 \name_primitive:NN \pdfmapfile \pdf_mapfile:D
625 \name_primitive:NN \pdfmapline \pdf_mapline:D
626 \name_primitive:NN \pdffontattr \pdf_fontattr:D
627 \name_primitive:NN \pdftrailer \pdf_trailer:D
628 \name_primitive:NN \pdffontexpand \pdf_fontexpand:D
629 %%\name_primitive:NN \vadjust [<pre spec>] <filler> { <vertical mode material> } (h, m)
630 \name_primitive:NN \pdfliteral \pdf_literal:D
631 %%\name_primitive:NN \special <pdfspecial spec>
632 \name_primitive:NN \pdfresettimer \pdf_resettimer:D
633 \name_primitive:NN \pdfsetrandomseed \pdf_setrandomseed:D
634 \name_primitive:NN \pdfnoligatures \pdf_noligatures:D

```

Only a little bit of X<sub>Y</sub>T<sub>E</sub>X and LuaT<sub>E</sub>X at the moment.

```

635 \name_primitive:NN \XeTeXversion \xetex_version:D
636 \name_primitive:NN \catcodetable \luatex_catcodetable:D
637 \name_primitive:NN \directlua \luatex_directlua:D
638 \name_primitive:NN \initcatcodetable \luatex_initcatcodetable:D
639 \name_primitive:NN \latelua \luatex_latelua:D
640 \name_primitive:NN \savecatcodetable \luatex_savecatcodetable:D

```

XeTeX adds `\strcmp` to the set of primitives, with the same implementation as `\pdfstrcmp` but a different name. To avoid having to worry about this later, the same internal name is used. Note that we want to use the original primitive name here so that `\name_undefine:N` (if used) will remove it.

```

641 \tex_begingroup:D
642 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
643 \tex_endgroup:D
644 \tex_expandafter:D \tex_ifx:D \tex_csname:D xetex_version:D\tex_endcsname:D
645 \tex_relax:D \tex_else:D
646 \name_primitive:NN \strcmp \pdf_strcmp:D
647 \tex_fi:D

```

## 95.7 expl3 code switches

`\ExplSyntaxOn` Here we define functions that are used to turn on and off the special conventions used in  
`\ExplSyntaxOff` the kernel of L<sup>A</sup>T<sub>E</sub>X3.  
`\ExplSyntaxStatus`

First of all, the space, tab and the return characters will all be ignored inside L<sup>A</sup>T<sub>E</sub>X3 code, the latter because endlines are set to a space instead. When space characters are needed in L<sup>A</sup>T<sub>E</sub>X3 code the ~ character will be used for that purpose.

Specification of the desired behavior:

- ExplSyntax can be either On or Off.
- The On switch is *<null>* if ExplSyntax is on.
- The Off switch is *<null>* if ExplSyntax is off.
- If the On switch is issued and not *<null>*, it records the current catcode scheme just prior to it being issued.
- An Off switch restores the catcode scheme to what it was just prior to the previous On switch.

```

648 \etex_protected:D \tex_def:D \ExplSyntaxOn {
649   \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
650   \tex_else:D
651     \etex_protected:D \tex_edef:D \ExplSyntaxOff {
652       \etex_unexpanded:D{
653         \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
654         \tex_def:D \ExplSyntaxStatus{0}
655       }
656       \tex_catcode:D 126=\tex_the:D \tex_catcode:D 126 \tex_relax:D
657       \tex_catcode:D 32=\tex_the:D \tex_catcode:D 32 \tex_relax:D
658       \tex_catcode:D 9=\tex_the:D \tex_catcode:D 9 \tex_relax:D
659       \tex_endlinechar:D =\tex_the:D \tex_endlinechar:D \tex_relax:D
660       \tex_catcode:D 95=\tex_the:D \tex_catcode:D 95 \tex_relax:D
661       \tex_catcode:D 58=\tex_the:D \tex_catcode:D 58 \tex_relax:D
662       \tex_catcode:D 124=\tex_the:D \tex_catcode:D 124 \tex_relax:D
663       \tex_catcode:D 38=\tex_the:D \tex_catcode:D 38 \tex_relax:D
664       \tex_catcode:D 94=\tex_the:D \tex_catcode:D 94 \tex_relax:D
665       \tex_catcode:D 34=\tex_the:D \tex_catcode:D 34 \tex_relax:D
666       \tex_noexpand:D \tex_fi:D
667     }
668     \tex_def:D \ExplSyntaxStatus { 1 }
669     \tex_catcode:D 126=10 \tex_relax:D % tilde is a space char.
670     \tex_catcode:D 32=9 \tex_relax:D % space is ignored
671     \tex_catcode:D 9=9 \tex_relax:D % tab also ignored
672     \tex_endlinechar:D =32 \tex_relax:D % endlines are space
673     \tex_catcode:D 95=11 \tex_relax:D % underscore letter
674     \tex_catcode:D 58=11 \tex_relax:D % colon letter
675     \tex_catcode:D 124=12 \tex_relax:D % vertical bar, other
676     \tex_catcode:D 38=4 \tex_relax:D % ampersand, alignment token
677     \tex_catcode:D 94=7 \tex_relax:D % caret, math superscript
678     \tex_catcode:D 34=12 \tex_relax:D % doublequote, other
679   \tex_fi:D
680 }

```

At this point we better set the status.

```
681 \tex_def:D \ExplSyntaxStatus { 1 }
```

`\ExplSyntaxNamesOn` Sometimes we need to be able to use names from the kernel of L<sup>A</sup>T<sub>E</sub>X3 without adhering it's  
`\ExplSyntaxNamesOff` conventions according to space characters. These macros provide the necessary settings.

```
682 \etex_protected:D \tex_def:D \ExplSyntaxNamesOn {
683   \tex_catcode:D '\_ =11\tex_relax:D
684   \tex_catcode:D '\: =11\tex_relax:D
685 }
686 \etex_protected:D \tex_def:D \ExplSyntaxNamesOff {
687   \tex_catcode:D '\_ =8\tex_relax:D
688   \tex_catcode:D '\: =12\tex_relax:D
689 }
```

## 95.8 Package loading

<code>\GetIdInfo</code>	Extract all information from a cvs or svn field. The formats are slightly different but
<code>\filedescription</code>	at least the information is in the same positions so we check in the date format so see
<code>\filename</code>	if it contains a / after the four-digit year. If it does it is cvs else svn and we extract
<code>\fileversion</code>	information. To be on the safe side we ensure that spaces in the argument are seen.
<code>\fileauthor</code>	
<code>\filedate</code>	
<code>\filenameext</code>	
<code>\filetimestamp</code>	
<code>\GetIdInfoAuxI:w</code>	
<code>\GetIdInfoAuxII:w</code>	
<code>\GetIdInfoAuxCVS:w</code>	
<code>\GetIdInfoAuxSVN:w</code>	

```

690 \etex_protected:D \tex_def:D \GetIdInfo {
691   \tex_begingroup:D
692   \tex_catcode:D 32=10 \tex_relax:D % needed? Probably for now.
693   \GetIdInfoMaybeMissing:w
694 }
695 \etex_protected:D \tex_def:D \GetIdInfoMaybeMissing:w$#1$#2{
696   \tex_def:D \l_kernel_tmpa_tl {#1}
697   \tex_def:D \l_kernel_tmpb_tl {Id}
698   \tex_ifx:D \l_kernel_tmpa_tl \l_kernel_tmpb_tl
699     \tex_def:D \l_kernel_tmpa_tl {
700       \tex_endgroup:D
701       \tex_def:D\filedescription{#2}
702       \tex_def:D\filename      {[unknown~name]}
703       \tex_def:D\fileversion   {000}
704       \tex_def:D\fileauthor    {[unknown~author]}
705       \tex_def:D\filedate      {0000/00/00}
706       \tex_def:D\filenameext    {[unknown~ext]}
707       \tex_def:D\filetimestamp {[unknown~timestamp]}
708     }
709   \tex_else:D
710     \tex_def:D \l_kernel_tmpa_tl {\GetIdInfoAuxI:w$#1$#2}
711   \tex_fi:D
712   \l_kernel_tmpa_tl
713 }
```

```

714 \etex_protected:D \tex_def:D\GetIdInfoAuxii:w$#1~#2.#3~#4~#5~#6~#7~#8$#9{
715   \tex_endgroup:D
716   \tex_def:D\filename{#2}
717   \tex_def:D\fileversion{#4}
718   \tex_def:D\filedescription{#9}
719   \tex_def:D\fileauthor{#7}
720   \GetIdInfoAuxii:w #5\tex_relax:D
721   #3\tex_relax:D#5\tex_relax:D#6\tex_relax:D
722 }

723 \etex_protected:D \tex_def:D\GetIdInfoAuxii:w #1#2#3#4#5#6\tex_relax:D{
724   \tex_ifx:D#5/
725     \tex_expandafter:D\GetIdInfoAuxCVS:w
726   \tex_else:D
727     \tex_expandafter:D\GetIdInfoAuxSVN:w
728   \tex_fi:D
729 }

730 \etex_protected:D \tex_def:D\GetIdInfoAuxCVS:w #1,v\tex_relax:D
731                                     #2\tex_relax:D#3\tex_relax:D{
732   \tex_def:D\filedate{#2}
733   \tex_def:D\filenameext{#1}
734   \tex_def:D\filetimestamp{#3}

```

When creating the format we want the information in the log straight away.

```

735 <initex>\tex_immediate:D\tex_write:D-1
736 <initex> {\filename;~ v\fileversion,~\filedate;~\filedescription}
737 }
738 \etex_protected:D \tex_def:D\GetIdInfoAuxSVN:w #1\tex_relax:D#2~#3~#4
739                                     \tex_relax:D#5Z\tex_relax:D{
740   \tex_def:D\filenameext{#1}
741   \tex_def:D\filedate{#2/#3/#4}
742   \tex_def:D\filetimestamp{#5}
743 <-package>\tex_immediate:D\tex_write:D-1
744 <-package> {\filename;~ v\fileversion,~\filedate;~\filedescription}
745 }
746 </initex | package>

```

Finally some corrections in the case we are running over  $\text{\LaTeX} 2_{\epsilon}$ .

We want to set things up so that experimental packages and regular packages can coexist with the former using the  $\text{\LaTeX} 3$  programming catcode settings. Since it cannot be the task of the end user to know how a package is constructed under the hood we make it so that the experimental packages have to identify themselves. As an example it can be done as

```

\RequirePackage{l3names}
\ProvidesExplPackage{agent}{2007/08/28}{007}{bonding module}

```

or by using the `\file{field}` informations from `\GetIdInfo` as the packages in this distribution do like this:

```
\RequirePackage{l3names}
\GetIdInfo$Id: l3names.dtx 2064 2010-10-03 23:27:33Z will $
    {L3 Experimental Box module}
\ProvidesExplPackage
    {\filename}{\filedate}{\fileversion}{\filedescription}
```

`\ProvidesExplPackage` First up is the identification. Rather trivial as we don't allow for options just yet.

```
\ProvidesExplClass
\ProvidesExplFile
747  \etex_protected:D \tex_def:D \ProvidesExplPackage#1#2#3#4{
748    \ProvidesPackage{#1}[#2~v#3~#4]
749    \ExplSyntaxOn
750  }
751  \etex_protected:D \tex_def:D \ProvidesExplClass#1#2#3#4{
752    \ProvidesClass{#1}[#2~v#3~#4]
753    \ExplSyntaxOn
754  }
755  \etex_protected:D \tex_def:D \ProvidesExplFile#1#2#3#4{
756    \ProvidesFile{#1}[#2~v#3~#4]
757    \ExplSyntaxOn
758  }
759 }
```

`\@pushfilename` The idea behind the code is to record whether or not the  $\text{\LaTeX}$ 3 syntax is on or off when  
`\@popfilename` about to load a file with class or package extension. This status stored in the parameter `\ExplSyntaxStatus` and set by `\ExplSyntaxOn` and `\ExplSyntaxOff` to 1 and 0 respectively is pushed onto the stack `\ExplSyntaxStack`. Then the catcodes are set back to normal, the file loaded with its options and finally the stack is popped again. The whole thing is a bit problematical. So let's take a look at what the desired behavior is: A package or class which declares itself of Expl type by using `\ProvidesExplClass` or `\ProvidesExplPackage` should automatically ensure the correct catcode scheme as soon as the identification part is over. Similarly, a package or class which uses the traditional `\ProvidesClass` or `\ProvidesPackage` commands should go back to the traditional catcode scheme. An example:

```
\RequirePackage{l3names}
\ProvidesExplPackage{foobar}{2009/05/07}{0.1}{Foobar package}
\cs_new:Npn \foo_bar:nn #1#2 {#1,#2}
...
\RequirePackage{array}
...
\cs_new:Npn \foo_bar:nnn #1#2#3 {#3,#2,#1}
```

Inside the `array` package, everything should behave as normal under traditional  $\text{\LaTeX}$  but as soon as we are back at the top level, we should use the new catcode regime.

Whenever L<sup>A</sup>T<sub>E</sub>X inputs a package file or similar, it calls upon `\@pushfilename` to push the name, the extension and the catcode of `@` of the file it was currently processing onto a file name stack. Similarly, after inputting such a file, this file name stack is popped again and the catcode of `@` is set to what it was before. If it is a package within package, `@` maintains catcode 11 whereas if it is package within document preamble `@` is reset to what it was in the preamble (which is usually catcode 12). We wish to adopt a similar technique. Every time an Expl package or class is declared, they will issue an `ExplSyntaxOn`. Then whenever we are about to load another file, we will first push this status onto a stack and then turn it off again. Then when done loading a file, we pop the stack and if `ExplSyntax` was On right before, so should it be now. The only problem with this is that we cannot guarantee that we get to the file name stack very early on. Therefore, if the `ExplSyntaxStack` is empty when trying to pop it, we ensure to turn `ExplSyntax` off again.

`\@pushfilename` is prepended with a small function pushing the current `ExplSyntaxStatus` (true/false) onto a stack. Then the current catcode regime is recorded and `ExplSyntax` is switched off.

`\@popfilename` is appended with a function for popping the `ExplSyntax` stack. However, chances are we didn't get to hook into the file stack early enough so L<sup>A</sup>T<sub>E</sub>X might try to pop the file name stack while the `ExplSyntaxStack` is empty. If the latter is empty, we just switch off `ExplSyntax`.

```

760 \tex_edef:D \@pushfilename{
761   \etex_unexpanded:D{
762     \tex_edef:D \ExplSyntaxStack{ \ExplSyntaxStatus \ExplSyntaxStack }
763     \ExplSyntaxOff
764   }
765   \etex_unexpanded:D\tex_expandafter:D{\@pushfilename }
766 }
767 \tex_edef:D \@popfilename{
768   \etex_unexpanded:D\tex_expandafter:D{\@popfilename
769     \tex_if:D 2\ExplSyntaxStack 2
770     \ExplSyntaxOff
771     \tex_else:D
772       \tex_expandafter:D\ExplSyntaxPopStack\ExplSyntaxStack\q_stop
773     \tex_fi:D
774   }
775 }
```

`\ExplSyntaxPopStack` Popping the stack is simple: Take the first token which is either 0 (false) or 1 (true) and  
`\ExplSyntaxStack` test if it is odd. Save the rest. The stack is initially empty set to 0 signalling that before `l3names` was loaded, the `ExplSyntax` was off.

```

776 \etex_protected:D\tex_def:D\ExplSyntaxPopStack#1#2\q_stop{
777   \tex_def:D\ExplSyntaxStack{#2}
778   \tex_ifodd:D#1\tex_relax:D
779   \ExplSyntaxOn
780   \tex_else:D
```

```

781 \ExplSyntaxOff
782 \tex_fi:D
783 }
784 \tex_def:D \ExplSyntaxStack{0}

```

## 95.9 Finishing up

A few of the ‘primitives’ assigned above have already been stolen by L<sup>A</sup>T<sub>E</sub>X, so assign them by hand to the saved real primitive.

```

785 \tex_let:D\tex_input:D \@@input
786 \tex_let:D\tex_underline:D \@@underline
787 \tex_let:D\tex_end:D \@@end
788 \tex_let:D\tex_everymath:D \frozen@everymath
789 \tex_let:D\tex_everydisplay:D \frozen@everydisplay
790 \tex_let:D\tex_italiccorr:D \@@italiccorr
791 \tex_let:D\tex_hyphen:D \@@hyph
792 \tex_let:D\luatex_catcodetable:D \luatexcatcodetable
793 \tex_let:D\luatex_initcatcodetable:D \luatexinitcatcodetable
794 \tex_let:D\luatex_savecatcodetable:D \luatexsavecatcodetable

```

T<sub>E</sub>X has a nasty habit of inserting a command with the name `\par` so we had better make sure that that command at least has a definition.

```

795 \tex_let:D\par \tex_par:D

```

This is the end for `l3names` when used on top of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>:

```

796 \tex_ifx:D\name_undefine:N\@gobble
797 \tex_def:D\name_pop_stack:w{}
798 \tex_else:D

```

But if traditional T<sub>E</sub>X code is disabled, do this...

As mentioned above, The L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> package mechanism will insert some code to handle the filename stack, and reset the package options, this code will die if the T<sub>E</sub>X primitives have gone, so skip past it and insert some equivalent code that will work.

First a version of `\ProvidesPackage` that can cope.

```

799 \tex_def:D\ProvidesPackage{
800 \tex_begingroup:D
801 \ExplSyntaxOff
802 \package_provides:w}

803 \tex_def:D\package_provides:w#1#2[#3]{
804 \tex_endgroup:D
805 \tex_immediate:D\tex_write:D-1{Package:~#1#2~#3}
806 \tex_expandafter:D\tex_xdef:D
807 \tex_csname:D ver@#1.sty\tex_endcsname:D{#1}}

```

In this case the catcode preserving stack is not maintained and `\ExplSyntaxOn` conventions stay in force once on. You'll need to turn them off explicitly with `\ExplSyntaxOff` (although as currently built on 2e, nothing except very experimental code will run in this mode!) Also note that `\RequirePackage` is a simple definition, just for one file, with no options.

```

808 \tex_def:D\name_pop_stack:w#1\relax{%
809   \ExplSyntaxOff
810   \tex_expandafter:D\@p@pfilename\@currnamestack\@nil
811   \tex_let:D\default@ds\@unknownoptionerror
812   \tex_global:D\tex_let:D\ds@\@empty
813   \tex_global:D\tex_let:D\@declaredoptions\@empty}

814 \tex_def:D\@p@pfilename#1#2#3#4\@nil{%
815   \tex_gdef:D\@currname{#1}%
816   \tex_gdef:D\@current{#2}%
817   \tex_catcode:D'\@#3%
818   \tex_gdef:D\@currnamestack{#4}}

819 \tex_def:D\NeedsTeXFormat#1{}
820 \tex_def:D\RequirePackage#1{
821   \tex_expandafter:D\tex_ifx:D
822     \tex_csname:D ver@#1.sty\tex_endcsname:D\tex_relax:D
823     \ExplSyntaxOn
824     \tex_input:D#1.sty\tex_relax:D
825   \tex_fi:D}
826 \tex_fi:D

```

The `\futurelet` just forces the special end of file marker to vanish, so the argument of `\name_pop_stack:w` does not cause an end-of-file error. (Normally I use `\expandafter` for this trick, but here the next token is in fact `\let` and that may be undefined.)

```

827 \tex_futurelet:D\name_tmp:\name_pop_stack:w

```

**expl3 dependency checks** We want the `expl3` bundle to be loaded ‘as one’; this command is used to ensure that one of the 13 packages isn’t loaded on its own.

```

828 <*\initex>
829 \etex_protected:D\tex_def:D \package_check_loaded_expl: {
830   \ifpackageloaded{expl3}{}{
831     \PackageError{expl3}{Cannot~load~the~expl3~modules~separately}{
832       The~expl3~modules~cannot~be~loaded~separately;\MessageBreak
833       please~\protect\usepackage{expl3}~instead.
834     }
835   }
836 }
837 </\initex>

838 </\package>

```

## 95.10 Showing memory usage

This section is from some old code from 1993; it'd be good to work out how it should be used in our code today.

During the development of the L<sup>A</sup>T<sub>E</sub>X3 kernel we need to be able to keep track of the memory usage. Therefore we generate empty pages while loading the kernel code, just to be able to check the memory usage.

```
839 <*showmemory>
840 \g_trace_statistics_status=2\scan_stop:
841 \cs_set_nopar:Npn\showMemUsage{
842   \if_horizontal_mode:
843     \tex_errmessage:D{Wrong~ mode~ H:~ something~ triggered~
844                       hmode~ above}
845   \else:
846     \tex_message:D{Mode ~ okay}
847   \fi:
848   \tex_shipout:D\hbox:w{ }
849 }
850 \showMemUsage
851 </showmemory>
```

## 96 l3basics implementation

We need l3names to get things going but we actually need it very early on, so it is loaded at the very top of the file l3basics.dtx. Also, most of the code below won't run until l3expan has been loaded.

### 96.1 Renaming some T<sub>E</sub>X primitives (again)

`\cs_set_eq:NwN` Having given all the tex primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.<sup>7</sup>

```
852 <*package>
853 \ProvidesExplPackage
854   {\filename}{\filedate}{\fileversion}{\filedescription}
855 \package_check_loaded_expl:
856 </package>
857 <*initex | package>
858 \tex_let:D \cs_set_eq:NwN          \tex_let:D
```

---

<sup>7</sup>This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the “tex...D” name in the cases where no good alternative exists.

`\if_true:` Then some conditionals.  
`\if_false:`  
`\or:` 859 `\cs_set_eq:NwN` `\if_true:` `\tex_iftrue:D`  
`\else:` 860 `\cs_set_eq:NwN` `\if_false:` `\tex_iffalse:D`  
`\fi:` 861 `\cs_set_eq:NwN` `\or:` `\tex_or:D`  
`\reverse_if:N` 862 `\cs_set_eq:NwN` `\else:` `\tex_else:D`  
`\if:w` 863 `\cs_set_eq:NwN` `\fi:` `\tex_fi:D`  
`\if_bool:N` 864 `\cs_set_eq:NwN` `\reverse_if:N` `\etex_unless:D`  
`\if_predicate:w` 865 `\cs_set_eq:NwN` `\if:w` `\tex_if:D`  
`\if_charcode:w` 866 `\cs_set_eq:NwN` `\if_bool:N` `\tex_ifodd:D`  
`\if_catcode:w` 867 `\cs_set_eq:NwN` `\if_predicate:w` `\tex_ifodd:D`  
868 `\cs_set_eq:NwN` `\if_charcode:w` `\tex_if:D`  
869 `\cs_set_eq:NwN` `\if_catcode:w` `\tex_ifcat:D`

`\if_meaning:w`

870 `\cs_set_eq:NwN` `\if_meaning:w` `\tex_ifx:D`

`\if_mode_math:` T<sub>E</sub>X lets us detect some if its modes.  
`\if_mode_horizontal:`  
`\if_mode_vertical:` 871 `\cs_set_eq:NwN` `\if_mode_math:` `\tex_ifmmode:D`  
`\if_mode_inner:` 872 `\cs_set_eq:NwN` `\if_mode_horizontal:` `\tex_ifhmode:D`  
873 `\cs_set_eq:NwN` `\if_mode_vertical:` `\tex_ifvmode:D`  
874 `\cs_set_eq:NwN` `\if_mode_inner:` `\tex_ifinner:D`

`\if_cs_exist:N`  
`\if_cs_exist:w`

875 `\cs_set_eq:NwN` `\if_cs_exist:N` `\etex_ifdefined:D`  
876 `\cs_set_eq:NwN` `\if_cs_exist:w` `\etex_ifcsname:D`

`\exp_after:wN` The three `\exp_` functions are used in the l3expan module where they are described.

`\exp_not:N`  
`\exp_not:n` 877 `\cs_set_eq:NwN` `\exp_after:wN` `\tex_expandafter:D`  
878 `\cs_set_eq:NwN` `\exp_not:N` `\tex_noexpand:D`  
879 `\cs_set_eq:NwN` `\exp_not:n` `\etex_unexpanded:D`

`\iow_shipout_x:Nn`  
`\token_to_meaning:N` 880 `\cs_set_eq:NwN` `\iow_shipout_x:Nn` `\tex_write:D`  
`\token_to_str:N` 881 `\cs_set_eq:NwN` `\token_to_meaning:N` `\tex_meaning:D`  
`\token_to_str:c` 882 `\cs_set_eq:NwN` `\token_to_str:N` `\tex_string:D`  
`\cs:w` 883 `\cs_set_eq:NwN` `\cs:w` `\tex_csname:D`  
`\cs_end:` 884 `\cs_set_eq:NwN` `\cs_end:` `\tex_endcsname:D`  
`\cs_meaning:N` 885 `\cs_set_eq:NwN` `\cs_meaning:N` `\tex_meaning:D`  
`\cs_meaning:c` 886 `\tex_def:D` `\cs_meaning:c` `{\exp_args:Nc\cs_meaning:N}`  
`\cs_show:N` 887 `\cs_set_eq:NwN` `\cs_show:N` `\tex_show:D`  
`\cs_show:c` 888 `\tex_def:D` `\cs_show:c` `{\exp_args:Nc\cs_show:N}`  
889 `\tex_def:D` `\token_to_str:c` `{\exp_args:Nc\token_to_str:N}`

`\scan_stop:` 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.

```

\group_begin:
\group_end:
890 \cs_set_eq:NwN \scan_stop: \tex_relax:D
891 \cs_set_eq:NwN \group_begin: \tex_begingroup:D
892 \cs_set_eq:NwN \group_end: \tex_endgroup:D

```

`\group_execute_after:N`

```
893 \cs_set_eq:NwN \group_execute_after:N \tex_aftergroup:D
```

`\pref_global:D`

`\pref_long:D`

`\pref_protected:D`

```

894 \cs_set_eq:NwN \pref_global:D \tex_global:D
895 \cs_set_eq:NwN \pref_long:D \tex_long:D
896 \cs_set_eq:NwN \pref_protected:D \etex_protected:D

```

## 96.2 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<sup>A</sup>T<sub>E</sub>X3 should be naturally robust; after all, the T<sub>E</sub>X primitives for assignments are and it can be a cause of problems if others aren't.

```

897 \cs_set_eq:NwN \cs_set_nopar:Npn \tex_def:D
898 \cs_set_eq:NwN \cs_set_nopar:Npx \tex_edef:D
899 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npn {
900 \pref_long:D \cs_set_nopar:Npn
901 }
902 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npx {
903 \pref_long:D \cs_set_nopar:Npx
904 }
905 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn {
906 \pref_protected:D \cs_set_nopar:Npn
907 }
908 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx {
909 \pref_protected:D \cs_set_nopar:Npx
910 }
911 \cs_set_protected_nopar:Npn \cs_set_protected:Npn {
912 \pref_protected:D \pref_long:D \cs_set_nopar:Npn
913 }
914 \cs_set_protected_nopar:Npn \cs_set_protected:Npx {
915 \pref_protected:D \pref_long:D \cs_set_nopar:Npx
916 }

```

`\cs_gset_nopar:Npn`

`\cs_gset_nopar:Npx`

`\cs_gset:Npn`

`\cs_gset:Npx`

`\cs_gset_protected_nopar:Npn`

`\cs_gset_protected_nopar:Npx`

`\cs_gset_protected:Npn`

`\cs_gset_protected:Npx`

Global versions of the above functions.

```
917 \cs_set_eq:NwN \cs_gset_nopar:Npn \tex_gdef:D
```

```

918 \cs_set_eq:NwN \cs_gset_nopar:Npx \tex_xdef:D
919 \cs_set_protected_nopar:Npn \cs_gset:Npn {
920   \pref_long:D \cs_gset_nopar:Npn
921 }
922 \cs_set_protected_nopar:Npn \cs_gset:Npx {
923   \pref_long:D \cs_gset_nopar:Npx
924 }
925 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn {
926   \pref_protected:D \cs_gset_nopar:Npn
927 }
928 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx {
929   \pref_protected:D \cs_gset_nopar:Npx
930 }
931 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn {
932   \pref_protected:D \pref_long:D \cs_gset_nopar:Npn
933 }
934 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx {
935   \pref_protected:D \pref_long:D \cs_gset_nopar:Npx
936 }

```

### 96.3 Selecting tokens

`\use:c` This macro grabs its argument and returns a csname from it.

```

937 \cs_set:Npn \use:c #1 { \cs:w#1\cs_end: }

```

`\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.

```

938 \cs_set_protected:Npn \use:x #1 {
939   \group_begin:
940   \cs_set:Npx \cs_tmp: {#1}
941   \exp_after:wN
942   \group_end:
943   \cs_tmp:
944 }

```

`\use:n` These macro grabs its arguments and returns it back to the input (with outer braces removed). `\use:n` is defined earlier for bootstrapping.

```

\use:nnn
\use:nnnn
945 \cs_set:Npn \use:n #1 {#1}
946 \cs_set:Npn \use:nn #1#2 {#1#2}
947 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
948 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}

```

`\use_i:nn` These macros are needed to provide functions with true and false cases, as introduced by Michael some time ago. By using `\exp_after:wN \use_i:nn \else:` constructions it is

possible to write code where the true or false case is able to access the following tokens from the input stream, which is not possible if the `\c_true_bool` syntax is used.

```
949 \cs_set:Npn \use_i:nn #1#2 {#1}
950 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

```
\use_i:nnnn
\use_ii:nnnn
\use_iii:nnnn
\use_i:nnnnn
\use_ii:nnnnn
\use_iii:nnnnn
\use_iv:nnnnn
\use_i_ii:nnnn
951 \cs_set:Npn \use_i:nnnn #1#2#3{#1}
952 \cs_set:Npn \use_ii:nnnn #1#2#3{#2}
953 \cs_set:Npn \use_iii:nnnn #1#2#3{#3}
954 \cs_set:Npn \use_i:nnnnn #1#2#3#4{#1}
955 \cs_set:Npn \use_ii:nnnnn #1#2#3#4{#2}
956 \cs_set:Npn \use_iii:nnnnn #1#2#3#4{#3}
957 \cs_set:Npn \use_iv:nnnnn #1#2#3#4{#4}
958 \cs_set:Npn \use_i_ii:nnnn #1#2#3{#1#2}
```

We also need something for picking up arguments from a longer list.

```
\use_none_delimit_by_q_nil:w
\use_none_delimit_by_q_stop:w
\use_none_delimit_by_q_recursion_stop:w
959 \cs_set:Npn \use_none_delimit_by_q_nil:w #1\q_nil{}
960 \cs_set:Npn \use_none_delimit_by_q_stop:w #1\q_stop{}
961 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop {}
```

Functions that gobble everything until they see either `\q_nil` or `\q_stop` resp.

```
\use_i_delimit_by_q_nil:nw
\use_i_delimit_by_q_stop:nw
\use_i_delimit_by_q_recursion_stop:nw
Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.
```

```
962 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2\q_nil{#1}
963 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2\q_stop{#1}
964 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

```
\use_i_after_fi:nw
\use_i_after_else:nw
\use_i_after_or:nw
\use_i_after_orelse:nw
Returns the first argument after ending the conditional.
965 \cs_set:Npn \use_i_after_fi:nw #1\fi:{\fi: #1}
966 \cs_set:Npn \use_i_after_else:nw #1\else:#2\fi:{\fi: #1}
967 \cs_set:Npn \use_i_after_or:nw #1\or: #2\fi: {\fi:#1}
968 \cs_set:Npn \use_i_after_orelse:nw #1 #2#3\fi: {\fi:#1}
```

## 96.4 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
\use_none:nnnnnnnnn
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.
969 \cs_set:Npn \use_none:n #1{}
970 \cs_set:Npn \use_none:nn #1#2{}
```

```

971 \cs_set:Npn \use_none:nnn #1#2#3{}
972 \cs_set:Npn \use_none:nnnn #1#2#3#4{}
973 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5{}
974 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6{}
975 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7{}
976 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8{}
977 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9{}

```

## 96.5 Expansion control from l3expan

`\exp_args:Nc` Moved here for now as it is going to be used right away.

```

978 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}

```

## 96.6 Conditional processing and definitions

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves `TEX` in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2 \prg_return_true: \else:
\if_meaning:w #1#3 \prg_return_true: \else:
\prg_return_false:
\fi: \fi:

```

Usually, a `TEX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `TEX` programmer to prove that he/she knows the  $2^n - 1$  table. We therefore provide the simpler interface.

`\prg_return_true:` These break statements put `TEX` in a *⟨true⟩* or *⟨false⟩* state. The idea is that the expansion of `\tex_romannumeral:D \c_zero` is *⟨null⟩* so we set off a `\tex_romannumeral:D`. It will on its way expand any `\else:` or `\fi:` that are waiting to be discarded anyway before finally arriving at the `\c_zero` we will place right after the conditional. After this expansion has terminated, we issue either `\if_true:` or `\if_false:` to put `TEX` in the correct state.

```

979 \cs_set:Npn \prg_return_true: { \exp_after:wN\if_true:\tex_romannumeral:D }
980 \cs_set:Npn \prg_return_false: {\exp_after:wN\if_false:\tex_romannumeral:D }

```

An extended state space could instead utilize `\tex_ifcase:D`:

```

\cs_set:Npn \prg_return_true: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_zero \tex_romannumeral:D
}

```

```

\cs_set:Npn \prg_return_false: {
  \exp_after:wN\text_ifcase:D \exp_after:wN \c_one \text_romannumeral:D
}
\cs_set:Npn \prg_return_error: {
  \exp_after:wN\text_ifcase:D \exp_after:wN \c_two \text_romannumeral:D
}

```

`\prg_set_conditional:Npnn`  
`\prg_new_conditional:Npnn`  
`\cs_set_protected_conditional:Npnn`  
`\cs_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.

```

981 \cs_set_protected:Npn \prg_set_conditional:Npnn #1{
982   \prg_get_parm_aux:nw{
983     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
984     \cs_set:Npn {parm}
985   }
986 }
987 \cs_set_protected:Npn \prg_new_conditional:Npnn #1{
988   \prg_get_parm_aux:nw{
989     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
990     \cs_new:Npn {parm}
991   }
992 }
993 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn #1{
994   \prg_get_parm_aux:nw{
995     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
996     \cs_set_protected:Npn {parm}
997   }
998 }
999 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn #1{
1000   \prg_get_parm_aux:nw{
1001     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1002     \cs_new_protected:Npn {parm}
1003   }
1004 }

```

`\prg_set_conditional:Nnn`  
`\prg_new_conditional:Nnn`  
`\cs_set_protected_conditional:Nnn`  
`\cs_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.

```

1005 \cs_set_protected:Npn \prg_set_conditional:Nnn #1{
1006   \exp_args:Nnf \prg_get_count_aux:nn{
1007     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1008     \cs_set:Npn {count}
1009   }{\cs_get_arg_count_from_signature:N #1}
1010 }
1011 \cs_set_protected:Npn \prg_new_conditional:Nnn #1{
1012   \exp_args:Nnf \prg_get_count_aux:nn{

```

```

1013     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1014     \cs_new:Npn {count}
1015   }{\cs_get_arg_count_from_signature:N #1}
1016 }
1017
1018 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn #1{
1019   \exp_args:Nnf \prg_get_count_aux:nn{
1020     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1021     \cs_set_protected:Npn {count}
1022   }{\cs_get_arg_count_from_signature:N #1}
1023 }
1024
1025 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn #1{
1026   \exp_args:Nnf \prg_get_count_aux:nn{
1027     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1028     \cs_new_protected:Npn {count}
1029   }{\cs_get_arg_count_from_signature:N #1}
1030 }

```

\prg\_set\_eq\_conditional:NNn  
\prg\_new\_eq\_conditional:NNn

The obvious setting-equal functions.

```

1031 \cs_set_protected:Npn \prg_set_eq_conditional:NNn #1#2#3 {
1032   \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3}
1033 }
1034 \cs_set_protected:Npn \prg_new_eq_conditional:NNn #1#2#3 {
1035   \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3}
1036 }

```

\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.

```

1037 \cs_set:Npn \prg_get_count_aux:nn #1#2 {#1{#2}}
1038 \cs_set:Npn \prg_get_parm_aux:nw #1#2#{#1{#2}}

```

e\_conditional\_parm\_aux:nnNNnnnn  
enerate\_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.

```

1039 \cs_set:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8{
1040   \prg_generate_conditional_aux:nnw{#5}{
1041     #4{#1}{#2}{#6}{#8}
1042   }#7,?, \q_recursion_stop
1043 }

```

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.

```

1044 \cs_set:Npn \prg_generate_conditional_aux:nnw #1#2#3,{
1045   \if:w ?#3
1046     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1047   \fi:
1048   \use:c{prg_generate_#3_form_#1:Nnnnn} #2
1049   \prg_generate_conditional_aux:nnw{#1}{#2}
1050 }

```

\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.

```

1051 \cs_set:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5{
1052   \exp_args:Nc #1 {#2_p:#3}#4{#5 \c_zero
1053   \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
1054 }
1055 }
1056 \cs_set:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5{
1057   \exp_args:Nc#1 {#2:#3TF}#4{#5 \c_zero
1058   \exp_after:wN \use_i:nn \else: \exp_after:wN \use_ii:nn \fi:
1059 }
1060 }
1061 \cs_set:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5{
1062   \exp_args:Nc#1 {#2:#3T}#4{#5 \c_zero
1063   \else:\exp_after:wN\use_none:nn\fi:\use:n
1064 }
1065 }
1066 \cs_set:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5{
1067   \exp_args:Nc#1 {#2:#3F}#4{#5 \c_zero
1068   \exp_after:wN\use_none:nn\fi:\use:n
1069 }
1070 }

```

\prg\_generate\_p\_form\_count:Nnnnn  
 \prg\_generate\_TF\_form\_count:Nnnnn  
 \prg\_generate\_T\_form\_count:Nnnnn  
 \prg\_generate\_F\_form\_count:Nnnnn

How to generate the various forms. The **count** types here use a number to insert the correct parameter text, otherwise like the **parm** functions above.

```

1071 \cs_set:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5{
1072   \cs_generate_from_arg_count:cNnn {#2_p:#3} #1 {#4}{#5 \c_zero
1073   \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
1074 }
1075 }
1076 \cs_set:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5{
1077   \cs_generate_from_arg_count:cNnn {#2:#3TF} #1 {#4}{#5 \c_zero
1078   \exp_after:wN\use_i:nn\else:\exp_after:wN\use_ii:nn\fi:
1079 }

```

```

1080 }
1081 \cs_set:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5{
1082   \cs_generate_from_arg_count:cNnn {#2:#3T} #1 {#4}{#5 \c_zero
1083     \else:\exp_after:wN\use_none:nn\fi:\use:n
1084 }
1085 }
1086 \cs_set:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5{
1087   \cs_generate_from_arg_count:cNnn {#2:#3F} #1 {#4}{#5 \c_zero
1088     \exp_after:wN\use_none:nn\fi:\use:n
1089 }
1090 }

prg_set_eq_conditional_aux:NNNn
prg_set_eq_conditional_aux:NNNw

1091 \cs_set:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4 {
1092   \prg_set_eq_conditional_aux:NNNw #1#2#3#4,?,\q_recursion_stop
1093 }

```

Manual clist loop over argument #4.

```

1094 \cs_set:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4, {
1095   \if:w ? #4 \scan_stop:
1096     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1097   \fi:
1098   #1 {
1099     \exp_args:NNc \cs_split_function:NN #2 {prg_conditional_form_#4:nnn}
1100   }{
1101     \exp_args:NNc \cs_split_function:NN #3 {prg_conditional_form_#4:nnn}
1102   }
1103   \prg_set_eq_conditional_aux:NNNw #1{#2}{#3}
1104 }

1105 \cs_set:Npn \prg_conditional_form_p:nnn #1#2#3 {#1_p:#2}
1106 \cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 {#1:#2TF}
1107 \cs_set:Npn \prg_conditional_form_T:nnn #1#2#3 {#1:#2T}
1108 \cs_set:Npn \prg_conditional_form_F:nnn #1#2#3 {#1:#2F}

```

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

1109 \tex_chardef:D \c_true_bool = 1~
1110 \tex_chardef:D \c_false_bool = 0~

```

## 96.7 Dissecting a control sequence

`\cs_to_str:N` This converts a control sequence into the character string of its name, removing the 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.

The route chosen is this: If `\token_to_str:N \a` produces a non-space escape char, then this will produce two tokens. If the escape char is non-printable, only one token is produced. If the escape char is a space, then a space token plus one token character token is produced. If we augment the result of this expansion with the letters `ax` we get the following three scenarios (with  $\langle X \rangle$  being a printable non-space escape character):

- $\langle X \rangle$ `aax`
- `aax`
- `aax`

In the second and third case, putting an auxiliary function in front reading undelimited arguments will treat them the same, removing the space token for us automatically. Therefore, if we test the second and third argument of what such a function reads, in case 1 we will get true and in cases 2 and 3 we will get false. If we choose to optimize for the usual case of a printable escape char, we can do it like this (again getting TeX to remove the leading space for us):

```

1111 \cs_set_nopar:Npn \cs_to_str:N {
1112   \if:w \exp_after:wN \cs_str_aux:w\token_to_str:N \a ax\q_stop
1113   \else:
1114     \exp_after:wN \exp_after:wN\exp_after:wN \use_ii:nn
1115     \fi:
1116     \exp_after:wN \use_none:n \token_to_str:N
1117   }
1118 \cs_set:Npn \cs_str_aux:w #1#2#3#4\q_stop{#2#3}

```

`\cs_split_function:NN` This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean  $\langle true \rangle$  or  $\langle false \rangle$  is returned with  $\langle true \rangle$  for when there is a colon in the function and  $\langle false \rangle$  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`.

```

1119 \group_begin:
1120   \tex_lccode:D '\@ = '\: \scan_stop:
1121   \tex_catcode:D '\@ = 12~
1122   \tex_lowercase:D {
1123   \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.

```

1124 \cs_set:Npn \cs_split_function:NN #1#2{
1125   \exp_after:wN \cs_split_function_aux:w
1126     \tex_romannumeral:D -'\q \cs_to_str:N #1 @a \q_stop #2
1127 }

```

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.

```

1128 \cs_set:Npn \cs_split_function_aux:w #1@#2#3\q_stop#4{
1129   \if_meaning:w a#2
1130     \exp_after:wN \use_i:nn
1131   \else:
1132     \exp_after:wN \use_ii:nn
1133   \fi:
1134   {#4{#1}{}}\c_false_bool}
1135   {\cs_split_function_auxii:w#2#3\q_stop #4{#1}}
1136 }
1137 \cs_set:Npn \cs_split_function_auxii:w #1@a\q_stop#2#3{
1138   #2{#3}{#1}\c_true_bool
1139 }

```

End of lowercase

```

1140 }

```

`\cs_get_function_name:N` Now returning the name is trivial: just discard the last two arguments. Similar for  
`\cs_get_function_signature:N` signature.

```

1141 \cs_set:Npn \cs_get_function_name:N #1 {
1142   \cs_split_function:NN #1\use_i:nnn
1143 }
1144 \cs_set:Npn \cs_get_function_signature:N #1 {
1145   \cs_split_function:NN #1\use_ii:nnn
1146 }

```

## 96.8 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\tex_relax:D` token. A control sequence is said to be *free* (to be defined) if it does not already exist and also meets the requirement that it does not contain a D signature. The reasoning behind this is that most of the time, a check for a free control sequence is when we wish to make a new control sequence and we do not want to let the user define a new “do not use” control sequence.

`\cs_if_exist_p:N` Two versions for checking existence. For the N form we firstly check for `\tex_relax:D`  
`\cs_if_exist_p:c` and then if it is in the hash table. There is no problem when inputting something like  
`\cs_if_exist:NTF` `\else:` or `\fi:` as T<sub>E</sub>X will only ever skip input in case the token tested against is  
`\cs_if_exist:cTF` `\tex_relax:D`.

```

1147 \prg_set_conditional:Npnn \cs_if_exist:N #1 {p,TF,T,F}{
1148   \if_meaning:w #1\tex_relax:D
1149   \prg_return_false:
1150   \else:
1151     \if_cs_exist:N #1
1152     \prg_return_true:
1153     \else:
1154       \prg_return_false:
1155     \fi:
1156   \fi:
1157 }
```

For the c form we firstly check if it is in the hash table and then for `\tex_relax:D` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1158 \prg_set_conditional:Npnn \cs_if_exist:c #1 {p,TF,T,F}{
1159   \if_cs_exist:w #1 \cs_end:
1160   \exp_after:wN \use_i:nn
1161   \else:
1162     \exp_after:wN \use_ii:nn
1163   \fi:
1164   {
1165     \exp_after:wN \if_meaning:w \cs:w #1\cs_end: \tex_relax:D
1166     \prg_return_false:
1167     \else:
1168       \prg_return_true:
1169     \fi:
1170   }
1171   \prg_return_false:
1172 }
```

```

\cs_if_do_not_use_p:N
\cs_if_do_not_use_aux:nnN
```

```

1173 \cs_set:Npn \cs_if_do_not_use_p:N #1{
1174   \cs_split_function:NN #1 \cs_if_do_not_use_aux:nnN
1175 }
1176 \cs_set:Npn \cs_if_do_not_use_aux:nnN #1#2#3{
1177   \str_if_eq_p:nn { D } {#2}
1178 }

```

`\cs_if_free_p:N` The simple implementation is one using the boolean expression parser: If it exists or  
`\cs_if_free_p:c` is do not use, then return false.

`\cs_if_free:N $\overline{TF}$`

```

\cs_if_free:c $\overline{TF}$  \prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
  \bool_if:nTF {\cs_if_exist_p:N #1 || \cs_if_do_not_use_p:N #1}
    {\prg_return_false:}\prg_return_true:}
}

```

However, this functionality may not be available this early on. We do something similar: The numerical values of true and false is one and zero respectively, which we can use. The problem again here is that the token we are checking may in fact be something that can disturb the scanner, so we have to be careful. We would like to do minimal evaluation so we ensure this.

```

1179 \prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
1180   \tex_ifnum:D \cs_if_exist_p:N #1 =\c_zero
1181   \exp_after:wN \use_i:nn
1182   \else:
1183     \exp_after:wN \use_ii:nn
1184   \fi:
1185   {
1186     \tex_ifnum:D \cs_if_do_not_use_p:N #1 =\c_zero
1187     \prg_return_true:
1188   }
1189   \else:
1190     \prg_return_false:
1191   \fi:
1192 }
1193 \prg_return_false:
1194 }
1195 \cs_set_nopar:Npn \cs_if_free_p:c{\exp_args:Nc\cs_if_free_p:N}
1196 \cs_set_nopar:Npn \cs_if_free:cTF{\exp_args:Nc\cs_if_free:N $\overline{TF}$ }
1197 \cs_set_nopar:Npn \cs_if_free:cT{\exp_args:Nc\cs_if_free:N $\overline{T}$ }
1198 \cs_set_nopar:Npn \cs_if_free:cF{\exp_args:Nc\cs_if_free:N $\overline{F}$ }

```

## 96.9 Defining and checking (new) functions

`\c_minus_one` We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the  
`\c_zero` log and the terminal and `\c_zero` which is used by some functions in the `l3alloc` module.  
`\c_sixteen` The rest are defined in the `l3int` module – at least for the ones that can be defined

with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can't be used until the allocation has been set up properly! The actual allocation mechanism is in `l3alloc` and as `TEX` wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.

```

1198 <*\initex>
1199 \cs_set_eq:NwN \c_minus_one\m@ne
1200 </!\initex>
1201 <*\package>
1202 \tex_countdef:D \c_minus_one = 10 ~
1203 \c_minus_one = -1 ~
1204 </!\package>
1205 \tex_chardef:D \c_sixteen = 16~
1206 \tex_chardef:D \c_zero = 0~

```

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.

```

1207 \cs_set_protected_nopar:Npn \iow_log:x {
1208   \tex_immediate:D \iow_shipout_x:Nn \c_minus_one
1209 }
1210 \cs_set_protected_nopar:Npn \iow_term:x {
1211   \tex_immediate:D \iow_shipout_x:Nn \c_sixteen
1212 }

```

`\msg_kernel_bug:x` This will show internal errors.

```

1213 \cs_set_protected_nopar:Npn \msg_kernel_bug:x #1 {
1214   \iow_term:x { This~is~a~LaTeX~bug:~check~coding! }
1215   \tex_errmessage:D {#1}
1216 }

```

`\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.

```

1217 <*\trace>
1218 \cs_set:Npn \cs_record_meaning:N #1{}
1219 </\trace>

```

`\chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if  $\langle csname \rangle$  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!

```

1220 \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1{
1221   \cs_if_free:NF #1
1222   {
1223     \msg_kernel_bug:x {Command~name~'\token_to_str:N #1'~already~defined!~
1224                       Current~meaning: \ \ \c_space_tl \c_space_tl \token_to_meaning:N #1
1225                       }
1226   }
1227   \*trace
1228   \cs_record_meaning:N#1
1229   % \iow_term:x{Defining~\token_to_str:N #1~on~%}
1230   \iow_log:x{Defining~\token_to_str:N #1~on~
1231             line~\tex_the:D \tex_inputlineno:D}
1232   \*trace
1233   }
1234   \cs_set_protected_nopar:Npn \chk_if_free_cs:c {
1235     \exp_args:Nc \chk_if_free_cs:N
1236   }
1237   \*package
1238   \tex_ifodd:D \@l@expl@log@functions@bool \else
1239     \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1 {
1240       \cs_if_free:NF #1
1241       {
1242         \msg_kernel_bug:x
1243         {
1244           Command~name~'\token_to_str:N #1'~already~defined!~
1245           Current~meaning: \ \ \c_space_tl \c_space_tl \token_to_meaning:N #1
1246         }
1247       }
1248     }
1249   \fi
1250   \*package

```

`\chk_if_exist_cs:N` This function issues a warning message when the control sequence in its argument does not exist.

```

1251 \cs_set_protected_nopar:Npn \chk_if_exist_cs:N #1 {
1252   \cs_if_exist:NF #1
1253   {
1254     \msg_kernel_bug:x {Command~ '\token_to_str:N #1'~
1255                       not~ yet~ defined!}
1256   }
1257   }
1258   \cs_set_protected_nopar:Npn \chk_if_exist_cs:c {
1259     \exp_args:Nc \chk_if_exist_cs:N

```

```
1260 }
```

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient.

`\str_if_eq:nnTF`

`\str_if_eq_p:xx`

`\str_if_eq:xxTF`

```
1261 \prg_set_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF } {
1262   \tex_ifnum:D \pdf_strcmp:D
1263   { \etex_unexpanded:D {#1} } { \etex_unexpanded:D {#2} }
1264   = \c_zero
1265   \prg_return_true: \else: \prg_return_false: \fi:
1266 }
1267 \prg_set_conditional:Npnn \str_if_eq:xx #1#2 { p , T , F , TF } {
1268   \tex_ifnum:D \pdf_strcmp:D {#1} {#2} = \c_zero
1269   \prg_return_true: \else: \prg_return_false: \fi:
1270 }
```

`\cs_if_eq_name_p:NN` An application of the above function, already streamlined for speed, so I put it in here.

```
1271 \prg_set_conditional:Npnn \cs_if_eq_name:NN #1#2{p}{
1272   \str_if_eq_p:nn {#1} {#2}
1273 }
```

## 96.10 More new definitions

`\cs_new_nopar:Npn` Global versions of the above functions.

`\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`

```
1274 \cs_set:Npn \cs_tmp:w #1#2 {
1275   \cs_set_protected_nopar:Npn #1 ##1
1276   {
1277     \chk_if_free_cs:N ##1
1278     #2 ##1
1279   }
1280 }
1281 \cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
1282 \cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1283 \cs_tmp:w \cs_new:Npn                \cs_gset:Npn
1284 \cs_tmp:w \cs_new:Npx                \cs_gset:Npx
1285 \cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1286 \cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1287 \cs_tmp:w \cs_new_protected:Npn      \cs_gset_protected:Npn
1288 \cs_tmp:w \cs_new_protected:Npx      \cs_gset_protected:Npx
```

`\cs_set_nopar:cpn` 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`

`\cs_set_nopar:cpn`

`\cs_gset_nopar:cpn`

`\cs_gset_nopar:cpn`

`\cs_new_nopar:cpn`

`\cs_new_nopar:cpn`

`\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.

```

1289 \cs_set:Npn \cs_tmp:w #1#2{
1290   \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 }
1291 }
1292 \cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1293 \cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1294 \cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1295 \cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1296 \cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1297 \cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

`\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
1298 \cs_tmp:w \cs_set:cpn \cs_set:Npn
1299 \cs_tmp:w \cs_set:cpx \cs_set:Npx
1300 \cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1301 \cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1302 \cs_tmp:w \cs_new:cpn \cs_new:Npn
1303 \cs_tmp:w \cs_new:cpx \cs_new:Npx

```

`\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`

```

1304 \cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1305 \cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1306 \cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1307 \cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1308 \cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1309 \cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

`\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
1310 \cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1311 \cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1312 \cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1313 \cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1314 \cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1315 \cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

## BACKWARDS COMPATIBILITY:

```

1316 \cs_set_eq:NwN \cs_gnew_nopar:Npn \cs_new_nopar:Npn
1317 \cs_set_eq:NwN \cs_gnew:Npn \cs_new:Npn
1318 \cs_set_eq:NwN \cs_gnew_protected_nopar:Npn \cs_new_protected_nopar:Npn

```

```

1319 \cs_set_eq:NwN      \cs_gnew_protected:Npn      \cs_new_protected:Npn
1320 \cs_set_eq:NwN      \cs_gnew_nopar:Npx          \cs_new_nopar:Npx
1321 \cs_set_eq:NwN      \cs_gnew:Npx              \cs_new:Npx
1322 \cs_set_eq:NwN \cs_gnew_protected_nopar:Npx \cs_new_protected_nopar:Npx
1323 \cs_set_eq:NwN      \cs_gnew_protected:Npx      \cs_new_protected:Npx
1324 \cs_set_eq:NwN      \cs_gnew_nopar:cpn          \cs_new_nopar:cpn
1325 \cs_set_eq:NwN      \cs_gnew:cpn                \cs_new:cpn
1326 \cs_set_eq:NwN \cs_gnew_protected_nopar:cpn \cs_new_protected_nopar:cpn
1327 \cs_set_eq:NwN      \cs_gnew_protected:cpn      \cs_new_protected:cpn
1328 \cs_set_eq:NwN      \cs_gnew_nopar:cpx          \cs_new_nopar:cpx
1329 \cs_set_eq:NwN      \cs_gnew:cpx                \cs_new:cpx
1330 \cs_set_eq:NwN \cs_gnew_protected_nopar:cpx \cs_new_protected_nopar:cpx
1331 \cs_set_eq:NwN      \cs_gnew_protected:cpx      \cs_new_protected:cpx

```

\use\_0\_parameter: For using parameters, i.e., when you need to define a function to process three parameters.

\use\_1\_parameter: See `xparse` for an application.

\use\_2\_parameter:

\use\_3\_parameter:

\use\_4\_parameter:

\use\_5\_parameter:

\use\_6\_parameter:

\use\_7\_parameter:

\use\_8\_parameter:

\use\_9\_parameter:

```

1332 \cs_set_nopar:cpn{use_0_parameter:}{ }
1333 \cs_set_nopar:cpn{use_1_parameter:}{\##1}
1334 \cs_set_nopar:cpn{use_2_parameter:}{\##1}{\##2}
1335 \cs_set_nopar:cpn{use_3_parameter:}{\##1}{\##2}{\##3}
1336 \cs_set_nopar:cpn{use_4_parameter:}{\##1}{\##2}{\##3}{\##4}
1337 \cs_set_nopar:cpn{use_5_parameter:}{\##1}{\##2}{\##3}{\##4}{\##5}
1338 \cs_set_nopar:cpn{use_6_parameter:}{\##1}{\##2}{\##3}{\##4}{\##5}{\##6}
1339 \cs_set_nopar:cpn{use_7_parameter:}{\##1}{\##2}{\##3}{\##4}{\##5}{\##6}{\##7}
1340 \cs_set_nopar:cpn{use_8_parameter:}{
1341   {\##1}{\##2}{\##3}{\##4}{\##5}{\##6}{\##7}{\##8}
1342 \cs_set_nopar:cpn{use_9_parameter:}{
1343   {\##1}{\##2}{\##3}{\##4}{\##5}{\##6}{\##7}{\##8}{\##9}

```

## 96.11 Copying definitions

\cs\_set\_eq:NN These macros allow us to copy the definition of a control sequence to another control sequence.

\cs\_set\_eq:cN

\cs\_set\_eq:Nc

\cs\_set\_eq:cc

The = sign allows us to define funny char tokens like = itself or `_` with this function. For the definition of `\c_space_chartok{~}` to work we need the `~` after the =.

\cs\_set\_eq:NN is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an ‘already defined’ error rather than ‘runaway argument’.

The c variants are not protected in order for their arguments to be constructed in the correct context.

```

1344 \cs_set_protected:Npn \cs_set_eq:NN #1 { \cs_set_eq:NwN #1=~ }
1345 \cs_set_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1346 \cs_set_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:Nnc \cs_set_eq:NN }
1347 \cs_set_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }

```

```

\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc
1348 \cs_new_protected:Npn \cs_new_eq:NN #1 {
1349   \chk_if_free_cs:N #1
1350   \pref_global:D \cs_set_eq:NN #1
1351 }
1352 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1353 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:Nnc \cs_new_eq:NN }
1354 \cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

```

\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc
1355 \cs_new_protected:Npn \cs_gset_eq:NN { \pref_global:D \cs_set_eq:NN }
1356 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:Nnc \cs_gset_eq:NN }
1357 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1358 \cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }

```

## BACKWARDS COMPATIBILITY

```

1359 \cs_set_eq:NN \cs_gnew_eq:NN \cs_new_eq:NN
1360 \cs_set_eq:NN \cs_gnew_eq:cN \cs_new_eq:cN
1361 \cs_set_eq:NN \cs_gnew_eq:Nc \cs_new_eq:Nc
1362 \cs_set_eq:NN \cs_gnew_eq:cc \cs_new_eq:cc

```

## 96.12 Undefined functions

The following function is used to free the main memory from the definition of some function that isn't in use any longer.

```

\cs_undefine:N
\cs_undefine:c
\cs_gundefine:N
\cs_gundefine:c
1363 \cs_new_protected_nopar:Npn \cs_undefine:N #1 {
1364   \cs_set_eq:NN #1 \c_undefined:D
1365 }
1366 \cs_new_protected_nopar:Npn \cs_undefine:c #1 {
1367   \cs_set_eq:cN {#1} \c_undefined:D
1368 }
1369 \cs_new_protected_nopar:Npn \cs_gundefine:N #1 {
1370   \cs_gset_eq:NN #1 \c_undefined:D
1371 }
1372 \cs_new_protected_nopar:Npn \cs_gundefine:c #1 {
1373   \cs_gset_eq:cN {#1} \c_undefined:D
1374 }

```

## 96.13 Diagnostic wrapper functions

```

\kernel_register_show:N
\kernel_register_show:c
1375 \cs_new_nopar:Npn \kernel_register_show:N #1 {

```

```

1376 \cs_if_exist:NTF #1
1377 {
1378   \tex_showthe:D #1
1379 }
1380 {
1381   \msg_kernel_bug:x {Register~ '\token_to_str:N #1'~ is~ not~ defined.}
1382 }
1383 }
1384 \cs_new_nopar:Npn \kernel_register_show:c { \exp_args:Nc \int_show:N }

```

## 96.14 Engine specific definitions

`\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.

```

\c_luatex_is_engine_bool
\etex_if_engine:TF
\luatex_if_engine:TF
1385 \cs_if_exist:NTF \xetex_version:D
1386 { \cs_new_eq:NN \c_xetex_is_engine_bool \c_true_bool }
1387 { \cs_new_eq:NN \c_xetex_is_engine_bool \c_false_bool }
1388 \prg_new_conditional:Npnn \xetex_if_engine: {TF,T,F} {
1389   \if_bool:N \c_xetex_is_engine_bool
1390     \prg_return_true: \else: \prg_return_false: \fi:
1391 }

1392 \cs_if_exist:NTF \luatex_directlua:D
1393 { \cs_new_eq:NN \c_luatex_is_engine_bool \c_true_bool }
1394 { \cs_new_eq:NN \c_luatex_is_engine_bool \c_false_bool }
1395 \prg_set_conditional:Npnn \xetex_if_engine: {TF,T,F}{
1396   \if_bool:N \c_xetex_is_engine_bool \prg_return_true:
1397   \else: \prg_return_false: \fi:
1398 }
1399 \prg_set_conditional:Npnn \luatex_if_engine: {TF,T,F}{
1400   \if_bool:N \c_luatex_is_engine_bool \prg_return_true:
1401   \else: \prg_return_false: \fi:
1402 }

```

## 96.15 Scratch functions

`\prg_do_nothing:` I don't think this function belongs here, but one place is as good as any other. I want to use this function when I want to express 'no operation'. It is for example used in templates where depending on the users settings we have to either select an function that does something, or one that does nothing.

```

1403 \cs_new_nopar:Npn \prg_do_nothing: {}

```

## 96.16 Defining functions from a given number of arguments

`\get_arg_count_from_signature:N` 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.

```

1404 \cs_set:Npn \cs_get_arg_count_from_signature:N #1{
1405   \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN
1406 }
1407 \cs_set:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3{
1408   \if_predicate:w #3 % \bool_if:NTF here
1409     \exp_after:wN \use_i:nn
1410   \else:
1411     \exp_after:wN\use_ii:nn
1412   \fi:
1413   {
1414     \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
1415     \use_none:nnnnnnnn #2 9876543210\q_stop
1416   }
1417   {-1}
1418 }
1419 \cs_set:Npn \cs_get_arg_count_from_signature_auxii:w #1#2\q_stop{#1}

```

A variant form we need right away.

```

1420 \cs_set_nopar:Npn \cs_get_arg_count_from_signature:c {
1421   \exp_args:Nc \cs_get_arg_count_from_signature:N
1422 }

```

`\cs_generate_from_arg_count:NNnn`  
`ate_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  $\text{\TeX}$  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

```

1423 \cs_set:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4{
1424   \tex_ifcase:D \etex_numexpr:D #3\text_relax:D
1425     \use_i_after_orelse:nw{#2#1}
1426   \or:
1427     \use_i_after_orelse:nw{#2#1 ##1}
1428   \or:
1429     \use_i_after_orelse:nw{#2#1 ##1##2}
1430   \or:
1431     \use_i_after_orelse:nw{#2#1 ##1##2##3}
1432   \or:
1433     \use_i_after_orelse:nw{#2#1 ##1##2##3##4}
1434   \or:

```

```

1435     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5}
1436 \or:
1437     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6}
1438 \or:
1439     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7}
1440 \or:
1441     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8}
1442 \or:
1443     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8##9}
1444 \else:
1445     \use_i_after_fi:nw{
1446         \cs_generate_from_arg_count_error_msg:Nn#1{#3}
1447         \use_none:n % to remove replacement text
1448     }
1449 \fi:
1450 {#4}
1451 }

```

A variant form we need right away.

```

1452 \cs_set_nopar:Npn \cs_generate_from_arg_count:cNnn {
1453     \exp_args:Nc \cs_generate_from_arg_count:NNnn
1454 }

```

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.

```

1455 \cs_set:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2 {
1456     \msg_kernel_bug:x {
1457         You're~ trying~ to~ define~ the~ command~ '\token_to_str:N #1'~
1458         with~ \use:n{\tex_the:D\etex_numexpr:D #2\tex_relax:D} ~
1459         arguments~ but~ I~ only~ allow~ 0-9~arguments.~Perhaps~you~
1460         forgot~to~use~a~colon~in~the~function~name?~
1461         I~ can~ probably~ not~ help~ you~ here
1462     }
1463 }

```

## 96.17 Using the signature to define functions

We can now combine some of the tools we have to provide a simple interface for defining functions. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, i.e., the number of arguments is read from the signature.

```

\cs_set:Nn We want to define \cs_set:Nn as
\cs_set:Nx
\cs_set_nopar:Nn \cs_set_protected:Npn \cs_set:Nn #1#2{
\cs_set_nopar:Nx \cs_generate_from_arg_count:NNnn #1\cs_set:Npn
\cs_set_protected:Nn {\cs_get_arg_count_from_signature:N #1}{#2}
\cs_set_protected:Nx }
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

1464 \cs_set:Npn \cs_tmp:w #1#2#3{
1465   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1466     \exp_not:N \cs_generate_from_arg_count:NNnn ##1
1467     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1468     {\exp_not:N\cs_get_arg_count_from_signature:N ##1}{##2}
1469   }
1470 }
```

Then we define the 32 variants beginning with N.

```

1471 \cs_tmp:w {set}{Nn}{Npn}
1472 \cs_tmp:w {set}{Nx}{Npx}
1473 \cs_tmp:w {set_nopar}{Nn}{Npn}
1474 \cs_tmp:w {set_nopar}{Nx}{Npx}
1475 \cs_tmp:w {set_protected}{Nn}{Npn}
1476 \cs_tmp:w {set_protected}{Nx}{Npx}
1477 \cs_tmp:w {set_protected_nopar}{Nn}{Npn}
1478 \cs_tmp:w {set_protected_nopar}{Nx}{Npx}
1479 \cs_tmp:w {gset}{Nn}{Npn}
1480 \cs_tmp:w {gset}{Nx}{Npx}
1481 \cs_tmp:w {gset_nopar}{Nn}{Npn}
1482 \cs_tmp:w {gset_nopar}{Nx}{Npx}
1483 \cs_tmp:w {gset_protected}{Nn}{Npn}
1484 \cs_tmp:w {gset_protected}{Nx}{Npx}
1485 \cs_tmp:w {gset_protected_nopar}{Nn}{Npn}
1486 \cs_tmp:w {gset_protected_nopar}{Nx}{Npx}
```

```

\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
1487 \cs_tmp:w {new}{Nn}{Npn}
1488 \cs_tmp:w {new}{Nx}{Npx}
1489 \cs_tmp:w {new_nopar}{Nn}{Npn}
1490 \cs_tmp:w {new_nopar}{Nx}{Npx}
1491 \cs_tmp:w {new_protected}{Nn}{Npn}
1492 \cs_tmp:w {new_protected}{Nx}{Npx}
1493 \cs_tmp:w {new_protected_nopar}{Nn}{Npn}
1494 \cs_tmp:w {new_protected_nopar}{Nx}{Npx}
```

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}
}
```

```

1495 \cs_set:Npn \cs_tmp:w #1#2#3{
1496   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1497     \exp_not:N\cs_generate_from_arg_count:cNnn {##1}
1498     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1499     {\exp_not:N\cs_get_arg_count_from_signature:c {##1}}{##2}
1500   }
1501 }

```

The 32 c variants.

\cs_set:cn	
\cs_set:cx	
\cs_set_nopar:cn	1502 \cs_tmp:w {set}{cn}{Npn}
\cs_set_nopar:cx	1503 \cs_tmp:w {set}{cx}{Npx}
\cs_set_protected:cn	1504 \cs_tmp:w {set_nopar}{cn}{Npn}
\cs_set_protected:cx	1505 \cs_tmp:w {set_nopar}{cx}{Npx}
\cs_set_protected_nopar:cn	1506 \cs_tmp:w {set_protected}{cn}{Npn}
\cs_set_protected_nopar:cx	1507 \cs_tmp:w {set_protected}{cx}{Npx}
\cs_gset:cn	1508 \cs_tmp:w {set_protected_nopar}{cn}{Npn}
\cs_gset:cx	1509 \cs_tmp:w {set_protected_nopar}{cx}{Npx}
\cs_gset_nopar:cn	1510 \cs_tmp:w {gset}{cn}{Npn}
\cs_gset_nopar:cx	1511 \cs_tmp:w {gset}{cx}{Npx}
\cs_gset_protected:cn	1512 \cs_tmp:w {gset_nopar}{cn}{Npn}
\cs_gset_protected:cx	1513 \cs_tmp:w {gset_nopar}{cx}{Npx}
\cs_gset_protected_nopar:cn	1514 \cs_tmp:w {gset_protected}{cn}{Npn}
\cs_gset_protected_nopar:cx	1515 \cs_tmp:w {gset_protected}{cx}{Npx}
	1516 \cs_tmp:w {gset_protected_nopar}{cn}{Npn}
	1517 \cs_tmp:w {gset_protected_nopar}{cx}{Npx}

  

\cs_new:cn	
\cs_new:cx	
\cs_new_nopar:cn	1518 \cs_tmp:w {new}{cn}{Npn}
\cs_new_nopar:cx	1519 \cs_tmp:w {new}{cx}{Npx}
\cs_new_protected:cn	1520 \cs_tmp:w {new_nopar}{cn}{Npn}
\cs_new_protected:cx	1521 \cs_tmp:w {new_nopar}{cx}{Npx}
\cs_new_protected_nopar:cn	1522 \cs_tmp:w {new_protected}{cn}{Npn}
\cs_new_protected_nopar:cx	1523 \cs_tmp:w {new_protected}{cx}{Npx}
	1524 \cs_tmp:w {new_protected_nopar}{cn}{Npn}
	1525 \cs_tmp:w {new_protected_nopar}{cx}{Npx}

Check if two control sequences are identical.

\cs_if_eq_p:NN	
\cs_if_eq_p:cN	
\cs_if_eq_p:Nc	1526 \prg_set_conditional:Npnn \cs_if_eq:NN #1#2{p,TF,T,F}{
\cs_if_eq_p:cc	1527   \if_meaning:w #1#2
\cs_if_eq:NNTF	1528   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF	1529 }
\cs_if_eq:NcTF	1530 \cs_new_nopar:Npn \cs_if_eq_p:cN {\exp_args:Nc \cs_if_eq_p:NN}
\cs_if_eq:ccTF	1531 \cs_new_nopar:Npn \cs_if_eq_p:cNTF {\exp_args:Nc \cs_if_eq:NNTF}
	1532 \cs_new_nopar:Npn \cs_if_eq:cNT {\exp_args:Nc \cs_if_eq:NNT}
	1533 \cs_new_nopar:Npn \cs_if_eq:cNF {\exp_args:Nc \cs_if_eq:NNF}
	1534 \cs_new_nopar:Npn \cs_if_eq_p:Nc {\exp_args:NNc \cs_if_eq_p:NN}

```

1535 \cs_new_nopar:Npn \cs_if_eq:NcTF {\exp_args:NNc \cs_if_eq:NNTF}
1536 \cs_new_nopar:Npn \cs_if_eq:NcT  {\exp_args:NNc \cs_if_eq:NNT}
1537 \cs_new_nopar:Npn \cs_if_eq:NcF  {\exp_args:NNc \cs_if_eq:NNF}
1538 \cs_new_nopar:Npn \cs_if_eq_p:cc {\exp_args:Ncc \cs_if_eq_p:NN}
1539 \cs_new_nopar:Npn \cs_if_eq:ccTF {\exp_args:Ncc \cs_if_eq:NNTF}
1540 \cs_new_nopar:Npn \cs_if_eq:ccT  {\exp_args:Ncc \cs_if_eq:NNT}
1541 \cs_new_nopar:Npn \cs_if_eq:ccF  {\exp_args:Ncc \cs_if_eq:NNF}

1542 </initex | package>

1543 <*showmemory>
1544 \showMemUsage
1545 </showmemory>

```

## 97 l3expan implementation

### 97.1 Internal functions and variables

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

This will expand `<token2>` once before processing `<token1>`. This is similar to `\exp_args:No` except that no braces are put around the result of expanding `<token2>`.

**TeXhackers note:** This is the primitive `\expandafter` which was renamed to fit into the naming conventions of L<sup>A</sup>T<sub>E</sub>X3.

`\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 <register>`

These functions evaluates a register as part of a `V` or `v` expansion (respectively). A register might exist as one of two things: A parameter-less non-long, non-protected macro or a built-in T<sub>E</sub>X register such as `\count`.

`\exp_eval_error_msg:w` `\exp_eval_error_msg:w <register>`

Used to generate an error message if a variable called as part of a `v` or `V` expansion

is defined as `\scan_stop:`. This typically indicates that an incorrect cs name has been used.

<code>\n:</code>
<code>\N:</code>
<code>\c:</code>
<code>\o:</code>
<code>\f:</code>
<code>\x:</code>
<code>\v:</code>
<code>\V:</code>
<code>\::</code>

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

Internal forms for the base expansion types.

## 97.2 Module code

We start by ensuring that the required packages are loaded.

```

1546 <*package>
1547 \ProvidesExplPackage
1548   {\filename}{\filedate}{\fileversion}{\filedescription}
1549 \package_check_loaded_expl:
1550 </package>
1551 <*initex | package>

\exp_after:wN These are defined in l3basics.
\exp_not:N
\exp_not:n
1552 <*bootstrap>
1553 \cs_set_eq:NwN \exp_after:wN \tex_expandafter:D
1554 \cs_set_eq:NwN \exp_not:N \tex_noexpand:D
1555 \cs_set_eq:NwN \exp_not:n \etex_unexpanded:D
1556 </bootstrap>

```

## 97.3 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L<sup>A</sup>T<sub>E</sub>X3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.<sup>8</sup>)

The definition of expansion functions with this technique happens in section 97.5. In section 97.4 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

---

<sup>8</sup>However, 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.

`\l_exp_tl` We need a scratch token list variable. We don't use `tl` methods so that `l3expan` can be loaded earlier.

```
1557 \cs_new_nopar:Npn \l_exp_tl {}
```

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 auxilliary 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.

```
1558 \cs_new:Npn\exp_arg_next:nnn#1#2#3{
1559   #2\:::{#3{#1}}
1560 }
1561 \cs_new:Npn\exp_arg_next_nobrace:nnn#1#2#3{
1562   #2\:::{#3#1}
1563 }
```

`\:::` The end marker is just another name for the identity function.

```
1564 \cs_new:Npn\:::#1{#1}
```

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
1565 \cs_new:Npn\::n#1\:::#2#3{
1566   #1\:::{#2{#3}}
1567 }
```

`\::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
1568 \cs_new:Npn\::N#1\:::#2#3{
1569   #1\:::{#2#3}
1570 }
```

`\::c` This function is used to skip an argument that is turned into as control sequence without expansion.

```
1571 \cs_new:Npn\::c#1\:::#2#3{
1572   \exp_after:wN\exp_arg_next_nobrace:nnn\cs:w #3\cs_end:{#1}{#2}
1573 }
```

`\::o` This function is used to expand an argument once.

```

1574 \cs_new:Npn \::o#1\:::#2#3{
1575   \exp_after:wN\exp_arg_next:nnn\exp_after:wN{#3}{#1}{#2}
1576 }

```

`\::f` This function is used to expand a token list until the first unexpandable token is found.  
`\exp_stop_f:` The underlying `\tex_romannumeral:D -'0` expands everything in its way to find something terminating the number and thereby expands the function in front of it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker; in case the scanner hits a number, this number also terminates the scanning and is left untouched. In the example shown earlier the scanning was stopped once  $\TeX$  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 *null*, we wind up with a fully expanded list, only  $\TeX$  has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

1577 \cs_new:Npn \::f#1\:::#2#3{
1578   \exp_after:wN\exp_arg_next:nnn
1579   \exp_after:wN{\tex_romannumeral:D -'0 #3}
1580   {#1}{#2}
1581 }
1582 \cs_new_nopar:Npn \exp_stop_f: {-}

```

`\::x` This function is used to expand an argument fully. We could use the new expandable primitive `\expanded` here, but we don't want to create incompatibilities between engines.

```

1583 \cs_new_protected:Npn \::x #1 \::: #2#3 {
1584   \cs_set_nopar:Npx \l_exp_tl {{#3}}
1585   \exp_after:wN \exp_arg_next:nnn \l_exp_tl {#1}{#2}
1586 }

```

`\::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 `csname` from its argument given in braces and then evaluates it as if it was a `V`. The sequence `\tex_romannumeral:D -'0` sets off an `f` type expansion. The argument is returned in braces.

```

1587 \cs_new:Npn \::V#1\:::#2#3{
1588   \exp_after:wN\exp_arg_next:nnn
1589   \exp_after:wN{
1590     \tex_romannumeral:D -'0
1591     \exp_eval_register:N #3
1592   }
1593   {#1}{#2}
1594 }

```

```

1595 \cs_new:Npn \::v#1\:::#2#3{
1596   \exp_after:wN\exp_arg_next:nnn
1597   \exp_after:wN{
1598     \tex_romannumeral:D -‘0
1599     \exp_eval_register:c {#3}
1600   }
1601   {#1}{#2}
1602 }

```

`\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<sub>E</sub>X register such as `\count`. For the T<sub>E</sub>X 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 `\tex_relax:D`. `\exp_eval_error_msg:w`

```

1603 \cs_set_nopar:Npn \exp_eval_register:N #1{
1604   \exp_after:wN \if_meaning:w \exp_not:N #1#1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\tex_relax:D`. In that case we throw an error. We could let T<sub>E</sub>X 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.

