

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

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

2009/06/01

## 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  $\epsilon$ -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 $\epsilon$ . 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 $\epsilon$  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>
1	Naming functions and variables	1
2	Documentation conventions	3
<b>II</b>	<b>The <code>l3names</code> package: A systematic naming scheme for <code>TeX</code></b>	<b>4</b>
3	Setting up the <code>L<sup>A</sup>T<sub>E</sub>X3</code> programming language	4
4	Using the modules	4
<b>III</b>	<b>The <code>l3basics</code> package: Basic Definitions</b>	<b>5</b>
<b>5</b>	<b>Predicates and conditionals</b>	<b>6</b>
5.1	Primitive conditionals . . . . .	7
5.2	Non-primitive conditionals . . . . .	8
5.3	Applications . . . . .	10
<b>6</b>	<b>Control sequences</b>	<b>10</b>
<b>7</b>	<b>Selecting and discarding tokens from the input stream</b>	<b>10</b>
7.1	Extending the interface . . . . .	12
7.2	Selecting tokens from delimited arguments . . . . .	12
<b>8</b>	<b>That which belongs in other modules but needs to be defined earlier</b>	<b>13</b>
<b>9</b>	<b>Defining functions</b>	<b>13</b>
9.1	Defining new functions using primitive parameter text . . . . .	14
9.2	Defining new functions using the signature . . . . .	16
9.3	Defining functions using primitive parameter text . . . . .	17
9.4	Defining functions using the signature (no checks) . . . . .	19
9.5	Undefining functions . . . . .	20
9.6	Copying function definitions . . . . .	20
9.7	Internal functions . . . . .	21

10 The innards of a function	21
11 Grouping and scanning	22
12 Checking the engine	23
<b>IV The <code>l3expan</code> package: Controlling Expansion of Function Arguments</b>	<b>23</b>
13 Brief overview	23
14 Defining new variants	24
14.1 Methods for defining variants . . . . .	24
15 Introducing the variants	25
16 Manipulating the first argument	26
17 Manipulating two arguments	27
18 Manipulating three arguments	27
19 Preventing expansion	28
20 Unbraced expansion	29
<b>V The <code>l3prg</code> package: Program control structures</b>	<b>29</b>
21 Conditionals and logical operations	29
22 Defining a set of conditional functions	30
23 The boolean data type	31
24 Boolean expressions	32
25 Case switches	34
26 Generic loops	35
27 Choosing modes	35

28 Alignment safe grouping and scanning	36
29 Producing $n$ copies	36
30 Sorting	37
<b>VI The l3quark package: “Quarks”</b>	<b>37</b>
31 Functions	38
32 Recursion	38
33 Constants	39
<b>VII The l3token package: A token of my appreciation...</b>	<b>39</b>
34 Character tokens	40
35 Generic tokens	43
35.1 Useless code: because we can! . . . . .	46
36 Peeking ahead at the next token	47
<b>VIII The l3int package: Integers/counters</b>	<b>48</b>
37 Functions	48
38 Formatting a counter value	50
38.1 Internal functions . . . . .	50
39 Variable and constants	51
40 Conversion	52
<b>IX The l3num package: Integers in macros</b>	<b>53</b>
41 Functions	53
42 Formatting a counter value	55

43 Variable and constants	55
44 Primitive functions	55
<b>X The l3intexpr package: Integer expressions</b>	<b>55</b>
45 Functions	56
46 Primitive functions	57
<b>XI The l3skip package: Dimension and skip registers</b>	<b>58</b>
47 Skip registers	59
47.1 Functions . . . . .	59
47.2 Formatting a skip register value . . . . .	61
47.3 Variable and constants . . . . .	61
48 Dim registers	61
48.1 Functions . . . . .	61
48.2 Variable and constants . . . . .	63
49 Muskip	64
<b>XII The l3tl package: Token Lists</b>	<b>64</b>
50 Functions	65
51 Predicates and conditionals	68
52 Working with the contents of token lists	70
53 Variables and constants	71
54 Searching for and replacing tokens	72
55 Heads or tails?	73
<b>XIII The l3toks package: Token Registers</b>	<b>74</b>

56 Allocation and use	74
57 Adding to the contents of token registers	76
58 Predicates and conditionals	77
59 Variable and constants	78
<b>XIV The l3seq package: Sequences</b>	<b>78</b>
60 Functions for creating/initialising sequences	79
61 Adding data to sequences	80
62 Working with sequences	81
63 Predicates and conditionals	82
64 Internal functions	82
65 Functions for ‘Sequence Stacks’	83
<b>XV The l3clist package: Comma separated lists</b>	<b>83</b>
66 Functions for creating/initialising comma-lists	84
67 Putting data in	85
68 Getting data out	86
69 Mapping functions	86
70 Predicates and conditionals	88
71 Higher level functions	88
72 Functions for ‘comma-list stacks’	89
73 Internal functions	90
<b>XVI The l3prop package: Property Lists</b>	<b>90</b>

<b>74 Functions</b>	<b>90</b>
<b>75 Predicates and conditionals</b>	<b>92</b>
<b>76 Internal functions</b>	<b>93</b>
 <b>XVII The l3io package: Low-level file i/o</b>	 <b>93</b>
<b>77 Functions for output streams</b>	<b>94</b>
77.1 Immediate writing . . . . .	94
77.2 Deferred writing . . . . .	95
77.3 Special characters for writing . . . . .	96
<b>78 Functions for input streams</b>	<b>96</b>
<b>79 Constants</b>	<b>97</b>
 <b>XVIII The l3msg package: Communicating with the user</b>	 <b>97</b>
<b>80 Creating new messages</b>	<b>98</b>
<b>81 Message classes</b>	<b>98</b>
<b>82 Redirecting messages</b>	<b>100</b>
<b>83 Support functions for output</b>	<b>100</b>
<b>84 Low-level functions</b>	<b>101</b>
<b>85 Kernel-specific functions</b>	<b>101</b>
<b>86 Variables and constants</b>	<b>102</b>
 <b>XIX The l3box package: Boxes</b>	 <b>103</b>
<b>87 Generic functions</b>	<b>103</b>
<b>88 Horizontal mode</b>	<b>106</b>
<b>89 Vertical mode</b>	<b>107</b>

<b>XX</b>	<b>The <code>l3xref</code> package: Cross references</b>	<b>108</b>
<b>XXI</b>	<b>The <code>l3keyval</code> package: Key-value parsing</b>	<b>108</b>
<b>90</b>	<b>Introduction</b>	<b>109</b>
<b>91</b>	<b>Functions</b>	<b>109</b>
<b>XXII</b>	<b>The <code>l3calc</code> package: Infix notation arithmetic in <math>\LaTeX</math>3</b>	<b>110</b>
<b>92</b>	<b>User functions</b>	<b>111</b>
<b>XXIII</b>	<b>The <code>l3file</code> package: File Loading</b>	<b>112</b>
<b>93</b>	<b>Loading files</b>	<b>113</b>
<b>94</b>	<b>Variables and constants</b>	<b>114</b>
<b>XXIV</b>	<b>Implementation</b>	<b>114</b>
<b>95</b>	<b><code>l3names</code> implementation</b>	<b>114</b>
95.1	Internal functions . . . . .	115
95.2	Package loading . . . . .	115
95.3	Catcode assignments . . . . .	115
95.4	Setting up primitive names . . . . .	116
95.5	Reassignment of primitives . . . . .	117
95.6	<code>expl3</code> code switches . . . . .	126
95.7	Package loading . . . . .	128
95.8	Finishing up . . . . .	131
95.9	Showing memory usage . . . . .	133
<b>96</b>	<b><code>l3basics</code> implementation</b>	<b>133</b>
96.1	Renaming some $\TeX$ primitives (again) . . . . .	133
96.2	Defining functions . . . . .	135
96.3	Selecting tokens . . . . .	136
96.4	Gobbling tokens from input . . . . .	137



96.5	Expansion control from l3expan . . . . .	137
96.6	Conditional processing and definitions . . . . .	138
96.7	Dissecting a control sequence . . . . .	141
96.8	Exist or free . . . . .	143
96.9	Defining and checking (new) functions . . . . .	145
96.10	More new definitions . . . . .	148
96.11	Copying definitions . . . . .	150
96.12	Undefining functions . . . . .	150
96.13	Engine specific definitions . . . . .	151
96.14	Scratch functions . . . . .	151
96.15	Defining functions from a given number of arguments . . . . .	151
96.16	Using the signature to define functions . . . . .	153
<b>97</b>	<b>l3expan implementation</b>	<b>156</b>
97.1	Internal functions and variables . . . . .	156
97.2	Module code . . . . .	157
97.3	General expansion . . . . .	158
97.4	Hand-tuned definitions . . . . .	161
97.5	Definitions with the ‘general’ technique . . . . .	162
97.6	Preventing expansion . . . . .	163
97.7	Defining function variants . . . . .	163
97.8	Last-unbraced versions . . . . .	165
<b>98</b>	<b>l3prg implementation</b>	<b>166</b>
98.1	Variables . . . . .	166
98.2	Module code . . . . .	166
98.3	Choosing modes . . . . .	167
98.4	Producing $n$ copies . . . . .	168
98.5	Booleans . . . . .	171
98.6	Parsing boolean expressions . . . . .	173
98.7	Case switch . . . . .	177
98.8	Sorting . . . . .	179
<b>99</b>	<b>l3quark implementation</b>	<b>181</b>

<b>1003token implementation</b>	<b>184</b>
100.1Documentation of internal functions . . . . .	184
100.2Module code . . . . .	185
100.3Character tokens . . . . .	185
100.4Generic tokens . . . . .	187
100.5Peeking ahead at the next token . . . . .	194
<b>1013int implementation</b>	<b>200</b>
101.1Internal functions and variables . . . . .	200
101.2Module loading and primitives definitions . . . . .	200
101.3Allocation and setting . . . . .	201
101.4Defining constants . . . . .	207
101.5Scanning and conversion . . . . .	208
<b>1023num implementation</b>	<b>211</b>
<b>1033intexpr implementation</b>	<b>213</b>
<b>1043skip implementation</b>	<b>217</b>
104.1Skip registers . . . . .	217
104.2Dimen registers . . . . .	221
104.3Muskips . . . . .	223
<b>1053tl implementation</b>	<b>223</b>
105.1Functions . . . . .	224
105.2Variables and constants . . . . .	229
105.3Predicates and conditionals . . . . .	229
105.4Working with the contents of token lists . . . . .	234
105.5Checking for and replacing tokens . . . . .	238
105.6Heads or tails? . . . . .	241
<b>1063toks implementation</b>	<b>242</b>
106.1Allocation and use . . . . .	243
106.2Adding to token registers' contents . . . . .	245
106.3Predicates and conditionals . . . . .	246
106.4Variables and constants . . . . .	247

<b>1073seq implementation</b>	<b>247</b>
107.1Allocating and initialisation . . . . .	247
107.2Predicates and conditionals . . . . .	248
107.3Getting data out . . . . .	249
107.4Putting data in . . . . .	250
107.5Mapping . . . . .	251
107.6Manipulation . . . . .	252
107.7Sequence stacks . . . . .	252
<b>1083clist implementation</b>	<b>253</b>
108.1Allocation and initialisation . . . . .	253
108.2Predicates and conditionals . . . . .	254
108.3Retrieving data . . . . .	255
108.4Storing data . . . . .	256
108.5Mapping . . . . .	257
108.6Higher level functions . . . . .	259
108.7Stack operations . . . . .	260
<b>1093prop implementation</b>	<b>261</b>
109.1Functions . . . . .	261
109.2Predicates and conditionals . . . . .	264
109.3Mapping functions . . . . .	264
<b>1103io implementation</b>	<b>266</b>
110.1Output streams . . . . .	266
110.2Input streams . . . . .	269
<b>1113msg implementation</b>	<b>270</b>
111.1Variables and constants . . . . .	270
111.2Output helper functions . . . . .	272
111.3Generic functions . . . . .	272
111.4General functions . . . . .	275
111.5Redirection functions . . . . .	279
111.6Kernel-specific functions . . . . .	279

<b>1123box implementation</b>	<b>281</b>
112.1Generic boxes . . . . .	282
112.2Vertical boxes . . . . .	284
112.3Horizontal boxes . . . . .	285
<b>1133xref implementation</b>	<b>286</b>
113.1Internal functions and variables . . . . .	286
113.2Module code . . . . .	286
<b>1143xref test file</b>	<b>289</b>
<b>1153keyval implementation</b>	<b>290</b>
115.1Internal functions and variables . . . . .	290
115.2Module code . . . . .	291
<b>1163calc implementation</b>	<b>297</b>
116.1Variables . . . . .	297
116.2Internal functions . . . . .	298
116.3Module code . . . . .	298
116.4Higher level commands . . . . .	307
<b>1173file implementation</b>	<b>310</b>
<b>Index</b>	<b>314</b>

## 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 L<sup>A</sup>T<sub>E</sub>X3 programming language is found in [expl3.pdf](#).

## 1 Naming functions and variables

L<sup>A</sup>T<sub>E</sub>X3 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 T<sub>E</sub>X primitives are initially `\let` to a D name, and some are then given a second name. Only the kernel team should use anything with a D specifier!
- N and n** These mean *no manipulation*, of a single token for N and of a set of tokens given in braces for n. Both pass the argument though exactly as given. Usually, if you use a single token for an n argument, all will be well.
- c** This means *cname*, and indicates that the argument will be turned into a cname before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The V and v specifiers are used to get the content of a variable without needing to worry about the underlying T<sub>E</sub>X structure containing the data. A V argument will be a single token (similar to N), for example `\foo:V \MyVariable`; on the other hand, using v a cname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.
- o** This means *expansion once*. In general, the V and v specifiers are favoured over o for recovering stored information. However, o is useful for correctly processing information with delimited arguments.
- x** The x specifier stands for *exhaustive expansion*: the plain T<sub>E</sub>X `\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.
- 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).
- tl** Token list variables: placeholder for a token list.
- toks** Token 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.

## 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</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</code> <i>&lt;sequence&gt;</i>
--	---

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 T<sub>E</sub>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</code> <i>&lt;cs&gt;</i>
-----------------------------	---

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</code> <i>&lt;true code&gt;</i> <i>&lt;false code&gt;</i>
------------------------------------	---

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  $\langle true\ code \rangle$  and  $\langle false\ code \rangle$ , respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the **expl3** modules should be defined in this way.

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

`\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  $\text{\LaTeX} 2_{\epsilon}$  or plain  $\text{\TeX}$ . In these cases, the text will include an extra ‘**TeXhackers note**’ section:

`\token_to_str:N *` `\token_to_str:N`  $\langle token \rangle$   
The normal description text.

**TeXhackers note:** Detail for the experienced  $\text{\TeX}$  or  $\text{\LaTeX} 2_{\epsilon}$  programmer. In this case, it would point out that this function is the  $\text{\TeX}$  primitive `\string`.

## Part II

# The l3names package

## A systematic naming scheme for $\text{\TeX}$

### 3 Setting up the $\text{\LaTeX} 3$ programming language

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

- defines new names for all  $\text{\TeX}$  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  $\text{\LaTeX} 2_{\epsilon}$  context.

### 4 Using the modules

The modules documented in **source3** are designed to be used on top of  $\text{\LaTeX} 2_{\epsilon}$  and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the  $\text{\LaTeX} 3$  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>\ExplSyntaxOn &lt;code&gt; \ExplSyntaxOff</code>
<code>\ExplSyntaxOff</code>	

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

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

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

<code>\ProvidesExplPackage</code>	<code>\RequirePackage{expl3}</code>
<code>\ProvidesExplClass</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 L<sup>A</sup>T<sub>E</sub>X3 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>\RequirePackage{l3names}</code>
<code>\filename</code>	
<code>\filenameext</code>	
<code>\filedate</code>	
<code>\fileversion</code>	
<code>\filetimestamp</code>	
<code>\fileauthor</code>	
<code>\filedescription</code>	

`\GetIdInfo $Id: < cvs or svn info field> $ {<description>}`

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

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or alike are loaded with usual L<sup>A</sup>T<sub>E</sub>X catcodes and the L<sup>A</sup>T<sub>E</sub>X3 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 \langle true code \rangle \else:
\langle false code \rangle \fi:
```

or in expressions utilizing the boolean logic parser:

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {\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  $\text{\TeX}$  and  $\text{\LaTeX}$ . 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  $\varepsilon$ - $\text{\TeX}$  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 `\intexpr_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>`. `\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 `\intexpr` for more.

**$\text{\TeX}$ hackers note:** These are equivalent to their corresponding  $\text{\TeX}$  primitive conditionals; `\reverse_if:N` is  $\varepsilon$ - $\text{\TeX}$ ’s `\unless`.

<code>\if_meaning:w</code>	*	<code>\if_meaning:w &lt;arg<sub>1</sub>&gt; &lt;arg<sub>2</sub>&gt; &lt;true code&gt; \else: &lt;false code&gt;</code>
		<code>\fi:</code>

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

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

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

<code>\if:w</code>	<code>*</code>	<code>\if:w</code>	<code>&lt;token<sub>1</sub>&gt;</code>	<code>&lt;token<sub>2</sub>&gt;</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false code&gt;</code>	<code>\fi:</code>
<code>\if_charcode:w</code>	<code>*</code>	<code>\if_catcode:w</code>	<code>&lt;token<sub>1</sub>&gt;</code>	<code>&lt;token<sub>2</sub>&gt;</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false</code>	
<code>\if_catcode:w</code>	<code>*</code>				<code>code&gt;</code>			<code>\fi:</code>

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

<code>\if_predicate:w</code>	<code>*</code>	<code>\if_predicate:w</code>	<code>&lt;predicate&gt;</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false code&gt;</code>	<code>\fi:</code>
------------------------------	----------------	------------------------------	--------------------------------	--------------------------------	---------------------	-------------------------------------	-------------------

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

<code>\if_bool:N</code>	<code>*</code>	<code>\if_bool:N</code>	<code>&lt;boolean&gt;</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false code&gt;</code>	<code>\fi:</code>
-------------------------	----------------	-------------------------	------------------------------	--------------------------------	---------------------	-------------------------------------	-------------------

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

<code>\if_cs_exist:N</code>	<code>*</code>	<code>\if_cs_exist:N</code>	<code>&lt;cs&gt;</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false code&gt;</code>	<code>\fi:</code>
<code>\if_cs_exist:w</code>	<code>*</code>	<code>\if_cs_exist:w</code>	<code>&lt;tokens&gt;</code>	<code>\cs_end:</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false</code>
				<code>code&gt;</code>			<code>\fi:</code>

Check if `<cs>` appears in the hash table or if the control sequence that can be formed from `<tokens>` appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

<code>\if_mode_horizontal:</code>	<code>*</code>	<code>\if_mode_horizontal:</code>	<code>&lt;true code&gt;</code>	<code>\else:</code>	<code>&lt;&gt;false code&gt;</code>	<code>\fi:</code>
<code>\if_mode_vertical:</code>	<code>*</code>					
<code>\if_mode_math:</code>	<code>*</code>					
<code>\if_mode_inner:</code>	<code>*</code>					

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>	<code>\cs_if_eq_name_p:NN</code>	<code>&lt;cs<sub>1</sub>&gt;</code>	<code>&lt;cs<sub>2</sub>&gt;</code>
----------------------------------	----------------------------------	-------------------------------------	-------------------------------------

Returns ‘true’ if `<cs1>` and `<cs2>` 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  $\langle cs_1 \rangle$  and  $\langle cs_2 \rangle$  have same meaning.

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

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

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

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

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

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

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

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

`\chk_if_free_cs:N <cs>`

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

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

`\chk_if_exist_cs:N <cs>`

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

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

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

## 5.3 Applications

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

`\str_if_eq_p:nn {<string1>} {<string2>}`

Expands to ‘true’ if  $\langle string_1 \rangle$  is the same as  $\langle string_2 \rangle$ , otherwise ‘false’. Ignores spaces within the strings.

<code>\str_if_eq_var_p:nf *</code>
------------------------------------

`\str_if_eq_var_p:nf {<string1>} {<string2>}`

A variant of `\str_if_eq_p:nn` which has the advantage of obeying spaces in at least the second argument.

## 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`  $\{\langle arg \rangle\}$

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>	★
---------------------	---

`\use:c`  $\{\langle cs \rangle\}$

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_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>	★

`\use_i:nn`  $\{\langle code_1 \rangle\}$   $\{\langle code_2 \rangle\}$

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>\use_ii:nnn</code>	★
<code>\use_iii:nnn</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.

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

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

`\use_i:nnnn {⟨arg1⟩} {⟨arg2⟩} {⟨arg3⟩} {⟨arg4⟩}`

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

## 7.1 Extending the interface

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

`\use_i_ii:nnn {⟨arg1⟩} {⟨arg2⟩} {⟨arg3⟩}`

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

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

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

## 7.2 Selecting tokens from delimited arguments

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

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

`\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil`

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

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

`\use_i_delimit_by_q_nil:nw {⟨arg⟩} ⟨balanced text⟩ \q_nil`

Gobbles *⟨balanced text⟩* and executes *⟨arg⟩* afterwards. This can also be used to get the first item in a token list.

<code>\use_i_after_fi:nw</code>	★	<code>\use_i_after_fi:nw {⟨arg⟩} \fi:</code>
<code>\use_i_after_else:nw</code>	★	<code>\use_i_after_else:nw {⟨arg⟩} \else: ⟨balanced text⟩ \fi:</code>
<code>\use_i_after_or:nw</code>	★	<code>\use_i_after_or:nw {⟨arg⟩} \or: ⟨balanced text⟩ \fi:</code>
<code>\use_i_after_orelse:nw</code>	★	<code>\use_i_after_orelse:nw {⟨arg⟩} \or:/\else: ⟨balanced text⟩ \fi:</code>

Executes *⟨arg⟩* 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>
------------------------------

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

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>\exp_not:n *</code>

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

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>
---------------------------------

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>

`\iow_log:x {\<message>}`  
`\iow_shipout_x:Nn <write_stream> {\<message>}`

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

<code>\msg_kernel_bug:x</code>
--------------------------------

`\msg_kernel_bug:x {\<message>}`

Internal function for calling errors in our code.

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

Placeholder for a function to be defined by `l3chk`.

<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_gnew:Npn</code>
<code>\cs_gnew:Npx</code>
<code>\cs_gnew:cpn</code>
<code>\cs_gnew:cpx</code>

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

Global versions of the above 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 TeX 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_gnew_nopar:Npn</code>
<code>\cs_gnew_nopar:Npx</code>
<code>\cs_gnew_nopar:cpn</code>
<code>\cs_gnew_nopar:cpx</code>

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

Like `\cs_new_nopar:Npn` but defines the new function globally. See comments above.

<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_gnew_protected:Npn</code>
<code>\cs_gnew_protected:Npx</code>
<code>\cs_gnew_protected:cpn</code>
<code>\cs_gnew_protected:cpx</code>

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

Global versions of the above functions.

<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.

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

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

Global versions of the above functions.

## 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_gnew:Nn</code>
<code>\cs_gnew:Nx</code>
<code>\cs_gnew:cn</code>
<code>\cs_gnew:cx</code>

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

Global versions of the above functions.

<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_gnew_nopar:Nn</code>
<code>\cs_gnew_nopar:Nx</code>
<code>\cs_gnew_nopar:cn</code>
<code>\cs_gnew_nopar:cx</code>

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

Global versions of the above 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_gnew_protected:Nn</code> <code>\cs_gnew_protected:Nx</code> <code>\cs_gnew_protected:cn</code> <code>\cs_gnew_protected:cx</code>	<code>\cs_gnew_protected:Nn &lt;cs&gt; {&lt;code&gt;}</code>
--	--

Global versions of the above functions.

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

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

Global versions of the above functions.

### 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:cpn</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>

`\cs_gset:Npn <cs> <parms> {\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>

`\cs_set_nopar:Npn <cs> <parms> {\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>

`\cs_gset_nopar:Npn <cs> <parms> {\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>	<code>\cs_gset_protected_nopar:Npn &lt;cs&gt; &lt;parms&gt; {\code}</code>
--	--

Global versions of the above functions.

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

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

<code>\cs_set:Nn</code> <code>\cs_set:Nx</code> <code>\cs_set:cn</code> <code>\cs_set:cX</code>	<code>\cs_set:Nn &lt;cs&gt; {\code}</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>	<code>\cs_gset:Nn &lt;cs&gt; {\code}</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>	<code>\cs_set_nopar:Nn &lt;cs&gt; {\code}</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>	<code>\cs_gset_nopar:Nn &lt;cs&gt; {\code}</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>	<code>\cs_set_protected:Nn &lt;cs&gt; {\code}</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>	<code>\cs_gset_protected:Nn &lt;cs&gt; {&lt;code&gt;}</code>
--	--

Global versions of the above functions.

<code>\cs_set_protected_nopar:Nn</code> <code>\cs_set_protected_nopar:cn</code> <code>\cs_set_protected_nopar:Nx</code> <code>\cs_set_protected_nopar:cx</code>	<code>\cs_set_protected_nopar:Nn &lt;cs&gt; {&lt;code&gt;}</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_gundefine:N</code> <code>\cs_gundefine:c</code>	<code>\cs_gundefine:N &lt;cs&gt;</code>
--	---

Undefines the control sequence globally.

## 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_gnew_eq:NN</code> <code>\cs_gnew_eq:cN</code> <code>\cs_gnew_eq:Nc</code> <code>\cs_gnew_eq:cc</code>	<code>\cs_new_eq:NN &lt;cs<sub>12 </sub></code>
--	---

Gives the function `<cs1 locally or globally the current meaning of <cs2. If <cs1 already exists then an error is called.`



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

`\cs_set_eq:cN`  $\langle cs_1 \rangle$   $\langle cs_2 \rangle$

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

<code>\cs_set_eq:NwN</code>
-----------------------------

`\cs_set_eq:NwN`  $\langle cs_1 \rangle$   $\langle cs_2 \rangle$   
`\cs_set_eq:NwN`  $\langle cs_1 \rangle = \langle cs_2 \rangle$

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

**TeXhackers note:** `\cs_set_eq:NwN` is the L<sup>A</sup>T<sub>E</sub>X3 name for TeX's `\let`.

## 9.7 Internal functions

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

`\pref_global:D` `\cs_set_nopar:Npn`

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

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

**TeXhackers note:** These prefixes are the primitives `\global`, `\long`, and `\protected`. The `\outer` prefix isn't used at all within L<sup>A</sup>T<sub>E</sub>X3 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>
-----------------------------

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

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>

`\token_to_str:N <arg>`

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

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

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

`\token_to_meaning:N <arg>`

This function returns the type and definition of  $\langle arg \rangle$  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  $\langle fn \rangle$ . The **signature** variants does the same but returns the signature  $\langle args \rangle$  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  $\langle post\ process \rangle$ , i.e.,  $\langle post\ process \rangle\{\langle name \rangle\}\{\langle signature \rangle\}\langle true \rangle/\langle false \rangle$ . 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  $\langle args \rangle$ , 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.

**TeXhackers note:** These are the TeX 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:`).

**TeXhackers note:** This is TeX's `\aftergroup`.

## 12 Checking the engine

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

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

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

<code>\luatex_if_engine:TF *</code>
-------------------------------------

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

This function detects if we're running a LuaTeX-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>\function</code>	<code>\variant argument</code>
<code>\cs_generate_variant:Nn</code>	<code>\cs_generate_variant:Nn</code>	<code>\foo:Nn</code>	<code>{c}</code>
	<code>\cs_generate_variant:Nn</code>	<code>\foo:Nn</code>	<code>{c,Nx,cx}</code>

This function greatly simplify the task of defining variations on a base function with differing expansion control. The first example is equivalent to writing

```
\cs_set_nopar:Npn \foo:cn { \exp_args:Nc \foo:Nn }
```

If used with a comma-separated list of variants, it does so for each variant form, i.e., the second example is equivalent to

```
\cs_set_nopar:Npn \foo:cn { \exp_args:Nc \foo:Nn }
\cs_set_nopar:Npn \foo:Nx { \exp_args:NNx \foo:Nn }
\cs_set_nopar:Npn \foo:cx { \exp_args:Ncx \foo:Nn }
```

## 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!

When pdfTeX 1.50 arrives, it will contain a primitive for performing the equivalent of an `x` expansion after only one expansion and most importantly: as an expandable operation.

`\exp_arg:x`

`\exp_arg:x {\langle arg \rangle}`  
 $\langle arg \rangle$  is expanded fully using an `x` expansion.

## 16 Manipulating the first argument

`\exp_args:No *`

`\exp_args:No \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 once, 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.

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

`\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.

`\exp_args:NV *`

`\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.

`\exp_args:Nv *`

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

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

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

## 18 Manipulating three arguments

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

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

`\exp_args:Nnnx <funct> <arg_1> <arg_2> <arg_3> ...`

All the above functions are non-expandable.

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

`\exp_args:NNoo <funct> <arg_1> <arg_2> <arg_3> ...`

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

## 19 Preventing expansion

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

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

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

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

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

`\exp_not:o {<token list>}`

Same as `\exp_not:n` except `<token list>` is expanded once for the `o` type and twice for the `d` type and the result of this expansion is then prohibited from being expanded further.

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

`\exp_not:V <register>`  
`\exp_not:v {<token list>}`

The value of `<register>` is retrieved and then passed on to `\exp_not:n` which will prohibit



further expansion. The `v` type first creates a control sequence from  $\langle token\ list \rangle$  but is otherwise identical to `V`.