```

1605   \if_meaning:w \tex_relax:D #1
1606   \exp_eval_error_msg:w
1607   \fi:

```

The next bit requires some explanation. The function must be initiated by the sequence `\tex_romannumeral:D -‘0` and we want to terminate this expansion chain by inserting an `\exp_stop_f:` token. However, we have to expand the register `#1` before we do that. If it is a T<sub>E</sub>X register, we need to execute the sequence `\exp_after:wN\exp_stop_f:\tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN\exp_stop_f: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

1608   \else:
1609     \exp_after:wN \use_i_ii:nnn
1610   \fi:
1611   \exp_after:wN \exp_stop_f: \tex_the:D #1
1612 }
1613 \cs_set_nopar:Npn \exp_eval_register:c #1{
1614   \exp_after:wN\exp_eval_register:N\cs:w #1\cs_end:
1615 }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
\exp_eval_error_msg:w ...erroneous variable used!

1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

1616 \group_begin:%
1617 \tex_catcode:D'\!=11\tex_relax:D%
1618 \tex_catcode:D'\ =11\tex_relax:D%
1619 \cs_gset:Npn\exp_eval_error_msg:w#1\tex_the:D#2{%
1620 \fi:\fi:\erroneous variable used!}%
1621 \group_end:%

```

## 97.4 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\pref_global:D` for example. This together with the fact that the ‘general’ concept above is slower means that we should convert whenever possible and perhaps remove all remaining occurrences by hand-encoding in the end.

```

\exp_args:No
\exp_args:NNo
\exp_args:NNNo
1622 \cs_new:Npn \exp_args:No #1#2{\exp_after:wN#1\exp_after:wN{#2}}
1623 \cs_new:Npn \exp_args:NNo #1#2#3{\exp_after:wN#1\exp_after:wN#2
1624 \exp_after:wN{#3}}
1625 \cs_new:Npn \exp_args:NNNo #1#2#3#4{\exp_after:wN#1\exp_after:wN#2
1626 \exp_after:wN#3\exp_after:wN{#4}}

```

```

\exp_args:Nc Here are the functions that turn their argument into csnames but are expandable.
\exp_args:cc
\exp_args:NNc
\exp_args:Ncc
\exp_args:Nccc
1627 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}
1628 \cs_new:Npn \exp_args:cc #1#2{\cs:w #1\exp_after:wN\cs_end:\cs:w #2\cs_end:}
1629 \cs_new:Npn \exp_args:NNc #1#2#3{\exp_after:wN#1\exp_after:wN#2
1630 \cs:w#3\cs_end:}
1631 \cs_new:Npn \exp_args:Ncc #1#2#3{\exp_after:wN#1
1632 \cs:w#2\exp_after:wN\cs_end:\cs:w#3\cs_end:}
1633 \cs_new:Npn \exp_args:Nccc #1#2#3#4{\exp_after:wN#1
1634 \cs:w#2\exp_after:wN\cs_end:\cs:w#3\exp_after:wN
1635 \cs_end:\cs:w #4\cs_end:}

```

`\exp_args:Nco` If we force that the third argument always has braces, we could implement this function with less tokens and only two arguments.

```

1636 \cs_new:Npn \exp_args:Nco #1#2#3{\exp_after:wN#1\cs:w#2\exp_after:wN
1637 \cs_end:\exp_after:wN{#3}}

```

### 97.5 Definitions with the ‘general’ technique

```

\exp_args:Nf
\exp_args:NV
\exp_args:Nv      1638 \cs_set_nopar:Npn \exp_args:Nf {\::f\:::}
\exp_args:Nx      1639 \cs_set_nopar:Npn \exp_args:Nv {\::v\:::}
                   1640 \cs_set_nopar:Npn \exp_args:NV {\::V\:::}
                   1641 \cs_set_protected_nopar:Npn \exp_args:Nx {\::x\:::}

```

`\exp_args:NNV` Here are the actual function definitions, using the helper functions above.

```

\exp_args:NNv
\exp_args:NNf 1642 \cs_set_nopar:Npn \exp_args:NNf {\:N\::f\:::}
\exp_args:NNx 1643 \cs_set_nopar:Npn \exp_args:NNv {\:N\::v\:::}
\exp_args:NVV 1644 \cs_set_nopar:Npn \exp_args:NNV {\:N\::V\:::}
\exp_args:Ncx 1645 \cs_set_protected_nopar:Npn \exp_args:NNx {\:N\::x\:::}
1646
\exp_args:Nfo 1647 \cs_set_protected_nopar:Npn \exp_args:Ncx {\:c\::x\:::}
\exp_args:Nff 1648 \cs_set_nopar:Npn \exp_args:Nfo {\:f\::o\:::}
\exp_args:Ncf 1649 \cs_set_nopar:Npn \exp_args:Nff {\:f\::f\:::}
\exp_args:Nco 1650 \cs_set_nopar:Npn \exp_args:Ncf {\:c\::f\:::}
\exp_args:Nnf 1651 \cs_set_nopar:Npn \exp_args:Nnf {\:n\::f\:::}
\exp_args:Nno 1652 \cs_set_nopar:Npn \exp_args:Nno {\:n\::o\:::}
\exp_args:NnV 1653 \cs_set_nopar:Npn \exp_args:NnV {\:n\::V\:::}
\exp_args:Nnx 1654 \cs_set_protected_nopar:Npn \exp_args:Nnx {\:n\::x\:::}
1655
\exp_args:Noo 1656 \cs_set_nopar:Npn \exp_args:Noc {\:o\::c\:::}
\exp_args:Noc 1657 \cs_set_nopar:Npn \exp_args:Noo {\:o\::o\:::}
\exp_args:Nox 1658 \cs_set_protected_nopar:Npn \exp_args:Nox {\:o\::x\:::}
\exp_args:Nxo 1659
\exp_args:Nxx 1660 \cs_set_nopar:Npn \exp_args:NVV {\:V\::V\:::}
1661
1662 \cs_set_protected_nopar:Npn \exp_args:Nxo {\:x\::o\:::}
1663 \cs_set_protected_nopar:Npn \exp_args:Nxx {\:x\::x\:::}

```

```

\exp_args:Ncco
\exp_args:Nccx
\exp_args:Ncnx      1664 \cs_set_nopar:Npn \exp_args:NNNV {\::N\::N\::V\:::}
\exp_args:NcNc      1665
\exp_args:NcNo      1666 \cs_set_nopar:Npn \exp_args:NNno {\::N\::n\::o\:::}
\exp_args:NNno      1667 \cs_set_protected_nopar:Npn \exp_args:NNnx {\::N\::n\::x\:::}
\exp_args:NNoo      1668 \cs_set_nopar:Npn \exp_args:NNoo {\::N\::o\::o\:::}
\exp_args:NNNV      1669 \cs_set_protected_nopar:Npn \exp_args:NNox {\::N\::o\::x\:::}
\exp_args:Nnno      1670
\exp_args:Nnnx      1671 \cs_set_nopar:Npn \exp_args:Nnnc {\::n\::n\::c\:::}
\exp_args:Nnox      1672 \cs_set_nopar:Npn \exp_args:Nnno {\::n\::n\::o\:::}
\exp_args:Nooo      1673 \cs_set_protected_nopar:Npn \exp_args:Nnnx {\::n\::n\::x\:::}
\exp_args:Noox      1674 \cs_set_protected_nopar:Npn \exp_args:Nnox {\::n\::o\::x\:::}
\exp_args:Nnnc      1675
\exp_args:NNnx      1676 \cs_set_nopar:Npn \exp_args:NcNc {\::c\::N\::c\:::}
\exp_args:NNoo
\exp_args:NNox

```

```

1677 \cs_set_nopar:Npn \exp_args:NcNo {\::c\::N\::o\::}
1678 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\::}
1679 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\::}
1680 \cs_set_protected_nopar:Npn \exp_args:Nccx {\::c\::c\::x\::}
1681 \cs_set_protected_nopar:Npn \exp_args:Ncnx {\::c\::n\::x\::}
1682
1683 \cs_set_protected_nopar:Npn \exp_args:Noox {\::o\::o\::x\::}
1684 \cs_set_nopar:Npn \exp_args:Nooo {\::o\::o\::o\::}

```

## 97.6 Preventing expansion

```

\exp_not:o
\exp_not:f
\exp_not:v
\exp_not:V
1685 \cs_new:Npn\exp_not:o#1{\exp_not:n\exp_after:wN{#1}}
1686 \cs_new:Npn\exp_not:f#1{
1687   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 #1}
1688 }
1689 \cs_new:Npn\exp_not:v#1{
1690   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:c {#1}}
1691 }
1692 \cs_new:Npn\exp_not:V#1{
1693   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:N #1}
1694 }

```

`\exp_not:c` A helper function.

```

1695 \cs_new:Npn\exp_not:c#1{\exp_after:wN\exp_not:N\cs:w#1\cs_end:}

```

## 97.7 Defining function variants

```

\cs_generate_variant:Nn #1 : Base form of a function; e.g., \tl_set:Nn
\cs_generate_variant_aux:nnNn #2 : One or more variant argument specifiers; e.g., {Nx,c,cx}
\cs_generate_variant_aux:nnw
\cs_generate_variant_aux:N

```

Split up the original base function to grab its name and signature consisting of  $k$  letters. Then we wish to iterate through the list of variant argument specifiers, and for each one construct a new function name using the original base name, the variant signature consisting of  $l$  letters and the last  $k - l$  letters of the base signature. For example, for a base function `\tl_set:Nn` which needs a `c` variant form, we want the new signature to be `cn`.

```

1696 \cs_new_protected:Npn \cs_generate_variant:Nn #1 {
1697   \chk_if_exist_cs:N #1
1698   \cs_split_function:NN #1 \cs_generate_variant_aux:nnNn
1699 }

```

We discard the boolean and then set off a loop through the desired variant forms.

```

1700 \cs_set:Npn \cs_generate_variant_aux:nnNn #1#2#3#4{

```

```

1701 \cs_generate_variant_aux:nnw {#1}{#2} #4,?,\q_recursion_stop
1702 }

```

Next is the real work to be done. We now have 1: base name, 2: base signature, 3: beginning of variant signature. To construct the new csname and the `\exp_args:Ncc` form, we need the variant signature. In our example, we wanted to discard the first two letters of the base signature because the variant form started with `cc`. This is the same as putting first `cc` in the signature and then `\use_none:nn` followed by the base signature `NNn`. We therefore call a small loop that outputs an `n` for each letter in the variant signature and use this to call the correct `\use_none:` variant. Firstly though, we check whether to terminate the loop.

```

1703 \cs_set:Npn \cs_generate_variant_aux:nnw #1 #2 #3, {
1704   \if:w ? #3
1705     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1706   \fi:

```

Then check if the variant form has already been defined.

```

1707 \cs_if_free:cTF {
1708   #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1709 }
1710 {

```

If not, then define it and then additionally check if the `\exp_args:N` form needed is defined.

```

1711   \cs_generate_variant_aux:ccpx { #1 : #2 }
1712   {
1713     #1:#3 \use:c{use_none:\cs_generate_variant_aux:N #3 ?}#2
1714   }
1715   {
1716     \exp_not:c { exp_args:N #3} \exp_not:c {#1:#2}
1717   }
1718   \cs_generate_internal_variant:n {#3}
1719 }

```

Otherwise tell that it was already defined.

```

1720 {
1721   \iow_log:x{
1722     Variant~\token_to_str:c {
1723       #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1724     }~already~defined;~ not~ changing~ it~on~line~
1725     \tex_the:D \tex_inputlineno:D
1726   }
1727 }

```

Recurse.

```

1728 \cs_generate_variant_aux:nnw{#1}{#2}
1729 }

```

The small loop for defining the required number of ns. Break when seeing a ?.

```

1730 \cs_set:Npn \cs_generate_variant_aux:N #1{
1731   \if:w ?#1 \exp_after:wN\use_none:nn \fi: n \cs_generate_variant_aux:N
1732 }

```

```

\cs_generate_variant_aux:Ncpx
\cs_generate_variant_aux:ccpx
\cs_generate_variant_aux:w

```

The idea here is to pick up protected parent functions, using the nature of the meaning string that they generate. The test here is almost the same as `\tl_if_empty:nTF`, but has to be hard-coded as that function is not yet available and because it has to match both long and short macros.

```

1733 \group_begin:
1734 \tex_lccode:D '\Z = '\d \scan_stop:
1735 \tex_lccode:D '\? ='\ \ \scan_stop:
1736 \tex_catcode:D '\P = 12 \scan_stop:
1737 \tex_catcode:D '\R = 12 \scan_stop:
1738 \tex_catcode:D '\O = 12 \scan_stop:
1739 \tex_catcode:D '\T = 12 \scan_stop:
1740 \tex_catcode:D '\E = 12 \scan_stop:
1741 \tex_catcode:D '\C = 12 \scan_stop:
1742 \tex_catcode:D '\Z = 12 \scan_stop:
1743 \tex_lowercase:D {
1744   \group_end:
1745   \cs_new_nopar:Npn \cs_generate_variant_aux:Ncpx #1
1746   {
1747     \exp_after:wN \cs_generate_variant_aux:w
1748     \tex_meaning:D #1 ? PROTECTEZ \q_stop
1749   }
1750   \cs_new_nopar:Npn \cs_generate_variant_aux:ccpx
1751   { \exp_args:Nc \cs_generate_variant_aux:Ncpx}
1752   \cs_new:Npn \cs_generate_variant_aux:w
1753   #1 ? PROTECTEZ #2 \q_stop
1754   {
1755     \exp_after:wN \tex_ifx:D \exp_after:wN
1756     \q_no_value \etex_detokenize:D {#1} \q_no_value
1757     \exp_after:wN \cs_new_protected_nopar:cpx
1758     \tex_else:D
1759     \exp_after:wN \cs_new_nopar:cpx
1760     \tex_fi:D
1761   }
1762 }

```

```
\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`

```

1763 \cs_new_protected:Npn \cs_generate_internal_variant:n #1 {
1764   \cs_if_free:cT { \exp_args:N #1 }{

```

We use `new` to log the definition if we have to make one.

```

1765     \cs_new:cpx { exp_args:N #1 }
1766               { \cs_generate_internal_variant_aux:n #1 : }
1767   }
1768 }

```

`\generate_internal_variant_aux:n` This command grabs char by char outputting `\::#1` (not expanded further) until we see a `::`. That colon is in fact also turned into `\:::` so that the required structure for `\exp_args...` commands is correctly terminated.

```

1769 \cs_new:Npn \cs_generate_internal_variant_aux:n #1 {
1770   \exp_not:c{\::#1}
1771   \if_meaning:w #1 :
1772     \exp_after:wN \use_none:n
1773   \fi:
1774   \cs_generate_internal_variant_aux:n
1775 }

```

## 97.8 Last-unbraced versions

`\exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced

1776 \cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
1777 \cs_new:Npn \::f_unbraced \:::#1#2 {
1778   \exp_after:wN \exp_arg_last_unbraced:nn
1779   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1780 }
1781 \cs_new:Npn \::o_unbraced \:::#1#2 {
1782   \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2 } {#1}
1783 }
1784 \cs_new:Npn \::V_unbraced \:::#1#2 {
1785   \exp_after:wN \exp_arg_last_unbraced:nn
1786   \exp_after:wN { \tex_romannumeral:D -'0 \exp_eval_register:N #2 } {#1}
1787 }
1788 \cs_new:Npn \::v_unbraced \:::#1#2 {
1789   \exp_after:wN \exp_arg_last_unbraced:nn
1790   \exp_after:wN {
1791     \tex_romannumeral:D -'0 \exp_eval_register:c {#2}
1792   } {#1}
1793 }

```

`\exp_last_unbraced:NV` Now the business end.

```

\exp_last_unbraced:No
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo

1794 \cs_new_nopar:Npn \exp_last_unbraced:Nf { \::f_unbraced \::: }
1795 \cs_new_nopar:Npn \exp_last_unbraced:NV { \::V_unbraced \::: }
1796 \cs_new_nopar:Npn \exp_last_unbraced:No { \::o_unbraced \::: }
1797 \cs_new_nopar:Npn \exp_last_unbraced:Nv { \::v_unbraced \::: }
1798 \cs_new_nopar:Npn \exp_last_unbraced:NcV {

```

```

1799 \::c \::V_unbraced \:::
1800 }
1801 \cs_new_nopar:Npn \exp_last_unbraced:NNV {
1802 \::N \::V_unbraced \:::
1803 }
1804 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3 {
1805 \exp_after:wN #1 \exp_after:wN #2 #3
1806 }
1807 \cs_new_nopar:Npn \exp_last_unbraced:NNNV {
1808 \::N \::N \::V_unbraced \:::
1809 }
1810 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4 {
1811 \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4
1812 }

```

## 97.9 Items held from earlier

\str\_if\_eq\_p:Vn These cannot come earlier as they need \cs\_generate\_variant:Nn.

```

\str_if_eq:VnTF
\str_if_eq_p:onTF
\str_if_eq_p:nV
\str_if_eq:nVTF
\str_if_eq_p:no
\str_if_eq:noTF
\str_if_eq_p:VV
\str_if_eq:VVTF
1813 \cs_generate_variant:Nn \str_if_eq_p:nn { V }
1814 \cs_generate_variant:Nn \str_if_eq_p:nn { o }
1815 \cs_generate_variant:Nn \str_if_eq_p:nn { nV }
1816 \cs_generate_variant:Nn \str_if_eq_p:nn { no }
1817 \cs_generate_variant:Nn \str_if_eq_p:nn { VV }
1818 \cs_generate_variant:Nn \str_if_eq:nnT { V }
1819 \cs_generate_variant:Nn \str_if_eq:nnT { o }
1820 \cs_generate_variant:Nn \str_if_eq:nnT { nV }
1821 \cs_generate_variant:Nn \str_if_eq:nnT { no }
1822 \cs_generate_variant:Nn \str_if_eq:nnT { VV }
1823 \cs_generate_variant:Nn \str_if_eq:nnF { V }
1824 \cs_generate_variant:Nn \str_if_eq:nnF { o }
1825 \cs_generate_variant:Nn \str_if_eq:nnF { nV }
1826 \cs_generate_variant:Nn \str_if_eq:nnF { no }
1827 \cs_generate_variant:Nn \str_if_eq:nnF { VV }
1828 \cs_generate_variant:Nn \str_if_eq:nnTF { V }
1829 \cs_generate_variant:Nn \str_if_eq:nnTF { o }
1830 \cs_generate_variant:Nn \str_if_eq:nnTF { nV }
1831 \cs_generate_variant:Nn \str_if_eq:nnTF { no }
1832 \cs_generate_variant:Nn \str_if_eq:nnTF { VV }
1833 </initex | package>

```

Show token usage:

```

1834 <*showmemory>
1835 \showMemUsage
1836 </showmemory>

```

## 98 l3prg implementation

### 98.1 Variables

<code>\l_tmpa_bool</code> <code>\g_tmpa_bool</code>	Reserved booleans.
--	--------------------

<code>\g_prg_inline_level_int</code>	Global variable to track the nesting of the stepwise inline loop.
--------------------------------------	---

### 98.2 Module code

We start by ensuring that the required packages are loaded.

```

1837 <*package>
1838 \ProvidesExplPackage
1839   {\filename}{\filedate}{\fileversion}{\filedescription}
1840 \package_check_loaded_expl:
1841 </package>
1842 <*initex | package>

```

<code>\prg_return_true:</code>	These are all defined in l3basics, as they are needed “early”. This is just a reminder that
<code>\prg_return_false:</code>	that is the case!

```

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn

```

### 98.3 Choosing modes

For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```

1843 \prg_set_conditional:Npnn \mode_if_vertical: {p,TF,T,F}{
1844   \if_mode_vertical:
1845   \prg_return_true: \else: \prg_return_false: \fi:
1846 }

```

<code>\mode_if_horizontal_p:</code>	For testing horizontal mode.
<code>\mode_if_horizontal:TF</code>	

```

1847 \prg_set_conditional:Npnn \mode_if_horizontal: {p,TF,T,F}{
1848   \if_mode_horizontal:
1849   \prg_return_true: \else: \prg_return_false: \fi:
1850 }

```

`\mode_if_inner_p:` For testing inner mode.

`\mode_if_inner:TF`

```
1851 \prg_set_conditional:Npnn \mode_if_inner: {p,TF,T,F}{
1852   \if_mode_inner:
1853   \prg_return_true: \else: \prg_return_false: \fi:
1854 }
```

`\mode_if_math_p:` For testing math mode. Uses the kern-save `\scan_align_safe_stop:`.

`\mode_if_math:TF`

```
1855 \prg_set_conditional:Npnn \mode_if_math: {p,TF,T,F}{
1856   \scan_align_safe_stop: \if_mode_math:
1857   \prg_return_true: \else: \prg_return_false: \fi:
1858 }
```

### Alignment safe grouping and scanning

`\group_align_safe_begin:` `\group_align_safe_end:` T<sub>E</sub>X’s alignment structures present many problems. As Knuth says himself in *T<sub>E</sub>X: 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<sub>E</sub>X 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<sub>E</sub>Xbook*...

```
1859 \cs_new_nopar:Npn \group_align_safe_begin: {
1860   \if_false:{\fi:\if_num:w'}=\c_zero\fi:}
1861 \cs_new_nopar:Npn \group_align_safe_end:   {\if_num:w'=\c_zero}\fi:}
```

`\scan_align_safe_stop:` When T<sub>E</sub>X 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<sub>E</sub>X’s scanning ahead. On the other hand we don’t want to insert a `\scan_stop:` every time as that will destroy kerning between letters<sup>9</sup> Unfortunately there is no way to detect if we’re in the beginning of an alignment cell as they have different characteristics depending on column number etc. However we *can* detect if we’re in an alignment cell by checking the current group type and we can also check if the previous node was a character or ligature. What is done here is that `\scan_stop:` is only inserted iff a) we’re in the outer part of an alignment cell and b) the last node *wasn’t* a char node or a ligature node.

```
1862 \cs_new_nopar:Npn \scan_align_safe_stop: {
```

---

<sup>9</sup>Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

```

1863 \int_compare:nNnT \etex_currentgrouptype:D = \c_six
1864 {
1865   \int_compare:nNnF \etex_lastnodetype:D = \c_zero
1866   {
1867     \int_compare:nNnF \etex_lastnodetype:D = \c_seven
1868     \scan_stop:
1869   }
1870 }
1871 }

```

## 98.4 Producing $n$ copies

```

\prg_replicate:nn
\prg_replicate_aux:N
\prg_replicate_first_aux:N

```

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. Finally we must ensure that the cascade comes to a peaceful end so we make it so that the original csname `TeX` is creating is simply `\prg_do_nothing:` expanding to nothing.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it will severely affect the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use. An alternative approach is to create a string of `m`'s with `\int_to_roman:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

1872 \cs_new_nopar:Npn \prg_replicate:nn #1{
1873   \cs:w prg_do_nothing:
1874   \exp_after:wN\prg_replicate_first_aux:N
1875   \tex_romannumeral:D -'\q \int_eval:n{#1} \cs_end:
1876   \cs_end:
1877 }
1878 \cs_new_nopar:Npn \prg_replicate_aux:N#1{
1879   \cs:w prg_replicate_#1:n\prg_replicate_aux:N
1880 }
1881 \cs_new_nopar:Npn \prg_replicate_first_aux:N#1{
1882   \cs:w prg_replicate_first_#1:n\prg_replicate_aux:N
1883 }

```

Then comes all the functions that do the hard work of inserting all the copies.

```

1884 \cs_new_nopar:Npn \prg_replicate_ :n #1{% no, this is not a typo!
1885 \cs_new:cpn {prg_replicate_0:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}}
1886 \cs_new:cpn {prg_replicate_1:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}}#1}
1887 \cs_new:cpn {prg_replicate_2:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1}
1888 \cs_new:cpn {prg_replicate_3:n}#1{
1889 \cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1#1}
1890 \cs_new:cpn {prg_replicate_4:n}#1{
1891 \cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1#1#1}
1892 \cs_new:cpn {prg_replicate_5:n}#1{
1893 \cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1#1#1#1}
1894 \cs_new:cpn {prg_replicate_6:n}#1{
1895 \cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1#1#1#1#1}
1896 \cs_new:cpn {prg_replicate_7:n}#1{
1897 \cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1#1#1#1#1#1}
1898 \cs_new:cpn {prg_replicate_8:n}#1{
1899 \cs_end:{#1#1#1#1#1#1#1#1#1}}#1#1#1#1#1#1#1#1}
1900 \cs_new:cpn {prg_replicate_9:n}#1{
1901 \cs_end:{#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...

```

1902 \cs_new:cpn {prg_replicate_first_ -:n}#1{\cs_end: \ERROR }
1903 \cs_new:cpn {prg_replicate_first_0:n}#1{\cs_end: }
1904 \cs_new:cpn {prg_replicate_first_1:n}#1{\cs_end: #1}
1905 \cs_new:cpn {prg_replicate_first_2:n}#1{\cs_end: #1#1}
1906 \cs_new:cpn {prg_replicate_first_3:n}#1{\cs_end: #1#1#1}
1907 \cs_new:cpn {prg_replicate_first_4:n}#1{\cs_end: #1#1#1#1}
1908 \cs_new:cpn {prg_replicate_first_5:n}#1{\cs_end: #1#1#1#1#1}
1909 \cs_new:cpn {prg_replicate_first_6:n}#1{\cs_end: #1#1#1#1#1#1}
1910 \cs_new:cpn {prg_replicate_first_7:n}#1{\cs_end: #1#1#1#1#1#1#1}
1911 \cs_new:cpn {prg_replicate_first_8:n}#1{\cs_end: #1#1#1#1#1#1#1#1}
1912 \cs_new:cpn {prg_replicate_first_9:n}#1{\cs_end: #1#1#1#1#1#1#1#1#1}

```

```

\prg_stepwise_function:nnnN
prg_stepwise_function_incr:nnnN
prg_stepwise_function_decr:nnnN

```

A stepwise function. Firstly we check the direction of the steps #2 since that will depend on which test we should use. If the step is positive we use a greater than test, otherwise a less than test. If the test comes out true exit, otherwise perform #4, add the step to #1 and try again with this new value of #1.

```

1913 \cs_new:Npn \prg_stepwise_function:nnnN #1#2{
1914 \int_compare:nNnTF{#2}<\c_zero
1915 {\exp_args:Nf\prg_stepwise_function_decr:nnnN }
1916 {\exp_args:Nf\prg_stepwise_function_incr:nnnN }
1917 {\int_eval:n{#1}}{#2}
1918 }
1919 \cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4{
1920 \int_compare:nNnF {#1}>{#3}
1921 {
1922 #4{#1}

```

```

1923     \exp_args:Nf \prg_stepwise_function_incr:nnnN
1924     {\int_eval:n{#1 + #2}}
1925     {#2}{#3}{#4}
1926   }
1927 }
1928 \cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4{
1929   \int_compare:nNnF {#1}<{#3}
1930   {
1931     #4{#1}
1932     \exp_args:Nf \prg_stepwise_function_decr:nnnN
1933     {\int_eval:n{#1 + #2}}
1934     {#2}{#3}{#4}
1935   }
1936 }

```

`\g_prg_inline_level_int` This function uses the same approach as for instance `\clist_map_inline:Nn` to allow arbitrary nesting. First construct the special function and then call an auxiliary one which just carries the newly constructed `cname`. Must make assignments global when we maintain our own stack.

```

1937 \int_new:N\g_prg_inline_level_int
1938 \cs_new_protected:Npn\prg_stepwise_inline:nnnn #1#2#3#4{
1939   \int_gincr:N \g_prg_inline_level_int
1940   \cs_gset_nopar:cpn{prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}##1{#4}
1941   \int_compare:nNnTF {#2}<\c_zero
1942   {\exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
1943   {\exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
1944   {prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}
1945   {\int_eval:n{#1}} {#2} {#3}
1946   \int_gdecr:N \g_prg_inline_level_int
1947 }
1948 \cs_new:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4{
1949   \int_compare:nNnF {#2}>{#4}
1950   {
1951     #1{#2}
1952     \exp_args:NNf \prg_stepwise_inline_incr:Nnnn #1
1953     {\int_eval:n{#2 + #3}} {#3}{#4}
1954   }
1955 }
1956 \cs_new:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4{
1957   \int_compare:nNnF {#2}<{#4}
1958   {
1959     #1{#2}
1960     \exp_args:NNf \prg_stepwise_inline_decr:Nnnn #1
1961     {\int_eval:n{#2 + #3}} {#3}{#4}
1962   }
1963 }

```

`\prg_stepwise_variable:nnnNn` Almost the same as above. Just store the value in `#4` and execute `#5`.  
`rg_stepwise_variable_decr:nnnNn`  
`rg_stepwise_variable_incr:nnnNn`

```

1964 \cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2 {
1965   \int_compare:nNnTF {#2}<\c_zero
1966     {\exp_args:Nf\prg_stepwise_variable_decr:nnnNn}
1967     {\exp_args:Nf\prg_stepwise_variable_incr:nnnNn}
1968     {\int_eval:n{#1}}{#2}
1969 }
1970 \cs_new_protected:Npn \prg_stepwise_variable_incr:nnnNn #1#2#3#4#5 {
1971   \int_compare:nNnF {#1}>{#3}
1972   {
1973     \cs_set_nopar:Npn #4{#1} #5
1974     \exp_args:Nf \prg_stepwise_variable_incr:nnnNn
1975     {\int_eval:n{#1 + #2}}{#2}{#3}#4{#5}
1976   }
1977 }
1978 \cs_new_protected:Npn \prg_stepwise_variable_decr:nnnNn #1#2#3#4#5 {
1979   \int_compare:nNnF {#1}<{#3}
1980   {
1981     \cs_set_nopar:Npn #4{#1} #5
1982     \exp_args:Nf \prg_stepwise_variable_decr:nnnNn
1983     {\int_eval:n{#1 + #2}}{#2}{#3}#4{#5}
1984   }
1985 }

```

## 98.5 Booleans

For normal booleans we set them to either `\c_true_bool` or `\c_false_bool` and then use `\if_bool:N` to choose the right branch. The functions return either the TF, T, or F case *after* ending the `\if_bool:N`. We only define the N versions here as the c versions can easily be constructed with the expansion module.

```

\bool_new:N      Defining and setting a boolean is easy.
\bool_new:c
\bool_set_true:N 1986 \cs_new_protected_nopar:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\bool_set_true:c 1987 \cs_new_protected_nopar:Npn \bool_new:c #1 { \cs_new_eq:cN {#1} \c_false_bool }
\bool_set_false:N 1988 \cs_new_protected_nopar:Npn \bool_set_true:N #1 { \cs_set_eq:NN #1 \c_true_bool }
\bool_set_false:c 1989 \cs_new_protected_nopar:Npn \bool_set_true:c #1 { \cs_set_eq:cN {#1} \c_true_bool }
\bool_gset_true:N 1990 \cs_new_protected_nopar:Npn \bool_set_false:N #1 { \cs_set_eq:NN #1 \c_false_bool }
\bool_gset_true:c 1991 \cs_new_protected_nopar:Npn \bool_set_false:c #1 { \cs_set_eq:cN {#1} \c_false_bool }
\bool_gset_false:N 1992 \cs_new_protected_nopar:Npn \bool_gset_true:N #1 { \cs_gset_eq:NN #1 \c_true_bool }
\bool_gset_false:c 1993 \cs_new_protected_nopar:Npn \bool_gset_true:c #1 { \cs_gset_eq:cN {#1} \c_true_bool }
1994 \cs_new_protected_nopar:Npn \bool_gset_false:N #1 { \cs_gset_eq:NN #1 \c_false_bool }
1995 \cs_new_protected_nopar:Npn \bool_gset_false:c #1 { \cs_gset_eq:cN {#1} \c_false_bool }

```

```

\bool_set_eq:NN  Setting a boolean to another is also pretty easy.
\bool_set_eq:Nc
\bool_set_eq:cN 1996 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:cc 1997 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_gset_eq:NN 1998 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:Nc
\bool_gset_eq:cN
\bool_gset_eq:cc

```

```

1999 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
2000 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
2001 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
2002 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
2003 \cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc

```

\l\_tmpa\_bool A few booleans just if you need them.

```

\g_tmpa_bool
2004 \bool_new:N \l_tmpa_bool
2005 \bool_new:N \g_tmpa_bool

```

\bool\_if\_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```

\bool_if_p:c
\bool_if:N $\overline{TF}$ 
\bool_if:c $\overline{TF}$ 
2006 \prg_set_conditional:Npn \bool_if:N #1 {p,TF,T,F}{
2007   \if_bool:N #1 \prg_return_true: \else: \prg_return_false: \fi:
2008 }
2009 \cs_generate_variant:Nn \bool_if_p:N {c}
2010 \cs_generate_variant:Nn \bool_if:N $\overline{TF}$  {c}
2011 \cs_generate_variant:Nn \bool_if:NT {c}
2012 \cs_generate_variant:Nn \bool_if:N $\overline{F}$  {c}

```

\bool\_while\_do:Nn A while loop where the boolean is tested before executing the statement. The ‘while’ version executes the code as long as the boolean is true; the ‘until’ version executes the code as long as the boolean is false.

```

\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn
2013 \cs_new:Npn \bool_while_do:Nn #1 #2 {
2014   \bool_if:NT #1 {#2 \bool_while_do:Nn #1 {#2}}
2015 }
2016 \cs_generate_variant:Nn \bool_while_do:Nn {c}

2017 \cs_new:Npn \bool_until_do:Nn #1 #2 {
2018   \bool_if:N $\overline{F}$  #1 {#2 \bool_until_do:Nn #1 {#2}}
2019 }
2020 \cs_generate_variant:Nn \bool_until_do:Nn {c}

```

\bool\_do\_while:Nn A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```

\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn
2021 \cs_new:Npn \bool_do_while:Nn #1 #2 {
2022   #2 \bool_if:NT #1 {\bool_do_while:Nn #1 {#2}}
2023 }
2024 \cs_generate_variant:Nn \bool_do_while:Nn {c}

2025 \cs_new:Npn \bool_do_until:Nn #1 #2 {
2026   #2 \bool_if:N $\overline{F}$  #1 {\bool_do_until:Nn #1 {#2}}
2027 }
2028 \cs_generate_variant:Nn \bool_do_until:Nn {c}

```

## 98.6 Parsing boolean expressions

<code>\bool_if_p:n</code>	Evaluating the truth value of a list of predicates is done using an input syntax somewhat
<code>\bool_if:nTF</code>	similar to the one found in other programming languages with ( and ) for grouping, !
<code>\bool_get_next:N</code>	for logical ‘Not’, && for logical ‘And’ and    for logical Or. We shall use the terms Not,
<code>\bool_cleanup:N</code>	And, Or, Open and Close for these operations.
<code>\bool_choose:NN</code>	Any expression is terminated by a Close operation. Evaluation happens from left to right
<code>\bool_!:w</code>	in the following manner using a GetNext function:
<code>\bool_Not:w</code>	
<code>\bool_Not:w</code>	
<code>\bool_(w</code>	<ul style="list-style-type: none"> <li>• If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.</li> </ul>
<code>\bool_p:w</code>	
<code>\bool_8_1:w</code>	<ul style="list-style-type: none"> <li>• If a Not is seen, insert a negating function (if-even in this case) and call GetNext.</li> </ul>
<code>\bool_I_1:w</code>	
<code>\bool_8_0:w</code>	<ul style="list-style-type: none"> <li>• 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.</li> </ul>
<code>\bool_I_0:w</code>	
<code>\bool_)_0:w</code>	
<code>\bool_)_1:w</code>	The Eval function then contains a post-processing operation which grabs the instruction
<code>\bool_S_0:w</code>	following the predicate. This is either And, Or or Close. In each case the truth value is
<code>\bool_S_1:w</code>	used to determine where to go next. The following situations can arise:

**$\langle true \rangle$ And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

**$\langle false \rangle$ And** Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return  $\langle false \rangle$ .

**$\langle true \rangle$ Or** Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return  $\langle true \rangle$ .

**$\langle false \rangle$ Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

**$\langle true \rangle$ Close** Current truth value is true, Close seen, return  $\langle true \rangle$ .

**$\langle false \rangle$ Close** Current truth value is false, Close seen, return  $\langle false \rangle$ .

We introduce an additional Stop operation with the following semantics:

**$\langle true \rangle$ Stop** Current truth value is true, return  $\langle true \rangle$ .

**$\langle false \rangle$ Stop** Current truth value is false, return  $\langle false \rangle$ .

The reasons for this follow below.

Now for how these works in practice. The canonical true and false values have numerical values 1 and 0 respectively. We evaluate this using the primitive `\tex_number:D` operation. First we issue a `\group_align_safe_begin:` as we are using && as syntax

shorthand for the And operation and we need to hide it for T<sub>E</sub>X. 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.

```

2029 \cs_new:Npn \bool_if_p:n #1{
2030   \group_align_safe_begin:
2031   \bool_get_next:N ( #1 )S
2032 }

```

The GetNext operation. We make it a switch: If not a `!` or `(`, we assume it is a predicate.

```

2033 \cs_new:Npn \bool_get_next:N #1{
2034   \use:c {
2035     bool_
2036     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
2037     :w
2038   } #1
2039 }

```

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.

```

2040 \cs_new:Npn \bool_get_not_next:N #1{
2041   \use:c {
2042     bool_not_
2043     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
2044     :w
2045   } #1
2046 }

```

We need these later on to nullify the unity operation `!!`.

```

2047 \cs_new:Npn \bool_get_next:NN #1#2{
2048   \bool_get_next:N #2
2049 }
2050 \cs_new:Npn \bool_get_not_next:NN #1#2{
2051   \bool_get_not_next:N #2
2052 }

```

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.

```

2053 \cs_new:cpn { bool_!:w } #1#2 {
2054   \if_meaning:w ( #2
2055     \exp_after:wN \bool_Not:w
2056   \else:
2057     \if_meaning:w ! #2
2058     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_next:NN

```

```

2059     \else:
2060         \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
2061     \fi:
2062 \fi:
2063 #2
2064 }

```

Variant called when already inside a NOT. Essentially the opposite of the above.

```

2065 \cs_new:cpn { bool_not_!:w } #1#2 {
2066     \if_meaning:w ( #2
2067         \exp_after:wN \bool_not_Not:w
2068     \else:
2069         \if_meaning:w ! #2
2070         \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_not_next:NN
2071     \else:
2072         \exp_after:wN \exp_after:wN \exp_after:wN \bool_not_Not:N
2073     \fi:
2074 \fi:
2075 #2
2076 }

```

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.

```

2077 \cs_new:Npn \bool_Not:w {
2078     \exp_after:wN \tex_number:D \bool_get_not_next:N
2079 }
2080 \cs_new:Npn \bool_not_Not:w {
2081     \exp_after:wN \tex_number:D \bool_get_next:N
2082 }

```

These occur when processing `!<bool>` and can be evaluated directly.

```

2083 \cs_new:Npn \bool_Not:N #1 {
2084     \exp_after:wN \bool_p:w
2085     \if_meaning:w #1 \c_true_bool
2086         \c_false_bool
2087     \else:
2088         \c_true_bool
2089     \fi:
2090 }
2091 \cs_new:Npn \bool_not_Not:N #1 {
2092     \exp_after:wN \bool_p:w
2093     \if_meaning:w #1 \c_true_bool
2094         \c_true_bool
2095     \else:
2096         \c_false_bool
2097     \fi:
2098 }

```

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 `!(...)`.

```

2099 \cs_new:cpn {bool_(w)#1{
2100   \exp_after:wN \bool_cleanup:N \tex_number:D \bool_get_next:N
2101 }
2102 \cs_new:cpn {bool_not_(w)#1{
2103   \exp_after:wN \bool_not_cleanup:N \tex_number:D \bool_get_next:N
2104 }

```

Otherwise just evaluate the predicate and look for And, Or or Close afterward.

```

2105 \cs_new:cpn {bool_p:w}{\exp_after:wN \bool_cleanup:N \tex_number:D }
2106 \cs_new:cpn {bool_not_p:w}{\exp_after:wN \bool_not_cleanup:N \tex_number:D }

```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```

2107 \cs_new_nopar:Npn \bool_cleanup:N #1{
2108   \exp_after:wN \bool_choose:NN \exp_after:wN #1
2109   \int_to_roman:w-'\q
2110 }
2111 \cs_new_nopar:Npn \bool_not_cleanup:N #1{
2112   \exp_after:wN \bool_not_choose:NN \exp_after:wN #1
2113   \int_to_roman:w-'\q
2114 }

```

Branching the six way switch. Reversals should be reasonably straightforward. When programming this, however, I got things around the wrong way a few times. (Will's hacks onto Morten's code, that is.)

```

2115 \cs_new_nopar:Npn \bool_choose:NN #1#2{ \use:c{bool_#2_#1:w} }
2116 \cs_new_nopar:Npn \bool_not_choose:NN #1#2{ \use:c{bool_not_#2_#1:w} }

```

Continues scanning. Must remove the second `&` or `|`.

```

2117 \cs_new_nopar:cpn{bool_&_1:w}&{\bool_get_next:N}
2118 \cs_new_nopar:cpn{bool_|_0:w}|{\bool_get_next:N}
2119 \cs_new_nopar:cpn{bool_not_&_0:w}&{\bool_get_next:N}
2120 \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.

```

2121 \cs_new_nopar:cpn{bool_)_0:w}{ \c_false_bool }
2122 \cs_new_nopar:cpn{bool_)_1:w}{ \c_true_bool }
2123 \cs_new_nopar:cpn{bool_not_)_0:w}{ \c_true_bool }
2124 \cs_new_nopar:cpn{bool_not_)_1:w}{ \c_false_bool }
2125 \cs_new_nopar:cpn{bool_S_0:w}{\group_align_safe_end: \c_false_bool }
2126 \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.

```

2127 \cs_new:cpn{bool_&_0:w}&{\bool_eval_skip_to_end:Nw \c_false_bool}
2128 \cs_new:cpn{bool_|_1:w}|{\bool_eval_skip_to_end:Nw \c_true_bool}
2129 \cs_new:cpn{bool_not_&_1:w}&{\bool_eval_skip_to_end:Nw \c_false_bool}
2130 \cs_new:cpn{bool_not_|_0:w}|{\bool_eval_skip_to_end:Nw \c_true_bool}

```

```

\bool_eval_skip_to_end:Nw
\bool_eval_skip_to_end_aux:Nw
\bool_eval_skip_to_end_auxii:Nw

```

There is always at least one ) waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first And. Note the extra Close at the end.

```
\c_false_bool  && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first Close. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

This contains two Open markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a () pair and what preceded the Open – but leave the contents as it may contain Open tokens itself – leaving

```
(abc && xyz) && ((xyz) && (def)))
```

Another round of this gives us

```
(abc && xyz
```

which still contains an Open so we remove another () pair, giving us

```
abc && xyz && ((xyz) && (def)))
```

Again we read up to a Close and again find Open tokens:

```
abc && xyz && ((xyz
```

Further reduction gives us

```
(xyz && (def)))
```

and then

```
(xyz && (def
```

with reduction to

```
xyz && (def))
```

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

This whole operation could be made a lot simpler if we were allowed to do simple pattern matching. With a new enough pdfTeX one can do that sort of thing to test for existence of particular tokens.

```
2131 \cs_new:Npn \bool_eval_skip_to_end:Nw #1#2){
2132   \bool_eval_skip_to_end_aux:Nw #1 #2(\q_no_value\q_stop{#2}
2133 }
```

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

```
2134 \cs_new:Npn \bool_eval_skip_to_end_aux:Nw #1#2(#3#4\q_stop#5{
2135   \quark_if_no_value:NTF #3
2136   { #1 }
2137   { \bool_eval_skip_to_end_auxii:Nw #1 #5 }
2138 }
```

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!

```
2139 \cs_new:Npn \bool_eval_skip_to_end_auxii:Nw #1#2(#3){
2140   \bool_eval_skip_to_end:Nw #1#3 )
2141 }
```

\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
2142 \cs_new:Npn \bool_set:Nn #1#2 {\tex_chardef:D #1 = \bool_if_p:n {#2}}
2143 \cs_new:Npn \bool_gset:Nn #1#2 {
2144   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
2145 }
2146 \cs_generate_variant:Nn \bool_set:Nn {c}
2147 \cs_generate_variant:Nn \bool_gset:Nn {c}
```

\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.

```
2148 \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.

```

2149 \cs_new:Npn \bool_xor_p:nn #1#2 {
2150   \int_compare:nNnTF {\bool_if_p:n { #1 }} = {\bool_if_p:n { #2 }}
2151   {\c_false_bool}{\c_true_bool}
2152 }

2153 \prg_set_conditional:Npnn \bool_if:n #1 {TF,T,F}{
2154   \if_predicate:w \bool_if_p:n{#1}
2155   \prg_return_true: \else: \prg_return_false: \fi:
2156 }
```

`\bool_while_do:nn` #1 : Predicate test  
`\bool_until_do:nn` #2 : Code to execute

```

\bool_do_while:nn
\bool_do_until:nn
2157 \cs_new:Npn \bool_while_do:nn #1#2 {
2158   \bool_if:nT {#1} { #2 \bool_while_do:nn {#1}{#2} }
2159 }
2160 \cs_new:Npn \bool_until_do:nn #1#2 {
2161   \bool_if:nF {#1} { #2 \bool_until_do:nn {#1}{#2} }
2162 }
2163 \cs_new:Npn \bool_do_while:nn #1#2 {
2164   #2 \bool_if:nT {#1} { \bool_do_while:nn {#1}{#2} }
2165 }
2166 \cs_new:Npn \bool_do_until:nn #1#2 {
2167   #2 \bool_if:nF {#1} { \bool_do_until:nn {#1}{#2} }
2168 }
```

## 98.7 Case switch

`\prg_case_int:nnn` This case switch is in reality quite simple. It takes three arguments:  
`\prg_case_int_aux:nnn`

1. An integer expression you wish to find.
2. A list of pairs of  $\{\langle integer\ expr \rangle\} \{\langle code \rangle\}$ . The list can be as long as is desired and  $\langle integer\ expr \rangle$  can be negative.
3. The code to be executed if the value wasn't found.

We don't need the else case here yet, so leave it dangling in the input stream.

```

2169 \cs_new:Npn \prg_case_int:nnn #1 #2 {
```

We will be parsing on #1 for each step so we might as well evaluate it first in case it is complicated.

```

2170   \exp_args:Nf \prg_case_int_aux:nnn { \int_eval:n{#1}} #2
```

The ? below is just so there are enough arguments when we reach the end. And it made you look. ;-)

```

2171 \q_recursion_tail ? \q_recursion_stop
2172 }
2173 \cs_new:Npn \prg_case_int_aux:nnn #1#2#3{

```

If we reach the end, return the else case. We just remove braces.

```

2174 \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}

```

Otherwise we compare (which evaluates #2 for us)

```

2175 \int_compare:nNnTF{#1}={#2}

```

If true, we want to remove the remainder of the list, the else case and then execute the code specified. \prg\_end\_case:nw {#3} does just that in one go. This means f style expansion works the way one wants it to work.

```

2176 { \prg_end_case:nw {#3} }
2177 { \prg_case_int_aux:nnn {#1}}
2178 }

```

\prg\_case\_dim:nnn Same as \prg\_case\_dim:nnn except it is for *<dim>* registers.  
\prg\_case\_dim\_aux:nnn

```

2179 \cs_new:Npn \prg_case_dim:nnn #1 #2 {
2180 \exp_args:No \prg_case_dim_aux:nnn {\dim_use:N \dim_eval:n{#1}} #2
2181 \q_recursion_tail ? \q_recursion_stop
2182 }
2183 \cs_new:Npn \prg_case_dim_aux:nnn #1#2#3{
2184 \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
2185 \dim_compare:nNnTF{#1}={#2}
2186 { \prg_end_case:nw {#3} }
2187 { \prg_case_dim_aux:nnn {#1}}
2188 }

```

\prg\_case\_str:nnn Same as \prg\_case\_dim:nnn except it is for strings.  
\prg\_case\_str\_aux:nnn

```

2189 \cs_new:Npn \prg_case_str:nnn #1 #2 {
2190 \prg_case_str_aux:nnn {#1} #2
2191 \q_recursion_tail ? \q_recursion_stop
2192 }
2193 \cs_new:Npn \prg_case_str_aux:nnn #1#2#3{
2194 \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
2195 \str_if_eq:xxTF{#1}{#2}
2196 { \prg_end_case:nw {#3} }
2197 { \prg_case_str_aux:nnn {#1}}
2198 }

```

`\prg_case_tl:Nnn` Same as `\prg_case_dim:nnn` except it is for token list variables.  
`\prg_case_tl_aux:NNn`

```

2199 \cs_new:Npn \prg_case_tl:Nnn #1 #2 {
2200   \prg_case_tl_aux:NNn #1 #2
2201   \q_recursion_tail ? \q_recursion_stop
2202 }
2203 \cs_new:Npn \prg_case_tl_aux:NNn #1#2#3{
2204   \quark_if_recursion_tail_stop_do:Nn #2{\use:n}
2205   \tl_if_eq:NNTF #1 #2
2206   { \prg_end_case:nw {#3} }
2207   { \prg_case_tl_aux:NNn #1}
2208 }

```

`\prg_end_case:nw` Ending a case switch is always performed the same way so we optimize for this. #1 is the code to execute, #2 the remainder, and #3 the dangling else case.

```

2209 \cs_new:Npn \prg_end_case:nw #1#2\q_recursion_stop#3{#1}

```

## 98.8 Sorting

`\prg_define_quicksort:nnn` #1 is the name, #2 and #3 are the tokens enclosing the argument. For the somewhat strange *<clist>* type which doesn't enclose the items but uses a separator we define it by hand afterwards. When doing the first pass, the algorithm wraps all elements in braces and then uses a generic quicksort which works on token lists.

As an example

```
\prg_define_quicksort:nnn{seq}{\seq_elt:w}{\seq_elt_end:w}
```

defines the user function `\seq_quicksort:n` and furthermore expects to use the two functions `\seq_quicksort_compare:nnTF` which compares the items and `\seq_quicksort_function:n` which is placed before each sorted item. It is up to the programmer to define these functions when needed. For the `seq` type a sequence is a token list variable, so one additionally has to define

```
\cs_set_nopar:Npn \seq_quicksort:N{\exp_args:No\seq_quicksort:n}
```

For details on the implementation see “Sorting in TeX’s Mouth” by Bernd Raichle. Firstly we define the function for parsing the initial list and then the braced list afterwards.

```

2210 \cs_new_protected_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
2211   \cs_set:cpx{#1_quicksort:n}##1{
2212     \exp_not:c{#1_quicksort_start_partition:w} ##1
2213     \exp_not:n{#2\q_nil#3\q_stop}
2214   }
2215   \cs_set:cpx{#1_quicksort_braced:n}##1{
2216     \exp_not:c{#1_quicksort_start_partition_braced:n} ##1

```

```

2217 \exp_not:N\q_nil\exp_not:N\q_stop
2218 }
2219 \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
2220 \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2221 \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{-}{-}
2222 }
2223 \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
2224 \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2225 \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{-}{-}
2226 }

```

Now for doing the partitions.

```

2227 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2228 \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2229 {
2230 \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2231 \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2232 \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2233 }
2234 {##1}{##2}{##3}{##4}
2235 }
2236 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2237 \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2238 {
2239 \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2240 \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2241 \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2242 }
2243 {##1}{##2}{##3}{##4}
2244 }
2245 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2246 \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2247 {
2248 \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2249 \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2250 \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2251 }
2252 {##1}{##2}{##3}{##4}
2253 }
2254 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2255 \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2256 {
2257 \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2258 \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2259 \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2260 }
2261 {##1}{##2}{##3}{##4}
2262 }

```

This part of the code handles the two branches in each sorting. Again we will also have

to do it braced.

```

2263 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2264   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}##3}}
2265 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2266   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3{##4}}}}
2267 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2268   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}##2}{##3}}
2269 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2270   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2{##4}}}{##3}}
2271 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2272   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##4}##3}}
2273 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2274   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3{##4}}}}
2275 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2276   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}##2}{##3}}
2277 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2278   \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.

```

2279 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2280   \exp_not:c{#1_quicksort_braced:n}{##2}
2281   \exp_not:c{#1_quicksort_function:n}{##1}
2282   \exp_not:c{#1_quicksort_braced:n}{##3}
2283 }
2284 }

```

`\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.

```

2285 \prg_define_quicksort:nnn {prg}{-}{-}

```

`\prg_quicksort_function:n`  
`\prg_quicksort_compare:nnTF`

```

2286 \cs_set:Npn \prg_quicksort_function:n {\ERROR}
2287 \cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}

```

## 98.9 Variable type and scope

`\prg_variable_get_scope:N` Expandable functions to find the type of a variable, and to return g if the variable is global. The trick for `\prg_variable_get_scope:N` is the same as that in `\cs_split_function:NN`, but it can be simplified as the requirements here are less complex.

`\prg_variable_get_type:N`  
`\prg_variable_get_type:w`

```

2288 \group_begin:
2289   \tex_lccode:D '\& = '\g \tex_relax:D
2290   \tex_catcode:D '\& = \c_twelve \tex_relax:D
2291   \tl_to_lowercase:n {

```

```

2292 \group_end:
2293 \cs_new_nopar:Npn \prg_variable_get_scope:N #1 {
2294   \exp_last_unbraced:Nf \prg_variable_get_scope_aux:w
2295   { \cs_to_str:N #1 \exp_stop_f: \q_stop }
2296 }
2297 \cs_new_nopar:Npn \prg_variable_get_scope_aux:w #1#2 \q_stop {
2298   \token_if_eq_meaning:NNT & #1 {g}
2299 }
2300 }
2301 \group_begin:
2302 \tex_lccode:D '\& = '\_ \tex_relax:D
2303 \tex_catcode:D '\& = \c_twelve \tex_relax:D
2304 \tl_to_lowercase:n {
2305   \group_end:
2306   \cs_new_nopar:Npn \prg_variable_get_type:N #1 {
2307     \exp_after:wN \p;rg_variable_get_type_aux:w
2308     \token_to_str:N #1 & a \q_stop
2309   }
2310   \cs_new_nopar:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_stop {
2311     \token_if_eq_meaning:NNTF a #2 {
2312       #1
2313     }{
2314       \prg_variable_get_type_aux:w #2#3 \q_stop
2315     }
2316   }
2317 }

```

## 98.10 Mapping to variables

`\prg_new_map_functions:Nn`    The idea here is to generate all of the various mapping functions in one go. Everything is done with expansion so that the performance hit is taken at definition time and not at point of use. The inline version uses a counter as this keeps things nestable, and global to avoid problems with, for example, table cells.

`\prg_set_map_functions:Nn`

```

2318 \cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 {
2319   \cs_if_free:cTF { #2 _map_function:NN }
2320   { \prg_set_map_functions:Nn #1 {#2} }
2321   {
2322     \msg_kernel_error:nxx { code } { csname-already-defined }
2323     { \token_to_str:c { #2 _map_function:NN } }
2324   }
2325 }
2326 \cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 {
2327   \cs_gset_nopar:cpx { #2 _map_function:NN } ##1##2
2328   {
2329     \exp_not:N \tl_if_empty:NF ##1
2330     {
2331       \exp_not:N \exp_after:wN
2332       \exp_not:c { #2 _map_function_aux:Nw }

```

```

2333         \exp_not:N \exp_after:wN ##2 ##1
2334         \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2335     }
2336 }
2337 \cs_gset:cpx { #2 _map_function:nN } ##1##2
2338 {
2339     \exp_not:N \tl_if_blank:nF {##1}
2340     {
2341         \exp_not:c { #2 _map_function_aux:Nw } ##2 ##1
2342         \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2343     }
2344 }
2345 \cs_gset:cpx { #2 _map_function_aux:Nw } ##1##2 #1
2346 {
2347     \exp_not:N \quark_if_recursion_tail_stop:n {##2}
2348     ##1 {##2}
2349     \exp_not:c { #2 _map_function_aux:Nw } ##1
2350 }
2351 \cs_if_free:cT { g_ #2 _map_inline_int }
2352 { \int_new:c { g_ #2 _map_inline_int } }
2353 \cs_gset_protected_nopar:cpx { #2 _map_inline:Nn } ##1##2
2354 {
2355     \exp_not:N \tl_if_empty:NF ##1
2356     {
2357         \exp_not:N \int_gincr:N \exp_not:c { g_ #2 _map_inline_int }
2358         \cs_gset:cpn
2359         {
2360             #2 _map_inline_
2361             \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2362             :n
2363         }
2364         #####1 {##2}
2365         \exp_not:N \exp_last_unbraced:NcV
2366         \exp_not:c { #2 _map_function_aux:Nw }
2367         {
2368             #2 _map_inline_
2369             \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2370             :n
2371         }
2372         ##1 \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2373         \exp_not:N \int_gdecr:N \exp_not:c { g_ #2 _map_inline_int }
2374     }
2375 }
2376 \cs_gset_protected:cpx { #2 _map_inline:nn } ##1##2
2377 {
2378     \exp_not:N \tl_if_empty:nF {##1}
2379     {
2380         \exp_not:N \int_gincr:N \exp_not:c { g_ #2 _map_inline_int }
2381         \cs_gset:cpn
2382         {

```

```

2383         #2 _map_inline_
2384         \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2385         :n
2386     }
2387     #####1 {##2}
2388     \exp_not:N \exp_args:Nc
2389     \exp_not:c { #2 _map_function_aux:Nw }
2390     {
2391         #2 _map_inline_
2392         \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2393         :n
2394     }
2395     ##1 \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2396     \exp_not:N \int_gdec:N \exp_not:c { g_ #2 _map_inline_int }
2397 }
2398 }
2399 \cs_gset_eq:cN { #2 _map_break: }
2400 \use_none_delimit_by_q_recursion_stop:w
2401 }

```

That's it (for now).

```

2402 </initex | package>

2403 <*showmemory>
2404 \showMemUsage
2405 </showmemory>

```

## 99 l3quark implementation

We start by ensuring that the required packages are loaded. We check for `l3expan` since this a basic package that is essential for use of any higher-level package.

```

2406 <*package>
2407 \ProvidesExplPackage
2408   {\filename}{\filedate}{\fileversion}{\filedescription}
2409   \package_check_loaded_expl:
2410 </package>
2411 <*initex | package>

```

`\quark_new:N` Allocate a new quark.

```

2412 \cs_new_protected_nopar:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }

```

`\q_stop` `\q_stop` is often used as a marker in parameter text, `\q_no_value` is the canonical missing value, and `\q_nil` represents a nil pointer in some data structures.

```

2413 \quark_new:N \q_stop
2414 \quark_new:N \q_no_value
2415 \quark_new:N \q_nil

```

**\q\_error** We need two additional quarks. **\q\_error** delimits the end of the computation for purposes of error recovery. **\q\_mark** is used in parameter text when we need a scanning boundary that is distinct from **\q\_stop**.

```

2416 \quark_new:N\q_error
2417 \quark_new:N\q_mark

```

**\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.

```

2418 \quark_new:N\q_recursion_tail
2419 \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 we found the end marker.  
**\quark\_if\_recursion\_tail\_stop:N** To avoid this, we use a recursion end marker every time we do this kind of task. Also, if  
**\quark\_if\_recursion\_tail\_stop:o** the recursion end marker is found, we wrap things up and finish.

```

2420 \cs_new:Npn \quark_if_recursion_tail_stop:n #1 {
2421   \exp_after:wN\if_meaning:w
2422     \quark_if_recursion_tail_aux:w #1?\q_stop\q_recursion_tail\q_recursion_tail
2423     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2424   \fi:
2425 }
2426 \cs_new:Npn \quark_if_recursion_tail_stop:N #1 {
2427   \if_meaning:w#1\q_recursion_tail
2428     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2429   \fi:
2430 }
2431 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n {o}

```

**\rk\_if\_recursion\_tail\_stop\_do:nn**  
**\rk\_if\_recursion\_tail\_stop\_do:Nn**  
**\rk\_if\_recursion\_tail\_stop\_do:on**

```

2432 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2 {
2433   \exp_after:wN\if_meaning:w
2434     \quark_if_recursion_tail_aux:w #1?\q_stop\q_recursion_tail\q_recursion_tail
2435     \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2436   \else:
2437     \exp_after:wN\use_none:n
2438   \fi:
2439   {#2}
2440 }
2441 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2 {

```

```

2442 \if_meaning:w #1\q_recursion_tail
2443 \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2444 \else:
2445 \exp_after:wN\use_none:n
2446 \fi:
2447 {#2}
2448 }
2449 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn {on}

```

\quark\_if\_recursion\_tail\_aux:w

```

2450 \cs_new:Npn \quark_if_recursion_tail_aux:w #1#2 \q_stop \q_recursion_tail {#1}

```

\quark\_if\_no\_value\_p:N Here we test if we found a special quark as the first argument. We better start with  
\quark\_if\_no\_value\_p:n \q\_no\_value as the first argument since the whole thing may otherwise loop if #1 is  
\quark\_if\_no\_value:N $\underline{TF}$  wrongly given a string like aabc instead of a single token.<sup>10</sup>  
\quark\_if\_no\_value:n $\underline{TF}$

```

2451 \prg_new_conditional:Nnn \quark_if_no_value:N {p,TF,T,F} {
2452 \if_meaning:w \q_no_value #1
2453 \prg_return_true: \else: \prg_return_false: \fi:
2454 }

```

These tests are easy with \pdf\_strcmp:D available.

```

2455 \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
2456 \if_num:w \pdf_strcmp:D
2457 {\exp_not:N \q_no_value}
2458 {\exp_not:n{#1}} = \c_zero
2459 \prg_return_true: \else: \prg_return_false:
2460 \fi:
2461 }

```

\quark\_if\_nil\_p:N A function to check for the presence of \q\_nil.

\quark\_if\_nil:N $\underline{TF}$

```

2462 \prg_new_conditional:Nnn \quark_if_nil:N {p,TF,T,F} {
2463 \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi:
2464 }

```

\quark\_if\_nil\_p:n A function to check for the presence of \q\_nil.

\quark\_if\_nil\_p:V

\quark\_if\_nil\_p:o

\quark\_if\_nil:n $\underline{TF}$

\quark\_if\_nil:V $\underline{TF}$

\quark\_if\_nil:o $\underline{TF}$

```

2465 \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
2466 \if_num:w \pdf_strcmp:D
2467 {\exp_not:N \q_nil}
2468 {\exp_not:n{#1}} = \c_zero
2469 \prg_return_true: \else: \prg_return_false:
2470 \fi:
2471 }
2472 \cs_generate_variant:Nn \quark_if_nil_p:n {V}

```

---

<sup>10</sup>It may still loop in special circumstances however!

```

2473 \cs_generate_variant:Nn \quark_if_nil:nTF {V}
2474 \cs_generate_variant:Nn \quark_if_nil:nT {V}
2475 \cs_generate_variant:Nn \quark_if_nil:nF {V}
2476 \cs_generate_variant:Nn \quark_if_nil_p:n {o}
2477 \cs_generate_variant:Nn \quark_if_nil:nTF {o}
2478 \cs_generate_variant:Nn \quark_if_nil:nT {o}
2479 \cs_generate_variant:Nn \quark_if_nil:nF {o}

```

Show token usage:

```

2480
2481 <*showmemory>
2482 \showMemUsage
2483 </showmemory>

```

## 100 l3token implementation

### 100.1 Documentation of internal functions

\l\_peek\_true\_tl  
\l\_peek\_false\_tl

These token list variables are used internally when choosing either the true or false branches of a test.

\l\_peek\_search\_tl

Used to store \l\_peek\_search\_token.

\peek\_tmp:w

Scratch function used to gobble tokens from the input stream.

\l\_peek\_true\_aux\_tl  
\c\_peek\_true\_remove\_next\_tl

These token list variables are used internally when choosing either the true or false branches of a test.

\peek\_ignore\_spaces\_execute\_branches:  
\peek\_ignore\_spaces\_aux:

Functions used to ignore space tokens in the input stream.

## 100.2 Module code

First a few required packages to get this going.

```

2484 <*package>
2485 \ProvidesExplPackage
2486   {\filename}{\filedate}{\fileversion}{\filedescription}
2487 \package_check_loaded_expl:
2488 </package>
2489 <*initex | package>

```

## 100.3 Character tokens

```

\char_set_catcode:w
\char_set_catcode:nn
\char_value_catcode:w
\char_value_catcode:n
\char_show_value_catcode:w
\char_show_value_catcode:n

2490 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
2491 \cs_new_protected_nopar:Npn \char_set_catcode:nn #1#2 {
2492   \char_set_catcode:w #1 = \int_eval:w #2\int_eval_end:
2493 }
2494 \cs_new_nopar:Npn \char_value_catcode:w { \int_use:N \tex_catcode:D }
2495 \cs_new_nopar:Npn \char_value_catcode:n #1 {
2496   \char_value_catcode:w \int_eval:w #1\int_eval_end:
2497 }
2498 \cs_new_nopar:Npn \char_show_value_catcode:w {
2499   \tex_showthe:D \tex_catcode:D
2500 }
2501 \cs_new_nopar:Npn \char_show_value_catcode:n #1 {
2502   \char_show_value_catcode:w \int_eval:w #1\int_eval_end:
2503 }

```

```

\char_make_escape:N
\char_make_begin_group:N
\char_make_end_group:N
\char_make_math_shift:N
\char_make_alignment:N
\char_make_end_line:N
\char_make_parameter:N
\char_make_math_superscript:N
\char_make_math_subscript:N
\char_make_ignore:N
\char_make_space:N
\char_make_letter:N
\char_make_other:N
\char_make_active:N
\char_make_comment:N
\char_make_invalid:N

2504 \cs_new_protected_nopar:Npn \char_make_escape:N #1 { \char_set_catcode:nn {'#1} {\c
2505 \cs_new_protected_nopar:Npn \char_make_begin_group:N #1 { \char_set_catcode:nn {'#1} {\c
2506 \cs_new_protected_nopar:Npn \char_make_end_group:N #1 { \char_set_catcode:nn {'#1} {\c
2507 \cs_new_protected_nopar:Npn \char_make_math_shift:N #1 { \char_set_catcode:nn {'#1} {\c
2508 \cs_new_protected_nopar:Npn \char_make_alignment:N #1 { \char_set_catcode:nn {'#1} {\c
2509 \cs_new_protected_nopar:Npn \char_make_end_line:N #1 { \char_set_catcode:nn {'#1} {\c
2510 \cs_new_protected_nopar:Npn \char_make_parameter:N #1 { \char_set_catcode:nn {'#1} {\c
2511 \cs_new_protected_nopar:Npn \char_make_math_superscript:N #1 { \char_set_catcode:nn {'#1} {\c
2512 \cs_new_protected_nopar:Npn \char_make_math_subscript:N #1 { \char_set_catcode:nn {'#1} {\c
2513 \cs_new_protected_nopar:Npn \char_make_ignore:N #1 { \char_set_catcode:nn {'#1} {\c
2514 \cs_new_protected_nopar:Npn \char_make_space:N #1 { \char_set_catcode:nn {'#1} {\c
2515 \cs_new_protected_nopar:Npn \char_make_letter:N #1 { \char_set_catcode:nn {'#1} {\c
2516 \cs_new_protected_nopar:Npn \char_make_other:N #1 { \char_set_catcode:nn {'#1} {\c
2517 \cs_new_protected_nopar:Npn \char_make_active:N #1 { \char_set_catcode:nn {'#1} {\c
2518 \cs_new_protected_nopar:Npn \char_make_comment:N #1 { \char_set_catcode:nn {'#1} {\c
2519 \cs_new_protected_nopar:Npn \char_make_invalid:N #1 { \char_set_catcode:nn {'#1} {\c

```

<code>\char_make_escape:n</code>			
<code>\char_make_begin_group:n</code>			
<code>\char_make_end_group:n</code>	2520	<code>\cs_new_protected_nopar:Npn \char_make_escape:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_math_shift:n</code>	2521	<code>\cs_new_protected_nopar:Npn \char_make_begin_group:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_alignment:n</code>	2522	<code>\cs_new_protected_nopar:Npn \char_make_end_group:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_end_line:n</code>	2523	<code>\cs_new_protected_nopar:Npn \char_make_math_shift:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_parameter:n</code>	2524	<code>\cs_new_protected_nopar:Npn \char_make_alignment:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_math_superscript:n</code>	2525	<code>\cs_new_protected_nopar:Npn \char_make_end_line:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_math_subscript:n</code>	2526	<code>\cs_new_protected_nopar:Npn \char_make_parameter:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_ignore:n</code>	2527	<code>\cs_new_protected_nopar:Npn \char_make_math_superscript:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_space:n</code>	2528	<code>\cs_new_protected_nopar:Npn \char_make_math_subscript:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_letter:n</code>	2529	<code>\cs_new_protected_nopar:Npn \char_make_ignore:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_other:n</code>	2530	<code>\cs_new_protected_nopar:Npn \char_make_space:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_active:n</code>	2531	<code>\cs_new_protected_nopar:Npn \char_make_letter:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_comment:n</code>	2532	<code>\cs_new_protected_nopar:Npn \char_make_other:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
<code>\char_make_invalid:n</code>	2533	<code>\cs_new_protected_nopar:Npn \char_make_active:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
	2534	<code>\cs_new_protected_nopar:Npn \char_make_comment:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>
	2535	<code>\cs_new_protected_nopar:Npn \char_make_invalid:n</code>	<code>#1 { \char_set_catcode:nn {#1} {\c</code>

`\char_set_mathcode:w`  
`\char_set_mathcode:nn`  
`\char_gset_mathcode:w`  
`\char_gset_mathcode:nn`  
`\char_value_mathcode:w`  
`\char_value_mathcode:n`  
`\char_show_value_mathcode:w`  
`\char_show_value_mathcode:n`

Math codes.

```

2536 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
2537 \cs_new_protected_nopar:Npn \char_set_mathcode:nn #1#2 {
2538   \char_set_mathcode:w #1 = \int_eval:w #2\int_eval_end:
2539 }
2540 \cs_new_protected_nopar:Npn \char_gset_mathcode:w { \pref_global:D \tex_mathcode:D }
2541 \cs_new_protected_nopar:Npn \char_gset_mathcode:nn #1#2 {
2542   \char_gset_mathcode:w #1 = \int_eval:w #2\int_eval_end:
2543 }
2544 \cs_new_nopar:Npn \char_value_mathcode:w { \int_use:N \tex_mathcode:D }
2545 \cs_new_nopar:Npn \char_value_mathcode:n #1 {
2546   \char_value_mathcode:w \int_eval:w #1\int_eval_end:
2547 }
2548 \cs_new_nopar:Npn \char_show_value_mathcode:w { \tex_showthe:D \tex_mathcode:D }
2549 \cs_new_nopar:Npn \char_show_value_mathcode:n #1 {
2550   \char_show_value_mathcode:w \int_eval:w #1\int_eval_end:
2551 }

```

`\char_set_lccode:w`  
`\char_set_lccode:nn`  
`\char_value_lccode:w`  
`\char_value_lccode:n`  
`\char_show_value_lccode:w`  
`\char_show_value_lccode:n`

```

2552 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
2553 \cs_new_protected_nopar:Npn \char_set_lccode:nn #1#2{
2554   \char_set_lccode:w #1 = \int_eval:w #2\int_eval_end:
2555 }
2556 \cs_new_nopar:Npn \char_value_lccode:w {\int_use:N\tex_lccode:D}
2557 \cs_new_nopar:Npn \char_value_lccode:n #1{\char_value_lccode:w
2558   \int_eval:w #1\int_eval_end:}
2559 \cs_new_nopar:Npn \char_show_value_lccode:w {\tex_showthe:D\tex_lccode:D}
2560 \cs_new_nopar:Npn \char_show_value_lccode:n #1{
2561   \char_show_value_lccode:w \int_eval:w #1\int_eval_end:}

```

```

\char_set_uccode:w
\char_set_uccode:nn
\char_value_uccode:w
\char_value_uccode:n
\char_show_value_uccode:w
\char_show_value_uccode:n

2562 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
2563 \cs_new_protected_nopar:Npn \char_set_uccode:nn #1#2{
2564   \char_set_uccode:w #1 = \int_eval:w #2\int_eval_end:
2565 }
2566 \cs_new_nopar:Npn \char_value_uccode:w {\int_use:N\tex_uccode:D}
2567 \cs_new_nopar:Npn \char_value_uccode:n #1{\char_value_uccode:w
2568   \int_eval:w #1\int_eval_end:}
2569 \cs_new_nopar:Npn \char_show_value_uccode:w {\tex_showthe:D\tex_uccode:D}
2570 \cs_new_nopar:Npn \char_show_value_uccode:n #1{
2571   \char_show_value_uccode:w \int_eval:w #1\int_eval_end:}

\char_set_sfcode:w
\char_set_sfcode:nn
\char_value_sfcode:w
\char_value_sfcode:n
\char_show_value_sfcode:w
\char_show_value_sfcode:n

2572 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D
2573 \cs_new_protected_nopar:Npn \char_set_sfcode:nn #1#2 {
2574   \char_set_sfcode:w #1 = \int_eval:w #2\int_eval_end:
2575 }
2576 \cs_new_nopar:Npn \char_value_sfcode:w { \int_use:N \tex_sfcode:D }
2577 \cs_new_nopar:Npn \char_value_sfcode:n #1 {
2578   \char_value_sfcode:w \int_eval:w #1\int_eval_end:
2579 }
2580 \cs_new_nopar:Npn \char_show_value_sfcode:w { \tex_showthe:D \tex_sfcode:D }
2581 \cs_new_nopar:Npn \char_show_value_sfcode:n #1 {
2582   \char_show_value_sfcode:w \int_eval:w #1\int_eval_end:
2583 }

```

## 100.4 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}

```

We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```

\c_group_begin_token
\c_group_end_token
\c_math_shift_token
\c_alignment_tab_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_letter_token
\c_other_char_token
\c_active_char_token

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:nn{'\*}{3}
2589 \token_new:Nn \c_math_shift_token {*}
2590 \char_set_catcode:nn{'\*}{4}
2591 \token_new:Nn \c_alignment_tab_token {*}
2592 \token_new:Nn \c_parameter_token {#}
2593 \token_new:Nn \c_math_superscript_token {^}
2594 \char_set_catcode:nn{'\*}{8}
2595 \token_new:Nn \c_math_subscript_token {_}
2596 \token_new:Nn \c_space_token {~}

```

```

2597 \token_new:Nn \c_letter_token {a}
2598 \token_new:Nn \c_other_char_token {1}
2599 \char_set_catcode:nn{'\*}{13}
2600 \cs_gset_nopar:Npn \c_active_char_token {\exp_not:N*}
2601 \group_end:

```

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for  
`\token_if_group_begin:NTF` this.

```

2602 \prg_new_conditional:Nnn \token_if_group_begin:N {p,TF,T,F} {
2603   \if_catcode:w \exp_not:N #1\c_group_begin_token
2604     \prg_return_true: \else: \prg_return_false: \fi:
2605 }

```

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.  
`\token_if_group_end:NTF`

```

2606 \prg_new_conditional:Nnn \token_if_group_end:N {p,TF,T,F} {
2607   \if_catcode:w \exp_not:N #1\c_group_end_token
2608     \prg_return_true: \else: \prg_return_false: \fi:
2609 }

```

`\token_if_math_shift_p:N` Check if token is a math shift token. We use the constant `\c_math_shift_token` for  
`\token_if_math_shift:NTF` this.

```

2610 \prg_new_conditional:Nnn \token_if_math_shift:N {p,TF,T,F} {
2611   \if_catcode:w \exp_not:N #1\c_math_shift_token
2612     \prg_return_true: \else: \prg_return_false: \fi:
2613 }

```

`\token_if_alignment_tab_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token`  
`\token_if_alignment_tab:NTF` for this.

```

2614 \prg_new_conditional:Nnn \token_if_alignment_tab:N {p,TF,T,F} {
2615   \if_catcode:w \exp_not:N #1\c_alignment_tab_token
2616     \prg_return_true: \else: \prg_return_false: \fi:
2617 }

```

`\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 T<sub>E</sub>X a bit to avoid an error message.

```

2618 \prg_new_conditional:Nnn \token_if_parameter:N {p,TF,T,F} {
2619   \exp_after:wN\if_catcode:w \cs:w c_parameter_token\cs_end:\exp_not:N #1
2620     \prg_return_true: \else: \prg_return_false: \fi:
2621 }

```

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token`  
`\token_if_math_superscript:NTF` for this.

```

2622 \prg_new_conditional:Nnn \token_if_math_superscript:N {p,TF,T,F} {
2623   \if_catcode:w \exp_not:N #1\c_math_superscript_token
2624   \prg_return_true: \else: \prg_return_false: \fi:
2625 }

```

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token`  
`\token_if_math_subscript:N $\underline{TF}$`  for this.

```

2626 \prg_new_conditional:Nnn \token_if_math_subscript:N {p,TF,T,F} {
2627   \if_catcode:w \exp_not:N #1\c_math_subscript_token
2628   \prg_return_true: \else: \prg_return_false: \fi:
2629 }

```

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.  
`\token_if_space:N $\underline{TF}$`

```

2630 \prg_new_conditional:Nnn \token_if_space:N {p,TF,T,F} {
2631   \if_catcode:w \exp_not:N #1\c_space_token
2632   \prg_return_true: \else: \prg_return_false: \fi:
2633 }

```

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_letter_token` for this.  
`\token_if_letter:N $\underline{TF}$`

```

2634 \prg_new_conditional:Nnn \token_if_letter:N {p,TF,T,F} {
2635   \if_catcode:w \exp_not:N #1\c_letter_token
2636   \prg_return_true: \else: \prg_return_false: \fi:
2637 }

```

`\token_if_other_char_p:N` Check if token is an other char token. We use the constant `\c_other_char_token` for  
`\token_if_other_char:N $\underline{TF}$`  this.

```

2638 \prg_new_conditional:Nnn \token_if_other_char:N {p,TF,T,F} {
2639   \if_catcode:w \exp_not:N #1\c_other_char_token
2640   \prg_return_true: \else: \prg_return_false: \fi:
2641 }

```

`\token_if_active_char_p:N` Check if token is an active char token. We use the constant `\c_active_char_token` for  
`\token_if_active_char:N $\underline{TF}$`  this.

```

2642 \prg_new_conditional:Nnn \token_if_active_char:N {p,TF,T,F} {
2643   \if_catcode:w \exp_not:N #1\c_active_char_token
2644   \prg_return_true: \else: \prg_return_false: \fi:
2645 }

```

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.  
`\token_if_eq_meaning:NN $\underline{TF}$`

```

2646 \prg_new_conditional:Nnn \token_if_eq_meaning:NN {p,TF,T,F} {
2647   \if_meaning:w #1 #2
2648   \prg_return_true: \else: \prg_return_false: \fi:
2649 }

```

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

`\token_if_eq_catcode:NNTF`

```

2650 \prg_new_conditional:Nnn \token_if_eq_catcode:NN {p,TF,T,F} {
2651   \if_catcode:w \exp_not:N #1 \exp_not:N #2
2652   \prg_return_true: \else: \prg_return_false: \fi:
2653 }

```

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NNTF`

```

2654 \prg_new_conditional:Nnn \token_if_eq_charcode:NN {p,TF,T,F} {
2655   \if_charcode:w \exp_not:N #1 \exp_not:N #2
2656   \prg_return_true: \else: \prg_return_false: \fi:
2657 }

```

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like `\long macro:#1->#1` so we simply check to see if the meaning contains `->`. Argument #2 in the code below will be empty if the string `->` isn't present, proof that the token was not a macro (which is why we reverse the emptiness test). However this function will fail on its own auxiliary function (and a few other private functions as well) but that should certainly never be a problem!

`\token_if_macro:NTF`

`\token_if_macro_p_aux:w`

```

2658 \prg_new_conditional:Nnn \token_if_macro:N {p,TF,T,F} {
2659   \exp_after:wN \token_if_macro_p_aux:w \token_to_meaning:N #1 -> \q_stop
2660 }
2661 \cs_new_nopar:Npn \token_if_macro_p_aux:w #1 -> #2 \q_stop{
2662   \if_predicate:w \tl_if_empty_p:n{#2}
2663   \prg_return_false: \else: \prg_return_true: \fi:
2664 }

```

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. We use `\scan_stop:` for this.

`\token_if_cs:NTF`

```

2665 \prg_new_conditional:Nnn \token_if_cs:N {p,TF,T,F} {
2666   \if_predicate:w \token_if_eq_catcode_p:NN \scan_stop: #1
2667   \prg_return_true: \else: \prg_return_false: \fi:}

```

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  will temporarily convert `\exp_not:N`  $\langle token \rangle$  into `\scan_stop:` if  $\langle token \rangle$  is expandable.

`\token_if_expandable:NTF`

```

2668 \prg_new_conditional:Nnn \token_if_expandable:N {p,TF,T,F} {
2669   \cs_if_exist:NTF #1 {
2670     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2671     \prg_return_false: \else: \prg_return_true: \fi:
2672   } {
2673     \prg_return_false:
2674   }
2675 }

```

```

\token_if_chardef_p:N
\token_if_mathchardef_p:N
\token_if_int_register_p:N
\token_if_skip_register_p:N
\token_if_dim_register_p:N
\token_if_toks_register_p:N
\token_if_protected_macro_p:N
\token_if_long_macro_p:N
\token_if_protected_long_macro_p:N
\token_if_chardef:NTF
\token_if_mathchardef:NTF
\token_if_long_macro:NTF
\token_if_protected_macro:NTF
\token_if_protected_long_macro:NTF
\token_if_dim_register:NTF
\token_if_skip_register:NTF
\token_if_int_register:NTF
\token_if_toks_register:NTF
\token_if_chardef_p_aux:w
\token_if_mathchardef_p_aux:w
\token_if_int_register_p_aux:w
\token_if_skip_register_p_aux:w
\token_if_dim_register_p_aux:w
\token_if_toks_register_p_aux:w
\token_if_protected_macro_p_aux:w
\token_if_long_macro_p_aux:w
\token_if_protected_long_macro_p_aux:w

```

Most of these functions have to check the meaning of the token in question so we need to do some checkups on which characters are output by `\token_to_meaning:N`. As usual, these characters have catcode 12 so we must do some serious substitutions in the code below...

```

2676 \group_begin:
2677   \char_set_lccode:nn {'T}{\T}
2678   \char_set_lccode:nn {'F}{\F}
2679   \char_set_lccode:nn {'X}{\X}
2680   \char_set_lccode:nn {'Y}{\Y}
2681   \char_set_lccode:nn {'Z}{\Z}
2682   \char_set_lccode:nn {'?'}{\?}
2683   \tl_map_inline:nn{\X\Y\Z\M\C\H\A\R\O\U\S\K\I\P\L\G\P\E}
2684     {\char_set_catcode:nn {#1}{12}}

```

We convert the token list to lowercase and restore the catcode and lowercase code changes.

```

2685 \tl_to_lowercase:n{
2686   \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>`.

```

2687 \prg_new_conditional:Nnn \token_if_chardef:N {p,TF,T,F} {
2688   \exp_after:wN \token_if_chardef_aux:w
2689   \token_to_meaning:N #1?CHAR"\q_stop
2690 }
2691 \cs_new_nopar:Npn \token_if_chardef_aux:w #1?CHAR"#2\q_stop{
2692   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2693 }

2694 \prg_new_conditional:Nnn \token_if_mathchardef:N {p,TF,T,F} {
2695   \exp_after:wN \token_if_mathchardef_aux:w
2696   \token_to_meaning:N #1?MAYHCHAR"\q_stop
2697 }
2698 \cs_new_nopar:Npn \token_if_mathchardef_aux:w #1?MAYHCHAR"#2\q_stop{
2699   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2700 }

```

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.

```

2701 \prg_new_conditional:Nnn \token_if_int_register:N {p,TF,T,F} {
2702   \if_meaning:w \tex_countdef:D #1
2703   \prg_return_false:
2704 }
2705 \exp_after:wN \token_if_int_register_aux:w
2706   \token_to_meaning:N #1?COUXY\q_stop
2707 \fi:

```

```

2708 }
2709 \cs_new_nopar:Npn \token_if_int_register_aux:w #1?COUXY#2\q_stop{
2710   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2711 }

```

Skip registers are done the same way as the integer registers.

```

2712 \prg_new_conditional:Nnn \token_if_skip_register:N {p,TF,T,F} {
2713   \if_meaning:w \tex_skipdef:D #1
2714   \prg_return_false:
2715   \else:
2716     \exp_after:wN \token_if_skip_register_aux:w
2717     \token_to_meaning:N #1?SKIP\q_stop
2718   \fi:
2719 }
2720 \cs_new_nopar:Npn \token_if_skip_register_aux:w #1?SKIP#2\q_stop{
2721   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2722 }

```

Dim registers. No news here

```

2723 \prg_new_conditional:Nnn \token_if_dim_register:N {p,TF,T,F} {
2724   \if_meaning:w \tex_dimendef:D #1
2725   \c_false_bool
2726   \else:
2727     \exp_after:wN \token_if_dim_register_aux:w
2728     \token_to_meaning:N #1?ZIMEX\q_stop
2729   \fi:
2730 }
2731 \cs_new_nopar:Npn \token_if_dim_register_aux:w #1?ZIMEX#2\q_stop{
2732   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2733 }

```

Toks registers.

```

2734 \prg_new_conditional:Nnn \token_if_toks_register:N {p,TF,T,F} {
2735   \if_meaning:w \tex_toksdef:D #1
2736   \prg_return_false:
2737   \else:
2738     \exp_after:wN \token_if_toks_register_aux:w
2739     \token_to_meaning:N #1?YOKS\q_stop
2740   \fi:
2741 }
2742 \cs_new_nopar:Npn \token_if_toks_register_aux:w #1?YOKS#2\q_stop{
2743   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2744 }

```

Protected macros.

```

2745 \prg_new_conditional:Nnn \token_if_protected_macro:N {p,TF,T,F} {
2746   \exp_after:wN \token_if_protected_macro_aux:w

```

```

2747 \token_to_meaning:N #1?PROYECY EZ~MACRO\q_stop
2748 }
2749 \cs_new_nopar:Npn \token_if_protected_macro_aux:w #1?PROYECY EZ~MACRO#2\q_stop{
2750 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2751 }

```

Long macros.

```

2752 \prg_new_conditional:Nnn \token_if_long_macro:N {p,TF,T,F} {
2753 \exp_after:wN \token_if_long_macro_aux:w
2754 \token_to_meaning:N #1?LOXG~MACRO\q_stop
2755 }
2756 \cs_new_nopar:Npn \token_if_long_macro_aux:w #1?LOXG~MACRO#2\q_stop{
2757 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2758 }

```

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.

```

2759 \prg_new_conditional:Nnn \token_if_protected_long_macro:N {p,TF,T,F} {
2760 \exp_after:wN \token_if_protected_long_macro_aux:w
2761 \token_to_meaning:N #1?PROYECY EZ?LOXG~MACRO\q_stop
2762 }
2763 \cs_new_nopar:Npn \token_if_protected_long_macro_aux:w #1
2764 ?PROYECY EZ?LOXG~MACRO#2\q_stop{
2765 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2766 }

```

Finally the `\tl_to_lowercase:n` ends!

```

2767 }

```

We do not provide a function for testing if a control sequence is “outer” since we don't use that in L<sup>A</sup>T<sub>E</sub>X3.

```

\et_prefix_arg_replacement_aux:w
\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N

```

In the `xparse` package we sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```

2768 \group_begin:
2769 \char_set_lccode:nn {'\?}{\:}
2770 \char_set_catcode:nn{'M}{12}
2771 \char_set_catcode:nn{'A}{12}
2772 \char_set_catcode:nn{'C}{12}
2773 \char_set_catcode:nn{'R}{12}
2774 \char_set_catcode:nn{'\0}{12}
2775 \tl_to_lowercase:n{

```

```

2776 \group_end:
2777 \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:w #1MACRO?#2->#3\q_stop#4{
2778   #4{#1}{#2}{#3}
2779 }
2780 \cs_new_nopar:Npn \token_get_prefix_spec:N #1{
2781   \token_if_macro:NTF #1{
2782     \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2783     \token_to_meaning:N #1\q_stop\use_i:nnn
2784   }{\scan_stop:}
2785 }
2786 \cs_new_nopar:Npn \token_get_arg_spec:N #1{
2787   \token_if_macro:NTF #1{
2788     \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2789     \token_to_meaning:N #1\q_stop\use_ii:nnn
2790   }{\scan_stop:}
2791 }
2792 \cs_new_nopar:Npn \token_get_replacement_spec:N #1{
2793   \token_if_macro:NTF #1{
2794     \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2795     \token_to_meaning:N #1\q_stop\use_iii:nnn
2796   }{\scan_stop:}
2797 }
2798 }

```

Useless code: because we can!

<pre> \token_if_primitive_p:N \token_if_primitive_p_aux:N \token_if_primitive:NTF </pre>	<p>It is rather hard to determine if a token is a primitive. First we can check if it is a control sequence or active character. If either, we check if it is a macro. Then we can go through a tedious process of testing for different register types... I don't actually think this function is useful but you never know.</p>
--	---

```

2799 \prg_new_conditional:Nnn \token_if_primitive:N {p,TF,T,F} {
2800   \if_predicate:w \token_if_cs_p:N #1
2801     \if_predicate:w \token_if_macro_p:N #1
2802       \prg_return_false:
2803     \else:
2804       \token_if_primitive_p_aux:N #1
2805     \fi:
2806   \else:
2807     \if_predicate:w \token_if_active_char_p:N #1
2808       \if_predicate:w \token_if_macro_p:N #1
2809         \prg_return_false:
2810       \else:
2811         \token_if_primitive_p_aux:N #1
2812       \fi:
2813     \else:
2814       \prg_return_false:
2815     \fi:
2816   \fi:

```

```

2817 }
2818 \cs_new_nopar:Npn \token_if_primitive_p_aux:N #1{
2819   \if_predicate:w \token_if_chardef_p:N #1 \c_false_bool
2820   \else:
2821     \if_predicate:w \token_if_mathchardef_p:N #1 \prg_return_false:
2822     \else:
2823       \if_predicate:w \token_if_int_register_p:N #1 \prg_return_false:
2824       \else:
2825         \if_predicate:w \token_if_skip_register_p:N #1 \prg_return_false:
2826         \else:
2827           \if_predicate:w \token_if_dim_register_p:N #1 \prg_return_false:
2828           \else:
2829             \if_predicate:w \token_if_toks_register_p:N #1 \prg_return_false:
2830             \else:

```

We made it!

```

2831       \prg_return_true:
2832     \fi:
2833   \fi:
2834 \fi:
2835 \fi:
2836 \fi:
2837 \fi:
2838 }

```

## 100.5 Peeking ahead at the next token

\l\_peek\_token  
\g\_peek\_token  
\l\_peek\_search\_token

We define some other tokens which will initially be the character ?.

```

2839 \token_new:Nn \l_peek_token {?}
2840 \token_new:Nn \g_peek_token {?}
2841 \token_new:Nn \l_peek_search_token {?}

```

\peek\_after:NN  
\peek\_gafter:NN

\peek\_after:NN takes two argument where the first is a function acting on \l\_peek\_token and the second is the next token in the input stream which \l\_peek\_token is set equal to. \peek\_gafter:NN does the same globally to \g\_peek\_token.

```

2842 \cs_new_protected_nopar:Npn \peek_after:NN {\tex_futurelet:D \l_peek_token }
2843 \cs_new_protected_nopar:Npn \peek_gafter:NN {
2844   \pref_global:D \tex_futurelet:D \g_peek_token
2845 }

```

For normal purposes there are four main cases:

1. peek at the next token.
2. peek at the next non-space token.

3. peek at the next token and remove it.
4. peek at the next non-space token and remove it.

The generic functions will take four arguments: The token to search for, the test function to run on it and the true/false cases. The general algorithm is this:

1. Store the token to search for in `\l_peek_search_token`.
2. In order to avoid doubling of hash marks where it seems unnatural we put the `<true>` and `<false>` cases through an `x` type expansion but using `\exp_not:n` to avoid any expansion. This has the same effect as putting it through a `<toks>` register but is faster. Also put in a special alignment safe group end.
3. Put in an alignment safe group begin.
4. Peek ahead and call the function which will act on the next token in the input stream.

`\l_peek_true_tl` Two dedicated token list variables that store the true and false cases.  
`\l_peek_false_tl`

```
2846 \tl_new:N \l_peek_true_tl
2847 \tl_new:N \l_peek_false_tl
```

`\peek_tmp:w` Scratch function used for storing the token to be removed if found.

```
2848 \cs_new_nopar:Npn \peek_tmp:w {}
```

`\l_peek_search_tl` We also use this token list variable for storing the token we want to compare. This turns out to be useful.

```
2849 \tl_new:N \l_peek_search_tl
```

`\peek_token_generic:NNTF` #1 : the function to execute (obey or ignore spaces, etc.),  
#2 : the special token we're looking for.

```
2850 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4 {
2851   \cs_set_eq:NN \l_peek_search_token #2
2852   \tl_set:Nn \l_peek_search_tl {#2}
2853   \tl_set:Nn \l_peek_true_tl { \group_align_safe_end: #3 }
2854   \tl_set:Nn \l_peek_false_tl { \group_align_safe_end: #4 }
2855   \group_align_safe_begin:
2856   \peek_after:NN #1
2857 }
2858 \cs_new_protected:Npn \peek_token_generic:NNT #1#2#3 {
2859   \peek_token_generic:NNTF #1#2 {#3} {}
2860 }
2861 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3 {
2862   \peek_token_generic:NNTF #1#2 {} {#3}
2863 }
```

`\peek_token_remove_generic:NNTF` If we want to be able to remove any character from the input stream we might as well do it the same way for all characters so we define this as little differently from above.

```

2864 \cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4 {
2865   \cs_set_eq:NN \l_peek_search_token #2
2866   \tl_set:Nn \l_peek_search_tl {#2}
2867   \tl_set:Nn \l_peek_true_aux_tl {#3}
2868   \tl_set_eq:NN \l_peek_true_tl \c_peek_true_remove_next_tl
2869   \tl_set:Nn \l_peek_false_tl {\group_align_safe_end: #4}
2870   \group_align_safe_begin:
2871     \peek_after:NN #1
2872   }
2873 \cs_new:Npn \peek_token_remove_generic:NNT #1#2#3 {
2874   \peek_token_remove_generic:NNTF #1#2 {#3} {}
2875 }
2876 \cs_new:Npn \peek_token_remove_generic:NNF #1#2#3 {
2877   \peek_token_remove_generic:NNTF #1#2 {} {#3}
2878 }

```

`\l_peek_true_aux_tl` `\c_peek_true_remove_next_tl` Two token list variables to help with removing the character from the input stream.

```

2879 \tl_new:N \l_peek_true_aux_tl
2880 \tl_const:Nn \c_peek_true_remove_next_tl {\group_align_safe_end:
2881   \tex_afterassignment:D \l_peek_true_aux_tl \cs_set_eq:NN \peek_tmp:w
2882 }

```

`\peek_execute_branches_meaning:` `\peek_execute_branches_catcode:` `\peek_execute_branches_charcode:` There are three major tests between tokens in T<sub>E</sub>X: meaning, catcode and charcode. Hence we define three basic test functions that set in after the ignoring phase is over and done with.

`\peek_execute_branches_charcode_aux:NN`

```

2883 \cs_new_nopar:Npn \peek_execute_branches_meaning: {
2884   \if_meaning:w \l_peek_token \l_peek_search_token
2885     \exp_after:wN \l_peek_true_tl
2886   \else:
2887     \exp_after:wN \l_peek_false_tl
2888   \fi:
2889 }
2890 \cs_new_nopar:Npn \peek_execute_branches_catcode: {
2891   \if_catcode:w \exp_not:N\l_peek_token \exp_not:N\l_peek_search_token
2892     \exp_after:wN \l_peek_true_tl
2893   \else:
2894     \exp_after:wN \l_peek_false_tl
2895   \fi:
2896 }

```

For the charcode version we do things a little differently. We want to check the token directly but if we do this we face problems if the next thing in the input stream is a braced group or a space token. The braced group would be read as a complete argument and the space would be gobbled by T<sub>E</sub>X's argument reading routines. Hence we test for both

of these and if one of them is found we just execute the false result directly since no one should ever try to use the `charcode` function for searching for `\c_group_begin_token` or `\c_space_token`. The same is true for `\c_group_end_token`, as this can only occur if the function is at the end of a group.

```

2897 \cs_new_nopar:Npn \peek_execute_branches_charcode: {
2898   \bool_if:nTF {
2899     \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
2900     \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token ||
2901     \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
2902   }
2903   { \l_peek_false_tl }

```

Otherwise we call a small auxiliary function that just grabs the next token. We can do that because it really is a single token; we just have insert it again afterwards. Also we stored the token we were looking for in the token list variable `\l_peek_search_tl` so we unpack it again for this function.

```

2904   { \exp_after:wN \peek_execute_branches_charcode_aux:NN \l_peek_search_tl }
2905 }

```

Then we just do the usual `\if_charcode:w` comparison. We also remember to insert `#2` again after executing the true or false branches.

```

2906 \cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2{
2907   \if_charcode:w \exp_not:N #1\exp_not:N#2
2908     \exp_after:wN \l_peek_true_tl
2909   \else:
2910     \exp_after:wN \l_peek_false_tl
2911   \fi:
2912   #2
2913 }

```

`\peek_def_aux:nnnn`  
`\peek_def_aux_ii:nnnnn`

This function aids defining conditional variants without too much repeated code. I hope that it doesn't detract too much from the readability.

```

2914 \cs_new_nopar:Npn \peek_def_aux:nnnn #1#2#3#4 {
2915   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { TF }
2916   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { T }
2917   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { F }
2918 }
2919 \cs_new_protected_nopar:Npn \peek_def_aux_ii:nnnnn #1#2#3#4#5 {
2920   \cs_new_nopar:cpx { #1 #5 } {
2921     \tl_if_empty:nF {#2} {
2922       \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 }
2923     }
2924     \exp_not:c { #3 #5 }
2925     \exp_not:n { #4 }
2926   }
2927 }

```

`\peek_meaning:N $\underline{T}$ F` Here we use meaning comparison with `\if_meaning:w`.

```
2928 \peek_def_aux:nnnn
2929 { peek_meaning:N }
2930 {}
2931 { peek_token_generic:NN }
2932 { \peek_execute_branches_meaning: }
```

`\peek_meaning_ignore_spaces:N $\underline{T}$ F`

```
2933 \peek_def_aux:nnnn
2934 { peek_meaning_ignore_spaces:N }
2935 { \peek_execute_branches_meaning: }
2936 { peek_token_generic:NN }
2937 { \peek_ignore_spaces_execute_branches: }
```

`\peek_meaning_remove:N $\underline{T}$ F`

```
2938 \peek_def_aux:nnnn
2939 { peek_meaning_remove:N }
2940 {}
2941 { peek_token_remove_generic:NN }
2942 { \peek_execute_branches_meaning: }
```

`\peek_meaning_remove_ignore_spaces:N $\underline{T}$ F`

```
2943 \peek_def_aux:nnnn
2944 { peek_meaning_remove_ignore_spaces:N }
2945 { \peek_execute_branches_meaning: }
2946 { peek_token_remove_generic:NN }
2947 { \peek_ignore_spaces_execute_branches: }
```

`\peek_catcode:N $\underline{T}$ F` Here we use catcode comparison with `\if_catcode:w`.

```
2948 \peek_def_aux:nnnn
2949 { peek_catcode:N }
2950 {}
2951 { peek_token_generic:NN }
2952 { \peek_execute_branches_catcode: }
```

`\peek_catcode_ignore_spaces:N $\underline{T}$ F`

```
2953 \peek_def_aux:nnnn
2954 { peek_catcode_ignore_spaces:N }
2955 { \peek_execute_branches_catcode: }
2956 { peek_token_generic:NN }
2957 { \peek_ignore_spaces_execute_branches: }
```

`\peek_catcode_remove:N $\underline{TF}$`

```
2958 \peek_def_aux:nnnn
2959 { peek_catcode_remove:N }
2960 {}
2961 { peek_token_remove_generic:NN }
2962 { \peek_execute_branches_catcode: }
```

`atcode_remove_ignore_spaces:N $\underline{TF}$`

```
2963 \peek_def_aux:nnnn
2964 { peek_catcode_remove_ignore_spaces:N }
2965 { \peek_execute_branches_catcode: }
2966 { peek_token_remove_generic:NN }
2967 { \peek_ignore_spaces_execute_branches: }
```

`\peek_charcode:N $\underline{TF}$`  Here we use charcode comparison with `\if_charcode:w`.

```
2968 \peek_def_aux:nnnn
2969 { peek_charcode:N }
2970 {}
2971 { peek_token_generic:NN }
2972 { \peek_execute_branches_charcode: }
```

`peek_charcode_ignore_spaces:N $\underline{TF}$`

```
2973 \peek_def_aux:nnnn
2974 { peek_charcode_ignore_spaces:N }
2975 { \peek_execute_branches_charcode: }
2976 { peek_token_generic:NN }
2977 { \peek_ignore_spaces_execute_branches: }
```

`\peek_charcode_remove:N $\underline{TF}$`

```
2978 \peek_def_aux:nnnn
2979 { peek_charcode_remove:N }
2980 {}
2981 { peek_token_remove_generic:NN }
2982 { \peek_execute_branches_charcode: }
```

`arcode_remove_ignore_spaces:N $\underline{TF}$`

```
2983 \peek_def_aux:nnnn
2984 { peek_charcode_remove_ignore_spaces:N }
2985 { \peek_execute_branches_charcode: }
2986 { peek_token_remove_generic:NN }
2987 { \peek_ignore_spaces_execute_branches: }
```

`\peek_ignore_spaces_aux:` Throw away a space token and search again. We could define this in a more devious way where the auxiliary function gobbles the space token but then what do we do if we decide that a certain function should ignore more than one specific token? For example someone might find it interesting to define a `\peek_` function that ignores `a`'s and `b`'s! Or maybe different kinds of “funny spaces”... Therefore I have decided to use this version which uses `\tex_afterassignment:D` to call the auxiliary function after the next token has been removed by `\cs_set_eq:NN`. That way it is easily extensible.

```

2988 \cs_new_nopar:Npn \peek_ignore_spaces_aux: {
2989   \peek_after:NN \peek_ignore_spaces_execute_branches:
2990 }
2991 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches: {
2992   \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
2993   { \tex_afterassignment:D \peek_ignore_spaces_aux:
2994     \cs_set_eq:NN \peek_tmp:w
2995   }
2996   \peek_execute_branches:
2997 }

2998 </initex | package>

2999 <*showmemory>
3000 \showMemUsage
3001 </showmemory>

```

## 101 l3int implementation

### 101.1 Internal functions and variables

`\int_advance:w` `\int_advance:w` *<int register>* *<optional ‘by’>* *<number>* *<space>*  
 Increments the count register by the specified amount.

**T<sub>E</sub>Xhackers note:** This is T<sub>E</sub>X’s `\advance`.

`\int_convert_number_to_letter:n *` `\int_convert_number_to_letter:n` *{<integer expression>}*

Internal function for turning a number for a different base into a letter or digit.

<code>\int_pre_eval_one_arg:Nn</code> <code>\int_pre_eval_two_args:Nnn</code>	<code>\int_pre_eval_one_arg:Nn</code> <i>&lt;function&gt;</i> <i>{&lt;integer expression&gt;}</i> <code>\int_pre_eval_one_arg:Nnn</code> <i>&lt;function&gt;</i> <i>{&lt;int expr<sub>1</sub>&gt;}</i> <i>{&lt;int expr<sub>2</sub>&gt;}</i>
--	--

These are expansion helpers; they evaluate their integer expressions before handing them off to the specified *function*.

<code>\int_get_sign_and_digits:n *</code> <code>\int_get_sign:n *</code> <code>\int_get_digits:n *</code>	<code>\int_get_sign_and_digits:n {\number}</code>
---	---

From an argument that may or may not include a + or - sign, these functions expand to the respective components of the number.

## 101.2 Module loading and primitives definitions

We start by ensuring that the required packages are loaded.

```

3002 <*package>
3003 \ProvidesExplPackage
3004   {\filename}{\filedate}{\fileversion}{\filedescription}
3005 \package_check_loaded_expl:
3006 </package>
3007 <*initex | package>

```

<code>\int_value:w</code> <code>\int_eval:n</code> <code>\int_eval:w</code> <code>\int_eval_end:</code> <code>\if_int_compare:w</code> <code>\if_int_odd:w</code> <code>\if_num:w</code> <code>\if_case:w</code> <code>\int_to_roman:w</code> <code>\int_advance:w</code>	Here are the remaining primitives for number comparisons and expressions.	<pre> 3008 \cs_set_eq:NN \int_value:w \tex_number:D 3009 \cs_set_eq:NN \int_eval:w \etex_numexpr:D 3010 \cs_set_protected:Npn \int_eval_end: {\tex_relax:D} 3011 \cs_set_eq:NN \if_int_compare:w \tex_ifnum:D 3012 \cs_new_eq:NN \if_num:w \tex_ifnum:D 3013 \cs_set_eq:NN \if_int_odd:w \tex_ifodd:D 3014 \cs_new_eq:NN \if_case:w \tex_ifcase:D 3015 \cs_new_eq:NN \int_to_roman:w \tex_romannumeral:D 3016 \cs_new_eq:NN \int_advance:w \tex_advance:D </pre>
--	---	--

`\int_eval:n` Wrapper for `\int_eval:w`. Can be used in an integer expression or directly in the input stream.

```

3017 \cs_set:Npn \int_eval:n #1{
3018   \int_value:w \int_eval:w #1\int_eval_end:
3019 }

```

## 101.3 Allocation and setting

`\int_new:N` For the L<sup>A</sup>T<sub>E</sub>X3 format:  
`\int_new:c`

```

3020 <*initex>
3021 \alloc_new:nnnN {int} {11} {\c_max_register_int} \tex_countdef:D
3022 </initex>

```

For ‘l3in2e’:

```

3023 <*package>
3024 \cs_new_protected_nopar:Npn \int_new:N #1 {
3025   \chk_if_free_cs:N #1
3026   \newcount #1
3027 }
3028 </package>

3029 \cs_generate_variant:Nn \int_new:N {c}

```

\int\_set:Nn Setting counters is again something that I would like to make uniform at the moment to  
 \int\_set:cn get a better overview.

```

\int_gset:Nn
\int_gset:cn
3030 \cs_new_protected_nopar:Npn \int_set:Nn #1#2{#1 \int_eval:w #2\int_eval_end:
3031 <*check>
3032 \chk_local_or_pref_global:N #1
3033 </check>
3034 }
3035 \cs_new_protected_nopar:Npn \int_gset:Nn {
3036 <*check>
3037   \pref_global_chk:
3038 </check>
3039 <-check> \pref_global:D
3040   \int_set:Nn }
3041 \cs_generate_variant:Nn \int_set:Nn {cn}
3042 \cs_generate_variant:Nn \int_gset:Nn {cn}

```

\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
3043 \cs_new_protected_nopar:Npn \int_set_eq:NN #1#2 {
3044   \int_set:Nn #1 {#2}
3045 }
3046 \cs_generate_variant:Nn \int_set_eq:NN { c }
3047 \cs_generate_variant:Nn \int_set_eq:NN { Nc }
3048 \cs_generate_variant:Nn \int_set_eq:NN { cc }
3049 \cs_new_protected_nopar:Npn \int_gset_eq:NN #1#2 {
3050   \int_gset:Nn #1 {#2}
3051 }
3052 \cs_generate_variant:Nn \int_gset_eq:NN { c }
3053 \cs_generate_variant:Nn \int_gset_eq:NN { Nc }
3054 \cs_generate_variant:Nn \int_gset_eq:NN { cc }

```

\int\_incr:N Incrementing and decrementing of integer registers is done with the following functions.

```

\int_decr:N
\int_gincr:N
\int_gdecr:N
\int_incr:c
\int_decr:c
\int_gincr:c
\int_gdecr:c
3055 \cs_new_protected_nopar:Npn \int_incr:N #1{\int_advance:w#1\c_one
3056 <*check>
3057   \chk_local_or_pref_global:N #1
3058 </check>

```

```

3059 }
3060 \cs_new_protected_nopar:Npn \int_decr:N #1{\int_advance:w#1\c_minus_one
3061 <*check>
3062     \chk_local_or_pref_global:N #1
3063 </check>
3064 }
3065 \cs_new_protected_nopar:Npn \int_gincr:N {

```

We make sure that a local variable is not updated globally by changing the internal test (i.e. `\chk_local_or_pref_global:N`) before making the assignment. This is done by `\pref_global_chk:` which also issues the necessary `\pref_global:D`. This is not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```

3066 <*check>
3067     \pref_global_chk:
3068 </check>
3069 <-check> \pref_global:D
3070     \int_incr:N}
3071 \cs_new_protected_nopar:Npn \int_gdecr:N {
3072 <*check>
3073     \pref_global_chk:
3074 </check>
3075 <-check> \pref_global:D
3076     \int_decr:N}

```

With the `\int_add:Nn` functions we can shorten the above code. If this makes it too slow ...

```

3077 \cs_set_protected_nopar:Npn \int_incr:N #1{\int_add:Nn#1\c_one}
3078 \cs_set_protected_nopar:Npn \int_decr:N #1{\int_add:Nn#1\c_minus_one}
3079 \cs_set_protected_nopar:Npn \int_gincr:N #1{\int_gadd:Nn#1\c_one}
3080 \cs_set_protected_nopar:Npn \int_gdecr:N #1{\int_gadd:Nn#1\c_minus_one}

3081 \cs_generate_variant:Nn \int_incr:N {c}
3082 \cs_generate_variant:Nn \int_decr:N {c}
3083 \cs_generate_variant:Nn \int_gincr:N {c}
3084 \cs_generate_variant:Nn \int_gdecr:N {c}

```

`\int_zero:N` Functions that reset an `<int>` register to zero.

```

\int_zero:c
\int_gzero:N
\int_gzero:c
3085 \cs_new_protected_nopar:Npn \int_zero:N #1 {#1=\c_zero}
3086 \cs_generate_variant:Nn \int_zero:N {c}

3087 \cs_new_protected_nopar:Npn \int_gzero:N #1 {\pref_global:D #1=\c_zero}
3088 \cs_generate_variant:Nn \int_gzero:N {c}

```

`\int_add:Nn` Adding and subtracting to and from a counter ... We should think of using these functions

```

\int_add:cn
\int_gadd:Nn
\int_gadd:cn
3089 \cs_new_protected_nopar:Npn \int_add:Nn #1#2{
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

```

We need to say by in case the first argument is a register accessed by its number, e.g., `\count23`. Not that it should ever happen but...

```

3090     \int_advance:w #1 by \int_eval:w #2\int_eval_end:
3091 <*check>
3092     \chk_local_or_pref_global:N #1
3093 </check>
3094 }
3095 \cs_new_nopar:Npn \int_sub:Nn #1#2{
3096     \int_advance:w #1-\int_eval:w #2\int_eval_end:
3097 <*check>
3098     \chk_local_or_pref_global:N #1
3099 </check>
3100 }
3101 \cs_new_protected_nopar:Npn \int_gadd:Nn {
3102 <*check>
3103     \pref_global_chk:
3104 </check>
3105 <-check> \pref_global:D
3106     \int_add:Nn }
3107 \cs_new_protected_nopar:Npn \int_gsub:Nn {
3108 <*check>
3109     \pref_global_chk:
3110 </check>
3111 <-check> \pref_global:D
3112     \int_sub:Nn }
3113 \cs_generate_variant:Nn \int_add:Nn {cn}
3114 \cs_generate_variant:Nn \int_gadd:Nn {cn}
3115 \cs_generate_variant:Nn \int_sub:Nn {cn}
3116 \cs_generate_variant:Nn \int_gsub:Nn {cn}

```

`\int_use:N` Here is how counters are accessed:

```

\int_use:c
3117 \cs_new_eq:NN \int_use:N \tex_the:D
3118 \cs_new_nopar:Npn \int_use:c #1{\int_use:N \cs:w#1\cs_end:}

```

`\int_show:N`

```

\int_show:c
3119 \cs_new_eq:NN \int_show:N \kernel_register_show:N
3120 \cs_new_eq:NN \int_show:c \kernel_register_show:c

```

`\int_to_arabic:n` Nothing exciting here.

```

3121 \cs_new_nopar:Npn \int_to_arabic:n #1{ \int_eval:n{#1}}

```

`\int_roman_lcuc_mapping:Nnn`

Using T<sub>E</sub>X's built-in feature for producing roman numerals has some surprising features. One is the the characters resulting from `\int_to_roman:w` have category code 12 so they may fail in certain comparison tests. Therefore we use a mapping from the character

TeX produces to the character we actually want which will give us letters with category code 11.

```

3122 \cs_new_protected_nopar:Npn \int_roman_lcuc_mapping:Nnn #1#2#3{
3123   \cs_set_nopar:cpn {int_to_lc_roman_#1:}{#2}
3124   \cs_set_nopar:cpn {int_to_uc_roman_#1:}{#3}
3125 }

```

Here are the default mappings. I haven't found any examples of say Turkish doing the mapping `i \i` `I` but at least there is a possibility for it if needed. Note: I have now asked a Turkish person and he tells me they do the `i I` mapping.

```

3126 \int_roman_lcuc_mapping:Nnn i i I
3127 \int_roman_lcuc_mapping:Nnn v v V
3128 \int_roman_lcuc_mapping:Nnn x x X
3129 \int_roman_lcuc_mapping:Nnn l l L
3130 \int_roman_lcuc_mapping:Nnn c c C
3131 \int_roman_lcuc_mapping:Nnn d d D
3132 \int_roman_lcuc_mapping:Nnn m m M

```

For the delimiter we cheat and let it gobble its arguments instead.

```

3133 \int_roman_lcuc_mapping:Nnn Q \use_none:nn \use_none:nn

```

`\int_to_roman:n` The commands for producing the lower and upper case roman numerals run a loop on one character at a time and also carries some information for upper or lower case with `\int_to_Roman:n` it. We put it through `\int_eval:n` first which is safer and more flexible.

```

3134 \cs_new_nopar:Npn \int_to_roman:n #1 {
3135   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN l
3136   \int_to_roman:w \int_eval:n {#1} Q
3137 }
3138 \cs_new_nopar:Npn \int_to_Roman:n #1 {
3139   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN u
3140   \int_to_roman:w \int_eval:n {#1} Q
3141 }
3142 \cs_new_nopar:Npn \int_to_roman_lcuc:NN #1#2{
3143   \use:c {int_to_#1c_roman_#2:}
3144   \int_to_roman_lcuc:NN #1
3145 }

```

`\int_convert_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (`#1`) is compared to the total number of symbols available at each place (`#2`). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an `f`-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy. This is more or less the same as `\int_convert_number_with_rule:nnN` but 'pre-packaged'.

```

3146 \cs_new_nopar:Npn \int_convert_to_symbols:nnn #1#2#3 {
3147   \int_compare:nNnTF {#1} > {#2}
3148   {
3149     \exp_args:Nf \int_convert_to_symbols:nnn
3150     { \int_div_truncate:nn { #1 - 1 } {#2} } {#2} {#3}
3151     \exp_args:Nf \prg_case_int:nnn
3152     { \int_eval:n { 1 + \int_mod:nn { #1 - 1 } {#2} } }
3153     {#3} { }
3154   }
3155   { \exp_args:Nf \prg_case_int:nnn { \int_eval:n {#1} } {#3} { } }
3156 }

```

`\int_convert_number_with_rule:nnN` This is our major workhorse for conversions. #1 is the number we want converted, #2 is the base number, and #3 is the function converting the number. This function expects to receive a non-negative integer and as such is ideal for something using `\if_case:w` internally.

The basic example is this: We want to convert the number 50 (#1) into an alphabetic equivalent **ax**. For the English language our list contains 26 elements so this is our argument #2 while the function #3 just turns 1 into a, 2 into b, etc. Hence our goal is to turn 50 into the sequence #3{1}#1{24} so what we do is to first divide 50 by 26 and truncating the result returning 1. Then before we execute this we call the function again but this time on the result of the remainder of the division. This goes on until the remainder is less than or equal to the base number where we just call the function #3 directly on the number.

We do a little pre-expansion of the arguments below as they otherwise have a tendency to grow quite large.

```

3157 \cs_set_nopar:Npn \int_convert_number_with_rule:nnN #1#2#3{
3158   \int_compare:nNnTF {#1}>{#2}
3159   {
3160     \exp_args:Nf \int_convert_number_with_rule:nnN
3161     { \int_div_truncate:nn {#1-1}{#2} }{#2}
3162     #3

```

Note that we have to nudge our modulus function so it won't return 0 as that wouldn't work with `\if_case:w` when that expects a positive number to produce a letter.

```

3163     \exp_args:Nf #3 { \int_eval:n{1+\int_mod:nn {#1-1}{#2}} }
3164   }
3165   { \exp_args:Nf #3{ \int_eval:n{#1} } }
3166 }

```

As can be seen it is even simpler to convert to number systems that contain 0, since then we don't have to add or subtract 1 here and there.

`\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.

```

3167 \cs_new_nopar:Npn \int_to_alph:n #1 {
3168   \int_convert_to_symbols:nnn {#1} { 26 }
3169   {
3170     { 1 } { a }
3171     { 2 } { b }
3172     { 3 } { c }
3173     { 4 } { d }
3174     { 5 } { e }
3175     { 6 } { f }
3176     { 7 } { g }
3177     { 8 } { h }
3178     { 9 } { i }
3179     { 10 } { j }
3180     { 11 } { k }
3181     { 12 } { l }
3182     { 13 } { m }
3183     { 14 } { n }
3184     { 15 } { o }
3185     { 16 } { p }
3186     { 17 } { q }
3187     { 18 } { r }
3188     { 19 } { s }
3189     { 20 } { t }
3190     { 21 } { u }
3191     { 22 } { v }
3192     { 23 } { w }
3193     { 24 } { x }
3194     { 25 } { y }
3195     { 26 } { z }
3196   }
3197 }
3198 \cs_new_nopar:Npn \int_to_Alph:n #1 {
3199   \int_convert_to_symbols:nnn {#1} { 26 }
3200   {
3201     { 1 } { A }
3202     { 2 } { B }
3203     { 3 } { C }
3204     { 4 } { D }
3205     { 5 } { E }
3206     { 6 } { F }
3207     { 7 } { G }
3208     { 8 } { H }
3209     { 9 } { I }
3210     { 10 } { J }
3211     { 11 } { K }
3212     { 12 } { L }
3213     { 13 } { M }
3214     { 14 } { N }
3215     { 15 } { O }
3216     { 16 } { P }

```

```

3217     { 17 } { Q }
3218     { 18 } { R }
3219     { 19 } { S }
3220     { 20 } { T }
3221     { 21 } { U }
3222     { 22 } { V }
3223     { 23 } { W }
3224     { 24 } { X }
3225     { 25 } { Y }
3226     { 26 } { Z }
3227   }
3228 }

```

`\int_to_symbol:n` Turning a number into a symbol is also easy enough.

```

3229 \cs_new_nopar:Npn \int_to_symbol:n #1{
3230   \mode_if_math:TF
3231   {
3232     \int_convert_number_with_rule:nnN {#1}{9}
3233     \int_symbol_math_conversion_rule:n
3234   }
3235   {
3236     \int_convert_number_with_rule:nnN {#1}{9}
3237     \int_symbol_text_conversion_rule:n
3238   }
3239 }

```

`\int_symbol_math_conversion_rule:n` Nothing spectacular here.

`\int_symbol_text_conversion_rule:n`

```

3240 \cs_new_nopar:Npn \int_symbol_math_conversion_rule:n #1 {
3241   \if_case:w #1
3242   \or: *
3243   \or: \dagger
3244   \or: \ddagger
3245   \or: \mathsection
3246   \or: \mathparagraph
3247   \or: \ /
3248   \or: **
3249   \or: \dagger\dagger
3250   \or: \ddagger\ddagger
3251   \fi:
3252 }
3253 \cs_new_nopar:Npn \int_symbol_text_conversion_rule:n #1 {
3254   \if_case:w #1
3255   \or: \textasteriskcentered
3256   \or: \textdagger
3257   \or: \textdaggerdbl
3258   \or: \textsection
3259   \or: \textparagraph
3260   \or: \textbardbl

```

```

3261 \or: \textasteriskcentered\textasteriskcentered
3262 \or: \textdagger\textdagger
3263 \or: \textdaggerdbl\textdaggerdbl
3264 \fi:
3265 }

```

`\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
3266 \int_new:N \l_tmpa_int
3267 \int_new:N \l_tmpb_int
3268 \int_new:N \l_tmpc_int
3269 \int_new:N \g_tmpa_int
3270 \int_new:N \g_tmpb_int

```

`\int_pre_eval_one_arg:Nn` These are handy when handing down values to other functions. All they do is evaluate the number in advance.

`\int_pre_eval_two_args:Nnn`

```

3271 \cs_set_nopar:Npn \int_pre_eval_one_arg:Nn #1#2{
3272 \exp_args:Nf#1{\int_eval:n{#2}}
3273 \cs_set_nopar:Npn \int_pre_eval_two_args:Nnn #1#2#3{
3274 \exp_args:Nff#1{\int_eval:n{#2}}{\int_eval:n{#3}}
3275 }

```

## 101.4 Scanning and conversion

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by  $\text{T}_{\text{E}}\text{X}$ .

`\int_from_roman_aux:NN`  
`\int_from_roman_end:w`  
`\int_from_roman_clean_up:w`

```

3276 \cs_new_nopar:Npn \int_from_roman:n #1 {
3277 \tl_if_blank:nF {#1}
3278 {
3279 \tex_expandafter:D \int_from_roman_end:w
3280 \tex_number:D \etex_numexpr:D
3281 \int_from_roman_aux:NN #1 Q \q_stop
3282 }
3283 }
3284 \cs_new_nopar:Npn \int_from_roman_aux:NN #1#2 {
3285 \str_if_eq:nnTF {#1} { Q }
3286 {#1#2}
3287 {
3288 \str_if_eq:nnTF {#2} { Q }
3289 {
3290 \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3291 { \int_from_roman_clean_up:w }
3292 +
3293 \use:c { c_int_from_roman_ #1 _int }
3294 #2
3295 }

```

```

3296 {
3297   \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3298   { \int_from_roman_clean_up:w }
3299   \cs_if_exist:cF { c_int_from_roman_ #2 _int }
3300   { \int_from_roman_clean_up:w }
3301   \int_compare:nNnTF
3302   { \use:c { c_int_from_roman_ #1 _int } }
3303   <
3304   { \use:c { c_int_from_roman_ #2 _int } }
3305   {
3306     + \use:c { c_int_from_roman_ #2 _int }
3307     - \use:c { c_int_from_roman_ #1 _int }
3308     \int_from_roman_aux:NN
3309   }
3310   {
3311     + \use:c { c_int_from_roman_ #1 _int }
3312     \int_from_roman_aux:NN #2
3313   }
3314 }
3315 }
3316 }
3317 \cs_new_nopar:Npn \int_from_roman_end:w #1 Q #2 \q_stop {
3318   \tl_if_empty:nTF {#2} {#1} {#2}
3319 }
3320 \cs_new_nopar:Npn \int_from_roman_clean_up:w #1 Q { + 0 Q -1 }

```

\int\_convert\_from\_base\_ten:nn  
\int\_convert\_from\_base\_ten\_aux:nnn  
\int\_convert\_number\_to\_letter:n

Converting from base ten (#1) to a second base (#2) starts with a simple sign check. As the input is base 10 TeX can then do the actual work with the sign itself.

```

3321 \cs_new:Npn \int_convert_from_base_ten:nn #1#2 {
3322   \int_compare:nNnTF {#1} < { 0 }
3323   {
3324     -
3325     \exp_args:Nnf \int_convert_from_base_ten_aux:nnn
3326     { } { \int_eval:n { 0 - ( #1 ) } } {#2}
3327   }
3328   {
3329     \exp_args:Nnf \int_convert_from_base_ten_aux:nnn
3330     { } { \int_eval:n {#1} } {#2}
3331   }
3332 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is build up as argument #1, which is why it starts off empty in the above. At each pass, the value in #2 is checked to see if it is less than the new base (#3). If it is the it is converted directly and the rest of the output is added in. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and the remainder is carried forward to the next round.S

```

3333 \cs_new:Npn \int_convert_from_base_ten_aux:nnn #1#2#3 {
3334   \int_compare:nNnTF {#2} < {#3}
3335   {
3336     \int_convert_number_to_letter:n {#2}
3337     #1
3338   }
3339   {
3340     \exp_args:Nff \int_convert_from_base_ten_aux:nnn
3341     {
3342       \int_convert_number_to_letter:n
3343       { \int_mod:nn {#2} {#3} }
3344       #1
3345     }
3346     { \int_div_truncate:nn {#2} {#3} }
3347     {#3}
3348   }
3349 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged.

```

3350 \cs_new:Npn \int_convert_number_to_letter:n #1 {
3351   \prg_case_int:nnn { #1 - 9 }
3352   {
3353     { 1 } { A }
3354     { 2 } { B }
3355     { 3 } { C }
3356     { 4 } { D }
3357     { 5 } { E }
3358     { 6 } { F }
3359     { 7 } { G }
3360     { 8 } { H }
3361     { 9 } { I }
3362     { 10 } { J }
3363     { 11 } { K }
3364     { 12 } { L }
3365     { 13 } { M }
3366     { 14 } { N }
3367     { 15 } { O }
3368     { 16 } { P }
3369     { 17 } { Q }
3370     { 18 } { R }
3371     { 19 } { S }
3372     { 20 } { T }
3373     { 21 } { U }
3374     { 22 } { V }
3375     { 23 } { W }
3376     { 24 } { X }
3377     { 25 } { Y }
3378     { 26 } { Z }
3379   }

```

```

3380     {#1}
3381 }

```

```

\int_convert_to_base_ten:nn
\int_convert_to_base_ten_aux:nnN
\int_convert_to_base_ten_aux:nnN
\int_convert_to_base_ten_aux:N
\int_get_sign_and_digits:n
\int_get_sign:n
\int_get_digits:n
\int_get_sign_and_digits_aux:nnN
\int_get_sign_and_digits_aux:nnN

```

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.

```

3382 \cs_new:Npn \int_convert_to_base_ten:nn #1#2 {
3383   \int_eval:n
3384   {
3385     \int_get_sign:n {#1}
3386     \exp_args:Nf \int_convert_to_base_ten_aux:nn
3387     { \int_get_digits:n {#1} } {#2}
3388   }
3389 }
3390 \cs_new:Npn \int_convert_to_base_ten_aux:nn #1#2 {
3391   \int_convert_to_base_ten_aux:nnN { 0 } { #2 } #1 \q_nil
3392 }
3393 \cs_new:Npn \int_convert_to_base_ten_aux:nnN #1#2#3 {
3394   \quark_if_nil:NTF #3
3395   {#1}
3396   {
3397     \exp_args:Nf \int_convert_to_base_ten_aux:nnN
3398     { \int_eval:n { #1 * #2 + \int_convert_to_base_ten_aux:N #3 } }
3399     {#2}
3400   }
3401 }

```

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.

```

3402 \cs_new:Npn \int_convert_to_base_ten_aux:N #1 {
3403   \int_compare:nNnTF { '#1 } < { 58 }
3404   {#1}
3405   {
3406     \int_eval:n
3407     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3408   }
3409 }

```

Finding a number and its sign requires dealing with an arbitrary list of + and - symbols. This is done by working through token by token until there is something else at the start of the input. The sign of the input is tracked by the first Boolean used by the auxiliary function.

```

3410 \cs_new:Npn \int_get_sign_and_digits:n #1 {
3411   \int_get_sign_and_digits_aux:nnNN {#1}
3412   \c_true_bool \c_true_bool \c_true_bool
3413 }
3414 \cs_new:Npn \int_get_sign:n #1 {
3415   \int_get_sign_and_digits_aux:nnNN {#1}

```

```

3416     \c_true_bool \c_true_bool \c_false_bool
3417 }
3418 \cs_new:Npn \int_get_digits:n #1 {
3419   \int_get_sign_and_digits_aux:nNNN {#1}
3420   \c_true_bool \c_false_bool \c_true_bool
3421 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3422 \cs_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4 {
3423   \tl_if_head_eq_charcode:fNTF {#1} -
3424   {
3425     \bool_if:NTF #2
3426     {
3427       \int_get_sign_and_digits_aux:oNNN
3428       { \use_none:n #1 } \c_false_bool #3#4
3429     }
3430     {
3431       \int_get_sign_and_digits_aux:oNNN
3432       { \use_none:n #1 } \c_true_bool #3#4
3433     }
3434   }
3435   {
3436     \tl_if_head_eq_charcode:fNTF {#1} +
3437     { \int_get_sign_and_digits_aux:oNNN { \use_none:n #1 } #2#3#4 }
3438     {
3439       \bool_if:NT #3 { \bool_if:NF #2 - }
3440       \bool_if:NT #4 {#1}
3441     }
3442   }
3443 }
3444 \cs_generate_variant:Nn \int_get_sign_and_digits_aux:nNNN { o }

```

```

\int_from_binary:n
\int_from_hexadecimal:n
\int_from_octal:n
\int_to_binary:n
\int_to_hexadecimal:n
\int_to_octal:n

```

Wrappers around the generic function.

```

3445 \cs_new:Npn \int_from_binary:n #1 {
3446   \int_convert_to_base_ten:nn {#1} { 2 }
3447 }
3448 \cs_new:Npn \int_from_hexadecimal:n #1 {
3449   \int_convert_to_base_ten:nn {#1} { 16 }
3450 }
3451 \cs_new:Npn \int_from_octal:n #1 {
3452   \int_convert_to_base_ten:nn {#1} { 8 }
3453 }
3454 \cs_new:Npn \int_to_binary:n #1 {
3455   \int_convert_from_base_ten:nn {#1} { 2 }
3456 }
3457 \cs_new:Npn \int_to_hexadecimal:n #1 {
3458   \int_convert_from_base_ten:nn {#1} { 16 }

```

```

3459 }
3460 \cs_new:Npn \int_to_octal:n #1 {
3461   \int_convert_from_base_ten:nn {#1} { 8 }
3462 }

```

`\int_from_alpha:n` The aim here is to iterate through the input, converting one letter at a time to a number.  
`\int_from_alpha_aux:n` The same approach is also used for base conversion, but this needs a different final  
`\int_from_alpha_aux:nN` auxiliary.  
`\int_from_alpha_aux:N`

```

3463 \cs_new:Npn \int_from_alpha:n #1 {
3464   \int_eval:n
3465   {
3466     \int_get_sign:n {#1}
3467     \exp_args:Nf \int_from_alpha_aux:n
3468     { \int_get_digits:n {#1} }
3469   }
3470 }
3471 \cs_new:Npn \int_from_alpha_aux:n #1 {
3472   \int_from_alpha_aux:nN { 0 } #1 \q_nil
3473 }
3474 \cs_new:Npn \int_from_alpha_aux:nN #1#2 {
3475   \quark_if_nil:NTF #2
3476   {#1}
3477   {
3478     \exp_args:Nf \int_from_alpha_aux:nN
3479     { \int_eval:n { #1 * 26 + \int_from_alpha_aux:N #2 } }
3480   }
3481 }
3482 \cs_new:Npn \int_from_alpha_aux:N #1 {
3483   \int_eval:n
3484   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }
3485 }

```

`\int_compare_p:n` Comparison tests using a simple syntax where only one set of braces is required and  
`\int_compare:nTF` additional operators such as `!=` and `>=` are supported. First some notes on the idea  
 behind this. We wish to support writing code like

```
\int_compare_p:n { 5 + \l_tmpa_int != 4 - \l_tmpb_int }
```

In other words, we want to somehow add the missing `\int_eval:w` where required. We can start evaluating from the left using `\int_eval:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let TeX evaluate this left hand side of the (in)equality.

```

3486 \prg_set_conditional:Npnn \int_compare:n #1{p,TF,T,F}{
3487   \exp_after:wN \int_compare_auxi:w \int_value:w
3488   \int_eval:w #1\q_stop
3489 }

```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is  $\langle null \rangle$ . We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3490 \cs_set:Npn \int_compare_auxi:w #1#2\q_stop{
3491   \exp_after:wN \int_compare_auxii:w \tex_romannumeral:D
3492   \if:w #1- \else: -\fi: #1#2 \q_mark #1#2 \q_stop
3493 }

```

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.

```

3494 \cs_set:Npn \int_compare_auxii:w #1#2#3\q_mark{
3495   \use:c{
3496     int_compare_
3497     #1 \if_meaning:w =#2 = \fi:
3498     :w}
3499 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3500 \cs_set:cpn {int_compare_=:w} #1=#2\q_stop{
3501   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3502   \prg_return_true: \else: \prg_return_false: \fi:
3503 }

```

So is the one using `==` – we just have to use `==` in the parameter text.

```

3504 \cs_set:cpn {int_compare_==:w} #1==#2\q_stop{
3505   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3506   \prg_return_true: \else: \prg_return_false: \fi:
3507 }

```

Not equal is just about reversing the truth value.

```

3508 \cs_set:cpn {int_compare_!=:w} #1!=#2\q_stop{
3509   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3510   \prg_return_false: \else: \prg_return_true: \fi:
3511 }

```

Less than and greater than are also straight forward.

```

3512 \cs_set:cpn {int_compare_<:w} #1<#2\q_stop{
3513   \if_int_compare:w #1<\int_eval:w #2 \int_eval_end:
3514   \prg_return_true: \else: \prg_return_false: \fi:

```

```

3515 }
3516 \cs_set:cpn {int_compare_>:w} #1>#2\q_stop{
3517   \if_int_compare:w #1>\int_eval:w #2 \int_eval_end:
3518   \prg_return_true: \else: \prg_return_false: \fi:
3519 }

```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```

3520 \cs_set:cpn {int_compare_<=:w} #1<=#2\q_stop{
3521   \if_int_compare:w #1>\int_eval:w #2 \int_eval_end:
3522   \prg_return_false: \else: \prg_return_true: \fi:
3523 }
3524 \cs_set:cpn {int_compare_>=:w} #1>=#2\q_stop{
3525   \if_int_compare:w #1<\int_eval:w #2 \int_eval_end:
3526   \prg_return_false: \else: \prg_return_true: \fi:
3527 }

```

`\int_compare_p:nNn` More efficient but less natural in typing.

`\int_compare:nNnTF`

```

3528 \prg_set_conditional:Npnn \int_compare:nNn #1#2#3{p}{
3529   \if_int_compare:w \int_eval:w #1 #2 \int_eval:w #3
3530   \int_eval_end:
3531   \prg_return_true: \else: \prg_return_false: \fi:
3532 }
3533 \cs_set_nopar:Npn \int_compare:nNnT #1#2#3 {
3534   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3535   \tex_expandafter:D \use:n
3536   \tex_else:D
3537   \tex_expandafter:D \use_none:n
3538   \tex_fi:D
3539 }
3540 \cs_set_nopar:Npn \int_compare:nNnF #1#2#3 {
3541   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3542   \tex_expandafter:D \use_none:n
3543   \tex_else:D
3544   \tex_expandafter:D \use:n
3545   \tex_fi:D
3546 }
3547 \cs_set_nopar:Npn \int_compare:nNnTF #1#2#3 {
3548   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3549   \tex_expandafter:D \use_i:nn
3550   \tex_else:D
3551   \tex_expandafter:D \use_ii:nn
3552   \tex_fi:D
3553 }

```

`\int_max:nn` Functions for min, max, and absolute value.

`\int_min:nn`

`\int_abs:n`

```

3554 \cs_set:Npn \int_abs:n #1{

```

```

3555 \int_value:w
3556 \if_int_compare:w \int_eval:w #1<\c_zero
3557 -
3558 \fi:
3559 \int_eval:w #1\int_eval_end:
3560 }
3561 \cs_set:Npn \int_max:nn #1#2{
3562 \int_value:w \int_eval:w
3563 \if_int_compare:w
3564 \int_eval:w #1>\int_eval:w #2\int_eval_end:
3565 #1
3566 \else:
3567 #2
3568 \fi:
3569 \int_eval_end:
3570 }
3571 \cs_set:Npn \int_min:nn #1#2{
3572 \int_value:w \int_eval:w
3573 \if_int_compare:w
3574 \int_eval:w #1<\int_eval:w #2\int_eval_end:
3575 #1
3576 \else:
3577 #2
3578 \fi:
3579 \int_eval_end:
3580 }

```

`\int_div_truncate:nn` As `\int_eval:w` rounds the result of a division we also provide a version that truncates the result.  
`\int_div_round:nn`  
`\int_mod:nn`

Initial version didn't work correctly with eTeX's implementation.

```

3581 %\cs_set:Npn \int_div_truncate_raw:nn #1#2 {
3582 % \int_eval:n{ (2*#1 - #2) / (2* #2) }
3583 %}

```

New version by Heiko:

```

3584 \cs_set:Npn \int_div_truncate:nn #1#2 {
3585 \int_value:w \int_eval:w
3586 \if_int_compare:w \int_eval:w #1 = \c_zero
3587 0
3588 \else:
3589 (#1
3590 \if_int_compare:w \int_eval:w #1 < \c_zero
3591 \if_int_compare:w \int_eval:w #2 < \c_zero
3592 -( #2 +
3593 \else:
3594 +( #2 -

```

```

3595         \fi:
3596     \else:
3597         \if_int_compare:w \int_eval:w #2 < \c_zero
3598             +( #2 +
3599         \else:
3600             -( #2 -
3601         \fi:
3602     \fi:
3603     1)/2)
3604     \fi:
3605     /(#2)
3606 \int_eval_end:
3607 }

```

For the sake of completeness:

```

3608 \cs_set:Npn \int_div_round:nn #1#2 {\int_eval:n{(#1)/(#2)}}

```

Finally there's the modulus operation.

```

3609 \cs_set:Npn \int_mod:nn #1#2 {
3610     \int_value:w
3611     \int_eval:w
3612     #1 - \int_div_truncate:nn {#1}{#2} * (#2)
3613     \int_eval_end:
3614 }

```

\int\_if\_odd\_p:n A predicate function.

```

\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
3615 \prg_set_conditional:Npnn \int_if_odd:n #1 {p,TF,T,F} {
3616     \if_int_odd:w \int_eval:w #1\int_eval_end:
3617     \prg_return_true: \else: \prg_return_false: \fi:
3618 }
3619 \prg_set_conditional:Npnn \int_if_even:n #1 {p,TF,T,F} {
3620     \if_int_odd:w \int_eval:w #1\int_eval_end:
3621     \prg_return_false: \else: \prg_return_true: \fi:
3622 }

```

\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
3623 \cs_set:Npn \int_while_do:nn #1#2{
3624     \int_compare:nT {#1}{#2} \int_while_do:nn {#1}{#2}}
3625 }
3626 \cs_set:Npn \int_until_do:nn #1#2{
3627     \int_compare:nF {#1}{#2} \int_until_do:nn {#1}{#2}}
3628 }
3629 \cs_set:Npn \int_do_while:nn #1#2{
3630     #2 \int_compare:nT {#1}{\int_do_while:nNnn {#1}{#2}}
3631 }

```

```

3632 \cs_set:Npn \int_do_until:nn #1#2{
3633   #2 \int_compare:nF {#1}{\int_do_until:nn {#1}{#2}}
3634 }

```

\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
3635 \cs_set:Npn \int_while_do:nNnn #1#2#3#4{
3636   \int_compare:nNnT {#1}#2{#3}{#4 \int_while_do:nNnn {#1}#2{#3}{#4}}
3637 }
3638 \cs_set:Npn \int_until_do:nNnn #1#2#3#4{
3639   \int_compare:nNnF {#1}#2{#3}{#4 \int_until_do:nNnn {#1}#2{#3}{#4}}
3640 }
3641 \cs_set:Npn \int_do_while:nNnn #1#2#3#4{
3642   #4 \int_compare:nNnT {#1}#2{#3}{\int_do_while:nNnn {#1}#2{#3}{#4}}
3643 }
3644 \cs_set:Npn \int_do_until:nNnn #1#2#3#4{
3645   #4 \int_compare:nNnF {#1}#2{#3}{\int_do_until:nNnn {#1}#2{#3}{#4}}
3646 }

```

## 101.5 Defining constants

\int\_const:Nn As stated, most constants can be defined as \tex\_chardef:D or \tex\_mathchardef:D  
\int\_const:cn but that's engine dependent.

```

3647 \cs_new_protected_nopar:Npn \int_const:Nn #1#2 {
3648   \int_compare:nTF { #2 > \c_minus_one }
3649   {
3650     \int_compare:nTF { #2 > \c_max_register_int }
3651     {
3652       \int_new:N #1
3653       \int_gset:Nn #1 {#2}
3654     }
3655     {
3656       \chk_if_free_cs:N #1
3657       \tex_global:D \tex_mathchardef:D #1 = \int_eval:n {#2}
3658     }
3659   }
3660   {
3661     \int_new:N #1
3662     \int_gset:Nn #1 {#2}
3663   }
3664 }
3665 \cs_generate_variant:Nn \int_const:Nn { c }

```

\c\_max\_register\_int This is here as this particular integer is needed both in package mode and to bootstrap l3alloc

```

3666 \tex_mathchardef:D \c_max_register_int = 32767 \scan_stop:

```

```

\c_minus_one
\c_zero
\c_one
\c_two
\c_three 3667 %% \tex_countdef:D \c_minus_one = 10 \scan_stop:
\c_four   3668 %% \c_minus_one = -1 \scan_stop: %% in l3basics
\c_five   3669 %\int_const:Nn \c_zero {0}
\c_six    3670 \int_const:Nn \c_one {1}
\c_seven  3671 \int_const:Nn \c_two {2}
\c_eight  3672 \int_const:Nn \c_three {3}
\c_nine   3673 \int_const:Nn \c_four {4}
\c_ten    3674 \int_const:Nn \c_five {5}
\c_eleven 3675 \int_const:Nn \c_six {6}
\c_twelve 3676 \int_const:Nn \c_seven {7}
\c_thirteen 3677 \int_const:Nn \c_eight {8}
\c_fourteen 3678 \int_const:Nn \c_nine {9}
\c_fifteen 3679 \int_const:Nn \c_ten {10}
\c_sixteen 3680 \int_const:Nn \c_eleven {11}
\c_seventeen 3681 \int_const:Nn \c_twelve {12}
\c_eighteen 3682 \int_const:Nn \c_thirteen {13}
\c_nineteen 3683 \int_const:Nn \c_fourteen {14}
\c_twenty 3684 \int_const:Nn \c_fifteen {15}
\c_thirty 3685 %% \tex_chardef:D \c_sixteen = 16\scan_stop: %% in l3basics
\c_fifty 3686 \int_const:Nn \c_thirty_two {32}
\c_hundred
\c_thousand
\c_ten_thousand
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three 3687 \int_const:Nn \c_hundred_one {101}
\c_ten_thousand_four 3688 \int_const:Nn \c_twohundred_fifty_five {255}
\c_twenty_thousand 3689 \int_const:Nn \c_twohundred_fifty_six {256}
3690 \int_const:Nn \c_thousand {1000}
3691 \int_const:Nn \c_ten_thousand {10000}
3692 \int_const:Nn \c_ten_thousand_one {10001}
3693 \int_const:Nn \c_ten_thousand_two {10002}
3694 \int_const:Nn \c_ten_thousand_three {10003}
3695 \int_const:Nn \c_ten_thousand_four {10004}
3696 \int_const:Nn \c_twenty_thousand {20000}

```

And the usual constants, others are still missing. Please, make every constant a real constant at least for the moment. We can easily convert things in the end when we have found what constants are used in critical places and what not.

The next one may seem a little odd (obviously!) but is useful when dealing with logical operators.

```

\c_max_int 3697 \int_const:Nn \c_max_int {2147483647}

```

The largest number allowed is  $2^{31} - 1$

```

\c_int_from_roman_i_int
\c_int_from_roman_v_int
\c_int_from_roman_x_int
\l_int_from_roman_l_int
\c_int_from_roman_c_int
\c_int_from_roman_d_int
\c_int_from_roman_m_int
\c_int_from_roman_I_int
\c_int_from_roman_V_int
\c_int_from_roman_X_int
\c_int_from_roman_L_int
\c_int_from_roman_C_int
\c_int_from_roman_D_int
\c_int_from_roman_M_int

```

Delayed from earlier.

```

3698 \int_const:cn { c_int_from_roman_i_int } { 1 }
3699 \int_const:cn { c_int_from_roman_v_int } { 5 }
3700 \int_const:cn { c_int_from_roman_x_int } { 10 }
3701 \int_const:cn { c_int_from_roman_l_int } { 50 }
3702 \int_const:cn { c_int_from_roman_c_int } { 100 }

```

```

3703 \int_const:cn { c_int_from_roman_d_int } { 500 }
3704 \int_const:cn { c_int_from_roman_m_int } { 1000 }
3705 \int_const:cn { c_int_from_roman_I_int } { 1 }
3706 \int_const:cn { c_int_from_roman_V_int } { 5 }
3707 \int_const:cn { c_int_from_roman_X_int } { 10 }
3708 \int_const:cn { c_int_from_roman_L_int } { 50 }
3709 \int_const:cn { c_int_from_roman_C_int } { 100 }
3710 \int_const:cn { c_int_from_roman_D_int } { 500 }
3711 \int_const:cn { c_int_from_roman_M_int } { 1000 }

```

Needed from the tl module:

```

3712 \int_new:N \g_tl_inline_level_int

```

## 101.6 Backwards compatibility

```

3713 \cs_set_eq:NN \intexpr_value:w \int_value:w
3714 \cs_set_eq:NN \intexpr_eval:w \int_eval:w
3715 \cs_set_eq:NN \intexpr_eval_end: \int_eval_end:
3716 \cs_set_eq:NN \if_intexpr_compare:w \if_int_compare:w
3717 \cs_set_eq:NN \if_intexpr_odd:w \if_int_odd:w
3718 \cs_set_eq:NN \if_intexpr_case:w \if_case:w
3719 \cs_set_eq:NN \intexpr_eval:n \int_eval:n
3720
3721 \cs_set_eq:NN \intexpr_compare_p:n \int_compare_p:n
3722 \cs_set_eq:NN \intexpr_compare:nTF \int_compare:nTF
3723 \cs_set_eq:NN \intexpr_compare:nT \int_compare:nT
3724 \cs_set_eq:NN \intexpr_compare:nF \int_compare:nF
3725
3726 \cs_set_eq:NN \intexpr_compare_p:nNn \int_compare_p:nNn
3727 \cs_set_eq:NN \intexpr_compare:nNnTF \int_compare:nNnTF
3728 \cs_set_eq:NN \intexpr_compare:nNnT \int_compare:nNnT
3729 \cs_set_eq:NN \intexpr_compare:nNnF \int_compare:nNnF
3730
3731 \cs_set_eq:NN \intexpr_abs:n \int_abs:n
3732 \cs_set_eq:NN \intexpr_max:nn \int_max:nn
3733 \cs_set_eq:NN \intexpr_min:nn \int_min:nn
3734
3735 \cs_set_eq:NN \intexpr_div_truncate:nn \int_div_truncate:nn
3736 \cs_set_eq:NN \intexpr_div_round:nn \int_div_round:nn
3737 \cs_set_eq:NN \intexpr_mod:nn \int_mod:nn
3738
3739 \cs_set_eq:NN \intexpr_if_odd_p:n \int_if_odd_p:n
3740 \cs_set_eq:NN \intexpr_if_odd:nTF \int_if_odd:nTF
3741 \cs_set_eq:NN \intexpr_if_odd:nT \int_if_odd:nT
3742 \cs_set_eq:NN \intexpr_if_odd:nF \int_if_odd:nF
3743
3744 \cs_set_eq:NN \intexpr_if_even_p:n \int_if_even_p:n
3745 \cs_set_eq:NN \intexpr_if_even:nTF \int_if_even:nTF

```

```

3746 \cs_set_eq:NN \intexpr_if_even:nT \int_if_even:nT
3747 \cs_set_eq:NN \intexpr_if_even:nF \int_if_even:nF
3748
3749 \cs_set_eq:NN \intexpr_while_do:nn \int_while_do:nn
3750 \cs_set_eq:NN \intexpr_until_do:nn \int_until_do:nn
3751 \cs_set_eq:NN \intexpr_do_while:nn \int_do_while:nn
3752 \cs_set_eq:NN \intexpr_do_until:nn \int_do_until:nn
3753
3754 \cs_set_eq:NN \intexpr_while_do:nNnn \int_while_do:nNnn
3755 \cs_set_eq:NN \intexpr_until_do:nNnn \int_until_do:nNnn
3756 \cs_set_eq:NN \intexpr_do_while:nNnn \int_do_while:nNnn
3757 \cs_set_eq:NN \intexpr_do_until:nNnn \int_do_until:nNnn
3758 </initex | package>

```

Show token usage:

```

3759 <*showmemory>
3760 \showMemUsage
3761 </showmemory>

```

## 102 l3skip implementation

We start by ensuring that the required packages are loaded.

```

3762 <*package>
3763 \ProvidesExplPackage
3764   {\filename}{\filedate}{\fileversion}{\filedescription}
3765   \package_check_loaded_expl:
3766 </package>
3767 <*initex | package>

```

### 102.1 Skip registers

`\skip_new:N` Allocation of a new internal registers.

`\skip_new:c`

```

3768 <*initex>
3769 \alloc_new:nnnN {skip} \c_zero \c_max_register_int \tex_skipdef:D
3770 </initex>
3771 <*package>
3772 \cs_new_protected_nopar:Npn \skip_new:N #1 {
3773   \chk_if_free_cs:N #1
3774   \newskip #1
3775 }
3776 </package>
3777 \cs_generate_variant:Nn \skip_new:N {c}

```

`\skip_set:Nn` Setting skips is again something that I would like to make uniform at the moment to get  
`\skip_set:cn` a better overview.

`\skip_gset:Nn`

`\skip_gset:cn`

```

3778 \cs_new_protected_nopar:Npn \skip_set:Nn #1#2 {
3779   #1\skip_eval:n{#2}
3780   (*check)
3781   \chk_local_or_pref_global:N #1
3782   (/check)
3783 }
3784 \cs_new_protected_nopar:Npn \skip_gset:Nn {
3785   (*check)
3786   \pref_global_chk:
3787   (/check)
3788   (-check) \pref_global:D
3789   \skip_set:Nn
3790 }
3791 \cs_generate_variant:Nn \skip_set:Nn {cn}
3792 \cs_generate_variant:Nn \skip_gset:Nn {cn}

```

\skip\_zero:N Reset the register to zero.

```

\skip_gzero:N
\skip_zero:c
\skip_gzero:c
3793 \cs_new_protected_nopar:Npn \skip_zero:N #1{
3794   #1\c_zero_skip \scan_stop:
3795   (*check)
3796   \chk_local_or_pref_global:N #1
3797   (/check)
3798 }
3799 \cs_new_protected_nopar:Npn \skip_gzero:N {

```

We make sure that a local variable is not updated globally by changing the internal test (i.e. \chk\_local\_or\_pref\_global:N) before making the assignment. This is done by \pref\_global\_chk: which also issues the necessary \pref\_global:D. This is not very efficient, but this code will be only included for debugging purposes. Using \pref\_global:D in front of the local function is better in the production versions.

```

3800 (*check)
3801 \pref_global_chk:
3802 (/check)
3803 (-check) \pref_global:D
3804 \skip_zero:N
3805 }
3806 \cs_generate_variant:Nn \skip_zero:N {c}
3807 \cs_generate_variant:Nn \skip_gzero:N {c}

```

\skip\_add:Nn Adding and subtracting to and from <skip>s

```

\skip_add:cn
\skip_gadd:Nn
3808 \cs_new_protected_nopar:Npn \skip_add:Nn #1#2 {

```

\skip\_gadd:cn We need to say by in case the first argument is a register accessed by its number, e.g.,  
\skip\_sub:Nn \skip23.  
\skip\_gsub:Nn

```

3809 \tex_advance:D#1 by \skip_eval:n{#2}
3810 (*check)

```

```

3811 \chk_local_or_pref_global:N #1
3812 </check>
3813 }
3814 \cs_generate_variant:Nn \skip_add:Nn {cn}

3815 \cs_new_protected_nopar:Npn \skip_sub:Nn #1#2{
3816 \tex_advance:D#1-\skip_eval:n{#2}
3817 <*check>
3818 \chk_local_or_pref_global:N #1
3819 </check>
3820 }

3821 \cs_new_protected_nopar:Npn \skip_gadd:Nn {
3822 <*check>
3823 \pref_global_chk:
3824 </check>
3825 <-check> \pref_global:D
3826 \skip_add:Nn
3827 }
3828 \cs_generate_variant:Nn \skip_gadd:Nn {cn}

3829 \cs_new_nopar:Npn \skip_gsub:Nn {
3830 <*check>
3831 \pref_global_chk:
3832 </check>
3833 <-check> \pref_global:D
3834 \skip_sub:Nn
3835 }

```

\skip\_horizontal:N Inserting skips.

```

\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n
3836 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
3837 \cs_generate_variant:Nn \skip_horizontal:N {c}

3838 \cs_new_nopar:Npn \skip_horizontal:n #1 { \skip_horizontal:N \skip_eval:n{#1} }
3839 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
3840 \cs_generate_variant:Nn \skip_vertical:N {c}
3841 \cs_new_nopar:Npn \skip_vertical:n #1 { \skip_vertical:N \skip_eval:n{#1} }

```

\skip\_use:N Here is how skip registers are accessed:

```

\skip_use:c
3842 \cs_new_eq:NN \skip_use:N \tex_the:D
3843 \cs_generate_variant:Nn \skip_use:N {c}

```

\skip\_show:N Diagnostics.

```

\skip_show:c
3844 \cs_new_eq:NN \skip_show:N \kernel_register_show:N
3845 \cs_generate_variant:Nn \skip_show:N {c}

```

`\skip_eval:n` Evaluating a calc expression.

```
3846 \cs_new_protected_nopar:Npn \skip_eval:n #1 { \etex_glueexpr:D #1 \scan_stop: }
```

`\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`

```
3847 %%\chk_if_free_cs:N \l_tmpa_skip
3848 %%\tex_skipdef:D\l_tmpa_skip 255 %currently taken up by \skip@
3849 \skip_new:N \l_tmpa_skip
3850 \skip_new:N \l_tmpb_skip
3851 \skip_new:N \l_tmpc_skip
3852 \skip_new:N \g_tmpa_skip
3853 \skip_new:N \g_tmpb_skip
```

`\c_zero_skip`

`\c_max_skip`

```
3854 \*!package)
3855 \skip_new:N \c_zero_skip
3856 \skip_set:Nn \c_zero_skip {0pt}
3857 \skip_new:N \c_max_skip
3858 \skip_set:Nn \c_max_skip {16383.99999pt}
3859 \*!package)
3860 \*!initex)
3861 \cs_set_eq:NN \c_zero_skip \z@
3862 \cs_set_eq:NN \c_max_skip \maxdimen
3863 \*!initex)
```

`\skip_if_infinite_glue_p:n`

`\skip_if_infinite_glue:nTF`

With  $\varepsilon$ -TeX we all of a sudden get access to a lot 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. `\skip_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.

```
3864 \prg_new_conditional:Nnn \skip_if_infinite_glue:n {p,TF,T,F} {
3865   \bool_if:nTF {
3866     \int_compare_p:nNn {\etex_gluestretchorder:D #1 } > \c_zero ||
3867     \int_compare_p:nNn {\etex_glueshrinkorder:D #1 } > \c_zero
3868   } {\prg_return_true:} {\prg_return_false:}
3869 }
```

`\o_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.

```
3870 \cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4{
3871   \skip_if_infinite_glue:nTF {#1}
3872   {
3873     #3 = \c_zero_skip
```

```

3874     #4 = \c_zero_skip
3875     #2
3876   }
3877   {
3878     #3 = \etex_gluestretch:D #1 \scan_stop:
3879     #4 = \etex_glueshrink:D #1 \scan_stop:
3880   }
3881 }

```

## 102.2 Dimen registers

`\dim_new:N` Allocating  $\langle dim \rangle$  registers...

```

\dim_new:c
3882 \*initex
3883 \alloc_new:nnnN {dim} \c_zero \c_max_register_int \tex_dimendef:D
3884 \*initex
3885 \*package
3886 \cs_new_protected_nopar:Npn \dim_new:N #1 {
3887   \chk_if_free_cs:N #1
3888   \newdimen #1
3889 }
3890 \*package
3891 \cs_generate_variant:Nn \dim_new:N {c}

```

`\dim_set:Nn` We add `\dim_eval:n` in order to allow simple arithmetic and a space just for those using  
`\dim_set:cn` `\dimen1` or alike. See OR!

```

\dim_set:Nc
\dim_gset:Nn
\dim_gset:cn
\dim_gset:Nc
\dim_gset:cc
3892 \cs_new_protected_nopar:Npn \dim_set:Nn #1#2 { #1~ \dim_eval:n{#2} }
3893 \cs_generate_variant:Nn \dim_set:Nn {cn,Nc}
3894 \cs_new_protected_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn }
3895 \cs_generate_variant:Nn \dim_gset:Nn {cn,Nc,cc}

```

`\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
3896 \cs_new_protected_nopar:Npn \dim_set_max:Nn #1#2 {
3897   \dim_compare:nNnT {#1} < {#2} { \dim_set:Nn #1 {#2} }
3898 }
3899 \cs_generate_variant:Nn \dim_set_max:Nn { c }
3900 \cs_new_protected_nopar:Npn \dim_set_min:Nn #1#2 {
3901   \dim_compare:nNnT {#1} > {#2} { \dim_set:Nn #1 {#2} }
3902 }
3903 \cs_generate_variant:Nn \dim_set_min:Nn { c }
3904 \cs_new_protected_nopar:Npn \dim_gset_max:Nn #1#2 {
3905   \dim_compare:nNnT {#1} < {#2} { \dim_gset:Nn #1 {#2} }
3906 }
3907 \cs_generate_variant:Nn \dim_gset_max:Nn { c }

```

```

3908 \cs_new_protected_nopar:Npn \dim_gset_min:Nn #1#2 {
3909   \dim_compare:nNnT {#1} > {#2} { \dim_gset:Nn #1 {#2} }
3910 }
3911 \cs_generate_variant:Nn \dim_gset_min:Nn { c }

\dim_zero:N Resetting.
\dim_gzero:N
\dim_zero:c 3912 \cs_new_protected_nopar:Npn \dim_zero:N #1 { #1\c_zero_skip }
\dim_gzero:c 3913 \cs_generate_variant:Nn \dim_zero:N {c}

3914 \cs_new_protected_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N }
3915 \cs_generate_variant:Nn \dim_gzero:N {c}

\dim_add:Nn Addition.
\dim_add:cn
\dim_add:Nc 3916 \cs_new_protected_nopar:Npn \dim_add:Nn #1#2{
\dim_gadd:Nn We need to say by in case the first argument is a register accessed by its number, e.g.,
\dim_gadd:cn \dimen23.

3917   \tex_advance:D#1 by \dim_eval:n{#2}\scan_stop:
3918 }
3919 \cs_generate_variant:Nn \dim_add:Nn {cn,Nc}

3920 \cs_new_protected_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
3921 \cs_generate_variant:Nn \dim_gadd:Nn {cn}

\dim_sub:Nn Subtracting.
\dim_sub:cn
\dim_sub:Nc 3922 \cs_new_protected_nopar:Npn \dim_sub:Nn #1#2 { \tex_advance:D#1-#2\scan_stop: }
\dim_gsub:Nn 3923 \cs_generate_variant:Nn \dim_sub:Nn {cn,Nc}
\dim_gsub:cn 3924 \cs_new_protected_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn }
3925 \cs_generate_variant:Nn \dim_gsub:Nn {cn}

\dim_use:N Accessing a  $\langle dim \rangle$ .
\dim_use:c 3926 \cs_new_eq:NN \dim_use:N \tex_the:D
3927 \cs_generate_variant:Nn \dim_use:N {c}

\dim_show:N Diagnostics.
\dim_show:c 3928 \cs_new_eq:NN \dim_show:N \kernel_register_show:N
3929 \cs_generate_variant:Nn \dim_show:N {c}

\l_tmpa_dim Some scratch registers.
\l_tmpb_dim
\l_tmpc_dim 3930 \dim_new:N \l_tmpa_dim
\l_tmpd_dim 3931 \dim_new:N \l_tmpb_dim
\g_tmpa_dim 3932 \dim_new:N \l_tmpc_dim
\g_tmpb_dim 3933 \dim_new:N \l_tmpd_dim
3934 \dim_new:N \g_tmpa_dim
3935 \dim_new:N \g_tmpb_dim

```

`\c_zero_dim` Just aliases.

`\c_max_dim`

```
3936 \cs_new_eq:NN \c_zero_dim \c_zero_skip
```

```
3937 \cs_new_eq:NN \c_max_dim \c_max_skip
```

`\dim_eval:n` Evaluating a calc expression.

```
3938 \cs_new_protected_nopar:Npn \dim_eval:n #1 { \etex_dimexpr:D #1 \scan_stop: }
```

`\if_dim:w` Primitives.

`\dim_value:w`

`\dim_eval:w`

`\dim_eval_end:`

```
3939 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
```

```
3940 \cs_set_eq:NN \dim_value:w \tex_number:D
```

```
3941 \cs_set_eq:NN \dim_eval:w \etex_dimexpr:D
```

```
3942 \cs_set_protected:Npn \dim_eval_end: {\tex_relax:D}
```

`\dim_compare_p:nNn`

`\dim_compare:nNnTF`

```
3943 \prg_new_conditional:Nnn \dim_compare:nNn {p,TF,T,F} {
```

```
3944   \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
```

```
3945   \prg_return_true: \else: \prg_return_false: \fi:
```

```
3946 }
```

`\dim_compare_p:n` [This code plus comments lifted directly from the `\int_compare:nTF` function.] Comparison tests using a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. First some notes on the idea behind this. We wish to support writing code like

`\dim_compare:nNnTF`

```
\dim_compare_p:n { 5 + \l_tmpa_dim != 4 - \l_tmpb_dim }
```

In other words, we want to somehow add the missing `\dim_eval:w` where required. We can start evaluating from the left using `\dim:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let  $\TeX$  evaluate this left hand side of the (in)equality.

```
3947 \prg_new_conditional:Npnn \dim_compare:n #1 {p,TF,T,F} {
```

```
3948   \exp_after:wN \dim_compare_auxi:w \dim_value:w
```

```
3949   \dim_eval:w #1 \q_stop
```

```
3950 }
```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is `\langle null \rangle`. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3951 \cs_new:Npn \dim_compare_auxi:w #1#2 \q_stop {
3952   \exp_after:wN \dim_compare_auxii:w \tex_romannumeral:D
3953   \if:w #1- \else: -\fi: #1#2 \q_mark #1#2 \q_stop
3954 }

```

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.

```

3955 \cs_new:Npn \dim_compare_auxii:w #1#2#3\q_mark{
3956   \use:c{
3957     dim_compare_ #1 \if_meaning:w =#2 = \fi:
3958     :w}
3959 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3960 \cs_new:cpn {dim_compare_=:w} #1 = #2 \q_stop {
3961   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3962   \prg_return_true: \else: \prg_return_false: \fi:
3963 }

```

So is the one using == – we just have to use == in the parameter text.

```

3964 \cs_new:cpn {dim_compare_=:w} #1 == #2 \q_stop {
3965   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3966   \prg_return_true: \else: \prg_return_false: \fi:
3967 }

```

Not equal is just about reversing the truth value.

```

3968 \cs_new:cpn {dim_compare_!=:w} #1 != #2 \q_stop {
3969   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3970   \prg_return_false: \else: \prg_return_true: \fi:
3971 }

```

Less than and greater than are also straight forward.

```

3972 \cs_new:cpn {dim_compare_<:w} #1 < #2 \q_stop {
3973   \if_dim:w #1 sp < \dim_eval:w #2 \dim_eval_end:
3974   \prg_return_true: \else: \prg_return_false: \fi:
3975 }
3976 \cs_new:cpn {dim_compare_>:w} #1 > #2 \q_stop {
3977   \if_dim:w #1 sp > \dim_eval:w #2 \dim_eval_end:
3978   \prg_return_true: \else: \prg_return_false: \fi:
3979 }

```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```

3980 \cs_new:cpn {dim_compare_<=:w} #1 <= #2 \q_stop {
3981   \if_dim:w #1 sp > \dim_eval:w #2 \dim_eval_end:
3982   \prg_return_false: \else: \prg_return_true: \fi:
3983 }
3984 \cs_new:cpn {dim_compare_>=:w} #1 >= #2 \q_stop {
3985   \if_dim:w #1 sp < \dim_eval:w #2 \dim_eval_end:
3986   \prg_return_false: \else: \prg_return_true: \fi:
3987 }

```

\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
3988 \cs_new_nopar:Npn \dim_while_do:nNnn #1#2#3#4{
3989   \dim_compare:nNnT {#1}#2{#3}{#4} \dim_while_do:nNnn {#1}#2{#3}{#4}}
3990 }
3991 \cs_new_nopar:Npn \dim_until_do:nNnn #1#2#3#4{
3992   \dim_compare:nNnF {#1}#2{#3}{#4} \dim_until_do:nNnn {#1}#2{#3}{#4}}
3993 }
3994 \cs_new_nopar:Npn \dim_do_while:nNnn #1#2#3#4{
3995   #4 \dim_compare:nNnT {#1}#2{#3}{\dim_do_while:nNnn {#1}#2{#3}{#4}}
3996 }
3997 \cs_new_nopar:Npn \dim_do_until:nNnn #1#2#3#4{
3998   #4 \dim_compare:nNnF {#1}#2{#3}{\dim_do_until:nNnn {#1}#2{#3}{#4}}
3999 }

```

## 102.3 Muskips

\muskip\_new:N And then we add muskips.

```

4000 \*initex
4001 \alloc_new:nnnN {muskip} \c_zero \c_max_register_int \tex_muskipdef:D
4002 \*initex
4003 \*package
4004 \cs_new_protected_nopar:Npn \muskip_new:N #1 {
4005   \chk_if_free_cs:N #1
4006   \newmuskip #1
4007 }
4008 \*package

```

\muskip\_set:Nn Simple functions for muskips.

```

\muskip_gset:Nn
\muskip_add:Nn
\muskip_gadd:Nn
\muskip_sub:Nn
\muskip_gsub:Nn
4009 \cs_new_protected_nopar:Npn \muskip_set:Nn#1#2{\etex_muexpr:D#2\scan_stop:}
4010 \cs_new_protected_nopar:Npn \muskip_gset:Nn{\pref_global:D\muskip_set:Nn}
4011 \cs_new_protected_nopar:Npn \muskip_add:Nn#1#2{\tex_advance:D#1\etex_muexpr:D#2\scan_stop:}
4012 \cs_new_protected_nopar:Npn \muskip_gadd:Nn{\pref_global:D\muskip_add:Nn}
4013 \cs_new_protected_nopar:Npn \muskip_sub:Nn#1#2{\tex_advance:D#1-\etex_muexpr:D#2\scan_stop:}
4014 \cs_new_protected_nopar:Npn \muskip_gsub:Nn{\pref_global:D\muskip_sub:Nn}

```

`\muskip_use:N` Accessing a  $\langle muskip \rangle$ .

```
4015 \cs_new_eq:NN \muskip_use:N \tex_the:D
```

`\muskip_show:N`

```
4016 \cs_new_eq:NN \muskip_show:N \kernel_register_show:N
```

```
4017 \</initex | package>
```

## 103 l3tl implementation

We start by ensuring that the required packages are loaded.

```
4018 \*package>
4019 \ProvidesExplPackage
4020   {\filename}{\filedate}{\fileversion}{\filedescription}
4021 \package_check_loaded_expl:
4022 \</package>

4023 \*initex | package>
```

A token list variable is a control sequence that holds tokens. The interface is similar to that for token registers, but beware that the behavior vis á vis `\cs_set_nopar:Npx` etc. ... is different. (You see this comes from Denys' implementation.)