```
\exp_stop_f: \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.

**T<sub>E</sub>Xhackers 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

```
\exp_last_unbraced:Nf
\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:NcV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNV
\exp_last_unbraced:NNNo \exp_last_unbraced:NV \langle token \rangle \langle variable name \rangle
```

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

```
\prg_return_true:
```

```
\prg_return_false:
```

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

```
\prg_set_conditional:Nnn
```

```
\prg_set_conditional:Npnn
```

```
\prg_new_conditional:Nnn
```

```
\prg_new_conditional:Npnn
```

```
\prg_set_protected_conditional:Nnn
```

```
\prg_set_protected_conditional:Npnn
```

```
\prg_new_protected_conditional:Nnn
```

```
\prg_new_protected_conditional:Npnn
```

```
\prg_set_conditional:Nnn <test> <conds> <code>
```

```
\prg_set_conditional:Npnn <test> <param> <conds> <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).

<pre> \bool_new:N \bool_new:c </pre>	<code>\bool_new:N &lt;bool&gt;</code>
--------------------------------------	---------------------------------------

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

<pre> \bool_set_true:N \bool_set_true:c \bool_set_false:N \bool_set_false:c \bool_gset_true:N \bool_gset_true:c \bool_gset_false:N \bool_gset_false:c </pre>	<code>\bool_gset_false:N &lt;bool&gt;</code>
--	--

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

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

`\bool_set_eq:NN`  $\langle bool_1 \rangle$   $\langle bool_2 \rangle$   
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`  $\langle bool \rangle$   $\{\langle true \rangle\}$   $\{\langle false \rangle\}$   
`\bool_if_p:N`  $\langle bool \rangle$

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`  $\langle bool \rangle$   $\{\langle code \rangle\}$   
`\bool_until_do:Nn`  $\langle bool \rangle$   $\{\langle code \rangle\}$

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

## 24 Boolean expressions

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

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

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

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

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

The functions evaluate the truth value of *<boolean expression>* 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{
  \intexpr_compare_p:n {1=1} &&
  (
    \intexpr_compare_p:n {2=3} ||
    \intexpr_compare_p:n {4=4} ||
    \intexpr_compare_p:n {1=\error} % is skipped
  ) &&
  !(\intexpr_compare_p:n {2=4})
}

```

from above returns *<true>*.

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

<code>\bool_not_p:n *</code>	<code>\bool_not_p:n {&lt;boolean expression&gt;}</code>
------------------------------	---

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

<code>\bool_xor_p:nn *</code>	<code>\bool_xor_p:nn {&lt;boolean expression&gt;} {&lt;boolean expression&gt;}</code>
-------------------------------	---

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

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

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

```

\prg_case_int:nnn{2*5}{
  {5}{Small} {4+6}{Medium} {-2*10}{Negative}
}{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  $\langle else\ case \rangle$  is executed. The example above will return “Medium”.

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

```

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

```

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

```

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

```

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

The function is 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> {<code1>} <tl var.2> {<code2>} ... <tl var.n>
  {<coden>}
\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.

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

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

## 26 Generic loops

<code>\bool_while_do:nn</code>	
<code>\bool_until_do:nn</code>	
<code>\bool_do_while:nn</code>	<code>\bool_while_do:nn {&lt;boolean expression&gt;} {&lt;code&gt;}</code>
<code>\bool_do_until:nn</code>	<code>\bool_until_do:nn {&lt;boolean expression&gt;} {&lt;code&gt;}</code>

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

## 27 Choosing modes

<code>\mode_if_vertical_p: *</code>	
<code>\mode_if_vertical:TF *</code>	<code>\mode_if_vertical:TF {&lt;true code&gt;} {&lt;false code&gt;}</code>

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

<code>\mode_if_horizontal_p: *</code>	
<code>\mode_if_horizontal:TF *</code>	<code>\mode_if_horizontal:TF {&lt;true code&gt;} {&lt;false code&gt;}</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 {&lt;true code&gt;} {&lt;false code&gt;}</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 {&lt;true code&gt;} {&lt;false code&gt;}</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>
-------------------------------------

`\scan_align_safe_stop:`

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>

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

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

## 29 Producing $n$ copies

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

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

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

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

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

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

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

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

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

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

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

`\prg_stepwise_variable:nnnNn {<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>	<code>\prg_quicksort:n { {\langle item_1 \rangle} {\langle item_2 \rangle} ... {\langle item_n \rangle} }</code>
-------------------------------	--

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

<code>\prg_quicksort_function:n</code>	<code>\prg_quicksort_function:n {\langle element \rangle}</code>
<code>\prg_quicksort_compare:nnTF</code>	<code>\prg_quicksort_compare:nnTF {\langle element_1 \rangle} {\langle element_2 \rangle}</code>

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

```
\cs_set_nopar:Nn\prg_quicksort_function:n {{#1}}
\cs_set_nopar:Nn\prg_quicksort_compare:nnTF {\intexpr_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:Nn\prg_quicksort_compare:nnTF {
\intexpr_compare:nNnTF{\tl_compare:nn{#1}{#2}}>\c_zero }
```

## 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

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

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

`\quark_if_nil:NTF <token> <true code> <false 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:nTF &lt;tokens&gt; &lt;true code&gt; &lt;false code&gt;</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_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>	

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.

---

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

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

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

<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 *⟨list element⟩* 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 *⟨list element⟩* 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 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_set_catcode:nn {&lt;char number&gt;} {&lt;number&gt;}</code>
<code>\char_value_catcode:w</code>	<code>\char_set_catcode:w &lt;char&gt; = &lt;number&gt;</code>
<code>\char_show_value_catcode:n</code>	<code>\char_value_catcode:n {&lt;char number&gt;}</code>
<code>\char_show_value_catcode:w</code>	<code>\char_show_value_catcode:n {&lt;char number&gt;}</code>

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

`\char_set_catcode` is more usefully abstracted below.

**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`.

<code>\char_set_lccode:nn</code>	
<code>\char_set_lccode:w</code>	
<code>\char_value_lccode:n</code>	<code>\char_set_lccode:nn {⟨char⟩} {⟨number⟩}</code>
<code>\char_value_lccode:w</code>	<code>\char_set_lccode:w ⟨char⟩ = ⟨number⟩</code>
<code>\char_show_value_lccode:n</code>	<code>\char_value_lccode:n {⟨char⟩}</code>
<code>\char_show_value_lccode:w</code>	<code>\char_show_value_lccode:n {⟨char⟩}</code>

Set the lower case representation of  $\langle char \rangle$  for when  $\langle char \rangle$  is being converted in `\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:** `\char_set_lccode:w` is the T<sub>E</sub>X primitive `\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 {\langle char \rangle} {\langle number \rangle} \char_set_uccode:w \langle char \rangle = \langle number \rangle \char_value_uccode:n {\langle char \rangle} \char_show_value_uccode:n {\langle char \rangle} </pre>
---	--

Set the uppercase representation of  $\langle char \rangle$  for when  $\langle char \rangle$  is being converted in `\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:** `\char_set_uccode:w` is the T<sub>E</sub>X primitive `\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 {\langle char \rangle} {\langle number \rangle} \char_set_sfcode:w \langle char \rangle = \langle number \rangle \char_value_sfcode:n {\langle char \rangle} \char_show_value_sfcode:n {\langle char \rangle} </pre>
---	--

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

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

<pre> \char_set_mathcode:nn \char_set_mathcode:w \char_gset_mathcode:nn \char_gset_mathcode:w \char_value_mathcode:n \char_value_mathcode:w \char_show_value_mathcode:n \char_show_value_mathcode:w </pre>	<pre> \char_set_mathcode:nn {\langle char \rangle} {\langle number \rangle} \char_set_mathcode:w \langle char \rangle = \langle number \rangle \char_value_mathcode:n {\langle char \rangle} \char_show_value_mathcode:n {\langle char \rangle} </pre>
--	--

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

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

## 35 Generic tokens

<code>\token_new:Nn</code>
----------------------------

`\token_new:Nn <token1> {<token2>}`

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>

`\token_if_group_begin:NTF <token> {<true>} {<false>}`

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>

`\token_if_group_end:NTF <token> {<true>} {<false>}`

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

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

`\token_if_math_shift:NTF <token> {<true>} {<false>}`

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

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

`\token_if_alignment_tab:NTF <token> {<true>} {<false>}`

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

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

`\token_if_parameter:NTF <token> {<true>} {<false>}`

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

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

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

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

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

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

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

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

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

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

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

$\backslash\text{token\_if\_active\_char\_p:N} \star$ $\backslash\text{token\_if\_active\_char:N}\underline{TF} \star$	$\backslash\text{token\_if\_active\_char:N}\underline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
---	---

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

$\backslash\text{token\_if\_eq\_meaning\_p:NN} \star$ $\backslash\text{token\_if\_eq\_meaning:NN}\underline{TF} \star$	$\backslash\text{token\_if\_eq\_meaning:NN}\underline{TF} \langle token_1 \rangle \langle token_2 \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
---	---

Check if the meaning of two tokens are identical.

$\backslash\text{token\_if\_eq\_catcode\_p:NN} \star$ $\backslash\text{token\_if\_eq\_catcode:NN}\underline{TF} \star$	$\backslash\text{token\_if\_eq\_catcode:NN}\underline{TF} \langle token_1 \rangle \langle token_2 \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
---	---

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

$\backslash\text{token\_if\_eq\_charcode\_p:NN} \star$ $\backslash\text{token\_if\_eq\_charcode:NN}\underline{TF} \star$	$\backslash\text{token\_if\_eq\_catcode:NN}\underline{TF} \langle token_1 \rangle \langle token_2 \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
---	---



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?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
\l_peek_token
\g_peek_token
\l_peek_search_token
```

Some useful variables. Initially they are set to ?.

```
\peek_after:NN
\peek_gafter:NN \peek_after:NN <function><token>
```

Assign  $\langle token \rangle$  to  $\backslash l\_peek\_token$  and then run  $\langle function \rangle$  which should perform some sort of test on this token. Leaves  $\langle token \rangle$  in the input stream.  $\backslash peek\_gafter:NN$  does this globally to the token  $\backslash g\_peek\_token$ .

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

```
\peek_meaning:N $\overline{TF}$ 
\peek_meaning_ignore_spaces:N $\overline{TF}$ 
\peek_meaning_remove:N $\overline{TF}$ 
\peek_meaning_remove_ignore_spaces:N $\overline{TF}$  \peek_meaning:N $\overline{TF}$  <token> {<true>} {<false>}
```

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

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

```
\peek_charcode:N $\overline{TF}$ 
\peek_charcode_ignore_spaces:N $\overline{TF}$ 
\peek_charcode_remove:N $\overline{TF}$ 
\peek_charcode_remove_ignore_spaces:N $\overline{TF}$  \peek_charcode:N $\overline{TF}$  <token> {<true>} {<false>}
```

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

```
\peek_catcode:N $\overline{TF}$ 
\peek_catcode_ignore_spaces:N $\overline{TF}$ 
\peek_catcode_remove:N $\overline{TF}$ 
\peek_catcode_remove_ignore_spaces:N $\overline{TF}$  \peek_catcode:N $\overline{TF}$  <token> {<true>} {<false>}
```

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

<code>\peek_token_generic:NNTF</code> <code>\peek_token_remove_generic:NNTF</code>	<code>\peek_token_generic:NNTF &lt;token&gt;&lt;function&gt; {\true} {\false}</code>
---	--

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

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

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

## Part VIII

# The `l3int` package

## Integers/counters

$\text{\LaTeX}3$  maintains two type of integer registers for internal use. One (associated with the name `num`) for low level uses in the allocation mechanism using macros only and `int`: the one described here.

The `int` type uses the built-in counter registers of  $\text{\TeX}$  and is therefore relatively fast compared to the `num` type and should be preferred in all cases as there is little chance we should ever run out of registers when being based on at least  $\varepsilon\text{-TeX}$ .

## 37 Functions

<code>\int_new:N</code> <code>\int_new:c</code> <code>^^A\int_new_l:N</code>	<code>\int_new:N &lt;int&gt;</code>
--	-------------------------------------

Globally defines  $\langle int \rangle$  to be a new variable of type `int` although you can still choose if it should be an `\l_` or `\g_` type. There is no way to define constant counters with these functions.

**$\text{\TeX}$ hackers note:** `\int_new:N` is the equivalent to plain  $\text{\TeX}$ 's `\newcount`. However, the internal register allocation is done differently.

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

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

Increments  $\langle int \rangle$  by one. For global variables the global versions should be used.

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

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

Decrements  $\langle int \rangle$  by one. For global variables the global versions should be used.

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

`\int_set:Nn     $\langle int \rangle$  { $\langle integer\ expr \rangle$ }`

These functions will set the  $\langle int \rangle$  register to the  $\langle integer\ expr \rangle$  value. This value can contain simple calc-like expressions as provided by  $\epsilon$ -TeX.

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

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

These functions sets the  $\langle int \rangle$  register to zero either locally or globally.

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

`\int_add:Nn     $\langle int \rangle$  { $\langle integer\ expr \rangle$ }`

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

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

`\int_gsub:Nn     $\langle int \rangle$  { $\langle integer\ expr \rangle$ }`

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

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

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

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

**TeXhackers note:** The function `\int_use:N` could be implemented directly as the 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-explaining.

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

`\int_show:N <int>`

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

**T<sub>E</sub>Xhackers note:** The function `\int_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-explaining.

## 38 Formatting a counter value

<code>\int_to_arabic:n *</code> <code>\int_to_alph:n *</code> <code>\int_to_Alph:n *</code> <code>\int_to_roman:n *</code> <code>\int_to_Roman:n *</code> <code>\int_to_symbol:n *</code>
--

`\int_to_alph:n {<integer>}`  
`\int_to_alph:n <int>`

If some  $\langle integer \rangle$  or the the current value of a  $\langle int \rangle$  should be displayed or typeset in a special ways (e.g., as uppercase roman numerals) these function can be used. We need braces if the argument is a simple  $\langle integer \rangle$ , they can be omitted in case of a  $\langle int \rangle$ . By default the letters produced by `\int_to_roman:n` and `\int_to_Roman:n` have catcode 11.

All functions are fully expandable and will therefore produce the correct output when used inside of deferred writes, etc. In case the number in an `alph` or `Alph` function is greater than the default base number (26) it follows a simple conversion rule so that 27 is turned into `aa`, 50 into `ax` and so on and so forth. These two functions can be modified quite easily to take a different base number and conversion rule so that other languages can be supported.

**T<sub>E</sub>Xhackers note:** These are more or less the internal L<sup>A</sup>T<sub>E</sub>X2 functions `\@arabic`, `\@alph`, `\@Alph`, `\@roman`, `\@Roman`, and `\@fnsymbol` except that `\int_to_symbol:n` is also allowed outside math mode.

### 38.1 Internal functions

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

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

Converts  $\langle integer \rangle$  to it 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_to_number:w *</code>	<code>\int_to_number:w &lt;integer&gt; &lt;space&gt;</code>
---------------------------------	---

Converts `<integer>` to its numerical string. 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 `\number` 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</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_alph_default_conversion_rule:n</code>	
<code>\int_Alph_default_conversion_rule:n</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>

  

	<code>\int_alph_default_conversion_rule:n {\&lt;int&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.

## 39 Variable and constants

<code>\int_const:Nn</code>	<code>\int_const:Nn \c_&lt;value&gt; {\&lt;value&gt;}</code>
----------------------------	--

Defines an integer constant of a certain `<value>`. If the constant is negative or very large it internally uses an `<int>` register.

```

\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

```

Set of constants denoting useful values.

**T<sub>E</sub>Xhackers note:** Some of these constants have been available under L<sup>A</sup>T<sub>E</sub>X2 under names like `\m@ne`, `\z@`, `\@ne`, `\tw@`, `\thr@@`, etc.

```
\c_max_int
```

Constant that denote the maximum value which can be stored in an `⟨int⟩` register.

```

\l_tmpa_int
\l_tmpb_int
\l_tmpc_int
\g_tmpa_int
\g_tmpb_int

```

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

## 40 Conversion

```
\int_convert_from_base_ten:nn
```

```
\int_convert_from_base_ten:nn {⟨number⟩} {⟨base⟩}
```



Converts the base 10 number  $\langle number \rangle$  into its equivalent representation written in base  $\langle base \rangle$ . Expandable.

<code>\int_convert_to_base_ten:nn</code> <code>\int_convert_to_base_ten:c</code>
---

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

Converts the base  $\langle base \rangle$  number  $\langle number \rangle$  into its equivalent representation written in base 10.  $\langle number \rangle$  can consist of digits and ascii letters. Expandable.

## Part IX

# The l3num package

## Integers in macros

Instead of using counter registers for manipulation of integer values it is sometimes useful to keep such values in macros. For this L<sup>A</sup>T<sub>E</sub>X3 offers the type “num”.

One reason is the limited number of registers inside T<sub>E</sub>X. However, when using  $\varepsilon$ -T<sub>E</sub>X this is no longer an issue. It remains to be seen if there are other compelling reasons to keep this module.

It turns out there might be as with a  $\langle num \rangle$  data type, the allocation module can do its bookkeeping without the aid of  $\langle int \rangle$  registers.

## 41 Functions

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

`\num_new:N { $\langle num \rangle$ }`

Defines  $\langle num \rangle$  to be a new variable of type num (initialized to zero). There is no way to define constant counters with these functions.

<code>\num_incr:N</code> <code>\num_incr:c</code> <code>\num_gincr:N</code> <code>\num_gincr:c</code>
--

`\num_incr:N { $\langle num \rangle$ }`

Increments  $\langle num \rangle$  by one. For global variables the global versions should be used.

<code>\num_decr:N</code> <code>\num_decr:c</code> <code>\num_gdecr:N</code> <code>\num_gdecr:c</code>
--

`\num_decr:N { $\langle num \rangle$ }`

Decrements  $\langle num \rangle$  by one. For global variables the global versions should be used.

<code>\num_zero:N</code>
<code>\num_zero:c</code>
<code>\num_gzero:N</code>
<code>\num_gzero:c</code>

`\num_zero:N    ⟨num⟩`

Resets  $\langle num \rangle$  to zero. For global variables the global versions should be used.

<code>\num_set:Nn</code>
<code>\num_set:cn</code>
<code>\num_gset:Nn</code>
<code>\num_gset:cn</code>

`\num_set:Nn    ⟨num⟩ {⟨integer⟩}`

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

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

`\num_gset_eq:NN    ⟨num1⟩ ⟨num2⟩`

These functions will set the  $\langle num_1 \rangle$  register equal to  $\langle num_2 \rangle$ .

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

`\num_gset_eq:NN    ⟨num1⟩ ⟨num2⟩`

These functions will globally set the  $\langle num_1 \rangle$  register equal to  $\langle num_2 \rangle$ .

<code>\num_add:Nn</code>
<code>\num_add:cn</code>
<code>\num_gadd:Nn</code>
<code>\num_gadd:cn</code>

`\num_add:Nn    ⟨num⟩ {⟨integer⟩}`

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

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

`\num_use:N    ⟨num⟩`

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

**T<sub>E</sub>Xhackers note:** Since these  $\langle num \rangle$ s are implemented as macros, the function `\num_use:N` is effectively a noop and mainly there for consistency with similar functions in other modules.

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

`\num_show:N    ⟨num⟩`

This function pauses the compilation and displays the integer value kept in  $\langle num \rangle$  on the console output.

<code>\num_elt_count:n</code>	<code>\num_elt_count:n {⟨balanced text⟩}</code>
<code>\num_elt_count_prop:Nn</code>	<code>\num_elt_count_prop:Nn ⟨prop⟩ {⟨balanced text⟩}</code>

Discards their arguments and puts a +1 in the input stream. Used to count elements in a token list.

## 42 Formatting a counter value

See the `l3int` module for ways of doing this.

## 43 Variable and constants

<code>\c_max_register_num</code>	Maximum number of registers; possibly engine-specific.
----------------------------------	--

<code>\l_tmpa_num</code> <code>\l_tmpb_num</code> <code>\l_tmpc_num</code> <code>\g_tmpa_num</code> <code>\g_tmpb_num</code>	Scratch register for immediate use. They are not used by conditionals or predicate functions.
--	---

## 44 Primitive functions

<code>\if_num:w</code>	<code>\if_num:w ⟨number<sub>1</sub>⟩ ⟨rel⟩ ⟨number<sub>2</sub>⟩ ⟨true⟩ \else: ⟨false⟩ \fi:</code>
------------------------	---

Compare two numbers. It is recommended to use `\intexpr_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 `\ifnum`.

<code>\if_case:w</code> <code>\or:</code>	<code>\if_case:w ⟨number⟩ ⟨case<sub>0</sub>⟩ \or: ⟨case<sub>1</sub>⟩ \or: ... \else: ⟨default⟩ \fi:</code>
--	--

Chooses case `⟨number⟩`. If you wish to use negative numbers as well, you can offset them with `\intexpr_eval:n`.

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

## Part X

# The l3intexpr package

## Integer expressions

This module sets up evaluation of integer expressions and allows mixing `int` and `num` registers.

An integer expression is one that contains integers in the form of numbers, registers containing numbers, i.e., `int` and `num` registers plus constants like `\c_one`, standard operators `+`, `-`, `/` and `*` and parentheses to group sub-expressions.

### 45 Functions

`\intexpr_eval:n *`

`\intexpr_eval:n {<int expr>}`

The result of this expansion is a properly terminated *<number>*, i.e., one that can be used with `\if_case:w` and others. For example,

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

evaluates to `-6`. The result is returned after two expansions so if you find that you need to pass on the result to another function using the expansion engine, the recommendation is to use an `f` type expansion if in an expandable context or `x` otherwise.

`\intexpr_compare_p:n *`  
`\intexpr_compare:nTF *`

`\intexpr_compare_p:n {<int expr1> <rel> <int expr2>}`

Compares *<int expr<sub>1</sub>>* with *<int expr<sub>2</sub>>* using C-like relational operators, i.e.

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>

Both integer expressions are evaluated fully in the process. Note the syntax, which allows natural input in the style of

`\intexpr_compare_p:n {5+3 != \l_tmpb_int}`

`=` is added for the sake of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  users accustomed to using a single equal sign.

`\intexpr_compare_p:nNn *`  
`\intexpr_compare:nNnTF *`

`\intexpr_compare_p:nNn {<int expr1>}<rel>{<int expr2>}`

Compares *<int expr<sub>1</sub>>* with *<int expr<sub>2</sub>>* using one of the relations `=`, `>` or `<`. This is faster than the variant above but at the cost of requiring a little more typing and not supporting the extended set of relational operators. Note that if both expressions are normal integer variables as in

```
\intexpr_compare:nNnTF \l_temp_int < \c_zero {negative}{non-negative}
```

you can safely omit the braces.

<code>\intexpr_max:nn *</code>	<code>\intexpr_max:nn    {&lt;int expr<sub>1</sub>&gt;} {&lt;int expr<sub>2</sub>&gt;}</code>
<code>\intexpr_min:nn *</code>	

Return the largest or smallest of two integer expressions.

<code>\intexpr_abs:n *</code>	<code>\intexpr_abs:n    {&lt;int expr&gt;}</code>
-------------------------------	---

Return the numerical value of an integer expression.

<code>\intexpr_if_odd:nTF *</code>	<code>\intexpr_if_odd:nTF    {&lt;int expr&gt;} {&lt;true&gt;} {&lt;false&gt;}</code>
<code>\intexpr_if_odd_p:n *</code>	
<code>\intexpr_if_even:nTF *</code>	
<code>\intexpr_if_even_p:n *</code>	

These functions test if an integer expression is even or odd.

<code>\intexpr_div_truncate:nn *</code>	<code>\intexpr_div_truncate:n    {&lt;int expr&gt;} {&lt;int expr&gt;}</code>
<code>\intexpr_div_round:nn *</code>	
<code>\intexpr_mod:nn *</code>	

  

<code>\intexpr_mod:nn</code>	<code>\intexpr_mod:nn    {&lt;int expr&gt;} {&lt;int expr&gt;}</code>
------------------------------	---

If you want the result of a division to be truncated use `\intexpr_div_truncate:nn`. `\intexpr_div_round:nn` is added for completeness. `\intexpr_mod:nn` returns the remainder of a division.

## 46 Primitive functions

<code>\intexpr_value:w</code>	<code>\intexpr_value:w &lt;integer&gt;</code>
<code>\intexpr_value:w</code>	<code>\intexpr_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>\intexpr_eval:w</code>	<code>\intexpr_eval:w &lt;int expr&gt; \intexpr_eval_end:</code>
<code>\intexpr_eval_end:</code>	

Evaluates `<int expr>`. The evaluation stops when an unexpandable token of catcode other than 12 is reached or `\intexpr_end:` is read. The latter is gobbled by the scanner mechanism.

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

<code>\if_intexpr_compare:w</code>	<code>\if_intexpr_compare:w &lt;number&gt; &lt;rel&gt; &lt;number&gt; &lt;true&gt;</code> <code>\else: &lt;false&gt; \fi:</code>
------------------------------------	---

Compare two numbers. It is recommended to use `\intexpr_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 `\ifnum`.

<code>\if_intexpr_odd:w</code>	<code>\if_intexpr_odd:w &lt;number&gt; &lt;true&gt; \else: &lt;false&gt; \fi:</code>
--------------------------------	--

Execute `<true>` if `<number>` is odd, `<false>` otherwise.

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

<code>\if_intexpr_case:w</code> <code>\or:</code>	<code>\if_intexpr_case:w &lt;number&gt; &lt;case&gt; \or: &lt;case&gt; \or: ...</code> <code>\else:</code> <code>&lt;default&gt; \fi:</code>
--	--

Chooses case `<number>`. If you wish to use negative numbers as well, you can offset them with `\intexpr_eval:n`.

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

<code>\intexpr_while_do:nn</code> <code>\intexpr_until_do:nn</code> <code>\intexpr_do_while:nn</code> <code>\intexpr_do_until:nn</code>	<code>\intexpr_while_do:nn {&lt;int expr&gt; &lt;rel&gt; &lt;int expr&gt;} {&lt;code&gt;}</code>
--	--

`\intexpr_while_do:nn` tests the integer expressions against each other using a C-like `<rel>` as in `\intexpr_compare_p:n` and if true performs the `<code>` until the test fails. `\intexpr_do_while:nn` is similar but executes the `<code>` 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. They could be omitted as it is just a matter of switching the arguments in the test.

<code>\intexpr_while_do:nNnn</code> <code>\intexpr_until_do:nNnn</code> <code>\intexpr_do_while:nNnn</code> <code>\intexpr_do_until:nNnn</code>	<code>\intexpr_while_do:nNnn &lt;int expr&gt; &lt;rel&gt; &lt;int expr&gt; {&lt;code&gt;}</code>
--	--

Exactly as above but instead using the syntax of `\intexpr_compare_p:nNn`.

## Part XI

# 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**).

## 47 Skip registers

### 47.1 Functions

<pre>\skip_new:N \skip_new:c ^^A\skip_new_l:N</pre>
---

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

Defines  $\langle skip \rangle$  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`. However, the internal register allocation is done differently.

<pre>\skip_zero:N \skip_zero:c \skip_gzero:N \skip_gzero:c</pre>
--

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

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

<pre>\skip_set:Nn \skip_set:cn \skip_gset:Nn \skip_gset:cn</pre>
--

`\skip_set:Nn`  $\langle skip \rangle$   $\{ \langle skip \text{ value} \rangle \}$

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

<pre>\skip_add:Nn \skip_add:cn \skip_gadd:Nn \skip_gadd:cn</pre>
--

`\skip_add:Nn`  $\langle skip \rangle$   $\{ \langle length \rangle \}$

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.

<pre>\skip_sub:Nn \skip_gsub:Nn</pre>
---------------------------------------

`\skip_gsub:Nn`  $\langle skip \rangle$   $\{ \langle length \rangle \}$

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 `<skip>` 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 `<skip>` 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-explaining.

<code>\skip_horizontal:N</code>	<code>\skip_horizontal:N &lt;skip&gt;</code> <code>\skip_horizontal:n {&lt;length&gt;}</code>
<code>\skip_horizontal:c</code>	
<code>\skip_horizontal:n</code>	
<code>\skip_vertical:N</code>	
<code>\skip_vertical:c</code>	
<code>\skip_vertical:n</code>	

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

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

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

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

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN {&lt;skip&gt;} {&lt;action&gt;}</code>
	<code>&lt;dimen<sub>1</sub>&gt; &lt;dimen<sub>2</sub>&gt;</code>

Checks if `<skip>` contains finite glue. If it does then it assigns `<dimen1>` the stretch component and `<dimen2>` the shrink component. If it contains infinite glue set `<dimen1>` and `<dimen2>` to zero and execute `#2` which is usually an error or warning message of some sort.

<code>\skip_eval:n *</code>
-----------------------------

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

Evaluates the value of `<skip expr>` 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.



**T<sub>E</sub>Xhackers note:** This is the  $\varepsilon$ -T<sub>E</sub>X primitive `\glueexpr` turned into a function taking an argument.

## 47.2 Formatting a skip register value

## 47.3 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>	Set of constants denoting useful values.
---------------------------	--

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

# 48 Dim registers

## 48.1 Functions

<code>\dim_new:N</code> <code>\dim_new:c</code> <code>^^A\dim_new_l:N</code>	<code>\dim_new:N</code> $\langle dim \rangle$ Defines $\langle dim \rangle$ to be a new variable of type <code>dim</code> .
--	--

**T<sub>E</sub>Xhackers note:** `\dim_new:N` is the equivalent to plain T<sub>E</sub>X's `\newdimen`. However, the internal register allocation is done differently.

<code>\dim_zero:N</code> <code>\dim_zero:c</code> <code>\dim_gzero:N</code> <code>\dim_gzero:c</code>	<code>\dim_zero:N</code> $\langle dim \rangle$ Locally or globally reset $\langle dim \rangle$ to zero. For global variables the global versions should be used.
--	---

```

\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_add:Nn
\dim_add:Nc
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn

```

```
\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.

```

\dim_sub:Nn
\dim_sub:Nc
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

```

```
\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.

```

\dim_use:N
\dim_use:c

```

```
\dim_use:N <dim>
```

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

**T<sub>E</sub>Xhackers note:** The function `\dim_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.

```

\dim_show:N
\dim_show:c

```

```
\dim_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 `\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-explaining.

`\dim_eval:n` `\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.

`\if_dim:w` `\if_dim:w <dimen1> <rel> <dimen2> <true> \else: <false> \fi:`

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`.

`\dim_compare:nNnTF *` `\dim_compare:nNnTF {<dim expr> <rel> {<dim expr>}`  
`\dim_compare_p:nNn *` `{<true>} {<false>}`

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.

**T<sub>E</sub>Xhackers note:** This is the T<sub>E</sub>X primitive `\ifdim` turned into a function.

`\dim_while_do:nNnn`  
`\dim_until_do:nNnn`  
`\dim_do_while:nNnn`  
`\dim_do_until:nNnn` `\dim_while_do:nNnn <dim expr> <rel> <dim expr> <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.

## 48.2 Variable and constants

`\c_max_dim` Constant that denotes the maximum value which can be stored in a `<dim>` register.

`\c_zero_dim` Set of constants denoting useful values.

<code>\l_tmpa_dim</code>
<code>\l_tmpb_dim</code>
<code>\l_tmpc_dim</code>
<code>\l_tmpd_dim</code>
<code>\g_tmpa_dim</code>
<code>\g_tmpb_dim</code>

Scratch register for immediate use.

## 49 Muskip

<code>\muskip_new:N</code>
<code>^^A\muskip_new_l:N</code>

`\muskip_new:N`  $\langle muskip \rangle$   
 Defines  $\langle muskip \rangle$  to be a new variable of type `muskip`.

**T<sub>E</sub>Xhackers note:** `\muskip_new:N` is the equivalent to plain T<sub>E</sub>X's `\newmuskip`. However, the internal register allocation is done differently.

<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`.

## Part XII

# 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.

## 50 Functions

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

`\tl_new:Nn <tl var.> {<initial token list>}`

Defines `<tl var.>` 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_use:N</code>
<code>\tl_use:c</code>

`\tl_use:N <tl var.>`

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>

`\tl_show:N <tl var.>`  
`\tl_show:n {<token list>}`

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  $\langle token\ list\rangle$  to the left of  $\langle tl\ var.\rangle$ .  $\langle token\ list\rangle$  might be subject to expansion before assignment.

<code>\tl_put_right:Nn</code>
<code>\tl_put_right:NV</code>
<code>\tl_put_right:No</code>
<code>\tl_put_right:Nx</code>
<code>\tl_put_right:cn</code>
<code>\tl_put_right:cV</code>
<code>\tl_put_right:co</code>

`\tl_put_right:Nn <tl var.> {\<token list>}`

These functions append  $\langle token list \rangle$  to the right of  $\langle tl var. \rangle$ .

<code>\tl_gput_left:Nn</code>
<code>\tl_gput_left:No</code>
<code>\tl_gput_left:NV</code>
<code>\tl_gput_left:Nx</code>
<code>\tl_gput_left:cn</code>
<code>\tl_gput_left:co</code>
<code>\tl_gput_left:cV</code>

`\tl_gput_left:Nn <tl var.> {\<token list>}`

These functions will append  $\langle token list \rangle$  globally to the left of  $\langle tl var. \rangle$ .

<code>\tl_gput_right:Nn</code>
<code>\tl_gput_right:No</code>
<code>\tl_gput_right:NV</code>
<code>\tl_gput_right:Nx</code>
<code>\tl_gput_right:cn</code>
<code>\tl_gput_right:co</code>
<code>\tl_gput_right:cV</code>

`\tl_gput_right:Nn <tl var.> {\<token list>}`

These functions will globally append  $\langle token list \rangle$  to the right of  $\langle tl var. \rangle$ .

A word of warning is appropriate here: Token list variables are implemented as macros and as such currently inherit some of the peculiarities of how T<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.12`

Fast form for `\tl_set:No <tl var.12`

when  $\langle tl var. _2 \rangle$  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  $\langle tl\ var.\rangle$  as a string list with all characters catcoded to ‘other’.

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

`\tl_to_str:n {\langle token list\rangle}`

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 {\langle catcode setup\rangle} {\langle token list\rangle}`

Returns the result of re-tokenising  $\langle token\ list\rangle$  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:Nnx</code> <code>\tl_gset_rescan:Nnn</code> <code>\tl_gset_rescan:Nnx</code>
--

`\tl_set_rescan:Nnn <tl var.> {\langle catcode setup\rangle} {\langle token list\rangle}`

Sets  $\langle tl\ var.\rangle$  to the result of re-tokenising  $\langle token\ list\rangle$  with the catcode setup (and whatever other redefinitions) specified.

**TeXhackers note:** This is a wrapper around  $\varepsilon$ -TeX’s `\scantokens`.

## 51 Predicates and conditionals

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

`\tl_if_empty_p:N <tl var.>`

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>
--

`\tl_if_empty:NTF <tl var.> {\langle true code\rangle} {\langle false code\rangle}`

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>
--

`\tl_if_eq_p:NN <tl var.1> <tl var.2>`

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>

`\tl_if_eq:NNTF <tl var.1> <tl var.2> {\true code} {\false code}`

Execute  $\langle true\ code \rangle$  if  $\langle tl\ var.1 \rangle$  holds the same token list as  $\langle tl\ var.2 \rangle$  and  $\langle false\ code \rangle$  otherwise.

<code>\tl_if_eq:nnTF *</code>
<code>\tl_if_eq:nVTF *</code>
<code>\tl_if_eq:noTF *</code>
<code>\tl_if_eq:VnTF *</code>
<code>\tl_if_eq:onTF *</code>
<code>\tl_if_eq:VVTF *</code>
<code>\tl_if_eq:ooTF *</code>
<code>\tl_if_eq:xxTF *</code>
<code>\tl_if_eq:xnTF *</code>
<code>\tl_if_eq:nxTF *</code>
<code>\tl_if_eq:xVTF *</code>
<code>\tl_if_eq:xoTF *</code>
<code>\tl_if_eq:VxTF *</code>
<code>\tl_if_eq:oxTF *</code>

`\tl_if_eq:nnTF {\tlist_1} {\tlist_2} {\true code} {\false code}`

Execute  $\langle true\ code \rangle$  if the two token lists  $\langle tlist_1 \rangle$  and  $\langle tlist_2 \rangle$  are identical. These functions are expandable if a new enough version of pdfTeX is being used.

<code>\tl_if_eq_p:nn *</code>
<code>\tl_if_eq_p:nV *</code>
<code>\tl_if_eq_p:no *</code>
<code>\tl_if_eq_p:Vn *</code>
<code>\tl_if_eq_p:on *</code>
<code>\tl_if_eq_p:VV *</code>
<code>\tl_if_eq_p:oo *</code>
<code>\tl_if_eq_p:xx *</code>
<code>\tl_if_eq_p:xn *</code>
<code>\tl_if_eq_p:nx *</code>
<code>\tl_if_eq_p:xV *</code>
<code>\tl_if_eq_p:xo *</code>
<code>\tl_if_eq_p:Vx *</code>
<code>\tl_if_eq_p:ox *</code>

`\tl_if_eq_p:nn {\tlist_1} {\tlist_2}`

Predicates function which returns ‘true’ if the two token list are identical and ‘false’ otherwise. These are only defined if a new enough version of pdfTeX is in use.

<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>

`\tl_if_empty:nTF {\token list} {\true code} {\false 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_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_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`.

**TeXhackers note:** These are the TeX primitives `\lowercase` and `\uppercase` renamed.

## 52 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.

<code>\tl_map_inline:nn</code>
<code>\tl_map_inline:Nn</code>
<code>\tl_map_inline:cn</code>

`\tl_map_inline:nn {<token list>} {<inline function>}`  
`\tl_map_inline:Nn <tl var.> {<inline function>}`

Allows a syntax like `\tl_map_inline:nn {<token list>} {\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>

`\tl_map_variable:nNn {<token list>} <temp> {<action>}`  
`\tl_map_variable:NNn <tl var.> <temp> {<action>}`

Assigns *<temp>* to each element on *<token list>* and executes *<action>*. As there is an assignment in this process it is not expandable.

**TeXhackers 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>	<code>\tl_map_break:</code>
-----------------------------	-----------------------------

For breaking out of a loop. Must not be nested inside a primitive `\if` structure.

<code>\tl_reverse:n</code>	<code>\tl_reverse:n</code> $\{\langle token_1 \rangle \langle token_2 \rangle \dots \langle token_n \rangle\}$
<code>\tl_reverse:V</code>	
<code>\tl_reverse:o</code>	
<code>\tl_reverse:N</code>	<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  $\langle token_n \rangle \dots \langle token_2 \rangle \langle token_1 \rangle$ . Note that spaces in this token list are gobbled in the process.

Note also that braces are lost in the process of reversing a  $\langle tl\ var. \rangle$ . That is, `\tl_set:Nn \l_tmpa_tl {a{bcd}e} \tl_reverse:N \l_tmpa_tl` will result in `ebcda`. This behaviour is probably more of a bug than a feature.

<code>\tl_elt_count:n *</code>	<code>\tl_elt_count:n</code> $\{\langle token\ list \rangle\}$
<code>\tl_elt_count:V *</code>	
<code>\tl_elt_count:o *</code>	
<code>\tl_elt_count:N *</code>	<code>\tl_elt_count:N</code> $\langle tl\ var. \rangle$

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.

## 53 Variables and constants

<code>\c_job_name_tl</code>	Constant that gets the ‘job name’ assigned when $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ starts.
-----------------------------	---

**$\mathrm{T}_{\mathrm{E}}\mathrm{X}$ hackers note:** This is the new name for the primitive `\jobname`. It is a constant that is set by  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  and should not be overwritten by the package.

<code>\c_empty_tl</code>	Constant that is always empty.
--------------------------	--------------------------------

**$\mathrm{T}_{\mathrm{E}}\mathrm{X}$ hackers note:** This was named `\@empty` in  $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}2$  and `\empty` in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ .

<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.

<code>\l_tl_replace_toks</code>	Internal register used in the replace functions.
---------------------------------	--

<code>\l_testa_tl</code>
<code>\l_testb_tl</code>
<code>\g_testa_tl</code>
<code>\g_testb_tl</code>

Registers used for conditional processing if the engine doesn't support arbitrary string comparison.

<code>\g_tl_inline_level_num</code>
-------------------------------------

Internal register used in the inline map functions.

## 54 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>

`\tl_if_in:NnTF`  $\langle tl\ var.\rangle$   $\{\langle item\rangle\}$   $\{\langle true\ code\rangle\}$   $\{\langle false\ code\rangle\}$

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>	<code>\tl_remove_all_in:Nn &lt;tl var.&gt; {&lt;item&gt;}</code>
--	--

Removes *all* occurrences of  $\langle item \rangle$  from  $\langle tl var. \rangle$ . Note that  $\langle item \rangle$  cannot contain brace groups nor  $\#_6$  tokens.

## 55 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>\tl_tail:n</code> <code>\tl_tail:f</code> <code>\tl_head_i:n</code> <code>\tl_head_iii:n</code> <code>\tl_head_iii:f</code> <code>\tl_head:w</code> <code>\tl_tail:w</code> <code>\tl_head_i:w</code> <code>\tl_head_iii:w</code>	<code>\tl_head:n { &lt;token<sub>1</sub>&gt;&lt;token<sub>2</sub>&gt;...&lt;token<sub>n</sub>&gt; }</code> <code>\tl_tail:n { &lt;token<sub>1</sub>&gt;&lt;token<sub>2</sub>&gt;...&lt;token<sub>n</sub>&gt; }</code> <code>\tl_head:w &lt;token<sub>1</sub>&gt;&lt;token<sub>2</sub>&gt;...&lt;token<sub>n</sub>&gt; \q_nil</code>
--	---

These functions return either the head or the tail of a list, thus in the above example `\tl_head:n` would return  $\langle token_1 \rangle$  and `\tl_tail:n` would return  $\langle token_2 \rangle \dots \langle token_n \rangle$ . `\tl_head_iii:n` returns the first three tokens. The `:w` versions require some care as they use a delimited argument internally.

**T<sub>E</sub>Xhackers 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>\tl_if_head_eq_meaning:nNTF *</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  $\langle true \rangle$  if the first token in  $\langle token list \rangle$  is equal to  $\langle token \rangle$  and  $\langle false \rangle$  otherwise. The `meaning` version compares the two tokens with `\if_meaning:w`.

<code>\tl_if_head_eq_charcode_p:nN *</code> <code>\tl_if_head_eq_charcode_p:fN *</code> <code>\tl_if_head_eq_charcode:nNTF *</code> <code>\tl_if_head_eq_charcode:fNTF *</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  $\langle true \rangle$  if the first token in  $\langle token list \rangle$  is equal to  $\langle token \rangle$  and  $\langle false \rangle$  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>\tl_if_head_eq_catcode:nNTF *</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 XIII

# 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 *<toks>* (token registers) and *<tl var.>* (token list variable) is their behavior regarding expansion. While *<tl vars>* expand fully (i.e., until only unexpandable tokens are left) inside an argument that is subject to expansion (i.e., denoted by *x*) *<toks>*'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 *<token list>* 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 *<toks>* can contain the `#` character (i.e., characters of catcode 6), whereas a *<tl var.>* will require its input to be sanitised before that is possible.

If you're not sure which to use between a *<tl var.>* or a *<toks>*, 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 *<toks>* (e.g., `l3prop` uses *<toks>* to store its property lists).

If you're storing ad-hoc data for later use (possibly from direct user input) then usually a *<tl var.>* will be what you want.

## 56 Allocation and use

<code>\toks_new:N</code> <code>\toks_new:c</code> <code>^^A\toks_new_l:N</code>	<code>\toks_new:N &lt;toks&gt;</code>
---	---------------------------------------

Defines *<toks>* 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 <toks>`

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  $\langle toks \rangle$  to hold the token list  $\langle token list \rangle$ .

**T<sub>E</sub>Xhackers note:** `\toks_set:Nn` could have been specified in plain T<sub>E</sub>X by  $\langle toks \rangle = \{\langle token list \rangle\}$  but all other functions have no counterpart in plain T<sub>E</sub>X. Additionally the functions above the global variants described below will check for correct local and global assignments, something that isn't available 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  $\langle toks \rangle$  to globally hold the token list  $\langle token list \rangle$ .

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

`\toks_set_eq:NN <toks1> <toks2>`

Set  $\langle toks_1 \rangle$  to the value of  $\langle toks_2 \rangle$ . Don't try to use `\toks_set:Nn` for this purpose if the second argument is also a token register.

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

`\toks_gset_eq:NN <toks1> <toks2>`

The  $\langle toks_1 \rangle$  globally set to the value of  $\langle toks_2 \rangle$ . Don't try to use `\toks_gset:Nn` for this purpose if the second argument is also a token register.

<code>\toks_clear:N</code>
<code>\toks_clear:c</code>
<code>\toks_gclear:N</code>
<code>\toks_gclear:c</code>

`\toks_clear:N <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.

**T<sub>E</sub>Xhackers note:** Something like `\showthe <toks>`.

## 57 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.

## 58 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.

## 59 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 XIV

# 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.

## 60 Functions for creating/initialising sequences

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

`\seq_new:N <sequence>`

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>

`\seq_clear:N <sequence>`

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>

`\seq_clear_new:N <sequence>`

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>

`\seq_set_eq:NN <seq1> <seq2>`

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>

`\seq_gset_eq:NN <seq1> <seq2>`

Function that globally makes  $\langle seq_1 \rangle$  identical to  $\langle seq_2 \rangle$ .

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

`\seq_gconcat:NNN <seq1> <seq2> <seq3>`

Function that concatenates  $\langle seq_2 \rangle$  and  $\langle seq_3 \rangle$  and globally assigns the result to  $\langle seq_1 \rangle$ .

## 61 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>

`\seq_put_left:Nn <sequence> <token list>`

Locally appends *<token list>* as a single item to the left of *<sequence>*. *<token list>* 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>

`\seq_put_right:Nn <sequence> <token list>`

Locally appends *<token list>* as a single item to the right of *<sequence>*. *<token list>* 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>

`\seq_gput_left:Nn <sequence> <token list>`

Globally appends *<token list>* as a single item to the left of *<sequence>*.

<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 <sequence> <token list>`

Globally appends *<token list>* as a single item to the right of *<sequence>*.

<code>\seq_gput_right:Nc</code>
---------------------------------

Variant of the above used in the `xor` package. Will probably be moved soon to that package. (Sep 2008)

## 62 Working with sequences

<code>\seq_get:NN</code> <code>\seq_get:cN</code>	<code>\seq_get:NN &lt;sequence&gt; &lt;tl var.&gt;</code>
--	---

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 <sequence> \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.

<code>\seq_map_variable:NNn</code> <code>\seq_map_variable:cNn</code>	<code>\seq_map_variable:NNn &lt;sequence&gt; &lt;tl var.&gt; {&lt;code using tl var.&gt;}</code>
--	--

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>	<code>\seq_map_function:NN &lt;sequence&gt; &lt;function&gt;</code>
--	---

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>	<code>\seq_map_inline:Nn &lt;sequence&gt; {&lt;inline function&gt;}</code>
--	--

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_show:N</code> <code>\seq_show:c</code>	<code>\seq_show:N &lt;sequence&gt;</code>
--	---

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 <sequence>`

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 <seq>`

Function that removes any duplicate entries in `<seq>`.

## 63 Predicates and conditionals

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

`\seq_if_empty_p:N <sequence>`

This predicate returns ‘true’ if `<sequence>` is ‘empty’ i.e., doesn’t contain any items. Note that this is ‘false’ even if the `<sequence>` only contains a single empty item.

<code>\seq_if_empty:NTF</code>
<code>\seq_if_empty:cTF</code>

`\seq_if_empty:NTF <sequence> {\true code} {\false code}`

Set of conditionals that test whether or not a particular `<sequence>` is empty and if so executes either `<true code>` or `<false code>`.

<code>\seq_if_in:NnTF</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 `<item>` is in `<sequence>`. Depending on the result either `<true code>` or `<false code>` is executed.

## 64 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 `<sequence>` 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 `<sequence>` to `<tl var.>` using `<assign1>` and assigns the tail to `<sequence>` using `<assign2>`. 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_map_break:</code>
<code>\seq_map_break:n</code>

Functions used to implement mapping 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.

## 65 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.)

<code>\seq_push:Nn</code>
<code>\seq_push:NV</code>
<code>\seq_push:No</code>
<code>\seq_push:cn</code>
<code>\seq_gpush:Nn</code>
<code>\seq_gpush:NV</code>
<code>\seq_gpush:No</code>
<code>\seq_gpush:Nv</code>
<code>\seq_gpush:cn</code>

`\seq_push:Nn`  $\langle stack \rangle$   $\{ \langle token list \rangle \}$

Locally or globally pushes  $\langle token list \rangle$  as a single item onto the  $\langle stack \rangle$ .

<code>\seq_pop:NN</code>
<code>\seq_pop:cN</code>
<code>\seq_gpop:NN</code>
<code>\seq_gpop:cN</code>

`\seq_pop:NN`  $\langle stack \rangle$   $\langle tl var. \rangle$

Functions that assign the top item of  $\langle stack \rangle$  to  $\langle tl var. \rangle$  and removes it from  $\langle stack \rangle$ !

<code>\seq_top:NN</code>
<code>\seq_top:cN</code>

`\seq_top:NN`  $\langle stack \rangle$   $\langle tl var. \rangle$

Functions that locally assign the top item of  $\langle stack \rangle$  to the  $\langle tl var. \rangle$ . Item is *not* removed from  $\langle stack \rangle$ !

## Part XV

# The l3clist package

## Comma separated lists

L<sup>A</sup>T<sub>E</sub>X3 implements a data type called ‘clist (comma-lists)’. 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 comma-list might contain any tokens except for commas that are used to structure comma-lists (braces are need if commas are part of the value). It is also possible to map functions on such comma-lists so that they are executed for every item of the comma-list.

All functions that return items from a comma-list 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 comma-list can currently be done with either something like `\clist_put_left:Nn` or with a “stack” function like `\clist_push:Nn` which has the same effect. Maybe one should therefore remove the “left” functions totally.

## 66 Functions for creating/initialising comma-lists

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

`\clist_new:N \langle comma-list\rangle`

Defines  $\langle comma-list\rangle$  to be a variable of type clist.

<code>\clist_clear:N</code>
<code>\clist_clear:c</code>
<code>\clist_gclear:N</code>
<code>\clist_gclear:c</code>

`\clist_clear:N \langle comma-list\rangle`

These functions locally or globally clear  $\langle comma-list\rangle$ .

<code>\clist_clear_new:N</code>
<code>\clist_clear_new:c</code>
<code>\clist_gclear_new:N</code>
<code>\clist_gclear_new:c</code>

`\clist_clear_new:N \langle comma-list\rangle`

These functions locally or globally clear  $\langle comma-list\rangle$  if it exists or otherwise allocates it.



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

`\clist_set_eq:NN <clist1> <clist2>`  
Function that locally makes  $\langle clist_1 \rangle$  identical to  $\langle clist_2 \rangle$ .

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

`\clist_gset_eq:NN <clist1> <clist2>`  
Function that globally makes  $\langle clist_1 \rangle$  identical to  $\langle clist_2 \rangle$ .

## 67 Putting data in

<code>\clist_put_left:Nn</code>
<code>\clist_put_left:NV</code>
<code>\clist_put_left:No</code>
<code>\clist_put_left:Nx</code>
<code>\clist_put_left:cn</code>
<code>\clist_put_left:co</code>
<code>\clist_put_left:cV</code>

`\clist_put_left:Nn <comma-list> <token list>`  
Locally appends  $\langle token list \rangle$  as a single item to the left of  $\langle comma-list \rangle$ .  $\langle token list \rangle$  might get expanded before appending according to the variant used.

<code>\clist_put_right:Nn</code>
<code>\clist_put_right:No</code>
<code>\clist_put_right:NV</code>
<code>\clist_put_right:Nx</code>
<code>\clist_put_right:cn</code>
<code>\clist_put_right:co</code>
<code>\clist_put_right:cV</code>

`\clist_put_right:Nn <comma-list> <token list>`  
Locally appends  $\langle token list \rangle$  as a single item to the right of  $\langle comma-list \rangle$ .  $\langle token list \rangle$  might get expanded before appending according to the variant used.

<code>\clist_gput_left:Nn</code>
<code>\clist_gput_left:NV</code>
<code>\clist_gput_left:No</code>
<code>\clist_gput_left:Nx</code>
<code>\clist_gput_left:cn</code>
<code>\clist_gput_left:cV</code>
<code>\clist_gput_left:co</code>

`\clist_gput_left:Nn <comma-list> <token list>`  
Globally appends  $\langle token list \rangle$  as a single item to the right of  $\langle comma-list \rangle$ .

<code>\clist_gput_right:Nn</code>
<code>\clist_gput_right:NV</code>
<code>\clist_gput_right:No</code>
<code>\clist_gput_right:Nx</code>
<code>\clist_gput_right:cn</code>
<code>\clist_gput_right:cV</code>
<code>\clist_gput_right:co</code>

`\clist_gput_right:Nn <comma-list> <token list>`

Globally appends  $\langle token list \rangle$  as a single item to the right of  $\langle comma-list \rangle$ .

## 68 Getting data out

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

`\clist_use:N <clist>`

Function that inserts the  $\langle clist \rangle$  into the processing stream. Mainly useful if one knows what the  $\langle clist \rangle$  contains, e.g., for displaying the content of template parameters.

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

`\clist_show:N <clist>`

Function that pauses the compilation and displays  $\langle clist \rangle$  in the terminal output and in the log file. (Usually used for diagnostic purposes.)

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

`\clist_display:N <clist>`

As with `\clist_show:N` but pretty prints the output one line per element.

<code>\clist_get:NN</code>
<code>\clist_get:cN</code>

`\clist_get:NN <comma-list> <tl var.>`

Functions that locally assign the left-most item of  $\langle comma-list \rangle$  to the token list variable  $\langle tl var. \rangle$ . Item is not removed from  $\langle comma-list \rangle$ ! 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.

## 69 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  $\langle clist \rangle$ . 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.

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

This function applies  $\langle function \rangle$  (which must be a function with one argument) to every item of  $\langle comma-list \rangle$ .  $\langle function \rangle$  is not executed within a sub-group so that side effects can be achieved locally. The operation is expandable which means that it can be used within write operations etc.

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

Applies  $\langle inline function \rangle$  (which should be the direct coding for a function with one argument (i.e. use `##1` as the placeholder for this argument)) to every item of  $\langle comma-list \rangle$ .  $\langle inline function \rangle$  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. These functions can be nested.

<code>\clist_map_variable:NNn</code> <code>\clist_map_variable:cNn</code> <code>\clist_map_variable:nNn</code>	<code>\clist_map_variable:NNn</code> $\langle comma-list \rangle$ $\langle temp-var \rangle$ $\{ \langle action \rangle \}$
--	---

Assigns  $\langle temp-var \rangle$  to each element in  $\langle clist \rangle$  and then executes  $\langle action \rangle$  which should contain  $\langle temp-var \rangle$ . As the operation performs an assignment, it is not expandable.

**TeXhackers 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.

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

For breaking out of a loop. To be used inside TF type functions as in the example below.

```
\cs_new_nopar:Npn \test_function:n #1 {
  \intexpr_compare:nTF {#1 > 3} {\clist_map_break:}{‘‘#1’’}
}
\clist_map_function:nN {1,2,3,4,5,6,7,8}\test_function:n
```

This would return ‘‘1’’‘‘2’’‘‘3’’.

## 70 Predicates and conditionals

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

`\clist_if_empty_p:N <comma-list>`

This predicate returns ‘true’ if  $\langle comma-list \rangle$  is ‘empty’ i.e., doesn’t contain any tokens.

<code>\clist_if_empty:NTF *</code> <code>\clist_if_empty:cTF *</code>
--

`\clist_if_empty:NTF <comma-list> {<true code>} {<false code>}`

Set of conditionals that test whether or not a particular  $\langle comma-list \rangle$  is empty and if so executes either  $\langle true code \rangle$  or  $\langle false code \rangle$ .

<code>\clist_if_eq_p:NN *</code> <code>\clist_if_eq_p:cN *</code> <code>\clist_if_eq_p:Nc *</code> <code>\clist_if_eq_p:cc *</code>
--

`\clist_if_eq_p:N <comma-list1> <comma-list2>`

This predicate returns ‘true’ if the two comma lists are identical.

<code>\clist_if_eq:NNTF *</code> <code>\clist_if_eq:cNTF *</code> <code>\clist_if_eq:NcTF *</code> <code>\clist_if_eq:ccTF *</code>
--

`\clist_if_eq:NNTF <comma-list1> <comma-list2> {<true code>} {<false code>}`

Check if  $\langle comma-list_1 \rangle$  and  $\langle comma-list_2 \rangle$  are equal and execute either  $\langle true code \rangle$  or  $\langle false code \rangle$  accordingly.

<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>
--

`\clist_if_in:NnTF <comma-list> {<item>} {<true code>} {<false code>}`

Function that tests if  $\langle item \rangle$  is in  $\langle comma-list \rangle$ . Depending on the result either  $\langle true code \rangle$  or  $\langle false code \rangle$  is executed.

## 71 Higher level functions

<code>\clist_concat:NNN</code> <code>\clist_concat:ccc</code> <code>\clist_gconcat:NNN</code> <code>\clist_gconcat:ccc</code>
--

`\clist_gconcat:NNN <clist1> <clist2> <clist3>`

Function that concatenates  $\langle clist_2 \rangle$  and  $\langle clist_3 \rangle$  and locally or globally assigns the result to  $\langle clist_1 \rangle$ .

<code>\clist_remove_duplicates:N</code> <code>\clist_gremove_duplicates:N</code>
---

`\clist_gremove_duplicates:N <clist>`

Function that removes any duplicate entries in  $\langle clist \rangle$ .

<code>\clist_remove_element:Nn</code> <code>\clist_gremove_element:Nn</code>
---

`\clist_gremove_element:Nn <clist> <element>`

Function that removes  $\langle element \rangle$  from  $\langle clist \rangle$ , if present.

**T<sub>E</sub>Xhackers note:** This is similar in concept to `\@removeelement`, except that the syntax is clearer and the initial and final lists have the same name automatically.

## 72 Functions for ‘comma-list stacks’

Special comma-lists 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 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  $\langle stack \rangle$ !

## 73 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 *<comma-list>* 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 *<comma-list>* to *<tl var.>* using *<assign<sub>1</sub>>* and assigns the tail to *<comma-list>* using *<assign<sub>2</sub>>*. 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 XVI

# 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.

## 74 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>*.