### 103.1 Functions

`\tl_new:N` We provide one allocation function (which checks that the name is not used) and two  
`\tl_new:c` clear functions that locally or globally clear the token list. The allocation function has  
`\tl_new:Nn` two arguments to specify an initial value. This is the only way to give values to constants.  
`\tl_new:cn`  
`\tl_new:Nx`

```
4024 \cs_new_protected:Npn \tl_new:Nn #1#2{
4025   \chk_if_free_cs:N #1
```

If checking we don't allow constants to be defined.

```
4026 \*check>
4027   \chk_var_or_const:N #1
4028 \</check>
```

Otherwise any variable type is allowed.

```
4029 \cs_gset_nopar:Npn #1{#2}
4030 }
4031 \cs_generate_variant:Nn \tl_new:Nn {cn}
4032 \cs_new_protected:Npn \tl_new:Nx #1#2{
```

```

4033 \chk_if_free_cs:N #1
4034 <check> \chk_var_or_const:N #1
4035 \cs_gset_nopar:Npx #1{#2}
4036 }
4037 \cs_new_protected_nopar:Npn \tl_new:N #1{\tl_new:Nn #1{}}
4038 \cs_new_protected_nopar:Npn \tl_new:c #1{\tl_new:cn {#1}{}}

```

`\tl_const:Nn` For creating constant token lists: there is not actually anything here that cannot be achieved using `\tl_new:N` and `\tl_set:Nn`

```

4039 \cs_new_protected:Npn \tl_const:Nn #1#2 {
4040   \tl_new:N #1
4041   \tl_gset:Nn #1 {#2}
4042 }

```

`\tl_use:N` Perhaps this should just be enabled when checking?

```

\tl_use:c
4043 \cs_new_nopar:Npn \tl_use:N #1 {
4044   \if_meaning:w #1 \tex_relax:D

```

If `<tl var.>` equals `\tex_relax:D` it is probably stemming from a `\cs:w... \cs_end:` that was created by mistake somewhere.

```

4045 \msg_kernel_bug:x {Token~list~variable~ '\token_to_str:N #1'~
4046                   has~ an~ erroneous~ structure!}
4047 \else:
4048   \exp_after:wN #1
4049 \fi:
4050 }
4051 \cs_generate_variant:Nn \tl_use:N {c}

```

`\tl_show:N` Showing a `<tl var.>` is just `\show`ing it and I don't really care about checking that it's malformed at this stage.

```

\tl_show:c
\tl_show:n
4052 \cs_new_nopar:Npn \tl_show:N #1 { \cs_show:N #1 }
4053 \cs_generate_variant:Nn \tl_show:N {c}
4054 \cs_set_eq:NN \tl_show:n \etex_showtokens:D

```

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain # tokens.

```

\tl_set:NV
\tl_set:Nv
4055 \cs_new_protected:Npn \tl_set:Nn #1#2 {
4056   \cs_set_nopar:Npx #1 { \exp_not:n {#2} }
\tl_set:No
4057 }
\tl_set:Nf
4058 \cs_new_protected:Npn \tl_set:Nx #1#2 {
\tl_set:Nx
4059   \cs_set_nopar:Npx #1 {#2}
\tl_set:cn
4060 }
\tl_set:cV
4061 \cs_new_protected:Npn \tl_gset:Nn #1#2 {
\tl_set:cv
4062   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
\tl_set:co
4063 }
\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:cV
\tl_gset:cv

```

```

4064 \cs_new_protected:Npn \tl_gset:Nx #1#2 {
4065   \cs_gset_nopar:Npx #1 {#2}
4066 }
4067 \cs_generate_variant:Nn \tl_set:Nn { NV }
4068 \cs_generate_variant:Nn \tl_set:Nn { Nv }
4069 \cs_generate_variant:Nn \tl_set:Nn { No }
4070 \cs_generate_variant:Nn \tl_set:Nn { Nf }
4071 \cs_generate_variant:Nn \tl_set:Nn { cV }
4072 \cs_generate_variant:Nn \tl_set:Nn { c }
4073 \cs_generate_variant:Nn \tl_set:Nn { cv }
4074 \cs_generate_variant:Nn \tl_set:Nn { co }
4075 \cs_generate_variant:Nn \tl_set:Nx { c }
4076 \cs_generate_variant:Nn \tl_gset:Nn { NV }
4077 \cs_generate_variant:Nn \tl_gset:Nn { Nv }
4078 \cs_generate_variant:Nn \tl_gset:Nn { No }
4079 \cs_generate_variant:Nn \tl_gset:Nn { Nf }
4080 \cs_generate_variant:Nn \tl_gset:Nn { c }
4081 \cs_generate_variant:Nn \tl_gset:Nn { cV }
4082 \cs_generate_variant:Nn \tl_gset:Nn { cv }
4083 \cs_generate_variant:Nn \tl_gset:Nx { c }

```

\tl\_set\_eq:NN For setting token list variables equal to each other. First checking:

```

\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc
4084 \cs_new_protected_nopar:Npn \tl_set_eq:NN #1#2{
4085   \chk_exist_cs:N #1 \cs_set_eq:NN #1#2
4086   \chk_local_or_pref_global:N #1 \chk_var_or_const:N #2
4087 }
4088 \cs_new_protected_nopar:Npn \tl_gset_eq:NN #1#2{
4089   \chk_exist_cs:N #1 \cs_gset_eq:NN #1#2
4090   \chk_global:N #1 \chk_var_or_const:N #2
4091 }
4092 \check
4093 \check

```

Non-checking versions are easy.

```

4094 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
4095 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4096 \check
4097 \check

```

The rest again with the expansion module.

```

4098 \cs_generate_variant:Nn \tl_set_eq:NN {Nc,c,cc}
4099 \cs_generate_variant:Nn \tl_gset_eq:NN {Nc,c,cc}

```

\tl\_clear:N Clearing a token list variable.

```

\tl_clear:c
\tl_gclear:N
\tl_gclear:c
4100 \cs_new_protected_nopar:Npn \tl_clear:N #1{\tl_set_eq:NN #1\c_empty_tl}
4101 \cs_generate_variant:Nn \tl_clear:N {c}
4102 \cs_new_protected_nopar:Npn \tl_gclear:N #1{\tl_gset_eq:NN #1\c_empty_tl}
4103 \cs_generate_variant:Nn \tl_gclear:N {c}

```

`\tl_clear_new:N` These macros check whether a token list exists. If it does it is cleared, if it doesn't it is allocated.  
`\tl_clear_new:c`

```

4104 <*check>
4105 \cs_new_protected_nopar:Npn \tl_clear_new:N #1{
4106   \chk_var_or_const:N #1
4107   \if_predicate:w \cs_if_exist_p:N #1
4108     \tl_clear:N #1
4109   \else:
4110     \tl_new:N #1
4111   \fi:
4112 }
4113 </check>
4114 <-check>\cs_new_eq:NN \tl_clear_new:N \tl_clear:N
4115 \cs_generate_variant:Nn \tl_clear_new:N {c}

```

`\tl_gclear_new:N` These are the global versions of the above.  
`\tl_gclear_new:c`

```

4116 <*check>
4117 \cs_new_protected_nopar:Npn \tl_gclear_new:N #1{
4118   \chk_var_or_const:N #1
4119   \if_predicate:w \cs_if_exist_p:N #1
4120     \tl_gclear:N #1
4121   \else:
4122     \tl_new:N #1
4123   \fi:}
4124 </check>
4125 <-check>\cs_new_eq:NN \tl_gclear_new:N \tl_gclear:N
4126 \cs_generate_variant:Nn \tl_gclear_new:N {c}

```

`\tl_put_right:Nn` Adding to one end of a token list is done partially using hand tuned functions for performance reasons.  
`\tl_put_right:NV`  
`\tl_put_right:Nv`  
`\tl_put_right:No`  
`\tl_put_right:Nx`  
`\tl_put_right:cn`  
`\tl_put_right:cV`  
`\tl_put_right:cv`  
`\tl_put_right:cx`  
`\tl_gput_right:Nn`  
`\tl_gput_right:NV`  
`\tl_gput_right:Nv`  
`\tl_gput_right:No`  
`\tl_gput_right:Nx`  
`\tl_gput_right:cn`  
`\tl_gput_right:cV`  
`\tl_gput_right:cv`  
`\tl_gput_right:co`  
`\tl_gput_right:cx`

```

4127 \cs_new_protected:Npn \tl_put_right:Nn #1#2 {
4128   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4129 }
4130 \cs_new_protected:Npn \tl_put_right:NV #1#2 {
4131   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4132 }
4133 \cs_new_protected:Npn \tl_put_right:Nv #1#2 {
4134   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4135 }
4136 \cs_new_protected:Npn \tl_put_right:Nx #1#2 {
4137   \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 }
4138 }
4139 \cs_new_protected:Npn \tl_put_right:No #1#2 {
4140   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4141 }
4142 \cs_new_protected:Npn \tl_gput_right:Nn #1#2 {
4143   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }

```

```

4144 }
4145 \cs_new_protected:Npn \tl_gput_right:NV #1#2 {
4146   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4147 }
4148 \cs_new_protected:Npn \tl_gput_right:Nv #1#2 {
4149   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4150 }
4151 \cs_new_protected:Npn \tl_gput_right:No #1#2 {
4152   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4153 }
4154 \cs_new_protected:Npn \tl_gput_right:Nx #1#2 {
4155   \cs_gset_nopar:Npx #1 { \exp_not:o #1 #2 }
4156 }
4157 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4158 \cs_generate_variant:Nn \tl_put_right:NV { c }
4159 \cs_generate_variant:Nn \tl_put_right:Nv { c }
4160 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4161 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4162 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4163 \cs_generate_variant:Nn \tl_gput_right:Nv { c }
4164 \cs_generate_variant:Nn \tl_gput_right:No { c }
4165 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

\tl\_put\_left:Nn Adding to the left is basically the same as putting on the right.

```

\tl_put_left:NV
\tl_put_left:Nv
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:cv
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:Nv
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn
\tl_gput_left:cV
\tl_gput_left:cv
\tl_gput_left:cx
4166 \cs_new_protected:Npn \tl_put_left:Nn #1#2 {
4167   \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4168 }
4169 \cs_new_protected:Npn \tl_put_left:NV #1#2 {
4170   \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4171 }
4172 \cs_new_protected:Npn \tl_put_left:Nv #1#2 {
4173   \cs_set_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4174 }
4175 \cs_new_protected:Npn \tl_put_left:Nx #1#2 {
4176   \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 }
4177 }
4178 \cs_new_protected:Npn \tl_put_left:No #1#2 {
4179   \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4180 }
4181 \cs_new_protected:Npn \tl_gput_left:Nn #1#2 {
4182   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4183 }
4184 \cs_new_protected:Npn \tl_gput_left:NV #1#2 {
4185   \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4186 }
4187 \cs_new_protected:Npn \tl_gput_left:Nv #1#2 {
4188   \cs_gset_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4189 }
4190 \cs_new_protected:Npn \tl_gput_left:No #1#2 {

```

```

4191 \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4192 }
4193 \cs_new_protected:Npn \tl_gput_left:Nx #1#2 {
4194 \cs_gset_nopar:Npx #1 { #2 \exp_not:o #1 }
4195 }
4196 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4197 \cs_generate_variant:Nn \tl_put_left:Nv { c }
4198 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4199 \cs_generate_variant:Nn \tl_gput_left:Nx { c }
4200 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4201 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
4202 \cs_generate_variant:Nn \tl_gput_left:Nx { c }
4203 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

`\tl_gset:Nc` These two functions are included because they are necessary in Denys' implementations.  
`\tl_set:Nc` The `:Nc` convention (see the expansion module) is very unusual at first sight, but it works nicely over all modules, so we would like to keep it.

Construct a control sequence on the fly from `#2` and save it in `#1`.

```

4204 \cs_new_protected_nopar:Npn \tl_gset:Nc {
4205 <*check>
4206 \pref_global_chk:
4207 </check>
4208 <-check> \pref_global:D
4209 \tl_set:Nc}

```

`\pref_global_chk:` will turn the variable `check` in `\tl_set:No` into a global check.

```

4210 \cs_new_protected_nopar:Npn \tl_set:Nc #1#2{\tl_set:No #1{\cs:w#2\cs_end:}}

```

## 103.2 Variables and constants

`\c_job_name_tl` Inherited from the `expl3` name for the primitive: this needs to actually contain the text of the jobname rather than the name of the primitive, of course.

```

4211 \tl_new:N \c_job_name_tl
4212 \tl_set:Nx \c_job_name_tl { \tex_jobname:D }

```

`\c_empty_tl` Two constants which are often used.

```

4213 \tl_const:Nn \c_empty_tl { }

```

`\c_space_tl` A space as a token list (as opposed to as a character).

```

4214 \tl_const:Nn \c_space_tl { ~ }

```

`\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.

```
4215 \tl_new:N \g_tmpa_tl
4216 \tl_new:N \g_tmpb_tl
```

`\l_kernel_testa_tl` Local temporaries. These are the ones for test routines. This means that one can safely use other temporaries when calling test routines.

```
4217 \tl_new:N \l_kernel_testa_tl
4218 \tl_new:N \l_kernel_testb_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.

```
4219 \tl_new:N \l_tmpa_tl
4220 \tl_new:N \l_tmpb_tl
```

`\l_kernel_tmpa_tl` These are local temporary token list variables reserved for use by the kernel. They should not be used by other modules.

```
4221 \tl_new:N \l_kernel_tmpa_tl
4222 \tl_new:N \l_kernel_tmpb_tl
```

### 103.3 Predicates and conditionals

We also provide a few conditionals, both in expandable form (with `\c_true_bool`) and in ‘brace-form’, the latter are denoted by **TF** at the end, as explained elsewhere.

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

`\tl_if_empty_p:c`  
`\tl_if_empty:N $\underline{TF}$`   
`\tl_if_empty:c $\underline{TF}$`

```
4223 \prg_set_conditional:Npnn \tl_if_empty:N #1 {p,TF,T,F} {
4224   \if_meaning:w #1 \c_empty_tl
4225   \prg_return_true: \else: \prg_return_false: \fi:
4226 }
4227 \cs_generate_variant:Nn \tl_if_empty_p:N {c}
4228 \cs_generate_variant:Nn \tl_if_empty:N $\underline{TF}$  {c}
4229 \cs_generate_variant:Nn \tl_if_empty:NT {c}
4230 \cs_generate_variant:Nn \tl_if_empty:N $\underline{F}$  {c}
```

`\tl_if_eq_p:NN` Returns `\c_true_bool` iff the two token list variables are equal.

`\tl_if_eq_p:Nc`  
`\tl_if_eq_p:cN`  
`\tl_if_eq_p:cc`  
`\tl_if_eq:NN $\underline{TF}$`   
`\tl_if_eq:Nc $\underline{TF}$`   
`\tl_if_eq:cN $\underline{TF}$`   
`\tl_if_eq:cc $\underline{TF}$`

```
4231 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 {p,TF,T,F} {
4232   \if_meaning:w #1 #2 \prg_return_true: \else: \prg_return_false: \fi:
4233 }
```

```

4234 \cs_generate_variant:Nn \tl_if_eq_p:NN {Nc,c,cc}
4235 \cs_generate_variant:Nn \tl_if_eq:NNTF {Nc,c,cc}
4236 \cs_generate_variant:Nn \tl_if_eq:NNT {Nc,c,cc}
4237 \cs_generate_variant:Nn \tl_if_eq:NNF {Nc,c,cc}

```

`\tl_if_eq:nnTF` A simple store and compare routine.

```

\l_tl_tmpa_tl
\l_tl_tmpb_tl
4238 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF } {
4239   \group_begin:
4240     \tl_set:Nn \l_tl_tmpa_tl {#1}
4241     \tl_set:Nn \l_tl_tmpb_tl {#2}
4242     \tex_ifx:D \l_tl_tmpa_tl \l_tl_tmpb_tl
4243     \group_end:
4244     \prg_return_true:
4245   \tex_else:D
4246     \group_end:
4247     \prg_return_false:
4248   \tex_fi:D
4249 }
4250 \tl_new:N \l_tl_tmpa_tl
4251 \tl_new:N \l_tl_tmpb_tl

```

`\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:`  
`\tl_if_empty:nTF` 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.

```

4252 \prg_new_conditional:Npnn \tl_if_empty:n #1 {p,TF,T,F} {
4253   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4254   \prg_return_true: \else: \prg_return_false: \fi:
4255 }
4256 \cs_generate_variant:Nn \tl_if_empty_p:n {V}
4257 \cs_generate_variant:Nn \tl_if_empty:nTF {V}
4258 \cs_generate_variant:Nn \tl_if_empty:nT {V}
4259 \cs_generate_variant:Nn \tl_if_empty:nF {V}
4260 \cs_generate_variant:Nn \tl_if_empty_p:n {o}
4261 \cs_generate_variant:Nn \tl_if_empty:nTF {o}
4262 \cs_generate_variant:Nn \tl_if_empty:nT {o}
4263 \cs_generate_variant:Nn \tl_if_empty:nF {o}

```

`\tl_if_blank_p:n` This is based on the answers in “Around the Bend No 2” but is safer as the tests listed there all have one small flaw: If the input in the test is two tokens with the same meaning as the internal delimiter, they will fail since one of them is mistaken for the actual delimiter. In our version below we make sure to pass the input through `\tl_to_str:n`

```

\tl_if_blank:VTF
\tl_if_blank:oTF

```

`\tl_if_blank_p_aux:w`

which ensures that all the tokens are converted to catcode 12. However we use an `a` with catcode 11 as delimiter so we can *never* get into the same problem as the solutions in “Around the Bend No 2”.

```

4264 \prg_new_conditional:Npnn \tl_if_blank:n #1 {p,TF,T,F} {
4265   \exp_after:wN \tl_if_blank_p_aux:w \tl_to_str:n {#1} aa..\q_stop
4266 }
4267 \cs_new:Npn \tl_if_blank_p_aux:w #1#2 a #3#4 \q_stop {
4268   \if_meaning:w #3 #4 \prg_return_true: \else: \prg_return_false: \fi:
4269 }
4270 \cs_generate_variant:Nn \tl_if_blank_p:n {V}
4271 \cs_generate_variant:Nn \tl_if_blank:nTF {V}
4272 \cs_generate_variant:Nn \tl_if_blank:nT {V}
4273 \cs_generate_variant:Nn \tl_if_blank:nF {V}
4274 \cs_generate_variant:Nn \tl_if_blank_p:n {o}
4275 \cs_generate_variant:Nn \tl_if_blank:nTF {o}
4276 \cs_generate_variant:Nn \tl_if_blank:nT {o}
4277 \cs_generate_variant:Nn \tl_if_blank:nF {o}

```

`\tl_if_single:nTF` If the argument is a single token. ‘Space’ is considered ‘true’.

`\tl_if_single_p:n`

```

4278 \prg_new_conditional:Nnn \tl_if_single:n {p,TF,T,F} {
4279   \tl_if_empty:nTF {#1}
4280     {\prg_return_false:}
4281     {
4282       \tl_if_blank:nTF {#1}
4283       {\prg_return_true:}
4284       {
4285         \tl_if_single_aux:w #1 \q_stop
4286       }
4287     }
4288 }

```

Use `\exp_after:wN` below I know what I’m doing. Use `\exp_args:NV` or `\exp_args_unbraced:NV` for more flexibility in your own code.

```

4289 \prg_new_conditional:Nnn \tl_if_single:N {p,TF,T,F} {
4290   \tl_if_empty:NTF #1
4291     {\prg_return_false:}
4292     {
4293       \exp_after:wN \tl_if_blank:nTF #1
4294       {\prg_return_true:}
4295       {
4296         \exp_after:wN \tl_if_single_aux:w #1 \q_stop
4297       }
4298     }
4299 }

4300 \cs_new:Npn \tl_if_single_aux:w #1#2 \q_stop {
4301   \tl_if_empty:nTF {#2} \prg_return_true: \prg_return_false:
4302 }

```

## 103.4 Working with the contents of token lists

`\tl_to_lowercase:n` Just some names for a few primitives.

`\tl_to_uppercase:n`

4303 `\cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D`

4304 `\cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D`

`\tl_to_str:n` Another name for a primitive.

4305 `\cs_new_eq:NN \tl_to_str:n \etex_detokenize:D`

`\tl_to_str:N` These functions return the replacement text of a token list as a string list with all characters catcoded to ‘other’.

`\tl_to_str:c`

`\tl_to_str_aux:w`

4306 `\cs_new_nopar:Npn \tl_to_str:N {\exp_after:wN\tl_to_str_aux:w`

4307 `\token_to_meaning:N}`

4308 `\cs_new_nopar:Npn \tl_to_str_aux:w #1>{}`

4309 `\cs_generate_variant:Nn \tl_to_str:N {c}`

`\tl_map_function:nN` Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

`\tl_map_function:NN`

`\tl_map_function:cN`

`\tl_map_function_aux:NN`

4310 `\cs_new:Npn \tl_map_function:nN #1#2{`

4311 `\tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop`

4312 `}`

4313 `\cs_new_nopar:Npn \tl_map_function:NN #1#2{`

4314 `\exp_after:wN \tl_map_function_aux:Nn`

4315 `\exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop`

4316 `}`

4317 `\cs_new:Npn \tl_map_function_aux:Nn #1#2{`

4318 `\quark_if_recursion_tail_stop:n{#2}`

4319 `#1{#2} \tl_map_function_aux:Nn #1`

4320 `}`

4321 `\cs_generate_variant:Nn \tl_map_function:NN {cN}`

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the

`\tl_map_inline:Nn` counter `\g_tl_inline_level_int` to make them nestable. We can also make use of

`\tl_map_inline:cn` `\tl_map_function:Nn` from before.

`\tl_map_inline_aux:n`

`\g_tl_inline_level_int`

4322 `\cs_new_protected:Npn \tl_map_inline:nn #1#2{`

4323 `\int_gincr:N \g_tl_inline_level_int`

4324 `\cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}`

4325 `##1{#2}`

4326 `\exp_args:Nc \tl_map_function_aux:Nn`

4327 `{tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}`

4328 `#1 \q_recursion_tail\q_recursion_stop`

4329 `\int_gdecr:N \g_tl_inline_level_int`

4330 `}`

```

4331 \cs_new_protected:Npn \tl_map_inline:Nn #1#2{
4332   \int_gincr:N \g_tl_inline_level_int
4333   \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4334   ##1{#2}
4335   \exp_last_unbraced:NcV \tl_map_function_aux:Nn
4336   {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4337   #1 \q_recursion_tail\q_recursion_stop
4338   \int_gdecr:N \g_tl_inline_level_int
4339 }
4340 \cs_generate_variant:Nn \tl_map_inline:Nn {c}

```

\tl\_map\_variable:nNn \tl\_map\_variable:nNn *<token list>* *<temp>* *<action>* assigns *<temp>* to each element and executes *<action>*.  
\tl\_map\_variable:NNn  
\tl\_map\_variable:cNn

```

4341 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3{
4342   \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop
4343 }

```

Next really has to be v/V args

```

4344 \cs_new_protected_nopar:Npn \tl_map_variable:NNn {\exp_args:No \tl_map_variable:nNn}
4345 \cs_generate_variant:Nn \tl_map_variable:NNn {c}

```

\tl\_map\_variable\_aux:Nnn The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```

4346 \cs_new_protected:Npn \tl_map_variable_aux:Nnn #1#2#3{
4347   \tl_set:Nn #1{#3}
4348   \quark_if_recursion_tail_stop:N #1
4349   #2 \tl_map_variable_aux:Nnn #1{#2}
4350 }

```

\tl\_map\_break: The break statement.

```

4351 \cs_new_eq:NN \tl_map_break: \use_none_delimit_by_q_recursion_stop:w

```

\tl\_reverse:n Reversal of a token list is done by taking one token at a time and putting it in front of the ones before it.

```

\tl_reverse:V
\tl_reverse:o
\tl_reverse_aux:nN
4352 \cs_new:Npn \tl_reverse:n #1{
4353   \tl_reverse_aux:nN {} #1 \q_recursion_tail\q_recursion_stop
4354 }
4355 \cs_new:Npn \tl_reverse_aux:nN #1#2{
4356   \quark_if_recursion_tail_stop_do:nn {#2}{ #1 }
4357   \tl_reverse_aux:nN {#2#1}
4358 }
4359 \cs_generate_variant:Nn \tl_reverse:n {V,o}

```

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which in turn is removed by the `f` expansion which comes to a halt.

```

4360 \cs_new_protected_nopar:Npn \tl_reverse:N #1 {
4361   \tl_set:Nf #1 { \tl_reverse:o { #1 \exp_stop_f: } }
4362 }

```

`\tl_elt_count:n` Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. `\tl_elt_count_aux:n` grabs the element and replaces it by +1. The 0 to ensure it works on an empty list.

`\tl_elt_count:V`

`\tl_elt_count:o`

`\tl_elt_count:N`

```

4363 \cs_new:Npn \tl_elt_count:n #1{
4364   \int_eval:n {
4365     0 \tl_map_function:nN {#1} \tl_elt_count_aux:n
4366   }
4367 }
4368 \cs_generate_variant:Nn \tl_elt_count:n {V,o}
4369 \cs_new_nopar:Npn \tl_elt_count:N #1{
4370   \int_eval:n {
4371     0 \tl_map_function:NN #1 \tl_elt_count_aux:n
4372   }
4373 }

```

`\tl_num_elt_count_aux:n` Helper function for counting elements in a token list.

```

4374 \cs_new:Npn \tl_elt_count_aux:n #1 { + 1 }

```

`\tl_set_rescan:Nnn` These functions store the `{⟨token list⟩}` in `⟨tl var.⟩` after redefining catcodes, etc., in argument #2.

`\tl_gset_rescan:Nnn`

`\tl_set_rescan:Nno` #1 : `⟨tl var.⟩`

`\tl_gset_rescan:Nno` #2 : `{⟨catcode setup, etc.⟩}`

#3 : `{⟨token list⟩}`

```

4375 \cs_new_protected:Npn \tl_set_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_set:Nn }
4376 \cs_new_protected:Npn \tl_gset_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
4377 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno }
4378 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno }

```

`\tl_set_rescan_aux:NNnn` This macro uses a trick to extract an unexpanded token list after it's rescanned with `\etex_scantokens:D`. This technique was first used (as far as I know) by Heiko Oberdiek in his `catchfile` package, albeit for real files rather than the 'fake' `\scantokens` one.

The basic problem arises because `\etex_scantokens:D` emulates a file read, which inserts an EOF marker into the expansion; the simplistic

`\exp_args:Nno \cs_set:Npn \tmp:w { \etex_scantokens:D {some text} }` unfortunately doesn't work, calling the error:

! File ended while scanning definition of `\tmp:w`.

(LuaTeX works around this problem with its `\scantextokens` primitive.)

Usually, we'd define `\etex_everyeof:D` to be `\exp_not:N` to gobble the EOF marker, but since we're not expanding the token list, it gets left in there and we have the same basic problem.

Instead, we define `\etex_everyeof:D` to contain a marker that's impossible to occur within the scanned text; that is, the same char twice with different catcodes. (For some reason, we *don't* need to insert a `\exp_not:N` token after it to prevent the EOF marker to expand. Anyone know why?)

A helper function is can be used to save the token list delimited by the special marker, keeping the catcode redefinitions hidden away in a group.

`\c_two_ats_with_two_catcodes_tl` A tl with two @ characters with two different catcodes. Used as a special marker for delimited text.

```

4379 \group_begin:
4380   \tex_lccode:D '\A = '\@ \scan_stop:
4381   \tex_lccode:D '\B = '\@ \scan_stop:
4382   \tex_catcode:D '\A = 8 \scan_stop:
4383   \tex_catcode:D '\B = 3 \scan_stop:
4384   \tl_to_lowercase:n {
4385     \group_end:
4386     \tl_const:Nn \c_two_ats_with_two_catcodes_tl { A B }
4387   }

```

```

#1 : \tl_set function
#2 : <tl var.>
#3 : {(catcode setup, etc.)}
#4 : {(token list)}

```

Note that if you change `\etex_everyeof:D` in #3 then you'd better do it correctly!

```

4388 \cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4 {
4389   \group_begin:
4390     \toks_set:NV \etex_everyeof:D \c_two_ats_with_two_catcodes_tl
4391     \tex_endlinechar:D = \c_minus_one
4392     #3
4393     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
4394     \exp_args:NNNV
4395     \group_end:
4396     #1 #2 \l_tmpa_tl
4397   }

```

`\tl_rescan_aux:w`

```

4398 \exp_after:wN \cs_set:Npn
4399 \exp_after:wN \tl_rescan_aux:w
4400 \exp_after:wN #
4401 \exp_after:wN 1 \c_two_ats_with_two_catcodes_tl {
4402   \tl_set:Nn \l_tmpa_tl {#1}
4403 }

```

`\tl_set_rescan:Nnx` These functions store the full expansion of  $\{\langle token\ list\rangle\}$  in  $\langle tl\ var.\rangle$  after redefining catcodes, etc., in argument #2.

#1 :  $\langle tl\ var.\rangle$   
 #2 :  $\{\langle catcode\ setup,\ etc.\rangle\}$   
 #3 :  $\{\langle token\ list\rangle\}$

The expanded versions are much simpler because the `\etex_scantokens:D` can occur within the expansion.

```

4404 \cs_new_protected:Npn \tl_set_rescan:Nnx #1#2#3 {
4405   \group_begin:
4406     \etex_everyeof:D { \exp_not:N }
4407     \tex_endlinechar:D = \c_minus_one
4408     #2
4409     \tl_set:Nx \l_kernel_tmpa_tl { \etex_scantokens:D {#3} }
4410     \exp_args:NNNV
4411   \group_end:
4412   \tl_set:Nn #1 \l_kernel_tmpa_tl
4413 }
```

Globally is easier again:

```

4414 \cs_new_protected:Npn \tl_gset_rescan:Nnx #1#2#3 {
4415   \group_begin:
4416     \etex_everyeof:D { \exp_not:N }
4417     \tex_endlinechar:D = \c_minus_one
4418     #2
4419     \tl_gset:Nx #1 { \etex_scantokens:D {#3} }
4420   \group_end:
4421 }
```

`\tl_rescan:nn` The inline wrapper for `\etex_scantokens:D`.

#1 : Catcode changes (etc.)  
 #2 : Token list to re-tokenise

```

4422 \cs_new_protected:Npn \tl_rescan:nn #1#2 {
4423   \group_begin:
4424     \toks_set:NV \etex_everyeof:D \c_two_at_with_two_catcodes_tl
4425     \tex_endlinechar:D = \c_minus_one
4426     #1
4427     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
4428   \exp_args:NV \group_end:
4429   \l_tmpa_tl
4430 }
```

## 103.5 Checking for and replacing tokens

`\tl_if_in:NnTF` See the replace functions for further comments. In this part we don't care too much about brace stripping since we are not interested in passing on the tokens which are split

off in the process.

```

4431 \prg_new_protected_conditional:Npnn \tl_if_in:Nn #1#2 {TF,T,F} {
4432   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
4433     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
4434   }
4435   \exp_after:wN \tl_tmp:w #1 #2 \q_no_value \q_stop
4436 }
4437 \cs_generate_variant:Nn \tl_if_in:NnTF {c}
4438 \cs_generate_variant:Nn \tl_if_in:NnT {c}
4439 \cs_generate_variant:Nn \tl_if_in:NnF {c}

```

\tl\_if\_in:nnTF  
 \tl\_if\_in:VnTF  
 \tl\_if\_in:onTF

```

4440 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 {TF,T,F} {
4441   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
4442     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
4443   }
4444   \tl_tmp:w #1 #2 \q_no_value \q_stop
4445 }
4446 \cs_generate_variant:Nn \tl_if_in:nnTF {V}
4447 \cs_generate_variant:Nn \tl_if_in:nnT {V}
4448 \cs_generate_variant:Nn \tl_if_in:nnF {V}
4449 \cs_generate_variant:Nn \tl_if_in:nnTF {o}
4450 \cs_generate_variant:Nn \tl_if_in:nnT {o}
4451 \cs_generate_variant:Nn \tl_if_in:nnF {o}

```

\\_l\_tl\_replace\_tl  
 \tl\_replace\_in:Nnn  
 \tl\_replace\_in:cnn  
 \tl\_greplace\_in:Nnn  
 \tl\_greplace\_in:cnn  
 \\_tl\_replace\_in\_aux:NNnn

The concept here is that only the first occurrence should be replaced. The first step is to define an auxiliary which will match the appropriate item, with a trailing marker. If the last token is the marker there is nothing to do, otherwise replace the token and clean up (hence the second use of `\_tl_tmp:w`). To prevent loosing braces or spaces there are a couple of empty groups and the strange-looking `\use:n`.

```

4452 \tl_new:N \_l_tl_replace_tl
4453 \cs_new_protected_nopar:Npn \tl_replace_in:Nnn {
4454   \_tl_replace_in_aux:NNnn \tl_set_eq:NN
4455 }
4456 \cs_new_protected:Npn \_tl_replace_in_aux:NNnn #1#2#3#4 {
4457   \cs_set:Npn \_tl_tmp:w ##1 #3 ##2 \q_stop
4458   {
4459     \quark_if_no_value:nF {##2}
4460     {
4461       \tl_set:No \_l_tl_replace_tl { ##1 #4 }
4462       \cs_set:Npn \_tl_tmp:w #####1 #3 \q_no_value {
4463         \tl_put_right:No \_l_tl_replace_tl {#####1}
4464       }
4465       \_tl_tmp:w \prg_do_nothing: ##2
4466       #1 #2 \_l_tl_replace_tl
4467     }

```

```

4468     }
4469     \use:n
4470     {
4471         \exp_after:wN \_tl_tmp:w \exp_after:wN
4472         \prg_do_nothing:
4473     }
4474     #2 #3 \q_no_value \q_stop
4475 }
4476 \cs_new_protected_nopar:Npn \tl_greplace_in:Nnn {
4477     \_tl_replace_in_aux:NNnn \tl_gset_eq:NN
4478 }
4479 \cs_generate_variant:Nn \tl_replace_in:Nnn { c }
4480 \cs_generate_variant:Nn \tl_greplace_in:Nnn { c }

```

```

\tl_replace_all_in:Nnn
\tl_replace_all_in:Nnn
\tl_greplace_all_in:cnn
\tl_greplace_all_in:cnn
\_tl_replace_all_in_aux:NNnn

```

A similar approach here but with a loop built in.

```

4481 \cs_new_protected_nopar:Npn \tl_replace_all_in:Nnn {
4482     \_tl_replace_all_in_aux:NNnn \tl_set_eq:NN
4483 }
4484 \cs_new_protected_nopar:Npn \tl_greplace_all_in:Nnn {
4485     \_tl_replace_all_in_aux:NNnn \tl_gset_eq:NN
4486 }
4487 \cs_new_protected:Npn \_tl_replace_all_in_aux:NNnn #1#2#3#4 {
4488     \tl_clear:N \_l_tl_replace_tl
4489     \cs_set:Npn \_tl_tmp:w ##1 #3 ##2 \q_stop
4490     {
4491         \quark_if_no_value:nTF {##2}
4492         { \tl_put_right:No \_l_tl_replace_tl {##1} }
4493         {
4494             \tl_put_right:No \_l_tl_replace_tl { ##1 #4 }
4495             \_tl_tmp:w \prg_do_nothing: ##2 \q_stop
4496         }
4497     }
4498     \use:n
4499     {
4500         \exp_after:wN \_tl_tmp:w \exp_after:wN
4501         \prg_do_nothing:
4502     }
4503     #2 #3 \q_no_value \q_stop
4504     #1 #2 \_l_tl_replace_tl
4505 }
4506 \cs_generate_variant:Nn \tl_replace_all_in:Nnn { c }
4507 \cs_generate_variant:Nn \tl_greplace_all_in:Nnn { c }

```

```

\tl_remove_in:Nn
\tl_remove_in:cn
\tl_gremove_in:Nn
\tl_gremove_in:cn

```

Next comes a series of removal functions. I have just implemented them as subcases of the replace functions for now (I'm lazy).

```

4508 \cs_new_protected:Npn \tl_remove_in:Nn #1#2{\tl_replace_in:Nnn #1{#2}{}}
4509 \cs_new_protected:Npn \tl_gremove_in:Nn #1#2{\tl_greplace_in:Nnn #1{#2}{}}

```

```

4510 \cs_generate_variant:Nn \tl_remove_in:Nn {cn}
4511 \cs_generate_variant:Nn \tl_gremove_in:Nn {cn}

```

\tl\_remove\_all\_in:Nn Same old, same old.

```

\tl_remove_all_in:cn
\tl_remove_all_in:cn
\tl_gremove_all_in:Nn
\tl_gremove_all_in:cn
4512 \cs_new_protected:Npn \tl_remove_all_in:Nn #1#2{
4513   \tl_replace_all_in:Nnn #1{#2}{f}
4514 }
4515 \cs_new_protected:Npn \tl_gremove_all_in:Nn #1#2{
4516   \tl_greplace_all_in:Nnn #1{#2}{f}
4517 }
4518 \cs_generate_variant:Nn \tl_remove_all_in:Nn {cn}
4519 \cs_generate_variant:Nn \tl_gremove_all_in:Nn {cn}

```

## 103.6 Heads or tails?

\tl\_head:n These functions pick up either the head or the tail of a list. \tl\_head\_iii:n returns the first three items on a list.

```

\tl_head:n
\tl_head:V
\tl_head:v
4520 \cs_new:Npn \tl_head:n #1{\tl_head:w #1\q_stop}
\tl_head_i:n
4521 \cs_new_eq:NN \tl_head_i:n \tl_head:n
\tl_tail:n
4522 \cs_new:Npn \tl_tail:n #1{\tl_tail:w #1\q_stop}
\tl_tail:V
4523 \cs_generate_variant:Nn \tl_tail:n {f}
\tl_tail:v
4524 \cs_new:Npn \tl_head_iii:n #1{\tl_head_iii:w #1\q_stop}
\tl_tail:f
4525 \cs_generate_variant:Nn \tl_head_iii:n {f}
\tl_head_iii:n
4526 \cs_new:Npn \tl_head:w #1#2\q_stop{#1}
\tl_head_iii:f
4527 \cs_new_eq:NN \tl_head_i:w \tl_head:w
\tl_head:w
4528 \cs_new:Npn \tl_tail:w #1#2\q_stop{#2}
\tl_head_i:w
4529 \cs_new:Npn \tl_head_iii:w #1#2#3#4\q_stop{#1#2#3}
\tl_tail:w
4530 \cs_generate_variant:Nn \tl_head:n { V }
\tl_head_iii:w
4531 \cs_generate_variant:Nn \tl_head:n { v }
4532 \cs_generate_variant:Nn \tl_tail:n { V }
4533 \cs_generate_variant:Nn \tl_tail:n { v }

```

\tl\_if\_head\_eq\_meaning\_p:nN When we want to check if the first token of a list equals something specific it is usually either to see if it is a control sequence or a character. Hence we make two different functions as the internal test is different. \tl\_if\_head\_meaning\_eq:nNTF uses \if\_meaning:w and will consider the tokens  $b_{11}$  and  $b_{12}$  different. \tl\_if\_head\_char\_eq:nNTF on the other hand only compares character codes so would regard  $b_{11}$  and  $b_{12}$  as equal but would also regard two primitives as equal.

```

\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_p:fN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF
4534 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 {p,TF,T,F} {
4535   \exp_after:wN \if_meaning:w \tl_head:w #1 \q_stop #2
4536   \prg_return_true: \else: \prg_return_false: \fi:
4537 }

```

For the charcode and catcode versions we insert \exp\_not:N in front of both tokens. If you need them to expand fully as T<sub>E</sub>X does itself with these you can use an f type expansion.

```

4538 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 {p,TF,T,F} {
4539   \exp_after:wN \if:w \exp_after:wN \exp_not:N
4540     \tl_head:w #1 \q_stop \exp_not:N #2
4541   \prg_return_true: \else: \prg_return_false: \fi:
4542 }

```

Actually the default is already an f type expansion.

```

4543 %% \cs_new:Npn \tl_if_head_eq_charcode_p:fN #1#2{
4544   \exp_after:wN \if_charcode:w \tl_head:w #1 \q_stop \exp_not:N #2
4545   \c_true_bool
4546   \else:
4547   \c_false_bool
4548   \fi:
4549 }
4550 %% \def_long_test_function_new:npn {tl_if_head_eq_charcode:fN}#1#2{
4551 %%   \if_predicate:w \tl_if_head_eq_charcode_p:fN {#1}#2}

```

These :fN variants are broken; temporary patch:

```

4552 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN {f}
4553 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF {f}
4554 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT {f}
4555 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF {f}

```

And now catcodes:

```

4556 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1#2 {p,TF,T,F} {
4557   \exp_after:wN \if_catcode:w \exp_after:wN \exp_not:N
4558     \tl_head:w #1 \q_stop \exp_not:N #2
4559   \prg_return_true: \else: \prg_return_false: \fi:
4560 }

```

`\_tl\_check\_exists:N` When used as a package, there is an option to be picky and to check definitions exist. The message text for this is created later, as the mechanism is not yet in place.

```

4561 <*package>
4562 \tex_ifodd:D \@l@expl@check@declarations@bool \scan_stop:
4563 \cs_set_protected:Npn \_tl_check_exists:N #1
4564 {
4565   \cs_if_exist:NF #1
4566   {
4567     \msg_kernel_error:nnx { check } { non-declared-variable }
4568     { \token_to_str:N #1 }
4569   }
4570 }
4571 \cs_set_protected:Npn \tl_set:Nn #1#2
4572 {
4573   \_tl_check_exists:N #1
4574   \cs_set_nopar:Npx #1 { \exp_not:n {#2} }
4575 }

```

```

4576 \cs_set_protected:Npn \tl_set:Nx #1#2
4577 {
4578   \_tl_check_exists:N #1
4579   \cs_set_nopar:Npx #1 {#2}
4580 }
4581 \cs_set_protected:Npn \tl_gset:Nn #1#2
4582 {
4583   \_tl_check_exists:N #1
4584   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4585 }
4586 \cs_set_protected:Npn \tl_gset:Nx #1#2
4587 {
4588   \_tl_check_exists:N #1
4589   \cs_gset_nopar:Npx #1 {#2}
4590 }
4591 \cs_set_protected:Npn \tl_set_eq:NN #1#2
4592 {
4593   \_tl_check_exists:N #1
4594   \_tl_check_exists:N #2
4595   \cs_set_eq:NN #1 #2
4596 }
4597 \cs_set_protected:Npn \tl_gset_eq:NN #1#2
4598 {
4599   \_tl_check_exists:N #1
4600   \_tl_check_exists:N #2
4601   \cs_gset_eq:NN #1 #2
4602 }
4603 \cs_set_protected:Npn \tl_put_right:Nn #1#2 {
4604   \_tl_check_exists:N #1
4605   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4606 }
4607 \cs_set_protected:Npn \tl_put_right:NV #1#2 {
4608   \_tl_check_exists:N #1
4609   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4610 }
4611 \cs_set_protected:Npn \tl_put_right:Nv #1#2 {
4612   \_tl_check_exists:N #1
4613   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4614 }
4615 \cs_set_protected:Npn \tl_put_right:No #1#2 {
4616   \_tl_check_exists:N #1
4617   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4618 }
4619 \cs_set_protected:Npn \tl_put_right:Nx #1#2 {
4620   \_tl_check_exists:N #1
4621   \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 }
4622 }
4623 \cs_set_protected:Npn \tl_gput_right:Nn #1#2 {
4624   \_tl_check_exists:N #1
4625   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }

```

```

4626 }
4627 \cs_set_protected:Npn \tl_gput_right:NV #1#2 {
4628   \_tl_check_exists:N #1
4629   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4630 }
4631 \cs_set_protected:Npn \tl_gput_right:Nv #1#2 {
4632   \_tl_check_exists:N #1
4633   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4634 }
4635 \cs_set_protected:Npn \tl_gput_right:Nx #1#2 {
4636   \_tl_check_exists:N #1
4637   \cs_gset_nopar:Npx #1 { \exp_not:o #1 #2 }
4638 }
4639 \cs_set_protected:Npn \tl_put_left:Nn #1#2 {
4640   \_tl_check_exists:N #1
4641   \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4642 }
4643 \cs_set_protected:Npn \tl_put_left:NV #1#2 {
4644   \_tl_check_exists:N #1
4645   \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4646 }
4647 \cs_set_protected:Npn \tl_put_left:Nv #1#2 {
4648   \_tl_check_exists:N #1
4649   \cs_set_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4650 }
4651 \cs_set_protected:Npn \tl_put_left:No #1#2 {
4652   \_tl_check_exists:N #1
4653   \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4654 }
4655 \cs_set_protected:Npn \tl_put_left:Nx #1#2 {
4656   \_tl_check_exists:N #1
4657   \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 }
4658 }
4659 \cs_set_protected:Npn \tl_gput_left:Nn #1#2 {
4660   \_tl_check_exists:N #1
4661   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4662 }
4663 \cs_set_protected:Npn \tl_gput_left:NV #1#2 {
4664   \_tl_check_exists:N #1
4665   \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4666 }
4667 \cs_set_protected:Npn \tl_gput_left:Nv #1#2 {
4668   \_tl_check_exists:N #1
4669   \cs_gset_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4670 }
4671 \cs_set_protected:Npn \tl_gput_left:Nx #1#2 {
4672   \_tl_check_exists:N #1
4673   \cs_gset_nopar:Npx #1 { #2 \exp_not:o #1 }
4674 }
4675 \tex_fi:D

```

```
4676 </package>
```

Show token usage:

```
4677 <*showmemory>
4678 \showMemUsage
4679 </showmemory>
```

## 104 l3toks implementation

We start by ensuring that the required packages are loaded.

```
4680 <*package>
4681 \ProvidesExplPackage
4682   {\filename}{\filedate}{\fileversion}{\filedescription}
4683 \package_check_loaded_expl:
4684 </package>
4685 <*initex | package>
```

### 104.1 Allocation and use

`\toks_new:N` Allocates a new token register.

```
\toks_new:c
4686 <*initex>
4687 \alloc_new:nnnN {toks} \c_zero \c_max_register_int \tex_toksdef:D
4688 </initex>

4689 <*package>
4690 \cs_new_protected_nopar:Npn \toks_new:N #1 {
4691   \chk_if_free_cs:N #1
4692   \newtoks #1
4693 }
4694 </package>

4695 \cs_generate_variant:Nn \toks_new:N {c}
```

`\toks_use:N` This function returns the contents of a token register.

```
\toks_use:c
4696 \cs_new_eq:NN \toks_use:N \tex_the:D
4697 \cs_generate_variant:Nn \toks_use:N {c}
```

`\toks_set:Nn` `\toks_set:Nn<toks><stuff>` stores `<stuff>` without expansion in `<toks>`. `\toks_set:No` and `\toks_set:Nv` expand `<stuff>` once and fully.

```
\toks_set:Nv
\toks_set:No
\toks_set:Nx
\toks_set:Nf
\toks_set:cn
\toks_set:co
\toks_set:cV
\toks_set:cv
\toks_set:cx
\toks_set:cf

4698 <*check>
4699 \cs_new_protected_nopar:Npn \toks_set:Nn #1 { \chk_local:N #1 #1 }
4700 \cs_generate_variant:Nn \toks_set:Nn {No,Nf}
4701 </check>
```

If we don't check if  $\langle toks \rangle$  is a local register then the `\toks_set:Nn` function has nothing to do. We implement `\toks_set:No/d/f` by hand when not checking because this is going to be used *extensively* in keyval processing! TODO: (Will) Can we get some numbers published on how necessary this is? On the other hand I'm happy to believe Morten :)

```

4702 <!*check>
4703 \cs_new_eq:NN \toks_set:Nn \prg_do_nothing:
4704 \cs_new_protected:Npn \toks_set:Nv #1#2 {
4705   #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:N #2 }
4706 }
4707 \cs_new_protected:Npn \toks_set:Nv #1#2 {
4708   #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:c {#2} }
4709 }
4710 \cs_new_protected:Npn \toks_set:No #1#2 { #1 \exp_after:wN {#2} }
4711 \cs_new_protected:Npn \toks_set:Nf #1#2 {
4712   #1 \exp_after:wN { \int_to_roman:w -'0#2 }
4713 }
4714 </!check>

4715 \cs_generate_variant:Nn \toks_set:Nn {Nx,cn,cV,cv,co,cx,cf}

```

```

\toks_gset:Nn These functions are the global variants of the above.
\toks_gset:Nv
\toks_gset:No
\toks_gset:Nx
\toks_gset:cn
\toks_gset:cV
\toks_set_eq:NN \toks_set_eq:NN<toks1><toks2> copies the contents of <toks2> in <toks1>.
\toks_gset:co
\toks_set_eq:Nc
\toks_gset:cx
\toks_set_eq:cN
\toks_set_eq:cc
\toks_gset_eq:NN
\toks_gset_eq:Nc
\toks_gset_eq:cN
\toks_gset_eq:cc
4716 <check>\cs_new_protected_nopar:Npn \toks_gset:Nn #1 { \chk_global:N #1 \pref_global:D #1 }
4717 <!check>\cs_new_eq:NN \toks_gset:Nn \pref_global:D
4718 \cs_generate_variant:Nn \toks_gset:Nn {NV,No,Nx,cn,cV,co,cx}

4719 <*check>
4720 \cs_new_protected_nopar:Npn \toks_set_eq:NN #1#2 {
4721   \chk_local:N #1
4722   \chk_var_or_const:N #2
4723   #1 #2
4724 }
4725 \cs_new_protected_nopar:Npn \toks_gset_eq:NN #1#2 {
4726   \chk_global:N #1
4727   \chk_var_or_const:N #2
4728   \pref_global:D #1 #2
4729 }
4730 </check>
4731 <!*check>
4732 \cs_new_eq:NN \toks_set_eq:NN \prg_do_nothing:
4733 \cs_new_eq:NN \toks_gset_eq:NN \pref_global:D
4734 </!check>
4735 \cs_generate_variant:Nn \toks_set_eq:NN {Nc,cN,cc}
4736 \cs_generate_variant:Nn \toks_gset_eq:NN {Nc,cN,cc}

```

```

\toks_clear:N These functions clear a token register, either locally or globally.
\toks_gclear:N
\toks_clear:c
\toks_gclear:c

```

```

4737 \cs_new_protected_nopar:Npn \toks_clear:N #1 {
4738   #1\c_empty_toks
4739 <check>\chk_local_or_pref_global:N #1
4740 }

```

```

4741 \cs_new_protected_nopar:Npn \toks_gclear:N {
4742 <check> \pref_global_chk:
4743 <!check> \pref_global:D
4744 \toks_clear:N
4745 }

```

```

4746 \cs_generate_variant:Nn \toks_clear:N {c}
4747 \cs_generate_variant:Nn \toks_gclear:N {c}

```

\toks\_use\_clear:N These functions clear a token register (locally or globally) after returning the contents.

\toks\_use\_clear:c They make sure that clearing the register does not interfere with following tokens. In  
\toks\_use\_gclear:N other words, the contents of the register might operate on what follows in the input  
\toks\_use\_gclear:c stream.

```

4748 \cs_new_protected_nopar:Npn \toks_use_clear:N #1 {
4749 \exp_last_unbraced:NNV \toks_clear:N #1 #1
4750 }

```

```

4751 \cs_new_protected_nopar:Npn \toks_use_gclear:N {
4752 <check> \pref_global_chk:
4753 <!check> \pref_global:D
4754 \toks_use_clear:N
4755 }

```

```

4756 \cs_generate_variant:Nn \toks_use_clear:N {c}
4757 \cs_generate_variant:Nn \toks_use_gclear:N {c}

```

\toks\_show:N This function shows the contents of a token register on the terminal.

```

\toks_show:c
4758 \cs_new_eq:NN \toks_show:N \kernel_register_show:N
4759 \cs_generate_variant:Nn \toks_show:N {c}

```

## 104.2 Adding to token registers' contents

\toks\_put\_left:Nn \toks\_put\_left:Nn <toks><stuff> adds the tokens of *stuff* on the 'left-side' of the token  
\toks\_put\_left:NV register <toks>. \toks\_put\_left:No does the same, but expands the tokens once. We  
\toks\_put\_left:No need to look out for brace stripping so we add a token, which is then later removed.

```

\toks_put_left:Nx
\toks_put_left:cn
\toks_put_left:cV
\toks_put_left:co
4760 \cs_new_protected_nopar:Npn \toks_put_left:Nn #1 {
4761 \exp_after:wN \toks_put_left_aux:w \exp_after:wN \q_nil
4762 \toks_use:N #1 \q_stop #1
4763 }
\toks_gput_left:Nn
\toks_gput_left:NV
\toks_gput_left:No
\toks_gput_left:Nx
\toks_gput_left:cn
\toks_gput_left:cV
\toks_gput_left:co
\toks_put_left_aux:w

```

```

4764 \cs_generate_variant:Nn \toks_put_left:Nn {NV,No,Nx,cn,co,cV}

4765 \cs_new_protected_nopar:Npn \toks_gput_left:Nn {
4766 <check> \pref_global_chk:
4767 <!check> \pref_global:D
4768 \toks_put_left:Nn
4769 }

4770 \cs_generate_variant:Nn \toks_gput_left:Nn {NV,No,Nx,cn,cV,co}

```

A helper function for `\toks_put_left:Nn`. Its arguments are subsequently the tokens of `<stuff>`, the token register `<toks>` and the current contents of `<toks>`. We make sure to remove the token we inserted earlier.

```

4771 \cs_new:Npn \toks_put_left_aux:w #1\q_stop #2#3 {
4772 #2 \exp_after:wN { \use_i:nn {#3} #1 }
4773 <check> \chk_local_or_pref_global:N #2
4774 }

```

```

\toks_put_right:Nn
\toks_put_right:NV
\toks_put_right:No
\toks_put_right:Nx
\toks_put_right:cn
\toks_put_right:cV
\toks_put_right:co
\toks_gput_right:Nn
\toks_gput_right:NV
\toks_gput_right:No
\toks_gput_right:Nx
\toks_gput_right:cn
\toks_gput_right:cV
\toks_gput_right:co

```

These macros add a list of tokens to the right of a token register.

```

4775 \cs_new_protected:Npn \toks_put_right:Nn #1#2 {
4776 #1 \exp_after:wN { \toks_use:N #1 #2 }
4777 <check> \chk_local_or_pref_global:N #1
4778 }

4779 \cs_new_protected_nopar:Npn \toks_gput_right:Nn {
4780 <check> \pref_global_chk:
4781 <!check> \pref_global:D
4782 \toks_put_right:Nn
4783 }

```

A couple done by hand for speed.

```

4784 <check> \cs_generate_variant:Nn \toks_put_right:Nn {No}
4785 <!*check>
4786 \cs_new_protected:Npn \toks_put_right:NV #1#2 {
4787 #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4788 \exp_after:wN \toks_use:N \exp_after:wN #1
4789 \int_to_roman:w -'0 \exp_eval_register:N #2
4790 }
4791 }
4792 \cs_new_protected:Npn \toks_put_right:No #1#2 {
4793 #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4794 \exp_after:wN \toks_use:N \exp_after:wN #1 #2
4795 }
4796 }
4797 </!check>
4798 \cs_generate_variant:Nn \toks_put_right:Nn {Nx,cn,cV,co}
4799 \cs_generate_variant:Nn \toks_gput_right:Nn {NV,No,Nx,cn,cV,co}

```

`\toks_put_right:Nf` We implement `\toks_put_right:Nf` by hand because I think I might use it in the `l3keyval` module in which case it is going to be used a lot.

```

4800 <check>\cs_generate_variant:Nn \toks_put_right:Nn {Nf}
4801 <!*check>
4802 \cs_new_protected:Npn \toks_put_right:Nf #1#2 {
4803   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4804     \exp_after:wN \toks_use:N \exp_after:wN #1 \int_to_roman:w -'0#2
4805   }
4806 }
4807 </!check>

```

### 104.3 Predicates and conditionals

`\toks_if_empty_p:N` `\toks_if_empty:NTF<toks><true code><false code>` tests if a token register is empty and executes either `<true code>` or `<false code>`. This test had the advantage of being expandable. Otherwise one has to do an `x` type expansion in order to prevent problems with parameter tokens.

```

4808 \prg_new_conditional:Nnn \toks_if_empty:N {p,TF,T,F} {
4809   \tl_if_empty:VTF #1 {\prg_return_true:} {\prg_return_false:}
4810 }

4811 \cs_generate_variant:Nn \toks_if_empty_p:N {c}
4812 \cs_generate_variant:Nn \toks_if_empty:NTF {c}
4813 \cs_generate_variant:Nn \toks_if_empty:NT {c}
4814 \cs_generate_variant:Nn \toks_if_empty:NF {c}

```

`\toks_if_eq_p:NN` This function test whether two token registers have the same contents.

```

\toks_if_eq_p:cN
\toks_if_eq_p:Nc
\toks_if_eq_p:cc
\toks_if_eq:NNTF
\toks_if_eq:NcTF
\toks_if_eq:cNTF
\toks_if_eq:ccTF

4815 \prg_new_conditional:Nnn \toks_if_eq:NN {p,TF,T,F} {
4816   \str_if_eq:xxTF {\toks_use:N #1} {\toks_use:N #2}
4817   {\prg_return_true:} {\prg_return_false:}
4818 }
4819 \cs_generate_variant:Nn \toks_if_eq_p:NN {Nc,c,cc}
4820 \cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
4821 \cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
4822 \cs_generate_variant:Nn \toks_if_eq:NNF {Nc,c,cc}

```

### 104.4 Variables and constants

`\l_tmpa_toks` Some scratch registers ...

```

\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks

4823 \tex_toksdef:D \l_tmpa_toks = 255\scan_stop:
4824 <initex>\seq_put_right:Nn \g_toks_allocation_seq {255}
4825 \toks_new:N \l_tmpb_toks
4826 \toks_new:N \l_tmpc_toks

```

```

4827 \toks_new:N \g_tmpa_toks
4828 \toks_new:N \g_tmpb_toks
4829 \toks_new:N \g_tmpc_toks

```

`\c_empty_toks` And here is a constant, which is a (permanently) empty token register.

```

4830 \toks_new:N \c_empty_toks

```

`\l_tl_replace_toks` And here is one for tl vars. Can't define it there as the allocation isn't set up at that point.

```

4831 \toks_new:N \l_tl_replace_toks

4832 </initex | package>

```

Show token usage:

```

4833 <*showmemory>
4834 \showMemUsage
4835 </showmemory>

```

## 105 l3seq implementation

```

4836 <*package>
4837 \ProvidesExplPackage
4838   {\filename}{\filedate}{\fileversion}{\filedescription}
4839 \package_check_loaded_expl:
4840 </package>

```

A sequence is a control sequence whose top-level expansion is of the form ‘`\seq_elt:w`  $\langle text_1 \rangle$  `\seq_elt_end: ... \seq_elt:w`  $\langle text_n \rangle$  ...’. We use explicit delimiters instead of braces around  $\langle text \rangle$  to allow efficient searching for an item in the sequence.

`\seq_elt:w` We allocate the delimiters and make them errors if executed.  
`\seq_elt_end:`

```

4841 <*initex | package>
4842 \cs_new:Npn \seq_elt:w {\ERROR}
4843 \cs_new:Npn \seq_elt_end: {\ERROR}

```

### 105.1 Allocating and initialisation

`\seq_new:N` Sequences are implemented using token lists.

```

\seq_new:c
4844 \cs_new_eq:NN \seq_new:N \tl_new:N
4845 \cs_new_eq:NN \seq_new:c \tl_new:c

```

<code>\seq_clear:N</code>	Clearing a sequence is the same as clearing a token list.
<code>\seq_clear:c</code>	
<code>\seq_gclear:N</code>	4846 <code>\cs_new_eq:NN \seq_clear:N \tl_clear:N</code>
<code>\seq_gclear:c</code>	4847 <code>\cs_new_eq:NN \seq_clear:c \tl_clear:c</code>
	4848 <code>\cs_new_eq:NN \seq_gclear:N \tl_gclear:N</code>
	4849 <code>\cs_new_eq:NN \seq_gclear:c \tl_gclear:c</code>
<code>\seq_clear_new:N</code>	Clearing a sequence is the same as clearing a token list.
<code>\seq_clear_new:c</code>	
<code>\seq_gclear_new:N</code>	4850 <code>\cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N</code>
<code>\seq_gclear_new:c</code>	4851 <code>\cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c</code>
	4852 <code>\cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N</code>
	4853 <code>\cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c</code>
<code>\seq_set_eq:NN</code>	We can set one seq equal to another.
<code>\seq_set_eq:Nc</code>	
<code>\seq_set_eq:cN</code>	4854 <code>\cs_new_eq:NN \seq_set_eq:NN \cs_set_eq:NN</code>
<code>\seq_set_eq:cc</code>	4855 <code>\cs_new_eq:NN \seq_set_eq:cN \cs_set_eq:cN</code>
	4856 <code>\cs_new_eq:NN \seq_set_eq:Nc \cs_set_eq:Nc</code>
	4857 <code>\cs_new_eq:NN \seq_set_eq:cc \cs_set_eq:cc</code>
<code>\seq_gset_eq:NN</code>	And of course globally which seems to be needed far more often. <sup>11</sup>
<code>\seq_gset_eq:cN</code>	
<code>\seq_gset_eq:Nc</code>	4858 <code>\cs_new_eq:NN \seq_gset_eq:NN \cs_gset_eq:NN</code>
<code>\seq_gset_eq:cc</code>	4859 <code>\cs_new_eq:NN \seq_gset_eq:cN \cs_gset_eq:cN</code>
	4860 <code>\cs_new_eq:NN \seq_gset_eq:Nc \cs_gset_eq:Nc</code>
	4861 <code>\cs_new_eq:NN \seq_gset_eq:cc \cs_gset_eq:cc</code>
<code>\seq_concat:NNN</code>	<code>\seq_concat:NNN</code> $\langle seq\ 1 \rangle$ $\langle seq\ 2 \rangle$ $\langle seq\ 3 \rangle$ will locally assign $\langle seq\ 1 \rangle$ the concatenation of
<code>\seq_concat:ccc</code>	$\langle seq\ 2 \rangle$ and $\langle seq\ 3 \rangle$ .
	4862 <code>\cs_new_protected_nopar:Npn \seq_concat:NNN #1#2#3 {</code>
	4863 <code>\tl_set:Nx #1 { \exp_not:V #2 \exp_not:V #3 }</code>
	4864 <code>}</code>
	4865 <code>\cs_generate_variant:Nn \seq_concat:NNN {ccc}</code>
<code>\seq_gconcat:NNN</code>	<code>\seq_gconcat:NNN</code> $\langle seq\ 1 \rangle$ $\langle seq\ 2 \rangle$ $\langle seq\ 3 \rangle$ will globally assign $\langle seq\ 1 \rangle$ the concatenation
<code>\seq_gconcat:ccc</code>	of $\langle seq\ 2 \rangle$ and $\langle seq\ 3 \rangle$ .
	4866 <code>\cs_new_protected_nopar:Npn \seq_gconcat:NNN #1#2#3 {</code>
	4867 <code>\tl_gset:Nx #1 { \exp_not:V #2 \exp_not:V #3 }</code>
	4868 <code>}</code>
	4869 <code>\cs_generate_variant:Nn \seq_gconcat:NNN {ccc}</code>

---

<sup>11</sup>To save a bit of space these functions could be made identical to those from the `tl` or `clist` module.

## 105.2 Predicates and conditionals

`\seq_if_empty_p:N` A predicate which evaluates to `\c_true_bool` iff the sequence is empty.  
`\seq_if_empty_p:c`  
`\seq_if_empty:N $\overline{TF}$`  4870 `\prg_new_eq_conditional:Nnn \seq_if_empty:N \tl_if_empty:N {p,TF,T,F}`  
`\seq_if_empty:c $\overline{TF}$`  4871 `\prg_new_eq_conditional:Nnn \seq_if_empty:c \tl_if_empty:c {p,TF,T,F}`

`\seq_if_empty_err:N` Signals an error if the sequence is empty.

```
4872 \cs_new_nopar:Npn \seq_if_empty_err:N #1 {
4873   \if_meaning:w #1 \c_empty_tl
```

As I said before, I don't think we need to provide checks for this kind of error, since it is a severe internal macro package error that can not be produced by the user directly. Can it? So the next line of code should be probably removed. (Will: I have no idea what this comment means.)

```
4874   \tl_clear:N \l_kernel_testa_tl % catch prefixes
4875   \msg_kernel_bug:x {Empty-sequence-'\token_to_str:N#1'}
4876   \fi:
4877 }
```

`\seq_if_in:Nn $\overline{TF}$`  `\seq_if_in:Nn $\overline{TF}$`   $\langle seq \rangle \langle item \rangle \langle true\ case \rangle \langle false\ case \rangle$  will check whether  $\langle item \rangle$  is in  $\langle seq \rangle$   
`\seq_if_in:N $\overline{VTF}$`  and then either execute the  $\langle true\ case \rangle$  or the  $\langle false\ case \rangle$ .  $\langle true\ case \rangle$  and  $\langle false\ case \rangle$   
`\seq_if_in:cn $\overline{TF}$`  may contain incomplete `\if_charcode:w` statements.  
`\seq_if_in:c $\overline{VTF}$`   
`\seq_if_in:co $\overline{TF}$`  Note that `##2` in the definition below for `\seq_tmp:w` contains exactly one token which  
`\seq_if_in:cx $\overline{TF}$`  we can compare with `\q_no_value`.

```
4878 \prg_new_protected_conditional:Nnn \seq_if_in:Nn {TF,T,F} {
4879   \cs_set:Npn \seq_tmp:w ##1 \seq_elt:w #2 \seq_elt_end: ##2##3 \q_stop {
4880     \if_meaning:w \q_no_value ##2
4881     \prg_return_false: \else: \prg_return_true: \fi:
4882   }
4883   \exp_after:wN \seq_tmp:w #1 \seq_elt:w #2 \seq_elt_end: \q_no_value \q_stop
4884 }

4885 \cs_generate_variant:Nn \seq_if_in:Nn $\overline{TF}$  { NV, cV, co, c, cx}
4886 \cs_generate_variant:Nn \seq_if_in:NnT { NV, cV, co, c, cx}
4887 \cs_generate_variant:Nn \seq_if_in:NnF { NV, cV, co, c, cx}
```

## 105.3 Getting data out

`\seq_get:NN` `\seq_get:NN`  $\langle sequence \rangle \langle cmd \rangle$  defines  $\langle cmd \rangle$  to be the left-most element of  $\langle sequence \rangle$ .  
`\seq_get:cN`  
`\seq_get_aux:w` 4888 `\cs_new_protected_nopar:Npn \seq_get:NN #1 {`  
4889 `\seq_if_empty_err:N #1`  
4890 `\exp_after:wN \seq_get_aux:w #1 \q_stop`  
4891 `}`

```

4892 \cs_new_protected:Npn \seq_get_aux:w \seq_elt:w #1 \seq_elt_end: #2 \q_stop #3 {
4893   \tl_set:Nn #3 {#1}
4894 }
4895 \cs_generate_variant:Nn \seq_get:NN {c}

```

`\seq_pop_aux:nnNN` `\seq_pop_aux:nnNN`  $\langle def_1 \rangle$   $\langle def_2 \rangle$   $\langle sequence \rangle$   $\langle cmd \rangle$  assigns the left\_most element of  $\langle sequence \rangle$  to  $\langle cmd \rangle$  using  $\langle def_2 \rangle$ , and assigns the tail of  $\langle sequence \rangle$  to  $\langle sequence \rangle$  using  $\langle def_1 \rangle$ .

```

4896 \cs_new_protected:Npn \seq_pop_aux:nnNN #1#2#3 {
4897   \seq_if_empty_err:N #3
4898   \exp_after:wN \seq_pop_aux:w #3 \q_stop #1#2#3
4899 }
4900 \cs_new_protected:Npn \seq_pop_aux:w
4901   \seq_elt:w #1 \seq_elt_end: #2\q_stop #3#4#5#6 {
4902   #3 #5 {#2}
4903   #4 #6 {#1}
4904 }

```

`\seq_show:N`  
`\seq_show:c`

```

4905 \cs_new_eq:NN \seq_show:N \tl_show:N
4906 \cs_new_eq:NN \seq_show:c \tl_show:c

```

`\seq_display:N`  
`\seq_display:c`

```

4907 \cs_new_protected_nopar:Npn \seq_display:N #1 {
4908   \iow_term:x { Sequence~\token_to_str:N #1~contains~
4909     the~elements~(without~outer~braces): }
4910   \toks_clear:N \l_tmpa_toks
4911   \seq_map_inline:Nn #1 {
4912     \toks_if_empty:NF \l_tmpa_toks {
4913       \toks_put_right:Nx \l_tmpa_toks {^~J>~}
4914     }
4915     \toks_put_right:Nx \l_tmpa_toks {
4916       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
4917     }
4918   }
4919   \toks_show:N \l_tmpa_toks
4920 }
4921 \cs_generate_variant:Nn \seq_display:N {c}

```

## 105.4 Putting data in

`\seq_put_aux:Nnn` `\seq_put_aux:Nnn`  $\langle sequence \rangle$   $\langle left \rangle$   $\langle right \rangle$  adds the elements specified by  $\langle left \rangle$  to the left of  $\langle sequence \rangle$ , and those specified by  $\langle right \rangle$  to the right.

```

4922 \cs_new_protected:Npn \seq_put_aux:Nnn #1 {

```

```

4923 \exp_after:wN \seq_put_aux:w #1 \q_stop #1
4924 }
4925 \cs_new_protected:Npn \seq_put_aux:w #1\q_stop #2#3#4 { \tl_set:Nn #2 {#3#1#4} }

```

\seq\_put\_left:Nn Here are the usual operations for adding to the left and right.

```

\seq_put_left:NV
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:co
\seq_put_right:Nn
\seq_put_right:No
\seq_put_right:NV
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:co

```

```

4926 \cs_new_protected:Npn \seq_put_left:Nn #1#2 {
4927 \seq_put_aux:Nnn #1 {\seq_elt:w #2\seq_elt_end:} {}
4928 }

```

We can't put in a \prg\_do\_nothing: instead of {} above since this argument is passed literally (and we would end up with many \prg\_do\_nothing:s inside the sequences).

```

4929 \cs_generate_variant:Nn \seq_put_left:Nn {NV,No,Nx,c,cV,co}

4930 \cs_new_protected:Npn \seq_put_right:Nn #1#2{
4931 \seq_put_aux:Nnn #1{\seq_elt:w #2\seq_elt_end:}}

4932 \cs_generate_variant:Nn \seq_put_right:Nn {NV,No,Nx,c,cV,co}

```

```

\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:co
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:co

```

```

4933 \cs_new_protected:Npn \seq_gput_left:Nn {
4934 <*check>
4935 \pref_global_chk:
4936 </check>
4937 <!*check>
4938 \pref_global:D
4939 </!check>
4940 \seq_put_left:Nn
4941 }

4942 \cs_new_protected:Npn \seq_gput_right:Nn {
4943 <*check>
4944 \pref_global_chk:
4945 </check>
4946 <!*check>
4947 \pref_global:D
4948 </!check>
4949 \seq_put_right:Nn
4950 }

```

```

4951 \cs_generate_variant:Nn \seq_gput_left:Nn {NV,No,Nx,c,cV,co}

```

```

4952 \cs_generate_variant:Nn \seq_gput_right:Nn {NV,No,Nx,c,cV,co}

```

## 105.5 Mapping

`\seq_map_variable:NNn` Nothing spectacular here.  
`\seq_map_variable:cNn`  
`\seq_map_variable_aux:Nnw`

```

4953 \cs_new_protected:Npn \seq_map_variable_aux:Nnw #1#2 \seq_elt:w #3 \seq_elt_end: {
4954   \tl_set:Nn #1 {#3}
4955   \quark_if_nil:NT #1 \seq_map_break:
4956   #2
4957   \seq_map_variable_aux:Nnw #1{#2}
4958 }
4959 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3 {
4960   \tl_set:Nn #2 {\seq_map_variable_aux:Nnw #2{#3}}
4961   \exp_after:wN #2 #1 \seq_elt:w \q_nil\seq_elt_end: \q_stop
4962 }
4963 \cs_generate_variant:Nn \seq_map_variable:NNn {c}

```

`\seq_map_break:` Terminate a mapping function at the point of execution. The latter takes an argument  
`\seq_map_break:n` to be executed after cleaning up the map.

```

4964 \cs_new_eq:NN \seq_map_break: \use_none_delimit_by_q_stop:w
4965 \cs_new_eq:NN \seq_map_break:n \use_i_delimit_by_q_stop:nw

```

`\seq_map_function:NN` `\seq_map_function:NN`  $\langle sequence \rangle$   $\langle cmd \rangle$  applies  $\langle cmd \rangle$  to each element of  $\langle sequence \rangle$ ,  
`\seq_map_function:cN` from left to right. Since we don't have braces, this implementation is not very efficient. It  
might be better to say that  $\langle cmd \rangle$  must be a function with one argument that is delimited  
by `\seq_elt_end:.`

```

4966 \cs_new_protected_nopar:Npn \seq_map_function:NN #1#2 {
4967   \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2{##1}}
4968   #1 \use_none:n \q_stop
4969   \cs_set_eq:NN \seq_elt:w \ERROR
4970 }
4971 \cs_generate_variant:Nn \seq_map_function:NN {c}

```

`\seq_map_inline:Nn` When no braces are used, this version of mapping seems more natural.  
`\seq_map_inline:cn`

```

4972 \cs_new_protected_nopar:Npn \seq_map_inline:Nn #1#2 {
4973   \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2}
4974   #1 \use_none:n \q_stop
4975   \cs_set_eq:NN \seq_elt:w \ERROR
4976 }
4977 \cs_generate_variant:Nn \seq_map_inline:Nn {c}

```

## 105.6 Manipulation

`\l_clist_remove_clist` A common scratch space for the removal routines.

```

4978 \seq_new:N \l_seq_remove_seq

```

`\seq_remove_duplicates_aux:NN`  
`\seq_remove_duplicates_aux:n`  
`\seq_remove_duplicates:N`  
`\seq_gremove_duplicates:N`

Copied from `\clist_remove_duplicates`.

```

4979 \cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2 {
4980   \seq_clear:N \l_seq_remove_seq
4981   \seq_map_function:NN #2 \seq_remove_duplicates_aux:n
4982   #1 #2 \l_seq_remove_seq
4983 }
4984 \cs_new_protected:Npn \seq_remove_duplicates_aux:n #1 {
4985   \seq_if_in:NnF \l_seq_remove_seq {#1} {
4986     \seq_put_right:Nn \l_seq_remove_seq {#1}
4987   }
4988 }

4989 \cs_new_protected_nopar:Npn \seq_remove_duplicates:N {
4990   \seq_remove_duplicates_aux:NN \seq_set_eq:NN
4991 }
4992 \cs_new_protected_nopar:Npn \seq_gremove_duplicates:N {
4993   \seq_remove_duplicates_aux:NN \seq_gset_eq:NN
4994 }

```

## 105.7 Sequence stacks

`\seq_push:Nn`  
`\seq_push:Nv`  
`\seq_push:No`  
`\seq_push:cn`  
`\seq_pop:NN`  
`\seq_pop:cN`

Since sequences can be used as stacks, we ought to have both ‘push’ and ‘pop’. In most cases they are nothing more than new names for old functions.

```

4995 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
4996 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
4997 \cs_new_eq:NN \seq_push:No \seq_put_left:No
4998 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
4999 \cs_new_protected_nopar:Npn \seq_pop:NN { \seq_pop_aux:nnNN \tl_set:Nn \tl_set:Nn }
5000 \cs_generate_variant:Nn \seq_pop:NN {c}

```

`\seq_gpush:Nn`  
`\seq_gpush:Nv`  
`\seq_gpush:No`  
`\seq_gpush:cn`  
`\seq_gpush:Nv`  
`\seq_gpop:NN`  
`\seq_gpop:cN`

I don’t agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of `\seq_gpop:NN` the value is nevertheless returned locally.

```

5001 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
5002 \cs_new_protected_nopar:Npn \seq_gpop:NN { \seq_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn }
5003 \cs_generate_variant:Nn \seq_gpush:Nn {NV,No,c,Nv}
5004 \cs_generate_variant:Nn \seq_gpop:NN {c}

```

`\seq_top:NN`  
`\seq_top:cN`

Looking at the top element of the stack without removing it is done with this operation.

```

5005 \cs_new_eq:NN \seq_top:NN \seq_get:NN
5006 \cs_new_eq:NN \seq_top:cN \seq_get:cN

5007 </initex | package>

```

Show token usage:

```
5008 <*showmemory>
5009 %\showMemUsage
5010 </showmemory>
```

## 106 l3clist implementation

We start by ensuring that the required packages are loaded.

```
5011 <*package>
5012 \ProvidesExplPackage
5013   {\filename}{\filedate}{\fileversion}{\filedescription}
5014 \package_check_loaded_expl:
5015 </package>
5016 <*initex | package>
```

### 106.1 Allocation and initialisation

`\clist_new:N` Comma-Lists are implemented using token lists.

`\clist_new:c`

```
5017 \cs_new_eq:NN \clist_new:N \tl_new:N
5018 \cs_generate_variant:Nn \clist_new:N {c}
```

`\clist_clear:N`

Clearing a comma-list is the same as clearing a token list.

`\clist_clear:c`

`\clist_gclear:N`

`\clist_gclear:c`

```
5019 \cs_new_eq:NN \clist_clear:N \tl_clear:N
5020 \cs_generate_variant:Nn \clist_clear:N {c}
5021 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
5022 \cs_generate_variant:Nn \clist_gclear:N {c}
```

`\clist_clear_new:N`

Clearing a comma-list is the same as clearing a token list.

`\clist_clear_new:c`

`\clist_gclear_new:N`

`\clist_gclear_new:c`

```
5023 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
5024 \cs_generate_variant:Nn \clist_clear_new:N {c}
5025 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
5026 \cs_generate_variant:Nn \clist_gclear_new:N {c}
```

`\clist_set_eq:NN`

We can set one `<clist>` equal to another.

`\clist_set_eq:cN`

`\clist_set_eq:Nc`

`\clist_set_eq:cc`

```
5027 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
5028 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
5029 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
5030 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
```

\clist\_gset\_eq:NN An of course globally which seems to be needed far more often.

```
\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc
5031 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
5032 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
5033 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
5034 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

## 106.2 Predicates and conditionals

```
\clist_if_empty_p:N
\clist_if_empty_p:c
\clist_if_empty:N $\underline{TF}$ 
\clist_if_empty:c $\underline{TF}$ 
5035 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N {p,TF,T,F}
5036 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c {p,TF,T,F}
```

\clist\_if\_empty\_err:N Signals an error if the comma-list is empty.