```

\prop_put:Nnn
\prop_put:NnV
\prop_put:cnn
\prop_gput:Nnn
\prop_gput:Nno
\prop_gput:NnV
\prop_gput:Nnx
\prop_gput:cnn
\prop_gput:ccx

```

```
\prop_put:Nnn <prop> {<key>} {<token list>}
```

Locally or globally associates  $\langle token\ list \rangle$  with  $\langle key \rangle$  in the  $\langle prop \rangle$   $\langle prop \rangle$ . If  $\langle key \rangle$  has already a meaning within  $\langle prop \rangle$  this value is overwritten.

The  $\langle key \rangle$  must not contain unescaped # tokens but the  $\langle token\ list \rangle$  may.

```
\prop_gput_if_new:Nnn
```

```
\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  $\langle prop \rangle$ .

```

\prop_get:NnN
\prop_get:cnN
\prop_gget:NnN
\prop_gget:cnN

```

```
\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  $\backslash q\_no\_value$ . The assignment is done either locally or globally.

```

\prop_set_eq:NN
\prop_set_eq:cN
\prop_set_eq:Nc
\prop_set_eq:cc
\prop_gset_eq:NN
\prop_gset_eq:cN
\prop_gset_eq:Nc
\prop_gset_eq:cc

```

```
\prop_set_eq:NN <prop1> <prop2>
```

A fast assignment of  $\langle prop \rangle$ s.

```
\prop_get_gdel:NnN
```

```
\prop_get_gdel:NnN <prop> {<key>} <tl var.>
```

Like  $\backslash prop\_get:NnN$  but additionally removes  $\langle key \rangle$  (and its  $\langle info \rangle$ ) from  $\langle prop \rangle$ .

```
\prop_del:Nn
```

```
\prop_gdel:Nn
```

```
\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:NN &lt;prop&gt; &lt;function&gt;</code>
<code>\prop_map_function:cN *</code>	
<code>\prop_map_function:Nc *</code>	
<code>\prop_map_function:cc *</code>	

Maps *<function>* which should be a function with two arguments (*<key>* and *<info>*) over every *<key>* *<info>* pair of *<prop>*. Expandable.

<code>\prop_map_inline:Nn</code>	<code>\prop_map_inline:Nn &lt;prop&gt; {\inline function}</code>
<code>\prop_map_inline:cn</code>	

Just like `\prop_map_function:NN` but with the function of two arguments supplied as inline code. Within *<inline function>* refer to the arguments via #1 (*<key>*) and #2 (*<info>*). Nestable.

<code>\prop_map_break:</code>	<code>\prop_map_inline:Nn &lt;prop&gt; {</code> <code>... \&lt;break test&gt;:T {\prop_map_break:} }</code>
-------------------------------	--

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:N &lt;prop&gt;</code>
<code>\prop_show:c</code>	

Pauses the compilation and shows *<prop>* on the terminal output and in the log file.

<code>\prop_display:N</code>	<code>\prop_display:N &lt;prop&gt;</code>
<code>\prop_display:c</code>	

As with `\prop_show:N` but pretty prints the output one line per property pair.

## 75 Predicates and conditionals

<code>\prop_if_empty_p:N</code>	<code>\prop_if_empty_p:N &lt;prop&gt; {\true code} {\false code}</code>
<code>\prop_if_empty_p:c</code>	

Predicates to test whether or not a particular *<prop>* is empty.

<code>\prop_if_empty:NTF *</code>	<code>\prop_if_empty:NTF &lt;prop&gt; {\true code} {\false code}</code>
<code>\prop_if_empty:cTF *</code>	

Set of conditionals that test whether or not a particular *<prop>* is empty.

<code>\prop_if_eq_p:NN *</code>	<code>\prop_if_eq:NMF &lt;prop<sub>12 </sub></code>
<code>\prop_if_eq_p:cN *</code>	
<code>\prop_if_eq_p:Nc *</code>	
<code>\prop_if_eq_p:cc *</code>	
<code>\prop_if_eq:NNTF *</code>	
<code>\prop_if_eq:cNTF *</code>	
<code>\prop_if_eq:NcTF *</code>	
<code>\prop_if_eq:NcTF *</code>	
<code>\prop_if_eq:ccTF *</code>	
<code>\prop_if_eq:ccTF *</code>	



Execute  $\langle false\ code \rangle$  if  $\langle prop_1 \rangle$  doesn't hold the same token list as  $\langle prop_2 \rangle$ . Only expandable for new versions of pdfTeX.

$\backslash prop\_if\_in:NnTF$ $\backslash prop\_if\_in:NVTF$ $\backslash prop\_if\_in:NoTF$ $\backslash prop\_if\_in:cnTF$ $\backslash prop\_if\_in:ccTF$	$\backslash prop\_if\_in:NnTF\ \langle prop \rangle\ \{\langle key \rangle\}\ \{\langle true\ code \rangle\}\ \{\langle false\ code \rangle\}$ Tests if $\langle key \rangle$ is used in $\langle prop \rangle$ and then either executes $\langle true\ code \rangle$ or $\langle false\ code \rangle$ .
--	---

## 76 Internal functions

$\backslash q\_prop$	Quark used to delimit property lists internally.
----------------------	--

$\backslash prop\_put\_aux:w$ $\backslash prop\_put\_if\_new\_aux:w$	Internal functions implementing the put operations.
---	---

$\backslash prop\_get\_aux:w$ $\backslash prop\_gget\_aux:w$ $\backslash prop\_get\_del\_aux:w$ $\backslash prop\_del\_aux:w$	Internal functions implementing the get and delete operations.
--	--

$\backslash prop\_if\_in\_aux:w$	Internal function implementing the key test operation.
----------------------------------	--

$\backslash prop\_map\_function\_aux:w$	Internal function implementing the map operations.
---	--

$\backslash g\_prop\_inline\_level\_num$	Fake integer used in internal name for function used inside $\backslash prop\_map\_inline:NN$ .
--	---

$\backslash prop\_split\_aux:Nnn$	$\backslash prop\_split\_aux:Nnn\ \langle prop \rangle\ \langle key \rangle\ \langle cmd \rangle$ 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 $\backslash q\_no\_value$ and the 3rd arg is empty; otherwise the 3rd argument has the two extra tokens $\langle key \rangle\ \backslash q\_no\_value$ at the end. This function is used to implement various get operations.
-----------------------------------	--

## Part XVII

# The l3io package

# Low-level file i/o

TeX is capable of reading from and writing up to 16 individual streams. These i/o operations are accessible in L<sup>A</sup>TeX3 with functions from the `\io..` modules. In most cases it will be sufficient for the programmer to use the functions provided by the auxiliary file module, but here are the necessary functions for manipulating private streams.

Sometimes it is not known beforehand how much text is going to be written with a single call. As a result some internal TeX buffer may overflow. To avoid this kind of problem, L<sup>A</sup>TeX3 maintains beside direct write operations like `\iow_now:Nx` also so called “long” writes where the output is broken into individual lines on every blank in the text to be written. The resulting files are difficult to read for humans but since they usually serve only as internal storage this poses no problem.

Beside the functions that immediately act (e.g., `\iow_now:Nx`, etc.) we also have deferred operations that are saved away until the next page is finished. This allows to expand the `<tokens>` at the right time to get correct page numbers etc.

## 77 Functions for output streams

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

`\iow_new:N <stream>`

Defines `<stream>` to be a new identifier denoting an output stream for use in subsequent functions.

**TeXhackers note:** `\iow_new:N` corresponds to the plain TeX `\newwrite` allocation routine.

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

`\iow_open:Nn <stream> {<file name>}`

Opens output stream `<stream>` to write to `<file name>`. The output stream is immediately available for use. If the `<stream>` was already used as an output stream to some other file, this file gets closed first.<sup>6</sup> Also, all output streams still open at the end of the TeX run will be automatically closed.

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

`\iow_close:N \iow_open:Nn <stream>`

Closes output stream `<stream>`.

### 77.1 Immediate writing

<code>\iow_now:Nx</code>
<code>\iow_now:Nn</code>

`\iow_now:Nx <stream> {<tokens>}`

`\iow_now:Nx` immediately writes the expansion of `<tokens>` to the output stream `<stream>`.

---

<sup>6</sup>This is a precaution since on some OS it is possible to open the same file for output more than once which then results in some internal errors at the end of the run.

If  $\langle stream \rangle$  is not open output goes to the terminal. The variant `\iow_now:Nn` writes out  $\langle tokens \rangle$  without any further expansion.

**T<sub>E</sub>Xhackers note:** These are the equivalent of T<sub>E</sub>X's `\immediate\write` with and without expansion control.

<code>\iow_log:n</code>
<code>\iow_log:x</code>
<code>\iow_term:x</code>
<code>\iow_term:n</code>

`\iow_log:x { $\langle tokens \rangle$ }`

These functions write to the log file (also known as the transcript file) or to the terminal respectively. They are equivalent to `\iow_now:Nx` where  $\langle stream \rangle$  is the transcript file (`\c_iow_log_stream`) or the terminal (`\c_iow_term_stream`).

<code>\iow_now_buffer_safe:Nn</code>
<code>\iow_now_buffer_safe:Nx</code>

`\iow_now_buffer_safe:Nn  $\langle stream \rangle$  { $\langle tokens \rangle$ }`

Like `\iow_now:Nx` but splits  $\langle tokens \rangle$  at every blank into separate lines. When potentially (very) long token lists are being written, this avoids the problem of buffer overflow when reading back the file.

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

`\iow_now_when_avail:Nn  $\langle stream \rangle$  { $\langle tokens \rangle$ }`

This special function first checks if the  $\langle stream \rangle$  is open for writing. If not it does nothing otherwise it behaves like `\iow_now:Nn`.

## 77.2 Deferred writing

<code>\iow_shipout:Nn</code>
<code>\iow_shipout:Nx</code>

`\iow_shipout_x:Nn  $\langle stream \rangle$  { $\langle tokens \rangle$ }`

These functions write  $\langle tokens \rangle$  to the specified  $\langle stream \rangle$ , but unlike with `\iow_now:Nn` which write the material immediately, `\iow_shipout:Nn` waits until the processing of the current page has finished.

<code>\iow_shipout_x:Nn</code>
<code>\iow_shipout_x:Nx</code>

`\iow_shipout_x:Nn  $\langle stream \rangle$  { $\langle tokens \rangle$ }`

More useful than the `\iow_shipout:Nn` functions, these functions also write  $\langle tokens \rangle$  to  $\langle stream \rangle$  at the end of the processing of the current page; however, before  $\langle tokens \rangle$  are written they are subjected to a further `x` expansion stage.

This is useful for writing messages that depend on counters (such as the page number) that are not known until the page has been completed.

`\iow_shipout_x:Nn` always needs `{}` around the second argument.

**T<sub>E</sub>Xhackers note:** `\iow_shipout_x:Nn` is equivalent to T<sub>E</sub>X's `\write`.

## 77.3 Special characters for writing

`\iow_newline:` `\iow_newline:`

Function that produces a new line when used within the *⟨token list⟩* that gets written some output stream in non-verbatim mode.

`\iow_space:` `\iow_space:`

Function that produces a space when used within the *⟨token list⟩* that gets written to some output stream in an expanding mode.

`\iow_char:N` `\iow_space:N \⟨char⟩`  
`\iow_space:N \%`

Inserts *⟨char⟩* into the output stream. Useful when trying to write difficult characters such as %, {, }, etc., in messages.

## 78 Functions for input streams

`\ior_new:N` `\ior_new:N ⟨stream⟩`

This function defines *⟨stream⟩* to be a new input stream constant.

**T<sub>E</sub>Xhackers note:** This is the new name and new implementation for plain T<sub>E</sub>X's `\newread`.

`\ior_open:Nn` `\ior_open:Nn ⟨stream⟩ {⟨file name⟩}`

This function opens *⟨stream⟩* as an input stream for the external file *⟨file name⟩*. If *⟨file name⟩* doesn't exist or is an empty file the stream is considered to be fully read, a condition which can be tested with `\ior_if_eof:NTF` etc. If *⟨stream⟩* was already used to read from some other file this file will be closed first. The input stream is ready for immediate use.

`\ior_close:N` `\ior_close:N ⟨stream⟩`

This function closes the read stream *⟨stream⟩*.

**T<sub>E</sub>Xhackers note:** This is a new name for `\closein` but it is considered bad practice to make use of this knowledge :-)

`\ior_if_eof_p:N *`  
`\ior_if_eof:NTF *` `\ior_if_eof:NTF ⟨stream⟩ {⟨true code⟩} {⟨false code⟩}`

Conditional that tests if some input stream is fully read. The condition is also true if the input stream is not open.

<code>\if_eof:w</code>
------------------------

`\if_eof:w <stream> <true code> \else: <false code> \fi:`

**TeXhackers note:** This is the primitive `\ifeof` but we allow only a *<stream>* and not a plain number after it.

<code>\ior_to:NN</code>
<code>\ior_gto:NN</code>

`\ior_to:NN <stream> <tl var.>`

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input stream *<stream>* and places the result locally or globally into the token list variable. If *<stream>* is not open, input is requested from the terminal.

## 79 Constants

<code>\c_iow_comment_char</code>
<code>\c_iow_lbrace_char</code>
<code>\c_iow_rbrace_char</code>

Constants that can be used to represent comment character, left and right brace in token lists that should be written to a file.

<code>\c_iow_term_stream</code>
<code>\c_ior_term_stream</code>

Input or output stream denoting the terminal. If used as an input stream the user is prompted with the name of the token list variable (that is used in the call `\ior_to:NN` or `\ior_gto:NN`) followed by an equal sign. If you don't want an automatic prompt of this sort "misuse" `\c_iow_log_stream` as an input stream.

<code>\c_iow_log_stream</code>
<code>\c_ior_log_stream</code>

Output stream that writes only to the transcript file (e.g., the `.log` file on most systems). You may "misuse" this stream as an input stream. In this case it acts as a terminal stream without user prompting.

## Part XVIII

# 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.

## 80 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.

```
\msg_new:nnnnn
\msg_new:nnnn
\msg_new:nnn
\msg_set:nnnnn
\msg_set:nnnn
\msg_set:nnn
```

`\msg_new:nnnn <module> <name> <text> <more text> <code>`  
Creates new message *<name>* for *<module>* to produce *<text>* initially, *<more text>* if requested by the user and to insert *<code>* after the message when used as an error. *<text>* and *<more text>* can use up to two macro parameters (#1 and #2), which are supplied by the message system. Inside *<text>* and *<more text>* spaces are not ignored.

## 81 Message classes

Creating message output requires the message to be given a class.

```
\msg_class_new:nn
\msg_class_set:nn
```

`\msg_class_new:nn <class> <code>`  
Creates new *<class>* to output a message, using *<code>* to process the message text.

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.

```
\msg_fatal:nxxx
\msg_fatal:nxx
\msg_fatal:nn
```

`\msg_fatal:nxxx <module> <name> <arg one> <arg two>`  
Issues *<module>* error message *<name>*, passing *<arg one>* and *<arg two>* to the text-creating functions. The T<sub>E</sub>X run then halts.

```
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn
```

```
\msg_error:nnxx <module> <name> <arg one> <arg two>
```

Issues *<module>* error message *<name>*, passing *<arg one>* and *<arg two>* to the text-creating functions.

**TeXhackers note:** The standard output here is similar to `\PackageError`.

```
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn
```

```
\msg_warning:nnxx <module> <name> <arg one> <arg two>
```

Prints *<module>* message *<name>* to the terminal, passing *<arg one>* and *<arg two>* to the text-creating functions.

**TeXhackers note:** The standard output here is similar to `\PackageWarningNoLine`.

```
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn
```

```
\msg_info:nnxx <module> <name> <arg one> <arg two>
```

Prints *<module>* message *<name>* to the log, passing *<arg one>* and *<arg two>* to the text-creating functions.

**TeXhackers note:** The standard output here is similar to `\PackageInfoNoLine`.

```
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn
```

```
\msg_info:nnxx <module> <name> <arg one> <arg two>
```

Prints *<module>* message *<name>* to the log, passing *<arg one>* and *<arg two>* to the text-creating functions. No continuation text is added.

```
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nn
```

```
\msg_info:nnxx <module> <name> <arg one> <arg two>
```

Prints *<module>* message *<name>* to the log, passing *<arg one>* and *<arg two>* to the text-creating functions. No continuation text is added.

```
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn
```

```
\msg_none:nnxx <module> <name> <arg one> <arg two>
```

Does nothing: used for redirecting other message classes.

## 82 Redirecting messages

`\msg_redirect_class:nn` `\msg_redirect_class:nn` *<class one>* *<class two>*

Redirect all messages of *<class one>* to appear as those of *<class two>*.

`\msg_redirect_module:nnn` `\msg_redirect_module:nnn` *<class one>* *<module>* *<class two>*

Redirect *<module>* messages of *<class one>* to appear in *<class two>*.

**T<sub>E</sub>Xhackers 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 { trace } { module } { none }
```

`\msg_redirect_name:nnn` `\msg_redirect_name:nnn` *<module>* *<message>* *<class>*

Redirect *<module>* *<message>* to appear using *<class>*.

## 83 Support functions for output

`\msg_line_context:` `\msg_line_context:`

Prints the current line number preceded by `\c_msg_on_line_tl`.

`\msg_line_number:` `\msg_line_number:`

Prints the current line number.

`\msg_newline:`  
`\msg_two_newlines:` `\msg_newline:`

Print one or two newlines with no continuation information.

`\msg_space:`  
`\msg_two_spaces:`  
`\msg_four_spaces:` `\msg_space:`

Print one, two or four spaces: needed where a literal space would otherwise be gobbled by T<sub>E</sub>X.



## 84 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.

<pre> \msg_generic_new:nnnn \msg_generic_new:nnn \msg_generic_new:nn \msg_generic_set:nnnn \msg_generic_set:nnn \msg_generic_set:nn </pre>	<pre> \msg_generic_new:nnnn &lt;name&gt; &lt;text&gt; &lt;more text&gt; &lt;code&gt; </pre>
--	---

Creates new message *<name>* to produce *<text>* initially, *<more text>* if requested by the user and to insert *<code>* after the message when used as an error. *<text>* and *<more text>* can use up to two macro parameters (#1 and #2), which are supplied by the message system. Inside *<text>* and *<more text>* spaces are not ignored.

<pre> \msg_direct_interrupt:xxxxn </pre>	<pre> \msg_direct_interrupt:xxxxn &lt;first line&gt; &lt;text&gt; &lt;continuation&gt; &lt;more text&gt; &lt;code&gt; </pre>
--	--

Executes a T<sub>E</sub>X error, interrupting compilation. The *<first line>* is displayed followed by *<text>* and the input prompt. *<more text>* is displays if requested by the user, and *<code>* is inserted after the message. If *<more text>* is blank a default is supplied. Each line of *<text>* (broken with \\) begins with *<continuation>*.

<pre> \msg_direct_log:xx \msg_direct_term:xx </pre>	<pre> \msg_direct_log:xx &lt;text&gt; &lt;continuation&gt; </pre>
---	---

Prints *<text>* to either the log or terminal. New lines (broken with \\) start with *<continuation>*.

## 85 Kernel-specific functions

<pre> \msg_kernel_new:nnnn \msg_kernel_new:nnn \msg_kernel_new:nn \msg_kernel_set:nnnn \msg_kernel_set:nnn \msg_kernel_set:nn </pre>	<pre> \msg_kernel_new:nnn &lt;name&gt; &lt;text&gt; &lt;more text&gt; &lt;code&gt; </pre>
--	---

Creates new kernel message *<name>* to produce *<text>* initially, *<more text>* if requested by the user and to insert *<code>* after the message when used as an error. *<text>* and *<more text>* can use up to two macro parameters (#1 and #2), which are supplied by the message system.

<code>\msg_kernel_fatal:nxx</code> <code>\msg_kernel_fatal:nx</code> <code>\msg_kernel_fatal:n</code>	<code>\msg_kernel_fatal:nxx &lt;name&gt; &lt;arg one&gt; &lt;arg two&gt;</code>
---	---

Issues kernel error message  $\langle name \rangle$ , passing  $\langle arg one \rangle$  and  $\langle arg two \rangle$  to the text-creating functions. The TeX run then halts. Cannot be redirected.

<code>\msg_kernel_error:nxx</code> <code>\msg_kernel_error:nx</code> <code>\msg_kernel_error:n</code>	<code>\msg_kernel_error:nxx &lt;name&gt; &lt;arg one&gt; &lt;arg two&gt;</code>
---	---

Issues kernel error message  $\langle name \rangle$ , passing  $\langle arg one \rangle$  and  $\langle arg two \rangle$  to the text-creating functions. Cannot be redirected.

<code>\msg_kernel_warning:nxx</code> <code>\msg_kernel_warning:nx</code> <code>\msg_kernel_warning:n</code>	<code>\msg_kernel_warning:nxx &lt;name&gt; &lt;arg one&gt; &lt;arg two&gt;</code>
---	---

Prints kernel message  $\langle name \rangle$  to the terminal, passing  $\langle arg one \rangle$  and  $\langle arg two \rangle$  to the text-creating functions.

<code>\msg_kernel_info:nxx</code> <code>\msg_kernel_info:nx</code> <code>\msg_kernel_info:n</code>	<code>\msg_kernel_info:nxx &lt;name&gt; &lt;arg one&gt; &lt;arg two&gt;</code>
--	--

Prints kernel message  $\langle name \rangle$  to the log, passing  $\langle arg one \rangle$  and  $\langle arg two \rangle$  to the text-creating functions.

<code>\msg_kernel_bug:x</code>	<code>\msg_kernel_bug:x &lt;text&gt;</code>
--------------------------------	---

Short-cut for “This is a LaTeX bug: check coding” errors.

## 86 Variables and constants

<code>\c_msg_fatal_tl</code> <code>\c_msg_error_tl</code> <code>\c_msg_warning_tl</code> <code>\c_msg_info_tl</code>	Simple headers for errors.
---	----------------------------

<code>\c_msg_fatal_text_tl</code> <code>\c_msg_help_text_tl</code> <code>\c_msg_kernel_bug_text_tl</code> <code>\c_msg_kernel_bug_more_text_tl</code> <code>\c_msg_no_info_text_tl</code> <code>\c_msg_return_text_tl</code>	Various pieces of text for use in messages, which are not changed by the code here although they could be to alter the language.
---	--

<code>\c_msg_on_line_tl</code>	The “on line” phrase for line numbers.
--------------------------------	--

<code>\c_msg_text_prefix_tl</code> <code>\c_msg_more_text_prefix_tl</code> <code>\c_msg_code_prefix_tl</code>	Header information for storing the “paths” to parts of a message.
---	---

<code>\l_msg_class_tl</code> <code>\l_msg_current_class_tl</code>	Information about message method, used for filtering.
--	---

<code>\l_msg_names_clist</code>	List of all of the message names defined.
---------------------------------	---

<code>\l_msg_redirect_classes_prop</code> <code>\l_msg_redirect_names_prop</code>	Re-direction lists containing the class of message to convert an different one.
--	---

<code>\l_msg_redirect_classes_clist</code>	List so that filtering does not loop.
--	---------------------------------------

## Part XIX

# 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_`.

## 87 Generic functions

<code>\box_new:N</code> <code>\box_new:c</code> <code>^^A\box_new_l:N</code>	<code>\box_new:N</code> $\langle box \rangle$
--	---

Defines  $\langle box \rangle$  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`. However, the internal register allocation is done differently.

<code>\if_hbox:N</code> <code>\if_vbox:N</code> <code>\if_box_empty:N</code>	<code>\if_hbox:N</code> $\langle box \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code> <code>\if_box_empty:N</code> $\langle box \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
--	---

`\if_hbox:N` and `\if_vbox:N` check if  $\langle box \rangle$  is an horizontal or vertical box resp.

`\if_box_empty:N` tests if  $\langle box \rangle$  is empty (void) and executes `code` according to the test outcome.

**T<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:N<math>\overline{TF}</math></code> <code>\box_if_horizontal:c<math>\overline{TF}</math></code>	<code>\box_if_horizontal:N<math>\overline{TF}</math>    <math>\langle box \rangle</math> <math>\{\langle true\ code \rangle\}</math> <math>\{\langle false\ code \rangle\}</math></code>
--	--

Tests if  $\langle box \rangle$  is an horizontal box and executes  $\langle code \rangle$  accordingly.

<code>\box_if_vertical_p:N</code> <code>\box_if_vertical_p:c</code> <code>\box_if_vertical:N<math>\overline{TF}</math></code> <code>\box_if_vertical:c<math>\overline{TF}</math></code>	<code>\box_if_vertical:N<math>\overline{TF}</math>    <math>\langle box \rangle</math> <math>\{\langle true\ code \rangle\}</math> <math>\{\langle false\ code \rangle\}</math></code>
--	--

Tests if  $\langle box \rangle$  is a vertical box and executes  $\langle code \rangle$  accordingly.

<code>\box_if_empty_p:N</code> <code>\box_if_empty_p:c</code> <code>\box_if_empty:N<math>\overline{TF}</math></code> <code>\box_if_empty:c<math>\overline{TF}</math></code>	<code>\box_if_empty:N<math>\overline{TF}</math>    <math>\langle box \rangle</math> <math>\{\langle true\ code \rangle\}</math> <math>\{\langle false\ code \rangle\}</math></code>
--	---

Tests if  $\langle box \rangle$  is empty (void) and executes `code` according to the test outcome.

**T<sub>E</sub>Xhackers note:** `\box_if_empty:N $\overline{TF}$`  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:NN    <math>\langle box_1 \rangle</math> <math>\langle box_2 \rangle</math></code>
--	--

Sets  $\langle box_1 \rangle$  equal to  $\langle box_2 \rangle$ . Note that this eradicates 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:NN    <math>\langle box_1 \rangle</math> <math>\langle box_2 \rangle</math></code>
--	---

Globally sets  $\langle box_1 \rangle$  equal to  $\langle box_2 \rangle$ .

<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>	<code>\box_set_to_last:N    <math>\langle box \rangle</math></code>
--	---

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`     $\{ \langle \textit{dimen} \rangle \}$      $\{ \langle \textit{box function} \rangle \}$

Moves  $\langle \textit{box function} \rangle$   $\langle \textit{dimen} \rangle$  in the direction specified.  $\langle \textit{box function} \rangle$  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`     $\langle \textit{box} \rangle$

Clears  $\langle \textit{box} \rangle$  by setting it to the constant `\c_void_box`. `\box_gclear:N` does it globally.

<code>\box_use:N</code>
<code>\box_use:c</code>
<code>\box_use_clear:N</code>
<code>\box_use_clear:c</code>

`\box_use:N`     $\langle \textit{box} \rangle$

`\box_use_clear:N`     $\langle \textit{box} \rangle$

`\box_use:N` puts a copy of  $\langle \textit{box} \rangle$  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>

`\box_ht:N`     $\langle \textit{box} \rangle$

Returns the height, depth, and width of  $\langle \textit{box} \rangle$  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_show:N</code>
<code>\box_show:c</code>

`\box_show:N`     $\langle \textit{box} \rangle$

Writes the contents of  $\langle \textit{box} \rangle$  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).

## 88 Horizontal mode

<code>\hbox:n</code>	<code>\hbox:n {&lt;contents&gt;}</code>
----------------------	---

Places a `hbox` of natural size.

<code>\hbox_set:Nn</code>	<code>\hbox_set:Nn &lt;box&gt; {&lt;contents&gt;}</code>
<code>\hbox_set:cn</code>	
<code>\hbox_gset:Nn</code>	
<code>\hbox_gset:cn</code>	

Sets `<box>` to be a vertical mode box containing `<contents>`. It has its natural size. `\hbox_gset:Nn` does it globally.

<code>\hbox_set_to_wd:Nnn</code>	<code>\hbox_set_to_wd:Nnn &lt;box&gt; {&lt;dimen&gt;} {&lt;contents&gt;}</code>
<code>\hbox_set_to_wd:cnn</code>	
<code>\hbox_gset_to_wd:Nnn</code>	
<code>\hbox_gset_to_wd:cnn</code>	

Sets `<box>` to contain `<contents>` and have width `<dimen>`. `\hbox_gset_to_wd:Nn` does it globally.

<code>\hbox_to_wd:nn</code>	<code>\hbox_to_wd:nn {&lt;dimen&gt;} &lt;contents&gt;</code>
<code>\hbox_to_zero:n</code>	

Places a `<box>` of width `<dimen>` containing `<contents>`. `\hbox_to_zero:n` is a shorthand for a width of zero.

<code>\hbox_set_inline_begin:N</code>	<code>\hbox_set_inline_begin:N &lt;box&gt; &lt;contents&gt;</code>
<code>\hbox_set_inline_begin:c</code>	
<code>\hbox_set_inline_end:</code>	
<code>\hbox_gset_inline_begin:N</code>	
<code>\hbox_gset_inline_begin:c</code>	
<code>\hbox_gset_inline_end:</code>	

Sets `<box>` to contain `<contents>`. This type is useful for use in environment definitions.

<code>\hbox_unpack:N</code>	<code>\hbox_unpack:N &lt;box&gt;</code>
<code>\hbox_unpack:c</code>	
<code>\hbox_unpack_clear:N</code>	
<code>\hbox_unpack_clear:c</code>	

`\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`.

## 89 Vertical mode

<code>\vbox:n</code>	<code>\vbox:n {<math>\langle contents \rangle</math>}</code>
----------------------	--

Places a `vbox` of natural size with baseline equal to the baseline of the last line in the box.

<code>\vbox_set:Nn</code>	<code>\vbox_set:Nn <math>\langle box \rangle</math> {<math>\langle contents \rangle</math>}</code>
<code>\vbox_set:cn</code>	
<code>\vbox_gset:Nn</code>	
<code>\vbox_gset:cn</code>	

Sets  $\langle box \rangle$  to be a vertical mode box containing  $\langle contents \rangle$ . It has its natural size. `\vbox_gset:Nn` does it globally.

<code>\vbox_set_to_ht:Nnn</code>	<code>\vbox_set_to_ht:Nnn <math>\langle box \rangle</math> {<math>\langle dimen \rangle</math>} {<math>\langle contents \rangle</math>}</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>	

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_begin:N <math>\langle box \rangle</math> {<math>\langle contents \rangle</math>}</code>
<code>\vbox_set_inline_end:</code>	
<code>\vbox_gset_inline_begin:N</code>	
<code>\vbox_gset_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 <math>\langle box_1 \rangle</math> <math>\langle box_2 \rangle</math> {<math>\langle dimen \rangle</math>}</code>
--	---

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_ht:nn {<math>\langle dimen \rangle</math>} {<math>\langle contents \rangle</math>}</code>
<code>\vbox_to_zero:n</code>	

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 &lt;box&gt;</code>
--	---

`\vbox_unpack:N` unpacks the contents of the `<box>` register and `\vbox_unpack_clear:N` also clears the `<box>` after unpacking it.

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

## Part XX

# The l3xref package

## Cross references

<code>\xref_set_label:n</code>	<code>\xref_set_label:n {&lt;name&gt;}</code>
--------------------------------	---

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.

<code>\xref_new:nn</code>	<code>\xref_new:nn {&lt;type&gt;} {&lt;value&gt;}</code>
---------------------------	--

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.

<code>\xref_deferred_new:nn</code>	<code>\xref_deferred_new:nn {&lt;type&gt;} {&lt;value&gt;}</code>
------------------------------------	---

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.

<code>\xref_get_value:nn *</code>	<code>\xref_get_value:nn {&lt;type&gt;} {&lt;name&gt;}</code>
-----------------------------------	---

Extracts the cross reference information of type `<type>` for the label `<name>`. This operation is expandable.

## Part XXI

# The l3keyval package



# Key-value parsing

## 90 Introduction

Currently, this module only provides functions for extracting keys and values from a list. How this information is used is up to the programmer.

A *⟨keyval list⟩* is a list consisting of

```
key 1 = value 1 ,  
key 2      ,  
key 3 = value 3 ,
```

The function names for retrieving the keys and values are long but explain what they do. All these functions start with the name `\KV_parse_`. If a value is surrounded by braces, one level (and only one) is removed from the value. This is useful if you need to input a value which contains `=` or `,.` There are two primary actions we can take on a *⟨keyval list⟩*.

- A *⟨keyval list⟩* can be sanitized so that top level active commas or equal signs are converted into catcode 12. When declaring templates in the preamble one can probably safely assume that there are no active commas or equals; these things should only be active in the document. The name for this is either `no_sanitize` or `sanitize`.
- Spaces on either side of a key or value can be trimmed. When using the L<sup>A</sup>T<sub>E</sub>X3 programming interface, spaces are automatically ignored so there it would be a waste of time to search for extra spaces since there would be none. At the document level however, spaces must be removed. The name for this is either `no_space_removal` or `space_removal`. Note that when `space_removal` is called you get an additional option where you can decide if one level of braces should be stripped from the key and/or value; see the description of the boolean `\l_KV_remove_one_level_of_braces_bool` for details.

During the parsing process, keys or values are not expanded and no `#`s are doubled. When the parsing process is over, the keys and values are executed in the form

```
\KV_key_value_elt:nn{key 1}{value 1}  
\KV_key_no_value_elt:n{key 2}  
\KV_key_value_elt:nn{key 3}{value 3}
```

It is up to the programmer to provide a suitable definition of these two functions before starting the parsing process.

## 91 Functions

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

`\KV_parse_no_space_removal_no_sanitize:n {⟨keyval`

Parses the keys and values literally. For use when spaces are ignored and = and , have normal catcodes.

<code>\KV_parse_space_removal_no_sanitize:n</code>	<code>\KV_parse_space_removal_no_sanitize:n {&lt;keyval list&gt;}</code>
--	--

As above but also removes spaces around keys and values. For use when spaces are not ignored.

<code>\KV_parse_space_removal_sanitize:n</code>	<code>\KV_parse_space_removal_sanitize:n {&lt;keyval list&gt;}</code>
---	---

As above but additionally also replaces top-level active = and , with harmless versions.

<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 `<true>` 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. Setting this boolean has no effect when you call the `no_space_removal` functions since not stripping braces is a rare request and would clutter the otherwise elegant code of the `no_space_removal` functions for very little gain. If you need to preserve braces choose the slower `space_removal` functions.

<code>\KV_key_no_value_elt:n</code>	<code>\KV_key_no_value_elt:n {&lt;key&gt;}</code>
<code>\KV_key_value_elt:nn</code>	<code>\KV_key_value_elt:n {&lt;key&gt;} {&lt;value&gt;}</code>

Functions returned by the `\KV_parse_` functions. The default definition of these two functions is an error message!

## Part XXII

# The l3calc package

## Infix notation arithmetic in L<sup>A</sup>T<sub>E</sub>X3

This is pretty much a straight adaption of the `calc` package and as such has same syntax for the *<calc expression>*. However, there are some noticeable differences.

- The `calc` expression is expanded fully, which means there are no problems with unfinished conditionals. However, the contents of `\widthof` etc. is not expanded at all. This includes uses in traditional L<sup>A</sup>T<sub>E</sub>X as in the `array` package, which tries to do an `\edef` several times. The code used in `l3calc` provides self-protection for these cases.
- Muskip registers are supported although they can only be used in `\ratio` if already evaluating a muskip expression. For the other three register types, you can use points.

- All results are rounded, not truncated. More precisely, the primitive TeX operations `\divide` and `\multiply` are not used. The only instance where one will observe an effect is when dividing integers.

This version of `l3calc` is now a complete replacement for the original `calc` package providing the same functionality and will prevent the original `calc` package from loading.

## 92 User functions

```
\maxof
\minof
\widthof
\heightof
\depthof
\totalheightof
\ratio
\real
\setlength
\gsetlength
\addtolength
\gaddtolength
\setcounter
\addtocounter
\stepcounter
```

See documentation for L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> package `calc`.

```
\calc_maxof:nn
\calc_minof:nn
\calc_widthof:n
\calc_heightof:n
\calc_depthof:n
\calc_totalheightof:n
\calc_ratio:nn
\calc_real:n
\calc_setcounter:nn
\calc_addtocounter:nn
\calc_stepcounter:n
```

Equivalent commands as the above in the `expl3` namespace.

```
\calc_int_set:Nn
\calc_int_gset:Nn
\calc_int_add:Nn
\calc_int_gadd:Nn
\calc_int_sub:Nn
\calc_int_gsub:Nn
```

`\calc_int_set:Nn <int> {<calc expression>}`

Evaluates *<calc expression>* and either adds or subtracts it from *<int>* or sets *<int>* to it. These operations can also be global.

```

\calc_dim_set:Nn
\calc_dim_gset:Nn
\calc_dim_add:Nn
\calc_dim_gadd:Nn
\calc_dim_sub:Nn
\calc_dim_gsub:Nn

```

```
\calc_dim_set:Nn <dim> {<calc expression>}
```

Evaluates  $\langle calc\ expression \rangle$  and either adds or subtracts it from  $\langle dim \rangle$  or sets  $\langle dim \rangle$  to it. These operations can also be global.

```

\calc_skip_set:Nn
\calc_skip_gset:Nn
\calc_skip_add:Nn
\calc_skip_gadd:Nn
\calc_skip_sub:Nn
\calc_skip_gsub:Nn

```

```
\calc_skip_set:Nn <skip> {<calc expression>}
```

Evaluates  $\langle calc\ expression \rangle$  and either adds or subtracts it from  $\langle skip \rangle$  or sets  $\langle skip \rangle$  to it. These operations can also be global.

```

\calc_muskip_set:Nn
\calc_muskip_gset:Nn
\calc_muskip_add:Nn
\calc_muskip_gadd:Nn
\calc_muskip_sub:Nn
\calc_muskip_gsub:Nn

```

```
\calc_muskip_set:Nn <muskip> {<calc expression>}
```

Evaluates  $\langle calc\ expression \rangle$  and either adds or subtracts it from  $\langle muskip \rangle$  or sets  $\langle muskip \rangle$  to it. These operations can also be global.

```
\calc_calculate_box_size:nnn
```

```
\calc_calculate_box_size:nnn {<dim-set>}
{<item1> <item2> ... <itemn>} {<contents>}
```

Sets  $\langle contents \rangle$  in a temporary box `\l_tmpa_box`. Then  $\langle dim-set \rangle$  is put in front of a loop that inserts  $+ \langle item_i \rangle$  in front of `\l_tmpa_box` and this is evaluated. For instance, if we wanted to determine the total height of the text `xyz` and store it in `\l_tmpa_dim`, we would call it as.

```

\calc_calculate_box_size:nnn
{<dim-set:Nn\l_tmpa_dim>}{\box_ht:N\box_dp:N}{xyz}

```

Similarly, if we wanted the difference between height and depth, we could call it as

```

\calc_calculate_box_size:nnn
{<dim-set:Nn\l_tmpa_dim>}{\box_ht:N{-\box_dp:N}}{xyz}

```

## Part XXIII

# The l3file package

## File Loading

### 93 Loading files

The need to test if a file is available and to load a file if found is covered at a low level here.

<code>\file_if_exist_p:n</code>
<code>\file_if_exist:nTF</code>

`\file_if_exist:nTF <file> <true code> <false code>`

Tests if `<file>` exists.

**T<sub>E</sub>Xhackers note:** This is L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\IfFileExists`, although it does not return name of the file (*cf.* `\@filef@und`).

<code>\file_add_path:nN</code>
--------------------------------

`\file_add_path:nN <file> <tl var.>`

Searches for `<file>` on the T<sub>E</sub>X path and using `\l_file_search_path_clist`. If `<file>` is found, `<tl var.>` is set to the file name plus any path information needed. If `<file>` is not found, `<tl var.>` will be blank.

**T<sub>E</sub>Xhackers note:** This is similar to obtaining `\@filef@und` from L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\IfFileExists`.

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

`\file_input:n <file>`

Inputs `<file>` if it found (according to the same rule as for `\file_if_exist:n`).

**T<sub>E</sub>Xhackers note:** This acts in a similar way to L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\input`, as it will not lead to a T<sub>E</sub>X loop if the file is not found.

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

`\file_input_no_record:n <file>`

Inputs `<file>` if it found (according to the same rule as for `\file_if_exist:n`, but does not add it to `\g_file_record_clist`. The file is still added to `\g_file_record_full_clist`.

**T<sub>E</sub>Xhackers note:** This is similar to L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\@input@`.

<code>\file_input_no_check:n</code>
<code>\file_input_no_check_no_record:n</code>

`\file_input_no_check:n <file>`

Inputs  $\langle file \rangle$  directly without checking if it exists. The `no_record` version does not record the input in

**T<sub>E</sub>Xhackers note:** `\g_file_record_clist`. These are simple wrappers around the T<sub>E</sub>X `\input` primitive

<code>\file_list:</code> <code>\file_list_full:</code>	<code>\file_list:</code> $\langle file \rangle$
---	---

Lists files loaded in current L<sup>A</sup>T<sub>E</sub>X run: the `full` version lists all files.

## 94 Variables and constants

<code>\g_file_record_clist</code> <code>\g_file_record_full_clist</code>
---

Used to track the files that have been loaded.

<code>\l_file_search_path_clist</code>
--

List of paths to search for a file in addition to those searched by T<sub>E</sub>X.

**T<sub>E</sub>Xhackers note:** This is `\input@path` in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>.

<code>\l_file_test_read_stream</code>
---------------------------------------

Input stream used to carry out file tests.

<code>\l_file_tmp_bool</code>
-------------------------------

Internal scratch switch.

<code>\l_file_tmp_tl</code>
-----------------------------

Internal scratch token list variable.

## Part XXIV

# Implementation

## 95 l3names implementation

This is the base part of L<sup>A</sup>T<sub>E</sub>X 3 defining things like `catcodes` and redefining the T<sub>E</sub>X primitives, as well as setting up the code to load `expl3` modules in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>.

## 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 L<sup>A</sup>T<sub>E</sub>X's file-loading functions to support `\ExplSyntax`.

## 95.2 Package loading

Before anything else, check that we're using  $\varepsilon$ -T<sub>E</sub>X; no point continuing otherwise.

```

1 <*initex | package>
2 \begingroup
3 \def\firstoftwo#1#2{#1}
4 \def\secondoftwo#1#2{#2}
5 \def\etexmissingerror{Not running under e-TeX}
6 \def\etexmissinghelp{%
7   This package requires e-TeX.^^J%
8   Try compiling the document with 'elatex' instead of 'latex'.^^J%
9   When using pdfTeX, try 'pdfelatex' instead of 'pdflatex'%
10 }%
11 \expandafter\ifx\csname eTeXversion\endcsname\relax
12   \expandafter\secondoftwo\else\expandafter\firstoftwo\fi
13   {\endgroup}%
14 <initex>      \expandafter\errhelp\expandafter{\etexmissinghelp}%
15 <initex>      \expandafter\errmessage\expandafter{\etexmissingerror}%
16 <package>     \PackageError{l3names}{\etexmissingerror}{\etexmissinghelp}%
17   \endgroup
18   \endinput
19 }
20 </initex | package>

```

## 95.3 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.

```

21 <*initex | package>
22 \edef\ExplSyntaxOff{
23   \unexpanded{\ifodd \ExplSyntaxStatus\relax

```

```

24 \def\ExplSyntaxStatus{0}
25 }
26 \catcode 126=\the \catcode 126 \relax
27 \catcode 32=\the \catcode 32 \relax
28 \catcode 9=\the \catcode 9 \relax
29 \endlinechar =\the \endlinechar \relax
30 \catcode 95=\the \catcode 95 \relax
31 \catcode 58=\the \catcode 58 \relax
32 \noexpand\fi
33 }
34 \catcode126=10\relax % tilde is a space char.
35 \catcode32=9\relax % space is ignored
36 \catcode9=9\relax % tab also ignored
37 \endlinechar=32\relax % endline is space
38 \catcode95=11\relax % underscore letter
39 \catcode58=11\relax % colon letter

```

## 95.4 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.

```

40 \let\tex_let:D\let
41 </initex | package>

```

and now an internal function to possibly remove the old name.

```

42 <*initex>
43 \long\def\name_undefine:N#1{
44   \tex_let:D#1\c_undefined}
45 </initex>

46 <*package>
47 \DeclareOption{removeoldnames}{
48   \long\def\name_undefine:N#1{
49     \tex_let:D#1\c_undefined}}

50 \DeclareOption{keepoldnames}{
51   \long\def\name_undefine:N#1{}}

52 \ExecuteOptions{keepoldnames}

53 \ProcessOptions
54 </package>

```



The internal function to give the new name and possibly undefine the old name.

```

55 <*initex | package>
56 \long\def\name_primitive:NN#1#2{
57   \tex_let:D #2 #1
58   \name_undefine:N #1
59 }
```

## 95.5 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.

```

60 \name_primitive:NN \           \tex_space:D
61 \name_primitive:NN \/         \tex_italiccor:D
62 \name_primitive:NN \-        \tex_hyphen:D
```

Now all the other primitives.

```

63 \name_primitive:NN \let       \tex_let:D
64 \name_primitive:NN \def       \tex_def:D
65 \name_primitive:NN \edef      \tex_edef:D
66 \name_primitive:NN \gdef      \tex_gdef:D
67 \name_primitive:NN \xdef      \tex_xdef:D
68 \name_primitive:NN \chardef   \tex_chardef:D
69 \name_primitive:NN \countdef   \tex_countdef:D
70 \name_primitive:NN \dimendef   \tex_dimendef:D
71 \name_primitive:NN \skipdef    \tex_skipdef:D
72 \name_primitive:NN \muskipdef  \tex_muskipdef:D
73 \name_primitive:NN \mathchardef \tex_mathchardef:D
74 \name_primitive:NN \toksdef    \tex_toksdef:D
75 \name_primitive:NN \futurelet  \tex_futurelet:D
76 \name_primitive:NN \advance    \tex_advance:D
77 \name_primitive:NN \divide     \tex_divide:D
78 \name_primitive:NN \multiply   \tex_multiply:D
79 \name_primitive:NN \font       \tex_font:D
80 \name_primitive:NN \fam        \tex_fam:D
81 \name_primitive:NN \global     \tex_global:D
82 \name_primitive:NN \long       \tex_long:D
83 \name_primitive:NN \outer      \tex_outer:D
84 \name_primitive:NN \setlanguage \tex_setlanguage:D
85 \name_primitive:NN \globaldefs \tex_globaldefs:D
86 \name_primitive:NN \afterassignment \tex_afterassignment:D
87 \name_primitive:NN \aftergroup  \tex_aftergroup:D
88 \name_primitive:NN \expandafter \tex_expandafter:D
89 \name_primitive:NN \noexpand    \tex_noexpand:D
90 \name_primitive:NN \begingroup  \tex_begingroup:D
91 \name_primitive:NN \endgroup    \tex_endgroup:D
92 \name_primitive:NN \halign      \tex_halign:D
93 \name_primitive:NN \valign      \tex_valign:D
94 \name_primitive:NN \cr          \tex_cr:D
95 \name_primitive:NN \crcr       \tex_crcr:D
96 \name_primitive:NN \noalign     \tex_noalign:D
```

97	\name_primitive:NN	\omit	\tex_omit:D
98	\name_primitive:NN	\span	\tex_span:D
99	\name_primitive:NN	\tabskip	\tex_tabskip:D
100	\name_primitive:NN	\everycr	\tex_everycr:D
101	\name_primitive:NN	\if	\tex_if:D
102	\name_primitive:NN	\ifcase	\tex_ifcase:D
103	\name_primitive:NN	\ifcat	\tex_ifcat:D
104	\name_primitive:NN	\ifnum	\tex_ifnum:D
105	\name_primitive:NN	\ifodd	\tex_ifodd:D
106	\name_primitive:NN	\ifdim	\tex_ifdim:D
107	\name_primitive:NN	\ifeof	\tex_ifeof:D
108	\name_primitive:NN	\ifhbox	\tex_ifhbox:D
109	\name_primitive:NN	\ifvbox	\tex_ifvbox:D
110	\name_primitive:NN	\ifvoid	\tex_ifvoid:D
111	\name_primitive:NN	\ifx	\tex_ifx:D
112	\name_primitive:NN	\iffalse	\tex_iffalse:D
113	\name_primitive:NN	\iftrue	\tex_iftrue:D
114	\name_primitive:NN	\ifhmode	\tex_ifhmode:D
115	\name_primitive:NN	\ifmmode	\tex_ifmmode:D
116	\name_primitive:NN	\ifvmode	\tex_ifvmode:D
117	\name_primitive:NN	\ifinner	\tex_ifinner:D
118	\name_primitive:NN	\else	\tex_else:D
119	\name_primitive:NN	\fi	\tex_fi:D
120	\name_primitive:NN	\or	\tex_or:D
121	\name_primitive:NN	\immediate	\tex_immediate:D
122	\name_primitive:NN	\closeout	\tex_closeout:D
123	\name_primitive:NN	\openin	\tex_openin:D
124	\name_primitive:NN	\openout	\tex_openout:D
125	\name_primitive:NN	\read	\tex_read:D
126	\name_primitive:NN	\write	\tex_write:D
127	\name_primitive:NN	\closein	\tex_closein:D
128	\name_primitive:NN	\newlinechar	\tex_newlinechar:D
129	\name_primitive:NN	\input	\tex_input:D
130	\name_primitive:NN	\endinput	\tex_endinput:D
131	\name_primitive:NN	\inputlineno	\tex_inputlineno:D
132	\name_primitive:NN	\errmessage	\tex_errmessage:D
133	\name_primitive:NN	\message	\tex_message:D
134	\name_primitive:NN	\show	\tex_show:D
135	\name_primitive:NN	\showthe	\tex_showthe:D
136	\name_primitive:NN	\showbox	\tex_showbox:D
137	\name_primitive:NN	\showlists	\tex_showlists:D
138	\name_primitive:NN	\errhelp	\tex_errhelp:D
139	\name_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
140	\name_primitive:NN	\tracingcommands	\tex_tracingcommands:D
141	\name_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
142	\name_primitive:NN	\tracingmacros	\tex_tracingmacros:D
143	\name_primitive:NN	\tracingonline	\tex_tracingonline:D
144	\name_primitive:NN	\tracingoutput	\tex_tracingoutput:D
145	\name_primitive:NN	\tracingpages	\tex_tracingpages:D
146	\name_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
147	\name_primitive:NN	\tracingrestores	\tex_tracingrestores:D
148	\name_primitive:NN	\tracingstats	\tex_tracingstats:D
149	\name_primitive:NN	\pausing	\tex_pausing:D
150	\name_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D

151	\name_primitive:NN	\showboxdepth	\tex_showboxdepth:D
152	\name_primitive:NN	\batchmode	\tex_batchmode:D
153	\name_primitive:NN	\errorstopmode	\tex_errorstopmode:D
154	\name_primitive:NN	\nonstopmode	\tex_nonstopmode:D
155	\name_primitive:NN	\scrollmode	\tex_scrollmode:D
156	\name_primitive:NN	\end	\tex_end:D
157	\name_primitive:NN	\csname	\tex_csname:D
158	\name_primitive:NN	\endcsname	\tex_endcsname:D
159	\name_primitive:NN	\ignorespaces	\tex_ignorespaces:D
160	\name_primitive:NN	\relax	\tex_relax:D
161	\name_primitive:NN	\the	\tex_the:D
162	\name_primitive:NN	\mag	\tex_mag:D
163	\name_primitive:NN	\language	\tex_language:D
164	\name_primitive:NN	\mark	\tex_mark:D
165	\name_primitive:NN	\topmark	\tex_topmark:D
166	\name_primitive:NN	\firstmark	\tex_firstmark:D
167	\name_primitive:NN	\botmark	\tex_botmark:D
168	\name_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
169	\name_primitive:NN	\splitbotmark	\tex_splitbotmark:D
170	\name_primitive:NN	\fontname	\tex_fontname:D
171	\name_primitive:NN	\escapechar	\tex_escapechar:D
172	\name_primitive:NN	\endlinechar	\tex_endlinechar:D
173	\name_primitive:NN	\mathchoice	\tex_mathchoice:D
174	\name_primitive:NN	\delimiter	\tex_delimiter:D
175	\name_primitive:NN	\mathaccent	\tex_mathaccent:D
176	\name_primitive:NN	\mathchar	\tex_mathchar:D
177	\name_primitive:NN	\mskip	\tex_mskip:D
178	\name_primitive:NN	\radical	\tex_radical:D
179	\name_primitive:NN	\vcenter	\tex_vcenter:D
180	\name_primitive:NN	\mkern	\tex_mkern:D
181	\name_primitive:NN	\above	\tex_above:D
182	\name_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
183	\name_primitive:NN	\atop	\tex_atop:D
184	\name_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
185	\name_primitive:NN	\over	\tex_over:D
186	\name_primitive:NN	\overwithdelims	\tex_overwithdelims:D
187	\name_primitive:NN	\displaystyle	\tex_displaystyle:D
188	\name_primitive:NN	\textstyle	\tex_textstyle:D
189	\name_primitive:NN	\scriptstyle	\tex_scriptstyle:D
190	\name_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
191	\name_primitive:NN	\nonscript	\tex_nonscript:D
192	\name_primitive:NN	\eqno	\tex_eqno:D
193	\name_primitive:NN	\leqno	\tex_leqno:D
194	\name_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D
195	\name_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
196	\name_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
197	\name_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
198	\name_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
199	\name_primitive:NN	\displayindent	\tex_displayindent:D
200	\name_primitive:NN	\displaywidth	\tex_displaywidth:D
201	\name_primitive:NN	\everydisplay	\tex_everydisplay:D
202	\name_primitive:NN	\predisplaysize	\tex_predisplaysize:D
203	\name_primitive:NN	\predisdisplaypenalty	\tex_predisdisplaypenalty:D
204	\name_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D

205	\name_primitive:NN	\mathbin	\tex_mathbin:D
206	\name_primitive:NN	\mathclose	\tex_mathclose:D
207	\name_primitive:NN	\mathinner	\tex_mathinner:D
208	\name_primitive:NN	\mathop	\tex_mathop:D
209	\name_primitive:NN	\displaylimits	\tex_displaylimits:D
210	\name_primitive:NN	\limits	\tex_limits:D
211	\name_primitive:NN	\nolimits	\tex_nolimits:D
212	\name_primitive:NN	\mathopen	\tex_mathopen:D
213	\name_primitive:NN	\mathord	\tex_mathord:D
214	\name_primitive:NN	\mathpunct	\tex_mathpunct:D
215	\name_primitive:NN	\mathrel	\tex_mathrel:D
216	\name_primitive:NN	\overline	\tex_overline:D
217	\name_primitive:NN	\underline	\tex_underline:D
218	\name_primitive:NN	\left	\tex_left:D
219	\name_primitive:NN	\right	\tex_right:D
220	\name_primitive:NN	\binoppenalty	\tex_binoppenalty:D
221	\name_primitive:NN	\relpenalty	\tex_relpenalty:D
222	\name_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
223	\name_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
224	\name_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
225	\name_primitive:NN	\everymath	\tex_everymath:D
226	\name_primitive:NN	\mathsurround	\tex_mathsurround:D
227	\name_primitive:NN	\medmuskip	\tex_medmuskip:D
228	\name_primitive:NN	\thinmuskip	\tex_thinmuskip:D
229	\name_primitive:NN	\thickmuskip	\tex_thickmuskip:D
230	\name_primitive:NN	\scriptspace	\tex_scriptspace:D
231	\name_primitive:NN	\noboundary	\tex_noboundary:D
232	\name_primitive:NN	\accent	\tex_accent:D
233	\name_primitive:NN	\char	\tex_char:D
234	\name_primitive:NN	\discretionary	\tex_discretionary:D
235	\name_primitive:NN	\hfil	\tex_hfil:D
236	\name_primitive:NN	\hfilneg	\tex_hfilneg:D
237	\name_primitive:NN	\hfill	\tex_hfill:D
238	\name_primitive:NN	\hskip	\tex_hskip:D
239	\name_primitive:NN	\hss	\tex_hss:D
240	\name_primitive:NN	\vfil	\tex_vfil:D
241	\name_primitive:NN	\vfilneg	\tex_vfilneg:D
242	\name_primitive:NN	\vfill	\tex_vfill:D
243	\name_primitive:NN	\vskip	\tex_vskip:D
244	\name_primitive:NN	\vss	\tex_vss:D
245	\name_primitive:NN	\unskip	\tex_unskip:D
246	\name_primitive:NN	\kern	\tex_kern:D
247	\name_primitive:NN	\unkern	\tex_unkern:D
248	\name_primitive:NN	\hrule	\tex_hrule:D
249	\name_primitive:NN	\vrule	\tex_vrule:D
250	\name_primitive:NN	\leaders	\tex_leaders:D
251	\name_primitive:NN	\cleaders	\tex_cleaders:D
252	\name_primitive:NN	\xleaders	\tex_xleaders:D
253	\name_primitive:NN	\lastkern	\tex_lastkern:D
254	\name_primitive:NN	\lastskip	\tex_lastskip:D
255	\name_primitive:NN	\indent	\tex_indent:D
256	\name_primitive:NN	\par	\tex_par:D
257	\name_primitive:NN	\noindent	\tex_noindent:D
258	\name_primitive:NN	\vadjust	\tex_vadjust:D

259	\name_primitive:NN	\baselineskip	\tex_baselineskip:D
260	\name_primitive:NN	\lineskip	\tex_lineskip:D
261	\name_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
262	\name_primitive:NN	\clubpenalty	\tex_clubpenalty:D
263	\name_primitive:NN	\widowpenalty	\tex_widowpenalty:D
264	\name_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
265	\name_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
266	\name_primitive:NN	\linepenalty	\tex_linepenalty:D
267	\name_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
268	\name_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
269	\name_primitive:NN	\adjdemerits	\tex_adjdemerits:D
270	\name_primitive:NN	\hangafter	\tex_hangafter:D
271	\name_primitive:NN	\hangindent	\tex_hangindent:D
272	\name_primitive:NN	\parshape	\tex_parshape:D
273	\name_primitive:NN	\hsize	\tex_hsize:D
274	\name_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
275	\name_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
276	\name_primitive:NN	\leftskip	\tex_leftskip:D
277	\name_primitive:NN	\rightskip	\tex_rightskip:D
278	\name_primitive:NN	\looseness	\tex_looseness:D
279	\name_primitive:NN	\parskip	\tex_parskip:D
280	\name_primitive:NN	\parindent	\tex_parindent:D
281	\name_primitive:NN	\uchyph	\tex_uchyph:D
282	\name_primitive:NN	\emergencystretch	\tex_emergencystretch:D
283	\name_primitive:NN	\pretolerance	\tex_pretolerance:D
284	\name_primitive:NN	\tolerance	\tex_tolerance:D
285	\name_primitive:NN	\spaceskip	\tex_spaceskip:D
286	\name_primitive:NN	\xspaceskip	\tex_xspaceskip:D
287	\name_primitive:NN	\parfillskip	\tex_parfillskip:D
288	\name_primitive:NN	\everypar	\tex_everypar:D
289	\name_primitive:NN	\prevgraf	\tex_prevgraf:D
290	\name_primitive:NN	\spacefactor	\tex_spacefactor:D
291	\name_primitive:NN	\shipout	\tex_shipout:D
292	\name_primitive:NN	\vsize	\tex_vsize:D
293	\name_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
294	\name_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
295	\name_primitive:NN	\topskip	\tex_topskip:D
296	\name_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
297	\name_primitive:NN	\maxdepth	\tex_maxdepth:D
298	\name_primitive:NN	\output	\tex_output:D
299	\name_primitive:NN	\deadcycles	\tex_deadcycles:D
300	\name_primitive:NN	\pagedepth	\tex_pagedepth:D
301	\name_primitive:NN	\pagestretch	\tex_pagestretch:D
302	\name_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
303	\name_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
304	\name_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
305	\name_primitive:NN	\pageshrink	\tex_pageshrink:D
306	\name_primitive:NN	\pagegoal	\tex_pagegoal:D
307	\name_primitive:NN	\pagetotal	\tex_pagetotal:D
308	\name_primitive:NN	\outputpenalty	\tex_outputpenalty:D
309	\name_primitive:NN	\hoffset	\tex_hoffset:D
310	\name_primitive:NN	\voffset	\tex_voffset:D
311	\name_primitive:NN	\insert	\tex_insert:D
312	\name_primitive:NN	\holdinginserts	\tex_holdinginserts:D

313	\name_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
314	\name_primitive:NN	\insertpenalties	\tex_insertpenalties:D
315	\name_primitive:NN	\lower	\tex_lower:D
316	\name_primitive:NN	\moveleft	\tex_moveleft:D
317	\name_primitive:NN	\moveright	\tex_moveright:D
318	\name_primitive:NN	\raise	\tex_raise:D
319	\name_primitive:NN	\copy	\tex_copy:D
320	\name_primitive:NN	\lastbox	\tex_lastbox:D
321	\name_primitive:NN	\vsplit	\tex_vsplit:D
322	\name_primitive:NN	\unhbox	\tex_unhbox:D
323	\name_primitive:NN	\unhcopy	\tex_unhcopy:D
324	\name_primitive:NN	\unvbox	\tex_unvbox:D
325	\name_primitive:NN	\unvcopy	\tex_unvcopy:D
326	\name_primitive:NN	\setbox	\tex_setbox:D
327	\name_primitive:NN	\hbox	\tex_hbox:D
328	\name_primitive:NN	\vbox	\tex_vbox:D
329	\name_primitive:NN	\vtop	\tex_vtop:D
330	\name_primitive:NN	\prevdepth	\tex_prevdepth:D
331	\name_primitive:NN	\badness	\tex_badness:D
332	\name_primitive:NN	\hbadness	\tex_hbadness:D
333	\name_primitive:NN	\vbadness	\tex_vbadness:D
334	\name_primitive:NN	\hfuzz	\tex_hfuzz:D
335	\name_primitive:NN	\vfuzz	\tex_vfuzz:D
336	\name_primitive:NN	\overfullrule	\tex_overfullrule:D
337	\name_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
338	\name_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
339	\name_primitive:NN	\splittopskip	\tex_splittopskip:D
340	\name_primitive:NN	\everyhbox	\tex_everyhbox:D
341	\name_primitive:NN	\everyvbox	\tex_everyvbox:D
342	\name_primitive:NN	\nullfont	\tex_nullfont:D
343	\name_primitive:NN	\textfont	\tex_textfont:D
344	\name_primitive:NN	\scriptfont	\tex_scriptfont:D
345	\name_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
346	\name_primitive:NN	\fontdimen	\tex_fontdimen:D
347	\name_primitive:NN	\hyphenchar	\tex_hyphenchar:D
348	\name_primitive:NN	\skewchar	\tex_skewchar:D
349	\name_primitive:NN	\defaultshyphenchar	\tex_defaultshyphenchar:D
350	\name_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
351	\name_primitive:NN	\number	\tex_number:D
352	\name_primitive:NN	\romannumeral	\tex_romannumeral:D
353	\name_primitive:NN	\string	\tex_string:D
354	\name_primitive:NN	\lowercase	\tex_lowercase:D
355	\name_primitive:NN	\uppercase	\tex_uppercase:D
356	\name_primitive:NN	\meaning	\tex_meaning:D
357	\name_primitive:NN	\penalty	\tex_penalty:D
358	\name_primitive:NN	\unpenalty	\tex_unpenalty:D
359	\name_primitive:NN	\lastpenalty	\tex_lastpenalty:D
360	\name_primitive:NN	\special	\tex_special:D
361	\name_primitive:NN	\dump	\tex_dump:D
362	\name_primitive:NN	\patterns	\tex_patterns:D
363	\name_primitive:NN	\hyphenation	\tex_hyphenation:D
364	\name_primitive:NN	\time	\tex_time:D
365	\name_primitive:NN	\day	\tex_day:D
366	\name_primitive:NN	\month	\tex_month:D

367	<code>\name_primitive:NN \year</code>	<code>\tex_year:D</code>
368	<code>\name_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
369	<code>\name_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
370	<code>\name_primitive:NN \count</code>	<code>\tex_count:D</code>
371	<code>\name_primitive:NN \dimen</code>	<code>\tex_dimen:D</code>
372	<code>\name_primitive:NN \skip</code>	<code>\tex_skip:D</code>
373	<code>\name_primitive:NN \toks</code>	<code>\tex_toks:D</code>
374	<code>\name_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
375	<code>\name_primitive:NN \box</code>	<code>\tex_box:D</code>
376	<code>\name_primitive:NN \wd</code>	<code>\tex_wd:D</code>
377	<code>\name_primitive:NN \ht</code>	<code>\tex_ht:D</code>
378	<code>\name_primitive:NN \dp</code>	<code>\tex_dp:D</code>
379	<code>\name_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
380	<code>\name_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
381	<code>\name_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
382	<code>\name_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
383	<code>\name_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
384	<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  $\epsilon$ -T<sub>E</sub>X extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

385	<code>\name_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
386	<code>\name_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
387	<code>\name_primitive:NN \unless</code>	<code>\etex_unless:D</code>
388	<code>\name_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
389	<code>\name_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
390	<code>\name_primitive:NN \marks</code>	<code>\etex_marks:D</code>
391	<code>\name_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
392	<code>\name_primitive:NN \firstmarks</code>	<code>\etex_firstmarks:D</code>
393	<code>\name_primitive:NN \botmarks</code>	<code>\etex_botmarks:D</code>
394	<code>\name_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
395	<code>\name_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
396	<code>\name_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
397	<code>\name_primitive:NN \detokenize</code>	<code>\etex_detokenize:D</code>
398	<code>\name_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
399	<code>\name_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
400	<code>\name_primitive:NN \readline</code>	<code>\etex_readline:D</code>
401	<code>\name_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
402	<code>\name_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
403	<code>\name_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
404	<code>\name_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
405	<code>\name_primitive:NN \currentiflevel</code>	<code>\etex_currentiflevel:D</code>
406	<code>\name_primitive:NN \currentifbranch</code>	<code>\etex_currentifbranch:D</code>
407	<code>\name_primitive:NN \currentifttype</code>	<code>\etex_currentifttype:D</code>
408	<code>\name_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
409	<code>\name_primitive:NN \currentgrouplevel</code>	<code>\etex_currentgrouplevel:D</code>
410	<code>\name_primitive:NN \currentgrouptype</code>	<code>\etex_currentgrouptype:D</code>
411	<code>\name_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
412	<code>\name_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
413	<code>\name_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
414	<code>\name_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
415	<code>\name_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
416	<code>\name_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>
417	<code>\name_primitive:NN \fontchardp</code>	<code>\etex_fontchardp:D</code>

418	\name_primitive:NN	\fontcharwd	\etex_fontcharwd:D
419	\name_primitive:NN	\fontcharic	\etex_fontcharic:D
420	\name_primitive:NN	\parshapeindent	\etex_parshapeindent:D
421	\name_primitive:NN	\parshapelength	\etex_parshapelength:D
422	\name_primitive:NN	\parshapedimen	\etex_parshapedimen:D
423	\name_primitive:NN	\numexpr	\etex_numexpr:D
424	\name_primitive:NN	\dimexpr	\etex_dimexpr:D
425	\name_primitive:NN	\glueexpr	\etex_glueexpr:D
426	\name_primitive:NN	\muexpr	\etex_muexpr:D
427	\name_primitive:NN	\gluestretch	\etex_gluestretch:D
428	\name_primitive:NN	\glueshrink	\etex_glueshrink:D
429	\name_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
430	\name_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
431	\name_primitive:NN	\gluetomu	\etex_gluetomu:D
432	\name_primitive:NN	\mutoglu	\etex_mutoglu:D
433	\name_primitive:NN	\lastlinefit	\etex_lastlinefit:D
434	\name_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D
435	\name_primitive:NN	\clubpenalties	\etex_clubpenalties:D
436	\name_primitive:NN	\widowpenalties	\etex_widowpenalties:D
437	\name_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
438	\name_primitive:NN	\middle	\etex_middle:D
439	\name_primitive:NN	\savinghyphcodes	\etex_savinghyphcodes:D
440	\name_primitive:NN	\savingvdiscards	\etex_savingvdiscards:D
441	\name_primitive:NN	\pagediscards	\etex_pagediscards:D
442	\name_primitive:NN	\splitdiscards	\etex_splitdiscards:D
443	\name_primitive:NN	\TeXETstate	\etex_TeXXETstate:D
444	\name_primitive:NN	\beginL	\etex_beginL:D
445	\name_primitive:NN	\endL	\etex_endL:D
446	\name_primitive:NN	\beginR	\etex_beginR:D
447	\name_primitive:NN	\endR	\etex_endR:D
448	\name_primitive:NN	\predisplaydirection	\etex_predisplaydirection:D
449	\name_primitive:NN	\everyeof	\etex_everyeof:D
450	\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 them a double pdf prefix. The list below covers pdfTeXv 1.30.4.

451	%% integer registers:		
452	\name_primitive:NN	\pdfoutput	\pdf_output:D
453	\name_primitive:NN	\pdfminorversion	\pdf_minorversion:D
454	\name_primitive:NN	\pdfcompresslevel	\pdf_compresslevel:D
455	\name_primitive:NN	\pdfdecimaldigits	\pdf_decimaldigits:D
456	\name_primitive:NN	\pdfimageresolution	\pdf_imageresolution:D
457	\name_primitive:NN	\pdfpkresolution	\pdf_pkresolution:D
458	\name_primitive:NN	\pdftracingfonts	\pdf_tracingfonts:D
459	\name_primitive:NN	\pdfuniqueresname	\pdf_uniqueresname:D
460	\name_primitive:NN	\pdfadjustspacing	\pdf_adjustspacing:D
461	\name_primitive:NN	\pdfprotrudechars	\pdf_protrudechars:D
462	\name_primitive:NN	\efcode	\pdf_efcode:D
463	\name_primitive:NN	\lpcode	\pdf_lpcode:D
464	\name_primitive:NN	\rpcode	\pdf_rpcode:D
465	\name_primitive:NN	\pdfforcepagebox	\pdf_forcepagebox:D