```
5037 \cs_new_protected_nopar:Npn \clist_if_empty_err:N #1 {
5038   \if_meaning:w #1 \c_empty_tl
5039   \tl_clear:N \l_kernel_testa_tl % catch prefixes
5040   \msg_kernel_bug:x {Empty-comma-list~'\token_to_str:N #1'}
5041   \fi:
5042 }
```

\clist\_if\_eq\_p:NN Returns \c\_true iff the two comma-lists are equal.

```
\clist_if_eq_p:Nc
\clist_if_eq_p:cN
\clist_if_eq_p:cc
\clist_if_eq:NN $\underline{TF}$ 
\clist_if_eq:cN $\underline{TF}$ 
\clist_if_eq:Nc $\underline{TF}$ 
\clist_if_eq:cc $\underline{TF}$ 
\clist_if_eq:Nc $\underline{TF}$ 
5043 \prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN {p,TF,T,F}
5044 \prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN {p,TF,T,F}
5045 \prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc {p,TF,T,F}
5046 \prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc {p,TF,T,F}
```

\clist\_if\_in:NnTF  $\langle\textit{clist}\rangle\langle\textit{item}\rangle\langle\textit{true case}\rangle\langle\textit{false case}\rangle$  will check whether  $\langle\textit{item}\rangle$  is in  $\langle\textit{clist}\rangle$  and then either execute the  $\langle\textit{true case}\rangle$  or the  $\langle\textit{false case}\rangle$ .  $\langle\textit{true case}\rangle$  and  $\langle\textit{false case}\rangle$  may contain incomplete \if\_charcode:w statements.

```
\clist_if_in:NcTF
\clist_if_in:cNTF
\clist_if_in:cN $\underline{TF}$ 
\clist_if_in:ccTF
5047 \prg_new_protected_conditional:Nnn \clist_if_in:Nn {TF,T,F} {
5048   \cs_set:Npn \clist_tmp:w ##1,##2,##2##3 \q_stop {
5049     \if_meaning:w \q_no_value ##2
5050     \prg_return_false: \else: \prg_return_true: \fi:
5051   }
5052   \exp_last_unbraced:NNo \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
5053 }
```

```
5054 \cs_generate_variant:Nn \clist_if_in:NnTF {NV,No,cn,cV,co}
5055 \cs_generate_variant:Nn \clist_if_in:NnT {NV,No,cn,cV,co}
5056 \cs_generate_variant:Nn \clist_if_in:NnF {NV,No,cn,cV,co}
```

### 106.3 Retrieving data

`\clist_use:N` Using a  $\langle clist \rangle$  is just executing it but if  $\langle clist \rangle$  equals `\scan_stop:` it is probably stemming from a `\cs:w ... \cs_end:` that was created by mistake somewhere.

```

5057 \cs_new_nopar:Npn \clist_use:N #1 {
5058   \if_meaning:w #1 \scan_stop:
5059     \msg_kernel_bug:x {
5060       Comma~list~ '\token_to_str:N #1'~ has~ an~ erroneous~ structure!}
5061   \else:
5062     \exp_after:wN #1
5063   \fi:
5064 }
5065 \cs_generate_variant:Nn \clist_use:N {c}

```

`\clist_get:NN`  $\langle comma-list \rangle \langle cmd \rangle$  defines  $\langle cmd \rangle$  to be the left-most element of  $\langle comma-list \rangle$ .  
`\clist_get:cN`  
`\clist_get_aux:w`

```

5066 \cs_new_protected_nopar:Npn \clist_get:NN #1 {
5067   \clist_if_empty_err:N #1
5068   \exp_after:wN \clist_get_aux:w #1,\q_stop
5069 }

5070 \cs_new_protected:Npn \clist_get_aux:w #1,#2\q_stop #3 { \tl_set:Nn #3{#1} }

5071 \cs_generate_variant:Nn \clist_get:NN {cN}

```

`\clist_pop_aux:nnNN`  $\langle def_1 \rangle \langle def_2 \rangle \langle comma-list \rangle \langle cmd \rangle$  assigns the left-most element of  $\langle comma-list \rangle$  to  $\langle cmd \rangle$  using  $\langle def_2 \rangle$ , and assigns the tail of  $\langle comma-list \rangle$  to  $\langle comma-list \rangle$  using  $\langle def_1 \rangle$ .  
`\clist_pop_aux:w`  
`\clist_pop_auxi:w`

```

5072 \cs_new_protected:Npn \clist_pop_aux:nnNN #1#2#3 {
5073   \clist_if_empty_err:N #3
5074   \exp_after:wN \clist_pop_aux:w #3,\q_nil\q_stop #1#2#3
5075 }

```

After the assignments below, if there was only one element in the original `clist`, it now contains only `\q_nil`.

```

5076 \cs_new_protected:Npn \clist_pop_aux:w #1,#2\q_stop #3#4#5#6 {
5077   #4 #6 {#1}
5078   #3 #5 {#2}
5079   \quark_if_nil:NTF #5 { #3 #5 {} }{ \clist_pop_auxi:w #2 #3#5 }
5080 }

5081 \cs_new:Npn \clist_pop_auxi:w #1,\q_nil #2#3 { #2#3{#1} }

```

`\clist_show:N`  
`\clist_show:c`

```

5082 \cs_new_eq:NN \clist_show:N \tl_show:N
5083 \cs_new_eq:NN \clist_show:c \tl_show:c

```

\clist\_display:N  
\clist\_display:c

```

5084 \cs_new_protected_nopar:Npn \clist_display:N #1 {
5085   \iow_term:x { Comma-list~\token_to_str:N #1~contains~
5086     the~elements~(without~outer~braces): }
5087   \toks_clear:N \l_tmpa_toks
5088   \clist_map_inline:Nn #1 {
5089     \toks_if_empty:NF \l_tmpa_toks {
5090       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
5091     }
5092     \toks_put_right:Nx \l_tmpa_toks {
5093       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
5094     }
5095   }
5096   \toks_show:N \l_tmpa_toks
5097 }
5098 \cs_generate_variant:Nn \clist_display:N {c}

```

## 106.4 Storing data

\clist\_put\_aux:NNnnNn

The generic put function. When adding we have to distinguish between an empty *<clist>* and one that contains at least one item (otherwise we accumulate commas).

MH says: Perhaps we should make sure that empty arguments don't get on the stack as that is probably a mistake. That's what I've implemented here. Since `\tl_if_empty:nF` is expandable prefixes are still allowed.

```

5099 \cs_new_protected:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6 {
5100   \clist_if_empty:NTF #5 { #1 #5 {#6} } {
5101     \tl_if_empty:nF {#6} { #2 #5{#3#6#4} }
5102   }
5103 }

```

\clist\_put\_left:Nn

The operations for adding to the left.

\clist\_put\_left:NV

\clist\_put\_left:No

\clist\_put\_left:Nx

\clist\_put\_left:cn

\clist\_put\_left:cV

\clist\_put\_left:co

\clist\_gput\_left:Nn

Global versions.

```

5104 \cs_new_protected_nopar:Npn \clist_put_left:Nn {
5105   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn {} ,
5106 }
5107 \cs_generate_variant:Nn \clist_put_left:Nn {NV,No,Nx,cn,cV,co}

```

\clist\_gput\_left:NV

\clist\_gput\_left:No

\clist\_gput\_left:Nx

\clist\_gput\_left:cn

\clist\_gput\_left:cV

\clist\_gput\_left:co

```

5108 \cs_new_protected_nopar:Npn \clist_gput_left:Nn {
5109   \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_left:Nn {} ,
5110 }
5111 \cs_generate_variant:Nn \clist_gput_left:Nn {NV,No,Nx,cn,cV,co}

```

\clist\_gput\_left:co

Adding something to the right side is almost the same.

\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

```

5112 \cs_new_protected_nopar:Npn \clist_put_right:Nn {
5113   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn , {}
5114 }
5115 \cs_generate_variant:Nn \clist_put_right:Nn {NV,No,Nx,cn,cV,co}

```

\clist\_gput\_right:Nn And here the global variants.

```

\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
5116 \cs_new_protected_nopar:Npn \clist_gput_right:Nn {
5117   \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , {}
5118 }
5119 \cs_generate_variant:Nn \clist_gput_right:Nn {NV,No,Nx,cn,cV,co}

```

## 106.5 Mapping

\clist\_map\_function:NN Using the above creating the comma mappings is easy..

```

\clist_map_function:Nc
\clist_map_function:cN
\clist_map_function:cc
\clist_map_function:nN
\clist_map_function:nc
\clist_map_inline:Nn
\clist_map_inline:cn
5120 \prg_new_map_functions:Nn , { clist }
5121 \cs_generate_variant:Nn \clist_map_function:NN { Nc }
5122 \cs_generate_variant:Nn \clist_map_function:NN { c }
5123 \cs_generate_variant:Nn \clist_map_function:NN { cc }
5124 \cs_generate_variant:Nn \clist_map_inline:Nn { c }
5125 \cs_generate_variant:Nn \clist_map_inline:Nn { nc }

```

\clist\_map\_inline:nn  
\clist\_map\_variable:NNn \clist\_map\_break:  
\clist\_map\_variable:NNn  
\clist\_map\_variable:cNn  
\clist\_map\_variable:NNn *<comma-list>* *<temp>* *<action>* assigns *<temp>* to each element and executes *<action>*.

```

5126 \cs_new_protected:Npn \clist_map_variable:nNn #1#2#3 {
5127   \tl_if_empty:nF {#1} {
5128     \clist_map_variable_aux:Nnw #2 {#3} #1
5129     , \q_recursion_tail , \q_recursion_stop
5130   }
5131 }

```

Something for v/V

```

5132 \cs_new_protected_nopar:Npn \clist_map_variable:NNn {\exp_args:No \clist_map_variable:nNn}
5133 \cs_generate_variant:Nn \clist_map_variable:NNn {cNn}

```

\clist\_map\_variable\_aux:Nnw The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```

5134 \cs_new_protected:Npn \clist_map_variable_aux:Nnw #1#2#3,{
5135   \cs_set_nopar:Npn #1{#3}
5136   \quark_if_recursion_tail_stop:N #1
5137   #2 \clist_map_variable_aux:Nnw #1{#2}
5138 }

```

## 106.6 Higher level functions

`\clist_concat_aux:NNNN` `\clist_concat:NNN`  $\langle \textit{clist 1} \rangle$   $\langle \textit{clist 2} \rangle$   $\langle \textit{clist 3} \rangle$  will globally assign  $\langle \textit{clist 1} \rangle$  the concatenation of  $\langle \textit{clist 2} \rangle$  and  $\langle \textit{clist 3} \rangle$ .  
`\clist_concat:ccc`  
`\clist_gconcat:NNN` Again the situation is a bit more complicated because of the use of commas between items, so if either list is empty we have to avoid adding a comma.  
`\clist_gconcat:ccc`

```

5139 \cs_new_protected_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4 {
5140   \tl_set:No \l_tmpa_tl {#3}
5141   \tl_set:No \l_tmpb_tl {#4}
5142   #1 #2 {
5143     \exp_not:V \l_tmpa_tl
5144     \tl_if_empty:NF \l_tmpa_tl { \tl_if_empty:NF \l_tmpb_tl , }
5145     \exp_not:V \l_tmpb_tl
5146   }
5147 }
5148 \cs_new_protected_nopar:Npn \clist_concat:NNN { \clist_concat_aux:NNNN \tl_set:Nx }
5149 \cs_new_protected_nopar:Npn \clist_gconcat:NNN { \clist_concat_aux:NNNN \tl_gset:Nx }
5150 \cs_generate_variant:Nn \clist_concat:NNN {ccc}
5151 \cs_generate_variant:Nn \clist_gconcat:NNN {ccc}

```

`\l_clist_remove_clist` A common scratch space for the removal routines.

```

5152 \clist_new:N \l_clist_remove_clist

```

`\clist_remove_duplicates_aux:NN` Removing duplicate entries in a  $\langle \textit{clist} \rangle$  is fairly straight forward. We use a temporary variable and then go through the list from left to right. For each element check if the  
`\clist_remove_duplicates_aux:n` element is already present in the list.  
`\clist_remove_duplicates:N`  
`\clist_gremove_duplicates:N`

```

5153 \cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2 {
5154   \clist_clear:N \l_clist_remove_clist
5155   \clist_map_function:NN #2 \clist_remove_duplicates_aux:n
5156   #1 #2 \l_clist_remove_clist
5157 }
5158 \cs_new_protected:Npn \clist_remove_duplicates_aux:n #1 {
5159   \clist_if_in:NnF \l_clist_remove_clist {#1} {
5160     \clist_put_right:Nn \l_clist_remove_clist {#1}
5161   }
5162 }

```

The high level functions are just for telling if it should be a local or global setting.

```

5163 \cs_new_protected_nopar:Npn \clist_remove_duplicates:N {
5164   \clist_remove_duplicates_aux:NN \clist_set_eq:NN
5165 }
5166 \cs_new_protected_nopar:Npn \clist_gremove_duplicates:N {
5167   \clist_remove_duplicates_aux:NN \clist_gset_eq:NN
5168 }

```

```

\clist_remove_element:Nn
\clist_gremove_element:Nn
\clist_remove_element_aux:NNn
\clist_remove_element_aux:n

```

The same general idea is used for removing elements: the parent functions just set things up for the internal ones.

```

5169 \cs_new_protected_nopar:Npn \clist_remove_element:Nn {
5170   \clist_remove_element_aux:NNn \clist_set_eq:NN
5171 }
5172 \cs_new_protected_nopar:Npn \clist_gremove_element:Nn {
5173   \clist_remove_element_aux:NNn \clist_gset_eq:NN
5174 }
5175 \cs_new_protected:Npn \clist_remove_element_aux:NNn #1#2#3 {
5176   \clist_clear:N \l_clist_remove_clist
5177   \cs_set:Npn \clist_remove_element_aux:n ##1 {
5178     \str_if_eq:nnF {#3} {##1} {
5179       \clist_put_right:Nn \l_clist_remove_clist {##1}
5180     }
5181   }
5182   \clist_map_function:NN #2 \clist_remove_element_aux:n
5183   #1 #2 \l_clist_remove_clist
5184 }
5185 \cs_new:Npn \clist_remove_element_aux:n #1 { }

```

## 106.7 Stack operations

We build stacks from comma-lists, but here we put the specific functions together.

```

\clist_push:Nn
\clist_push:No
\clist_push:NV
\clist_push:cn
\clist_pop:NN
\clist_pop:cN

```

Since comma-lists can be used as stacks, we ought to have both ‘push’ and ‘pop’. In most cases they are nothing more than new names for old functions.

```

5186 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
5187 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
5188 \cs_new_eq:NN \clist_push:No \clist_put_left:No
5189 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
5190 \cs_new_protected_nopar:Npn \clist_pop:NN {\clist_pop_aux:nnNN \tl_set:Nn \tl_set:Nn}
5191 \cs_generate_variant:Nn \clist_pop:NN {cN}

```

```

\clist_gpush:Nn
\clist_gpush:No
\clist_gpush:NV
\clist_gpush:cn
\clist_gpop:NN
\clist_gpop:cN

```

I don’t agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of `\clist_gpop:NN` the value is nevertheless returned locally.

```

5192 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
5193 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
5194 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
5195 \cs_generate_variant:Nn \clist_gpush:Nn {cN}

5196 \cs_new_protected_nopar:Npn \clist_gpop:NN {\clist_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn}
5197 \cs_generate_variant:Nn \clist_gpop:NN {cN}

```

`\clist_top:NN` Looking at the top element of the stack without removing it is done with this operation.  
`\clist_top:cN`

```
5198 \cs_new_eq:NN \clist_top:NN \clist_get:NN
5199 \cs_new_eq:NN \clist_top:cN \clist_get:cN
```

```
5200 </initex | package>
```

Show token usage:

```
5201 <*showmemory>
5202 %\showMemUsage
5203 </showmemory>
```

## 107 l3prop implementation

A property list is a token register whose contents is of the form

$$\backslash\mathbf{q\_prop} \langle key_1 \rangle \backslash\mathbf{q\_prop} \{ \langle info_1 \rangle \} \dots \backslash\mathbf{q\_prop} \langle key_n \rangle \backslash\mathbf{q\_prop} \{ \langle info_n \rangle \}$$

The property  $\langle key \rangle$ s and  $\langle info \rangle$ s might be arbitrary token lists; each  $\langle info \rangle$  is surrounded by braces.

We start by ensuring that the required packages are loaded.

```
5204 <*package>
5205 \ProvidesExplPackage
5206 { \filename } { \filedate } { \fileversion } { \filedescription }
5207 \package_check_loaded_expl:
5208 </package>
5209 <*initex | package>
```

`\q_prop` The separator between  $\langle key \rangle$ s and  $\langle info \rangle$ s and  $\langle key \rangle$ s.

```
5210 \quark_new:N\q_prop
```

To get values from property-lists, token lists should be passed to the appropriate functions.

### 107.1 Functions

`\prop_new:N` Property lists are implemented as token registers.

`\prop_new:c`

```
5211 \cs_new_eq:NN \prop_new:N \toks_new:N
5212 \cs_new_eq:NN \prop_new:c \toks_new:c
```

`\prop_clear:N` The same goes for clearing a property list, either locally or globally.

```

\prop_clear:c
\prop_gclear:N
\prop_gclear:c
5213 \cs_new_eq:NN \prop_clear:N \toks_clear:N
5214 \cs_new_eq:NN \prop_clear:c \toks_clear:c
5215 \cs_new_eq:NN \prop_gclear:N \toks_gclear:N
5216 \cs_new_eq:NN \prop_gclear:c \toks_gclear:c

```

`\prop_set_eq:NN` This makes two  $\langle prop \rangle$ s have the same contents.

```

\prop_set_eq:Nc
\prop_set_eq:cN
\prop_set_eq:cc
5217 \cs_new_eq:NN \prop_set_eq:NN \toks_set_eq:NN
5218 \cs_new_eq:NN \prop_set_eq:Nc \toks_set_eq:Nc
5219 \cs_new_eq:NN \prop_set_eq:cN \toks_set_eq:cN
5220 \cs_new_eq:NN \prop_set_eq:cc \toks_set_eq:cc
\prop_gset_eq:NN
\prop_gset_eq:Nc
\prop_gset_eq:cN
\prop_gset_eq:cc
5221 \cs_new_eq:NN \prop_gset_eq:NN \toks_gset_eq:NN
5222 \cs_new_eq:NN \prop_gset_eq:Nc \toks_gset_eq:Nc
5223 \cs_new_eq:NN \prop_gset_eq:cN \toks_gset_eq:cN
5224 \cs_new_eq:NN \prop_gset_eq:cc \toks_gset_eq:cc

```

`\prop_show:N` Show on the console the raw contents of a property list's token register.

```

\prop_show:c
5225 \cs_new_eq:NN \prop_show:N \toks_show:N
5226 \cs_new_eq:NN \prop_show:c \toks_show:c

```

`\prop_display:N` Pretty print the contents of a property list on the console.

```

\prop_display:c
5227 \cs_new_protected_nopar:Npn \prop_display:N #1 {
5228   \iow_term:x { Property-list~\token_to_str:N #1~contains~
5229     the~pairs~(without~outer~braces): }
5230   \toks_clear:N \l_tmpa_toks
5231   \prop_map_inline:Nn #1 {
5232     \toks_if_empty:NF \l_tmpa_toks {
5233       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
5234     }
5235     \toks_put_right:Nx \l_tmpa_toks {
5236       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \} \c_space_tl
5237       \c_space_tl => \c_space_tl
5238       \c_space_tl \iow_char:N \{ \exp_not:n {##2} \iow_char:N \}
5239     }
5240   }
5241   \toks_show:N \l_tmpa_toks
5242 }
5243 \cs_generate_variant:Nn \prop_display:N {c}

```

`\prop_split_aux:Nnn` `\prop_split_aux:Nnn $\langle prop \rangle$  $\langle key \rangle$  $\langle cmd \rangle$`  invokes  $\langle cmd \rangle$  with 3 arguments: 1st is the beginning of  $\langle prop \rangle$  before  $\langle key \rangle$ , 2nd is the value associated with  $\langle key \rangle$ , 3rd is the rest of  $\langle prop \rangle$  after  $\langle key \rangle$ . If there is no property  $\langle key \rangle$  in  $\langle prop \rangle$ , then the 2nd argument will be `\q_no_value` and the 3rd argument is empty; otherwise the 3rd argument has the extra tokens `\q_prop  $\langle key \rangle$  \q_prop \q_no_value` at the end.

```

5244 \cs_new_protected:Npn \prop_split_aux:Nnn #1#2#3{
5245   \cs_set:Npn \prop_tmp:w ##1 \q_prop #2 \q_prop ##2##3 \q_stop {
5246     #3 {##1}{##2}{##3}
5247   }
5248   \exp_after:wN \prop_tmp:w \toks_use:N #1 \q_prop #2 \q_prop \q_no_value \q_stop
5249 }

```

`\prop_get:NnN` `\prop_get:NnN`  $\langle prop \rangle$   $\langle key \rangle$   $\langle tl var. \rangle$  defines  $\langle tl var. \rangle$  to be the value associated with  $\langle key \rangle$  in  $\langle prop \rangle$ , `\q_no_value` if not found.  
`\prop_get:NVN`  
`\prop_get:cnN`  
`\prop_get:cVN`  
`\prop_get_aux:w`

```

5250 \cs_new_protected:Npn \prop_get:NnN #1#2 {
5251   \prop_split_aux:Nnn #1{#2}\prop_get_aux:w
5252 }
5253 \cs_new_protected:Npn \prop_get_aux:w #1#2#3#4 { \tl_set:Nn #4 {#2} }

5254 \cs_generate_variant:Nn \prop_get:NnN { NVN, cnN, cVN }

```

`\prop_gget:NnN` The global version of the previous function.  
`\prop_gget:NVN`  
`\prop_gget:NnN`  
`\prop_gget:NVN`  
`\prop_gget_aux:w`

```

5255 \cs_new_protected:Npn \prop_gget:NnN #1#2{
5256   \prop_split_aux:Nnn #1{#2}\prop_gget_aux:w}
5257 \cs_new_protected:Npn \prop_gget_aux:w #1#2#3#4{\tl_gset:Nx#4{\exp_not:n{#2}}}

5258 \cs_generate_variant:Nn \prop_gget:NnN { NVN, cnN, cVN }

```

`\prop_get_gdel:NnN` `\prop_get_gdel:NnN` is the same as `\prop_get:NnN` but the  $\langle key \rangle$  and its value are afterwards globally removed from  $\langle property\_list \rangle$ . One probably also needs the local variants or only the local one, or... We decide this later.  
`\prop_get_del_aux:w`

```

5259 \cs_new_protected:Npn \prop_get_gdel:NnN #1#2#3{
5260   \prop_split_aux:Nnn #1{#2}{\prop_get_del_aux:w #3{\toks_gset:Nn #1{#2}}}
5261   \cs_new_protected:Npn \prop_get_del_aux:w #1#2#3#4#5#6{
5262     \tl_set:Nn #1 {#5}
5263     \quark_if_no_value:NF #1 {
5264       \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#3\q_prop\q_no_value {#2{#4##1}}
5265       \prop_tmp:w #6}
5266   }

```

`\prop_put:Nnn` `\prop_put:Nnn`  $\{ \langle prop \rangle \}$   $\{ \langle key \rangle \}$   $\{ \langle info \rangle \}$  adds/changes the value associated with  $\langle key \rangle$  in  $\langle prop \rangle$  to  $\langle info \rangle$ .  
`\prop_put:NnV`  
`\prop_put:Nnx`  
`\prop_put:NVn`  
`\prop_put:NVV`  
`\prop_put:cnN`  
`\prop_put:cnx`  
`\prop_gput:Nnn`  
`\prop_gput:NVn`  
`\prop_gput:NnV`  
`\prop_gput:Nno`  
`\prop_gput:Nnx`  
`\prop_gput:cnN`  
`\prop_gput:ccx`  
`\prop_put_aux:w`

```

5267 \cs_new_protected:Npn \prop_put:Nnn #1#2{
5268   \prop_split_aux:Nnn #1{#2}{
5269     \prop_clear:N #1
5270     \prop_put_aux:w {\toks_put_right:Nn #1}{#2}
5271   }
5272 }

```

```

5273 \cs_new_protected:Npn \prop_gput:Nnn #1#2{
5274   \prop_split_aux:Nnn #1{#2}{
5275     \prop_gclear:N #1
5276     \prop_put_aux:w {\toks_gput_right:Nn #1}{#2}
5277   }
5278 }

5279 \cs_new_protected:Npn \prop_put_aux:w #1#2#3#4#5#6{
5280   #1{\q_prop#2\q_prop{#6}{#3}
5281   \tl_if_empty:nF{#5}
5282   {
5283     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
5284     \prop_tmp:w #5
5285   }
5286 }

5287 \cs_generate_variant:Nn \prop_put:Nnn { Nno , NnV, Nnx, NVn, NVV, cnn , cnx }

5288 \cs_generate_variant:Nn \prop_gput:Nnn { NVn,NnV,Nno,Nnx,Nox,cnn,ccx}

\prop_del:Nn \prop_del:Nn <prop><key> deletes the entry for <key> in <prop>, if any.
\prop_del:NV
\prop_gdel:NV
\prop_gdel:Nn
\prop_del_aux:w
5289 \cs_new_protected:Npn \prop_del:Nn #1#2{
5290   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_set:Nn #1}{#2}}}
5291 \cs_new_protected:Npn \prop_gdel:Nn #1#2{
5292   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_gset:Nn #1}{#2}}}
5293 \cs_new_protected:Npn \prop_del_aux:w #1#2#3#4#5{
5294   \cs_set_nopar:Npn \prop_tmp:w {#4}
5295   \quark_if_no_value:NF \prop_tmp:w {
5296     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
5297     \prop_tmp:w #5
5298   }
5299 }

5300 \cs_generate_variant:Nn \prop_del:Nn { NV }
5301 \cs_generate_variant:Nn \prop_gdel:Nn { NV }
5302 %

\prop_gput_if_new:Nnn \prop_gput_if_new:Nnn <prop> <key> <info> is equivalent to
\prop_put_if_new_aux:w
\prop_if_in:NnTF <prop><key>
{}%
{\prop_gput:Nnn
  <property_list>
  <key>
  <info>}}

```

Here we go (listening to Porgy & Bess in a recording with Ella F. and Louis A. which makes writing macros sometimes difficult; I find myself humming instead of working):

```

5303 \cs_new_protected:Npn \prop_gput_if_new:Nnn #1#2{
5304   \prop_split_aux:Nnn #1{#2}{\prop_put_if_new_aux:w #1{#2}}
5305 \cs_new_protected:Npn \prop_put_if_new_aux:w #1#2#3#4#5#6{
5306   \tl_if_empty:nT {#5}{#1{\q_prop#2\q_prop{#6}#3}}

```

## 107.2 Predicates and conditionals

`\prop_if_empty_p:N` This conditional takes a  $\langle prop \rangle$  as its argument and evaluates either the true or the false case, depending on whether or not  $\langle prop \rangle$  contains any properties.

```

\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF
5307 \prg_new_eq_conditional:Nnn \prop_if_empty:N \toks_if_empty:N {p,TF,T,F}
5308 \prg_new_eq_conditional:Nnn \prop_if_empty:c \toks_if_empty:c {p,TF,T,F}

```

`\prop_if_eq_p:NN` These functions test whether two property lists are equal.

```

\prop_if_eq_p:cN
\prop_if_eq_p:Nc
5309 \prg_new_eq_conditional:Nnn \prop_if_eq:NN \toks_if_eq:NN {p,TF,T,F}
5310 \prg_new_eq_conditional:Nnn \prop_if_eq:cN \toks_if_eq:cN {p,TF,T,F}
5311 \prg_new_eq_conditional:Nnn \prop_if_eq:Nc \toks_if_eq:Nc {p,TF,T,F}
5312 \prg_new_eq_conditional:Nnn \prop_if_eq:cc \toks_if_eq:cc {p,TF,T,F}
\prop_if_eq:NcTF
\prop_if_eq:cNTF
\prop_if_eq:NcTF
\prop_if_eq:cNTF
\prop_if_eq:NNTF
\prop_if_eq:NcTF
\prop_if_eq:ccTF
\prop_if_eq:ccNTF

```

`\prop_if_in:NnTF`  $\langle property\_list \rangle$   $\langle key \rangle$   $\langle true\_case \rangle$   $\langle false\_case \rangle$  will check whether or not  $\langle key \rangle$  is on the  $\langle property\_list \rangle$  and then select either the true or false case.

```

\prop_if_in:NVTF
\prop_if_in:NcTF
\prop_if_in:NNTF
\prop_if_in:NcTF
\prop_if_in:ccTF
\prop_if_in:ccNTF
\prop_if_in_aux:w
5313 \prg_new_protected_conditional:Nnn \prop_if_in:Nn {TF,T,F} {
5314   \prop_split_aux:Nnn #1 {#2} {\prop_if_in_aux:w}
5315 }
5316 \cs_new_nopar:Npn \prop_if_in_aux:w #1#2#3 {
5317   \quark_if_no_value:nTF {#2} {\prg_return_false:} {\prg_return_true:}
5318 }
5319 \cs_generate_variant:Nn \prop_if_in:NnTF {NV,No,cn,cc}
5320 \cs_generate_variant:Nn \prop_if_in:NnT {NV,No,cn,cc}
5321 \cs_generate_variant:Nn \prop_if_in:NnF {NV,No,cn,cc}

```

## 107.3 Mapping functions

`\prop_map_function:NN` Maps a function on every entry in the property list. The function must take 2 arguments: a key and a value.

`\prop_map_function:cN` First, some failed attempts:

`\prop_map_function:Nc`

`\prop_map_function:cc`

```

\prop_map_function_aux:w
\cs_new_nopar:Npn \prop_map_function:NN #1#2{
  \exp_after:wN \prop_map_function_aux:w
  \exp_after:wN #2 \toks_use:N #1 \q_prop{} \q_prop \q_no_value \q_stop
}
\cs_new_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \if_predicate:w \tl_if_empty_p:n{#2}

```

```

\exp_after:wN \prop_map_break:
\fi:
#1{#2}{#3}
\prop_map_function_aux:w #1
}

```

problem with the above implementation is that an empty key stops the mapping but all other functions in the module allow the use of empty keys (as one value)

```

\cs_set_nopar:Npn \prop_map_function:NN #1#2{
\exp_after:wN \prop_map_function_aux:w
\exp_after:wN #2 \toks_use:N #1 \q_prop \q_no_value \q_prop \q_no_value
}
\cs_set_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
\quark_if_no_value:nF{#2}
{
#1{#2}{#3}
\prop_map_function_aux:w #1
}
}

```

problem with the above implementation is that `\quark_if_no_value:nF` is fairly slow and if `\quark_if_no_value:NF` is used instead we have to do an assignment thus making the mapping not expandable (is that important?)

Here's the current version of the code:

```

5322 \cs_set_nopar:Npn \prop_map_function:NN #1#2 {
5323   \exp_after:wN \prop_map_function_aux:w
5324   \exp_after:wN #2 \toks_use:N #1 \q_prop \q_nil \q_prop \q_no_value \q_stop
5325 }
5326 \cs_set:Npn \prop_map_function_aux:w #1 \q_prop #2 \q_prop #3 {
5327   \if_meaning:w \q_nil #2
5328   \exp_after:wN \prop_map_break:
5329   \fi:
5330   #1{#2}{#3}
5331   \prop_map_function_aux:w #1
5332 }

```

(potential) problem with the above implementation is that it will return true if #2 contains more than just `\q_nil` thus executing whatever follows. Claim: this can't happen :-)  
so we should be ok

```

5333 \cs_generate_variant:Nn \prop_map_function:NN {c,Nc,cc}

```

```

\prop_map_inline:Nn
\prop_map_inline:cn
\g_prop_inline_level_int

```

The inline functions are straight forward. It takes longer to test if the list is empty than to run it on an empty list so we don't waste time doing that.

```

5334 \int_new:N \g_prop_inline_level_int
5335 \cs_new_protected_nopar:Npn \prop_map_inline:Nn #1#2 {
5336   \int_gincr:N \g_prop_inline_level_int
5337   \cs_gset:cpn {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
5338     ##1##2{#2}
5339   \prop_map_function:Nc #1
5340     {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
5341   \int_gdecr:N \g_prop_inline_level_int
5342 }

5343 \cs_generate_variant:Nn\prop_map_inline:Nn {cn}

```

`\prop_map_break:` The break statement.

```

5344 \cs_new_eq:NN \prop_map_break: \use_none_delimit_by_q_stop:w
5345 </initex | package>

```

Show token usage:

```

5346 <*showmemory>
5347 %\showMemUsage
5348 </showmemory>

```

## 108 l3io implementation

We start by ensuring that the required packages are loaded.

```

5349 <*package>
5350 \ProvidesExplPackage
5351   {\filename}{\filedate}{\fileversion}{\filedescription}
5352   \package_check_loaded_expl:
5353 </package>
5354 <*initex | package>

```

### 108.1 Variables and constants

`\c_iow_term_stream` Here we allocate two output streams for writing to the transcript file only (`\c_iow_log_stream`) and to both the terminal and transcript file (`\c_iow_term_stream`). Both can be used to read from and have equivalent `\c_ior` versions.

```

5355 \cs_new_eq:NN \c_iow_term_stream \c_sixteen
5356 \cs_new_eq:NN \c_ior_term_stream \c_sixteen
5357 \cs_new_eq:NN \c_iow_log_stream \c_minus_one
5358 \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
5359 \tl_const:Nn \c_iow_streams_tl
5360 {
5361   \c_zero
5362   \c_one
5363   \c_two
5364   \c_three
5365   \c_four
5366   \c_five
5367   \c_six
5368   \c_seven
5369   \c_eight
5370   \c_nine
5371   \c_ten
5372   \c_eleven
5373   \c_twelve
5374   \c_thirteen
5375   \c_fourteen
5376   \c_fifteen
5377 }
5378 \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.

```

\g_ior_streams_prop
5379 \prop_new:N \g_iow_streams_prop
5380 \prop_new:N \g_ior_streams_prop
5381 </initex | package>
5382 <*package>
5383 \prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e-reserved }
5384 \prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e-reserved }
5385 \prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e-reserved }
5386 \prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e-reserved }
5387 </package>
5388 <*initex | 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
5389 \int_new:N \l_iow_stream_int
5390 \cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int

```

## 108.2 Stream management

`\iow_raw_new:N` The lowest level for stream management is actually creating raw T<sub>E</sub>X streams. As these are very limited (even with  $\epsilon$ -T<sub>E</sub>X) this should not be addressed directly.

```

5391 </initex|package>
5392 <*initex>
5393 \alloc_setup_type:nnn { iow } \c_zero \c_sixteen
5394 \cs_new_protected_nopar:Npn \iow_raw_new:N #1 {
5395   \alloc_reg:NnNN g { iow } \tex_chardef:D #1
5396 }
5397 \alloc_setup_type:nnn { ior } \c_zero \c_sixteen
5398 \cs_new_protected_nopar:Npn \ior_raw_new:N #1 {
5399   \alloc_reg:NnNN g { ior } \tex_chardef:D #1
5400 }
5401 </initex>
5402 <*package>
5403 \cs_set_eq:NN \iow_raw_new:N \newwrite
5404 \cs_set_eq:NN \ior_raw_new:N \newread
5405 </package>
5406 <*initex|package>
5407 \cs_generate_variant:Nn \iow_raw_new:N { c }
5408 \cs_generate_variant:Nn \ior_raw_new:N { c }

```

\iow\_new:N These are not needed but are included for consistency with other variable types.

```

\iow_new:c
\ior_new:N
\ior_new:c
5409 \cs_new_protected_nopar:Npn \iow_new:N #1 {
5410   \cs_new_eq:NN #1 \c_iow_log_stream
5411 }
5412 \cs_generate_variant:Nn \iow_new:N { c }
5413 \cs_new_protected_nopar:Npn \ior_new:N #1 {
5414   \cs_new_eq:NN #1 \c_ior_log_stream
5415 }
5416 \cs_generate_variant:Nn \ior_new:N { c }

```

\iow\_open:Nn In both cases, opening a stream starts with a call to the closing function: this is safest.  
\iow\_open:cn There is then a loop through the allocation number list to find the first free stream  
\ior\_open:Nn number. When one is found the allocation can take place, the information can be stored  
\ior\_open:cn and finally the file can actually be opened.

```

5417 \cs_new_protected_nopar:Npn \iow_open:Nn #1#2 {
5418   \iow_close:N #1
5419   \int_set:Nn \l_iow_stream_int { \c_sixteen }
5420   \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n
5421   \int_compare:nTF { \l_iow_stream_int = \c_sixteen }
5422     { \msg_kernel_error:nn { iow } { streams-exhausted } }
5423     {
5424       \iow_stream_alloc:N #1
5425       \prop_gput:Nvn \g_iow_streams_prop \l_iow_stream_int {#2}
5426       \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
5427     }
5428 }
5429 \cs_generate_variant:Nn \iow_open:Nn { c }
5430 \cs_new_protected_nopar:Npn \ior_open:Nn #1#2 {

```

```

5431 \ior_close:N #1
5432 \int_set:Nn \l_ior_stream_int { \c_sixteen }
5433 \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
5434 \int_compare:nTF { \l_ior_stream_int = \c_sixteen }
5435   { \msg_kernel_error:nn { ior } { streams-exhausted } }
5436   {
5437     \ior_stream_alloc:N #1
5438     \prop_gput:Nvn \g_ior_streams_prop \l_ior_stream_int {#2}
5439     \tex_openin:D #1#2 \scan_stop:
5440   }
5441 }
5442 \cs_generate_variant:Nn \ior_open:Nn { c }

```

\ior\_alloc\_write:n These functions are used to see if a particular stream is available. The property list  
\ior\_alloc\_read:n contains file names for streams in use, so any unused ones are for the taking.

```

5443 \cs_new_protected_nopar:Npn \ior_alloc_write:n #1 {
5444   \prop_if_in:NnF \g_ior_streams_prop {#1}
5445   {
5446     \int_set:Nn \l_ior_stream_int {#1}
5447     \tl_map_break:
5448   }
5449 }
5450 \cs_new_protected_nopar:Npn \ior_alloc_read:n #1 {
5451   \prop_if_in:NnF \g_ior_streams_prop {#1}
5452   {
5453     \int_set:Nn \l_ior_stream_int {#1}
5454     \tl_map_break:
5455   }
5456 }

```

\ior\_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  
\ior\_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\_ior\_tmp\_stream other managed stream is available before deciding to open a new one. If a new one is  
\g\_ior\_tmp\_stream needed, we get the number allocated by L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> to get ‘back on track’ with allocation.

```

5457 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1 {
5458   \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
5459   { \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream } }
5460   {
5461     </initex | package>
5462     <*package>
5463     \ior_stream_alloc_aux:
5464     \int_compare:nT { \l_ior_stream_int = \c_sixteen }
5465     {
5466       \ior_raw_new:N \g_ior_tmp_stream
5467       \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }

```

```

5468         \cs_gset_eq:cN
5469         { g_iow_ \int_use:N \l_iow_stream_int _stream }
5470         \g_iow_tmp_stream
5471     }
5472 </package>
5473 <*initex>
5474     \iow_raw_new:c { g_iow_ \int_use:N \l_iow_stream_int _stream }
5475 </initex>
5476 <*initex | package>
5477     \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream }
5478 }
5479 }
5480 </initex | package>
5481 <*package>
5482 \cs_new_protected_nopar:Npn \iow_stream_alloc_aux: {
5483     \int_incr:N \l_iow_stream_int
5484     \int_compare:nT
5485     { \l_iow_stream_int < \c_sixteen }
5486     {
5487         \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
5488         {
5489             \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
5490             { \iow_stream_alloc_aux: }
5491         }
5492         { \iow_stream_alloc_aux: }
5493     }
5494 }
5495 </package>
5496 <*initex | package>
5497 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1 {
5498     \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
5499     { \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream } }
5500     {
5501 </initex | package>
5502 <*package>
5503     \ior_stream_alloc_aux:
5504     \int_compare:nT { \l_ior_stream_int = \c_sixteen }
5505     {
5506         \ior_raw_new:N \g_ior_tmp_stream
5507         \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }
5508         \cs_gset_eq:cN
5509         { g_ior_ \int_use:N \l_iow_stream_int _stream }
5510         \g_ior_tmp_stream
5511     }
5512 </package>
5513 <*initex>
5514     \ior_raw_new:c { g_ior_ \int_use:N \l_ior_stream_int _stream }
5515 </initex>
5516 <*initex | package>
5517     \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream }

```

```

5518     }
5519   }
5520   </initex|package>
5521   <*package>
5522   \cs_new_protected_nopar:Npn \ior_stream_alloc_aux: {
5523     \int_incr:N \l_ior_stream_int
5524     \int_compare:nT
5525       { \l_ior_stream_int < \c_sixteen }
5526     {
5527       \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
5528       {
5529         \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int
5530         { \ior_stream_alloc_aux: }
5531       }
5532       { \ior_stream_alloc_aux: }
5533     }
5534   }
5535   </package>
5536   <*initex|package>

```

\iow\_close:N Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```

\iow_close:c
\iow_close:N
\iow_close:c
5537   \cs_new_protected_nopar:Npn \iow_close:N #1 {
5538     \cs_if_exist:NT #1
5539     {
5540       \int_compare:nF { #1 = \c_minus_one }
5541       {
5542         \tex_immediate:D \tex_closeout:D #1
5543         \prop_gdel:NV \g_iow_streams_prop #1
5544         \cs_gundefine:N #1
5545       }
5546     }
5547   }
5548   \cs_generate_variant:Nn \iow_close:N { c }
5549   \cs_new_protected_nopar:Npn \ior_close:N #1 {
5550     \cs_if_exist:NT #1
5551     {
5552       \int_compare:nF { #1 = \c_minus_one }
5553       {
5554         \tex_closein:D #1
5555         \prop_gdel:NV \g_ior_streams_prop #1
5556         \cs_gundefine:N #1
5557       }
5558     }
5559   }
5560   \cs_generate_variant:Nn \ior_close:N { c }

```

`\iow_open_streams:` Simply show the property lists.

```
\ior_open_streams:
5561 \cs_new_protected_nopar:Npn \iow_open_streams: {
5562   \prop_display:N \g_iow_streams_prop
5563 }
5564 \cs_new_protected_nopar:Npn \ior_open_streams: {
5565   \prop_display:N \g_ior_streams_prop
5566 }
```

Text for the error messages.

```
5567 \msg_kernel_new:nnnn { iow } { streams-exhausted }
5568 {Output streams exhausted}
5569 {%
5570   TeX can only open up to 16 output streams at one time.\\%
5571   All 16 are currently in use, and something wanted to open
5572   another one.%
5573 }
5574 \msg_kernel_new:nnnn { ior } { streams-exhausted }
5575 {Input streams exhausted}
5576 {%
5577   TeX can only open up to 16 input streams at one time.\\%
5578   All 16 are currently in use, and something wanted to open
5579   another one.%
5580 }
```

### 108.3 Immediate writing

`\iow_now:Nx` An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```
5581 \cs_new_protected_nopar:Npn \iow_now:Nx { \tex_immediate:D \iow_shipout_x:Nn }
```

`\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.

```
5582 \cs_new_protected_nopar:Npn \iow_now:Nn #1#2 {
5583   \iow_now:Nx #1 { \exp_not:n {#2} }
5584 }
```

`\iow_log:n` Now we redefine two functions for which we needed a definition very early on.

```
\iow_log:x
5585 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_iow_log_stream }
\iow_term:n
5586 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_iow_log_stream }
\iow_term:x
5587 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
5588 \cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_iow_term_stream }
```

`\iow_now_when_avail:Nn` For writing only if the stream requested is open at all.

```

5589 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nn #1 {
5590   \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 }
5591 }
5592 \cs_generate_variant:Nn \iow_now_when_avail:Nn { c }
5593 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nx #1 {
5594   \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 }
5595 }
5596 \cs_generate_variant:Nn \iow_now_when_avail:Nx { c }

```

`\iow_now_buffer_safe:Nn` Another type of writing onto an output stream is used for potentially long token sequences. We break the output lines at every blank in the second argument. This avoids the problem of buffer overflow when reading back, or badly broken lines on systems with limited file records. The only thing we have to take care of, is the danger of two blanks in succession since these get converted into a `\par` when we read the stuff back. But this can happen only if things like two spaces find their way into the second argument. Usually, multiple spaces are removed by T<sub>E</sub>X's scanner.

`\iow_now_buffer_safe:Nx`

`\iow_now_buffer_safe_expanded_aux:w`

```

5597 \cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nn {
5598   \iow_now_buffer_safe_aux:w \iow_now:Nn
5599 }
5600 \cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nx {
5601   \iow_now_buffer_safe_aux:w \iow_now:Nn
5602 }
5603 \cs_new_protected_nopar:Npn \iow_now_buffer_safe_aux:w #1#2#3 {
5604   \group_begin: \tex_newlinechar:D'\ #1#2 {#3} \group_end:
5605 }

```

## 108.4 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive.

`\iow_shipout_x:Nx`

```

5606 \cs_set_eq:NN \iow_shipout_x:Nn \tex_write:D
5607 \cs_generate_variant:Nn \iow_shipout_x:Nn {Nx }

```

`\iow_shipout:Nn` With  $\epsilon$ -T<sub>E</sub>X available deferred writing is easy.

`\iow_shipout:Nx`

```

5608 \cs_new_protected_nopar:Npn \iow_shipout:Nn #1#2 {
5609   \iow_shipout_x:Nn #1 { \exp_not:n {#2} }
5610 }
5611 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }

```

## 109 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```

5612 \cs_new_nopar:Npn \iow_newline: { ^^J }

```

`\iow_char:N` Function to write any escaped char to an output stream.

```
5613 \cs_new:Npn \iow_char:N #1 { \cs_to_str:N #1 }
```

## 109.1 Reading input

`\if_eof:w` A simple primitive renaming.

```
5614 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.  
`\ior_if_eof:NTF` As the pool model means that closed streams are undefined control sequences, the test has two parts.

```
5615 \prg_new_conditional:Nnn \ior_if_eof:N { p , TF , T , F } {
5616   \cs_if_exist:NTF #1
5617   { \tex_ifeof:D #1 \prg_return_true: \else: \prg_return_false: \fi: }
5618   { \prg_return_true: }
5619 }
```

`\ior_to:NN` And here we read from files.

```
\ior_gto:NN
5620 \cs_new_protected_nopar:Npn \ior_to:NN #1#2 {
5621   \tex_read:D #1 to #2
5622 }
5623 \cs_new_protected_nopar:Npn \ior_gto:NN {
5624   \pref_global:D \ior_to:NN
5625 }
5626 </initex | package>
```

## 110 l3msg implementation

The usual lead-off.

```
5627 <*package>
5628 \ProvidesExplPackage
5629   {\filename}{\filedate}{\fileversion}{\filedescription}
5630 \package_check_loaded_expl:
5631 </package>
5632 <*initex | package>
```

L<sup>A</sup>T<sub>E</sub>X is handling context, so the T<sub>E</sub>X “noise” is turned down.

```
5633 \int_set:Nn \tex_errorcontextlines:D { \c_minus_one }
```

## 110.1 Variables and constants

`\c_msg_error_tl` Header information.

`\c_msg_warning_tl`

`\c_msg_info_tl`

```
5634 \tl_const:Nn \c_msg_error_tl { error }
5635 \tl_const:Nn \c_msg_warning_tl { warning }
5636 \tl_const:Nn \c_msg_info_tl { info }
```

`\msg_fatal_text:n`

Contextual header/footer information.

`\msg_see_documentation_text:n`

```
5637 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
5638 \cs_new:Npn \msg_see_documentation_text:n #1
5639 { See~the~#1~documentation~for~further~information }
```

`\c_msg_coding_error_text_tl`

Simple pieces of text for messages.

`\c_msg_fatal_text_tl`

`\c_msg_help_text_tl`

`\c_msg_kernel_bug_text_tl`

`\c_msg_kernel_bug_more_text_tl`

`\c_msg_no_info_text_tl`

`\c_msg_return_text_tl`

```
5640 \tl_const:Nn \c_msg_coding_error_text_tl {
5641   This~is~a~coding~error.
5642   \msg_two_newlines:
5643 }
5644 \tl_const:Nn \c_msg_fatal_text_tl {
5645   This~is~a~fatal~error:~LaTeX~will~abort
5646 }
5647 \tl_const:Nn \c_msg_help_text_tl {
5648   For~immediate~help~type~H~<return>
5649 }
5650 \tl_const:Nn \c_msg_kernel_bug_text_tl {
5651   This~is~a~LaTeX~bug:~check~coding!
5652 }
5653 \tl_const:Nn \c_msg_kernel_bug_more_text_tl {
5654   There~is~a~coding~bug~somewhere~around~here.  \\\
5655   This~probably~needs~examining~by~an~expert.
5656   \c_msg_return_text_tl
5657 }
5658 \tl_const:Nn \c_msg_no_info_text_tl {
5659   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
5660   \c_msg_return_text_tl
5661 }
5662 \tl_const:Nn \c_msg_return_text_tl {
5663   \\\ \\\
5664   Try~typing~<return>~to~proceed.
5665   \\\
5666   If~that~doesn't~work,~type~X~<return>~to~quit
5667 }
```

`\c_msg_hide_tl<spaces>`

An empty variable with a number of (category code 11) periods at the end of its name. This is used to push the T<sub>E</sub>X part of an error message “off the screen”.

```
5668 \group_begin:
```

```

5669 \char_make_letter:N \.
5670 \tl_to_lowercase:n {
5671   \group_end:
5672   \tl_const:Nn \c_msg_hide_tl.....
5673   { }
5674 }

```

`\c_msg_on_line_tl` Text for “on line”.

```

5675 \tl_const:Nn \c_msg_on_line_tl { on~line }

```

`\c_msg_text_prefix_tl` Prefixes for storage areas.

`\c_msg_more_text_prefix_tl`

```

5676 \tl_const:Nn \c_msg_text_prefix_tl { msg_text ~>~ }
5677 \tl_const:Nn \c_msg_more_text_prefix_tl { msg_text_more ~>~ }

```

`\l_msg_class_tl` For holding the current message method and that for redirection.

`\l_msg_current_class_tl`

`\l_msg_current_module_tl`

```

5678 \tl_new:N \l_msg_class_tl
5679 \tl_new:N \l_msg_current_class_tl
5680 \tl_new:N \l_msg_current_module_tl

```

`\l_msg_names_clist` Lists used for filtering.

```

5681 \clist_new:N \l_msg_names_clist

```

`\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`

```

5682 \prop_new:N \l_msg_redirect_classes_prop
5683 \prop_new:N \l_msg_redirect_names_prop

```

`\l_msg_redirect_classes_clist` To prevent an infinite loop.

```

5684 \clist_new:N \l_msg_redirect_classes_clist

```

`\l_msg_tmp_tl` A scratch variable.

```

5685 \tl_new:N \l_msg_tmp_tl

```

## 110.2 Output helper functions

`\msg_line_number:` For writing the line number nicely.

`\msg_line_context:`

```

5686 \cs_new_nopar:Npn \msg_line_number: {
5687   \toks_use:N \tex_inputlineno:D
5688 }

```

```

5689 \cs_new_nopar:Npn \msg_line_context: {
5690   \c_msg_on_line_tl
5691   \c_space_tl
5692   \msg_line_number:
5693 }

```

`\msg_newline:` Always forces a new line.  
`\msg_two_newlines:`

```

5694 \cs_new_nopar:Npn \msg_newline: { ^^J }
5695 \cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }

```

### 110.3 Generic functions

The lowest level functions make no assumptions about modules, *etc.*

`\msg_generic_new:nnn` Creating a new message is basically the same as the non-checking version, and so after a  
`\msg_generic_new:nn` check everything hands over.

```

5696 \cs_new_protected_nopar:Npn \msg_generic_new:nnn #1 {
5697   \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
5698   \msg_generic_set:nnn {#1}
5699 }
5700 \cs_new_protected_nopar:Npn \msg_generic_new:nn #1 {
5701   \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
5702   \msg_generic_set:nn {#1}
5703 }

```

`\msg_generic_set:nnn` Creating a message is quite simple. There must be a short text part, while the longer  
`\msg_generic_set:nn` text may or may not be available.

```

\msg_generic_set_clist:n
5704 \cs_new_protected_nopar:Npn \msg_generic_set:nnn #1#2#3 {
5705   \msg_generic_set_clist:n {#1}
5706   \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
5707   \cs_set:cpn { \c_msg_more_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#3}
5708 }
5709 \cs_new_protected_nopar:Npn \msg_generic_set:nn #1#2 {
5710   \msg_generic_set_clist:n {#1}
5711   \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
5712   \cs_set_eq:cN { \c_msg_more_text_prefix_tl #1 :xxxx } \c_undefined
5713 }
5714 \cs_new_protected_nopar:Npn \msg_generic_set_clist:n #1 {
5715   \clist_if_in:NnF \l_msg_names_clist { // #1 / } {
5716     \clist_put_right:Nn \l_msg_names_clist { // #1 / }
5717   }
5718 }

```

`\msg_direct_interrupt:xxxxx` The low-level interruption macro is rather opaque, unfortunately. The idea here is to  
`\msg_direct_interrupt:nn` create a message which hides all of TeX's own information by filling the output up with

dots. To achieve this, dots have to be letters. The odd `\c_msg_hide_tl<dots>` actually does the hiding: it is the large run of dots in the name that is important here. The meaning of `\\` is altered so that the explanation text is a simple run whilst the initial error has line-continuation shown.

```

5719 \group_begin:
5720 \char_set_lccode:nn {'\&} {'\ } % {
5721 \char_set_lccode:w '\} = '\ \scan_stop:
5722 \char_set_lccode:w '\& = '\!\scan_stop:
5723 \char_make_active:N \&
5724 \char_make_letter:N \.
5725 \tl_to_lowercase:n{
5726 \group_end:
5727 \cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 {
5728   \group_begin:
5729     \cs_set_nopar:Npn \\ {\msg_newline:}
5730     \cs_set_eq:NN \ \c_space_tl
5731     \tl_set:Nx \l_tmpa_tl {#5}
5732     \tl_set:Nx \l_tmpb_tl {\c_msg_no_info_text_tl}
5733     \msg_direct_interrupt_aux:n {#5}
5734     \tex_errhelp:D \l_msg_tmp_tl
5735     \cs_set_nopar:Npn \\ {\msg_newline: !~#3}
5736     \iow_term:x
5737       {\msg_newline: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!}
5738     \cs_set:Npn & {
5739       \tex_errmessage:D{
5740         \\ #1 \\ \\ #2 \\ \\ #4
5741         \tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
5742           { . \\ \c_msg_help_text_tl }
5743         \c_msg_hide_tl.....
5744       }
5745     }
5746   &
5747   \group_end:
5748 }
5749 }

5750 \cs_new_protected:Npn \msg_direct_interrupt_aux:n #1 {
5751   \cs_set_nopar:Npn \\ {\msg_newline: |~}
5752   \tl_if_empty:nTF {#1} {
5753     \tl_set:Nx \l_msg_tmp_tl { { \c_msg_no_info_text_tl } }
5754   }{
5755     \tl_set:Nx \l_msg_tmp_tl { {
5756       |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\
5757       #1^^J
5758       |.....^J
5759     } }
5760   }
5761 }

```

`\msg_direct_log:xx`    Printing to the log or terminal without a stop is rather easier.  
`\msg_direct_term:xx`

```

5762 \cs_new_protected:Npn \msg_direct_log:xx #1#2 {
5763   \group_begin:
5764     \cs_set:Npn \ { \msg_newline: #2 }
5765     \cs_set_eq:NN \ \c_space_tl
5766     \iow_log:x { #1 \msg_newline: }
5767   \group_end:
5768 }
5769 \cs_new_protected:Npn \msg_direct_term:xx #1#2 {
5770   \group_begin:
5771     \cs_set:Npn \ { \msg_newline: #2 }
5772     \cs_set_eq:NN \ \c_space_tl
5773     \iow_term:x { #1 \msg_newline: }
5774   \group_end:
5775 }

```

## 110.4 General functions

The main functions for messaging are built around the separation of module from the message name. These have short names as they will be widely used.

`\msg_new:nnnn`    For making messages: all aliases.  
`\msg_new:nnn`  
`\msg_set:nnnn`  
`\msg_set:nnn`

```

5776 \cs_new_protected_nopar:Npn \msg_new:nnnn #1#2 {
5777   \msg_generic_new:nnn { #1 / #2 }
5778 }
5779 \cs_new_protected_nopar:Npn \msg_new:nnn #1#2 {
5780   \msg_generic_new:nn { #1 / #2 }
5781 }
5782 \cs_new_protected_nopar:Npn \msg_set:nnnn #1#2 {
5783   \msg_generic_set:nnn { #1 / #2 }
5784 }
5785 \cs_new_protected_nopar:Npn \msg_set:nnn #1#2 {
5786   \msg_generic_set:nn { #1 / #2 }
5787 }

```

`\msg_class_new:nn`    Creating a new class produces three new functions, with varying numbers of arguments.  
`\msg_class_set:nn`    The `\msg_class_loop:n` function is set up so that redirection will work as desired.

```

5788 \cs_new_protected_nopar:Npn \msg_class_new:nn #1 {
5789   \chk_if_free_cs:c { msg_ #1 :nnxxxx }
5790   \prop_new:c { l_msg_redirect_ #1 _prop }
5791   \msg_class_set:nn {#1}
5792 }
5793 \cs_new_protected_nopar:Npn \msg_class_set:nn #1#2 {
5794   \prop_clear:c { l_msg_redirect_ #1 _prop }
5795   \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6 {

```

```

5796     \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6}
5797 }
5798 \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5 {
5799     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { }
5800 }
5801 \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4 {
5802     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { }
5803 }
5804 \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3 {
5805     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
5806 }
5807 \cs_set_protected:cpx { msg_ #1 :nn } ##1##2 {
5808     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
5809 }
5810 }

```

`\msg_use:nnnnxxxx` The main message-using macro creates two auxiliary functions: one containing the code for the message, and the second a loop function. There is then a hand-off to the system for checking if redirection is needed.

```

5811 \cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8 {
5812     \cs_set_nopar:Npn \msg_use_code: {
5813         \clist_clear:N \l_msg_redirect_classes_clist
5814         #2
5815     }
5816     \cs_set:Npn \msg_use_loop:n ##1 {
5817         \clist_if_in:NnTF \l_msg_redirect_classes_clist {#1} {
5818             \msg_kernel_error:nn { msg } { redirect-loop } {#1}
5819         }{
5820             \clist_put_right:Nn \l_msg_redirect_classes_clist {#1}
5821             \cs_if_exist:cTF { msg_ ##1 :nnxxxx } {
5822                 \use:c { msg_ ##1 :nnxxxx } {#3} {#4} {#5} {#6} {#7} {#8}
5823             }{
5824                 \msg_kernel_error:nnx { msg } { message-class-unknown } {##1}
5825             }
5826         }
5827     }
5828     \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 :xxxx } {
5829         \msg_use_aux:nnn {#1} {#3} {#4}
5830     }{
5831         \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4}
5832     }
5833 }

```

`\msg_use_code:` Blank definitions are initially created for these functions.

`\msg_use_loop:`

```

5834 \cs_new_nopar:Npn \msg_use_code: { }
5835 \cs_new:Npn \msg_use_loop:n #1 { }

```

`\msg_use_aux:nnn` The first auxiliary macro looks for a match by name: the most restrictive check.

```

5836 \cs_new_protected_nopar:Npn \msg_use_aux:nnn #1#2#3 {
5837   \tl_set:Nn \l_msg_current_class_tl {#1}
5838   \tl_set:Nn \l_msg_current_module_tl {#2}
5839   \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / } {
5840     \msg_use_loop_check:nn { names } { // #2 / #3 / }
5841   }{
5842     \msg_use_aux:nn {#1} {#2}
5843   }
5844 }

```

`\msg_use_aux:nn` The second function checks for general matches by module or for all modules.

```

5845 \cs_new_protected_nopar:Npn \msg_use_aux:nn #1#2 {
5846   \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2} {
5847     \msg_use_loop_check:nn {#1} {#2}
5848   }{
5849     \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } { * } {
5850       \msg_use_loop_check:nn {#1} { * }
5851     }{
5852       \msg_use_code:
5853     }
5854   }
5855 }

```

`\msg_use_loop_check:nn` When checking whether to loop, the same code is needed in a few places.

```

5856 \cs_new_protected:Npn \msg_use_loop_check:nn #1#2 {
5857   \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
5858   \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl {
5859     \msg_use_code:
5860   }{
5861     \msg_use_loop:n { \l_msg_class_tl }
5862   }
5863 }

```

`\msg_fatal:nnxxxx` For fatal errors, after the error message TeX bails out.

```

\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn
5864 \msg_class_new:nn { fatal } {
5865   \msg_direct_interrupt:xxxxx
5866   { \msg_fatal_text:n {#1} : ~ "#2" }
5867   {
5868     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5869   }
5870   {}
5871   { \msg_see_documentation_text:n {#1} }
5872   { \c_msg_fatal_text_tl }
5873   \tex_end:D
5874 }

```

`\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
5875 \msg_class_new:nn { error } {
5876   \msg_direct_interrupt:xxxxx
5877   { #1~ \c_msg_error_tl : ~ "#2" }
5878   {
5879     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5880   }
5881   {}
5882   { \msg_see_documentation_text:n {#1} }
5883   {
5884     \cs_if_exist:cTF { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
5885     {
5886       \use:c { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
5887       {#3} {#4} {#5} {#6}
5888     }
5889     { \c_msg_no_info_text_tl }
5890   }
5891 }

```

`\msg_warning:nnxxxx` Warnings are printed to the terminal.

```

\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn
5892 \msg_class_new:nn { warning } {
5893   \msg_direct_term:xx {
5894     \c_space_tl #1 ~ \c_msg_warning_tl :~
5895     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5896   }
5897   { ( #1 ) \c_space_tl \c_space_tl }
5898 }

```

`\msg_info:nnxxxx` Information only goes into the log.

```

\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn
5899 \msg_class_new:nn { info } {
5900   \msg_direct_log:xx {
5901     \c_space_tl #1~\c_msg_info_tl :~
5902     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5903   }
5904   { ( #1 ) \c_space_tl \c_space_tl }
5905 }

```

`\msg_log:nnxxxx` “Log” data is very similar to information, but with no extras added.

```

\msg_log:nnxxxx
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn
5906 \msg_class_new:nn { log } {
5907   \msg_direct_log:xx {
5908     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5909   }
5910   { }
5911 }

```

`\msg_trace:nnxxxx` Trace data is the same as log data, more or less  
`\msg_trace:nnxxx`  
`\msg_trace:nnxx`  
`\msg_trace:nnx`  
`\msg_trace:nn`

```

5912 \msg_class_new:nn { trace } {
5913   \msg_direct_log:xx {
5914     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5915   }
5916   { }
5917 }

```

`\msg_none:nnxxxx` The none message type is needed so that input can be gobbled.  
`\msg_none:nnxxx`  
`\msg_none:nnxx`  
`\msg_none:nnx`  
`\msg_none:nn`

```

5918 \msg_class_new:nn { none } { }

```

## 110.5 Redirection functions

`\msg_redirect_class:nn` Converts class one into class two.

```

5919 \cs_new_protected_nopar:Npn \msg_redirect_class:nn #1#2 {
5920   \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#2}
5921 }

```

`\msg_redirect_module:nnn` For when all messages of a class should be altered for a given module.

```

5922 \cs_new_protected_nopar:Npn \msg_redirect_module:nnn #1#2#3 {
5923   \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3}
5924 }

```

`\msg_redirect_name:nnn` Named message will always use the given class.

```

5925 \cs_new_protected_nopar:Npn \msg_redirect_name:nnn #1#2#3 {
5926   \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3}
5927 }

```

## 110.6 Kernel-specific functions

`\msg_kernel_new:nnnn` The kernel needs some messages of its own. These are created using pre-built functions.  
`\msg_kernel_new:nnn` Two functions are provided: one more general and one which only has the short text  
`\msg_kernel_set:nnnn` part.  
`\msg_kernel_set:nnn`

```

5928 \cs_new_protected_nopar:Npn \msg_kernel_new:nnnn #1#2 {
5929   \msg_new:nnnn { LaTeX } { #1 / #2 }
5930 }
5931 \cs_new_protected_nopar:Npn \msg_kernel_new:nnn #1#2 {
5932   \msg_new:nnn { LaTeX } { #1 / #2 }
5933 }
5934 \cs_new_protected_nopar:Npn \msg_kernel_set:nnnn #1#2 {
5935   \msg_set:nnnn { LaTeX } { #1 / #2 }

```

```

5936 }
5937 \cs_new_protected_nopar:Npn \msg_kernel_set:nnn #1#2 {
5938   \msg_set:nnn { LaTeX } { #1 / #2 }
5939 }

```

\msg\_kernel\_classes\_new:n Quickly make the fewer-arguments versions.

```

5940 \cs_new_protected_nopar:Npn \msg_kernel_classes_new:n #1 {
5941   \cs_new_protected:cpx { msg_kernel_ #1 :nnxxx } ##1##2##3##4##5
5942   {
5943     \exp_not:c { msg_kernel_ #1 :nnxxxx }
5944     {##1} {##2} {##3} {##4} {##5} { }
5945   }
5946   \cs_new_protected:cpx { msg_kernel_ #1 :nnxx } ##1##2##3##4
5947   {
5948     \exp_not:c { msg_kernel_ #1 :nnxxxx }
5949     {##1} {##2} {##3} {##4} { } { }
5950   }
5951   \cs_new_protected:cpx { msg_kernel_ #1 :nnx } ##1##2##3
5952   {
5953     \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
5954   }
5955   \cs_new_protected:cpx { msg_kernel_ #1 :nn } ##1##2
5956   {
5957     \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
5958   }
5959 }

```

\msg\_kernel\_fatal:nnxxxx Fatal kernel errors cannot be re-defined.

```

\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxx
\msg_kernel_fatal:nnx
\msg_kernel_fatal:nn
5960 \cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6 {
5961   \msg_direct_interrupt:xxxxx
5962   { \msg_fatal_text:n {LaTeX} }
5963   {
5964     \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5965     {#3} {#4} {#5} {#6}
5966   }
5967   {}
5968   { \msg_see_documentation_text:n {LaTeX3} }
5969   { \c_msg_fatal_text_tl }
5970   \tex_end:D
5971 }
5972 \msg_kernel_classes_new:n { fatal }

```

\msg\_kernel\_error:nnxxxx Neither can kernel errors.

```

\msg_kernel_error:nnxxx
\msg_kernel_error:nnxxx
\msg_kernel_error:nnxx
\msg_kernel_error:nnx
\msg_kernel_error:nn
5973 \cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6 {
5974   \msg_direct_interrupt:xxxxx
5975   { LaTeX~\c_msg_error_tl \c_space_tl "#2" }

```

```

5976 {
5977   \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5978   {#3} {#4} {#5} {#6}
5979 }
5980 {}
5981 { \msg_see_documentation_text:n {LaTeX3} }
5982 {
5983   \cs_if_exist:cTF
5984     { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5985     {
5986       \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5987       {#3} {#4} {#5} {#6}
5988     }
5989     { \c_msg_no_info_text_tl }
5990   }
5991 }
5992 \msg_kernel_classes_new:n { error }

```

```

\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

```

Life is much more simple for warnings and information messages, as these are just shortcuts to the standard classes.

```

5993 \cs_new_protected_nopar:Npn \msg_kernel_warning:nnxxxx #1#2 {
5994   \msg_warning:nnxxxx { LaTeX } { #1 / #2 }
5995 }
5996 \msg_kernel_classes_new:n { warning }
5997 \cs_new_protected_nopar:Npn \msg_kernel_info:nnxxxx #1#2 {
5998   \msg_info:nnxxxx { LaTeX } { #1 / #2 }
5999 }
6000 \msg_kernel_classes_new:n { info }

```

Error messages needed to actually implement the message system itself.

```

6001 \msg_kernel_new:nnnn { msg } { message-unknown }
6002 { Unknown~message~'#2'~for~module~'#1'. }
6003 {
6004   \c_msg_coding_error_text_tl
6005   LaTeX~was~asked~to~display~a~message~called~'#2'\
6006   by~the~module~'#1'~module:~this~message~does~not~exist.
6007   \c_msg_return_text_tl
6008 }
6009 \msg_kernel_new:nnnn { msg } { message-class-unknown }
6010 { Unknown~message~class~'#1'. }
6011 {
6012   LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
6013   this~was~never~defined.
6014 }
6015 \c_msg_return_text_tl
6016 }
6017 \msg_kernel_new:nnnn { msg } { redirect-loop }

```

```

6018 { Message~redirection~loop~for~message~class~'#1'. }
6019 {
6020   LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.\\
6021   The~original~message~here~has~been~lost.
6022   \c_msg_return_text_tl
6023 }

```

`\msg_kernel_bug:x` The L<sup>A</sup>T<sub>E</sub>X coding bug error gets re-visited here.

```

6024 \cs_set_protected:Npn \msg_kernel_bug:x #1 {
6025   \msg_direct_interrupt:xxxxx
6026   { \c_msg_kernel_bug_text_tl }
6027   { #1 }
6028   {}
6029   { \msg_see_documentation_text:n {LaTeX3} }
6030   { \c_msg_kernel_bug_more_text_tl }
6031 }

6032 </initex | package>

```

## 111 l3box implementation

Announce and ensure that the required packages are loaded.

```

6033 <*package>
6034 \ProvidesExplPackage
6035   {\filename}{\filedate}{\fileversion}{\filedescription}
6036   \package_check_loaded_expl:
6037 </package>
6038 <*initex | package>

```

The code in this module is very straight forward so I'm not going to comment it very extensively.

### 111.1 Generic boxes

`\box_new:N` Defining a new `<box>` register.  
`\box_new:c`

```

6039 <*initex>
6040 \alloc_new:nnnN {box} \c_zero \c_max_register_int \tex_mathchardef:D

```

Now, remember that `\box255` has a special role in T<sub>E</sub>X, it shouldn't be allocated...

```

6041 \seq_put_right:Nn \g_box_allocation_seq {255}
6042 </initex>

```

When we run on top of L<sup>A</sup>T<sub>E</sub>X, we just use its allocation mechanism.

```

6043 <*package>
6044 \cs_new_protected:Npn \box_new:N #1 {
6045   \chk_if_free_cs:N #1
6046   \newbox #1
6047 }
6048 </package>

6049 \cs_generate_variant:Nn \box_new:N {c}

```

The primitives for testing if a  $\langle box \rangle$  is empty/void or which type of box it is.

```

\if_hbox:N
\if_vbox:N
\if_box_empty:N
6050 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
6051 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
6052 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_horizontal:NTF
\box_if_horizontal:cTF
\box_if_vertical:NTF
\box_if_vertical:cTF
6053 \prg_new_conditional:Nnn \box_if_horizontal:N {p,TF,T,F} {
6054   \tex_ifhbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
6055 }
6056 \prg_new_conditional:Nnn \box_if_vertical:N {p,TF,T,F} {
6057   \tex_ifvbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
6058 }
6059 \cs_generate_variant:Nn \box_if_horizontal_p:N {c}
6060 \cs_generate_variant:Nn \box_if_horizontal:N {NTF} {c}
6061 \cs_generate_variant:Nn \box_if_horizontal:NT {c}
6062 \cs_generate_variant:Nn \box_if_horizontal:NF {c}
6063 \cs_generate_variant:Nn \box_if_vertical_p:N {c}
6064 \cs_generate_variant:Nn \box_if_vertical:N {NTF} {c}
6065 \cs_generate_variant:Nn \box_if_vertical:NT {c}
6066 \cs_generate_variant:Nn \box_if_vertical:NF {c}

```

Testing if a  $\langle box \rangle$  is empty/void.

```

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:NTF
\box_if_empty:cTF
6067 \prg_new_conditional:Nnn \box_if_empty:N {p,TF,T,F} {
6068   \tex_ifvoid:D #1 \prg_return_true: \else: \prg_return_false: \fi:
6069 }
6070 \cs_generate_variant:Nn \box_if_empty_p:N {c}
6071 \cs_generate_variant:Nn \box_if_empty:N {NTF} {c}
6072 \cs_generate_variant:Nn \box_if_empty:NT {c}
6073 \cs_generate_variant:Nn \box_if_empty:NF {c}

```

Assigning the contents of a box to be another box.

```

\box_set_eq:NN
\box_set_eq:cN
\box_set_eq:Nc
\box_set_eq:cc
6074 \cs_new_protected_nopar:Npn \box_set_eq:NN #1#2 {\tex_setbox:D #1 \tex_copy:D #2}
6075 \cs_generate_variant:Nn \box_set_eq:NN {cN,Nc,cc}

```

<pre> \box_set_eq_clear:NN \box_set_eq_clear:cN \box_set_eq_clear:Nc \box_set_eq_clear:cc </pre>	<p>Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).</p> <pre> 6076 \cs_new_protected_nopar:Npn \box_set_eq_clear:NN #1#2 {\tex_setbox:D #1 \tex_box:D #2} 6077 \cs_generate_variant:Nn \box_set_eq_clear:NN {cN,Nc,cc} </pre>
<pre> \box_gset_eq:NN \box_gset_eq:cN \box_gset_eq:Nc \box_gset_eq:cc \box_gset_eq_clear:NN \box_gset_eq_clear:cN \box_gset_eq_clear:Nc \box_gset_eq_clear:cc </pre>	<p>Global version of the above.</p> <pre> 6078 \cs_new_protected_nopar:Npn \box_gset_eq:NN {\pref_global:D\box_set_eq:NN} 6079 \cs_generate_variant:Nn \box_gset_eq:NN {cN,Nc,cc} 6080 \cs_new_protected_nopar:Npn \box_gset_eq_clear:NN {\pref_global:D\box_set_eq_clear:NN} 6081 \cs_generate_variant:Nn \box_gset_eq_clear:NN {cN,Nc,cc} </pre>
<pre> \box_set_to_last:N \box_set_to_last:c \box_gset_to_last:N \box_gset_to_last:c </pre>	<p>A different name for this read-only primitive.</p> <pre> 6082 \cs_new_eq:NN \l_last_box \tex_lastbox:D </pre>
<pre> \box_set_to_last:N \box_set_to_last:c \box_gset_to_last:N \box_gset_to_last:c </pre>	<p>Set a box to the previous box.</p> <pre> 6083 \cs_new_protected_nopar:Npn \box_set_to_last:N #1{\tex_setbox:D#1\l_last_box} 6084 \cs_generate_variant:Nn \box_set_to_last:N {c} 6085 \cs_new_protected_nopar:Npn \box_gset_to_last:N {\pref_global:D \box_set_to_last:N} 6086 \cs_generate_variant:Nn \box_gset_to_last:N {c} </pre>
<pre> \box_move_left:nn \box_move_right:nn \box_move_up:nn \box_move_down:nn </pre>	<p>Move box material in different directions.</p> <pre> 6087 \cs_new:Npn \box_move_left:nn #1#2{\tex_moveleft:D\dim_eval:n{#1} #2} 6088 \cs_new:Npn \box_move_right:nn #1#2{\tex_moveright:D\dim_eval:n{#1} #2} 6089 \cs_new:Npn \box_move_up:nn #1#2{\tex_raise:D\dim_eval:n{#1} #2} 6090 \cs_new:Npn \box_move_down:nn #1#2{\tex_lower:D\dim_eval:n{#1} #2} </pre>
<pre> \box_clear:N \box_clear:c \box_gclear:N \box_gclear:c </pre>	<p>Clear a <math>\langle box \rangle</math> register.</p> <pre> 6091 \cs_new_protected_nopar:Npn \box_clear:N #1{\box_set_eq_clear:NN #1 \c_empty_box } 6092 \cs_generate_variant:Nn \box_clear:N {c} 6093 \cs_new_protected_nopar:Npn \box_gclear:N {\pref_global:D\box_clear:N} 6094 \cs_generate_variant:Nn \box_gclear:N {c} </pre>
<pre> \box_ht:N \box_ht:c \box_dp:N \box_dp:c \box_wd:N \box_wd:c </pre>	<p>Accessing the height, depth, and width of a <math>\langle box \rangle</math> register.</p> <pre> 6095 \cs_new_eq:NN \box_ht:N \tex_ht:D 6096 \cs_new_eq:NN \box_dp:N \tex_dp:D 6097 \cs_new_eq:NN \box_wd:N \tex_wd:D 6098 \cs_generate_variant:Nn \box_ht:N {c} 6099 \cs_generate_variant:Nn \box_dp:N {c} 6100 \cs_generate_variant:Nn \box_wd:N {c} </pre>

`\box_set_ht:Nn` Measuring is easy: all primitive work. These primitives are not expandable, so the derived  
`\box_set_ht:cn` functions are not either.

`\box_set_dp:Nn`  
`\box_set_dp:cn`  
`\box_set_wd:Nn`  
`\box_set_wd:cn`

```

6101 \cs_new_protected_nopar:Npn \box_set_dp:Nn #1#2 {
6102   \box_dp:N #1 \etex_dimexpr:D #2 \scan_stop:
6103 }
6104 \cs_new_protected_nopar:Npn \box_set_ht:Nn #1#2 {
6105   \box_ht:N #1 \etex_dimexpr:D #2 \scan_stop:
6106 }
6107 \cs_new_protected_nopar:Npn \box_set_wd:Nn #1#2 {
6108   \box_wd:N #1 \etex_dimexpr:D #2 \scan_stop:
6109 }
6110 \cs_generate_variant:Nn \box_set_ht:Nn { c }
6111 \cs_generate_variant:Nn \box_set_dp:Nn { c }
6112 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

`\box_use_clear:N` Using a  $\langle box \rangle$ . These are just T<sub>E</sub>X primitives with meaningful names.

`\box_use_clear:c`  
`\box_use:N`  
`\box_use:c`

```

6113 \cs_new_eq:NN \box_use_clear:N \tex_box:D
6114 \cs_generate_variant:Nn \box_use_clear:N {c}
6115 \cs_new_eq:NN \box_use:N \tex_copy:D
6116 \cs_generate_variant:Nn \box_use:N {c}

```

`\box_show:N` Show the contents of a box and write it into the log file.

`\box_show:c`

```

6117 \cs_set_eq:NN \box_show:N \tex_showbox:D
6118 \cs_generate_variant:Nn \box_show:N {c}

```

`\c_empty_box` We allocate some  $\langle box \rangle$  registers here (and borrow a few from L<sup>A</sup>T<sub>E</sub>X).

`\l_tmpa_box`  
`\l_tmpb_box`

```

6119 <package>\cs_set_eq:NN \c_empty_box \voidb@x
6120 <package>\cs_new_eq:NN \l_tmpa_box \@tempboxa
6121 <initex>\box_new:N \c_empty_box
6122 <initex>\box_new:N \l_tmpa_box
6123 \box_new:N \l_tmpb_box

```

## 111.2 Vertical boxes

`\vbox:n` Put a vertical box directly into the input stream.

`\vbox_top:n`

```

6124 \cs_new_protected_nopar:Npn \vbox:n {\tex_vbox:D \scan_stop:}
6125 \cs_new_protected_nopar:Npn \vbox_top:n {\tex_vtop:D \scan_stop:}

```

`\vbox_set:Nn` Storing material in a vertical box with a natural height.

`\vbox_set:cn`  
`\vbox_gset:Nn`  
`\vbox_gset:cn`

```

6126 \cs_new_protected:Npn \vbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_vbox:D {#2}}
6127 \cs_generate_variant:Nn \vbox_set:Nn {cn}
6128 \cs_new_protected_nopar:Npn \vbox_gset:Nn {\pref_global:D \vbox_set:Nn}
6129 \cs_generate_variant:Nn \vbox_gset:Nn {cn}

```

<code>\vbox_set_top:Nn</code>	Storing material in a vertical box with a natural height and reference point at the baseline
<code>\vbox_set_top:cn</code>	of the first object in the box.
<code>\vbox_gset_top:Nn</code>	
<code>\vbox_gset_top:cn</code>	<pre> 6130 \cs_new_protected:Npn \vbox_set_top:Nn #1#2 {\tex_setbox:D #1 \tex_vtop:D {#2}} 6131 \cs_generate_variant:Nn \vbox_set_top:Nn {cn} 6132 \cs_new_protected_nopar:Npn \vbox_gset_top:Nn {\pref_global:D \vbox_set_top:Nn} 6133 \cs_generate_variant:Nn \vbox_gset_top:Nn {cn} </pre>
<code>\vbox_set_to_ht:Nnn</code>	Storing material in a vertical box with a specified height.
<code>\vbox_set_to_ht:cnn</code>	
<code>\vbox_gset_to_ht:Nnn</code>	
<code>\vbox_gset_to_ht:cnn</code>	
<code>\vbox_gset_to_ht:ccn</code>	<pre> 6134 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3 { 6135   \tex_setbox:D #1 \tex_vbox:D to #2 {#3} 6136 } 6137 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn {cnn} 6138 \cs_new_protected_nopar:Npn \vbox_gset_to_ht:Nnn {\pref_global:D \vbox_set_to_ht:Nnn} 6139 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn {cnn,ccn} </pre>
<code>\vbox_set_inline_begin:N</code>	Storing material in a vertical box. This type is useful in environment definitions.
<code>\vbox_set_inline_end:</code>	
<code>\vbox_gset_inline_begin:N</code>	
<code>\vbox_gset_inline_end:</code>	<pre> 6140 \cs_new_protected_nopar:Npn \vbox_set_inline_begin:N #1 { 6141   \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token } 6142 \cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token 6143 \cs_new_protected_nopar:Npn \vbox_gset_inline_begin:N { 6144   \pref_global:D \vbox_set_inline_begin:N } 6145 \cs_new_eq:NN \vbox_gset_inline_end: \c_group_end_token </pre>
<code>\vbox_to_ht:nn</code>	Put a vertical box directly into the input stream.
<code>\vbox_to_zero:n</code>	
	<pre> 6146 \cs_new_protected:Npn \vbox_to_ht:nn #1#2{\tex_vbox:D to \dim_eval:n{#1}{#2}} 6147 \cs_new_protected:Npn \vbox_to_zero:n #1 {\tex_vbox:D to \c_zero_dim {#1}} </pre>
<code>\vbox_set_split_to_ht:Nnn</code>	Splitting a vertical box in two.
	<pre> 6148 \cs_new_protected_nopar:Npn \vbox_set_split_to_ht:Nnn #1#2#3{ 6149   \tex_setbox:D #1 \tex_vsplit:D #2 to #3 6150 } </pre>
<code>\vbox_unpack:N</code>	Unpacking a box and if requested also clear it.
<code>\vbox_unpack:c</code>	
<code>\vbox_unpack_clear:N</code>	
<code>\vbox_unpack_clear:c</code>	<pre> 6151 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D 6152 \cs_generate_variant:Nn \vbox_unpack:N {c} 6153 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D 6154 \cs_generate_variant:Nn \vbox_unpack_clear:N {c} </pre>

### 111.3 Horizontal boxes

`\hbox:n` Put a horizontal box directly into the input stream.