```

466 \name_primitive:NN \pdfoptionalwaysusepdfpagebox \pdf_optionalwaysusepdfpagebox:D
467 \name_primitive:NN \pdfinclusionerrorlevel\pdf_inclusionerrorlevel:D
468 \name_primitive:NN \pdfoptionpdfinclusionerrorlevel \pdf_optionpdfinclusionerrorlevel:D
469 \name_primitive:NN \pdfimagehicolor \pdf_imagehicolor:D
470 \name_primitive:NN \pdfimageapplygamma \pdf_imageapplygamma:D
471 \name_primitive:NN \pdfgamma \pdf_gamma:D
472 \name_primitive:NN \pdfimagegamma \pdf_imagegamma:D
473 %% dimen registers:
474 \name_primitive:NN \pdfhorigin \pdf_horigin:D
475 \name_primitive:NN \pdfvorigin \pdf_vorigin:D
476 \name_primitive:NN \pdfpagewidth \pdf_pagewidth:D
477 \name_primitive:NN \pdfpageheight \pdf_pageheight:D
478 \name_primitive:NN \pdflinkmargin \pdf_linkmargin:D
479 \name_primitive:NN \pdfdestmargin \pdf_destmargin:D
480 \name_primitive:NN \pdfthreadmargin \pdf_threadmargin:D
481 %% token registers:
482 \name_primitive:NN \pdfpagesattr \pdf_pagesattr:D
483 \name_primitive:NN \pdfpageattr \pdf_pageattr:D
484 \name_primitive:NN \pdfpageresources \pdf_pageresources:D
485 \name_primitive:NN \pdfpkmode \pdf_pkmode:D
486 %% expandable commands:
487 \name_primitive:NN \pdftexrevision \pdf_texrevision:D
488 \name_primitive:NN \pdftexbanner \pdf_texbanner:D
489 \name_primitive:NN \pdfcreationdate \pdf_creationdate:D
490 \name_primitive:NN \pdfpageref \pdf_pageref:D
491 \name_primitive:NN \pdfxformname \pdf_xformname:D
492 \name_primitive:NN \pdffontname \pdf_fontname:D
493 \name_primitive:NN \pdffontobjnum \pdf_fontobjnum:D
494 \name_primitive:NN \pdffontsize \pdf_fontsize:D
495 \name_primitive:NN \pdfincludechars \pdf_includechars:D
496 \name_primitive:NN \leftmarginkern \pdf_leftmarginkern:D
497 \name_primitive:NN \rightmarginkern \pdf_rightmarginkern:D
498 \name_primitive:NN \pdfescapestring \pdf_escapestring:D
499 \name_primitive:NN \pdfescapename \pdf_escapename:D
500 \name_primitive:NN \pdfescapehex \pdf_escapehex:D
501 \name_primitive:NN \pdfunescapehex \pdf_unescapehex:D
502 \name_primitive:NN \pdfstrcmp \pdf_strcmp:D
503 \name_primitive:NN \pdfuniformdeviate \pdf_uniformdeviate:D
504 \name_primitive:NN \pdfnormaldeviate \pdf_normaldeviate:D
505 \name_primitive:NN \pdfmdfivesum \pdf_mdfivesum:D
506 \name_primitive:NN \pdffilemoddate \pdf_filemoddate:D
507 \name_primitive:NN \pdffilesize \pdf_filesize:D
508 \name_primitive:NN \pdffiledump \pdf_filedump:D
509 %% read-only integers:
510 \name_primitive:NN \pdftexversion \pdf_texversion:D
511 \name_primitive:NN \pdflastobj \pdf_lastobj:D
512 \name_primitive:NN \pdflastxform \pdf_lastxform:D
513 \name_primitive:NN \pdflastximage \pdf_lastximage:D
514 \name_primitive:NN \pdflastximagepages \pdf_lastximagepages:D
515 \name_primitive:NN \pdflastannot \pdf_lastannot:D
516 \name_primitive:NN \pdflastxpos \pdf_lastxpos:D
517 \name_primitive:NN \pdflastypos \pdf_lastypos:D
518 \name_primitive:NN \pdflastdemerits \pdf_lastdemerits:D
519 \name_primitive:NN \pdfelapsedtime \pdf_elapsedtime:D

```

```

520 \name_primitive:NN \pdfrandomseed      \pdf_randomseed:D
521 \name_primitive:NN \pdfshellescape     \pdf_shellescape:D
522 %% general commands:
523 \name_primitive:NN \pdfobj              \pdf_obj:D
524 \name_primitive:NN \pdfrefobj           \pdf_refobj:D
525 \name_primitive:NN \pdfxform            \pdf_xform:D
526 \name_primitive:NN \pdfrefxform         \pdf_refxform:D
527 \name_primitive:NN \pdfximage           \pdf_ximage:D
528 \name_primitive:NN \pdfrefximage        \pdf_refximage:D
529 \name_primitive:NN \pdfannot            \pdf_annot:D
530 \name_primitive:NN \pdfstartlink        \pdf_startlink:D
531 \name_primitive:NN \pdfendlink          \pdf_endlink:D
532 \name_primitive:NN \pdfoutline          \pdf_outline:D
533 \name_primitive:NN \pdfdest             \pdf_dest:D
534 \name_primitive:NN \pdfthread           \pdf_thread:D
535 \name_primitive:NN \pdfstartthread       \pdf_startthread:D
536 \name_primitive:NN \pdfendthread         \pdf_endthread:D
537 \name_primitive:NN \pdfsavepos          \pdf_savepos:D
538 \name_primitive:NN \pdfinfo             \pdf_info:D
539 \name_primitive:NN \pdfcatalog          \pdf_catalog:D
540 \name_primitive:NN \pdfnames            \pdf_names:D
541 \name_primitive:NN \pdfmapfile          \pdf_mapfile:D
542 \name_primitive:NN \pdfmapline          \pdf_mapline:D
543 \name_primitive:NN \pdffontattr         \pdf_fontattr:D
544 \name_primitive:NN \pdftrailer          \pdf_trailer:D
545 \name_primitive:NN \pdffontexpand       \pdf_fontexpand:D
546 %%\name_primitive:NN \vadjust [<pre spec>] <filler> { <vertical mode material> } (h, m)
547 \name_primitive:NN \pdfliteral          \pdf_literal:D
548 %%\name_primitive:NN \special <pdfspecial spec>
549 \name_primitive:NN \pdfresettimer       \pdf_resettimer:D
550 \name_primitive:NN \pdfsetrandomseed    \pdf_setrandomseed:D
551 \name_primitive:NN \pdfnoligatures      \pdf_noligatures:D

```

We’re ignoring XeTeX and LuaTeX right now except for a check whether they’re in use:

```

552 \name_primitive:NN \XeTeXversion        \xetex_version:D
553 \name_primitive:NN \directlua           \luatex_directlua: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.

```

554 \etex_ifdefined:D \strcmp
555   \etex_ifdefined:D \xetex_version:D
556     \name_primitive:NN \strcmp \pdf_strcmp:D
557   \tex_fi:D
558 \tex_fi:D

```

## 95.6 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 LATEX 3.
\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 `endline` is 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.

```

559 \tex_def:D\ExplSyntaxOn{
560   \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
561   \tex_else:D
562     \tex_edef:D\ExplSyntaxOff{
563       \etex_unexpanded:D{
564         \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
565         \tex_def:D \ExplSyntaxStatus{0}
566       }
567       \tex_catcode:D 126=\tex_the:D \tex_catcode:D 126 \tex_relax:D
568       \tex_catcode:D 32=\tex_the:D \tex_catcode:D 32 \tex_relax:D
569       \tex_catcode:D 9=\tex_the:D \tex_catcode:D 9 \tex_relax:D
570       \tex_endlinechar:D =\tex_the:D \tex_endlinechar:D \tex_relax:D
571       \tex_catcode:D 95=\tex_the:D \tex_catcode:D 95 \tex_relax:D
572       \tex_catcode:D 58=\tex_the:D \tex_catcode:D 58 \tex_relax:D
573       \tex_noexpand:D \tex_fi:D
574     }
575     \tex_def:D\ExplSyntaxStatus{1}
576     \tex_catcode:D 126=10 \tex_relax:D % tilde is a space char.
577     \tex_catcode:D 32=9 \tex_relax:D % space is ignored
578     \tex_catcode:D 9=9 \tex_relax:D % tab also ignored
579     \tex_endlinechar:D =32 \tex_relax:D % endline is space
580     \tex_catcode:D 95=11 \tex_relax:D % underscore letter
581     \tex_catcode:D 58=11 \tex_relax:D % colon letter
582   \tex_fi:D
583 }

```

At this point we better set the status.

```

584 \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 conventions according to space characters. These macros provide the necessary settings.

`\ExplSyntaxNamesOff`

```

585 \tex_def:D \ExplSyntaxNamesOn{
586   \tex_catcode:D '\_ =11\tex_relax:D
587   \tex_catcode:D '\: =11\tex_relax:D
588 }

```

```

589 \tex_def:D \ExplSyntaxNamesOff{
590   \tex_catcode:D '\_ =8\tex_relax:D
591   \tex_catcode:D '\:=12\tex_relax:D
592 }

```

## 95.7 Package loading

```

\GetIdInfo      Extract all information from a cvs or svn field. The formats are slightly different but
\filedescription at least the information is in the same positions so we check in the date format so see
\filename       if it contains a / after the four-digit year. If it does it is cvs else svn and we extract
\fileversion    information. To be on the safe side we ensure that spaces in the argument are seen.
\fileauthor
\filedate
\filenameext
\filetimestamp
\GetIdInfoAux:w
\GetIdInfoAuxii:w
\GetIdInfoAuxCVS:w
\GetIdInfoAuxSVN:w

```

```

593 \tex_def:D\GetIdInfo{
594   \tex_begingroup:D
595   \tex_catcode:D 32=10 \tex_relax:D % needed? Probably for now.
596   \GetIdInfoMaybeMissing:w
597 }

598 \tex_def:D\GetIdInfoMaybeMissing:w$#1$#2{
599   \tex_def:D \l_tmpa_tl {#1}
600   \tex_def:D \l_tmpb_tl {Id}
601   \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
602     \tex_def:D \l_tmpa_tl {
603       \tex_endgroup:D
604       \tex_def:D\filedescription{#2}
605       \tex_def:D\filename      {[unknown~name]}
606       \tex_def:D\fileversion   {000}
607       \tex_def:D\fileauthor    {[unknown~author]}
608       \tex_def:D\filedate      {0000/00/00}
609       \tex_def:D\filenameext   {[unknown~ext]}
610       \tex_def:D\filetimestamp {[unknown~timestamp]}
611     }
612   \tex_else:D
613     \tex_def:D \l_tmpa_tl {\GetIdInfoAux:w$#1$#2}
614   \tex_fi:D
615   \l_tmpa_tl
616 }

617 \tex_def:D\GetIdInfoAux:w$#1~#2.#3~#4~#5~#6~#7~#8$#9{
618   \tex_endgroup:D
619   \tex_def:D\filename{#2}
620   \tex_def:D\fileversion{#4}
621   \tex_def:D\filedescription{#9}
622   \tex_def:D\fileauthor{#7}
623   \GetIdInfoAuxii:w #5\tex_relax:D
624   #3\tex_relax:D#5\tex_relax:D#6\tex_relax:D
625 }

626 \tex_def:D\GetIdInfoAuxii:w #1#2#3#4#5#6\tex_relax:D{
627   \tex_ifx:D#5/
628     \tex_expandafter:D\GetIdInfoAuxCVS:w
629   \tex_else:D
630     \tex_expandafter:D\GetIdInfoAuxSVN:w

```

```

631 \tex_fi:D
632 }

633 \tex_def:D\GetIdInfoAuxCVS:w #1,v\tex_relax:D
634                                #2\tex_relax:D#3\tex_relax:D{
635 \tex_def:D\filedate{#2}
636 \tex_def:D\filenameext{#1}
637 \tex_def:D\filetimestamp{#3}

```

When creating the format we want the information in the log straight away.

```

638 <initex>\tex_immediate:D\tex_write:D-1
639 <initex> {\filename;~ v\fileversion,~\filedate;~\filedescription}
640 }
641 \tex_def:D\GetIdInfoAuxSVN:w #1\tex_relax:D#2-#3-#4
642                                \tex_relax:D#5Z\tex_relax:D{
643 \tex_def:D\filenameext{#1}
644 \tex_def:D\filedate{#2/#3/#4}
645 \tex_def:D\filetimestamp{#5}
646 <-package>\tex_immediate:D\tex_write:D-1
647 <-package> {\filename;~ v\fileversion,~\filedate;~\filedescription}
648 }
649 </initex | package>

```

Finally some corrections in the case we are running over L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>.

We want to set things up so that experimental packages and regular packages can coexist with the former using the L<sup>A</sup>T<sub>E</sub>X3 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 1362 2009-05-28 20:19:21Z joseph $
    {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`

```

650 <*package>
651 \tex_def:D \ProvidesExplPackage#1#2#3#4{
652 \ProvidesPackage{#1}[#2~v#3~#4]
653 \ExplSyntaxOn
654 }

```

```

655 \tex_def:D \ProvidesExplClass#1#2#3#4{
656   \ProvidesClass{#1}[#2~v#3~#4]
657   \ExplSyntaxOn
658 }

```

`\@pushfilename` The idea behind the code is to record whether or not the  $\text{\LaTeX}$ 3 syntax is on or off when about to load a file with class or package extension. This status stored in the parameter `\ExplSyntaxStatus` and set by `\ExplSyntaxOn` and `\ExplSyntaxOff` to 1 and 0 respectively is pushed onto the stack `\ExplSyntaxStack`. Then the catcodes are set back to normal, the file loaded with its options and finally the stack is popped again. The whole thing is a bit problematical. So let's take a look at what the desired behavior is: A package or class which declares itself of Expl type by using `\ProvidesExplClass` or `\ProvidesExplPackage` should automatically ensure the correct catcode scheme as soon as the identification part is over. Similarly, a package or class which uses the traditional `\ProvidesClass` or `\ProvidesPackage` commands should go back to the traditional catcode scheme. An example:

```

\RequirePackage{l3names}
\ProvidesExplPackage{foobar}{2009/05/07}{0.1}{Foobar package}
\cs_new:Nn \foo_bar:nn {#1,#2}
...
\RequirePackage{array}
...
\cs_new:Nn \foo_bar:nnn {#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  $\text{\LaTeX}$  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  $\text{\LaTeX}$  might try to pop the file name stack while the `\ExplSyntaxStack` is empty. If the latter is empty, we just switch off `\ExplSyntax`.

```

659 \tex_edef:D \@pushfilename{
660   \etex_unexpanded:D{
661     \tex_edef:D \ExplSyntaxStack{ \ExplSyntaxStatus \ExplSyntaxStack }
662     \ExplSyntaxOff
663   }
664   \etex_unexpanded:D\tex_expandafter:D{\@pushfilename }
665 }
666 \tex_edef:D \@popfilename{
667   \etex_unexpanded:D\tex_expandafter:D{\@popfilename
668     \tex_if:D 2\ExplSyntaxStack 2
669     \ExplSyntaxOff
670     \tex_else:D
671     \tex_expandafter:D\ExplSyntaxPopStack\ExplSyntaxStack\q_nil
672     \tex_fi:D
673   }
674 }

```

\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.

```

675 \tex_def:D\ExplSyntaxPopStack#1#2\q_nil{
676   \tex_def:D\ExplSyntaxStack{#2}
677   \tex_ifodd:D#1\tex_relax:D
678   \ExplSyntaxOn
679   \tex_else:D
680   \ExplSyntaxOff
681   \tex_fi:D
682 }
683 \tex_def:D \ExplSyntaxStack{0}

```

## 95.8 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.

```

684 \tex_let:D\tex_input:D      \@@input
685 \tex_let:D\tex_underline:D  \@@underline
686 \tex_let:D\tex_end:D        \@@end
687 \tex_let:D\tex_everymath:D  \frozen@everymath
688 \tex_let:D\tex_everydisplay:D \frozen@everydisplay
689 \tex_let:D\tex_italiccor:D  \@@italiccorr
690 \tex_let:D\tex_hyphen:D     \@@hyph

```

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.

```

691 \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>:

```

692 \tex_ifx:D\name_undefine:N\@gobble
693   \tex_def:D\name_pop_stack:w{}
694 \tex_else:D

```

But if traditional  $\TeX$  code is disabled, do this...

As mentioned above, The  $\text{\LaTeX} 2_\epsilon$  package mechanism will insert some code to handle the filename stack, and reset the package options, this code will die if the  $\TeX$  primitives have gone, so skip past it and insert some equivalent code that will work.

First a version of `\ProvidesPackage` that can cope.

```

695 \tex_def:D\ProvidesPackage{
696   \tex_begingroup:D
697   \ExplSyntaxOff
698   \package_provides:w}

699 \tex_def:D\package_provides:w#1#2[#3]{
700   \tex_endgroup:D
701   \tex_immediate:D\tex_write:D-1{Package:~#1#2~#3}
702   \tex_expandafter:D\tex_xdef:D
703     \tex_csname:D ver@#1.sty\tex_endcsname:D{#1}}

```

In this case the catcode preserving stack is not maintained and `\ExplSyntaxOn` conventions stay in force once on. You'll need to turn then off explicitly with `\ExplSyntaxOff` (although as currently built on 2e, nothing except very experimental code will run in this mode!) Also note that `\RequirePackage` is a simple definition, just for one file, with no options.

```

704 \tex_def:D\name_pop_stack:w#1\relax{%
705   \ExplSyntaxOff
706   \tex_expandafter:D\@p@pfilename\@currnamestack\@nil
707   \tex_let:D\default@ds\@unknownoptionerror
708   \tex_global:D\tex_let:D\ds@\@empty
709   \tex_global:D\tex_let:D\@declaredoptions\@empty}

710 \tex_def:D\@p@pfilename#1#2#3#4\@nil{%
711   \tex_gdef:D\@currname{#1}%
712   \tex_gdef:D\@currentx{#2}%
713   \tex_catcode:D'\@#3%
714   \tex_gdef:D\@currnamestack{#4}}

715 \tex_def:D\NeedsTeXFormat#1{}
716 \tex_def:D\RequirePackage#1{
717   \tex_expandafter:D\tex_ifx:D
718     \tex_csname:D ver@#1.sty\tex_endcsname:D\tex_relax:D
719     \ExplSyntaxOn
720     \tex_input:D#1.sty\tex_relax:D
721   \tex_fi:D}
722 \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.)

```

723 \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.

```

724 <*\linitex>
725 \tex_def:D \package_check_loaded_expl: {
726   \@ifpackageloaded{expl3}{\{
727     \PackageError{expl3}{Cannot~load~the~expl3~modules~separately}{
728       The~expl3~modules~cannot~be~loaded~separately;\MessageBreak
729       please~\protect\usepackage{expl3}~instead.
730     }
731   }
732 }
733 <\/\linitex>

734 <\/package>

```

## 95.9 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.

```

735 <*\showmemory>
736 \g_trace_statistics_status=2\scan_stop:
737 \cs_set_nopar:Npn\showMemUsage{
738   \if_horizontal_mode:
739     \tex_errmessage:D{Wrong~ mode~ H:~ something~ triggered~
740       hmode~ above}
741   \else:
742     \tex_message:D{Mode ~ okay}
743   \fi:
744   \tex_shipout:D\hbox:w{}
745 }
746 \showMemUsage
747 <\/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>

```

748 <*package>
749 \ProvidesExplPackage
750   {\filename}{\filedate}{\fileversion}{\filedescription}
751 \package_check_loaded_expl:
752 </package>
753 <*initex | package>
754 \tex_let:D \cs_set_eq:NwN          \tex_let:D

\if_true: Then some conditionals.
\if_false:
  \or:      755 \cs_set_eq:NwN \if_true:      \tex_iftrue:D
  \else:    756 \cs_set_eq:NwN \if_false:     \tex_iffalse:D
  \fi:      757 \cs_set_eq:NwN \or:          \tex_or:D
\reverse_if:N 758 \cs_set_eq:NwN \else:        \tex_else:D
  \if:w     759 \cs_set_eq:NwN \fi:          \tex_fi:D
\if_bool:N    760 \cs_set_eq:NwN \reverse_if:N \etex_unless:D
\if_predicate:w 761 \cs_set_eq:NwN \if:w          \tex_if:D
\if_charcode:w 762 \cs_set_eq:NwN \if_bool:N     \tex_ifodd:D
\if_catcode:w 763 \cs_set_eq:NwN \if_predicate:w \tex_ifodd:D
              764 \cs_set_eq:NwN \if_charcode:w \tex_if:D
              765 \cs_set_eq:NwN \if_catcode:w  \tex_ifcat:D

\if_meaning:w
              766 \cs_set_eq:NwN \if_meaning:w  \tex_ifx:D

\if_mode_math: TEX lets us detect some if its modes.
\if_mode_horizontal:
\if_mode_vertical:
\if_mode_inner:
              767 \cs_set_eq:NwN \if_mode_math:    \tex_ifmmode:D
              768 \cs_set_eq:NwN \if_mode_horizontal: \tex_ifhmode:D
              769 \cs_set_eq:NwN \if_mode_vertical:   \tex_ifvmode:D
              770 \cs_set_eq:NwN \if_mode_inner:     \tex_ifinner:D

\if_cs_exist:N
\if_cs_exist:w
              771 \cs_set_eq:NwN \if_cs_exist:N     \etex_ifdefined:D
              772 \cs_set_eq:NwN \if_cs_exist:w     \etex_ifcurname:D

\exp_after:wN The three \exp_ functions are used in the l3expan module where they are described.
\exp_not:N
\exp_not:n
              773 \cs_set_eq:NwN \exp_after:wN      \tex_expandafter:D
              774 \cs_set_eq:NwN \exp_not:N         \tex_noexpand:D
              775 \cs_set_eq:NwN \exp_not:n         \etex_unexpanded:D

\iow_shipout_x:Nn
\token_to_meaning:N
\token_to_str:N
\token_to_str:c
\cs:w
\cs_end:
\cs_meaning:N
\cs_meaning:c
\cs_show:N
\cs_show:c
              776 \cs_set_eq:NwN \iow_shipout_x:Nn \tex_write:D
              777 \cs_set_eq:NwN \token_to_meaning:N \tex_meaning:D
              778 \cs_set_eq:NwN \token_to_str:N   \tex_string: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.

```

779 \cs_set_eq:NwN \cs:w \tex_csname:D
780 \cs_set_eq:NwN \cs_end: \tex_endcsname:D
781 \cs_set_eq:NwN \cs_meaning:N \tex_meaning:D
782 \tex_def:D \cs_meaning:c {\exp_args:Nc\cs_meaning:N}
783 \cs_set_eq:NwN \cs_show:N \tex_show:D
784 \tex_def:D \cs_show:c {\exp_args:Nc\cs_show:N}
785 \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  
`\group_begin:` alignments. These safe versions are defined in the `l3prg` module.

`\group_end:`

```

786 \cs_set_eq:NwN \scan_stop: \tex_relax:D
787 \cs_set_eq:NwN \group_begin: \tex_begingroup:D
788 \cs_set_eq:NwN \group_end: \tex_endgroup:D

```

`\group_execute_after:N`

```

789 \cs_set_eq:NwN \group_execute_after:N \tex_aftergroup:D

```

`\pref_global:D`

`\pref_long:D`

`\pref_protected:D`

```

790 \cs_set_eq:NwN \pref_global:D \tex_global:D
791 \cs_set_eq:NwN \pref_long:D \tex_long:D
792 \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.

```

793 \cs_set_eq:NwN \cs_set_nopar:Npn \tex_def:D
794 \cs_set_eq:NwN \cs_set_nopar:Npx \tex_edef:D
795 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npn {
796 \pref_long:D \cs_set_nopar:Npn
797 }
798 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npx {
799 \pref_long:D \cs_set_nopar:Npx
800 }
801 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn {
802 \pref_protected:D \cs_set_nopar:Npn
803 }
804 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx {
805 \pref_protected:D \cs_set_nopar:Npx
806 }
807 \cs_set_protected_nopar:Npn \cs_set_protected:Npn {
808 \pref_protected:D \pref_long:D \cs_set_nopar:Npn
809 }
810 \cs_set_protected_nopar:Npn \cs_set_protected:Npx {
811 \pref_protected:D \pref_long:D \cs_set_nopar:Npx
812 }

```

<code>\cs_gset_nopar:Npn</code>	Global versions of the above functions.
<code>\cs_gset_nopar:Npx</code>	
<code>\cs_gset:Npn</code>	813 <code>\cs_set_eq:NwN \cs_gset_nopar:Npn \tex_gdef:D</code>
<code>\cs_gset:Npx</code>	814 <code>\cs_set_eq:NwN \cs_gset_nopar:Npx \tex_xdef:D</code>
<code>\cs_gset_protected_nopar:Npn</code>	815 <code>\cs_set_protected_nopar:Npn \cs_gset:Npn {</code>
<code>\cs_gset_protected_nopar:Npx</code>	816 <code>\pref_long:D \cs_gset_nopar:Npn</code>
<code>\cs_gset_protected:Npn</code>	817 <code>}</code>
<code>\cs_gset_protected:Npx</code>	818 <code>\cs_set_protected_nopar:Npn \cs_gset:Npx {</code>
	819 <code>\pref_long:D \cs_gset_nopar:Npx</code>
	820 <code>}</code>
	821 <code>\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn {</code>
	822 <code>\pref_protected:D \cs_gset_nopar:Npn</code>
	823 <code>}</code>
	824 <code>\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx {</code>
	825 <code>\pref_protected:D \cs_gset_nopar:Npx</code>
	826 <code>}</code>
	827 <code>\cs_set_protected_nopar:Npn \cs_gset_protected:Npn {</code>
	828 <code>\pref_protected:D \pref_long:D \cs_gset_nopar:Npn</code>
	829 <code>}</code>
	830 <code>\cs_set_protected_nopar:Npn \cs_gset_protected:Npx {</code>
	831 <code>\pref_protected:D \pref_long:D \cs_gset_nopar:Npx</code>
	832 <code>}</code>

## 96.3 Selecting tokens

`\use:c` This macro grabs its argument and returns a csname from it.

```
833 \cs_set:Npn \use:c #1 { \cs:w#1\cs_end: }
```

`\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
834 \cs_set:Npn \use:n #1 {#1}
835 \cs_set:Npn \use:nn #1#2 {#1#2}
836 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
837 \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.

```

838 \cs_set:Npn \use_i:nn #1#2 {#1}
839 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

`\use_i:nnn` We also need something for picking up arguments from a longer list.

```

\use_ii:nnn
\use_iii:nnn
\use_i:nnnn
\use_ii:nnnn
\use_iii:nnnn
\use_iv:nnnn
\use_i_ii:nnn
840 \cs_set:Npn \use_i:nnn #1#2#3{#1}
841 \cs_set:Npn \use_ii:nnn #1#2#3{#2}
842 \cs_set:Npn \use_iii:nnn #1#2#3{#3}
843 \cs_set:Npn \use_i:nnnn #1#2#3#4{#1}
844 \cs_set:Npn \use_ii:nnnn #1#2#3#4{#2}
```

```

845 \cs_set:Npn \use_iii:nnnn #1#2#3#4{#3}
846 \cs_set:Npn \use_iv:nnnn #1#2#3#4{#4}
847 \cs_set:Npn \use_i_ii:nnn #1#2#3{#1#2}

```

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil` or `\q_stop` resp.

`\use_none_delimit_by_q_stop:w`

`delimit_by_q_recursion_stop:w`

```

848 \cs_set:Npn \use_none_delimit_by_q_nil:w #1\q_nil{}
849 \cs_set:Npn \use_none_delimit_by_q_stop:w #1\q_stop{}
850 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop {}

```

`\use_i_delimit_by_q_nil:nw`

`\use_i_delimit_by_q_stop:nw`

`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.

```

851 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2\q_nil{#1}
852 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2\q_stop{#1}
853 \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.

```

854 \cs_set:Npn \use_i_after_fi:nw #1\fi:{\fi: #1}
855 \cs_set:Npn \use_i_after_else:nw #1\else:#2\fi:{\fi: #1}
856 \cs_set:Npn \use_i_after_or:nw #1\or: #2\fi: {\fi:#1}
857 \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.

```

858 \cs_set:Npn \use_none:n #1{}
859 \cs_set:Npn \use_none:nn #1#2{}
860 \cs_set:Npn \use_none:nnn #1#2#3{}
861 \cs_set:Npn \use_none:nnnn #1#2#3#4{}
862 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5{}
863 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6{}
864 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7{}
865 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8{}
866 \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.

```

867 \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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X in the correct state.

```
868 \cs_set:Npn \prg_return_true: { \exp_after:wN\if_true:\tex_romannumeral:D }
869 \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\tex_ifcase:D \exp_after:wN \c_one \tex_romannumeral:D
}
\cs_set:Npn \prg_return_error: {
\exp_after:wN\tex_ifcase:D \exp_after:wN \c_two \tex_romannumeral:D
}
```

`\prg_set_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.

`\prg_new_conditional:Npnn`

`\et_protected_conditional:Npnn`

`\ew_protected_conditional:Npnn`

```
870 \cs_set:Npn \prg_set_conditional:Npnn #1{
871 \prg_get_parm_aux:nw{
872 \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
873 \cs_set:Npn {parm}
874 }
875 }
876 \cs_set:Npn \prg_new_conditional:Npnn #1{
877 \prg_get_parm_aux:nw{
```

```

878     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
879     \cs_new:Npn {parm}
880   }
881 }
882 \cs_set:Npn \prg_set_protected_conditional:Npnn #1{
883   \prg_get_parm_aux:nw{
884     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
885     \cs_set_protected:Npn {parm}
886   }
887 }
888 \cs_set:Npn \prg_new_protected_conditional:Npnn #1{
889   \prg_get_parm_aux:nw{
890     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
891     \cs_new_protected:Npn {parm}
892   }
893 }

```

\prg\_set\_conditional:Nnn    The user functions for the types automatically inserting the correct parameter text based  
 \prg\_new\_conditional:Nnn    on the signature. Call aux function after calculating number of arguments, split the base  
 set\_protected\_conditional:Nnn function into name and signature and then use, e.g., \cs\_set:Npn to define it with.  
 new\_protected\_conditional:Nnn

```

894 \cs_set:Npn \prg_set_conditional:Nnn #1{
895   \exp_args:Nnf \prg_get_count_aux:nn{
896     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
897     \cs_set:Npn {count}
898   }{\cs_get_arg_count_from_signature:N #1}
899 }
900 \cs_set:Npn \prg_new_conditional:Nnn #1{
901   \exp_args:Nnf \prg_get_count_aux:nn{
902     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
903     \cs_new:Npn {count}
904   }{\cs_get_arg_count_from_signature:N #1}
905 }
906
907 \cs_set:Npn \prg_set_protected_conditional:Nnn #1{
908   \exp_args:Nnf \prg_get_count_aux:nn{
909     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
910     \cs_set_protected:Npn {count}
911   }{\cs_get_arg_count_from_signature:N #1}
912 }
913
914 \cs_set:Npn \prg_new_protected_conditional:Nnn #1{
915   \exp_args:Nnf \prg_get_count_aux:nn{
916     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
917     \cs_new_protected:Npn {count}
918   }{\cs_get_arg_count_from_signature:N #1}
919 }

```

\prg\_get\_parm\_aux:nw    For the Npnn type we must grab the parameter text before continuing. We make this  
 \prg\_get\_count\_aux:nn    a very generic function that takes one argument before reading everything up to a left  
                          brace. Something similar for the Nnn type.

```

920 \cs_set:Npn \prg_get_count_aux:nn #1#2 {#1{#2}}
921 \cs_set:Npn \prg_get_parm_aux:nw #1#2#{#1{#2}}

```

conditional\_parm\_aux:nnNNnnnn  
 erate\_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.

```

922 \cs_set:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8{
923   \prg_generate_conditional_aux:nnw{#5}{
924     #4{#1}{#2}{#6}{#8}
925   }#7,?, \q_recursion_stop
926 }

```

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.

```

927 \cs_set:Npn \prg_generate_conditional_aux:nnw #1#2#3,{
928   \if:w ?#3
929     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
930   \fi:
931   \use:c{prg_generate_#3_form_#1:Nnnnn} #2
932   \prg_generate_conditional_aux:nnw{#1}{#2}
933 }

```

rg\_generate\_p\_form\_parm:Nnnnn  
 g\_generate\_TF\_form\_parm:Nnnnn  
 rg\_generate\_T\_form\_parm:Nnnnn  
 rg\_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.

```

934 \cs_set:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5{
935   \exp_args:Nc #1 {#2_p:#3}#4{#5 \c_zero
936     \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
937   }
938 }
939 \cs_set:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5{
940   \exp_args:Nc#1 {#2:#3TF}#4{#5 \c_zero
941     \exp_after:wN \use_i:nn \else: \exp_after:wN \use_ii:nn \fi:
942   }
943 }
944 \cs_set:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5{
945   \exp_args:Nc#1 {#2:#3T}#4{#5 \c_zero
946     \else:\exp_after:wN\use_none:nn\fi:\use:n
947   }
948 }
949 \cs_set:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5{
950   \exp_args:Nc#1 {#2:#3F}#4{#5 \c_zero
951     \exp_after:wN\use_none:nn\fi:\use:n
952   }
953 }

```

g\_generate\_p\_form\_count:Nnnnn  
 generate\_TF\_form\_count:Nnnnn  
 g\_generate\_T\_form\_count:Nnnnn  
 g\_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.



```

954 \cs_set:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5{
955   \cs_generate_from_arg_count:cNnn {#2_p:#3} #1 {#4}{#5 \c_zero
956     \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
957   }
958 }
959 \cs_set:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5{
960   \cs_generate_from_arg_count:cNnn {#2:#3TF} #1 {#4}{#5 \c_zero
961     \exp_after:wN\use_i:nn\else:\exp_after:wN\use_ii:nn\fi:
962   }
963 }
964 \cs_set:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5{
965   \cs_generate_from_arg_count:cNnn {#2:#3T} #1 {#4}{#5 \c_zero
966     \else:\exp_after:wN\use_none:nn\fi:\use:n
967   }
968 }
969 \cs_set:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5{
970   \cs_generate_from_arg_count:cNnn {#2:#3F} #1 {#4}{#5 \c_zero
971     \exp_after:wN\use_none:nn\fi:\use:n
972   }
973 }

```

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
974 \tex_chardef:D \c_true_bool = 1~
975 \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
\cs_to_str_aux:w leading escape character. This turns out to be a non-trivial matter as there are different
cases:

```

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of `\tex_escapechar:D` is negative;
- when the escape character is a space.

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):

- $\langle X \rangle ax$

- aax
- aax

In the second and third case, putting an auxiliary function in front reading unlimited 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):

```

976 \cs_set_nopar:Npn \cs_to_str:N {
977   \if:w \exp_after:wN \cs_str_aux:w\token_to_str:N \a ax\q_nil
978   \else:
979     \exp_after:wN \exp_after:wN\exp_after:wN \use_ii:nn
980   \fi:
981   \exp_after:wN \use_none:n \token_to_str:N
982 }
983 \cs_set:Npn \cs_str_aux:w #1#2#3#4\q_nil{#2#3}

```

```

\cs_split_function:NN
\cs_split_function_aux:w
\cs_split_function_auxii:w

```

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean *<true>* or *<false>* is returned with *<true>* for when there is a colon in the function and *<false>* if there is not. Lastly, the second argument of `\cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `\cs_split_function:NN\foo_bar:cnx\use_i:nnn` as input becomes `\use_i:nnn {foo_bar}{cnx}\c_true_bool`.

Can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

```

984 \group_begin:
985   \tex_lccode:D '\@ = '\: \scan_stop:
986   \tex_catcode:D '\@ = 12~
987   \tex_lowercase:D {
988     \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.

```

989 \cs_set:Npn \cs_split_function:NN #1#2{
990   \exp_after:wN \cs_split_function_aux:w
991   \tex_romannumeral:D -'\q \cs_to_str:N #1 @a \q_nil #2
992 }

```

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.

```

993 \cs_set:Npn \cs_split_function_aux:w #1@#2#3\q_nil#4{
994   \if_meaning:w a#2
995     \exp_after:wN \use_i:nn

```

```

996 \else:
997   \exp_after:wN\use_ii:nn
998 \fi:
999 {#4{#1}{}}\c_false_bool}
1000 {\cs_split_function_auxii:w#2#3\q_nil #4{#1}}
1001 }
1002 \cs_set:Npn \cs_split_function_auxii:w #1@a\q_nil#2#3{
1003   #2{#3}{#1}\c_true_bool
1004 }

```

End of lowercase

```

1005 }

```

`\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.

```

1006 \cs_set:Npn \cs_get_function_name:N #1 {
1007   \cs_split_function:NN #1\use_i:nnn
1008 }
1009 \cs_set:Npn \cs_get_function_signature:N #1 {
1010   \cs_split_function:NN #1\use_ii:nnn
1011 }

```

## 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`.

```

1012 \prg_set_conditional:Npnn \cs_if_exist:N #1 {p,TF,T,F}{
1013   \if_meaning:w #1\tex_relax:D
1014   \prg_return_false:
1015   \else:
1016     \if_cs_exist:N #1
1017     \prg_return_true:
1018     \else:
1019       \prg_return_false:
1020     \fi:
1021   \fi:
1022 }

```

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.

```

1023 \prg_set_conditional:Npnn \cs_if_exist:c #1 {p,TF,T,F}{
1024   \if_cs_exist:w #1 \cs_end:
1025     \exp_after:wN \use_i:nn
1026   \else:
1027     \exp_after:wN \use_ii:nn
1028   \fi:
1029   {
1030     \exp_after:wN \if_meaning:w \cs:w #1\cs_end: \tex_relax:D
1031     \prg_return_false:
1032   \else:
1033     \prg_return_true:
1034   \fi:
1035   }
1036 \prg_return_false:
1037 }

```

`\cs_if_do_not_use_p:N`  
`\cs_if_do_not_use_aux:nnN`

```

1038 \cs_set:Npn \cs_if_do_not_use_p:N #1{
1039   \cs_split_function:NN #1 \cs_if_do_not_use_aux:nnN
1040 }
1041 \cs_set:Npn \cs_if_do_not_use_aux:nnN #1#2#3{
1042   \exp_after:wN\str_if_eq_p:nn \token_to_str:N D {#2}
1043 }

```

`\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.

```

1044 \prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
1045   \tex_ifnum:D \cs_if_exist_p:N #1 =\c_zero
1046   \exp_after:wN \use_i:nn
1047   \else:
1048     \exp_after:wN \use_ii:nn
1049   \fi:
1050   {
1051     \tex_ifnum:D \cs_if_do_not_use_p:N #1 =\c_zero
1052     \prg_return_true:

```

```

1053     \else:
1054         \prg_return_false:
1055     \fi:
1056 }
1057 \prg_return_false:
1058 }
1059 \cs_set_nopar:Npn \cs_if_free_p:c{\exp_args:Nc\cs_if_free_p:N}
1060 \cs_set_nopar:Npn \cs_if_free:cTF{\exp_args:Nc\cs_if_free:NTF}
1061 \cs_set_nopar:Npn \cs_if_free:cT{\exp_args:Nc\cs_if_free:NT}
1062 \cs_set_nopar:Npn \cs_if_free:cF{\exp_args:Nc\cs_if_free:NF}

```

## 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 log and the terminal and `\c_zero` which is used by some functions in the `l3num` module.

`\c_zero`

`\c_sixteen` The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can't be used until the allocation has been set up properly! The actual allocation mechanism is in `l3alloc` and as  $\TeX$  wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.

```

1063 <!\initex>
1064 \cs_set_eq:NwN \c_minus_one\m@ne
1065 </!\initex>
1066 <!\package>
1067 \tex_countdef:D \c_minus_one = 10 ~
1068 \c_minus_one = -1 ~
1069 </!\package>
1070 \tex_chardef:D \c_sixteen = 16~
1071 \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.

```

1072 \cs_set_nopar:Npn \iow_log:x {
1073     \tex_immediate:D \iow_shipout_x:Nn \c_minus_one
1074 }
1075 \cs_set_nopar:Npn \iow_term:x {
1076     \tex_immediate:D \iow_shipout_x:Nn \c_sixteen
1077 }

```

`\msg_kernel_bug:x` This will show internal errors.

```

1078 \cs_set_nopar:Npn \msg_kernel_bug:x #1 {

```

```

1079 \iow_term:x { This~is~a~LaTeX~bug:~check~coding! }
1080 \tex_errmessage:D {#1}
1081 }

```

`\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.

```

1082 <*trace>
1083 \cs_set:Npn \cs_record_meaning:N #1{}
1084 </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 `<csname>` is undefined or `\scan_stop:.` Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```

1085 \cs_set_nopar:Npn \chk_if_free_cs:N #1{
1086   \cs_if_free:NF #1
1087   {
1088     \msg_kernel_bug:x {Command~name~'\token_to_str:N #1'~
1089                       already~defined!~
1090                       Current~meaning:~'\token_to_meaning:N #1}
1091   }
1092 }
1093 <*trace>
1094 \cs_record_meaning:N#1
1095 % \iow_term:x{Defining~'\token_to_str:N #1~on~%}
1096 \iow_log:x{Defining~'\token_to_str:N #1~on~
1097           line~\tex_the:D \tex_inputlineno:D}
1098 </trace>
1099 }

```

`\chk_if_exist_cs:N` This function issues a warning message when the control sequence in its argument does not exist.

```

1100 \cs_set_nopar:Npn \chk_if_exist_cs:N #1 {
1101   \cs_if_exist:NF #1
1102   {
1103     \msg_kernel_bug:x {Command~ '\token_to_str:N #1'~
1104                       not~ yet~ defined!}
1105   }
1106 }
1107 \cs_set_nopar:Npn \chk_if_exist_cs:c {\exp_args:Nc \chk_if_exist_cs:N }

```

`\str_if_eq_p:nn` Takes 2 lists of characters as arguments and expands into `\c_true_bool` if they are equal, and `\c_false_bool` otherwise. Note that in the current implementation spaces in these strings are ignored.<sup>8</sup>

```

1108 \prg_set_conditional:Npnn \str_if_eq:nn #1#2{p}{
1109   \str_if_eq_p_aux:w #1\scan_stop:\#2\scan_stop:\

```

---

<sup>8</sup>This is a function which could use `\tlist_compare:xx`.

```

1110 }
1111 \cs_set_nopar:Npn \str_if_eq_p_aux:w #1#2\\#3#4\\{
1112   \if_meaning:w#1#3
1113     \if_meaning:w#1\scan_stop:\prg_return_true: \else:
1114     \if_meaning:w#3\scan_stop:\prg_return_false: \else:
1115     \str_if_eq_p_aux:w #2\\#4\\fi:fi:
1116   \else:\prg_return_false: \fi:}

```

`\cs_if_eq_name_p:NN` An application of the above function, already streamlined for speed, so I put it in here. It takes two control sequences as arguments and expands into true iff they have the same name. We make it long in case one of them is `\par`!

```

1117 \prg_set_conditional:Npnn \cs_if_eq_name:NN #1#2{p}{
1118   \exp_after:wN\exp_after:wN
1119   \exp_after:wN\str_if_eq_p_aux:w
1120   \exp_after:wN\token_to_str:N
1121   \exp_after:wN#1
1122   \exp_after:wN\scan_stop:
1123   \exp_after:wN\
1124   \token_to_str:N#2\scan_stop:\\}

```

`\str_if_eq_var_p:nf` A variant of `\str_if_eq_p:nn` which has the advantage of obeying spaces in at least the second argument. See `l3quark` for an application. From the hand of David Kastrup with slight modifications to make it fit with the remainder of the `expl3` language.

The macro builds a string of `\if:w` `\fi:` pairs from the first argument. The idea is to turn the comparison of `ab` and `cde` into

```

\tex_number:D
  \if:w \scan_stop: \if:w b\if:w a cde\scan_stop: '\fi: \fi: \fi:
13

```

The `'` is important here. If all tests are true, the `'` is read as part of the number in which case the returned number is 13 in octal notation so `\tex_number:D` returns 11. If one test returns false the `'` is never seen and then we get just 13. We wrap the whole process in an external `\if:w` in order to make it return either `\c_true_bool` or `\c_false_bool` since some parts of `l3prg` expect a predicate to return one of these two tokens.

```

1125 \prg_set_conditional:Npnn \str_if_eq_var:nf #1#2 {p} {
1126   \if:w \tex_number:D\str_if_eq_var_start:nnN{}{}#1\scan_stop:{#2}
1127 }
1128 \cs_set_nopar:Npn\str_if_eq_var_start:nnN#1#2#3{
1129   \if:w#3\scan_stop:\exp_after:wN\str_if_eq_var_stop:w\fi:
1130   \str_if_eq_var_start:nnN{\if:w#3#1}{#2\fi:}
1131 }
1132 \cs_set:Npn\str_if_eq_var_stop:w\str_if_eq_var_start:nnN#1#2#3{
1133   #1#3\scan_stop:#213~\prg_return_true:\else:\prg_return_false:\fi:
1134 }

```

## 96.10 More new definitions

`\cs_new_nopar:Npn` These are like `\cs_set_nopar:Npn` and `\cs_set_eq:NN`, but they first check that the argument command is not already in use. You may use `\pref_global:D`, `\pref_long:D`, `\pref_protected:D`, and `\tex_outer:D` as prefixes.

```

\cs_new_nopar:Npn
\cs_new_nopar:Npx
  \cs_new:Npn
  \cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
  \cs_new_protected:Npn
  \cs_new_protected:Npx
1135 \cs_set:Npn \cs_tmp:w #1#2{
1136   \cs_set_protected_nopar:Npn #1 ##1 {
1137     \chk_if_free_cs:N ##1
1138     #2 ##1
1139   }
1140 }
1141 \cs_tmp:w \cs_new_nopar:Npn \cs_set_nopar:Npn
1142 \cs_tmp:w \cs_new_nopar:Npx \cs_set_nopar:Npx
1143 \cs_tmp:w \cs_new:Npn \cs_set:Npn
1144 \cs_tmp:w \cs_new:Npx \cs_set:Npx
1145 \cs_tmp:w \cs_new_protected_nopar:Npn \cs_set_protected_nopar:Npn
1146 \cs_tmp:w \cs_new_protected_nopar:Npx \cs_set_protected_nopar:Npx
1147 \cs_tmp:w \cs_new_protected:Npn \cs_set_protected:Npn
1148 \cs_tmp:w \cs_new_protected:Npx \cs_set_protected:Npx

```

`\cs_gnew_nopar:Npn` Global versions of the above functions.