```

6155 \cs_new_protected_nopar:Npn \hbox:n {\tex_hbox:D \scan_stop:}

```

<code>\hbox_set:Nn</code> <code>\hbox_set:cn</code> <code>\hbox_gset:Nn</code> <code>\hbox_gset:cn</code>	<p>Assigning the contents of a box to be another box. This clears the second box globally (that's how T<sub>E</sub>X does it).</p> <pre> 6156 \cs_new_protected:Npn \hbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_hbox:D {#2}} 6157 \cs_generate_variant:Nn \hbox_set:Nn {cn} 6158 \cs_new_protected_nopar:Npn \hbox_gset:Nn {\pref_global:D \hbox_set:Nn} 6159 \cs_generate_variant:Nn \hbox_gset:Nn {cn} </pre>
<code>\hbox_set_to_wd:Nnn</code> <code>\hbox_set_to_wd:cnn</code> <code>\hbox_gset_to_wd:Nnn</code> <code>\hbox_gset_to_wd:cnn</code>	<p>Storing material in a horizontal box with a specified width.</p> <pre> 6160 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3 { 6161   \tex_setbox:D #1 \tex_hbox:D to \dim_eval:n{#2} {#3} 6162 } 6163 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn {cnn} 6164 \cs_new_protected_nopar:Npn \hbox_gset_to_wd:Nnn {\pref_global:D \hbox_set_to_wd:Nnn } 6165 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn {cnn} </pre>
<code>\hbox_set_inline_begin:N</code> <code>\hbox_set_inline_begin:c</code> <code>\hbox_set_inline_end:</code> <code>\hbox_gset_inline_begin:N</code> <code>\hbox_gset_inline_begin:c</code> <code>\hbox_gset_inline_end:</code>	<p>Storing material in a horizontal box. This type is useful in environment definitions.</p> <pre> 6166 \cs_new_protected_nopar:Npn \hbox_set_inline_begin:N #1 { 6167   \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token 6168 } 6169 \cs_generate_variant:Nn \hbox_set_inline_begin:N {c} 6170 \cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token 6171 \cs_new_protected_nopar:Npn \hbox_gset_inline_begin:N { 6172   \pref_global:D \hbox_set_inline_begin:N 6173 } 6174 \cs_generate_variant:Nn \hbox_gset_inline_begin:N {c} 6175 \cs_new_eq:NN \hbox_gset_inline_end: \c_group_end_token </pre>
<code>\hbox_to_wd:nn</code> <code>\hbox_to_zero:n</code>	<p>Put a horizontal box directly into the input stream.</p> <pre> 6176 \cs_new_protected:Npn \hbox_to_wd:nn #1#2 {\tex_hbox:D to #1 {#2}} 6177 \cs_new_protected:Npn \hbox_to_zero:n #1 {\tex_hbox:D to \c_zero_skip {#1}} </pre>
<code>\hbox_overlap_left:n</code> <code>\hbox_overlap_right:n</code>	<p>Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.</p> <pre> 6178 \cs_new_protected:Npn \hbox_overlap_left:n #1 {\hbox_to_zero:n {\tex_hss:D #1}} 6179 \cs_new_protected:Npn \hbox_overlap_right:n #1 {\hbox_to_zero:n {#1 \tex_hss:D}} </pre>
<code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <code>\hbox_unpack_clear:N</code> <code>\hbox_unpack_clear:c</code>	<p>Unpacking a box and if requested also clear it.</p> <pre> 6180 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D 6181 \cs_generate_variant:Nn \hbox_unpack:N {c} 6182 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D 6183 \cs_generate_variant:Nn \hbox_unpack_clear:N {c}  6184 &lt;/initex   package&gt;  6185 &lt;*showmemory&gt; 6186 \showMemUsage 6187 &lt;/showmemory&gt; </pre>

## 112 l3xref implementation

### 112.1 Internal functions and variables

<code>\g_xref_all_curr_immediate_fields_prop</code> <code>\g_xref_all_curr_deferred_fields_prop</code>
---

 What they say they are :)

<code>\xref_write</code>
--------------------------

 A stream for writing cross references, although they are not required to be in a separate file.

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

`\xref_define_label:nn {<name>} {<plist contents>}`

Define the property list for each label; used internally by `\xref_set_label:n`.

### 112.2 Module code

We start by ensuring that the required packages are loaded.

```
6188 <*package>
6189 \ProvidesExplPackage
6190   {\filename}{\filedate}{\fileversion}{\filedescription}
6191 \package_check_loaded_expl:
6192 </package>
6193 <*initex | package>
```

There are two kinds of information, namely information which is *immediate* like a section title and then there's *deferred* information like page numbers. Each reference type belong to one of these categories, which we save internally as the property lists `\g_xref_all_curr_immediate_fields_prop` and `\g_xref_all_curr_deferred_fields_prop` and the reference type `<xyz>` exists as the key-info pair `\xref_<xyz>_key {\l_xref_curr_<xyz>_tl}` on one of these lists. This way each new entry type is just added as another key-info pair.

When the cross references are generated at the beginning of the document each will turn into a control sequence. Thus `\label{mylab}` will internally refer to the property list `\g_xref_mylab_prop`.

The extraction of values from this property list can be done in several different ways but we want to keep the operation expandable. Therefore we use a dedicated function for each type of cross reference, which looks like this:

```
\xref_get_value_xyz_aux:w -> #1 \xref_xyz_key #2#3\q_nil{#2}
```

This will throw away all the bits we don't need. In case `xyz` is the first on the `mylab` property list `#1` is empty, if it's the last key-info pair `#3` is empty. The value of the field can be extracted with the function `\xref_get_value:nn` where the first argument is the type and the second the label name so here it would be `\xref_get_value:nn {xyz} {mylab}`.

`_all_curr_immediate_fields_prop` `_all_curr_deferred_fields_prop` The two main property lists for storing information. They contain key-info pairs for all known types.

```
6194 \prop_new:N \g_xref_all_curr_immediate_fields_prop
6195 \prop_new:N \g_xref_all_curr_deferred_fields_prop
```

`\xref_new:nn` `\xref_deferred_new:nn` `\xref_new_aux:nnn` Setting up a new cross reference type is fairly straight forward when we follow the game plan mentioned earlier.

```
6196 \cs_new_nopar:Npn \xref_new:nn {\xref_new_aux:nnn{immediate}}
6197 \cs_new_nopar:Npn \xref_deferred_new:nn {\xref_new_aux:nnn{deferred}}
6198 \cs_new_nopar:Npn \xref_new_aux:nnn #1#2#3{
```

First put the new type in the relevant property list.

```
6199 \prop_gput:ccx {g_xref_all_curr_ #1 _fields_prop}
6200 { xref_ #2 _key }
6201 { \exp_not:c {l_xref_curr_#2_tl } }
```

Then define the key to be a protected macro.<sup>12</sup>

```
6202 \cs_set_protected_nopar:cpn { xref_#2_key }{ }
6203 \tl_new:cn{l_xref_curr_#2_tl}{#3}
```

Now for the function extracting the value of a reference. We could do this with a simple `\prop_if_in` thing put since we want to do things in an expandable way we make a separate grabber for each type—this is also faster. The grabber function can be defined by using an intricate construction of `\exp_after:wN` and other goodies but I prefer readable code. The end result for the input `xyz` is

```
\cs_set_nopar:Npn\xref_get_value_xyz_aux:w #1\xref_xyz_key #2#3\q_nil{#2}
```

```
6204 \toks_set:Nx \l_tmpa_toks {
6205   \exp_not:n { \cs_set_nopar:cpn {xref_get_value_#2_aux:w} ##1 }
6206   \exp_not:N \q_prop
6207   \exp_not:c { xref_#2_key }
6208   \exp_not:N \q_prop
6209 }
6210 \toks_use:N \l_tmpa_toks ##2 ##3\q_nil {##2}
6211 }
```

---

<sup>12</sup>We could also set it equal to `\scan_stop:` but this just feels “cleaner”.

`\xref_get_value:nn` Getting the correct value for a given label-type pair is a matter of connecting the correct grabber functions and property list.

```

6212 \cs_new_nopar:Npn \xref_get_value:nn #1#2 {
6213   \cs_if_exist:cTF{g_xref_#2_prop}
6214   {

```

This next expansion may look a little weird but it isn't if you think about it!

```

6215     \exp_args:NcNc \exp_after:wN {xref_get_value_#1_aux:w}
6216     \toks_use:N {g_xref_#2_prop}

```

Better put in the stop marker.

```

6217     \q_nil
6218   }
6219   {??}
6220 }

```

Temporary! We expand the property list and so we can't have the `\q_prop` marker just expand!

```

6221 \cs_set_nopar:Npn \exp_after:cc #1#2 {
6222   \exp_after:wN \exp_after:wN
6223   \cs:w #1\exp_after:wN\cs_end: \cs:w #2\cs_end:
6224 }
6225 \cs_set_protected:Npn \q_prop {\q_prop}

```

`\xref_define_label:nn` Define the property list for each label. We better do this in two steps because the special catcode regime is in effect and since some of the info fields are very likely to contain actual text, we better make sure spaces aren't ignored! As for the meaning of other characters then it is a possibility to also have a field containing catcode instructions which can then be activated with `\etex_scantokens:D`.

```

6226 \cs_new_protected_nopar:Npn \xref_define_label:nn {
6227   \group_begin:
6228   \char_set_catcode:nn {'\ }c_ten
6229   \xref_define_label_aux:nn
6230 }

```

If the label is already taken we have a multiply defined label and we should do something about it. For now we don't do anything spectacular.

```

6231 \cs_new_nopar:Npn \xref_define_label_aux:nn #1#2 {
6232   \cs_if_free:cTF{g_xref_#1_prop}
6233   {\prop_new:c{g_xref_#1_prop}}{\WARNING}
6234   \toks_gset:cn{g_xref_#1_prop}{#2}
6235   \group_end:
6236 }

```

`\xref_set_label:n` Then the generic command for setting a label. We expand the immediate labels fully before calling the write function but make sure the deferred fields aren't expanded just yet. Due to property lists being implemented as token list registers we must expand the 'immediate' fields twice.

```

6237 \cs_set_nopar:Npn \xref_set_label:n #1{
6238   \cs_set_nopar:Npx \xref_tmp:w{\toks_use:N\g_xref_all_curr_immediate_fields_prop}
6239   \exp_args:NNx\iow_shipout_x:Nn \xref_write{
6240     \xref_define_label:nn {#1} {
6241       \xref_tmp:w
6242       \toks_use:N \g_xref_all_curr_deferred_fields_prop
6243     }
6244   }
6245 }
```

That's it (for now).

```

6246 </initex | package>

6247 <*showmemory>
6248 \showMemUsage
6249 </showmemory>
```

## 113 l3xref test file

```

6250 <*testfile>
6251 \documentclass{article}
6252 \usepackage{l3xref}
6253 \ExplSyntaxOn
6254 \cs_set_nopar:Npn \startrecording {\iow_open:Nn \xref_write {\jobname.xref}}
6255 \cs_set_nopar:Npn \DefineCrossReferences {
6256   \group_begin:
6257     \ExplSyntaxNamesOn
6258     \InputIfFileExists{\jobname.xref}{\{}}
6259   \group_end:
6260 }
6261 \AtBeginDocument{\DefineCrossReferences\startrecording}
6262
6263 \xref_new:nn {name}{\{}}
6264 \cs_set_nopar:Npn \setname{\tl_set:Nn\l_xref_curr_name_tl}
6265 \cs_set_nopar:Npn \getname{\xref_get_value:nn{name}}
6266
6267 \xref_deferred_new:nn {page}{\thepage}
6268 \cs_set_nopar:Npn \getpage{\xref_get_value:nn{page}}
6269
6270 \xref_deferred_new:nn {valuepage}{\number\value{page}}
6271 \cs_set_nopar:Npn \getvaluepage{\xref_get_value:nn{valuepage}}
6272
```

```

6273 \cs_set_eq:NN \setlabel \xref_set_label:n
6274
6275 \ExplSyntaxOff
6276 \begin{document}
6277 \pagenumbering{roman}
6278
6279 Text\setname{This is a name}\setlabel{testlabel1}. More
6280 text\setname{This is another name}\setlabel{testlabel2}. \clearpage
6281
6282 Text\setname{This is a third name}\setlabel{testlabel3}. More
6283 text\setname{Hello World!}\setlabel{testlabel4}. \clearpage
6284
6285 \pagenumbering{arabic}
6286
6287 Text\setname{Name 5}\setlabel{testlabel5}. More text\setname{Name
6288 6}\setlabel{testlabel6}. \clearpage
6289
6290 Text\setname{Name 7}\setlabel{testlabel 7}. More text\setname{Name
6291 8}\setlabel{testlabel8}. \clearpage
6292
6293 Now let's extract some values. \getname{testlabel1} on page
6294 \getpage{testlabel1} with value \getvaluepage{testlabel1}.
6295
6296 Now let's extract some values. \getname{testlabel 7} on page
6297 \getpage{testlabel 7} with value \getvaluepage{testlabel 7}.
6298 \end{document}
6299 </testfile>

```

## 114 l3keyval implementation

\KV_sanitize_outerlevel_active_equals:N \KV_sanitize_outerlevel_active_commas:N	\KV_sanitize_outerlevel_active_equals:N <tl var.>
--	---

Replaces catcode other = and , within a <tl var.> with active characters.

\KV_remove_surrounding_spaces:nw \KV_remove_surrounding_spaces_auxi:w ★	\KV_remove_surrounding_spaces:nw <tl> <token list> \q_nil \KV_remove_surrounding_spaces_auxi:w <token list> \q_3
--	---

Removes a possible leading space plus a possible ending space from a <token list>. The first version (which is not used in the code) stores it in <tl>.

\KV_add_value_element:w \KV_set_key_element:w	\KV_set_key_element:w <token list> \q_nil \KV_add_value_element:w \q_stop <token list> \q_nil
--	--

Specialised functions to strip spaces from their input and set the token registers `\l_KV_currkey_tl` or `\l_KV_currval_tl` respectively.

<pre> \KV_split_key_value_current:w \KV_split_key_value_space_removal:w \KV_split_key_value_space_removal_detect_error:wTF \KV_split_key_value_no_space_removal:w </pre>	<pre> \KV_split_key_value_current:w ... </pre>
--	--

These functions split keyval lists into chunks depending which sanitising method is being used. `\KV_split_key_value_current:w` is `\cs_set_eq:NN` to whichever is appropriate.

## 114.1 Module code

We start by ensuring that the required packages are loaded.

```

6300 <*package>
6301 \ProvidesExplPackage
6302   {\filename}{\filedate}{\fileversion}{\filedescription}
6303 \package_check_loaded_expl:
6304 </package>
6305 <*initex | package>

```

<pre> \l_KV_tmpa_tl \l_KV_tmpp_tl \c_KV_single_equal_sign_tl </pre>	<p>Various useful things.</p> <pre> 6306 \tl_new:N \l_KV_tmpa_tl 6307 \tl_new:N \l_KV_tmpp_tl 6308 \tl_const:Nn \c_KV_single_equal_sign_tl { = } </pre>
---	---

<pre> \l_KV_parse_tl \l_KV_currkey_tl \l_KV_currval_tl </pre>	<p>Some more useful things.</p> <pre> 6309 \tl_new:N \l_KV_parse_tl 6310 \tl_new:N \l_KV_currkey_tl 6311 \tl_new:N \l_KV_currval_tl </pre>
---	--

<pre> \l_KV_level_int </pre>	<p>This is used to track how deeply nested calls to the keyval processor are, so that the correct functions are always in use.</p> <pre> 6312 \int_new:N \l_KV_level_int </pre>
------------------------------	---

<pre> remove_one_level_of_braces_bool </pre>	<p>A boolean to control</p> <pre> 6313 \bool_new:N \l_KV_remove_one_level_of_braces_bool 6314 \bool_set_true:N \l_KV_remove_one_level_of_braces_bool </pre>
--	---

`process_space_removal_sanitiz`:NNn  
`s_space_removal_no_sanitiz`:NNn  
`o_space_removal_no_sanitiz`:NNn  
`\KV_process_aux`:NNNn

The wrapper function takes care of assigning the appropriate `elt` functions before and after the parsing step. In that way there is no danger of a mistake with the wrong functions being used.

```

6315 \cs_new_protected_nopar:Npn \KV_process_space_removal_sanitiz:NNn {
6316   \KV_process_aux:NNNn \KV_parse_space_removal_sanitiz:n
6317 }
6318 \cs_new_protected_nopar:Npn \KV_process_space_removal_no_sanitiz:NNn {
6319   \KV_process_aux:NNNn \KV_parse_space_removal_no_sanitiz:n
6320 }
6321 \cs_new_protected_nopar:Npn \KV_process_no_space_removal_no_sanitiz:NNn {
6322   \KV_process_aux:NNNn \KV_parse_no_space_removal_no_sanitiz:n
6323 }
6324 \cs_new_protected:Npn \KV_process_aux:NNNn #1#2#3#4 {
6325   \cs_set_eq:cN
6326     { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
6327     \KV_key_no_value_elt:n
6328   \cs_set_eq:cN
6329     { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
6330     \KV_key_value_elt:nn
6331   \cs_set_eq:NN \KV_key_no_value_elt:n #2
6332   \cs_set_eq:NN \KV_key_value_elt:nn #3
6333   \int_incr:N \l_KV_level_int
6334   #1 {#4}
6335   \int_decr:N \l_KV_level_int
6336   \cs_set_eq:Nc \KV_key_no_value_elt:n
6337     { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
6338   \cs_set_eq:Nc \KV_key_value_elt:nn
6339     { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
6340 }

```

`size_outerlevel_active_equals`:N  
`size_outerlevel_active_commas`:N

Some functions for sanitizing top level equals and commas. Replace `=13` and `,13` with `=12` and `,12` resp.

```

6341 \group_begin:
6342 \char_set_catcode:nn{'\}{13}
6343 \char_set_catcode:nn{'\,}{13}
6344 \char_set_lccode:nn{'\8}{'\=}
6345 \char_set_lccode:nn{'\9}{'\,}
6346 \tl_to_lowercase:n{\group_end:
6347 \cs_new_protected_nopar:Npn \KV_sanitiz_outerlevel_active_equals:N #1{
6348   \tl_replace_all_in:Nnn #1 = 8
6349 }
6350 \cs_new_nopar:Npn \KV_sanitiz_outerlevel_active_commas:N #1{
6351   \tl_replace_all_in:Nnn #1 , 9
6352 }
6353 }

```

`\KV_remove_surrounding_spaces:nw`  
`remove_surrounding_spaces_auxi:w`  
`move_surrounding_spaces_auxii:w`  
`\KV_set_key_element:w`  
`\KV_add_value_element:w`

The macro `\KV_remove_surrounding_spaces:nw` removes a possible leading space plus a possible ending space from its second argument and stores it in the token register #1.

Based on Around the Bend No. 15 but with some enhancements. For instance, this definition is purely expandable.

We use a funny token  $Q_3$  as a delimiter.

```

6354 \group_begin:
6355 \char_set_catcode:nn{'\Q}{3}

6356 \cs_new:Npn\KV_remove_surrounding_spaces:nw#1#2\q_nil{

```

The idea in this processing is to use a  $Q$  with strange catcode to remove a trailing space. But first, how to get this expansion going?

If you have read the fine print in the `l3expan` module, you'll know that the `f` type expansion will expand until the first non-expandable token is seen and if this token is a space, it will be gobbled. Sounds useful for removing a leading space but we also need to make sure that it does nothing but removing that space! Therefore we prepend the argument to be trimmed with an `\exp_not:N`. Now why is that? `\exp_not:N` in itself is an expandable command so will allow the `f` expansion to continue. If the first token in the argument to be trimmed is a space, it will be gobbled and the expansion stop. If the first token isn't a space, the `\exp_not:N` turns it temporarily into `\scan_stop:` which is unexpandable. The processing stops but the token following directly after `\exp_not:N` is now back to normal.

The function here allows you to insert arbitrary functions in the first argument but they should all be with an `f` type expansion. For the application in this module, we use `\tl_set:Nf`.

Once the expansion has been kick-started, we apply `\KV_remove_surrounding_spaces_auxi:w` to the replacement text of `#2`, adding a leading `\exp_not:N`. Note that no braces are stripped off of the original argument.

```

6357 #1{\KV_remove_surrounding_spaces_auxi:w \exp_not:N#2Q~Q}
6358 }

```

`\KV_remove_surrounding_spaces_auxi:w` removes a trailing space if present, then calls `\KV_remove_surrounding_spaces_auxii:w` to clean up any leftover bizarre  $Q$ s. In order for `\KV_remove_surrounding_spaces_auxii:w` to work properly we need to put back a  $Q$  first.

```

6359 \cs_new:Npn\KV_remove_surrounding_spaces_auxi:w#1~Q{
6360   \KV_remove_surrounding_spaces_auxii:w #1 Q
6361 }

```

Now all that is left to do is remove a leading space which should be taken care of by the function used to initiate the expansion. Simply return the argument before the funny  $Q$ .

```

6362 \cs_new:Npn\KV_remove_surrounding_spaces_auxii:w#1Q#2{#1}

```

Here are some specialized versions of the above. They do exactly what we want in one go. First trim spaces from the value and then put the result surrounded in braces onto `\l_KV_parse_tl`.

```

6363 \cs_new_protected:Npn\KV_add_value_element:w\q_stop#1\q_nil{
6364   \tl_set:Nf\l_KV_currval_tl {
6365     \KV_remove_surrounding_spaces_auxi:w \exp_not:N#1Q~Q
6366   }
6367   \tl_put_right:No\l_KV_parse_tl{
6368     \exp_after:wN { \l_KV_currval_tl }
6369   }
6370 }

```

When storing the key we firstly remove spaces plus the prepended `\q_no_value`.

```

6371 \cs_new_protected:Npn\KV_set_key_element:w#1\q_nil{
6372   \tl_set:Nf\l_KV_currkey_tl
6373   {
6374     \exp_last_unbraced:NNo \KV_remove_surrounding_spaces_auxi:w
6375     \exp_not:N \use_none:n #1Q~Q
6376   }

```

Afterwards we gobble an extra level of braces if that's what we are asked to do.

```

6377   \bool_if:NT \l_KV_remove_one_level_of_braces_bool
6378   {
6379     \exp_args:NNo \tl_set:No \l_KV_currkey_tl {
6380       \exp_after:wN \KV_add_element_aux:w \l_KV_currkey_tl \q_nil
6381     }
6382   }
6383 }
6384 \group_end:

```

`\KV_add_element_aux:w` A helper function for fixing braces around keys and values.

```

6385 \cs_new:Npn \KV_add_element_aux:w#1\q_nil{#1}

```

Parse a list of keyvals, put them into list form with entries like `\KV_key_no_value_elt:n{key1}` and `\KV_key_value_elt:nn{key2}{val2}`.

`\KV_parse_sanitize_aux:n` The slow parsing algorithm sanitizes active commas and equal signs at the top level first. Then uses `#1` as inspector of each element in the comma list.

```

6386 \cs_new_protected:Npn \KV_parse_sanitize_aux:n #1 {
6387   \group_begin:
6388     \tl_clear:N \l_KV_parse_tl
6389     \tl_set:Nn \l_KV_tmpa_tl {#1}
6390     \KV_sanitize_outerlevel_active_equals:N \l_KV_tmpa_tl
6391     \KV_sanitize_outerlevel_active_commas:N \l_KV_tmpa_tl
6392     \exp_last_unbraced:NNV \KV_parse_elt:w \q_no_value
6393     \l_KV_tmpa_tl , \q_nil ,

```

We evaluate the parsed keys and values outside the group so the token register is restored to its previous value.

```

6394 \exp_after:wN \group_end:
6395 \l_KV_parse_tl
6396 }

```

`\KV_parse_no_sanitize_aux:n` Like above but we don't waste time sanitizing. This is probably the one we will use for preamble parsing where catcodes of = and , are as expected!

```

6397 \cs_new_protected:Npn \KV_parse_no_sanitize_aux:n #1{
6398 \group_begin:
6399 \tl_clear:N \l_KV_parse_tl
6400 \KV_parse_elt:w \q_no_value #1 , \q_nil ,
6401 \exp_after:wN \group_end:
6402 \l_KV_parse_tl
6403 }

```

`\KV_parse_elt:w` This function will always have a `\q_no_value` stuffed in as the rightmost token in #1. In case there was a blank entry in the comma separated list we just run it again. The `\use_none:n` makes sure to gobble the quark `\q_no_value`. A similar test is made to check if we hit the end of the recursion.

```

6404 \cs_set:Npn \KV_parse_elt:w #1,{
6405 \tl_if_blank:oTF{\use_none:n #1}
6406 { \KV_parse_elt:w \q_no_value }
6407 {
6408 \quark_if_nil:oF {\use_ii:nn #1 }

```

If we made it to here we can start parsing the key and value. When done try, try again.

```

6409 {
6410 \KV_split_key_value_current:w #1==\q_nil
6411 \KV_parse_elt:w \q_no_value
6412 }
6413 }
6414 }

```

`\KV_split_key_value_current:w` The function called to split the keys and values.

```

6415 \cs_new:Npn \KV_split_key_value_current:w {\ERROR}

```

We provide two functions for splitting keys and values. The reason being that most of the time, we should probably be in the special coding regime where spaces are ignored. Hence it makes no sense to spend time searching for extra space tokens and we can do the settings directly. When comparing these two versions (neither doing any sanitizing) the `no_space_removal` version is more than 40% faster than `space_removal`.

It is up to functions like `\DeclareTemplate` to check which catcode regime is active and then pick up the version best suited for it.

```
split_key_value_space_removal:w
space_removal_detect_error:wTF
split_key_value_space_removal_aux:w
```

The code below removes extraneous spaces around the keys and values plus one set of braces around the entire value.

Unlike the version to be used when spaces are ignored, this one only grabs the key which is everything up to the first = and save the rest for closer inspection. Reason is that if a user has entered `mykey={{myval}}`, then the outer braces have already been removed before we even look at what might come after the key. So this is slightly more tedious (but only slightly) but at least it always removes only one level of braces.

```
6416 \cs_new_protected:Npn \KV_split_key_value_space_removal:w #1 = #2\q_nil{
```

First grab the key.

```
6417 \KV_set_key_element:w#1\q_nil
```

Then we start checking. If only a key was entered, #2 contains = and nothing else, so we test for that first.

```
6418 \tl_set:Nn\l_KV_tmpa_tl{#2}
6419 \tl_if_eq:NNTF\l_KV_tmpa_tl\c_KV_single_equal_sign_tl
```

Then we just insert the default key.

```
6420 {
6421   \tl_put_right:No\l_KV_parse_tl{
6422     \exp_after:wN \KV_key_no_value_elt:n
6423     \exp_after:wN {\l_KV_currkey_tl}
6424   }
6425 }
```

Otherwise we must take a closer look at what is left. The remainder of the original list up to the comma is now stored in #2 plus an additional ==, which wasn't gobbled during the initial reading of arguments. If there is an error then we can see at least one more = so we call an auxiliary function to check for this.

```
6426 {
6427   \KV_split_key_value_space_removal_detect_error:wTF#2\q_no_value\q_nil
6428   {\KV_split_key_value_space_removal_aux:w \q_stop #2}
6429   { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
6430 }
6431 }
```

The error test.

```
6432 \cs_new_protected:Npn
6433 \KV_split_key_value_space_removal_detect_error:wTF#1=#2#3\q_nil{
6434   \tl_if_head_eq_meaning:nNTF{#3}\q_no_value
6435 }
```

Now we can start extracting the value. Recall that #1 here starts with `\q_stop` so all braces are still there! First we try to see how much is left if we gobble three brace groups

from #1. If #1 is empty or blank, all three quarks are gobbled. If #1 consists of exactly one token or brace group, only the latter quark is left.

```

6436 \cs_new:Npn \KV_val_preserve_braces:NnN #1#2#3{{#2}}
6437 \cs_new_protected:Npn \KV_split_key_value_space_removal_aux:w #1=={
6438   \tl_set:Nx\l_KV_tmpa_tl{\exp_not:o{\use_none:nnn#1\q_nil\q_nil}}
6439   \tl_put_right:No\l_KV_parse_tl{
6440     \exp_after:wN \KV_key_value_elt:nn
6441     \exp_after:wN {\l_KV_currkey_tl}
6442   }

```

If there a blank space or nothing at all, \l\_KV\_tmpa\_tl is now completely empty.

```

6443   \tl_if_empty:NTF\l_KV_tmpa_tl

```

We just put an empty value on the stack.

```

6444   { \tl_put_right:Nn\l_KV_parse_tl{{}} }
6445   {

```

If there was exactly one brace group or token in #1, \l\_KV\_tmpa\_tl is now equal to \q\_nil. Then we can just pick it up as the second argument of #1. This will also take care of any spaces which might surround it.

```

6446   \quark_if_nil:NTF\l_KV_tmpa_tl
6447   {
6448     \bool_if:NTF \l_KV_remove_one_level_of_braces_bool
6449     {
6450       \tl_put_right:No\l_KV_parse_tl{
6451         \exp_after:wN{\use_ii:nnn #1\q_nil}
6452       }
6453     }
6454     {
6455       \tl_put_right:No\l_KV_parse_tl{
6456         \exp_after:wN{\KV_val_preserve_braces:NnN #1\q_nil}
6457       }
6458     }
6459   }

```

Otherwise we grab the value.

```

6460   { \KV_add_value_element:w #1\q_nil }
6461   }
6462 }

```

it\_key\_value\_no\_space\_removal:w This version is for when in the special coding regime where spaces are ignored so there is no need to do any fancy space hacks, however fun they may be. Since there are no spaces, a set of braces around a value is automatically stripped by T<sub>E</sub>X.

```

6463 \cs_new_protected:Npn \KV_split_key_value_no_space_removal:w #1#2=#3=#4\q_nil{
6464   \tl_set:Nn\l_KV_tmpa_tl{#4}

```

```

6465 \tl_if_empty:NTF \l_KV_tmpa_tl
6466 {
6467   \tl_put_right:Nn\l_KV_parse_tl{\KV_key_no_value_elt:n{#2}}
6468 }
6469 {
6470   \tl_if_eq:NNTF\c_KV_single_equal_sign_tl\l_KV_tmpa_tl
6471   {
6472     \tl_put_right:Nn\l_KV_parse_tl{\KV_key_value_elt:nn{#2}{#3}}
6473   }
6474   { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
6475 }
6476 }

```

\KV\_key\_no\_value\_elt:n  
\KV\_key\_value\_elt:nn

```

6477 \cs_new:Npn \KV_key_no_value_elt:n #1{\ERROR}
6478 \cs_new:Npn \KV_key_value_elt:nn #1#2{\ERROR}

```

no\_space\_removal\_no\_sanitize:n Finally we can put all the things together. \KV\_parse\_no\_space\_removal\_no\_sanitize:n is the version that disallows unmatched conditional and does no space removal.

```

6479 \cs_new_protected_nopar:Npn \KV_parse_no_space_removal_no_sanitize:n {
6480   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_no_space_removal:w
6481   \KV_parse_no_sanitize_aux:n
6482 }

```

parse\_space\_removal\_sanitize:n The other varieties can be defined in a similar manner. For the version needed at the document level, we can use this one.

```

6483 \cs_new_protected_nopar:Npn \KV_parse_space_removal_sanitize:n {
6484   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
6485   \KV_parse_sanitize_aux:n
6486 }

```

For preamble use by the non-programmer this is probably best.

```

6487 \cs_new_protected_nopar:Npn \KV_parse_space_removal_no_sanitize:n {
6488   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
6489   \KV_parse_no_sanitize_aux:n
6490 }

6491 \msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
6492 {Misplaced-equals-sign-in-key--value-input~\msg_line_context:}
6493 {
6494   I-am-trying-to-read-some-key--value-input-but-found-two-equals-
6495   signs\\%
6496   without~a-comma-between-them.
6497 }

6498 </initex | package>

```

```

6499 <*showmemory>
6500 \showMemUsage
6501 </showmemory>

```

The usual preliminaries.

```

6502 <*package>
6503 \ProvidesExplPackage
6504   {\filename}{\filedate}{\fileversion}{\filedescription}
6505 \package_check_loaded_expl:
6506 </package>
6507 <*initex | package>

```

#### 114.1.1 Variables and constants

\c_keys_root_tl	Where the keys are really stored.
\c_keys_properties_root_tl	<pre> 6508 \tl_const:Nn \c_keys_root_tl { keys~&gt;~ } 6509 \tl_const:Nn \c_keys_properties_root_tl { keys_properties } </pre>
\c_keys_value_forbidden_tl	Two marker token lists.
\c_keys_value_required_tl	<pre> 6510 \tl_const:Nn \c_keys_value_forbidden_tl { forbidden } 6511 \tl_const:Nn \c_keys_value_required_tl { required } </pre>
\l_keys_choice_int	Used for the multiple choice system.
\l_keys_choice_tl	<pre> 6512 \int_new:N \l_keys_choice_int 6513 \tl_new:N \l_keys_choice_tl </pre>
\l_keys_choice_code_tl	When creating multiple choices, the code is stored here.
	<pre> 6514 \tl_new:N \l_keys_choice_code_tl </pre>
\l_keys_key_tl	Storage for the current key name and the path of the key (key name plus module name).
\l_keys_path_tl	
\l_keys_property_tl	<pre> 6515 \tl_new:N \l_keys_key_tl 6516 \tl_new:N \l_keys_path_tl 6517 \tl_new:N \l_keys_property_tl </pre>
\l_keys_module_tl	The module for an entire set of keys.
	<pre> 6518 \tl_new:N \l_keys_module_tl </pre>
\l_keys_no_value_bool	To indicate that no value has been given.
	<pre> 6519 \bool_new:N \l_keys_no_value_bool </pre>
\l_keys_value_tl	A token variable for the given value.
	<pre> 6520 \tl_new:N \l_keys_value_tl </pre>

### 114.1.2 Internal functions

`\keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand.

```
6521 \cs_new_protected_nopar:Npn \keys_bool_set:Nn #1#2 {  
6522   \keys_cmd_set:nx { \l_keys_path_tl / true } {  
6523     \exp_not:c { bool_ #2 set_true:N }  
6524     \exp_not:N #1  
6525   }  
6526   \keys_cmd_set:nx { \l_keys_path_tl / false } {  
6527     \exp_not:N \use:c  
6528     { bool_ #2 set_false:N }  
6529     \exp_not:N #1  
6530   }  
6531   \keys_choice_make:  
6532   \cs_if_exist:NF #1 {  
6533     \bool_new:N #1  
6534   }  
6535   \keys_default_set:n { true }  
6536 }
```

`\keys_choice_code_store:x` The code for making multiple choices is stored in a token list as there should not be any # tokens.

```
6537 \cs_new_protected:Npn \keys_choice_code_store:x #1 {  
6538   \tl_set:cx { \c_keys_root_tl \l_keys_path_tl .choice_code_tl } {#1}  
6539 }
```

`\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.

```
6540 \cs_new_protected_nopar:Npn \keys_choice_find:n #1 {  
6541   \keys_execute_aux:nn { \l_keys_path_tl / \tl_to_str:n {#1} } {  
6542     \keys_execute_aux:nn { \l_keys_path_tl / unknown } { }  
6543   }  
6544 }
```

`\keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key.

```
6545 \cs_new_protected_nopar:Npn \keys_choice_make: {  
6546   \keys_cmd_set:nn { \l_keys_path_tl } {  
6547     \keys_choice_find:n {##1}  
6548   }  
6549   \keys_cmd_set:nn { \l_keys_path_tl / unknown } {  
6550     \msg_kernel_error:nnxx { keys } { choice-unknown }  
6551     { \l_keys_path_tl } {##1}  
6552   }  
6553 }
```

`\keys_choices_generate:n`    Creating multiple-choices means setting up the “indicator” code, then applying whatever  
`\keys_choices_generate_aux:n`    the user wanted.

```

6554 \cs_new_protected:Npn \keys_choices_generate:n #1 {
6555   \keys_choice_make:
6556   \int_zero:N \l_keys_choice_int
6557   \cs_if_exist:cTF {
6558     \c_keys_root_tl \l_keys_path_tl .choice_code_tl
6559   } {
6560     \tl_set:Nv \l_keys_choice_code_tl {
6561       \c_keys_root_tl \l_keys_path_tl .choice_code_tl
6562     }
6563   }{
6564     \msg_kernel_error:nxx { keys } { generate-choices-before-code }
6565     { \l_keys_path_tl }
6566   }
6567   \clist_map_function:nN {#1} \keys_choices_generate_aux:n
6568 }
6569 \cs_new_protected_nopar:Npn \keys_choices_generate_aux:n #1 {
6570   \keys_cmd_set:nx { \l_keys_path_tl / #1 } {
6571     \exp_not:n { \tl_set:Nn \l_keys_choice_tl } {#1}
6572     \exp_not:n { \int_set:Nn \l_keys_choice_int }
6573     { \int_use:N \l_keys_choice_int }
6574     \exp_not:V \l_keys_choice_code_tl
6575   }
6576   \int_incr:N \l_keys_choice_int
6577 }

```

`\keys_cmd_set:nn`    Creating a new command means setting properties and then creating a function with the  
`\keys_cmd_set:nx`    correct number of arguments.

```

\keys_cmd_set_aux:n
6578 \cs_new_protected:Npn \keys_cmd_set:nn #1#2 {
6579   \keys_cmd_set_aux:n {#1}
6580   \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
6581   \cs_set:Npn 1 {#2}
6582 }
6583 \cs_new_protected:Npn \keys_cmd_set:nx #1#2 {
6584   \keys_cmd_set_aux:n {#1}
6585   \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
6586   \cs_set:Npx 1 {#2}
6587 }
6588 \cs_new_protected_nopar:Npn \keys_cmd_set_aux:n #1 {
6589   \keys_property_undefine:n { #1 .default_tl }
6590   \cs_if_free:cT { \c_keys_root_tl #1 .req_tl }
6591   { \tl_new:c { \c_keys_root_tl #1 .req_tl } }
6592   \tl_clear:c { \c_keys_root_tl #1 .req_tl }
6593 }

```

`\keys_default_set:n`    Setting a default value is easy.  
`\keys_default_set:V`

```

6594 \cs_new_protected:Npn \keys_default_set:n #1 {
6595   \cs_if_free:cT { \c_keys_root_tl \l_keys_path_tl .default_tl }
6596   { \tl_new:c { \c_keys_root_tl \l_keys_path_tl .default_tl } }
6597   \tl_set:cn { \c_keys_root_tl \l_keys_path_tl .default_tl } {#1}
6598 }
6599 \cs_generate_variant:Nn \keys_default_set:n { V }

```

`\keys_define:nn` The main key-defining function mainly sets up things for `l3keyval` to use.

```

\keys_define_aux:nnn
\keys_define_aux:onn
6600 \cs_new_protected:Npn \keys_define:nn {
6601   \keys_define_aux:onn { \l_keys_module_tl }
6602 }
6603 \cs_new_protected:Npn \keys_define_aux:nnn #1#2#3 {
6604   \tl_set:Nn \l_keys_module_tl {#2}
6605   \KV_process_no_space_removal_no_sanitization:NNn
6606   \keys_define_elt:n \keys_define_elt:nn {#3}
6607   \tl_set:Nn \l_keys_module_tl {#1}
6608 }
6609 \cs_generate_variant:Nn \keys_define_aux:nnn { o }

```

`\keys_define_elt:n` The element processors for defining keys.

```

\keys_define_elt:nn
6610 \cs_new_protected_nopar:Npn \keys_define_elt:n #1 {
6611   \bool_set_true:N \l_keys_no_value_bool
6612   \keys_define_elt_aux:nn {#1} { }
6613 }
6614 \cs_new_protected:Npn \keys_define_elt:nn #1#2 {
6615   \bool_set_false:N \l_keys_no_value_bool
6616   \keys_define_elt_aux:nn {#1} {#2}
6617 }

```

`\keys_define_elt_aux:nn` The auxiliary function does most of the work.

```

6618 \cs_new_protected:Npn \keys_define_elt_aux:nn #1#2 {
6619   \keys_property_find:n {#1}
6620   \cs_set_eq:Nc \keys_tmp:w
6621   { \c_keys_properties_root_tl \l_keys_property_tl }
6622   \cs_if_exist:NTF \keys_tmp:w {
6623     \keys_define_key:n {#2}
6624   }{
6625     \msg_kernel_error:nnxx { keys } { property-unknown }
6626     { \l_keys_property_tl } { \l_keys_path_tl }
6627   }
6628 }

```

`\keys_define_key:n` Defining a new key means finding the code for the appropriate property then running it. As properties have signatures, a check can be made for required values without needing anything set explicitly.

```

6629 \cs_new_protected:Npn \keys_define_key:n #1 {
6630   \bool_if:NTF \l_keys_no_value_bool {
6631     \int_compare:nTF {
6632       \exp_args:Nc \cs_get_arg_count_from_signature:N
6633       { \l_keys_property_tl } = \c_zero
6634     } {
6635       \keys_tmp:w
6636     }{
6637       \msg_kernel_error:nnxx { key } { property-requires-value }
6638       { \l_keys_property_tl } { \l_keys_path_tl }
6639     }
6640   }{
6641     \keys_tmp:w {#1}
6642   }
6643 }

```

`\keys_execute:` Actually executing a key is done in two parts. First, look for the key itself, then look for  
`\keys_execute_unknown:` the unknown key with the same path. If both of these fail, complain!  
`\keys_execute_aux:nn`

```

6644 \cs_new_protected_nopar:Npn \keys_execute: {
6645   \keys_execute_aux:nn { \l_keys_path_tl } {
6646     \keys_execute_unknown:
6647   }
6648 }
6649 \cs_new_protected_nopar:Npn \keys_execute_unknown: {
6650   \keys_execute_aux:nn { \l_keys_module_tl / unknown } {
6651     \msg_kernel_error:nnxx { keys } { key-unknown } { \l_keys_path_tl }
6652     { \l_keys_module_tl }
6653   }
6654 }

```

If there is only one argument required, it is wrapped in braces so that everything is passed through properly. On the other hand, if more than one is needed it is down to the user to have put things in correctly! The use of `\q_keys_stop` here means that arguments do not run away (hence the nine empty groups), but that the module can clean up the spare groups at the end of executing the key.

```

6655 \cs_new_protected_nopar:Npn \keys_execute_aux:nn #1#2 {
6656   \cs_set_eq:Nc \keys_tmp:w { \c_keys_root_tl #1 .cmd:n }
6657   \cs_if_exist:NTF \keys_tmp:w {
6658     \exp_args:NV \keys_tmp:w \l_keys_value_tl
6659   }{
6660     #2
6661   }
6662 }

```

`\keys_if_exist:nnTF` A check for the existence of a key. This works by looking for the command function for the key (which ends `.cmd:n`).

```

6663 \prg_set_conditional:Nnn \keys_if_exist:nn {TF,T,F} {
6664   \cs_if_exist:cTF { \c_keys_root_tl #1 / #2 .cmd:n } {
6665     \prg_return_true:
6666   }{
6667     \prg_return_false:
6668   }
6669 }

```

`\keys_if_value_requirement:nTF` To test if a value is required or forbidden. Only one version is needed, so done by hand.

```

6670 \cs_new_nopar:Npn \keys_if_value_requirement:nTF #1 {
6671   \tl_if_eq:ccTF { c_keys_value_ #1 _tl } {
6672     \c_keys_root_tl \l_keys_path_tl .req_tl
6673   }
6674 }

```

`\keys_meta_make:n` To create a met-key, simply set up to pass data through.

`\keys_meta_make:x`

```

6675 \cs_new_protected_nopar:Npn \keys_meta_make:n #1 {
6676   \exp_last_unbraced:NNo \keys_cmd_set:nn \l_keys_path_tl
6677   \exp_after:wN { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }
6678 }
6679 \cs_new_protected_nopar:Npn \keys_meta_make:x #1 {
6680   \keys_cmd_set:nx { \l_keys_path_tl } {
6681     \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1}
6682   }
6683 }

```

`\keys_property_find:n` Searching for a property means finding the last “.” in the input, and storing the text before and after it.

`\keys_property_find_aux:n`

`\keys_property_find_aux:w`

```

6684 \cs_new_protected_nopar:Npn \keys_property_find:n #1 {
6685   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
6686   \tl_if_in:nnTF {#1} {.} {
6687     \keys_property_find_aux:n {#1}
6688   }{
6689     \msg_kernel_error:nnx { keys } { key-no-property } {#1}
6690   }
6691 }
6692 \cs_new_protected_nopar:Npn \keys_property_find_aux:n #1 {
6693   \keys_property_find_aux:w #1 \q_stop
6694 }
6695 \cs_new_protected_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop {
6696   \tl_if_in:nnTF {#2} { . } {
6697     \tl_set:Nx \l_keys_path_tl {
6698       \l_keys_path_tl \tl_to_str:n {#1} .
6699     }
6700     \keys_property_find_aux:w #2 \q_stop
6701   }{

```

```

6702     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
6703     \tl_set:Nn \l_keys_property_tl { . #2 }
6704   }
6705 }

```

`\keys_property_new:nn` Creating a new property is simply a case of making the correctly-named function.  
`\keys_property_new_arg:nn`

```

6706 \cs_new_nopar:Npn \keys_property_new:nn #1#2 {
6707   \cs_new:cpn { \c_keys_properties_root_tl #1 } {#2}
6708 }
6709 \cs_new_protected_nopar:Npn \keys_property_new_arg:nn #1#2 {
6710   \cs_new:cpn { \c_keys_properties_root_tl #1 } ##1 {#2}
6711 }

```

`\keys_property_undefine:n` Removing a property means undefining it.

```

6712 \cs_new_protected_nopar:Npn \keys_property_undefine:n #1 {
6713   \cs_set_eq:cN { \c_keys_root_tl #1 } \c_undefined
6714 }

```

`\keys_set:nn` The main setting function just does the set up to get `l3keyval` to do the hard work.

`\keys_set:nV`

`\keys_set:nv`

`\keys_set_aux:nnn`

`\keys_set_aux:onn`

```

6715 \cs_new_protected:Npn \keys_set:nn {
6716   \keys_set_aux:onn { \l_keys_module_tl }
6717 }
6718 \cs_generate_variant:Nn \keys_set:nn { nV, nv }
6719 \cs_new_protected:Npn \keys_set_aux:nnn #1#2#3 {
6720   \tl_set:Nn \l_keys_module_tl {#2}
6721   \KV_process_space_removal_sanitization:NNn
6722     \keys_set_elt:n \keys_set_elt:nn {#3}
6723   \tl_set:Nn \l_keys_module_tl {#1}
6724 }
6725 \cs_generate_variant:Nn \keys_set_aux:nnn { o }

```

`\keys_set_elt:n` The two element processors are almost identical, and pass the data through to the underlying auxiliary, which does the work.  
`\keys_set_elt:nn`

```

6726 \cs_new_protected_nopar:Npn \keys_set_elt:n #1 {
6727   \bool_set_true:N \l_keys_no_value_bool
6728   \keys_set_elt_aux:nn {#1} { }
6729 }
6730 \cs_new_protected:Npn \keys_set_elt:nn #1#2 {
6731   \bool_set_false:N \l_keys_no_value_bool
6732   \keys_set_elt_aux:nn {#1} {#2}
6733 }

```

`\keys_set_elt_aux:nn` First, set the current path and add a default if needed. There are then checks to see if  
`\keys_set_elt_aux:` the a value is required or forbidden. If everything passes, move on to execute the code.

```

6734 \cs_new_protected:Npn \keys_set_elt_aux:nn #1#2 {
6735   \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
6736   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
6737   \keys_value_or_default:n {#2}
6738   \keys_if_value_requirement:NTF { required } {
6739     \bool_if:NTF \l_keys_no_value_bool {
6740       \msg_kernel_error:nnx { keys } { value-required }
6741       { \l_keys_path_tl }
6742     }{
6743       \keys_set_elt_aux:
6744     }
6745   }{
6746     \keys_set_elt_aux:
6747   }
6748 }
6749 \cs_new_protected_nopar:Npn \keys_set_elt_aux: {
6750   \keys_if_value_requirement:NTF { forbidden } {
6751     \bool_if:NTF \l_keys_no_value_bool {
6752       \keys_execute:
6753     }{
6754       \msg_kernel_error:nnxx { keys } { value-forbidden }
6755       { \l_keys_path_tl } { \tl_use:N \l_keys_value_tl }
6756     }
6757   }{
6758     \keys_execute:
6759   }
6760 }

```

`\keys_show:nn` Showing a key is just a question of using the correct name.

```

6761 \cs_new_nopar:Npn \keys_show:nn #1#2 {
6762   \cs_show:c { \c_keys_root_tl #1 / \tl_to_str:n {#2} .cmd:n }
6763 }

```

`\keys_tmp:w` This scratch function is used to actually execute keys.

```

6764 \cs_new:Npn \keys_tmp:w {}

```

`\keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```

6765 \cs_new_protected:Npn \keys_value_or_default:n #1 {
6766   \tl_set:Nn \l_keys_value_tl {#1}
6767   \bool_if:NT \l_keys_no_value_bool {
6768     \cs_if_exist:cT { \c_keys_root_tl \l_keys_path_tl .default_tl } {
6769       \tl_set:Nv \l_keys_value_tl {
6770         \c_keys_root_tl \l_keys_path_tl .default_tl
6771       }
6772     }
6773   }
6774 }

```

`\keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```

6775 \cs_new_protected_nopar:Npn \keys_value_requirement:n #1 {
6776   \tl_set_eq:cc { \c_keys_root_tl \l_keys_path_tl .req_tl }
6777   { c_keys_value_ #1 _tl }
6778 }

```

`\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.

```

6779 \cs_new_protected_nopar:Npn \keys_variable_set:NnNN #1#2#3#4 {
6780   \cs_if_exist:NF #1 {
6781     \use:c { #2 _new:N } #1
6782   }
6783   \keys_cmd_set:nx { \l_keys_path_tl } {
6784     \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1}
6785   }
6786 }
6787 \cs_generate_variant:Nn \keys_variable_set:NnNN { c }

```

### 114.1.3 Properties

`.bool_set:N` One function for this.

`.bool_gset:N`

```

6788 \keys_property_new_arg:nn { .bool_set:N } {
6789   \keys_bool_set:Nn #1 { }
6790 }
6791 \keys_property_new_arg:nn { .bool_gset:N } {
6792   \keys_bool_set:Nn #1 { g }
6793 }

```

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

6794 \keys_property_new:nn { .choice: } {
6795   \keys_choice_make:
6796 }

```

`.choice_code:n` Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

`.choice_code:x`

```

6797 \keys_property_new_arg:nn { .choice_code:n } {
6798   \keys_choice_code_store:x { \exp_not:n {#1} }
6799 }
6800 \keys_property_new_arg:nn { .choice_code:x } {
6801   \keys_choice_code_store:x {#1}
6802 }

```

`.code:n` Creating code is simply a case of passing through to the underlying `set` function.

`.code:x`

```

6803 \keys_property_new_arg:nn { .code:n } {
6804   \keys_cmd_set:nn { \l_keys_path_tl } {#1}
6805 }
6806 \keys_property_new_arg:nn { .code:x } {
6807   \keys_cmd_set:nx { \l_keys_path_tl } {#1}
6808 }

```

`.default:n` Expansion is left to the internal functions.

```

.default:V
6809 \keys_property_new_arg:nn { .default:n } {
6810   \keys_default_set:n {#1}
6811 }
6812 \keys_property_new_arg:nn { .default:V } {
6813   \keys_default_set:V #1
6814 }

```

`.dim_set:N` Setting a variable is very easy: just pass the data along.

```

.dim_set:c
6815 \keys_property_new_arg:nn { .dim_set:N } {
6816   \keys_variable_set:NnNN #1 { dim } { } n
6817 }
6818 \keys_property_new_arg:nn { .dim_set:c } {
6819   \keys_variable_set:cnNN {#1} { dim } { } n
6820 }
6821 \keys_property_new_arg:nn { .dim_gset:N } {
6822   \keys_variable_set:NnNN #1 { dim } g n
6823 }
6824 \keys_property_new_arg:nn { .dim_gset:c } {
6825   \keys_variable_set:cnNN {#1} { dim } g n
6826 }

```

`.fp_set:N` Setting a variable is very easy: just pass the data along.

```

.fp_set:c
6827 \keys_property_new_arg:nn { .fp_set:N } {
6828   \keys_variable_set:NnNN #1 { fp } { } n
6829 }
6830 \keys_property_new_arg:nn { .fp_set:c } {
6831   \keys_variable_set:cnNN {#1} { fp } { } n
6832 }
6833 \keys_property_new_arg:nn { .fp_gset:N } {
6834   \keys_variable_set:NnNN #1 { fp } g n
6835 }
6836 \keys_property_new_arg:nn { .fp_gset:c } {
6837   \keys_variable_set:cnNN {#1} { fp } g n
6838 }

```

`.generate_choices:n` Making choices is easy.

```

6839 \keys_property_new_arg:nn { .generate_choices:n } {
6840   \keys_choices_generate:n {#1}
6841 }

```

.int\_set:N Setting a variable is very easy: just pass the data along.

```
.int_set:c
.int_gset:N
.int_gset:c
6842 \keys_property_new_arg:nn { .int_set:N } {
6843   \keys_variable_set:NnNN #1 { int } { } n
6844 }
6845 \keys_property_new_arg:nn { .int_set:c } {
6846   \keys_variable_set:cnNN {#1} { int } { } n
6847 }
6848 \keys_property_new_arg:nn { .int_gset:N } {
6849   \keys_variable_set:NnNN #1 { int } g n
6850 }
6851 \keys_property_new_arg:nn { .int_gset:c } {
6852   \keys_variable_set:cnNN {#1} { int } g n
6853 }
```

.meta:n Making a meta is handled internally.

```
.meta:x
6854 \keys_property_new_arg:nn { .meta:n } {
6855   \keys_meta_make:n {#1}
6856 }
6857 \keys_property_new_arg:nn { .meta:x } {
6858   \keys_meta_make:x {#1}
6859 }
```

.skip\_set:N Setting a variable is very easy: just pass the data along.

```
.skip_set:c
.skip_gset:N
.skip_gset:c
6860 \keys_property_new_arg:nn { .skip_set:N } {
6861   \keys_variable_set:NnNN #1 { skip } { } n
6862 }
6863 \keys_property_new_arg:nn { .skip_set:c } {
6864   \keys_variable_set:cnNN {#1} { skip } { } n
6865 }
6866 \keys_property_new_arg:nn { .skip_gset:N } {
6867   \keys_variable_set:NnNN #1 { skip } g n
6868 }
6869 \keys_property_new_arg:nn { .skip_gset:c } {
6870   \keys_variable_set:cnNN {#1} { skip } g n
6871 }
```

.tl\_set:N Setting a variable is very easy: just pass the data along.

```
.tl_set:c
.tl_set_x:N
.tl_set_x:c
.tl_gset:N
.tl_gset:c
.tl_gset_x:N
.tl_gset_x:c
6872 \keys_property_new_arg:nn { .tl_set:N } {
6873   \keys_variable_set:NnNN #1 { tl } { } n
6874 }
6875 \keys_property_new_arg:nn { .tl_set:c } {
6876   \keys_variable_set:cnNN {#1} { tl } { } n
6877 }
6878 \keys_property_new_arg:nn { .tl_set_x:N } {
6879   \keys_variable_set:NnNN #1 { tl } { } x
6880 }
```

```

6881 \keys_property_new_arg:nn { .tl_set_x:c } {
6882   \keys_variable_set:cnNN {#1} { tl } { } x
6883 }
6884 \keys_property_new_arg:nn { .tl_gset:N } {
6885   \keys_variable_set:NnNN #1 { tl } g n
6886 }
6887 \keys_property_new_arg:nn { .tl_gset:c } {
6888   \keys_variable_set:cnNN {#1} { tl } g n
6889 }
6890 \keys_property_new_arg:nn { .tl_gset_x:N } {
6891   \keys_variable_set:NnNN #1 { tl } g x
6892 }
6893 \keys_property_new_arg:nn { .tl_gset_x:c } {
6894   \keys_variable_set:cnNN {#1} { tl } g x
6895 }

```

.value\_forbidden: These are very similar, so both call the same function.  
.value\_required:

```

6896 \keys_property_new:nn { .value_forbidden: } {
6897   \keys_value_requirement:n { forbidden }
6898 }
6899 \keys_property_new:nn { .value_required: } {
6900   \keys_value_requirement:n { required }
6901 }

```

#### 114.1.4 Messages

For when there is a need to complain.

```

6902 \msg_kernel_new:nnnn { keys } { choice-unknown }
6903 { Choice~'#2'~unknown~for~key~'#1'. }
6904 {
6905   The~key~'#1'~takes~a~limited~number~of~values.\\
6906   The~input~given,~'#2',~is~not~on~the~list~accepted.
6907 }
6908 \msg_kernel_new:nnnn { keys } { generate-choices-before-code }
6909 { No~code~available~to~generate~choices~for~key~'#1'. }
6910 {
6911   \l_msg_coding_error_text_tl
6912   Before~using~.generate_choices:n~the~code~should~be~defined\\%
6913   with~.choice_code:n~or~.choice_code:x.
6914 }
6915 \msg_kernel_new:nnnn { keys } { key-no-property }
6916 { No~property~given~in~definition~of~key~'#1'. }
6917 {
6918   \c_msg_coding_error_text_tl
6919   Inside~\token_to_str:N \keys_define:nn \c_space_tl each~key~name
6920   needs~a~property: \\
6921   ~ ~ #1 .<property> \\

```

```

6922 LaTeX~did~not~find~a~'. '~to~indicate~the~start~of~a~property.
6923 }
6924 \msg_kernel_new:nnnn { keys } { key-unknown }
6925 { The~key~'#1'~is~unknown~and~is~being~ignored. }
6926 {
6927   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
6928   Check~that~you~have~spelled~the~key~name~correctly.
6929 }
6930 \msg_kernel_new:nnnn { keys } { property-requires-value }
6931 { The~property~'#1'~requires~a~value. }
6932 {
6933   \l_msg_coding_error_text_tl
6934   LaTeX~was~asked~to~set~property~'#2'~for~key~'#1'.\\
6935   No~value~was~given~for~the~property,~and~one~is~required.
6936 }
6937 \msg_kernel_new:nnnn { keys } { property-unknown }
6938 { The~key~property~'#1'~is~unknown. }
6939 {
6940   \l_msg_coding_error_text_tl
6941   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':\\
6942   this~property~is~not~defined.
6943 }
6944 \msg_kernel_new:nnnn { keys } { value-forbidden }
6945 { The~key~'#1'~does~not~taken~a~value. }
6946 {
6947   The~key~'#1'~should~be~given~without~a~value.\\
6948   LaTeX~will~ignore~the~given~value~'#2'.
6949 }
6950 \msg_kernel_new:nnnn { keys } { value-required }
6951 { The~key~'#1'~requires~a~value. }
6952 {
6953   The~key~'#1'~must~have~a~value.\\
6954   No~value~was~present:~the~key~will~be~ignored.
6955 }
6956 </initex | package>

```

## 115 l3file implementation

The usual lead-off.

```

6957 <*package>
6958 \ProvidesExplPackage
6959   {\filename}{\filedate}{\fileversion}{\filedescription}
6960 \package_check_loaded_expl:
6961 </package>
6962 <*initex | package>

```

`\g_file_current_name_tl` The name of the current file should be available at all times.

`\g_file_stack_seq`

```
6963 \tl_new:N \g_file_current_name_tl
6964 \seq_new:N \g_file_stack_seq
```

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 LaTeX2e.

```
6965 </initex | package>
6966 <*initex>
6967 \toks_put_right:Nn \tex_everyjob:D {
6968   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
6969 }
6970 </initex>
6971 <*package>
6972 \tl_gset_eq:NN \g_file_current_name_tl \@currname
6973 </package>
6974 <*initex | package>
```

`\g_file_record_seq` The total list of files used is recorded separately from the stack.

```
6975 \seq_new:N \g_file_record_seq
```

The current file name should be included in the file list!

```
6976 </initex | package>
6977 <*initex>
6978 \toks_put_right:Nn \tex_everyjob:D {
6979   \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
6980 }
6981 </initex>
6982 <*initex | package>
```

`\l_file_search_path_seq` The current search path.

```
6983 \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.

```
6984 </initex | package>
6985 <*package>
6986 \seq_new:N \l_file_search_path_saved_seq
6987 </package>
6988 <*initex | package>
```

`\l_file_name_tl`

`\g_file_test_stream`

`\file_if_exist:nTF`

`\file_if_exist:VTF`

`\file_if_exist_aux:n`

Checking if a file exists takes place in two parts. First, look on the TeX path, then look on the LaTeX path. The token list `\l_file_name_tl` is used as a marker for finding the file, and is also needed by `\file_input:n`.

```
6989 \tl_new:N \l_file_name_tl
```

```

6990 \prg_new_protected_conditional:Nnn \file_if_exist:n { T , F , TF } {
6991   \ior_open:Nn \g_file_test_stream {#1}
6992   \ior_if_eof:NTF \g_file_test_stream
6993   { \file_if_exist_path_aux:n {#1} }
6994   {
6995     \ior_close:N \g_file_test_stream
6996     \tl_set:Nn \l_file_name_tl {#1}
6997     \prg_return_true:
6998   }
6999 }
7000 \cs_new_protected_nopar:Npn \file_if_exist_path_aux:n #1 {
7001   \tl_clear:N \l_file_name_tl
7002   </initex | package>
7003   <*package>
7004   \cs_if_exist:NT \input@path
7005   {
7006     \seq_set_eq:NN \l_file_search_path_saved_seq
7007     \l_file_search_path_seq
7008     \clist_map_inline:Nn \input@path
7009     { \seq_put_right:Nn \l_file_search_path_seq {##1} }
7010   }
7011   </package>
7012   <*initex | package>
7013   \seq_map_inline:Nn \l_file_search_path_seq
7014   {
7015     \ior_open:Nn \g_file_test_stream { ##1 #1 }
7016     \ior_if_eof:NF \g_file_test_stream
7017     {
7018       \tl_set:Nn \l_file_name_tl { ##1 #1 }
7019       \seq_map_break:
7020     }
7021   }
7022   </initex | package>
7023   <*package>
7024   \cs_if_exist:NT \input@path
7025   {
7026     \seq_set_eq:NN \l_file_search_path_seq
7027     \l_file_search_path_saved_seq
7028   }
7029   </package>
7030   <*initex | package>
7031   \ior_close:N \g_file_test_stream
7032   \tl_if_empty:NTF \l_file_name_tl
7033   { \prg_return_false: }
7034   { \prg_return_true: }
7035 }
7036 \cs_generate_variant:Nn \file_if_exist:nT { V }
7037 \cs_generate_variant:Nn \file_if_exist:nF { V }
7038 \cs_generate_variant:Nn \file_if_exist:nTF { V }

```

`\file_input:n` Most of the work is done by the file test above.

```

\file_input:V
7039 \cs_new_protected_nopar:Npn \file_input:n #1 {
7040   \file_if_exist:nT {#1}
7041   {
7042     \initex | package>
7043     \*package>
7044     \@addtofilelist {#1}
7045     \</package>
7046     \*initex | package>
7047     \seq_gpush:NV \g_file_stack_seq \g_file_current_name_tl
7048     \tl_gset:Nn \g_file_current_name_tl {#1}
7049     \tex_expandafter:D \tex_input:D \l_file_name_tl ~
7050     \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
7051   }
7052 }
7053 \cs_generate_variant:Nn \file_input:n { V }

```

`\file_path_include:n` Wrapper functions to manage the search path.

```

\file_path_remove:n
7054 \cs_new_protected_nopar:Npn \file_path_include:n #1 {
7055   \seq_put_right:Nn \l_file_search_path_seq {#1}
7056   \seq_remove_duplicates:N \l_file_search_path_seq
7057 }
7058 \cs_new_protected_nopar:Npn \file_path_remove:n #1 {
7059   \seq_remove_element:Nn \l_file_search_path_seq {#1}
7060 }

```

`\file_list:` A function to list all files used to the log.

```

\file_list_aux:n
7061 \cs_new_protected_nopar:Npn \file_list: {
7062   \seq_remove_duplicates:N \g_file_record_seq
7063   \iow_log:n { ~File~List~* }
7064   \seq_map_function:NN \g_file_record_seq \file_list_aux:n
7065   \iow_log:n { ***** }
7066 }
7067 \cs_new_protected_nopar:Npn \file_list_aux:n #1 { \iow_log:n {#1} }

```

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```

7068 \</initex | package>
7069 \*package>
7070 \AtBeginDocument{
7071   \clist_map_inline:Nn \@filelist
7072   { \seq_put_right:Nn \g_file_record_seq {#1} }
7073 }
7074 \</package>

```

## 116 Implementation

We start by ensuring that the required packages are loaded.

```

7075 \*package>
7076 \ProvidesExplPackage
7077   {\filename}{\filedate}{\fileversion}{\filedescription}
7078 \package_check_loaded_expl:
7079 \</package>
7080 \*initex | package>

```

### 116.1 Constants

`\c_forty_four`

There is some speed to gain by moving numbers into fixed positions.

`\c_one_hundred`

`\c_one_thousand`

`\c_one_million`

`\c_one_hundred_million`

`\c_five_hundred_million`

`\c_one_thousand_million`

```

7081 \int_const:Nn \c_forty_four { 44 }
7082 \int_const:Nn \c_one_hundred { 100 }
7083 \int_const:Nn \c_one_thousand { 1000 }
7084 \int_const:Nn \c_one_million { 1 000 000 }
7085 \int_const:Nn \c_one_hundred_million { 100 000 000 }
7086 \int_const:Nn \c_five_hundred_million { 500 000 000 }
7087 \int_const:Nn \c_one_thousand_million { 1 000 000 000 }

```

`\c_fp_pi_by_four_decimal_int`

`\c_fp_pi_by_four_extended_int`

`\c_fp_pi_decimal_int`

`\c_fp_pi_extended_int`

`\c_fp_two_pi_decimal_int`

`\c_fp_two_pi_extended_int`

Parts of  $\pi$  for trigonometric range reduction, implemented as `int` variables for speed.

```

7088 \int_new:N \c_fp_pi_by_four_decimal_int
7089 \int_set:Nn \c_fp_pi_by_four_decimal_int { 785 398 158 }
7090 \int_new:N \c_fp_pi_by_four_extended_int
7091 \int_set:Nn \c_fp_pi_by_four_extended_int { 897 448 310 }
7092 \int_new:N \c_fp_pi_decimal_int
7093 \int_set:Nn \c_fp_pi_decimal_int { 141 592 653 }
7094 \int_new:N \c_fp_pi_extended_int
7095 \int_set:Nn \c_fp_pi_extended_int { 589 793 238 }
7096 \int_new:N \c_fp_two_pi_decimal_int
7097 \int_set:Nn \c_fp_two_pi_decimal_int { 283 185 307 }
7098 \int_new:N \c_fp_two_pi_extended_int
7099 \int_set:Nn \c_fp_two_pi_extended_int { 179 586 477 }

```

`\c_e_fp`

The value  $e$  as a ‘machine number’.

```

7100 \tl_new:N \c_e_fp
7101 \tl_set:Nn \c_e_fp { + 2.718281828 e 0 }

```

`\c_one_fp`

The constant value 1: used for fast comparisons.

```

7102 \tl_new:N \c_one_fp
7103 \tl_set:Nn \c_one_fp { + 1.000000000 e 0 }

```

`\c_pi_fp` The value  $\pi$  as a ‘machine number’.

```
7104 \tl_new:N \c_pi_fp
7105 \tl_set:Nn \c_pi_fp { + 3.141592654 e 0 }
```

`\c_undefined_fp` A marker for undefined values.

```
7106 \tl_new:N \c_undefined_fp
7107 \tl_set:Nn \c_undefined_fp { X 0.000000000 e 0 }
```

`\c_zero_fp` The constant zero value.

```
7108 \tl_new:N \c_zero_fp
7109 \tl_set:Nn \c_zero_fp { + 0.000000000 e 0 }
```

## 116.2 Variables

`\l_fp_arg_tl` A token list to store the formalised representation of the input for transcendental functions.

```
7110 \tl_new:N \l_fp_arg_tl
```

`\l_fp_count_int` A counter for things like the number of divisions possible.

```
7111 \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.

```
7112 \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`

```
7113 \int_new:N \l_fp_exp_integer_int
7114 \int_new:N \l_fp_exp_decimal_int
7115 \int_new:N \l_fp_exp_extended_int
7116 \int_new:N \l_fp_exp_exponent_int
```

`\l_fp_input_a_sign_int`

`\l_fp_input_a_integer_int`

`\l_fp_input_a_decimal_int`

`\l_fp_input_a_exponent_int`

`\l_fp_input_b_sign_int`

`\l_fp_input_b_integer_int`

`\l_fp_input_b_decimal_int`

`\l_fp_input_b_exponent_int`

Storage for the input: two storage areas as there are at most two inputs.

```
7117 \int_new:N \l_fp_input_a_sign_int
7118 \int_new:N \l_fp_input_a_integer_int
7119 \int_new:N \l_fp_input_a_decimal_int
7120 \int_new:N \l_fp_input_a_exponent_int
7121 \int_new:N \l_fp_input_b_sign_int
7122 \int_new:N \l_fp_input_b_integer_int
7123 \int_new:N \l_fp_input_b_decimal_int
7124 \int_new:N \l_fp_input_b_exponent_int
```

<code>\l_fp_input_a_extended_int</code>	For internal use, ‘extended’ floating point numbers are needed.
<code>\l_fp_input_b_extended_int</code>	<pre> 7125 \int_new:N \l_fp_input_a_extended_int 7126 \int_new:N \l_fp_input_b_extended_int </pre>
<pre> \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int </pre>	<p>Multiplication requires that the decimal part is split into parts so that there are no overflows.</p> <pre> 7127 \int_new:N \l_fp_mul_a_i_int 7128 \int_new:N \l_fp_mul_a_ii_int 7129 \int_new:N \l_fp_mul_a_iii_int 7130 \int_new:N \l_fp_mul_a_iv_int 7131 \int_new:N \l_fp_mul_a_v_int 7132 \int_new:N \l_fp_mul_a_vi_int 7133 \int_new:N \l_fp_mul_b_i_int 7134 \int_new:N \l_fp_mul_b_ii_int 7135 \int_new:N \l_fp_mul_b_iii_int 7136 \int_new:N \l_fp_mul_b_iv_int 7137 \int_new:N \l_fp_mul_b_v_int 7138 \int_new:N \l_fp_mul_b_vi_int </pre>
<pre> \l_fp_mul_output_int \l_fp_mul_output_tl </pre>	<p>Space for multiplication results.</p> <pre> 7139 \int_new:N \l_fp_mul_output_int 7140 \tl_new:N \l_fp_mul_output_tl </pre>
<pre> \l_fp_output_sign_int \l_fp_output_integer_int \l_fp_output_decimal_int \l_fp_output_exponent_int </pre>	<p>Output is stored in the same way as input.</p> <pre> 7141 \int_new:N \l_fp_output_sign_int 7142 \int_new:N \l_fp_output_integer_int 7143 \int_new:N \l_fp_output_decimal_int 7144 \int_new:N \l_fp_output_exponent_int </pre>
<code>\l_fp_output_extended_int</code>	<p>Again, for calculations an extended part.</p> <pre> 7145 \int_new:N \l_fp_output_extended_int </pre>
<code>\l_fp_round_carry_bool</code>	<p>To indicate that a digit needs to be carried forward.</p> <pre> 7146 \bool_new:N \l_fp_round_carry_bool </pre>
<code>\l_fp_round_decimal_tl</code>	<p>A temporary store when rounding, to build up the decimal part without needing to do any maths.</p> <pre> 7147 \tl_new:N \l_fp_round_decimal_tl </pre>
<pre> \l_fp_round_position_int \l_fp_round_target_int </pre>	<p>Used to check the position for rounding.</p> <pre> 7148 \int_new:N \l_fp_round_position_int 7149 \int_new:N \l_fp_round_target_int </pre>

`\l_fp_sign_tl` There are places where the sign needs to be set up ‘early’, so that the registers can be re-used.

```
7150 \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.

```
7151 \int_new:N \l_fp_split_sign_int
```

`\l_fp_tmp_int` A scratch int: used only where the value is not carried forward.

```
7152 \int_new:N \l_fp_tmp_int
```

`\l_fp_tmp_tl` A scratch token list variable for expanding material.

```
7153 \tl_new:N \l_fp_tmp_tl
```

`\l_fp_trig_octant_int` To track which octant the trigonometric input is in.

```
7154 \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`

```
7155 \int_new:N \l_fp_trig_sign_int
```

```
7156 \int_new:N \l_fp_trig_decimal_int
```

```
7157 \int_new:N \l_fp_trig_extended_int
```

### 116.3 Parsing numbers

`\fp_read:N` Reading a stored value is made easier as the format is designed to match the delimited  
`\fp_read_aux:w` function. This is always used to read the first value (register a).

```
7158 \cs_new_protected_nopar:Npn \fp_read:N #1 {
7159   \tex_expandafter:D \fp_read_aux:w #1 \q_stop
7160 }
7161 \cs_new_protected_nopar:Npn \fp_read_aux:w #1#2 . #3 e #4 \q_stop {
7162   \tex_if:D #1 -
7163     \l_fp_input_a_sign_int \c_minus_one
7164   \tex_else:D
7165     \l_fp_input_a_sign_int \c_one
7166   \tex_fi:D
7167   \l_fp_input_a_integer_int #2 \scan_stop:
7168   \l_fp_input_a_decimal_int #3 \scan_stop:
7169   \l_fp_input_a_exponent_int #4 \scan_stop:
7170 }
```

$\backslash$ fp_split:Nn $\backslash$ fp_split_sign: $\backslash$ fp_split_exponent: $\backslash$ fp_split_aux_i:w $\backslash$ fp_split_aux_ii:w $\backslash$ fp_split_aux_iii:w $\backslash$ fp_split_decimal:w $\backslash$ fp_split_decimal_aux:w	<p>The aim here is to use as much of T<sub>E</sub>X's mechanism as possible to pick up the numerical input without any mistakes. In particular, negative numbers have to be filtered out first in case the integer part is 0 (in which case T<sub>E</sub>X would drop the - sign). That process has to be done in a loop for cases where the sign is repeated. Finding an exponent is relatively easy, after which the next phase is to find the integer part, which will terminate with a ., and trigger the decimal-finding code. The later will allow the decimal to be too long, truncating the result.</p> <pre> 7171 \cs_new_protected_nopar:Npn \fp_split:Nn #1#2 { 7172   \tl_set:Nx \l_fp_tmp_tl {#2} 7173   \tl_set_rescan:Nno \l_fp_tmp_tl { \char_make_ignore:n { 32 } } 7174   { \l_fp_tmp_tl } 7175   \l_fp_split_sign_int \c_one 7176   \fp_split_sign: 7177   \use:c { l_fp_input_ #1 _sign_int } \l_fp_split_sign_int 7178   \tex_expandafter:D \fp_split_exponent:w \l_fp_tmp_tl e e \q_stop #1 7179 } 7180 \cs_new_protected_nopar:Npn \fp_split_sign: { 7181   \tex_ifnum:D \pdf_strcmp:D 7182   { \tex_expandafter:D \tl_head:w \l_fp_tmp_tl ? \q_stop } { - } 7183   = \c_zero 7184   \tl_set:Nx \l_fp_tmp_tl 7185   { 7186     \tex_expandafter:D 7187     \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop 7188   } 7189   \l_fp_split_sign_int -\l_fp_split_sign_int 7190   \tex_expandafter:D \fp_split_sign: 7191 \tex_else:D 7192   \tex_ifnum:D \pdf_strcmp:D 7193   { \tex_expandafter:D \tl_head:w \l_fp_tmp_tl ? \q_stop } { + } 7194   = \c_zero 7195   \tl_set:Nx \l_fp_tmp_tl 7196   { 7197     \tex_expandafter:D 7198     \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop 7199   } 7200   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D 7201   \fp_split_sign: 7202   \tex_fi:D 7203 \tex_fi:D 7204 } 7205 \cs_new_protected_nopar:Npn 7206   \fp_split_exponent:w #1 e #2 e #3 \q_stop #4 { 7207   \use:c { l_fp_input_ #4 _exponent_int } 7208   \etex_numexpr:D 0 #2 \scan_stop: 7209   \tex_afterassignment:D \fp_split_aux_i:w 7210   \use:c { l_fp_input_ #4 _integer_int } 7211   \etex_numexpr:D 0 #1 . . \q_stop #4 7212 } </pre>
--	---

```

7213 \cs_new_protected_nopar:Npn \fp_split_aux_i:w #1 . #2 . #3 \q_stop {
7214   \fp_split_aux_ii:w #2 000000000 \q_stop
7215 }
7216 \cs_new_protected_nopar:Npn \fp_split_aux_ii:w #1#2#3#4#5#6#7#8#9 {
7217   \fp_split_aux_iii:w {#1#2#3#4#5#6#7#8#9}
7218 }
7219 \cs_new_protected_nopar:Npn \fp_split_aux_iii:w #1#2 \q_stop {
7220   \l_fp_tmp_int 1 #1 \scan_stop:
7221   \tex_expandafter:D \fp_split_decimal:w
7222   \int_use:N \l_fp_tmp_int 000000000 \q_stop
7223 }
7224 \cs_new_protected_nopar:Npn \fp_split_decimal:w #1#2#3#4#5#6#7#8#9 {
7225   \fp_split_decimal_aux:w {#2#3#4#5#6#7#8#9}
7226 }
7227 \cs_new_protected_nopar:Npn \fp_split_decimal_aux:w #1#2#3 \q_stop #4 {
7228   \use:c { l_fp_input_ #4 _decimal_int } #1#2 \scan_stop:
7229   \tex_ifnum:D
7230     \etex_numexpr:D
7231       \use:c { l_fp_input_ #4 _integer_int } +
7232       \use:c { l_fp_input_ #4 _decimal_int }
7233     \scan_stop:
7234     = \c_zero
7235     \use:c { l_fp_input_ #4 _sign_int } \c_one
7236   \tex_fi:D
7237   \tex_ifnum:D
7238     \use:c { l_fp_input_ #4 _integer_int } < \c_one_thousand_million
7239   \tex_else:D
7240     \tex_expandafter:D \fp_overflow_msg:
7241   \tex_fi:D
7242 }

```

`\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.

```

7243 \cs_new_protected_nopar:Npn \fp_standardise:NNNN #1#2#3#4 {
7244   \tex_ifnum:D
7245     \etex_numexpr:D #2 + #3 = \c_zero
7246     #1 \c_one
7247     #4 \c_zero
7248     \tex_expandafter:D \use_none:nnnn
7249   \tex_else:D
7250     \tex_expandafter:D \fp_standardise_aux:NNNN
7251   \tex_fi:D
7252   #1#2#3#4
7253 }
7254 \cs_new_protected_nopar:Npn \fp_standardise_aux:NNNN #1#2#3#4 {
7255   \cs_set_protected_nopar:Npn \fp_standardise_aux:
7256   {
7257     \tex_ifnum:D #2 = \c_zero
7258     \tex_advance:D #3 \c_one_thousand_million

```

```

7259         \tex_expandafter:D \fp_standardise_aux:w
7260         \int_use:N #3 \q_stop
7261         \tex_expandafter:D \fp_standardise_aux:
7262         \tex_fi:D
7263     }
7264 \cs_set_protected_nopar:Npn
7265   \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9 \q_stop
7266   {
7267     #2 ##2 \scan_stop:
7268     #3 ##3##4##5##6##7##8##9 0 \scan_stop:
7269     \tex_advance:D #4 \c_minus_one
7270   }
7271 \fp_standardise_aux:
7272 \cs_set_protected_nopar:Npn \fp_standardise_aux:
7273   {
7274     \tex_ifnum:D #2 > \c_nine
7275     \tex_advance:D #2 \c_one_thousand_million
7276     \tex_expandafter:D \use_i:nn \tex_expandafter:D
7277     \fp_standardise_aux:w \int_use:N #2
7278     \tex_expandafter:D \fp_standardise_aux:
7279     \tex_fi:D
7280   }
7281 \cs_set_protected_nopar:Npn
7282   \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9
7283   {
7284     #2 ##1##2##3##4##5##6##7##8 \scan_stop:
7285     \tex_advance:D #3 \c_one_thousand_million
7286     \tex_divide:D #3 \c_ten
7287     \tl_set:Nx \l_fp_tmp_tl
7288     {
7289       ##9
7290       \tex_expandafter:D \use_none:n \int_use:N #3
7291     }
7292     #3 \l_fp_tmp_tl \scan_stop:
7293     \tex_advance:D #4 \c_one
7294   }
7295 \fp_standardise_aux:
7296 \tex_ifnum:D #4 < \c_one_hundred
7297 \tex_ifnum:D #4 > -\c_one_hundred
7298 \tex_else:D
7299   #1 \c_one
7300   #2 \c_zero
7301   #3 \c_zero
7302   #4 \c_zero
7303 \tex_fi:D
7304 \tex_else:D
7305   \tex_expandafter:D \fp_overflow_msg:
7306   \tex_fi:D
7307 }
7308 \cs_new_protected_nopar:Npn \fp_standardise_aux: { }

```

```
7309 \cs_new_protected_nopar:Npn \fp_standardise_aux:w { }
```

## 116.4 Internal utilities

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:
\fp_level_input_exponents_a:NNNNNNNNN
\fp_level_input_exponents_b:NNNNNNNNN
\fp_level_input_exponents_b:NNNNNNNNN

7310 \cs_new_protected_nopar:Npn \fp_level_input_exponents: {
7311   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7312     \tex_expandafter:D \fp_level_input_exponents_a:
7313   \tex_else:D
7314     \tex_expandafter:D \fp_level_input_exponents_b:
7315   \tex_fi:D
7316 }
7317 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a: {
7318   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7319     \tex_advance:D \l_fp_input_b_integer_int \c_one_thousand_million
7320   \tex_expandafter:D \use_i:nn \tex_expandafter:D
7321     \fp_level_input_exponents_a:NNNNNNNNN
7322     \int_use:N \l_fp_input_b_integer_int
7323   \tex_expandafter:D \fp_level_input_exponents_a:
7324   \tex_fi:D
7325 }
7326 \cs_new_protected_nopar:Npn
7327   \fp_level_input_exponents_a:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7328   \l_fp_input_b_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
7329   \tex_advance:D \l_fp_input_b_decimal_int \c_one_thousand_million
7330   \tex_divide:D \l_fp_input_b_decimal_int \c_ten
7331   \tl_set:Nx \l_fp_tmp_tl
7332     {
7333       #9
7334       \tex_expandafter:D \use_none:n
7335       \int_use:N \l_fp_input_b_decimal_int
7336     }
7337   \l_fp_input_b_decimal_int \l_fp_tmp_tl \scan_stop:
7338   \tex_advance:D \l_fp_input_b_exponent_int \c_one
7339 }
7340 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b: {
7341   \tex_ifnum:D \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
7342     \tex_advance:D \l_fp_input_a_integer_int \c_one_thousand_million
7343   \tex_expandafter:D \use_i:nn \tex_expandafter:D
7344     \fp_level_input_exponents_b:NNNNNNNNN
7345     \int_use:N \l_fp_input_a_integer_int
7346   \tex_expandafter:D \fp_level_input_exponents_b:
7347   \tex_fi:D
7348 }
7349 \cs_new_protected_nopar:Npn
7350   \fp_level_input_exponents_b:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7351   \l_fp_input_a_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
```

```

7352 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7353 \tex_divide:D \l_fp_input_a_decimal_int \c_ten
7354 \tl_set:Nx \l_fp_tmp_tl
7355 {
7356   #9
7357   \tex_expandafter:D \use_none:n
7358   \int_use:N \l_fp_input_a_decimal_int
7359 }
7360 \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
7361 \tex_advance:D \l_fp_input_a_exponent_int \c_one
7362 }

```

`\fp_tmp:w` Used for output of results, cutting down on `\tex_expandafter:D`. This is just a placeholder definition.

```

7363 \cs_new_protected_nopar:Npn \fp_tmp:w #1#2 { }

```

## 116.5 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
7364 \cs_new_protected_nopar:Npn \fp_new:N #1 {
7365   \tl_new:N #1
7366   \tl_gset_eq:NN #1 \c_zero_fp
7367 }
7368 \cs_generate_variant:Nn \fp_new:N { c }

```

`\fp_const:Nn` A simple wrapper.

```

\fp_const:cn
7369 \cs_new_protected_nopar:Npn \fp_const:Nn #1#2 {
7370   \cs_if_free:NTF #1
7371   {
7372     \fp_new:N #1
7373     \fp_gset:Nn #1 {#2}
7374   }
7375   {
7376     \msg_kernel_error:nx { variable-already-defined }
7377     { \token_to_str:N #1 }
7378   }
7379 }
7380 \cs_generate_variant:Nn \fp_const:Nn { c }

```

`\fp_zero:N` Zeroing fixed-points is pretty obvious.  
`\fp_zero:c`  
`\fp_gzero:N`  
`\fp_gzero:c`

```

7381 \cs_new_protected_nopar:Npn \fp_zero:N #1 {
7382   \tl_set_eq:NN #1 \c_zero_fp
7383 }
7384 \cs_new_protected_nopar:Npn \fp_gzero:N #1 {
7385   \tl_gset_eq:NN #1 \c_zero_fp
7386 }
7387 \cs_generate_variant:Nn \fp_zero:N { c }
7388 \cs_generate_variant:Nn \fp_gzero:N { c }

```

`\fp_set:Nn` To trap any input errors, a very simple version of the parser is run here. This will pick  
`\fp_set:cn` up any invalid characters at this stage, saving issues later. The splitting approach is the  
`\fp_gset:Nn` same as the more advanced function later.  
`\fp_gset:cn`  
`\fp_set_aux:NNn`

```

7389 \cs_new_protected_nopar:Npn \fp_set:Nn {
7390   \fp_set_aux:NNn \tl_set:Nn
7391 }
7392 \cs_new_protected_nopar:Npn \fp_gset:Nn {
7393   \fp_set_aux:NNn \tl_gset:Nn
7394 }
7395 \cs_new_protected_nopar:Npn \fp_set_aux:NNn #1#2#3 {
7396   \group_begin:
7397     \fp_split:Nn a {#3}
7398     \fp_standardise:NNNN
7399     \l_fp_input_a_sign_int
7400     \l_fp_input_a_integer_int
7401     \l_fp_input_a_decimal_int
7402     \l_fp_input_a_exponent_int
7403     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7404     \cs_set_protected_nopar:Npx \fp_tmp:w
7405     {
7406       \group_end:
7407       #1 \exp_not:N #2
7408       {
7409         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7410         -
7411         \tex_else:D
7412         +
7413         \tex_fi:D
7414         \int_use:N \l_fp_input_a_integer_int
7415         .
7416         \tex_expandafter:D \use_none:n
7417         \int_use:N \l_fp_input_a_decimal_int
7418         e
7419         \int_use:N \l_fp_input_a_exponent_int
7420       }
7421     }
7422     \fp_tmp:w
7423 }

```

```

7424 \cs_generate_variant:Nn \fp_set:Nn { c }
7425 \cs_generate_variant:Nn \fp_gset:Nn { c }

```

`\fp_set_from_dim:Nn` Here, dimensions are converted to fixed-points *via* a temporary variable. This ensures that they always convert as points. The code is then essentially the same as for `\fp_set:Nn`, but with the dimension passed so that it will be striped of the `pt` on the way through. The passage through a skip is used to remove any rubber part.

```

\fp_set_from_dim:Nn
\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn
\fp_set_from_dim_aux:NNn
\fp_set_from_dim_aux:w
  \l_fp_tmp_dim
  \l_fp_tmp_skip
7426 \cs_new_protected_nopar:Npn \fp_set_from_dim:Nn {
7427   \fp_set_from_dim_aux:NNn \tl_set:Nx
7428 }
7429 \cs_new_protected_nopar:Npn \fp_gset_from_dim:Nn {
7430   \fp_set_from_dim_aux:NNn \tl_gset:Nx
7431 }
7432 \cs_new_protected_nopar:Npn \fp_set_from_dim_aux:NNn #1#2#3 {
7433   \group_begin:
7434     \l_fp_tmp_skip \etex_glueexpr:D #3 \scan_stop:
7435     \l_fp_tmp_dim \l_fp_tmp_skip
7436     \fp_split:Nn a
7437     {
7438       \tex_expandafter:D \fp_set_from_dim_aux:w
7439       \dim_use:N \l_fp_tmp_dim
7440     }
7441     \fp_standardise:NNNN
7442     \l_fp_input_a_sign_int
7443     \l_fp_input_a_integer_int
7444     \l_fp_input_a_decimal_int
7445     \l_fp_input_a_exponent_int
7446     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7447     \cs_set_protected_nopar:Npx \fp_tmp:w
7448     {
7449       \group_end:
7450       #1 \exp_not:N #2
7451       {
7452         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7453         -
7454         \tex_else:D
7455         +
7456         \tex_fi:D
7457         \int_use:N \l_fp_input_a_integer_int
7458         .
7459         \tex_expandafter:D \use_none:n
7460         \int_use:N \l_fp_input_a_decimal_int
7461         e
7462         \int_use:N \l_fp_input_a_exponent_int
7463       }
7464     }
7465     \fp_tmp:w
7466   }
7467   \cs_set_protected_nopar:Npx \fp_set_from_dim_aux:w {

```

```

7468 \cs_set_nopar:Npn \exp_not:N \fp_set_from_dim_aux:w
7469     ##1 \tl_to_str:n { pt } {##1}
7470 }
7471 \fp_set_from_dim_aux:w
7472 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
7473 \cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
7474 \dim_new:N \l_fp_tmp_dim
7475 \skip_new:N \l_fp_tmp_skip

```

\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
7476 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
7477 \cs_new_eq:NN \fp_set_eq:cN \tl_set_eq:cN
7478 \cs_new_eq:NN \fp_set_eq:Nc \tl_set_eq:Nc
7479 \cs_new_eq:NN \fp_set_eq:cc \tl_set_eq:cc
7480 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
7481 \cs_new_eq:NN \fp_gset_eq:cN \tl_gset_eq:cN
7482 \cs_new_eq:NN \fp_gset_eq:Nc \tl_gset_eq:Nc
7483 \cs_new_eq:NN \fp_gset_eq:cc \tl_gset_eq:cc

```

\fp\_show:N Simple showing of the underlying variable.

```

\fp_show:c
7484 \cs_new_eq:NN \fp_show:N \tl_show:N
7485 \cs_new_eq:NN \fp_show:c \tl_show:c

```

\fp\_use:N The idea of the \fp\_use:N function to convert the stored value into something suitable for TeX to use as a number in an expandable manner. The first step is to deal with the sign, then work out how big the input is.

```

\fp_use:c
\fp_use_aux:w
\fp_use_none:w
\fp_use_small:w
\fp_use_large:w
7486 \cs_new_nopar:Npn \fp_use:N #1 {
7487     \tex_expandafter:D \fp_use_aux:w #1 \q_stop
7488 }
7489 \cs_generate_variant:Nn \fp_use:N { c }
7490 \cs_new_nopar:Npn \fp_use_aux:w #1#2 e #3 \q_stop {
7491     \tex_if:D #1 -
7492     -
7493     \tex_fi:D
7494     \tex_ifnum:D #3 > \c_zero
7495     \tex_expandafter:D \fp_use_large:w
7496     \tex_else:D
7497     \tex_ifnum:D #3 < \c_zero
7498     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7499     \fp_use_small:w
7500     \tex_else:D
7501     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7502     \fp_use_none:w
7503     \tex_fi:D
7504     \tex_fi:D
7505     #2 e #3 \q_stop
7506 }

```

When the exponent is zero, the input is simply returned as output.

```
7507 \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.

```
7508 \cs_new_nopar:Npn \fp_use_small:w #1 . #2 e #3 \q_stop {
7509   0 .
7510   \prg_replicate:nn { -#3 - 1 } { 0 }
7511   #1#2
7512 }
```

Life is more complex for large numbers. The decimal point needs to be shuffled, with potentially some zero-filling for very large values.

```
7513 \cs_new_nopar:Npn \fp_use_large:w #1 . #2 e #3 \q_stop {
7514   \tex_ifnum:D #3 < \c_ten
7515     \tex_expandafter:D \fp_use_large_aux_i:w
7516   \tex_else:D
7517     \tex_expandafter:D \fp_use_large_aux_ii:w
7518   \tex_fi:D
7519   #1#2 e #3 \q_stop
7520 }
7521 \cs_new_nopar:Npn \fp_use_large_aux_i:w #1#2 e #3 \q_stop {
7522   #1
7523   \use:c { fp_use_large_aux_ #3 :w } #2 \q_stop
7524 }
7525 \cs_new_nopar:cpn { fp_use_large_aux_1:w } #1#2 \q_stop { #1 . #2 }
7526 \cs_new_nopar:cpn { fp_use_large_aux_2:w } #1#2#3 \q_stop {
7527   #1#2 . #3
7528 }
7529 \cs_new_nopar:cpn { fp_use_large_aux_3:w } #1#2#3#4 \q_stop {
7530   #1#2#3 . #4
7531 }
7532 \cs_new_nopar:cpn { fp_use_large_aux_4:w } #1#2#3#4#5 \q_stop {
7533   #1#2#3#4 . #5
7534 }
7535 \cs_new_nopar:cpn { fp_use_large_aux_5:w } #1#2#3#4#5#6 \q_stop {
7536   #1#2#3#4#5 . #6
7537 }
7538 \cs_new_nopar:cpn { fp_use_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop {
7539   #1#2#3#4#5#6 . #7
7540 }
7541 \cs_new_nopar:cpn { fp_use_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop {
7542   #1#2#3#4#6#7 . #8
7543 }
7544 \cs_new_nopar:cpn { fp_use_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7545   #1#2#3#4#5#6#7#8 . #9
7546 }
7547 \cs_new_nopar:cpn { fp_use_large_aux_9:w } #1 \q_stop { #1 . }
```

```

7548 \cs_new_nopar:Npn \fp_use_large_aux_ii:w #1 e #2 \q_stop {
7549     #1
7550     \prg_replicate:nn { #2 - 9 } { 0 }
7551     .
7552 }

```

## 116.6 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`

```

7553 \cs_new_nopar:Npn \fp_to_dim:N #1 { \fp_use:N #1 pt }
7554 \cs_generate_variant:Nn \fp_to_dim:N { c }

```

`\fp_to_int:N` Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

`\fp_to_int:c`

```

\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

```

```

7555 \cs_new_nopar:Npn \fp_to_int:N #1 {
7556     \tex_expandafter:D \fp_to_int_aux:w #1 \q_stop
7557 }
7558 \cs_generate_variant:Nn \fp_to_int:N { c }
7559 \cs_new_nopar:Npn \fp_to_int_aux:w #1#2 e #3 \q_stop {
7560     \tex_if:D #1 -
7561     -
7562     \tex_fi:D
7563     \tex_ifnum:D #3 < \c_zero
7564     \tex_expandafter:D \fp_to_int_small:w
7565     \tex_else:D
7566     \tex_expandafter:D \fp_to_int_large:w
7567     \tex_fi:D
7568     #2 e #3 \q_stop
7569 }

```

For small numbers, if the decimal part is greater than a half then there is rounding up to do.

```

7570 \cs_new_nopar:Npn \fp_to_int_small:w #1 . #2 e #3 \q_stop {
7571     \tex_ifnum:D #3 > \c_one
7572     \tex_else:D
7573         \tex_ifnum:D #1 < \c_five
7574         0
7575     \tex_else:D
7576         1
7577     \tex_fi:D
7578     \tex_fi:D
7579 }

```

For large numbers, the idea is to split off the part for rounding, do the rounding and fill if needed.

```

7580 \cs_new_nopar:Npn \fp_to_int_large:w #1 . #2 e #3 \q_stop {
7581   \tex_ifnum:D #3 < \c_ten
7582     \tex_expandafter:D \fp_to_int_large_aux_i:w
7583   \tex_else:D
7584     \tex_expandafter:D \fp_to_int_large_aux_ii:w
7585   \tex_fi:D
7586   #1#2 e #3 \q_stop
7587 }
7588 \cs_new_nopar:Npn \fp_to_int_large_aux_i:w #1#2 e #3 \q_stop {
7589   \use:c { fp_to_int_large_aux_ #3 :w } #2 \q_stop {#1}
7590 }
7591 \cs_new_nopar:cpn { fp_to_int_large_aux_1:w } #1#2 \q_stop {
7592   \fp_to_int_large_aux:nnn { #2 0 } {#1}
7593 }
7594 \cs_new_nopar:cpn { fp_to_int_large_aux_2:w } #1#2#3 \q_stop {
7595   \fp_to_int_large_aux:nnn { #3 00 } {#1#2}
7596 }
7597 \cs_new_nopar:cpn { fp_to_int_large_aux_3:w } #1#2#3#4 \q_stop {
7598   \fp_to_int_large_aux:nnn { #4 000 } {#1#2#3}
7599 }
7600 \cs_new_nopar:cpn { fp_to_int_large_aux_4:w } #1#2#3#4#5 \q_stop {
7601   \fp_to_int_large_aux:nnn { #5 0000 } {#1#2#3#4}
7602 }
7603 \cs_new_nopar:cpn { fp_to_int_large_aux_5:w } #1#2#3#4#5#6 \q_stop {
7604   \fp_to_int_large_aux:nnn { #6 00000 } {#1#2#3#4#5}
7605 }
7606 \cs_new_nopar:cpn { fp_to_int_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop {
7607   \fp_to_int_large_aux:nnn { #7 000000 } {#1#2#3#4#5#6}
7608 }
7609 \cs_new_nopar:cpn
7610   { fp_to_int_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop {
7611   \fp_to_int_large_aux:nnn { #8 0000000 } {#1#2#3#4#5#6#7}
7612 }
7613 \cs_new_nopar:cpn
7614   { fp_to_int_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7615   \fp_to_int_large_aux:nnn { #9 00000000 } {#1#2#3#4#5#6#7#8}
7616 }
7617 \cs_new_nopar:cpn { fp_to_int_large_aux_9:w } #1 \q_stop {#1}
7618 \cs_new_nopar:Npn \fp_to_int_large_aux:nnn #1#2#3 {
7619   \tex_ifnum:D #1 < \c_five_hundred_million
7620     #3#2
7621   \tex_else:D
7622     \tex_number:D \etex_numexpr:D #3#2 + 1 \scan_stop:
7623   \tex_fi:D
7624 }
7625 \cs_new_nopar:Npn \fp_to_int_large_aux_ii:w #1 e #2 \q_stop {
7626   #1

```

```

7627 \prg_replicate:nn { #2 - 9 } { 0 }
7628 }

```

\fp\_to\_tl:N Converting to integers in an expandable manner is very similar to simply using floating  
\fp\_to\_tl:c point variables, particularly in the lead-off.

```

\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
7629 \cs_new_nopar:Npn \fp_to_tl:N #1 {
7630   \tex_expandafter:D \fp_to_tl_aux:w #1 \q_stop
7631 }
7632 \cs_generate_variant:Nn \fp_to_tl:N { c }
7633 \cs_new_nopar:Npn \fp_to_tl_aux:w #1#2 e #3 \q_stop {
7634   \tex_if:D #1 -
7635   -
7636   \tex_fi:D
7637   \tex_ifnum:D #3 < \c_zero
7638   \tex_expandafter:D \fp_to_tl_small:w
7639   \tex_else:D
7640   \tex_expandafter:D \fp_to_tl_large:w
7641   \tex_fi:D
7642   #2 e #3 \q_stop
7643 }

```

For ‘large’ numbers (exponent  $\geq 0$ ) there are two cases. For very large exponents ( $\geq 10$ ) life is easy: apart from dropping extra zeros there is no work to do. On the other hand, for intermediate exponent values the decimal needs to be moved, then zeros can be dropped.

```

7644 \cs_new_nopar:Npn \fp_to_tl_large:w #1 e #2 \q_stop {
7645   \tex_ifnum:D #2 < \c_ten
7646   \tex_expandafter:D \fp_to_tl_large_aux_i:w
7647   \tex_else:D
7648   \tex_expandafter:D \fp_to_tl_large_aux_ii:w
7649   \tex_fi:D
7650   #1 e #2 \q_stop
7651 }
7652 \cs_new_nopar:Npn \fp_to_tl_large_aux_i:w #1 e #2 \q_stop {
7653   \use:c { fp_to_tl_large_ #2 :w } #1 \q_stop
7654 }
7655 \cs_new_nopar:Npn \fp_to_tl_large_aux_ii:w #1 . #2 e #3 \q_stop {
7656   #1
7657   \fp_to_tl_large_zeros:NNNNNNNNN #2
7658   e #3
7659 }
7660 \cs_new_nopar:cpn { fp_to_tl_large_0:w } #1 . #2 \q_stop {
7661   #1
7662   \fp_to_tl_large_zeros:NNNNNNNNN #2
7663 }
7664 \cs_new_nopar:cpn { fp_to_tl_large_1:w } #1 . #2#3 \q_stop {
7665   #1#2
7666   \fp_to_tl_large_zeros:NNNNNNNNN #3 0
7667 }

```

```

7668 \cs_new_nopar:cpn { fp_to_tl_large_2:w } #1 . #2#3#4 \q_stop {
7669   #1#2#3
7670   \fp_to_tl_large_zeros:NNNNNNNN #4 00
7671 }
7672 \cs_new_nopar:cpn { fp_to_tl_large_3:w } #1 . #2#3#4#5 \q_stop {
7673   #1#2#3#4
7674   \fp_to_tl_large_zeros:NNNNNNNN #5 000
7675 }
7676 \cs_new_nopar:cpn { fp_to_tl_large_4:w } #1 . #2#3#4#5#6 \q_stop {
7677   #1#2#3#4#5
7678   \fp_to_tl_large_zeros:NNNNNNNN #6 0000
7679 }
7680 \cs_new_nopar:cpn { fp_to_tl_large_5:w } #1 . #2#3#4#5#6#7 \q_stop {
7681   #1#2#3#4#5#6
7682   \fp_to_tl_large_zeros:NNNNNNNN #7 00000
7683 }
7684 \cs_new_nopar:cpn { fp_to_tl_large_6:w } #1 . #2#3#4#5#6#7#8 \q_stop {
7685   #1#2#3#4#5#6#7
7686   \fp_to_tl_large_zeros:NNNNNNNN #8 000000
7687 }
7688 \cs_new_nopar:cpn { fp_to_tl_large_7:w } #1 . #2#3#4#5#6#7#8#9 \q_stop {
7689   #1#2#3#4#5#6#7#8
7690   \fp_to_tl_large_zeros:NNNNNNNN #9 0000000
7691 }
7692 \cs_new_nopar:cpn { fp_to_tl_large_8:w } #1 . {
7693   #1
7694   \use:c { fp_to_tl_large_8_aux:w }
7695 }
7696 \cs_new_nopar:cpn
7697 { fp_to_tl_large_8_aux:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7698   #1#2#3#4#5#6#7#8
7699   \fp_to_tl_large_zeros:NNNNNNNN #9 00000000
7700 }
7701 \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.

```

7702 \cs_new_nopar:Npn \fp_to_tl_small:w #1 e #2 \q_stop {
7703   \tex_ifnum:D #2 = \c_minus_one
7704   \tex_expandafter:D \fp_to_tl_small_one:w
7705   \tex_else:D
7706     \tex_ifnum:D #2 = -\c_two
7707     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7708     \fp_to_tl_small_two:w
7709   \tex_else:D
7710     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7711     \fp_to_tl_small_aux:w
7712   \tex_fi:D

```

```

7713 \tex_fi:D
7714 #1 e #2 \q_stop
7715 }
7716 \cs_new_nopar:Npn \fp_to_tl_small_one:w #1 . #2 e #3 \q_stop {
7717 \tex_ifnum:D \fp_use_ix:NNNNNNNN #2 > \c_four
7718 \tex_ifnum:D
7719 \etex_numexpr:D #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
7720 < \c_one_thousand_million
7721 0.
7722 \tex_expandafter:D \fp_to_tl_small_zeros:NNNNNNNN
7723 \tex_number:D
7724 \etex_numexpr:D
7725 #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
7726 \scan_stop:
7727 \tex_else:D
7728 1
7729 \tex_fi:D
7730 \tex_else:D
7731 0. #1
7732 \fp_to_tl_small_zeros:NNNNNNNN #2
7733 \tex_fi:D
7734 }
7735 \cs_new_nopar:Npn \fp_to_tl_small_two:w #1 . #2 e #3 \q_stop {
7736 \tex_ifnum:D \fp_use_iix_ix:NNNNNNNN #2 > \c_forty_four
7737 \tex_ifnum:D
7738 \etex_numexpr:D #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
7739 < \c_one_thousand_million
7740 0.0
7741 \tex_expandafter:D \fp_to_tl_small_zeros:NNNNNNNN
7742 \tex_number:D
7743 \etex_numexpr:D
7744 #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
7745 \scan_stop:
7746 \tex_else:D
7747 0.1
7748 \tex_fi:D
7749 \tex_else:D
7750 0.0
7751 #1
7752 \fp_to_tl_small_zeros:NNNNNNNN #2
7753 \tex_fi:D
7754 }
7755 \cs_new_nopar:Npn \fp_to_tl_small_aux:w #1 . #2 e #3 \q_stop {
7756 #1
7757 \fp_to_tl_large_zeros:NNNNNNNN #2
7758 e #3
7759 }

```

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.

```

7760 \cs_new_nopar:Npn \fp_to_tl_large_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7761   \tex_ifnum:D #9 = \c_zero
7762     \tex_ifnum:D #8 = \c_zero
7763       \tex_ifnum:D #7 = \c_zero
7764         \tex_ifnum:D #6 = \c_zero
7765           \tex_ifnum:D #5 = \c_zero
7766             \tex_ifnum:D #4 = \c_zero
7767               \tex_ifnum:D #3 = \c_zero
7768                 \tex_ifnum:D #2 = \c_zero
7769                   \tex_ifnum:D #1 = \c_zero
7770                     \tex_else:D
7771                       . #1
7772                         \tex_fi:D
7773                           \tex_else:D
7774                             . #1#2
7775                               \tex_fi:D
7776                                 \tex_else:D
7777                                   . #1#2#3
7778                                     \tex_fi:D
7779                                       \tex_else:D
7780                                         . #1#2#3#4
7781                                           \tex_fi:D
7782                                             \tex_else:D
7783                                               . #1#2#3#4#5
7784                                                 \tex_fi:D
7785                                                   \tex_else:D
7786                                                     . #1#2#3#4#5#6
7787                                                       \tex_fi:D
7788                                                         \tex_else:D
7789                                                           . #1#2#3#4#5#6#7
7790                                                             \tex_fi:D
7791                                                               \tex_else:D
7792                                                                 . #1#2#3#4#5#6#7#8
7793                                                                   \tex_fi:D
7794                                                                     \tex_else:D
7795                                                                     . #1#2#3#4#5#6#7#8#9
7796                                                                     \tex_fi:D
7797   }
7798 \cs_new_nopar:Npn \fp_to_tl_small_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7799   \tex_ifnum:D #9 = \c_zero
7800     \tex_ifnum:D #8 = \c_zero
7801       \tex_ifnum:D #7 = \c_zero
7802         \tex_ifnum:D #6 = \c_zero
7803           \tex_ifnum:D #5 = \c_zero
7804             \tex_ifnum:D #4 = \c_zero
7805               \tex_ifnum:D #3 = \c_zero
7806                 \tex_ifnum:D #2 = \c_zero
7807                   \tex_ifnum:D #1 = \c_zero

```

```

7808         \tex_else:D
7809         #1
7810         \tex_fi:D
7811     \tex_else:D
7812     #1#2
7813     \tex_fi:D
7814     \tex_else:D
7815     #1#2#3
7816     \tex_fi:D
7817     \tex_else:D
7818     #1#2#3#4
7819     \tex_fi:D
7820     \tex_else:D
7821     #1#2#3#4#5
7822     \tex_fi:D
7823     \tex_else:D
7824     #1#2#3#4#5#6
7825     \tex_fi:D
7826     \tex_else:D
7827     #1#2#3#4#5#6#7
7828     \tex_fi:D
7829     \tex_else:D
7830     #1#2#3#4#5#6#7#8
7831     \tex_fi:D
7832     \tex_else:D
7833     #1#2#3#4#5#6#7#8#9
7834     \tex_fi:D
7835 }

```

Some quick ‘return a few’ functions.

```

7836 \cs_new_nopar:Npn \fp_use_iix_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#8#9}
7837 \cs_new_nopar:Npn \fp_use_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#9}
7838 \cs_new_nopar:Npn \fp_use_i_to_vii:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7839     #1#2#3#4#5#6#7
7840 }
7841 \cs_new_nopar:Npn \fp_use_i_to_iix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7842     #1#2#3#4#5#6#7#8
7843 }

```

## 116.7 Rounding numbers

The results may well need to be rounded. A couple of related functions to do this for a stored value.

<pre> \fp_round_figures:Nn \fp_round_figures:cn \fp_ground_figures:Nn \fp_ground_figures:cn \fp_round_figures_aux:NNn </pre>	<p>Rounding to figures needs only an adjustment to the target by one (as the target is in decimal places).</p>
--	--

```

7844 \cs_new_protected_nopar:Npn \fp_round_figures:Nn {
7845   \fp_round_figures_aux:NNn \tl_set:Nn
7846 }
7847 \cs_generate_variant:Nn \fp_round_figures:Nn { c }
7848 \cs_new_protected_nopar:Npn \fp_ground_figures:Nn {
7849   \fp_round_figures_aux:NNn \tl_gset:Nn
7850 }
7851 \cs_generate_variant:Nn \fp_ground_figures:Nn { c }
7852 \cs_new_protected_nopar:Npn \fp_round_figures_aux:NNn #1#2#3 {
7853   \group_begin:
7854     \fp_read:N #2
7855     \int_set:Nn \l_fp_round_target_int { #3 - 1 }
7856     \tex_ifnum:D \l_fp_round_target_int < \c_ten
7857       \tex_expandafter:D \fp_round:
7858     \tex_fi:D
7859     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7860     \cs_set_protected_nopar:Npx \fp_tmp:w
7861     {
7862       \group_end:
7863       #1 \exp_not:N #2
7864       {
7865         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7866         -
7867         \tex_else:D
7868         +
7869         \tex_fi:D
7870         \int_use:N \l_fp_input_a_integer_int
7871         .
7872         \tex_expandafter:D \use_none:n
7873         \int_use:N \l_fp_input_a_decimal_int
7874         e
7875         \int_use:N \l_fp_input_a_exponent_int
7876       }
7877     }
7878     \fp_tmp:w
7879 }

```

```

\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.

```

7880 \cs_new_protected_nopar:Npn \fp_round_places:Nn {
7881   \fp_round_places_aux:NNn \tl_set:Nn
7882 }
7883 \cs_generate_variant:Nn \fp_round_places:Nn { c }
7884 \cs_new_protected_nopar:Npn \fp_ground_places:Nn {
7885   \fp_round_places_aux:NNn \tl_gset:Nn
7886 }
7887 \cs_generate_variant:Nn \fp_ground_places:Nn { c }
7888 \cs_new_protected_nopar:Npn \fp_round_places_aux:NNn #1#2#3 {
7889   \group_begin:

```

```

7890 \fp_read:N #2
7891 \int_set:Nn \l_fp_round_target_int
7892 { #3 + \l_fp_input_a_exponent_int }
7893 \tex_ifnum:D \l_fp_round_target_int < \c_ten
7894 \tex_expandafter:D \fp_round:
7895 \tex_fi:D
7896 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7897 \cs_set_protected_nopar:Npx \fp_tmp:w
7898 {
7899 \group_end:
7900 #1 \exp_not:N #2
7901 {
7902 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7903 -
7904 \tex_else:D
7905 +
7906 \tex_fi:D
7907 \int_use:N \l_fp_input_a_integer_int
7908 .
7909 \tex_expandafter:D \use_none:n
7910 \int_use:N \l_fp_input_a_decimal_int
7911 e
7912 \int_use:N \l_fp_input_a_exponent_int
7913 }
7914 }
7915 \fp_tmp:w
7916 }

```

`\fp_round:` The rounding approach is the same for decimal places and significant figures. There are  
`\fp_round_aux:NNNNNNNN` always nine decimal digits to round, so the code can be written to account for this. The  
`\fp_round_loop:N` 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.

```

7917 \cs_new_protected_nopar:Npn \fp_round: {
7918 \bool_set_false:N \l_fp_round_carry_bool
7919 \l_fp_round_position_int \c_eight
7920 \tl_clear:N \l_fp_round_decimal_tl
7921 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7922 \tex_expandafter:D \use_i:nn \tex_expandafter:D
7923 \fp_round_aux:NNNNNNNN \int_use:N \l_fp_input_a_decimal_int
7924 }
7925 \cs_new_protected_nopar:Npn \fp_round_aux:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7926 \fp_round_loop:N #9#8#7#6#5#4#3#2#1
7927 \bool_if:NT \l_fp_round_carry_bool
7928 { \tex_advance:D \l_fp_input_a_integer_int \c_one }
7929 \l_fp_input_a_decimal_int \l_fp_round_decimal_tl \scan_stop:
7930 \tex_ifnum:D \l_fp_input_a_integer_int < \c_ten
7931 \tex_else:D
7932 \l_fp_input_a_integer_int \c_one
7933 \tex_divide:D \l_fp_input_a_decimal_int \c_ten

```

```

7934     \tex_advance:D \l_fp_input_a_exponent_int \c_one
7935     \tex_fi:D
7936 }
7937 \cs_new_protected_nopar:Npn \fp_round_loop:N #1 {
7938     \tex_ifnum:D \l_fp_round_position_int < \l_fp_round_target_int
7939     \bool_if:NTF \l_fp_round_carry_bool
7940     { \l_fp_tmp_int \etex_numexpr:D #1 + \c_one \scan_stop: }
7941     { \l_fp_tmp_int \etex_numexpr:D #1 \scan_stop: }
7942     \tex_ifnum:D \l_fp_tmp_int = \c_ten
7943     \l_fp_tmp_int \c_zero
7944     \tex_else:D
7945     \bool_set_false:N \l_fp_round_carry_bool
7946     \tex_fi:D
7947     \tl_set:Nx \l_fp_round_decimal_tl
7948     { \int_use:N \l_fp_tmp_int \l_fp_round_decimal_tl }
7949     \tex_else:D
7950     \tl_set:Nx \l_fp_round_decimal_tl { 0 \l_fp_round_decimal_tl }
7951     \tex_ifnum:D \l_fp_round_position_int = \l_fp_round_target_int
7952     \tex_ifnum:D #1 > \c_four
7953     \bool_set_true:N \l_fp_round_carry_bool
7954     \tex_fi:D
7955     \tex_fi:D
7956     \tex_fi:D
7957     \tex_advance:D \l_fp_round_position_int \c_minus_one
7958     \tex_ifnum:D \l_fp_round_position_int > \c_minus_one
7959     \tex_expandafter:D \fp_round_loop:N
7960     \tex_fi:D
7961 }

```

## 116.8 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
7962 \cs_new_protected_nopar:Npn \fp_abs:N {
7963     \fp_abs_aux:NN \tl_set:Nn
7964 }
7965 \cs_new_protected_nopar:Npn \fp_gabs:N {
7966     \fp_abs_aux:NN \tl_gset:Nn
7967 }
7968 \cs_generate_variant:Nn \fp_abs:N { c }
7969 \cs_generate_variant:Nn \fp_gabs:N { c }
7970 \cs_new_protected_nopar:Npn \fp_abs_aux:NN #1#2 {
7971     \group_begin:
7972     \fp_read:N #2
7973     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7974     \cs_set_protected_nopar:Npx \fp_tmp:w
7975     {
7976         \group_end:
7977         #1 \exp_not:N #2

```

```

7978         {
7979         +
7980         \int_use:N \l_fp_input_a_integer_int
7981         .
7982         \tex_expandafter:D \use_none:n
7983         \int_use:N \l_fp_input_a_decimal_int
7984         e
7985         \int_use:N \l_fp_input_a_exponent_int
7986         }
7987     }
7988     \fp_tmp:w
7989 }

\fp_neg:N Just a bit more complex: read the input, reverse the sign and output the result.
\fp_neg:c
\fp_gneg:N 7990 \cs_new_protected_nopar:Npn \fp_neg:N {
\fp_gneg:c 7991 \fp_neg_aux:NN \tl_set:Nn
\fp_neg:NN 7992 }
7993 \cs_new_protected_nopar:Npn \fp_gneg:N {
7994 \fp_neg_aux:NN \tl_gset:Nn
7995 }
7996 \cs_generate_variant:Nn \fp_neg:N { c }
7997 \cs_generate_variant:Nn \fp_gneg:N { c }
7998 \cs_new_protected_nopar:Npn \fp_neg_aux:NN #1#2 {
7999 \group_begin:
8000 \fp_read:N #2
8001 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8002 \tl_set:Nx \l_fp_tmp_tl
8003 {
8004 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8005 +
8006 \tex_else:D
8007 -
8008 \tex_fi:D
8009 \int_use:N \l_fp_input_a_integer_int
8010 .
8011 \tex_expandafter:D \use_none:n
8012 \int_use:N \l_fp_input_a_decimal_int
8013 e
8014 \int_use:N \l_fp_input_a_exponent_int
8015 }
8016 \tex_expandafter:D \group_end: \tex_expandafter:D
8017 #1 \tex_expandafter:D #2 \tex_expandafter:D { \l_fp_tmp_tl }
8018 }

```

## 116.9 Basic arithmetic

`\fp_add:Nn`    The various addition functions are simply different ways to call the single master function  
`\fp_add:cn`    below. This pattern is repeated for the other arithmetic functions.  
`\fp_gadd:Nn`  
`\fp_gadd:cn`    *8019* `\cs_new_protected_nopar:Npn \fp_add:Nn {`  
`\fp_add_aux:NNn`    *8020*    `\fp_add_aux:NNn \tl_set:Nn`  
`\fp_add_core:`    *8021*    `}`  
`\fp_add_sum:`    *8022* `\cs_new_protected_nopar:Npn \fp_gadd:Nn {`  
`\fp_add_difference:`    *8023*    `\fp_add_aux:NNn \tl_gset:Nn`  
*8024*    `}`  
*8025*    `\cs_generate_variant:Nn \fp_add:Nn { c }`  
*8026*    `\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.

```

8027 \cs_new_protected_nopar:Npn \fp_add_aux:NNn #1#2#3 {
8028   \group_begin:
8029     \fp_read:N #2
8030     \fp_split:Nn b {#3}
8031     \fp_standardise:NNNN
8032     \l_fp_input_b_sign_int
8033     \l_fp_input_b_integer_int
8034     \l_fp_input_b_decimal_int
8035     \l_fp_input_b_exponent_int
8036     \fp_add_core:
8037     \fp_tmp:w #1#2
8038   }
8039   \cs_new_protected_nopar:Npn \fp_add_core: {
8040     \fp_level_input_exponents:
8041     \tex_ifnum:D
8042       \etex_numexpr:D
8043       \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8044       \scan_stop:
8045       > \c_zero
8046       \tex_expandafter:D \fp_add_sum:
8047     \tex_else:D
8048       \tex_expandafter:D \fp_add_difference:
8049     \tex_fi:D
8050     \l_fp_output_exponent_int \l_fp_input_a_exponent_int
8051     \fp_standardise:NNNN
8052     \l_fp_output_sign_int
8053     \l_fp_output_integer_int
8054     \l_fp_output_decimal_int
8055     \l_fp_output_exponent_int
8056     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8057     {
8058       \group_end:

```

```

8059     ##1 ##2
8060     {
8061         \tex_ifnum:D \l_fp_output_sign_int < \c_zero
8062         -
8063         \tex_else:D
8064         +
8065         \tex_fi:D
8066         \int_use:N \l_fp_output_integer_int
8067         .
8068         \tex_expandafter:D \use_none:n
8069         \tex_number:D \etex_numexpr:D
8070             \l_fp_output_decimal_int + \c_one_thousand_million
8071         e
8072         \int_use:N \l_fp_output_exponent_int
8073     }
8074 }
8075 }

```

Finding the sum of two numbers is trivially easy.

```

8076 \cs_new_protected_nopar:Npn \fp_add_sum: {
8077     \l_fp_output_sign_int \l_fp_input_a_sign_int
8078     \l_fp_output_integer_int
8079     \etex_numexpr:D
8080         \l_fp_input_a_integer_int + \l_fp_input_b_integer_int
8081     \scan_stop:
8082     \l_fp_output_decimal_int
8083     \etex_numexpr:D
8084         \l_fp_input_a_decimal_int + \l_fp_input_b_decimal_int
8085     \scan_stop:
8086     \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
8087     \tex_else:D
8088         \tex_advance:D \l_fp_output_integer_int \c_one
8089         \tex_advance:D \l_fp_output_decimal_int -\c_one_thousand_million
8090     \tex_fi:D
8091 }

```

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.

```

8092 \cs_new_protected_nopar:Npn \fp_add_difference: {
8093     \l_fp_output_integer_int
8094     \etex_numexpr:D
8095         \l_fp_input_a_integer_int - \l_fp_input_b_integer_int
8096     \scan_stop:
8097     \l_fp_output_decimal_int
8098     \etex_numexpr:D
8099         \l_fp_input_a_decimal_int - \l_fp_input_b_decimal_int

```

```

8100 \scan_stop:
8101 \tex_ifnum:D \l_fp_output_decimal_int < \c_zero
8102 \tex_advance:D \l_fp_output_integer_int \c_minus_one
8103 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8104 \tex_fi:D
8105 \tex_ifnum:D \l_fp_output_integer_int < \c_zero
8106 \l_fp_output_sign_int \l_fp_input_b_sign_int
8107 \tex_ifnum:D \l_fp_output_decimal_int = \c_zero
8108 \l_fp_output_integer_int -\l_fp_output_integer_int
8109 \tex_else:D
8110 \l_fp_output_decimal_int
8111 \etex_numexpr:D
8112 \c_one_thousand_million - \l_fp_output_decimal_int
8113 \scan_stop:
8114 \l_fp_output_integer_int
8115 \etex_numexpr:D
8116 - \l_fp_output_integer_int - \c_one
8117 \scan_stop:
8118 \tex_fi:D
8119 \tex_else:D
8120 \l_fp_output_sign_int \l_fp_input_a_sign_int
8121 \tex_fi:D
8122 }

```

`\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`

```

8123 \cs_new_protected_nopar:Npn \fp_sub:Nn {
8124 \fp_sub_aux:NNn \tl_set:Nn
8125 }
8126 \cs_new_protected_nopar:Npn \fp_gsub:Nn {
8127 \fp_sub_aux:NNn \tl_gset:Nn
8128 }
8129 \cs_generate_variant:Nn \fp_sub:Nn { c }
8130 \cs_generate_variant:Nn \fp_gsub:Nn { c }
8131 \cs_new_protected_nopar:Npn \fp_sub_aux:NNn #1#2#3 {
8132 \group_begin:
8133 \fp_read:N #2
8134 \fp_split:Nn b {#3}
8135 \fp_standardise:NNNN
8136 \l_fp_input_b_sign_int
8137 \l_fp_input_b_integer_int
8138 \l_fp_input_b_decimal_int
8139 \l_fp_input_b_exponent_int
8140 \tex_multiply:D \l_fp_input_b_sign_int \c_minus_one
8141 \fp_add_core:
8142 \fp_tmp:w #1#2
8143 }

```

```

\fp_mul:Nn The pattern is much the same for multiplication.
\fp_mul:cn
\fp_gmul:Nn 8144 \cs_new_protected_nopar:Npn \fp_mul:Nn {
\fp_gmul:cn 8145 \fp_mul_aux:NNn \tl_set:Nn
8146 }
\fp_mul_aux:NNn 8147 \cs_new_protected_nopar:Npn \fp_gmul:Nn {
\fp_mul_internal: 8148 \fp_mul_aux:NNn \tl_gset:Nn
8149 }
\fp_mul_split:NNNN 8150 \cs_generate_variant:Nn \fp_mul:Nn { c }
\fp_mul_split:w 8151 \cs_generate_variant:Nn \fp_gmul:Nn { c }
\fp_mul_end_level:
\fp_mul_end_level:NNNNNNNN

```

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

```

8152 \cs_new_protected_nopar:Npn \fp_mul_aux:NNn #1#2#3 {
8153 \group_begin:
8154 \fp_read:N #2
8155 \fp_split:Nn b {#3}
8156 \fp_standardise:NNNN
8157 \l_fp_input_b_sign_int
8158 \l_fp_input_b_integer_int
8159 \l_fp_input_b_decimal_int
8160 \l_fp_input_b_exponent_int
8161 \fp_mul_internal:
8162 \l_fp_output_exponent_int
8163 \etex_numexpr:D
8164 \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
8165 \scan_stop:
8166 \fp_standardise:NNNN
8167 \l_fp_output_sign_int
8168 \l_fp_output_integer_int
8169 \l_fp_output_decimal_int
8170 \l_fp_output_exponent_int
8171 \cs_set_protected_nopar:Npx \fp_tmp:w
8172 {
8173 \group_end:
8174 #1 \exp_not:N #2
8175 {
8176 \tex_ifnum:D
8177 \etex_numexpr:D
8178 \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8179 < \c_zero
8180 \tex_ifnum:D
8181 \etex_numexpr:D
8182 \l_fp_output_integer_int + \l_fp_output_decimal_int
8183 = \c_zero
8184 +
8185 \tex_else:D

```

```

8186         -
8187         \tex_fi:D
8188         \tex_else:D
8189         +
8190         \tex_fi:D
8191         \int_use:N \l_fp_output_integer_int
8192         .
8193         \tex_expandafter:D \use_none:n
8194         \tex_number:D \etex_numexpr:D
8195         \l_fp_output_decimal_int + \c_one_thousand_million
8196         e
8197         \int_use:N \l_fp_output_exponent_int
8198     }
8199 }
8200 \fp_tmp:w
8201 }

```

Done separately so that the internal use is a bit easier.

```

8202 \cs_new_protected_nopar:Npn \fp_mul_internal: {
8203   \fp_mul_split:NNNN \l_fp_input_a_decimal_int
8204     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8205   \fp_mul_split:NNNN \l_fp_input_b_decimal_int
8206     \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8207   \l_fp_mul_output_int \c_zero
8208   \tl_clear:N \l_fp_mul_output_tl
8209   \fp_mul_product:NN \l_fp_mul_a_i_int          \l_fp_mul_b_iii_int
8210   \fp_mul_product:NN \l_fp_mul_a_ii_int         \l_fp_mul_b_ii_int
8211   \fp_mul_product:NN \l_fp_mul_a_iii_int        \l_fp_mul_b_i_int
8212   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8213   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_iii_int
8214   \fp_mul_product:NN \l_fp_mul_a_i_int          \l_fp_mul_b_ii_int
8215   \fp_mul_product:NN \l_fp_mul_a_ii_int         \l_fp_mul_b_i_int
8216   \fp_mul_product:NN \l_fp_mul_a_iii_int        \l_fp_input_b_integer_int
8217   \fp_mul_end_level:
8218   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_ii_int
8219   \fp_mul_product:NN \l_fp_mul_a_i_int          \l_fp_mul_b_i_int
8220   \fp_mul_product:NN \l_fp_mul_a_ii_int        \l_fp_input_b_integer_int
8221   \fp_mul_end_level:
8222   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_i_int
8223   \fp_mul_product:NN \l_fp_mul_a_i_int          \l_fp_input_b_integer_int
8224   \fp_mul_end_level:
8225   \l_fp_output_decimal_int 0 \l_fp_mul_output_tl \scan_stop:
8226   \tl_clear:N \l_fp_mul_output_tl
8227   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_input_b_integer_int
8228   \fp_mul_end_level:
8229   \l_fp_output_integer_int 0 \l_fp_mul_output_tl \scan_stop:
8230 }

```

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.

```

8231 \cs_new_protected_nopar:Npn \fp_mul_split:NNNN #1#2#3#4 {
8232   \tex_advance:D #1 \c_one_thousand_million
8233   \cs_set_protected_nopar:Npn \fp_mul_split_aux:w
8234     ##1##2##3##4##5##6##7##8##9 \q_stop {
8235     #2 ##2##3##4 \scan_stop:
8236     #3 ##5##6##7 \scan_stop:
8237     #4 ##8##9 \scan_stop:
8238   }
8239   \tex_expandafter:D \fp_mul_split_aux:w \int_use:N #1 \q_stop
8240   \tex_advance:D #1 -\c_one_thousand_million
8241 }
8242 \cs_new_protected_nopar:Npn \fp_mul_product:NN #1#2 {
8243   \l_fp_mul_output_int
8244   \etex_numexpr:D \l_fp_mul_output_int + #1 * #2 \scan_stop:
8245 }

```

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.

```

8246 \cs_new_protected_nopar:Npn \fp_mul_end_level: {
8247   \tex_advance:D \l_fp_mul_output_int \c_one_thousand_million
8248   \tex_expandafter:D \use_i:nn \tex_expandafter:D
8249   \fp_mul_end_level:NNNNNNNNN \int_use:N \l_fp_mul_output_int
8250 }
8251 \cs_new_protected_nopar:Npn \fp_mul_end_level:NNNNNNNNN
8252   #1#2#3#4#5#6#7#8#9 {
8253   \tl_set:Nx \l_fp_mul_output_tl { #7#8#9 \l_fp_mul_output_tl }
8254   \l_fp_mul_output_int #1#2#3#4#5#6 \scan_stop:
8255 }

```

\fp\_div:Nn The pattern is much the same for multiplication.

```

\fp_div:cn
\fp_gdiv:Nn
\fp_gdiv:cn
\fp_div_aux:NNn
\fp_div_internal:
  \fp_div_loop:
\fp_div_divide:
\fp_div_divide_aux:
  \fp_div_store:
\fp_div_store_integer:
\fp_div_store_decimal:

```

```

8256 \cs_new_protected_nopar:Npn \fp_div:Nn {
8257   \fp_div_aux:NNn \tl_set:Nn
8258 }
8259 \cs_new_protected_nopar:Npn \fp_gdiv:Nn {
8260   \fp_div_aux:NNn \tl_gset:Nn
8261 }
8262 \cs_generate_variant:Nn \fp_div:Nn { c }
8263 \cs_generate_variant:Nn \fp_gdiv:Nn { c }

```

Division proper starts with a couple of tests. If the denominator is zero then a error is issued. On the other hand, if the numerator is zero then the result must be 0.0 and can be given with no further work.

```

8264 \cs_new_protected_nopar:Npn \fp_div_aux:NNn #1#2#3 {

```

```

8265 \group_begin:
8266   \fp_read:N #2
8267   \fp_split:Nn b {#3}
8268   \fp_standardise:NNNN
8269   \l_fp_input_b_sign_int
8270   \l_fp_input_b_integer_int
8271   \l_fp_input_b_decimal_int
8272   \l_fp_input_b_exponent_int
8273   \tex_ifnum:D
8274   \etex_numexpr:D
8275     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
8276     = \c_zero
8277   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8278   {
8279     \group_end:
8280     #1 \exp_not:N #2 { \c_undefined_fp }
8281   }
8282   \tex_else:D
8283   \tex_ifnum:D
8284   \etex_numexpr:D
8285     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8286     = \c_zero
8287   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8288   {
8289     \group_end:
8290     #1 \exp_not:N #2 { \c_zero_fp }
8291   }
8292   \tex_else:D
8293   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8294   \fp_div_internal:
8295   \tex_fi:D
8296   \tex_fi:D
8297   \fp_tmp:w #1#2
8298 }

```

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.

```

8299 \cs_new_protected_nopar:Npn \fp_div_internal: {
8300   \l_fp_output_integer_int \c_zero
8301   \l_fp_output_decimal_int \c_zero
8302   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
8303   \l_fp_div_offset_int \c_one_hundred_million
8304   \fp_div_loop:
8305   \l_fp_output_exponent_int
8306   \etex_numexpr:D
8307     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int

```

```

8308     \scan_stop:
8309 \fp_standardise:NNNN
8310     \l_fp_output_sign_int
8311     \l_fp_output_integer_int
8312     \l_fp_output_decimal_int
8313     \l_fp_output_exponent_int
8314 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8315 {
8316     \group_end:
8317     ##1 ##2
8318     {
8319         \tex_ifnum:D
8320         \etex_numexpr:D
8321             \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8322             < \c_zero
8323         \tex_ifnum:D
8324         \etex_numexpr:D
8325             \l_fp_output_integer_int + \l_fp_output_decimal_int
8326             = \c_zero
8327         +
8328         \tex_else:D
8329         -
8330         \tex_if:D
8331         \tex_else:D
8332         +
8333         \tex_if:D
8334         \int_use:N \l_fp_output_integer_int
8335         .
8336         \tex_expandafter:D \use_none:n
8337         \tex_number:D \etex_numexpr:D
8338             \l_fp_output_decimal_int + \c_one_thousand_million
8339         \scan_stop:
8340         e
8341         \int_use:N \l_fp_output_exponent_int
8342     }
8343 }
8344 }

```

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.

```

8345 \cs_new_protected_nopar:Npn \fp_div_loop: {
8346     \l_fp_count_int \c_zero
8347     \fp_div_divide:
8348     \fp_div_store:
8349     \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8350     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8351     \tex_expandafter:D \fp_div_loop_step:w
8352     \int_use:N \l_fp_input_a_decimal_int \q_stop
8353     \tex_ifnum:D

```

```

8354 \etex_numexpr:D
8355 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8356 > \c_zero
8357 \tex_ifnum:D \l_fp_div_offset_int > \c_zero
8358 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8359 \fp_div_loop:
8360 \tex_fi:D
8361 \tex_fi:D
8362 }

```

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.

```

8363 \cs_new_protected_nopar:Npn \fp_div_divide: {
8364 \tex_ifnum:D \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
8365 \tex_expandafter:D \fp_div_divide_aux:
8366 \tex_else:D
8367 \tex_ifnum:D \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
8368 \tex_else:D
8369 \tex_ifnum:D
8370 \l_fp_input_a_decimal_int < \l_fp_input_b_decimal_int
8371 \tex_else:D
8372 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8373 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8374 \tex_expandafter:D \fp_div_divide_aux:
8375 \tex_fi:D
8376 \tex_fi:D
8377 \tex_fi:D
8378 }
8379 \cs_new_protected_nopar:Npn \fp_div_divide_aux: {
8380 \tex_advance:D \l_fp_count_int \c_one
8381 \tex_advance:D \l_fp_input_a_integer_int -\l_fp_input_b_integer_int
8382 \tex_advance:D \l_fp_input_a_decimal_int -\l_fp_input_b_decimal_int
8383 \tex_ifnum:D \l_fp_input_a_decimal_int < \c_zero
8384 \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
8385 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8386 \tex_fi:D
8387 \fp_div_divide:
8388 }

```

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.

```

8389 \cs_new_protected_nopar:Npn \fp_div_store: { }
8390 \cs_new_protected_nopar:Npn \fp_div_store_integer: {
8391 \l_fp_output_integer_int \l_fp_count_int
8392 \cs_set_eq:NN \fp_div_store: \fp_div_store_decimal:

```

```

8393 }
8394 \cs_new_protected_nopar:Npn \fp_div_store_decimal: {
8395   \l_fp_output_decimal_int
8396   \etex_numexpr:D
8397     \l_fp_output_decimal_int +
8398     \l_fp_count_int * \l_fp_div_offset_int
8399   \scan_stop:
8400   \tex_divide:D \l_fp_div_offset_int \c_ten
8401 }
8402 \cs_new_protected_nopar:Npn
8403   \fp_div_loop_step:w #1#2#3#4#5#6#7#8#9 \q_stop {
8404   \l_fp_input_a_integer_int
8405   \etex_numexpr:D
8406     #2 + \l_fp_input_a_integer_int
8407   \scan_stop:
8408   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
8409 }

```

## 116.10 Arithmetic for internal use

For the more complex functions, it is only possible to deliver reliable 10 digit accuracy if the internal calculations are carried out to a higher degree of precision. This is done using a second set of functions so that the ‘user’ versions are not slowed down. These versions are also focussed on the needs of internal calculations. No error checking, sign checking or exponent levelling is done. For addition and subtraction, the arguments are:

- Integer part of input a.
- Decimal part of input a.
- Additional decimal part of input a.
- Integer part of input b.
- Decimal part of input b.
- Additional decimal part of input b.
- Integer part of output.
- Decimal part of output.
- Additional decimal part of output.

The situation for multiplication and division is a little different as they only deal with the decimal part.

`\fp_add:NNNNNNNNN` The internal sum is always exactly that: it is always a sum and there is no sign check.

```

8410 \cs_new_protected_nopar:Npn \fp_add:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8411   #7 \etex_numexpr:D #1 + #4 \scan_stop:
8412   #8 \etex_numexpr:D #2 + #5 \scan_stop:
8413   #9 \etex_numexpr:D #3 + #6 \scan_stop:
8414   \tex_ifnum:D #9 < \c_one_thousand_million
8415   \tex_else:D
8416     \tex_advance:D #8 \c_one
8417     \tex_advance:D #9 -\c_one_thousand_million
8418   \tex_fi:D
8419   \tex_ifnum:D #8 < \c_one_thousand_million
8420   \tex_else:D
8421     \tex_advance:D #7 \c_one
8422     \tex_advance:D #8 -\c_one_thousand_million
8423   \tex_fi:D
8424 }

```

`\fp_sub:NNNNNNNNNN` 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.

```

8425 \cs_new_protected_nopar:Npn \fp_sub:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8426   #7 \etex_numexpr:D #1 - #4 \scan_stop:
8427   #8 \etex_numexpr:D #2 - #5 \scan_stop:
8428   #9 \etex_numexpr:D #3 - #6 \scan_stop:
8429   \tex_ifnum:D #9 < \c_zero
8430     \tex_advance:D #8 \c_minus_one
8431     \tex_advance:D #9 \c_one_thousand_million
8432   \tex_fi:D
8433   \tex_ifnum:D #8 < \c_zero
8434     \tex_advance:D #7 \c_minus_one
8435     \tex_advance:D #8 \c_one_thousand_million
8436   \tex_fi:D
8437   \tex_ifnum:D #7 < \c_zero
8438     \tex_ifnum:D \etex_numexpr:D #8 + #9 = \c_zero
8439       #7 -#7
8440     \tex_else:D
8441       \tex_advance:D #7 \c_one
8442       #8 \etex_numexpr:D \c_one_thousand_million - #8 \scan_stop:
8443       #9 \etex_numexpr:D \c_one_thousand_million - #9 \scan_stop:
8444     \tex_fi:D
8445   \tex_fi:D
8446 }

```

`\fp_mul:NNNNNNN` Decimal-part only multiplication but with higher accuracy than the user version.

```

8447 \cs_new_protected_nopar:Npn \fp_mul:NNNNNNN #1#2#3#4#5#6 {
8448   \fp_mul_split:NNNN #1
8449     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int

```

```

8450 \fp_mul_split:NNNN #2
8451   \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
8452 \fp_mul_split:NNNN #3
8453   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8454 \fp_mul_split:NNNN #4
8455   \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
8456 \l_fp_mul_output_int \c_zero
8457 \tl_clear:N \l_fp_mul_output_tl
8458 \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_vi_int
8459 \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_v_int
8460 \fp_mul_product:NN \l_fp_mul_a_iii_int    \l_fp_mul_b_iv_int
8461 \fp_mul_product:NN \l_fp_mul_a_iv_int     \l_fp_mul_b_iii_int
8462 \fp_mul_product:NN \l_fp_mul_a_v_int      \l_fp_mul_b_ii_int
8463 \fp_mul_product:NN \l_fp_mul_a_vi_int     \l_fp_mul_b_i_int
8464 \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8465 \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_v_int
8466 \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_iv_int
8467 \fp_mul_product:NN \l_fp_mul_a_iii_int    \l_fp_mul_b_iii_int
8468 \fp_mul_product:NN \l_fp_mul_a_iv_int     \l_fp_mul_b_ii_int
8469 \fp_mul_product:NN \l_fp_mul_a_v_int      \l_fp_mul_b_i_int
8470 \fp_mul_end_level:
8471 \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_iv_int
8472 \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_iii_int
8473 \fp_mul_product:NN \l_fp_mul_a_iii_int    \l_fp_mul_b_ii_int
8474 \fp_mul_product:NN \l_fp_mul_a_iv_int     \l_fp_mul_b_i_int
8475 \fp_mul_end_level:
8476 \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_iii_int
8477 \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_ii_int
8478 \fp_mul_product:NN \l_fp_mul_a_iii_int    \l_fp_mul_b_i_int
8479 \fp_mul_end_level:
8480 #6 0 \l_fp_mul_output_tl \scan_stop:
8481 \tl_clear:N \l_fp_mul_output_tl
8482 \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_ii_int
8483 \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_i_int
8484 \fp_mul_end_level:
8485 \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_i_int
8486 \fp_mul_end_level:
8487 \fp_mul_end_level:
8488 #5 0 \l_fp_mul_output_tl \scan_stop:
8489 }

```

`\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.

```

8490 \cs_new_protected_nopar:Npn \fp_mul:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8491   \fp_mul_split:NNNN #2
8492   \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8493 \fp_mul_split:NNNN #3
8494   \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
8495 \fp_mul_split:NNNN #5

```

```

8496 \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8497 \fp_mul_split:NNNN #6
8498 \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
8499 \l_fp_mul_output_int \c_zero
8500 \tl_clear:N \l_fp_mul_output_tl
8501 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
8502 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
8503 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
8504 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
8505 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
8506 \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
8507 \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8508 \fp_mul_product:NN #1 \l_fp_mul_b_vi_int
8509 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
8510 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
8511 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
8512 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
8513 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
8514 \fp_mul_product:NN \l_fp_mul_a_vi_int #4
8515 \fp_mul_end_level:
8516 \fp_mul_product:NN #1 \l_fp_mul_b_v_int
8517 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
8518 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
8519 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
8520 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
8521 \fp_mul_product:NN \l_fp_mul_a_v_int #4
8522 \fp_mul_end_level:
8523 \fp_mul_product:NN #1 \l_fp_mul_b_iv_int
8524 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
8525 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8526 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8527 \fp_mul_product:NN \l_fp_mul_a_iv_int #4
8528 \fp_mul_end_level:
8529 #9 0 \l_fp_mul_output_tl \scan_stop:
8530 \tl_clear:N \l_fp_mul_output_tl
8531 \fp_mul_product:NN #1 \l_fp_mul_b_iii_int
8532 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8533 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8534 \fp_mul_product:NN \l_fp_mul_a_iii_int #4
8535 \fp_mul_end_level:
8536 \fp_mul_product:NN #1 \l_fp_mul_b_ii_int
8537 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8538 \fp_mul_product:NN \l_fp_mul_a_ii_int #4
8539 \fp_mul_end_level:
8540 \fp_mul_product:NN #1 \l_fp_mul_b_i_int
8541 \fp_mul_product:NN \l_fp_mul_a_i_int #4
8542 \fp_mul_end_level:
8543 #8 0 \l_fp_mul_output_tl \scan_stop:
8544 \tl_clear:N \l_fp_mul_output_tl
8545 \fp_mul_product:NN #1 #4

```

```

8546 \fp_mul_end_level:
8547 #7 0 \l_fp_mul_output_tl \scan_stop:
8548 }

```

`\fp_div_integer:NNNNN` Here, division is always by an integer, and so it is possible to use TeX's native calculations rather than doing it in macros. The idea here is to divide the decimal part, find any remainder, then do the real division of the two parts before adding in what is needed for the remainder.

```

8549 \cs_new_protected_nopar:Npn \fp_div_integer:NNNNN #1#2#3#4#5 {
8550 \l_fp_tmp_int #1
8551 \tex_divide:D \l_fp_tmp_int #3
8552 \l_fp_tmp_int \etex_numexpr:D #1 - \l_fp_tmp_int * #3 \scan_stop:
8553 #4 #1
8554 \tex_divide:D #4 #3
8555 #5 #2
8556 \tex_divide:D #5 #3
8557 \tex_multiply:D \l_fp_tmp_int \c_one_thousand
8558 \tex_divide:D \l_fp_tmp_int #3
8559 #5 \etex_numexpr:D #5 + \l_fp_tmp_int * \c_one_million \scan_stop:
8560 \tex_ifnum:D #5 > \c_one_thousand_million
8561 \tex_advance:D #4 \c_one
8562 \tex_advancd:D #5 -\c_one_thousand_million
8563 \tex_fi:D
8564 }

```

`\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. `\fp_extended_normalise_aux_i:w` This function simply calls the two loops to shift the input to the point of having a zero exponent. `\fp_extended_normalise_aux_ii:w`

```

\fp_extended_normalise_aux:NNNNNNNNN
8565 \cs_new_protected_nopar:Npn \fp_extended_normalise: {
8566 \fp_extended_normalise_aux_i:
8567 \fp_extended_normalise_aux_ii:
8568 }
8569 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i: {
8570 \tex_ifnum:D \l_fp_input_a_exponent_int > \c_zero
8571 \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8572 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8573 \tex_expandafter:D \fp_extended_normalise_aux_i:w
8574 \int_use:N \l_fp_input_a_decimal_int \q_stop
8575 \tex_expandafter:D \fp_extended_normalise_aux_i:
8576 \tex_fi:D
8577 }
8578 \cs_new_protected_nopar:Npn
8579 \fp_extended_normalise_aux_i:w #1#2#3#4#5#6#7#8#9 \q_stop {
8580 \l_fp_input_a_integer_int
8581 \etex_numexpr:D \l_fp_input_a_integer_int + #2 \scan_stop:
8582 \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:

```

```

8583 \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
8584 \tex_expandafter:D \fp_extended_normalise_aux_ii:w
8585 \int_use:N \l_fp_input_a_extended_int \q_stop
8586 }
8587 \cs_new_protected_nopar:Npn
8588 \fp_extended_normalise_aux_ii:w #1#2#3#4#5#6#7#8#9 \q_stop {
8589 \l_fp_input_a_decimal_int
8590 \tex_numexpr:D \l_fp_input_a_decimal_int + #2 \scan_stop:
8591 \l_fp_input_a_extended_int #3#4#5#6#7#8#9 0 \scan_stop:
8592 \tex_advance:D \l_fp_input_a_exponent_int \c_minus_one
8593 }
8594 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii: {
8595 \tex_ifnum:D \l_fp_input_a_exponent_int < \c_zero
8596 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8597 \tex_expandafter:D \use_i:nn \tex_expandafter:D
8598 \fp_extended_normalise_ii_aux:NNNNNNNNN
8599 \int_use:N \l_fp_input_a_decimal_int
8600 \tex_expandafter:D \fp_extended_normalise_aux_ii:
8601 \tex_fi:D
8602 }
8603 \cs_new_protected_nopar:Npn
8604 \fp_extended_normalise_ii_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8605 \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
8606 \l_fp_input_a_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
8607 \tex_else:D
8608 \tl_set:Nx \l_fp_tmp_tl
8609 {
8610 \int_use:N \l_fp_input_a_integer_int
8611 #1#2#3#4#5#6#7#8
8612 }
8613 \l_fp_input_a_integer_int \c_zero
8614 \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
8615 \tex_fi:D
8616 \tex_divide:D \l_fp_input_a_extended_int \c_ten
8617 \tl_set:Nx \l_fp_tmp_tl
8618 {
8619 #9
8620 \int_use:N \l_fp_input_a_extended_int
8621 }
8622 \l_fp_input_a_extended_int \l_fp_tmp_tl \scan_stop:
8623 \tex_advance:D \l_fp_input_a_exponent_int \c_one
8624 }

```

\fp\_extended\_normalise\_output:  
ormalise\_output\_aux\_i:NNNNNNNNN  
ormalise\_output\_aux\_ii:NNNNNNNNN  
extended\_normalise\_output\_aux:N

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.

```

8625 \cs_new_protected_nopar:Npn \fp_extended_normalise_output: {
8626 \tex_ifnum:D \l_fp_output_integer_int > \c_nine
8627 \tex_advance:D \l_fp_output_integer_int \c_one_thousand_million

```

```

8628 \tex_expandafter:D \use_i:nn \tex_expandafter:D
8629 \fp_extended_normalise_output_aux_i:NNNNNNNNN
8630 \int_use:N \l_fp_output_integer_int
8631 \tex_expandafter:D \fp_extended_normalise_output:
8632 \tex_fi:D
8633 }
8634 \cs_new_protected_nopar:Npn
8635 \fp_extended_normalise_output_aux_i:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8636 \l_fp_output_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
8637 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8638 \tl_set:Nx \l_fp_tmp_tl
8639 {
8640 #9
8641 \tex_expandafter:D \use_none:n
8642 \int_use:N \l_fp_output_decimal_int
8643 }
8644 \tex_expandafter:D \fp_extended_normalise_output_aux_ii:NNNNNNNNN
8645 \l_fp_tmp_tl
8646 }
8647 \cs_new_protected_nopar:Npn
8648 \fp_extended_normalise_output_aux_ii:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8649 \l_fp_output_decimal_int #1#2#3#4#5#6#7#8#9 \scan_stop:
8650 \fp_extended_normalise_output_aux:N
8651 }
8652 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux:N #1 {
8653 \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
8654 \tex_divide:D \l_fp_output_extended_int \c_ten
8655 \tl_set:Nx \l_fp_tmp_tl
8656 {
8657 #1
8658 \tex_expandafter:D \use_none:n
8659 \int_use:N \l_fp_output_extended_int
8660 }
8661 \l_fp_output_extended_int \l_fp_tmp_tl \scan_stop:
8662 \tex_advance:D \l_fp_output_exponent_int \c_one
8663 }

```

## 116.11 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  $-\pi < x \leq \pi$ .

```

8664 \cs_new_protected_nopar:Npn \fp_trig_normalise: {
8665 \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
8666 \l_fp_input_a_extended_int \c_zero
8667 \fp_extended_normalise:
8668 \fp_trig_normalise_aux:
8669 \tex_ifnum:D \l_fp_input_a_integer_int < \c_zero

```

```

8670     \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
8671     \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
8672     \tex_fi:D
8673     \tex_expandafter:D \fp_trig_octant:
8674 \tex_else:D
8675     \l_fp_input_a_sign_int      \c_one
8676     \l_fp_output_integer_int    \c_zero
8677     \l_fp_output_decimal_int    \c_zero
8678     \l_fp_output_exponent_int   \c_zero
8679     \tex_expandafter:D \fp_trig_overflow_msg:
8680 \tex_fi:D
8681 }
8682 \cs_new_protected_nopar:Npn \fp_trig_normalise_aux: {
8683     \tex_ifnum:D \l_fp_input_a_integer_int > \c_three
8684     \fp_sub:NNNNNNNNN
8685     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8686     \l_fp_input_a_extended_int
8687     \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
8688     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8689     \l_fp_input_a_extended_int
8690     \tex_expandafter:D \fp_trig_normalise_aux:
8691 \tex_else:D
8692     \tex_ifnum:D \l_fp_input_a_integer_int > \c_two
8693     \tex_ifnum:D \l_fp_input_a_decimal_int > \c_fp_pi_decimal_int
8694     \fp_sub:NNNNNNNNN
8695     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8696     \l_fp_input_a_extended_int
8697     \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
8698     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8699     \l_fp_input_a_extended_int
8700     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8701     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8702     \tex_expandafter:D \fp_trig_normalise_aux:
8703     \tex_fi:D
8704 \tex_fi:D
8705 \tex_fi:D
8706 }

```

`\fp_trig_octant:` Here, the input is further reduced into the range  $0 \leq x < \pi/4$ . This is pretty simple: 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  $\pi/2$ .

```

8707 \cs_new_protected_nopar:Npn \fp_trig_octant: {
8708     \l_fp_trig_octant_int \c_one
8709     \fp_trig_octant_aux:
8710     \tex_ifnum:D \l_fp_input_a_decimal_int < \c_ten
8711     \l_fp_input_a_decimal_int \c_zero
8712     \l_fp_input_a_extended_int \c_zero
8713     \tex_fi:D

```

```

8714 \tex_ifodd:D \l_fp_trig_octant_int
8715 \tex_else:D
8716   \fp_sub:NNNNNNNNN
8717   \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
8718   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8719   \l_fp_input_a_extended_int
8720   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8721   \l_fp_input_a_extended_int
8722 \tex_fi:D
8723 }
8724 \cs_new_protected_nopar:Npn \fp_trig_octant_aux: {
8725   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
8726   \fp_sub:NNNNNNNNN
8727   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8728   \l_fp_input_a_extended_int
8729   \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
8730   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8731   \l_fp_input_a_extended_int
8732   \tex_advance:D \l_fp_trig_octant_int \c_one
8733   \tex_expandafter:D \fp_trig_octant_aux:
8734 \tex_else:D
8735   \tex_ifnum:D
8736   \l_fp_input_a_decimal_int > \c_fp_pi_by_four_decimal_int
8737   \fp_sub:NNNNNNNNN
8738   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8739   \l_fp_input_a_extended_int
8740   \c_zero \c_fp_pi_by_four_decimal_int
8741   \c_fp_pi_by_four_extended_int
8742   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8743   \l_fp_input_a_extended_int
8744   \tex_advance:D \l_fp_trig_octant_int \c_one
8745   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8746   \fp_trig_octant_aux:
8747   \tex_fi:D
8748 \tex_fi:D
8749 }

```

`\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`  
`\fp_sin_aux:NNn`

```

8750 \cs_new_protected_nopar:Npn \fp_sin:Nn {
8751   \fp_sin_aux:NNn \tl_set:Nn
8752 }
8753 \cs_new_protected_nopar:Npn \fp_gsin:Nn {
8754   \fp_sin_aux:NNn \tl_gset:Nn
8755 }
8756 \cs_generate_variant:Nn \fp_sin:Nn { c }
8757 \cs_generate_variant:Nn \fp_gsin:Nn { c }

```

`\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  $1 \times 10^{-5}$ .

```

8758 \cs_new_protected_nopar:Npn \fp_sin_aux:NNn #1#2#3 {
8759   \group_begin:
8760     \fp_split:Nn a {#3}
8761     \fp_standardise:NNNN
8762     \l_fp_input_a_sign_int
8763     \l_fp_input_a_integer_int
8764     \l_fp_input_a_decimal_int
8765     \l_fp_input_a_exponent_int
8766     \tl_set:Nx \l_fp_arg_tl
8767     {
8768       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8769       -
8770       \tex_else:D
8771       +
8772       \tex_fi:D
8773       \int_use:N \l_fp_input_a_integer_int
8774       .
8775       \tex_expandafter:D \use_none:n
8776       \tex_number:D \etex_numexpr:D
8777       \l_fp_input_a_decimal_int + \c_one_thousand_million
8778       e
8779       \int_use:N \l_fp_input_a_exponent_int
8780     }
8781     \tex_ifnum:D \l_fp_input_a_exponent_int < -\c_five
8782     \cs_set_protected_nopar:Npx \fp_tmp:w
8783     {
8784       \group_end:
8785       #1 \exp_not:N #2 { \l_fp_arg_tl }
8786     }
8787     \tex_else:D
8788     \etex_ifcsname:D
8789     c_fp_sin ( \l_fp_arg_tl ) _fp
8790     \tex_endcsname:D
8791     \tex_else:D
8792     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8793     \fp_sin_aux_i:
8794     \tex_fi:D
8795     \cs_set_protected_nopar:Npx \fp_tmp:w
8796     {
8797       \group_end:
8798       #1 \exp_not:N #2
8799       { \use:c { c_fp_sin ( \l_fp_arg_tl ) _fp } }
8800     }
8801     \tex_fi:D
8802     \fp_tmp:w
8803 }

```

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.

```

8804 \cs_new_protected_nopar:Npn \fp_sin_aux_i: {
8805   \fp_trig_normalise:
8806   \fp_sin_aux_ii:
8807   \tex_ifnum:D \l_fp_output_integer_int = \c_one
8808     \l_fp_output_exponent_int \c_zero
8809   \tex_else:D
8810     \l_fp_output_integer_int \l_fp_output_decimal_int
8811     \l_fp_output_decimal_int \l_fp_output_extended_int
8812     \l_fp_output_exponent_int -\c_nine
8813   \tex_fi:D
8814   \fp_standardise:NNNN
8815     \l_fp_input_a_sign_int
8816     \l_fp_output_integer_int
8817     \l_fp_output_decimal_int
8818     \l_fp_output_exponent_int
8819   \tl_new:c { c_fp_sin ( \l_fp_arg_tl ) _fp }
8820   \tl_gset:cx { c_fp_sin ( \l_fp_arg_tl ) _fp }
8821   {
8822     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8823       +
8824     \tex_else:D
8825       -
8826     \tex_fi:D
8827     \int_use:N \l_fp_output_integer_int
8828     .
8829     \tex_expandafter:D \use_none:n
8830     \tex_number:D \etex_numexpr:D
8831       \l_fp_output_decimal_int + \c_one_thousand_million
8832     \scan_stop:
8833     e
8834     \int_use:N \l_fp_output_exponent_int
8835   }
8836 }
8837 \cs_new_protected_nopar:Npn \fp_sin_aux_ii: {
8838   \tex_ifcase:D \l_fp_trig_octant_int
8839   \tex_or:D
8840     \tex_expandafter:D \fp_trig_calc_sin:
8841   \tex_or:D
8842     \tex_expandafter:D \fp_trig_calc_cos:
8843   \tex_or:D
8844     \tex_expandafter:D \fp_trig_calc_cos:
8845   \tex_or:D
8846     \tex_expandafter:D \fp_trig_calc_sin:
8847   \tex_fi:D
8848 }

```

\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:
8849 \cs_new_protected_nopar:Npn \fp_cos:Nn {
8850   \fp_cos_aux:NNn \tl_set:Nn
8851 }
8852 \cs_new_protected_nopar:Npn \fp_gcos:Nn {
8853   \fp_cos_aux:NNn \tl_gset:Nn
8854 }
8855 \cs_generate_variant:Nn \fp_cos:Nn { c }
8856 \cs_generate_variant:Nn \fp_gcos:Nn { c }
8857 \cs_new_protected_nopar:Npn \fp_cos_aux:NNn #1#2#3 {
8858   \group_begin:
8859     \fp_split:Nn a {#3}
8860     \fp_standardise:NNNN
8861     \l_fp_input_a_sign_int
8862     \l_fp_input_a_integer_int
8863     \l_fp_input_a_decimal_int
8864     \l_fp_input_a_exponent_int
8865     \tl_set:Nx \l_fp_arg_tl
8866     {
8867       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8868       -
8869       \tex_else:D
8870       +
8871       \tex_fi:D
8872       \int_use:N \l_fp_input_a_integer_int
8873       .
8874       \tex_expandafter:D \use_none:n
8875       \tex_number:D \etex_numexpr:D
8876         \l_fp_input_a_decimal_int + \c_one_thousand_million
8877       e
8878       \int_use:N \l_fp_input_a_exponent_int
8879     }
8880     \etex_ifcsname:D c_fp_cos ( \l_fp_arg_tl ) _fp \tex_endcsname:D
8881     \tex_else:D
8882     \tex_expandafter:D \fp_cos_aux_i:
8883     \tex_fi:D
8884     \cs_set_protected_nopar:Npx \fp_tmp:w
8885     {
8886       \group_end:
8887       #1 \exp_not:N #2
8888       { \use:c { c_fp_cos ( \l_fp_arg_tl ) _fp } }
8889     }
8890     \fp_tmp:w
8891   }

```

Almost the same as for sine: just a bit of correction for the sign of the output.

```

8892 \cs_new_protected_nopar:Npn \fp_cos_aux_i: {
8893   \fp_trig_normalise:
8894   \fp_cos_aux_ii:

```

```

8895 \tex_ifnum:D \l_fp_output_integer_int = \c_one
8896 \l_fp_output_exponent_int \c_zero
8897 \tex_else:D
8898 \l_fp_output_integer_int \l_fp_output_decimal_int
8899 \l_fp_output_decimal_int \l_fp_output_extended_int
8900 \l_fp_output_exponent_int -\c_nine
8901 \tex_fi:D
8902 \fp_standardise:NNNN
8903 \l_fp_input_a_sign_int
8904 \l_fp_output_integer_int
8905 \l_fp_output_decimal_int
8906 \l_fp_output_exponent_int
8907 \tl_new:c { c_fp_cos ( \l_fp_arg_tl ) _fp }
8908 \tl_gset:cx { c_fp_cos ( \l_fp_arg_tl ) _fp }
8909 {
8910 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8911 +
8912 \tex_else:D
8913 -
8914 \tex_fi:D
8915 \int_use:N \l_fp_output_integer_int
8916 .
8917 \tex_expandafter:D \use_none:n
8918 \tex_number:D \etex_numexpr:D
8919 \l_fp_output_decimal_int + \c_one_thousand_million
8920 \scan_stop:
8921 e
8922 \int_use:N \l_fp_output_exponent_int
8923 }
8924 }
8925 \cs_new_protected_nopar:Npn \fp_cos_aux_ii: {
8926 \tex_ifcase:D \l_fp_trig_octant_int
8927 \tex_or:D
8928 \tex_expandafter:D \fp_trig_calc_cos:
8929 \tex_or:D
8930 \tex_expandafter:D \fp_trig_calc_sin:
8931 \tex_or:D
8932 \tex_expandafter:D \fp_trig_calc_sin:
8933 \tex_or:D
8934 \tex_expandafter:D \fp_trig_calc_cos:
8935 \tex_fi:D
8936 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8937 \tex_ifnum:D \l_fp_trig_octant_int > \c_two
8938 \l_fp_input_a_sign_int \c_minus_one
8939 \tex_fi:D
8940 \tex_else:D
8941 \tex_ifnum:D \l_fp_trig_octant_int > \c_two
8942 \tex_else:D
8943 \l_fp_input_a_sign_int \c_one
8944 \tex_fi:D

```

```

8945 \tex_fi:D
8946 }

```

```

\fp_trig_calc_cos:
\fp_trig_calc_sin:
\fp_trig_calc_Taylor:

```

These functions actually do the calculation for sine and cosine.

```

8947 \cs_new_protected_nopar:Npn \fp_trig_calc_cos: {
8948   \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
8949     \l_fp_output_integer_int \c_one
8950     \l_fp_output_decimal_int \c_zero
8951   \tex_else:D
8952     \l_fp_trig_sign_int \c_minus_one
8953   \fp_mul:NNNNNN
8954     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
8955     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
8956     \l_fp_trig_decimal_int \l_fp_trig_extended_int
8957   \fp_div_integer:NNNNN
8958     \l_fp_trig_decimal_int \l_fp_trig_extended_int
8959     \c_two
8960     \l_fp_trig_decimal_int \l_fp_trig_extended_int
8961   \l_fp_count_int \c_three
8962   \tex_ifnum:D \l_fp_trig_extended_int = \c_zero
8963     \tex_ifnum:D \l_fp_trig_decimal_int = \c_zero
8964       \l_fp_output_integer_int \c_one
8965       \l_fp_output_decimal_int \c_zero
8966       \l_fp_output_extended_int \c_zero
8967     \tex_else:D
8968       \l_fp_output_integer_int \c_zero
8969       \l_fp_output_decimal_int \c_one_thousand_million
8970       \l_fp_output_extended_int \c_zero
8971     \tex_fi:D
8972   \tex_else:D
8973     \l_fp_output_integer_int \c_zero
8974     \l_fp_output_decimal_int 999999999 \scan_stop:
8975     \l_fp_output_extended_int \c_one_thousand_million
8976   \tex_fi:D
8977   \tex_advance:D \l_fp_output_extended_int -\l_fp_trig_extended_int
8978   \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
8979   \tex_expandafter:D \fp_trig_calc_Taylor:
8980 \tex_fi:D
8981 }
8982 \cs_new_protected_nopar:Npn \fp_trig_calc_sin: {
8983   \l_fp_output_integer_int \c_zero
8984   \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
8985     \l_fp_output_decimal_int \c_zero
8986   \tex_else:D
8987     \l_fp_output_decimal_int \l_fp_input_a_decimal_int
8988     \l_fp_output_extended_int \l_fp_input_a_extended_int
8989     \l_fp_trig_sign_int \c_one
8990     \l_fp_trig_decimal_int \l_fp_input_a_decimal_int
8991     \l_fp_trig_extended_int \l_fp_input_a_extended_int

```

```

8992     \l_fp_count_int \c_two
8993     \tex_expandafter:D \fp_trig_calc_Taylor:
8994     \tex_fi:D
8995 }

```

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.

```

8996 \cs_new_protected_nopar:Npn \fp_trig_calc_Taylor: {
8997   \l_fp_trig_sign_int -\l_fp_trig_sign_int
8998   \fp_mul:NNNNNN
8999   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9000   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9001   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9002   \fp_mul:NNNNNN
9003   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9004   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9005   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9006   \fp_div_integer:NNNNN
9007   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9008   \l_fp_count_int
9009   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9010   \tex_advance:D \l_fp_count_int \c_one
9011   \fp_div_integer:NNNNN
9012   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9013   \l_fp_count_int
9014   \l_fp_trig_decimal_int \l_fp_trig_extended_int
9015   \tex_advance:D \l_fp_count_int \c_one
9016   \tex_ifnum:D \l_fp_trig_decimal_int > \c_zero
9017   \tex_ifnum:D \l_fp_trig_sign_int > \c_zero
9018   \tex_advance:D \l_fp_output_decimal_int \l_fp_trig_decimal_int
9019   \tex_advance:D \l_fp_output_extended_int
9020   \l_fp_trig_extended_int
9021   \tex_ifnum:D \l_fp_output_extended_int < \c_one_thousand_million
9022   \tex_else:D
9023     \tex_advance:D \l_fp_output_decimal_int \c_one
9024     \tex_advance:D \l_fp_output_extended_int
9025     -\c_one_thousand_million
9026   \tex_fi:D
9027   \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9028   \tex_else:D
9029     \tex_advance:D \l_fp_output_integer_int \c_one
9030     \tex_advance:D \l_fp_output_decimal_int
9031     -\c_one_thousand_million
9032   \tex_fi:D
9033   \tex_else:D
9034     \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9035     \tex_advance:D \l_fp_output_extended_int
9036     -\l_fp_input_a_extended_int
9037     \tex_ifnum:D \l_fp_output_extended_int < \c_zero

```

```

9038      \tex_advance:D \l_fp_output_decimal_int \c_minus_one
9039      \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
9040      \tex_fi:D
9041      \tex_ifnum:D \l_fp_output_decimal_int < \c_zero
9042      \tex_advance:D \l_fp_output_integer_int \c_minus_one
9043      \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9044      \tex_fi:D
9045      \tex_fi:D
9046      \tex_expandafter:D \fp_trig_calc_Taylor:
9047      \tex_fi:D
9048  }

```

`\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:
9049  \cs_new_protected_nopar:Npn \fp_tan:Nn {
9050    \fp_tan_aux:NNn \tl_set:Nn
9051  }
9052  \cs_new_protected_nopar:Npn \fp_gtan:Nn {
9053    \fp_tan_aux:NNn \tl_gset:Nn
9054  }
9055  \cs_generate_variant:Nn \fp_tan:Nn { c }
9056  \cs_generate_variant:Nn \fp_gtan:Nn { c }
9057  \cs_new_protected_nopar:Npn \fp_tan_aux:NNn #1#2#3 {
9058    \group_begin:
9059      \fp_split:Nn a {#3}
9060      \fp_standardise:NNNN
9061      \l_fp_input_a_sign_int
9062      \l_fp_input_a_integer_int
9063      \l_fp_input_a_decimal_int
9064      \l_fp_input_a_exponent_int
9065      \tl_set:Nx \l_fp_arg_tl
9066      {
9067        \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9068        -
9069        \tex_else:D
9070        +
9071        \tex_fi:D
9072        \int_use:N \l_fp_input_a_integer_int
9073        .
9074        \tex_expandafter:D \use_none:n
9075        \tex_number:D \etex_numexpr:D
9076          \l_fp_input_a_decimal_int + \c_one_thousand_million
9077        e
9078        \int_use:N \l_fp_input_a_exponent_int
9079      }
9080      \tex_ifnum:D \l_fp_input_a_exponent_int < -\c_five
9081      \cs_set_protected_nopar:Npx \fp_tmp:w

```

```

9082     {
9083         \group_end:
9084         #1 \exp_not:N #2 { \l_fp_arg_tl }
9085     }
9086 \tex_else:D
9087 \etex_ifcsname:D
9088   c_fp_tan ( \l_fp_arg_tl ) _fp
9089 \tex_endcsname:D
9090 \tex_else:D
9091   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9092   \fp_tan_aux_i:
9093 \tex_fi:D
9094 \cs_set_protected_nopar:Npx \fp_tmp:w
9095 {
9096   \group_end:
9097   #1 \exp_not:N #2
9098   { \use:c { c_fp_tan ( \l_fp_arg_tl ) _fp } }
9099 }
9100 \tex_fi:D
9101 \fp_tmp:w
9102 }

```

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.

```

9103 \cs_new_protected_nopar:Npn \fp_tan_aux_i: {
9104   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
9105   \tex_expandafter:D \fp_tan_aux_ii:
9106 \tex_else:D
9107   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9108   \c_zero_fp
9109   \tex_expandafter:D \fp_trig_overflow_msg:
9110 \tex_fi:D
9111 }
9112 \cs_new_protected_nopar:Npn \fp_tan_aux_ii: {
9113   \fp_trig_normalise:
9114   \fp_cos_aux_ii:
9115   \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9116   \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
9117   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9118   \c_undefined_fp
9119 \tex_else:D
9120   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9121   \fp_tan_aux_iii:
9122 \tex_fi:D
9123 \tex_else:D
9124   \tex_expandafter:D \fp_tan_aux_iii:
9125 \tex_fi:D

```

9126 }

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.

```

9127 \cs_new_protected_nopar:Npn \fp_tan_aux_iii: {
9128   \l_fp_input_b_integer_int \l_fp_output_decimal_int
9129   \l_fp_input_b_decimal_int \l_fp_output_extended_int
9130   \l_fp_input_b_exponent_int -\c_nine
9131   \fp_standardise:NNNN
9132   \l_fp_input_b_sign_int
9133   \l_fp_input_b_integer_int
9134   \l_fp_input_b_decimal_int
9135   \l_fp_input_b_exponent_int
9136   \fp_sin_aux_ii:
9137   \l_fp_input_a_integer_int \l_fp_output_decimal_int
9138   \l_fp_input_a_decimal_int \l_fp_output_extended_int
9139   \l_fp_input_a_exponent_int -\c_nine
9140   \fp_standardise:NNNN
9141   \l_fp_input_a_sign_int
9142   \l_fp_input_a_integer_int
9143   \l_fp_input_a_decimal_int
9144   \l_fp_input_a_exponent_int
9145   \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9146   \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
9147   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9148   \c_zero_fp
9149   \tex_else:D
9150   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9151   \fp_tan_aux_iv:
9152   \tex_fi:D
9153   \tex_else:D
9154   \tex_expandafter:D \fp_tan_aux_iv:
9155   \tex_fi:D
9156 }
9157 \cs_new_protected_nopar:Npn \fp_tan_aux_iv: {
9158   \l_fp_output_integer_int \c_zero
9159   \l_fp_output_decimal_int \c_zero
9160   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
9161   \l_fp_div_offset_int \c_one_hundred_million
9162   \fp_div_loop:
9163   \l_fp_output_exponent_int
9164   \etex_numexpr:D
9165     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
9166   \scan_stop:
9167   \tex_ifnum:D \l_fp_trig_octant_int < \c_three
9168   \l_fp_output_sign_int \c_one
9169   \tex_else:D
9170   \l_fp_output_sign_int \c_minus_one
9171   \tex_fi:D

```

```

9172 \fp_standardise:NNNN
9173   \l_fp_output_sign_int
9174   \l_fp_output_integer_int
9175   \l_fp_output_decimal_int
9176   \l_fp_output_exponent_int
9177 \tl_new:c { c_fp_tan ( \l_fp_arg_tl ) _fp }
9178 \tl_gset:cx { c_fp_tan ( \l_fp_arg_tl ) _fp }
9179 {
9180   \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9181   +
9182   \tex_else:D
9183   -
9184   \tex_fi:D
9185   \int_use:N \l_fp_output_integer_int
9186   .
9187   \tex_expandafter:D \use_none:n
9188   \tex_number:D \etex_numexpr:D
9189   \l_fp_output_decimal_int + \c_one_thousand_million
9190   \scan_stop:
9191   e
9192   \int_use:N \l_fp_output_exponent_int
9193 }
9194 }

```

## 116.12 Exponent and logarithm functions

Calculation of exponentials requires a number of precomputed values: first the positive integers.

```

\c_fp_exp_1_tl
\c_fp_exp_2_tl
\c_fp_exp_3_tl
\c_fp_exp_4_tl
\c_fp_exp_5_tl
\c_fp_exp_6_tl
\c_fp_exp_7_tl
\c_fp_exp_8_tl
\c_fp_exp_9_tl
\c_fp_exp_10_tl
\c_fp_exp_20_tl
\c_fp_exp_30_tl
\c_fp_exp_40_tl
\c_fp_exp_50_tl
\c_fp_exp_60_tl
\c_fp_exp_70_tl
\c_fp_exp_80_tl
\c_fp_exp_90_tl
\c_fp_exp_100_tl
\c_fp_exp_200_tl

```

```

9195 \tl_new:c { c_fp_exp_1_tl }
9196 \tl_set:cn { c_fp_exp_1_tl }
9197 { { 2 } { 718281828 } { 459045235 } { 0 } }
9198 \tl_new:c { c_fp_exp_2_tl }
9199 \tl_set:cn { c_fp_exp_2_tl }
9200 { { 7 } { 389056098 } { 930650227 } { 0 } }
9201 \tl_new:c { c_fp_exp_3_tl }
9202 \tl_set:cn { c_fp_exp_3_tl }
9203 { { 2 } { 008553692 } { 318766774 } { 1 } }
9204 \tl_new:c { c_fp_exp_4_tl }
9205 \tl_set:cn { c_fp_exp_4_tl }
9206 { { 5 } { 459815003 } { 314423908 } { 1 } }
9207 \tl_new:c { c_fp_exp_5_tl }
9208 \tl_set:cn { c_fp_exp_5_tl }
9209 { { 1 } { 484131591 } { 025766034 } { 2 } }
9210 \tl_new:c { c_fp_exp_6_tl }
9211 \tl_set:cn { c_fp_exp_6_tl }
9212 { { 4 } { 034287934 } { 927351226 } { 2 } }
9213 \tl_new:c { c_fp_exp_7_tl }
9214 \tl_set:cn { c_fp_exp_7_tl }

```

```

9215 { { 1 } { 096633158 } { 428458599 } { 3 } }
9216 \tl_new:c { c_fp_exp_8_tl }
9217 \tl_set:cn { c_fp_exp_8_tl }
9218 { { 2 } { 980957987 } { 041728275 } { 3 } }
9219 \tl_new:c { c_fp_exp_9_tl }
9220 \tl_set:cn { c_fp_exp_9_tl }
9221 { { 8 } { 103083927 } { 575384008 } { 3 } }
9222 \tl_new:c { c_fp_exp_10_tl }
9223 \tl_set:cn { c_fp_exp_10_tl }
9224 { { 2 } { 202646579 } { 480671652 } { 4 } }
9225 \tl_new:c { c_fp_exp_20_tl }
9226 \tl_set:cn { c_fp_exp_20_tl }
9227 { { 4 } { 851651954 } { 097902280 } { 8 } }
9228 \tl_new:c { c_fp_exp_30_tl }
9229 \tl_set:cn { c_fp_exp_30_tl }
9230 { { 1 } { 068647458 } { 152446215 } { 13 } }
9231 \tl_new:c { c_fp_exp_40_tl }
9232 \tl_set:cn { c_fp_exp_40_tl }
9233 { { 2 } { 353852668 } { 370199854 } { 17 } }
9234 \tl_new:c { c_fp_exp_50_tl }
9235 \tl_set:cn { c_fp_exp_50_tl }
9236 { { 5 } { 184705528 } { 587072464 } { 21 } }
9237 \tl_new:c { c_fp_exp_60_tl }
9238 \tl_set:cn { c_fp_exp_60_tl }
9239 { { 1 } { 142007389 } { 815684284 } { 26 } }
9240 \tl_new:c { c_fp_exp_70_tl }
9241 \tl_set:cn { c_fp_exp_70_tl }
9242 { { 2 } { 515438670 } { 919167006 } { 30 } }
9243 \tl_new:c { c_fp_exp_80_tl }
9244 \tl_set:cn { c_fp_exp_80_tl }
9245 { { 5 } { 540622384 } { 393510053 } { 34 } }
9246 \tl_new:c { c_fp_exp_90_tl }
9247 \tl_set:cn { c_fp_exp_90_tl }
9248 { { 1 } { 220403294 } { 317840802 } { 39 } }
9249 \tl_new:c { c_fp_exp_100_tl }
9250 \tl_set:cn { c_fp_exp_100_tl }
9251 { { 2 } { 688117141 } { 816135448 } { 43 } }
9252 \tl_new:c { c_fp_exp_200_tl }
9253 \tl_set:cn { c_fp_exp_200_tl }
9254 { { 7 } { 225973768 } { 125749258 } { 86 } }

```

\c\_fp\_exp\_-1\_tl Now the negative integers.

```

\c_fp_exp_-2_tl
\c_fp_exp_-3_tl
\c_fp_exp_-4_tl
\c_fp_exp_-5_tl
\c_fp_exp_-6_tl
\c_fp_exp_-7_tl
\c_fp_exp_-8_tl
\c_fp_exp_-9_tl
\c_fp_exp_-10_tl
\c_fp_exp_-20_tl
\c_fp_exp_-30_tl
\c_fp_exp_-40_tl
\c_fp_exp_-50_tl
\c_fp_exp_-60_tl
\c_fp_exp_-70_tl
\c_fp_exp_-80_tl
\c_fp_exp_-90_tl

```

```

9255 \tl_new:c { c_fp_exp_-1_tl }
9256 \tl_set:cn { c_fp_exp_-1_tl }
9257 { { 3 } { 678794411 } { 71442322 } { -1 } }
9258 \tl_new:c { c_fp_exp_-2_tl }
9259 \tl_set:cn { c_fp_exp_-2_tl }
9260 { { 1 } { 353352832 } { 366132692 } { -1 } }
9261 \tl_new:c { c_fp_exp_-3_tl }

```

```

9262 \tl_set:cn { c_fp_exp_-3_tl }
9263   { { 4 } { 978706836 } { 786394298 } { -2 } }
9264 \tl_new:c { c_fp_exp_-4_tl }
9265 \tl_set:cn { c_fp_exp_-4_tl }
9266   { { 1 } { 831563888 } { 873418029 } { -2 } }
9267 \tl_new:c { c_fp_exp_-5_tl }
9268 \tl_set:cn { c_fp_exp_-5_tl }
9269   { { 6 } { 737946999 } { 085467097 } { -3 } }
9270 \tl_new:c { c_fp_exp_-6_tl }
9271 \tl_set:cn { c_fp_exp_-6_tl }
9272   { { 2 } { 478752176 } { 666358423 } { -3 } }
9273 \tl_new:c { c_fp_exp_-7_tl }
9274 \tl_set:cn { c_fp_exp_-7_tl }
9275   { { 9 } { 118819655 } { 545162080 } { -4 } }
9276 \tl_new:c { c_fp_exp_-8_tl }
9277 \tl_set:cn { c_fp_exp_-8_tl }
9278   { { 3 } { 354626279 } { 025118388 } { -4 } }
9279 \tl_new:c { c_fp_exp_-9_tl }
9280 \tl_set:cn { c_fp_exp_-9_tl }
9281   { { 1 } { 234098040 } { 866795495 } { -4 } }
9282 \tl_new:c { c_fp_exp_-10_tl }
9283 \tl_set:cn { c_fp_exp_-10_tl }
9284   { { 4 } { 539992976 } { 248451536 } { -5 } }
9285 \tl_new:c { c_fp_exp_-20_tl }
9286 \tl_set:cn { c_fp_exp_-20_tl }
9287   { { 2 } { 061153622 } { 438557828 } { -9 } }
9288 \tl_new:c { c_fp_exp_-30_tl }
9289 \tl_set:cn { c_fp_exp_-30_tl }
9290   { { 9 } { 357622968 } { 840174605 } { -14 } }
9291 \tl_new:c { c_fp_exp_-40_tl }
9292 \tl_set:cn { c_fp_exp_-40_tl }
9293   { { 4 } { 248354255 } { 291588995 } { -18 } }
9294 \tl_new:c { c_fp_exp_-50_tl }
9295 \tl_set:cn { c_fp_exp_-50_tl }
9296   { { 1 } { 928749847 } { 963917783 } { -22 } }
9297 \tl_new:c { c_fp_exp_-60_tl }
9298 \tl_set:cn { c_fp_exp_-60_tl }
9299   { { 8 } { 756510762 } { 696520338 } { -27 } }
9300 \tl_new:c { c_fp_exp_-70_tl }
9301 \tl_set:cn { c_fp_exp_-70_tl }
9302   { { 3 } { 975449735 } { 908646808 } { -31 } }
9303 \tl_new:c { c_fp_exp_-80_tl }
9304 \tl_set:cn { c_fp_exp_-80_tl }
9305   { { 1 } { 804851387 } { 845415172 } { -35 } }
9306 \tl_new:c { c_fp_exp_-90_tl }
9307 \tl_set:cn { c_fp_exp_-90_tl }
9308   { { 8 } { 194012623 } { 990515430 } { -40 } }
9309 \tl_new:c { c_fp_exp_-100_tl }
9310 \tl_set:cn { c_fp_exp_-100_tl }
9311   { { 3 } { 720075976 } { 020835963 } { -44 } }

```

```

9312 \tl_new:c { c_fp_exp_-200_tl }
9313 \tl_set:cn { c_fp_exp_-200_tl }
9314 { { 1 } { 383896526 } { 736737530 } { -87 } }

```

`\fp_exp:Nn` The calculation of an exponent starts off starts in much the same way as the trigonometric  
`\fp_exp:cn` functions: normalise the input, look for a pre-defined value and if one is not found hand  
`\fp_gexp:Nn` off to the real workhorse function. The test for a definition of the result is used so that  
`\fp_gexp:cn` overflows do not result in any outcome being defined.

```

\fp_exp_aux:NNn
\fp_exp_internal:
\fp_exp_aux:
\fp_exp_integer:
\fp_exp_integer_tens:
\fp_exp_integer_units:
\fp_exp_integer_const:n
\fp_exp_integer_const:nnnn
\fp_exp_decimal:
\fp_exp_Taylor:
\fp_exp_const:Nx
\fp_exp_const:cx
9315 \cs_new_protected_nopar:Npn \fp_exp:Nn {
9316   \fp_exp_aux:NNn \tl_set:Nn
9317 }
9318 \cs_new_protected_nopar:Npn \fp_gexp:Nn {
9319   \fp_exp_aux:NNn \tl_gset:Nn
9320 }
9321 \cs_generate_variant:Nn \fp_exp:Nn { c }
9322 \cs_generate_variant:Nn \fp_gexp:Nn { c }
9323 \cs_new_protected_nopar:Npn \fp_exp_aux:NNn #1#2#3 {
9324   \group_begin:
9325     \fp_split:Nn a {#3}
9326     \fp_standardise:NNNN
9327     \l_fp_input_a_sign_int
9328     \l_fp_input_a_integer_int
9329     \l_fp_input_a_decimal_int
9330     \l_fp_input_a_exponent_int
9331     \l_fp_input_a_extended_int \c_zero
9332     \tl_set:Nx \l_fp_arg_tl
9333     {
9334       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9335       -
9336       \tex_else:D
9337       +
9338       \tex_fi:D
9339       \int_use:N \l_fp_input_a_integer_int
9340       .
9341       \tex_expandafter:D \use_none:n
9342       \tex_number:D \etex_numexpr:D
9343       \l_fp_input_a_decimal_int + \c_one_thousand_million
9344       e
9345       \int_use:N \l_fp_input_a_exponent_int
9346     }
9347     \etex_ifcsname:D c_fp_exp ( \l_fp_arg_tl ) _fp \tex_endcsname:D
9348     \tex_else:D
9349     \tex_expandafter:D \fp_exp_internal:
9350     \tex_fi:D
9351     \cs_set_protected_nopar:Npx \fp_tmp:w
9352     {
9353       \group_end:
9354       #1 \exp_not:N #2
9355       {

```

```

9356         \etex_ifcsname:D c_fp_exp ( \l_fp_arg_tl ) _fp
9357         \tex_endcsname:D
9358         \use:c { c_fp_exp ( \l_fp_arg_tl ) _fp }
9359     \tex_else:D
9360         \c_zero_fp
9361     \tex_fi:D
9362 }
9363 }
9364 \fp_tmp:w
9365 }

```

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

```

9366 \cs_new_protected_nopar:Npn \fp_exp_internal: {
9367     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_three
9368     \fp_extended_normalise:
9369     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9370     \tex_ifnum:D \l_fp_input_a_integer_int < 230 \scan_stop:
9371     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9372     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9373     \tex_expandafter:D \fp_exp_aux:
9374 \tex_else:D
9375     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9376     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9377     \tex_expandafter:D \fp_exp_overflow_msg:
9378     \tex_fi:D
9379 \tex_else:D
9380     \tex_ifnum:D \l_fp_input_a_integer_int < 230 \scan_stop:
9381     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9382     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9383     \tex_expandafter:D \fp_exp_aux:
9384 \tex_else:D
9385     \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9386     { \c_zero_fp }
9387     \tex_fi:D
9388 \tex_fi:D
9389 \tex_else:D
9390     \tex_expandafter:D \fp_exp_overflow_msg:
9391 \tex_fi:D
9392 }

```

The main algorithm makes use of the fact that

$$e^{nmp.q} = e^n e^m e^p e^{0.q}$$

and that there is a Taylor series that can be used to calculate  $e^{0.q}$ . Thus the approach needed is in three parts. First, the exponent of the integer part of the input is found using the pre-calculated constants. Second, the Taylor series is used to find the exponent

for the decimal part of the input. Finally, the two parts are multiplied together to give the result. As the normalisation code will already have dealt with any overflowing values, there are no further checks needed.

```

9393 \cs_new_protected_nopar:Npn \fp_exp_aux: {
9394   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
9395     \tex_expandafter:D \fp_exp_integer:
9396   \tex_else:D
9397     \l_fp_output_integer_int \c_one
9398     \l_fp_output_decimal_int \c_zero
9399     \l_fp_output_extended_int \c_zero
9400     \l_fp_output_exponent_int \c_zero
9401     \tex_expandafter:D \fp_exp_decimal:
9402   \tex_fi:D
9403 }

```

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).

```

9404 \cs_new_protected_nopar:Npn \fp_exp_integer: {
9405   \tex_ifnum:D \l_fp_input_a_integer_int < \c_one_hundred
9406     \l_fp_exp_integer_int \c_one
9407     \l_fp_exp_decimal_int \c_zero
9408     \l_fp_exp_extended_int \c_zero
9409     \l_fp_exp_exponent_int \c_zero
9410     \tex_expandafter:D \fp_exp_integer_tens:
9411   \tex_else:D
9412     \tl_set:Nx \l_fp_tmp_tl
9413     {
9414       \tex_expandafter:D \use_i:nnn
9415       \int_use:N \l_fp_input_a_integer_int
9416     }
9417     \l_fp_input_a_integer_int
9418     \etex_numexpr:D
9419       \l_fp_input_a_integer_int - \l_fp_tmp_tl 00
9420     \scan_stop:
9421   \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9422     \tex_ifnum:D \l_fp_output_integer_int > 200 \scan_stop:
9423     \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9424     { \c_zero_fp }
9425   \tex_else:D
9426     \fp_exp_integer_const:n { - \l_fp_tmp_tl 00 }
9427     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9428     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9429     \tex_expandafter:D \fp_exp_integer_tens:
9430   \tex_fi:D
9431   \tex_else:D
9432     \fp_exp_integer_const:n { \l_fp_tmp_tl 00 }

```

```

9433     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9434     \tex_expandafter:D \fp_exp_integer_tens:
9435     \tex_fi:D
9436 \tex_fi:D
9437 }

```

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.

```

9438 \cs_new_protected_nopar:Npn \fp_exp_integer_tens: {
9439   \l_fp_output_integer_int \l_fp_exp_integer_int
9440   \l_fp_output_decimal_int \l_fp_exp_decimal_int
9441   \l_fp_output_extended_int \l_fp_exp_extended_int
9442   \l_fp_output_exponent_int \l_fp_exp_exponent_int
9443   \tex_ifnum:D \l_fp_input_a_integer_int > \c_nine
9444   \tl_set:Nx \l_fp_tmp_tl
9445   {
9446     \tex_expandafter:D \use_i:nn
9447     \int_use:N \l_fp_input_a_integer_int
9448   }
9449   \l_fp_input_a_integer_int
9450   \etex_numexpr:D
9451   \l_fp_input_a_integer_int - \l_fp_tmp_tl 0
9452   \scan_stop:
9453   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9454   \fp_exp_integer_const:n { \l_fp_tmp_tl 0 }
9455   \tex_else:D
9456   \fp_exp_integer_const:n { - \l_fp_tmp_tl 0 }
9457   \tex_fi:D
9458   \fp_mul:NNNNNNNN
9459   \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9460   \l_fp_output_integer_int \l_fp_output_decimal_int
9461   \l_fp_output_extended_int
9462   \l_fp_output_integer_int \l_fp_output_decimal_int
9463   \l_fp_output_extended_int
9464   \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
9465   \fp_extended_normalise_output:
9466   \tex_fi:D
9467   \fp_exp_integer_units:
9468 }
9469 \cs_new_protected_nopar:Npn \fp_exp_integer_units: {
9470   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
9471   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9472   \fp_exp_integer_const:n { \int_use:N \l_fp_input_a_integer_int }
9473   \tex_else:D
9474   \fp_exp_integer_const:n
9475   { - \int_use:N \l_fp_input_a_integer_int }
9476   \tex_fi:D
9477   \fp_mul:NNNNNNNN

```

```

9478 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9479 \l_fp_output_integer_int \l_fp_output_decimal_int
9480 \l_fp_output_extended_int
9481 \l_fp_output_integer_int \l_fp_output_decimal_int
9482 \l_fp_output_extended_int
9483 \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
9484 \fp_extended_normalise_output:
9485 \tex_fi:D
9486 \fp_exp_decimal:
9487 }

```

Recovery of the stored constant values into the separate registers is done with a simple expansion then assignment.

```

9488 \cs_new_protected_nopar:Npn \fp_exp_integer_const:n #1 {
9489 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9490 \fp_exp_integer_const:nnnn
9491 \tex_csname:D c_fp_exp_ #1 _tl \tex_endcsname:D
9492 }
9493 \cs_new_protected_nopar:Npn \fp_exp_integer_const:nnnn #1#2#3#4 {
9494 \l_fp_exp_integer_int #1 \scan_stop:
9495 \l_fp_exp_decimal_int #2 \scan_stop:
9496 \l_fp_exp_extended_int #3 \scan_stop:
9497 \l_fp_exp_exponent_int #4 \scan_stop:
9498 }

```

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.

```

9499 \cs_new_protected_nopar:Npn \fp_exp_decimal: {
9500 \tex_ifnum:D \l_fp_input_a_decimal_int > \c_zero
9501 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9502 \l_fp_exp_integer_int \c_one
9503 \l_fp_exp_decimal_int \l_fp_input_a_decimal_int
9504 \l_fp_exp_extended_int \l_fp_input_a_extended_int
9505 \tex_else:D
9506 \l_fp_exp_integer_int \c_zero
9507 \tex_ifnum:D \l_fp_exp_extended_int = \c_zero
9508 \l_fp_exp_decimal_int
9509 \etex_numexpr:D
9510 \c_one_thousand_million - \l_fp_input_a_decimal_int
9511 \scan_stop:
9512 \l_fp_exp_extended_int \c_zero
9513 \tex_else:D
9514 \l_fp_exp_decimal_int
9515 \etex_numexpr:D
9516 999999999 - \l_fp_input_a_decimal_int
9517 \scan_stop:

```

```

9518         \l_fp_exp_extended_int
9519         \etex_numexpr:D
9520             \c_one_thousand_million - \l_fp_input_a_extended_int
9521         \scan_stop:
9522     \tex_fi:D
9523 \tex_fi:D
9524 \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
9525 \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
9526 \l_fp_input_b_extended_int  \l_fp_input_a_extended_int
9527 \l_fp_count_int \c_one
9528 \fp_exp_Taylor:
9529 \fp_mul:NNNNNNNNN
9530     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9531     \l_fp_output_integer_int \l_fp_output_decimal_int
9532     \l_fp_output_extended_int
9533     \l_fp_output_integer_int \l_fp_output_decimal_int
9534     \l_fp_output_extended_int
9535 \tex_fi:D
9536 \tex_ifnum:D \l_fp_output_extended_int < \c_five_hundred_million
9537 \tex_else:D
9538     \tex_advance:D \l_fp_output_decimal_int \c_one
9539     \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9540     \tex_else:D
9541         \l_fp_output_decimal_int \c_zero
9542         \tex_advance:D \l_fp_output_integer_int \c_one
9543     \tex_fi:D
9544 \tex_fi:D
9545 \fp_standardise:NNNN
9546     \l_fp_output_sign_int
9547     \l_fp_output_integer_int
9548     \l_fp_output_decimal_int
9549     \l_fp_output_exponent_int
9550 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9551 {
9552     +
9553     \int_use:N \l_fp_output_integer_int
9554     .
9555     \tex_expandafter:D \use_none:n
9556         \tex_number:D \etex_numexpr:D
9557             \l_fp_output_decimal_int + \c_one_thousand_million
9558     \scan_stop:
9559     e
9560     \int_use:N \l_fp_output_exponent_int
9561 }
9562 }

```

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`.

```

9563 \cs_new_protected_nopar:Npn \fp_exp_Taylor: {
9564   \tex_advance:D \l_fp_count_int \c_one
9565   \tex_multiply:D \l_fp_input_b_sign_int \l_fp_input_a_sign_int
9566   \fp_mul:NNNNNN
9567     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9568     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9569     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9570   \fp_div_integer:NNNNN
9571     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9572     \l_fp_count_int
9573     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9574   \tex_ifnum:D
9575     \etex_numexpr:D
9576       \l_fp_input_b_decimal_int + \l_fp_input_b_extended_int
9577       > \c_zero
9578   \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
9579     \tex_advance:D \l_fp_exp_decimal_int \l_fp_input_b_decimal_int
9580     \tex_advance:D \l_fp_exp_extended_int
9581       \l_fp_input_b_extended_int
9582   \tex_ifnum:D \l_fp_exp_extended_int < \c_one_thousand_million
9583   \tex_else:D
9584     \tex_advance:D \l_fp_exp_decimal_int \c_one
9585     \tex_advance:D \l_fp_exp_extended_int
9586       -\c_one_thousand_million
9587   \tex_fi:D
9588   \tex_ifnum:D \l_fp_exp_decimal_int < \c_one_thousand_million
9589   \tex_else:D
9590     \tex_advance:D \l_fp_exp_integer_int \c_one
9591     \tex_advance:D \l_fp_exp_decimal_int
9592       -\c_one_thousand_million
9593   \tex_fi:D
9594   \tex_else:D
9595     \tex_advance:D \l_fp_exp_decimal_int -\l_fp_input_b_decimal_int
9596     \tex_advance:D \l_fp_exp_extended_int
9597       -\l_fp_input_a_extended_int
9598   \tex_ifnum:D \l_fp_exp_extended_int < \c_zero
9599     \tex_advance:D \l_fp_exp_decimal_int \c_minus_one
9600     \tex_advance:D \l_fp_exp_extended_int \c_one_thousand_million
9601   \tex_fi:D
9602   \tex_ifnum:D \l_fp_exp_decimal_int < \c_zero
9603     \tex_advance:D \l_fp_exp_integer_int \c_minus_one
9604     \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
9605   \tex_fi:D
9606   \tex_fi:D
9607   \tex_expandafter:D \fp_exp_Taylor:

```

```

9608 \tex_fi:D
9609 }

```

This is set up as a function so that the power code can redirect the effect.

```

9610 \cs_new_protected_nopar:Npn \fp_exp_const:Nx #1#2 {
9611   \tl_new:N #1
9612   \tl_gset:Nx #1 {#2}
9613 }
9614 \cs_generate_variant:Nn \fp_exp_const:Nx { c }

```

\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
\c_fp_ln_10_4_tl
\c_fp_ln_10_5_tl
\c_fp_ln_10_6_tl
\c_fp_ln_10_7_tl
\c_fp_ln_10_8_tl
\c_fp_ln_10_9_tl
9615 \tl_new:c { c_fp_ln_10_1_tl }
9616 \tl_set:cn { c_fp_ln_10_1_tl }
9617   { { 2 } { 302585092 } { 994045684 } { 0 } }
9618 \tl_new:c { c_fp_ln_10_2_tl }
9619 \tl_set:cn { c_fp_ln_10_2_tl }
9620   { { 4 } { 605170185 } { 988091368 } { 0 } }
9621 \tl_new:c { c_fp_ln_10_3_tl }
9622 \tl_set:cn { c_fp_ln_10_3_tl }
9623   { { 6 } { 907755278 } { 982137052 } { 0 } }
9624 \tl_new:c { c_fp_ln_10_4_tl }
9625 \tl_set:cn { c_fp_ln_10_4_tl }
9626   { { 9 } { 210340371 } { 976182736 } { 0 } }
9627 \tl_new:c { c_fp_ln_10_5_tl }
9628 \tl_set:cn { c_fp_ln_10_5_tl }
9629   { { 1 } { 151292546 } { 497022842 } { 1 } }
9630 \tl_new:c { c_fp_ln_10_6_tl }
9631 \tl_set:cn { c_fp_ln_10_6_tl }
9632   { { 1 } { 381551055 } { 796427410 } { 1 } }
9633 \tl_new:c { c_fp_ln_10_7_tl }
9634 \tl_set:cn { c_fp_ln_10_7_tl }
9635   { { 1 } { 611809565 } { 095831979 } { 1 } }
9636 \tl_new:c { c_fp_ln_10_8_tl }
9637 \tl_set:cn { c_fp_ln_10_8_tl }
9638   { { 1 } { 842068074 } { 395226547 } { 1 } }
9639 \tl_new:c { c_fp_ln_10_9_tl }
9640 \tl_set:cn { c_fp_ln_10_9_tl }
9641   { { 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
9642 \tl_new:c { c_fp_ln_2_1_tl }
9643 \tl_set:cn { c_fp_ln_2_1_tl }
9644   { { 0 } { 693147180 } { 559945309 } { 0 } }
9645 \tl_new:c { c_fp_ln_2_2_tl }
9646 \tl_set:cn { c_fp_ln_2_2_tl }
9647   { { 1 } { 386294361 } { 119890618 } { 0 } }
9648 \tl_new:c { c_fp_ln_2_3_tl }

```

```

9649 \tl_set:cn { c_fp_ln_2_3_tl }
9650 { { 2 } { 079441541 } { 679835928 } { 0 } }

```

`\fp_ln:Nn` The approach for logarithms is again based on a mix of tables and Taylor series. Here,  
`\fp_ln:cn` the initial validation is a bit easier and so it is set up earlier, meaning less need to escape  
`\fp_gln:Nn` later on.  
`\fp_gln:cn`

```

\fp_ln_aux:NNn 9651 \cs_new_protected_nopar:Npn \fp_ln:Nn {
\fp_ln_aux: 9652 \fp_ln_aux:NNn \tl_set:Nn
\fp_ln_exponent: 9653 }
\fp_ln_internal: 9654 \cs_new_protected_nopar:Npn \fp_gln:Nn {
\fp_ln_exponent_units: 9655 \fp_ln_aux:NNn \tl_gset:Nn
\fp_ln_normalise: 9656 }
\fp_ln_normalise_aux:NNNNNNNN 9657 \cs_generate_variant:Nn \fp_ln:Nn { c }
\fp_ln_mantissa: 9658 \cs_generate_variant:Nn \fp_gln:Nn { c }
\fp_ln_mantissa_aux: 9659 \cs_new_protected_nopar:Npn \fp_ln_aux:NNn #1#2#3 {
\fp_ln_mantissa_divide_two: 9660 \group_begin:
\fp_ln_integer_const:nn 9661 \fp_split:Nn a {#3}
\fp_ln_Taylor: 9662 \fp_standardise:NNNN
\fp_ln_fixed: 9663 \l_fp_input_a_sign_int
\fp_ln_Taylor_aux: 9664 \l_fp_input_a_integer_int
9665 \l_fp_input_a_decimal_int
9666 \l_fp_input_a_exponent_int
9667 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9668 \tex_ifnum:D
9669 \etex_numexpr:D
9670 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
9671 > \c_zero
9672 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9673 \fp_ln_aux:
9674 \tex_else:D
9675 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9676 {
9677 \group_end:
9678 ##1 \exp_not:N ##2 { \c_zero_fp }
9679 }
9680 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9681 \fp_ln_error_msg:
9682 \tex_fi:D
9683 \tex_else:D
9684 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9685 {
9686 \group_end:
9687 ##1 \exp_not:N ##2 { \c_zero_fp }
9688 }
9689 \tex_expandafter:D \fp_ln_error_msg:
9690 \tex_fi:D
9691 \fp_tmp:w #1 #2
9692 }

```

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.

```

9693 \cs_new_protected_nopar:Npn \fp_ln_aux: {
9694   \tl_set:Nx \l_fp_arg_tl
9695   {
9696     +
9697     \int_use:N \l_fp_input_a_integer_int
9698     .
9699     \tex_expandafter:D \use_none:n
9700     \tex_number:D \etex_numexpr:D
9701       \l_fp_input_a_decimal_int + \c_one_thousand_million
9702     e
9703     \int_use:N \l_fp_input_a_exponent_int
9704   }
9705 \etex_ifcsname:D c_fp_ln ( \l_fp_arg_tl ) _fp \tex_endcsname:D
9706 \tex_else:D
9707   \tex_expandafter:D \fp_ln_exponent:
9708 \tex_fi:D
9709 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9710 {
9711   \group_end:
9712   ##1 \exp_not:N ##2
9713   { \use:c { c_fp_ln ( \l_fp_arg_tl ) _fp } }
9714 }
9715 }

```

The main algorithm here uses the fact the logarithm can be divided up, first taking out the powers of ten, then powers of two and finally using a Taylor series for the remainder.

$$\ln(10^n \times 2^m \times x) = \ln(10^n) \times \ln(2^m) \times \ln(x)$$

The second point to remember is that

$$\ln(x^{-1}) = -\ln(x)$$

which means that for the powers of 10 and 2 constants are only needed for positive powers.

The first step is to set up the sign for the output functions and work out the powers of ten in the exponent. First the larger powers are sorted out. The values for the constants are the same as those for the smaller ones, just with a shift in the exponent.

```

9716 \cs_new_protected_nopar:Npn \fp_ln_exponent: {
9717   \fp_ln_internal:
9718   \tex_ifnum:D \l_fp_output_extended_int < \c_five_hundred_million
9719   \tex_else:D
9720     \tex_advance:D \l_fp_output_decimal_int \c_one

```

```

9721 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9722 \tex_else:D
9723   \l_fp_output_decimal_int \c_zero
9724   \tex_advance:D \l_fp_output_integer_int \c_one
9725   \tex_fi:D
9726 \tex_fi:D
9727 \fp_standardise:NNNN
9728   \l_fp_output_sign_int
9729   \l_fp_output_integer_int
9730   \l_fp_output_decimal_int
9731   \l_fp_output_exponent_int
9732 \tl_new:c { c_fp_ln ( \l_fp_arg_tl ) _fp }
9733 \tl_gset:cx { c_fp_ln ( \l_fp_arg_tl ) _fp }
9734 {
9735   \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9736   +
9737   \tex_else:D
9738   -
9739   \tex_fi:D
9740   \int_use:N \l_fp_output_integer_int
9741   .
9742   \tex_expandafter:D \use_none:n
9743   \tex_number:D \etex_numexpr:D
9744     \l_fp_output_decimal_int + \c_one_thousand_million
9745   \scan_stop:
9746   e
9747   \int_use:N \l_fp_output_exponent_int
9748 }
9749 }
9750 \cs_new_protected_nopar:Npn \fp_ln_internal: {
9751   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_zero
9752     \l_fp_input_a_exponent_int - \l_fp_input_a_exponent_int
9753     \l_fp_output_sign_int \c_minus_one
9754   \tex_else:D
9755     \l_fp_output_sign_int \c_one
9756   \tex_fi:D
9757   \tex_ifnum:D \l_fp_input_a_exponent_int > \c_nine
9758     \tl_set:Nx \l_fp_tmp_tl
9759     {
9760       \tex_expandafter:D \use_i:nn
9761       \int_use:N \l_fp_input_a_exponent_int
9762     }
9763     \l_fp_input_a_exponent_int
9764     \etex_numexpr:D
9765       \l_fp_input_a_exponent_int - \l_fp_tmp_tl 0
9766     \scan_stop:
9767   \fp_ln_const:nn { 10 } { \l_fp_tmp_tl }
9768   \tex_advance:D \l_fp_exp_exponent_int \c_one
9769   \l_fp_output_integer_int \l_fp_exp_integer_int
9770   \l_fp_output_decimal_int \l_fp_exp_decimal_int

```

```

9771 \l_fp_output_extended_int \l_fp_exp_extended_int
9772 \l_fp_output_exponent_int \l_fp_exp_exponent_int
9773 \tex_else:D
9774 \l_fp_output_integer_int \c_zero
9775 \l_fp_output_decimal_int \c_zero
9776 \l_fp_output_extended_int \c_zero
9777 \l_fp_output_exponent_int \c_zero
9778 \tex_fi:D
9779 \fp_ln_exponent_units:
9780 }

```

Next the smaller powers of ten, which will need to be combined with the above: always an additive process.

```

9781 \cs_new_protected_nopar:Npn \fp_ln_exponent_units: {
9782 \tex_ifnum:D \l_fp_input_a_exponent_int > \c_zero
9783 \fp_ln_const:nn { 10 } { \int_use:N \l_fp_input_a_exponent_int }
9784 \fp_ln_normalise:
9785 \fp_add:NNNNNNNNN
9786 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9787 \l_fp_output_integer_int \l_fp_output_decimal_int
9788 \l_fp_output_extended_int
9789 \l_fp_output_integer_int \l_fp_output_decimal_int
9790 \l_fp_output_extended_int
9791 \tex_fi:D
9792 \fp_ln_mantissa:
9793 }

```

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.

```

9794 \cs_new_protected_nopar:Npn \fp_ln_normalise: {
9795 \tex_ifnum:D \l_fp_exp_exponent_int < \l_fp_output_exponent_int
9796 \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
9797 \tex_expandafter:D \use_i:nn \tex_expandafter:D
9798 \fp_ln_normalise_aux:NNNNNNNNN
9799 \int_use:N \l_fp_exp_decimal_int
9800 \tex_expandafter:D \fp_ln_normalise:
9801 \tex_fi:D
9802 }
9803 \cs_new_protected_nopar:Npn
9804 \fp_ln_normalise_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
9805 \tex_ifnum:D \l_fp_exp_integer_int = \c_zero
9806 \l_fp_exp_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9807 \tex_else:D
9808 \tl_set:Nx \l_fp_tmp_tl
9809 {
9810 \int_use:N \l_fp_exp_integer_int
9811 #1#2#3#4#5#6#7#8

```

```

9812     }
9813     \l_fp_exp_integer_int \c_zero
9814     \l_fp_exp_decimal_int \l_fp_tmp_tl \scan_stop:
9815     \tex_fi:D
9816     \tex_divide:D \l_fp_exp_extended_int \c_ten
9817     \tl_set:Nx \l_fp_tmp_tl
9818     {
9819         #9
9820         \int_use:N \l_fp_exp_extended_int
9821     }
9822     \l_fp_exp_extended_int \l_fp_tmp_tl \scan_stop:
9823     \tex_advance:D \l_fp_exp_exponent_int \c_one
9824 }

```

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*.

```

9825 \cs_new_protected_nopar:Npn \fp_ln_mantissa: {
9826     \l_fp_count_int \c_zero
9827     \l_fp_input_a_extended_int \c_zero
9828     \fp_ln_mantissa_aux:
9829     \tex_ifnum:D \l_fp_count_int > \c_zero
9830     \fp_ln_const:nn { 2 } { \int_use:N \l_fp_count_int }
9831     \fp_ln_normalise:
9832     \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9833     \tex_expandafter:D \fp_add:NNNNNNNNN
9834     \tex_else:D
9835     \tex_expandafter:D \fp_sub:NNNNNNNNN
9836     \tex_fi:D
9837     \l_fp_output_integer_int \l_fp_output_decimal_int
9838     \l_fp_output_extended_int
9839     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9840     \l_fp_output_integer_int \l_fp_output_decimal_int
9841     \l_fp_output_extended_int
9842     \tex_fi:D
9843     \tex_ifnum:D
9844     \etex_numexpr:D
9845     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int > \c_one
9846     \scan_stop:
9847     \tex_expandafter:D \fp_ln_Taylor:
9848     \tex_fi:D
9849 }
9850 \cs_new_protected_nopar:Npn \fp_ln_mantissa_aux: {
9851     \tex_ifnum:D \l_fp_input_a_integer_int > \c_one
9852     \tex_advance:D \l_fp_count_int \c_one
9853     \fp_ln_mantissa_divide_two:
9854     \tex_expandafter:D \fp_ln_mantissa_aux:
9855     \tex_fi:D
9856 }

```

A fast one-shot division by two.

```

9857 \cs_new_protected_nopar:Npn \fp_ln_mantissa_divide_two: {
9858   \tex_ifodd:D \l_fp_input_a_decimal_int
9859   \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9860   \tex_fi:D
9861   \tex_ifodd:D \l_fp_input_a_integer_int
9862   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9863   \tex_fi:D
9864   \tex_divide:D \l_fp_input_a_integer_int \c_two
9865   \tex_divide:D \l_fp_input_a_decimal_int \c_two
9866   \tex_divide:D \l_fp_input_a_extended_int \c_two
9867 }

```

Recovering constants makes use of the same auxiliary code as for exponents.

```

9868 \cs_new_protected_nopar:Npn \fp_ln_const:nn #1#2 {
9869   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9870   \fp_exp_integer_const:nnnn
9871   \tex_csname:D c_fp_ln_ #1 _ #2 _tl \tex_endcsname:D
9872 }

```

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’.

```

9873 \cs_new_protected_nopar:Npn \fp_ln_Taylor: {

```

```

9874 \group_begin:
9875   \l_fp_input_a_integer_int \c_zero
9876   \l_fp_input_a_exponent_int \c_zero
9877   \l_fp_input_b_integer_int \c_two
9878   \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
9879   \l_fp_input_b_exponent_int \c_zero
9880   \fp_div_internal:
9881   \fp_ln_fixed:
9882   \l_fp_input_a_integer_int \l_fp_output_integer_int
9883   \l_fp_input_a_decimal_int \l_fp_output_decimal_int
9884   \l_fp_input_a_exponent_int \l_fp_output_exponent_int
9885   \l_fp_input_a_extended_int \c_zero
9886   \l_fp_output_decimal_int \c_zero
9887   \l_fp_output_decimal_int \l_fp_input_a_decimal_int
9888   \l_fp_output_extended_int \l_fp_input_a_extended_int
9889   \fp_mul:NNNNNN
9890   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9891   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9892   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9893   \l_fp_count_int \c_one
9894   \fp_ln_Taylor_aux:
9895   \cs_set_protected_nopar:Npx \fp_tmp:w
9896   {
9897     \group_end:
9898     \exp_not:N \l_fp_exp_decimal_int
9899     \int_use:N \l_fp_output_decimal_int \scan_stop:
9900     \exp_not:N \l_fp_exp_extended_int
9901     \int_use:N \l_fp_output_extended_int \scan_stop:
9902     \exp_not:N \l_fp_exp_exponent_int
9903     \int_use:N \l_fp_output_exponent_int \scan_stop:
9904   }
9905   \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.

```

9906 \tex_advance:D \l_fp_exp_decimal_int \l_fp_exp_decimal_int
9907 \tex_ifnum:D \l_fp_exp_extended_int < \c_five_hundred_million
9908 \tex_else:D
9909   \tex_advance:D \l_fp_exp_extended_int -\c_five_hundred_million
9910   \tex_advance:D \l_fp_exp_decimal_int \c_one
9911 \tex_fi:D
9912 \tex_advance:D \l_fp_exp_extended_int \l_fp_exp_extended_int
9913 \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9914   \tex_expandafter:D \fp_add:NNNNNNNNN
9915 \tex_else:D
9916   \tex_expandafter:D \fp_sub:NNNNNNNNN
9917 \tex_fi:D
9918 \l_fp_output_integer_int \l_fp_output_decimal_int
9919   \l_fp_output_extended_int

```

```

9920 \c_zero \l_fp_exp_decimal_int \l_fp_exp_extended_int
9921 \l_fp_output_integer_int \l_fp_output_decimal_int
9922 \l_fp_output_extended_int
9923 }

```

The usual shifts to move to fixed-point working. This is done using the `output` registers as this saves a reassignment here.

```

9924 \cs_new_protected_nopar:Npn \fp_ln_fixed: {
9925   \tex_ifnum:D \l_fp_output_exponent_int < \c_zero
9926   \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9927   \tex_expandafter:D \use_i:nn \tex_expandafter:D
9928   \fp_ln_fixed_aux:NNNNNNNNN
9929   \int_use:N \l_fp_output_decimal_int
9930   \tex_expandafter:D \fp_ln_fixed:
9931   \tex_fi:D
9932 }
9933 \cs_new_protected_nopar:Npn
9934   \fp_ln_fixed_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
9935   \tex_ifnum:D \l_fp_output_integer_int = \c_zero
9936   \l_fp_output_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9937   \tex_else:D
9938     \tl_set:Nx \l_fp_tmp_tl
9939     {
9940       \int_use:N \l_fp_output_integer_int
9941       #1#2#3#4#5#6#7#8
9942     }
9943     \l_fp_output_integer_int \c_zero
9944     \l_fp_output_decimal_int \l_fp_tmp_tl \scan_stop:
9945   \tex_fi:D
9946   \tex_advance:D \l_fp_output_exponent_int \c_one
9947 }

```

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.

```

9948 \cs_new_protected_nopar:Npn \fp_ln_Taylor_aux: {
9949   \tex_advance:D \l_fp_count_int \c_two
9950   \fp_mul:NNNNNN
9951   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9952   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9953   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9954   \tex_ifnum:D
9955     \etex_numexpr:D
9956       \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
9957       > \c_zero
9958     \fp_div_integer:NNNNN
9959     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9960     \l_fp_count_int

```

```

9961 \l_fp_exp_decimal_int \l_fp_exp_extended_int
9962 \tex_advance:D \l_fp_output_decimal_int \l_fp_exp_decimal_int
9963 \tex_advance:D \l_fp_output_extended_int \l_fp_exp_extended_int
9964 \tex_ifnum:D \l_fp_output_extended_int < \c_one_thousand_million
9965 \tex_else:D
9966 \tex_advance:D \l_fp_output_decimal_int \c_one
9967 \tex_advance:D \l_fp_output_extended_int
9968 -\c_one_thousand_million
9969 \tex_fi:D
9970 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9971 \tex_else:D
9972 \tex_advance:D \l_fp_output_integer_int \c_one
9973 \tex_advance:D \l_fp_output_decimal_int
9974 -\c_one_thousand_million
9975 \tex_fi:D
9976 \tex_expandafter:D \fp_ln_Taylor_aux:
9977 \tex_fi:D
9978 }

```

`\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:
9979 \cs_new_protected_nopar:Npn \fp_pow:Nn {
9980 \fp_pow_aux:NNn \tl_set:Nn
9981 }
9982 \cs_new_protected_nopar:Npn \fp_gpow:Nn {
9983 \fp_pow_aux:NNn \tl_gset:Nn
9984 }
9985 \cs_generate_variant:Nn \fp_pow:Nn { c }
9986 \cs_generate_variant:Nn \fp_gpow:Nn { c }
9987 \cs_new_protected_nopar:Npn \fp_pow_aux:NNn #1#2#3 {
9988 \group_begin:
9989 \fp_read:N #2
9990 \l_fp_input_b_sign_int \l_fp_input_a_sign_int
9991 \l_fp_input_b_integer_int \l_fp_input_a_integer_int
9992 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
9993 \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
9994 \fp_split:Nn a {#3}
9995 \fp_standardise:NNNN
9996 \l_fp_input_a_sign_int
9997 \l_fp_input_a_integer_int
9998 \l_fp_input_a_decimal_int
9999 \l_fp_input_a_exponent_int
10000 \tex_ifnum:D
10001 \etex_numexpr:D
10002 \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10003 = \c_zero
10004 \tex_ifnum:D

```

```

10005 \etex_numexpr:D
10006 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10007 = \c_zero
10008 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10009 {
10010 \group_end:
10011 ##1 ##2 { \c_undefined_fp }
10012 }
10013 \tex_else:D
10014 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10015 {
10016 \group_end:
10017 ##1 ##2 { \c_zero_fp }
10018 }
10019 \tex_fi:D
10020 \tex_else:D
10021 \tex_ifnum:D
10022 \etex_numexpr:D
10023 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10024 = \c_zero
10025 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10026 {
10027 \group_end:
10028 ##1 ##2 { \c_one_fp }
10029 }
10030 \tex_else:D
10031 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10032 \fp_pow_aux_i:
10033 \tex_fi:D
10034 \tex_fi:D
10035 \fp_tmp:w #1 #2
10036 }

```

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.

```

10037 \cs_new_protected_nopar:Npn \fp_pow_aux_i: {
10038 \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
10039 \tl_set:Nn \l_fp_sign_tl { + }
10040 \tex_expandafter:D \fp_pow_aux_ii:
10041 \tex_else:D
10042 \l_fp_input_a_extended_int \c_zero
10043 \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
10044 \group_begin:
10045 \fp_extended_normalise:
10046 \tex_ifnum:D
10047 \etex_numexpr:D
10048 \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10049 = \c_zero

```

```

10050         \group_end:
10051         \tl_set:Nn \l_fp_sign_tl { - }
10052         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10053         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10054         \tex_expandafter:D \fp_pow_aux_ii:
10055     \tex_else:D
10056         \group_end:
10057         \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10058         {
10059             \group_end:
10060             ##1 ##2 { \c_undefined_fp }
10061         }
10062     \tex_fi:D
10063 \tex_else:D
10064     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10065     {
10066         \group_end:
10067         ##1 ##2 { \c_undefined_fp }
10068     }
10069 \tex_fi:D
10070 \tex_fi:D
10071 }

```

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.

```

10072 \cs_new_protected_nopar:Npn \fp_pow_aux_ii: {
10073     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10074     \tex_expandafter:D \fp_pow_aux_iv:
10075 \tex_else:D
10076     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
10077     \group_begin:
10078     \l_fp_input_a_extended_int \c_zero
10079     \fp_extended_normalise:
10080     \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
10081     \tex_ifnum:D \l_fp_input_a_integer_int > \c_ten
10082     \group_end:
10083     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10084     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10085     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10086     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10087     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10088     \fp_pow_aux_iv:
10089 \tex_else:D
10090     \group_end:
10091     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10092     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D

```

```

10093         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10094         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10095         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10096         \tex_expandafter:D \fp_pow_aux_iii:
10097     \tex_fi:D
10098 \tex_else:D
10099     \group_end:
10100     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10101     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10102     \tex_expandafter:D \fp_pow_aux_iv:
10103     \tex_fi:D
10104 \tex_else:D
10105     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10106     \fp_pow_aux_iv:
10107     \tex_fi:D
10108 \tex_fi:D
10109 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10110 {
10111     \group_end:
10112     ##1 ##2
10113     {
10114         \l_fp_sign_tl
10115         \int_use:N \l_fp_output_integer_int
10116         .
10117         \tex_expandafter:D \use_none:n
10118         \tex_number:D \etex_numexpr:D
10119         \l_fp_output_decimal_int + \c_one_thousand_million
10120         \scan_stop:
10121         e
10122         \int_use:N \l_fp_output_exponent_int
10123     }
10124 }
10125 }

```

For the small negative integer powers, the calculation is done for the positive power and the reciprocal is then taken.

```

10126 \cs_new_protected_nopar:Npn \fp_pow_aux_iii: {
10127     \l_fp_input_a_sign_int \c_one
10128     \fp_pow_aux_iv:
10129     \l_fp_input_a_integer_int \c_one
10130     \l_fp_input_a_decimal_int \c_zero
10131     \l_fp_input_a_exponent_int \c_zero
10132     \l_fp_input_b_integer_int \l_fp_output_integer_int
10133     \l_fp_input_b_decimal_int \l_fp_output_decimal_int
10134     \l_fp_input_b_exponent_int \l_fp_output_exponent_int
10135     \fp_div_internal:
10136 }

```

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.

```

10137 \cs_new_protected_nopar:Npn \fp_pow_aux_iv: {
10138   \group_begin:
10139     \l_fp_input_a_integer_int  \l_fp_input_b_integer_int
10140     \l_fp_input_a_decimal_int  \l_fp_input_b_decimal_int
10141     \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10142     \fp_ln_internal:
10143     \cs_set_protected_nopar:Npx \fp_tmp:w
10144     {
10145       \group_end:
10146       \exp_not:N \l_fp_input_b_sign_int
10147       \int_use:N \l_fp_output_sign_int \scan_stop:
10148       \exp_not:N \l_fp_input_b_integer_int
10149       \int_use:N \l_fp_output_integer_int \scan_stop:
10150       \exp_not:N \l_fp_input_b_decimal_int
10151       \int_use:N \l_fp_output_decimal_int \scan_stop:
10152       \exp_not:N \l_fp_input_b_extended_int
10153       \int_use:N \l_fp_output_extended_int \scan_stop:
10154       \exp_not:N \l_fp_input_b_exponent_int
10155       \int_use:N \l_fp_output_exponent_int \scan_stop:
10156     }
10157     \fp_tmp:w
10158     \l_fp_input_a_extended_int  \c_zero
10159     \fp_mul:NNNNNNNN
10160     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10161     \l_fp_input_a_extended_int
10162     \l_fp_input_b_integer_int \l_fp_input_b_decimal_int
10163     \l_fp_input_b_extended_int
10164     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10165     \l_fp_input_a_extended_int
10166     \tex_advance:D \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10167     \l_fp_output_integer_int  \c_zero
10168     \l_fp_output_decimal_int  \c_zero
10169     \l_fp_output_exponent_int \c_zero
10170     \cs_set_eq:NN \fp_exp_const:Nx \use_none:nn
10171     \fp_exp_internal:
10172   }

```

### 116.13 Tests for special values

`\fp_if_undefined_p:N` Testing for an undefined value is easy.

`\fp_if_undefined:N $\underline{TF}$`

```

10173 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { T , F , TF , p } {
10174   \tex_ifx:D #1 \c_undefined_fp

```

```

10175     \prg_return_true:
10176     \tex_else:D
10177     \prg_return_false:
10178     \tex_fi:D
10179 }

```

`\fp_if_zero_p:N` Testing for a zero fixed-point is also easy.

```

\fp_if_zero:N $\underline{TF}$ 
10180 \prg_new_conditional:Npnn \fp_if_zero:N #1 { T , F , TF , p } {
10181   \tex_ifx:D #1 \c_zero_fp
10182   \prg_return_true:
10183   \tex_else:D
10184   \prg_return_false:
10185   \tex_fi:D
10186 }

```

## 116.14 Floating-point conditionals

`\fp_compare:nNn $\underline{TF}$`  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.

```

\fp_compare_aux:N
  \fp_compare_=:
  \fp_compare_<:
\fp_compare_<_aux:
\fp_compare_absolute_a>b:
\fp_compare_absolute_a<b:
  \fp_compare_>:
10187 \prg_new_protected_conditional:Npnn \fp_compare:nNn #1#2#3
10188 { T , F , TF }
10189 {
10190   \group_begin:
10191     \fp_split:Nn a {#1}
10192     \fp_standardise:NNNN
10193     \l_fp_input_a_sign_int
10194     \l_fp_input_a_integer_int
10195     \l_fp_input_a_decimal_int
10196     \l_fp_input_a_exponent_int
10197     \fp_split:Nn b {#3}
10198     \fp_standardise:NNNN
10199     \l_fp_input_b_sign_int
10200     \l_fp_input_b_integer_int
10201     \l_fp_input_b_decimal_int
10202     \l_fp_input_b_exponent_int
10203     \fp_compare_aux:N #2
10204   }
10205 \prg_new_protected_conditional:Npnn \fp_compare:NNN #1#2#3
10206 { T , F , TF }
10207 {
10208   \group_begin:
10209     \fp_read:N #3
10210     \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10211     \l_fp_input_b_integer_int   \l_fp_input_a_integer_int
10212     \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int

```

```

10213 \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
10214 \fp_read:N #1
10215 \fp_compare_aux:N #2
10216 }
10217 \cs_new_protected_nopar:Npn \fp_compare_aux:N #1 {
10218 \cs_if_exist:cTF { fp_compare_#1: }
10219 { \use:c { fp_compare_#1: } }
10220 {
10221 \group_end:
10222 \prg_return_false:
10223 }
10224 }

```

For equality, the test is pretty easy as things are either equal or they are not.

```

10225 \cs_new_protected_nopar:cpn { fp_compare_=: } {
10226 \tex_ifnum:D \l_fp_input_a_sign_int = \l_fp_input_b_sign_int
10227 \tex_ifnum:D \l_fp_input_a_integer_int = \l_fp_input_b_integer_int
10228 \tex_ifnum:D \l_fp_input_a_decimal_int = \l_fp_input_b_decimal_int
10229 \tex_ifnum:D
10230 \l_fp_input_a_exponent_int = \l_fp_input_b_exponent_int
10231 \group_end:
10232 \prg_return_true:
10233 \tex_else:D
10234 \group_end:
10235 \prg_return_false:
10236 \tex_fi:D
10237 \tex_else:D
10238 \group_end:
10239 \prg_return_false:
10240 \tex_fi:D
10241 \tex_else:D
10242 \group_end:
10243 \prg_return_false:
10244 \tex_fi:D
10245 \tex_else:D
10246 \group_end:
10247 \prg_return_false:
10248 \tex_fi:D
10249 }

```

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.

```

10250 \cs_new_protected_nopar:cpn { fp_compare_>: } {
10251 \tex_ifnum:D \etex_numexpr:D
10252 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10253 = \c_zero
10254 \tex_ifnum:D \etex_numexpr:D
10255 \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10256 = \c_zero

```

```

10257     \group_end:
10258     \prg_return_false:
10259   \tex_else:D
10260     \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
10261       \group_end:
10262       \prg_return_false:
10263     \tex_else:D
10264       \group_end:
10265       \prg_return_true:
10266     \tex_fi:D
10267   \tex_fi:D
10268 \tex_else:D
10269   \tex_ifnum:D \etex_numexpr:D
10270     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10271     = \c_zero
10272   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10273     \group_end:
10274     \prg_return_true:
10275   \tex_else:D
10276     \group_end:
10277     \prg_return_false:
10278   \tex_fi:D
10279 \tex_else:D
10280   \use:c { fp_compare_>_aux: }
10281 \tex_fi:D
10282 \tex_fi:D
10283 }

```

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.

```

10284 \cs_new_protected_nopar:cpn { fp_compare_>_aux: } {
10285   \tex_ifnum:D \l_fp_input_a_sign_int > \l_fp_input_b_sign_int
10286     \group_end:
10287     \prg_return_true:
10288   \tex_else:D
10289     \tex_ifnum:D \l_fp_input_a_sign_int < \l_fp_input_b_sign_int
10290       \group_end:
10291       \prg_return_false:
10292     \tex_else:D
10293       \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10294         \use:c { fp_compare_absolute_a>b: }
10295       \tex_else:D
10296         \use:c { fp_compare_absolute_a<b: }
10297       \tex_fi:D
10298     \tex_fi:D
10299   \tex_fi:D
10300 }

```

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.

```
10301 \cs_new_protected_nopar:cpn { fp_compare_absolute_a>b: } {
10302   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
10303   \group_end:
10304   \prg_return_true:
10305 \tex_else:D
10306   \tex_ifnum:D \l_fp_input_a_exponent_int < \l_fp_input_b_exponent_int
10307   \group_end:
10308   \prg_return_false:
10309 \tex_else:D
10310   \tex_ifnum:D \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
10311   \group_end:
10312   \prg_return_true:
10313 \tex_else:D
10314   \tex_ifnum:D
10315     \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
10316   \group_end:
10317   \prg_return_false:
10318 \tex_else:D
10319   \tex_ifnum:D
10320     \l_fp_input_a_decimal_int > \l_fp_input_b_decimal_int
10321   \group_end:
10322   \prg_return_true:
10323 \tex_else:D
10324   \group_end:
10325   \prg_return_false:
10326   \tex_fi:D
10327   \tex_fi:D
10328   \tex_fi:D
10329   \tex_fi:D
10330 \tex_fi:D
10331 }
10332 \cs_new_protected_nopar:cpn { fp_compare_absolute_a<b: } {
10333   \tex_ifnum:D \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
10334   \group_end:
10335   \prg_return_true:
10336 \tex_else:D
10337   \tex_ifnum:D \l_fp_input_b_exponent_int < \l_fp_input_a_exponent_int
10338   \group_end:
10339   \prg_return_false:
10340 \tex_else:D
10341   \tex_ifnum:D \l_fp_input_b_integer_int > \l_fp_input_a_integer_int
10342   \group_end:
10343   \prg_return_true:
10344 \tex_else:D
10345   \tex_ifnum:D
10346     \l_fp_input_b_integer_int < \l_fp_input_a_integer_int
10347   \group_end:
10348   \prg_return_false:
```

```

10349         \tex_else:D
10350         \tex_ifnum:D
10351             \l_fp_input_b_decimal_int > \l_fp_input_a_decimal_int
10352         \group_end:
10353         \prg_return_true:
10354     \tex_else:D
10355     \group_end:
10356     \prg_return_false:
10357 \tex_fi:D
10358 \tex_fi:D
10359 \tex_fi:D
10360 \tex_fi:D
10361 \tex_fi:D
10362 }

```

This is just a case of reversing the two input values and then running the tests already defined.

```

10363 \cs_new_protected_nopar:cpn { fp_compare_<: } {
10364     \tl_set:Nx \l_fp_tmp_tl
10365     {
10366         \int_set:Nn \exp_not:N \l_fp_input_a_sign_int
10367         { \int_use:N \l_fp_input_b_sign_int }
10368         \int_set:Nn \exp_not:N \l_fp_input_a_integer_int
10369         { \int_use:N \l_fp_input_b_integer_int }
10370         \int_set:Nn \exp_not:N \l_fp_input_a_decimal_int
10371         { \int_use:N \l_fp_input_b_decimal_int }
10372         \int_set:Nn \exp_not:N \l_fp_input_a_exponent_int
10373         { \int_use:N \l_fp_input_b_exponent_int }
10374         \int_set:Nn \exp_not:N \l_fp_input_b_sign_int
10375         { \int_use:N \l_fp_input_a_sign_int }
10376         \int_set:Nn \exp_not:N \l_fp_input_b_integer_int
10377         { \int_use:N \l_fp_input_a_integer_int }
10378         \int_set:Nn \exp_not:N \l_fp_input_b_decimal_int
10379         { \int_use:N \l_fp_input_a_decimal_int }
10380         \int_set:Nn \exp_not:N \l_fp_input_b_exponent_int
10381         { \int_use:N \l_fp_input_a_exponent_int }
10382     }
10383     \l_fp_tmp_tl
10384     \use:c { fp_compare_>: }
10385 }

```

## 116.15 Messages

`\fp_overflow_msg:` A generic overflow message, used whenever there is a possible overflow.

```

10386 \msg_kernel_new:nnnn { fpu } { overflow }
10387 { Number~too~big. }
10388 {

```

```

10389     The~input~given~is~too~big~for~the~LaTeX~floating~point~unit. \\
10390     Further~errors~may~well~occur!
10391   }
10392   \cs_new_protected_nopar:Npn \fp_overflow_msg: {
10393     \msg_kernel_error:nn { fpu } { overflow }
10394   }

```

`\fp_exp_overflow_msg:` A slightly more helpful message for exponent overflows.

```

10395   \msg_kernel_new:nnnn { fpu } { exponent-overflow }
10396   { Number~too~big~for~exponent~unit. }
10397   {
10398     The~exponent~of~the~input~given~is~too~big~for~the~floating~point~
10399     unit:~the~maximum~input~value~for~an~exponent~is~230.
10400   }
10401   \cs_new_protected_nopar:Npn \fp_exp_overflow_msg: {
10402     \msg_kernel_error:nn { fpu } { exponent-overflow }
10403   }

```

`\fp_ln_error_msg:` Logarithms are only valid for positive number

```

10404   \msg_kernel_new:nnnn { fpu } { logarithm-input-error }
10405   { Invalid~input~to~ln~function. }
10406   { Logarithms~can~only~be~calculated~for~positive~numbers. }
10407   \cs_new_protected_nopar:Npn \fp_ln_error_msg: {
10408     \msg_kernel_error:nn { fpu } { logarithm-input-error }
10409   }

```

`\fp_trig_overflow_msg:` A slightly more helpful message for trigonometric overflows.

```

10410   \msg_kernel_new:nnnn { fpu } { trigonometric-overflow }
10411   { Number~too~big~for~trigonometry~unit. }
10412   {
10413     The~trigonometry~code~can~only~work~with~numbers~smaller~
10414     than~1000000000.
10415   }
10416   \cs_new_protected_nopar:Npn \fp_trig_overflow_msg: {
10417     \msg_kernel_error:nn { fpu } { trigonometric-overflow }
10418   }

10419   </initex | package>

```

## 117 Implementation

Announce and ensure that the required packages are loaded.

```

10420   <*package>

```

```

10421 \ProvidesExplPackage
10422   {\filename}{\filedate}{\fileversion}{\filedescription}
10423 \package_check_loaded_expl:
10424 </package>
10425 <*initex | package>

```

`\lua_now:x` 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

`\lua_shipout:x` before issuing an error. This needs to be expandable, so the same idea is used as for

`\lua_wrong_engine:` V-type expansion, with an appropriately-named but undefined function.

```

10426 \luatex_if_engine:TF
10427 {
10428   \cs_new_eq:NN \lua_now:x      \luatex_directlua:D
10429   \cs_new_eq:NN \lua_shipout:x \luatex_latelua:D
10430 }
10431 {
10432   \cs_new:Npn \lua_now:x #1 { \lua_wrong_engine: }
10433   \cs_new:Npn \lua_shipout:x #1 { \lua_wrong_engine: }
10434 }
10435 \group_begin:
10436 \char_make_letter:N\!
10437 \char_make_letter:N\%
10438 \cs_gset:Npn\lua_wrong_engine:{%
10439 \LuaTeX engine not in use!%
10440 }%
10441 \group_end:%

```

## 117.1 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`

```

10442 \int_new:N \g_cctab_allocate_int
10443 \int_set:Nn \g_cctab_allocate_int { -1 }
10444 \int_new:N \g_cctab_stack_int
10445 \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.

```

10446 \cs_new_protected_nopar:Npn \cctab_new:N #1 {
10447   \cs_if_free:NTF #1
10448   {
10449     \int_gadd:Nn \g_cctab_allocate_int { 2 }

```

```

10450 \int_compare:nNnTF
10451 { \g_cctab_allocate_int } < { \c_allocate_max_tl + 1 }
10452 {
10453   \tex_global:D \tex_mathchardef:D #1 \g_cctab_allocate_int
10454   \luatex_initcatcodetable:D #1
10455 }
10456 {
10457   \msg_kernel_error:nxx { code } { out-of-registers } { cctab }
10458 }
10459 }
10460 {
10461   \msg_kernel_error:nxx { code } { variable-already-defined }
10462   { \token_to_str:N #1 }
10463 }
10464 }
10465 \luatex_if_engine:F {
10466   \cs_set_protected_nopar:Npn \cctab_new:N #1 { \lua_wrong_engine: }
10467 }
10468 <*package>
10469 \luatex_if_engine:T {
10470   \cs_set_protected_nopar:Npn \cctab_new:N #1
10471   {
10472     \newcatcodetable #1
10473     \luatex_initcatcodetable:D #1
10474   }
10475 }
10476 </package>

```

`\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`

```

10477 \cs_new_protected_nopar:Npn \cctab_begin:N #1 {
10478   \seq_gpush:Nx \g_cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
10479   \luatex_catcodetable:D #1
10480   \int_gadd:Nn \g_cctab_stack_int { 2 }
10481   \int_compare:nNnT { \g_cctab_stack_int } > { 268435453 }
10482   { \msg_kernel_error:nn { code } { cctab-stack-full } }
10483   \luatex_savecatcodetable:D \g_cctab_stack_int
10484   \luatex_catcodetable:D \g_cctab_stack_int
10485 }
10486 \cs_new_protected_nopar:Npn \cctab_end: {
10487   \int_gsub:Nn \g_cctab_stack_int { 2 }
10488   \seq_gpop:NN \g_cctab_stack_seq \l_cctab_tmp_tl
10489   \quark_if_no_value:NT \l_cctab_tmp_tl
10490   { \tl_set:Nn \l_cctab_tmp_tl { 0 } }
10491   \luatex_catcodetable:D \l_cctab_tmp_tl \scan_stop:
10492 }
10493 \luatex_if_engine:F {
10494   \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \lua_wrong_engine: }
10495   \cs_set_protected_nopar:Npn \cctab_end: { \lua_wrong_engine: }

```

```

10496 }
10497 <*package>
10498 \luatex_if_engine:T {
10499   \cs_set_protected_nopar:Npn \cctab_begin:N #1
10500     { \BeginCatcodeRegime #1 }
10501   \cs_set_protected_nopar:Npn \cctab_end:
10502     { \EndCatcodeRegime }
10503 }
10504 </package>
10505 \tl_new:N \l_cctab_tmp_tl

```

`\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.

```

10506 \cs_new_protected:Npn \cctab_gset:Nn #1#2 {
10507   \group_begin:
10508     #2
10509     \luatex_savecatcodetable:D #1
10510   \group_end:
10511 }
10512 \luatex_if_engine:F {
10513   \cs_set_protected_nopar:Npn \cctab_gset:Nn #1#2 { \lua_wrong_engine: }
10514 }

```

`\c_code_cctab` Creating category code tables is easy using the function above. The `other` and `string` ones are done by completely ignoring the existing codes as this makes life a lot less complex. The table for `expl3` category codes is always needed, whereas when in package mode the rest can be copied from the existing L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> package `luatex`.

`\c_document_cctab`

`\c_initex_cctab`

`\c_other_cctab`

`\c_string_cctab`

```

10515 \luatex_if_engine:T {
10516   \cctab_new:N \c_code_cctab
10517   \cctab_gset:Nn \c_code_cctab { }
10518 }
10519 <*package>
10520 \luatex_if_engine:T {
10521   \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
10522   \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
10523   \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
10524   \cs_new_eq:NN \c_string_cctab \CatcodeTableString
10525 }
10526 </package>
10527 <!*package>
10528 \luatex_if_engine:T {
10529   \cctab_new:N \c_document_cctab
10530   \cctab_new:N \c_other_cctab
10531   \cctab_new:N \c_string_cctab
10532   \cctab_gset:Nn \c_document_cctab
10533     {

```

```

10534     \char_make_space:n      { 9 }
10535     \char_make_space:n      { 32 }
10536     \char_make_other:n      { 58 }
10537     \char_make_subscript:n  { 95 }
10538     \char_make_active:n     { 126 }
10539   }
10540   \cctab_gset:Nn \c_other_cctab
10541   {
10542     \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
10543     { \char_make_other:n {#1} }
10544   }
10545   \cctab_gset:Nn \c_string_cctab
10546   {
10547     \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
10548     { \char_make_other:n {#1} }
10549     \char_make_space:n { 32 }
10550   }
10551 }
10552 <\/!package>

10553 <\/initex | package>

```