```

\cs_gnew_nopar:Npn
  \cs_gnew:Npn
  \cs_gnew:Npx
\cs_gnew_protected_nopar:Npn
\cs_gnew_protected_nopar:Npx
  \cs_gnew_protected:Npn
  \cs_gnew_protected:Npx
1149 \cs_tmp:w \cs_gnew_nopar:Npn \cs_gset_nopar:Npn
1150 \cs_tmp:w \cs_gnew_nopar:Npx \cs_gset_nopar:Npx
1151 \cs_tmp:w \cs_gnew:Npn \cs_gset:Npn
1152 \cs_tmp:w \cs_gnew:Npx \cs_gset:Npx
1153 \cs_tmp:w \cs_gnew_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1154 \cs_tmp:w \cs_gnew_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1155 \cs_tmp:w \cs_gnew_protected:Npn \cs_gset_protected:Npn
1156 \cs_tmp:w \cs_gnew_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 *cname* argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn`*<string>**<rep-text>* will turn *<string>* into a *cname* 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.

```

1157 \cs_set:Npn \cs_tmp:w #1#2{
1158   \cs_new_nopar:Npn #1 { \exp_args:Nc #2 }
1159 }
1160 \cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1161 \cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1162 \cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1163 \cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1164 \cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1165 \cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx
1166 \cs_tmp:w \cs_gnew_nopar:cpn \cs_gnew_nopar:Npn
1167 \cs_tmp:w \cs_gnew_nopar:cpx \cs_gnew_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
\cs_gnew:cpn
\cs_gnew:cpx
1168 \cs_tmp:w \cs_set:cpn \cs_set:Npn
1169 \cs_tmp:w \cs_set:cpx \cs_set:Npx
1170 \cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1171 \cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1172 \cs_tmp:w \cs_new:cpn \cs_new:Npn
1173 \cs_tmp:w \cs_new:cpx \cs_new:Npx
1174 \cs_tmp:w \cs_gnew:cpn \cs_gnew:Npn
1175 \cs_tmp:w \cs_gnew:cpx \cs_gnew: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
\cs_gnew_protected_nopar:cpn
\cs_gnew_protected_nopar:cpx
1176 \cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1177 \cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1178 \cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1179 \cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1180 \cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1181 \cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx
1182 \cs_tmp:w \cs_gnew_protected_nopar:cpn \cs_gnew_protected_nopar:Npn
1183 \cs_tmp:w \cs_gnew_protected_nopar:cpx \cs_gnew_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
\cs_gnew_protected:cpn
\cs_gnew_protected:cpx
1184 \cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1185 \cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1186 \cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1187 \cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1188 \cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1189 \cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx
1190 \cs_tmp:w \cs_gnew_protected:cpn \cs_gnew_protected:Npn
1191 \cs_tmp:w \cs_gnew_protected:cpx \cs_gnew_protected:Npx
```

\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:
1192 \cs_set_nopar:cpn{use_0_parameter:}{}
1193 \cs_set_nopar:cpn{use_1_parameter:}{{##1}}
1194 \cs_set_nopar:cpn{use_2_parameter:}{{##1}{##2}}
1195 \cs_set_nopar:cpn{use_3_parameter:}{{##1}{##2}{##3}}
1196 \cs_set_nopar:cpn{use_4_parameter:}{{##1}{##2}{##3}{##4}}
1197 \cs_set_nopar:cpn{use_5_parameter:}{{##1}{##2}{##3}{##4}{##5}}
1198 \cs_set_nopar:cpn{use_6_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}}
1199 \cs_set_nopar:cpn{use_7_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}{##7}}
1200 \cs_set_nopar:cpn{use_8_parameter:}{
1201   {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}}
1202 \cs_set_nopar:cpn{use_9_parameter:}{
1203   {##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` The = sign allows us to define funny char tokens like = itself or `\_` with this function. For `\cs_set_eq:Nc` the definition of `\c_space_chartok{~}` to work we need the `~` after the =.

`\cs_set_eq:cc`

`\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.

```
1204 \cs_set_protected:Npn \cs_set_eq:NN #1 { \cs_set_eq:NwN #1=~ }
1205 \cs_set_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1206 \cs_set_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1207 \cs_set_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`

```
1208 \cs_new_protected:Npn \cs_new_eq:NN #1 {
1209   \chk_if_free_cs:N #1
1210   \cs_set_eq:NN #1
1211 }
1212 \cs_new_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1213 \cs_new_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1214 \cs_new_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`

```
1215 \cs_new_protected:Npn \cs_gset_eq:NN { \pref_global:D \cs_set_eq:NN }
1216 \cs_new_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1217 \cs_new_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1218 \cs_new_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
```

`\cs_gnew_eq:NN`

`\cs_gnew_eq:cN`

`\cs_gnew_eq:Nc`

`\cs_gnew_eq:cc`

```
1219 \cs_new_protected:Npn \cs_gnew_eq:NN #1 {
1220   \chk_if_free_cs:N #1
1221   \pref_global:D \cs_set_eq:NN #1
1222 }
1223 \cs_new_nopar:Npn \cs_gnew_eq:cN { \exp_args:Nc \cs_gnew_eq:NN }
1224 \cs_new_nopar:Npn \cs_gnew_eq:Nc { \exp_args:NNc \cs_gnew_eq:NN }
1225 \cs_new_nopar:Npn \cs_gnew_eq:cc { \exp_args:Ncc \cs_gnew_eq:NN }
```

## 96.12 Undefining functions

`\cs_gundefine:N` The following function is used to free the main memory from the definition of some function that isn’t in use any longer.

`\cs_gundefine:c`

```
1226 \cs_new_nopar:Npn \cs_gundefine:N #1{\cs_gset_eq:NN #1\c_undefined:D}
1227 \cs_new_nopar:Npn \cs_gundefine:c #1{
1228   \cs_gset_eq:cN {#1} \c_undefined:D
1229 }
```

### 96.13 Engine specific definitions

<code>\c_xetex_is_engine_bool</code>	In some cases it will be useful to know which engine we're running. Don't provide a <code>_p</code>
<code>\c_luatex_is_engine_bool</code>	predicate because the <code>_bool</code> is used for the same thing.

```

1230 \xetex_if_engine:TF
1231 \luatex_if_engine:TF
1232 \if_cs_exist:N \xetex_version:D
1233 \cs_new_eq:NN \c_xetex_is_engine_bool \c_true_bool
1234 \else:
1235 \cs_new_eq:NN \c_xetex_is_engine_bool \c_false_bool
1236 \fi:
1237 \prg_new_conditional:Npnn \xetex_if_engine: {TF,T,F} {
1238 \if_bool:N \c_xetex_is_engine_bool
1239 \prg_return_true: \else: \prg_return_false: \fi:
1240 }
1241
1242 \if_cs_exist:N \luatex_directlua:D
1243 \cs_new_eq:NN \c_luatex_is_engine_bool \c_true_bool
1244 \else:
1245 \cs_new_eq:NN \c_luatex_is_engine_bool \c_false_bool
1246 \fi:
1247 \prg_set_conditional:Npnn \xetex_if_engine: {TF,T,F}{
1248 \if_bool:N \c_xetex_is_engine_bool \prg_return_true:
1249 \else: \prg_return_false: \fi:
1250 }
1251 \prg_set_conditional:Npnn \luatex_if_engine: {TF,T,F}{
1252 \if_bool:N \c_luatex_is_engine_bool \prg_return_true:
1253 \else: \prg_return_false: \fi:
1254 }

```

## 96.14 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.

```
1252 \cs_new_nopar:Npn \prg_do_nothing: {}
```

### 96.15 Defining functions from a given number of arguments

et\_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.

```
1253 \cs_set:Npn \cs_get_arg_count_from_signature:N #1{
1254   \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN
1255 }
```

```

1256 \cs_set:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3{
1257   \if_predicate:w #3 % \bool_if:NTF here
1258     \exp_after:wN \use_i:nn
1259   \else:
1260     \exp_after:wN\use_ii:nn
1261   \fi:
1262   {
1263     \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
1264     \use_none:nnnnnnnnn #2 9876543210\q_nil
1265   }
1266   {-1}
1267 }
1268 \cs_set:Npn \cs_get_arg_count_from_signature_auxii:w #1#2\q_nil{#1}

```

A variant form we need right away.

```

1269 \cs_set_nopar:Npn \cs_get_arg_count_from_signature:c {
1270   \exp_args:Nc \cs_get_arg_count_from_signature:N
1271 }

```

`_generate_from_arg_count:NNnn`  
`e_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

```

1272 \cs_set:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4{
1273   \tex_ifcase:D \etex_numexpr:D #3\tex_relax:D
1274     \use_i_after_orelse:nw{#2#1}
1275   \or:
1276     \use_i_after_orelse:nw{#2#1 ##1}
1277   \or:
1278     \use_i_after_orelse:nw{#2#1 ##1##2}
1279   \or:
1280     \use_i_after_orelse:nw{#2#1 ##1##2##3}
1281   \or:
1282     \use_i_after_orelse:nw{#2#1 ##1##2##3##4}
1283   \or:
1284     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5}
1285   \or:
1286     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6}
1287   \or:
1288     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7}
1289   \or:
1290     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8}
1291   \or:
1292     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8##9}
1293   \else:
1294     \use_i_after_fi:nw{
1295       \cs_generate_from_arg_count_error_msg:Nn#1{#3}
1296       \use_none:n % to remove replacement text

```

```

1297     }
1298     \fi:
1299     {#4}
1300 }

```

A variant form we need right away.

```

1301 \cs_set_nopar:Npn \cs_generate_from_arg_count:cNnn {
1302   \exp_args:Nc \cs_generate_from_arg_count:NNnn
1303 }

```

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.

```

1304 \cs_set:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2 {
1305   \msg_kernel_bug:x {
1306     You're trying to define the command '\token_to_str:N #1'~
1307     with~ \use:n{\tex_the:D\etex_numexpr:D #2\tex_relax:D} ~
1308     arguments~ but~ I~ only~ allow~ 0-9~arguments.~Perhaps~you~
1309     forgot~to~use~a~colon~in~the~function~name?~
1310     I~ can~ probably~ not~ help~ you~ here
1311   }
1312 }

```

## 96.16 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` 1313 `\cs_set:Npn \cs_tmp:w #1#2#3{`

`\cs_gset_protected:Nx` 1314 `\cs_set_protected:cpx {cs_#1:#2}##1##2{`

`\cs_gset_protected:Nx` 1315 `\exp_not:N \cs_generate_from_arg_count:NNnn ##1`

`\cs_gset_protected_nopar:Nn` 1316 `\exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:`

`\cs_gset_protected_nopar:Nx` 1317 `{\exp_not:N\cs_get_arg_count_from_signature:N ##1}{##2}`

1318 `}`

1319 `}`

Then we define the 32 variants beginning with N.

```

1320 \cs_tmp:w {set}{Nn}{Npn}

```

```

1321 \cs_tmp:w {set}{Nx}{Npx}
1322 \cs_tmp:w {set_nopar}{Nn}{Npn}
1323 \cs_tmp:w {set_nopar}{Nx}{Npx}
1324 \cs_tmp:w {set_protected}{Nn}{Npn}
1325 \cs_tmp:w {set_protected}{Nx}{Npx}
1326 \cs_tmp:w {set_protected_nopar}{Nn}{Npn}
1327 \cs_tmp:w {set_protected_nopar}{Nx}{Npx}
1328 \cs_tmp:w {gset}{Nn}{Npn}
1329 \cs_tmp:w {gset}{Nx}{Npx}
1330 \cs_tmp:w {gset_nopar}{Nn}{Npn}
1331 \cs_tmp:w {gset_nopar}{Nx}{Npx}
1332 \cs_tmp:w {gset_protected}{Nn}{Npn}
1333 \cs_tmp:w {gset_protected}{Nx}{Npx}
1334 \cs_tmp:w {gset_protected_nopar}{Nn}{Npn}
1335 \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
\cs_gnew:Nn
\cs_gnew:Nx
\cs_gnew_nopar:Nn
\cs_gnew_nopar:Nx
\cs_gnew_protected:Nn
\cs_gnew_protected:Nx
\cs_gnew_protected_nopar:Nn
\cs_gnew_protected_nopar:Nx
1336 \cs_tmp:w {new}{Nn}{Npn}
1337 \cs_tmp:w {new}{Nx}{Npx}
1338 \cs_tmp:w {new_nopar}{Nn}{Npn}
1339 \cs_tmp:w {new_nopar}{Nx}{Npx}
1340 \cs_tmp:w {new_protected}{Nn}{Npn}
1341 \cs_tmp:w {new_protected}{Nx}{Npx}
1342 \cs_tmp:w {new_protected_nopar}{Nn}{Npn}
1343 \cs_tmp:w {new_protected_nopar}{Nx}{Npx}
1344 \cs_tmp:w {gnew}{Nn}{Npn}
1345 \cs_tmp:w {gnew}{Nx}{Npx}
1346 \cs_tmp:w {gnew_nopar}{Nn}{Npn}
1347 \cs_tmp:w {gnew_nopar}{Nx}{Npx}
1348 \cs_tmp:w {gnew_protected}{Nn}{Npn}
1349 \cs_tmp:w {gnew_protected}{Nx}{Npx}
1350 \cs_tmp:w {gnew_protected_nopar}{Nn}{Npn}
1351 \cs_tmp:w {gnew_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}
}

```

```

1352 \cs_set:Npn \cs_tmp:w #1#2#3{
1353   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1354     \exp_not:N\cs_generate_from_arg_count:cNnn {##1}
1355     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1356     {\exp_not:N\cs_get_arg_count_from_signature:c {##1}}{##2}
1357   }
1358 }

```

```

\cs_set:cn The 32 c variants.
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx

```

```

1359 \cs_tmp:w {set}{cn}{Npn}
1360 \cs_tmp:w {set}{cx}{Npx}
1361 \cs_tmp:w {set_nopar}{cn}{Npn}
1362 \cs_tmp:w {set_nopar}{cx}{Npx}
1363 \cs_tmp:w {set_protected}{cn}{Npn}
1364 \cs_tmp:w {set_protected}{cx}{Npx}
1365 \cs_tmp:w {set_protected_nopar}{cn}{Npn}
1366 \cs_tmp:w {set_protected_nopar}{cx}{Npx}
1367 \cs_tmp:w {gset}{cn}{Npn}
1368 \cs_tmp:w {gset}{cx}{Npx}
1369 \cs_tmp:w {gset_nopar}{cn}{Npn}
1370 \cs_tmp:w {gset_nopar}{cx}{Npx}
1371 \cs_tmp:w {gset_protected}{cn}{Npn}
1372 \cs_tmp:w {gset_protected}{cx}{Npx}
1373 \cs_tmp:w {gset_protected_nopar}{cn}{Npn}
1374 \cs_tmp:w {gset_protected_nopar}{cx}{Npx}

```

\cs\_new:cn

\cs\_new:cx

\cs\_new\_nopar:cn

\cs\_new\_nopar:cx

\cs\_new\_protected:cn

\cs\_new\_protected:cx

\cs\_new\_protected\_nopar:cn

\cs\_new\_protected\_nopar:cx

\cs\_gnew:cn

\cs\_gnew:cx

\cs\_gnew\_nopar:cn

\cs\_gnew\_nopar:cx

\cs\_gnew\_protected:cn

\cs\_gnew\_protected:cx

\cs\_gnew\_protected\_nopar:cn

\cs\_gnew\_protected\_nopar:cx

```

1375 \cs_tmp:w {new}{cn}{Npn}
1376 \cs_tmp:w {new}{cx}{Npx}
1377 \cs_tmp:w {new_nopar}{cn}{Npn}
1378 \cs_tmp:w {new_nopar}{cx}{Npx}
1379 \cs_tmp:w {new_protected}{cn}{Npn}
1380 \cs_tmp:w {new_protected}{cx}{Npx}
1381 \cs_tmp:w {new_protected_nopar}{cn}{Npn}
1382 \cs_tmp:w {new_protected_nopar}{cx}{Npx}
1383 \cs_tmp:w {gnew}{cn}{Npn}
1384 \cs_tmp:w {gnew}{cx}{Npx}
1385 \cs_tmp:w {gnew_nopar}{cn}{Npn}
1386 \cs_tmp:w {gnew_nopar}{cx}{Npx}
1387 \cs_tmp:w {gnew_protected}{cn}{Npn}
1388 \cs_tmp:w {gnew_protected}{cx}{Npx}
1389 \cs_tmp:w {gnew_protected_nopar}{cn}{Npn}
1390 \cs_tmp:w {gnew_protected_nopar}{cx}{Npx}

```

\cs\_if\_eq\_p:NN

\cs\_if\_eq\_p:cN

\cs\_if\_eq\_p:Nc

\cs\_if\_eq\_p:cc

\cs\_if\_eq:NN $\overline{TF}$

\cs\_if\_eq:cN $\overline{TF}$

\cs\_if\_eq:Nc $\overline{TF}$

\cs\_if\_eq:cc $\overline{TF}$

Check if two control sequences are identical.

```

1391 \prg_set_conditional:Npnn \cs_if_eq:NN #1#2{p,TF,T,F}{
1392   \if_meaning:w #1#2
1393   \prg_return_true: \else: \prg_return_false: \fi:
1394 }
1395 \cs_new_nopar:Npn \cs_if_eq_p:cN {\exp_args:Nc \cs_if_eq_p:NN}
1396 \cs_new_nopar:Npn \cs_if_eq:cN $\overline{TF}$  {\exp_args:Nc \cs_if_eq:NN $\overline{TF}$ }
1397 \cs_new_nopar:Npn \cs_if_eq:cNT {\exp_args:Nc \cs_if_eq:NNT}
1398 \cs_new_nopar:Npn \cs_if_eq:cNF {\exp_args:Nc \cs_if_eq:NNF}
1399 \cs_new_nopar:Npn \cs_if_eq_p:Nc {\exp_args:NNc \cs_if_eq_p:NN}
1400 \cs_new_nopar:Npn \cs_if_eq:Nc $\overline{TF}$  {\exp_args:NNc \cs_if_eq:NN $\overline{TF}$ }
1401 \cs_new_nopar:Npn \cs_if_eq:NcT {\exp_args:NNc \cs_if_eq:NNT}
1402 \cs_new_nopar:Npn \cs_if_eq:NcF {\exp_args:NNc \cs_if_eq:NNF}
1403 \cs_new_nopar:Npn \cs_if_eq_p:cc {\exp_args:Ncc \cs_if_eq_p:NN}
1404 \cs_new_nopar:Npn \cs_if_eq:cc $\overline{TF}$  {\exp_args:Ncc \cs_if_eq:NN $\overline{TF}$ }
1405 \cs_new_nopar:Npn \cs_if_eq:ccT {\exp_args:Ncc \cs_if_eq:NNT}
1406 \cs_new_nopar:Npn \cs_if_eq:ccF {\exp_args:Ncc \cs_if_eq:NNF}

```

Finally some code that is needed as we do not distribute the file module at the moment (so we simply define the needed function via an existing L<sup>A</sup>T<sub>E</sub>X command) and some other stuff which was set up elsewhere, in undistributed modules.

```

1407 \cs_new_nopar:Npn\file_not_found:nTF #1#2#3{\IfFileExists{#1}{#3}{#2}}

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
1408 \cs_set:Npn \prg_set_eq_conditional:NNn #1#2#3 {
1409   \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3}
1410 }
1411 \cs_set:Npn \prg_new_eq_conditional:NNn #1#2#3 {
1412   \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3}
1413 }

g_set_eq_conditional_aux:NNNn
g_set_eq_conditional_aux:NNNw
1414 \cs_set:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4 {
1415   \prg_set_eq_conditional_aux:NNNw #1#2#3#4,?,\q_recursion_stop
1416 }

```

Manual clist loop over argument #4.

```

1417 \cs_set:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4, {
1418   \if:w ? #4 \scan_stop:
1419     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1420   \fi:
1421   #1 {
1422     \exp_args:Nnc \cs_split_function:NN #2 {prg_conditional_form_#4:nnn}
1423   }{
1424     \exp_args:Nnc \cs_split_function:NN #3 {prg_conditional_form_#4:nnn}
1425   }
1426   \prg_set_eq_conditional_aux:NNNw #1{#2}{#3}
1427 }

1428 \cs_set:Npn \prg_conditional_form_p:nnn #1#2#3 {#1_p:#2}
1429 \cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 {#1:#2TF}
1430 \cs_set:Npn \prg_conditional_form_T:nnn #1#2#3 {#1:#2T}
1431 \cs_set:Npn \prg_conditional_form_F:nnn #1#2#3 {#1:#2F}

1432 </initex | package>

1433 <*showmemory>
1434 \showMemUsage
1435 </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>`.



**T<sub>E</sub>Xhackers 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`.

`\n::`  
`\N::`  
`\c::`  
`\o::`  
`\f::`  
`\x::`  
`\v::`  
`\V::`  
`\::`

`\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.

```

1436 <*package>
1437 \ProvidesExplPackage
1438   {\filename}{\filedate}{\fileversion}{\filedescription}
1439 \package_check_loaded_expl:
1440 </package>
1441 <*initex | package>

```

`\exp_after:wN` These are defined in `l3basics`.

```

\exp_not:N
\exp_not:n
1442 <*bootstrap>
1443 \cs_set_eq:NwN \exp_after:wN \tex_expandafter:D
1444 \cs_set_eq:NwN \exp_not:N \tex_noexpand:D
1445 \cs_set_eq:NwN \exp_not:n \etex_unexpanded:D
1446 </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>9</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`.

`\l_exp_tl` We need a scratch token list variable. We don't use `tl` methods so that `l3expan` can be loaded earlier.

```
1447 \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.

`\exp_arg_next_nobrace:nnn`

```
1448 \cs_new:Npn\exp_arg_next:nnn#1#2#3{
1449   #2\:::{#3{#1}}
1450 }
1451 \cs_new:Npn\exp_arg_next_nobrace:nnn#1#2#3{
1452   #2\:::{#3#1}
1453 }
```

`\:::` The end marker is just another name for the identity function.

```
1454 \cs_new:Npn\:::#1{#1}
```

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
1455 \cs_new:Npn\::n#1\:::#2#3{
1456   #1\:::{#2{#3}}
1457 }
```

---

<sup>9</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.

`\::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```

1458 \cs_new:Npn\::N#1\:::#2#3{
1459   #1\:::#2#3}
1460 }

```

`\::c` This function is used to skip an argument that is turned into as control sequence without expansion.

```

1461 \cs_new:Npn\::c#1\:::#2#3{
1462   \exp_after:wN\exp_arg_next_nobrace:nnn\cs:w #3\cs_end:{#1}{#2}
1463 }

```

`\::o` This function is used to expand an argument once.

```

1464 \cs_new:Npn\::o#1\:::#2#3{
1465   \exp_after:wN\exp_arg_next:nnn\exp_after:wN{#3}{#1}{#2}
1466 }

```

`\::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  $\text{\TeX}$  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  $\text{\TeX}$  has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

1467 \cs_new:Npn\::f#1\:::#2#3{
1468   \exp_after:wN\exp_arg_next:nnn
1469   \exp_after:wN{\tex_romannumeral:D -'0 #3}
1470   {#1}{#2}
1471 }
1472 \cs_new_nopar:Npn \exp_stop_f: {~}

```

`\::x` This function is used to expand an argument fully. If the pdf $\text{\TeX}$  primitive `\expanded`  
`\exp_arg:x` is present, we use it.

```

1473 \cs_new_eq:NN \exp_arg:x \expanded % Move eventually.
1474 \cs_if_free:NTF\exp_arg:x{
1475   \cs_new:Npn\::x#1\:::#2#3{
1476     \cs_set_nopar:Npx \l_exp_tl{#{#3}}
1477     \exp_after:wN\exp_arg_next:nnn\l_exp_tl{#1}{#2}}
1478 }
1479 {
1480   \cs_new:Npn\::x#1\:::#2#3{
1481     \exp_after:wN\exp_arg_next:nnn

```

```

1482     \exp_after:wN{\exp_arg:x{{#3}}}{#1}{#2}
1483   }
1484 }

```

`\::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.

```

1485 \cs_new:Npn \::V#1\::: #2#3{
1486   \exp_after:wN\exp_arg_next:nnn
1487   \exp_after:wN{
1488     \tex_romannumeral:D -'0
1489     \exp_eval_register:N #3
1490   }
1491   {#1}{#2}
1492 }
1493 \cs_new:Npn \::v#1\::: #2#3{
1494   \exp_after:wN\exp_arg_next:nnn
1495   \exp_after:wN{
1496     \tex_romannumeral:D -'0
1497     \exp_eval_register:c {#3}
1498   }
1499   {#1}{#2}
1500 }

```

`\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  $\text{\TeX}$  register such as `\count`. For the  $\text{\TeX}$  registers we have to utilize a `\tex_the:D` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\tex_the:D` and when not to. What we do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\tex_relax:D`.

```

1501 \cs_set_nopar:Npn \exp_eval_register:N #1{
1502   \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  $\text{\TeX}$  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.

```

1503   \if_meaning:w \tex_relax:D #1
1504   \exp_eval_error_msg:w
1505   \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`.

```

1506 \else:
1507   \exp_after:wN \use_i_ii:nnn
1508 \fi:
1509 \exp_after:wN \exp_stop_f: \tex_the:D #1
1510 }
1511 \cs_set_nopar:Npn \exp_eval_register:c #1{
1512   \exp_after:wN\exp_eval_register:N\cs:w #1\cs_end:
1513 }
```

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!

l.155 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```

1514 \group_begin:%
1515 \tex_catcode:D'\!=11\tex_relax:D%
1516 \tex_catcode:D'\ =11\tex_relax:D%
1517 \cs_gset:Npn\exp_eval_error_msg:w#1\tex_the:D#2{%
1518   \fi:\fi:\erroneous variable used!}%
1519 \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
1520 \cs_new:Npn \exp_args:No #1#2{\exp_after:wN#1\exp_after:wN{#2}}
1521 \cs_new:Npn \exp_args:NNo #1#2#3{\exp_after:wN#1\exp_after:wN#2
1522   \exp_after:wN{#3}}
1523 \cs_new:Npn \exp_args:NNNo #1#2#3#4{\exp_after:wN#1\exp_after:wN#2
1524   \exp_after:wN#3\exp_after:wN{#4}}

\exp_args:Nc
\exp_args:cc
\exp_args:NNc
\exp_args:Ncc
\exp_args:Nccc
1525 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}
```

Here are the functions that turn their argument into csnames but are expandable.

```

1526 \cs_new:Npn \exp_args:cc #1#2{\cs:w #1\exp_after:wN\cs_end:\cs:w #2\cs_end:}
1527 \cs_new:Npn \exp_args:NNc #1#2#3{\exp_after:wN#1\exp_after:wN#2
1528   \cs:w#3\cs_end:}
1529 \cs_new:Npn \exp_args:Ncc #1#2#3{\exp_after:wN#1
1530   \cs:w#2\exp_after:wN\cs_end:\cs:w#3\cs_end:}
1531 \cs_new:Npn \exp_args:Nccc #1#2#3#4{\exp_after:wN#1
1532   \cs:w#2\exp_after:wN\cs_end:\cs:w#3\exp_after:wN
1533   \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.

```

1534 \cs_new:Npn \exp_args:Nco #1#2#3{\exp_after:wN#1\cs:w#2\exp_after:wN
1535   \cs_end:\exp_after:wN{#3}}

```

## 97.5 Definitions with the ‘general’ technique

```

\exp_args:Nf
\exp_args:Nv
\exp_args:Nx

```

```

1536 \cs_set_nopar:Npn \exp_args:Nf {\:f\:::}
1537 \cs_set_nopar:Npn \exp_args:Nv {\:v\:::}
1538 \cs_set_nopar:Npn \exp_args:NV {\:V\:::}
1539 \cs_set_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
\exp_args:NNx
\exp_args:NNV
\exp_args:Ncx
\exp_args:Nfo
\exp_args:Nff
\exp_args:Ncf
\exp_args:Nco
\exp_args:Nnf
\exp_args:Nno
\exp_args:Nnx
\exp_args:Noo
\exp_args:Noc
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

```

```

1540 \cs_set_nopar:Npn \exp_args:NNf {\:N\:f\:::}
1541 \cs_set_nopar:Npn \exp_args:NNv {\:N\:v\:::}
1542 \cs_set_nopar:Npn \exp_args:NNV {\:N\:V\:::}
1543 \cs_set_nopar:Npn \exp_args:NNx {\:N\:x\:::}
1544
1545 \cs_set_nopar:Npn \exp_args:Ncx {\:c\:x\:::}
1546 \cs_set_nopar:Npn \exp_args:Nfo {\:f\:o\:::}
1547 \cs_set_nopar:Npn \exp_args:Nff {\:f\:f\:::}
1548 \cs_set_nopar:Npn \exp_args:Ncf {\:c\:f\:::}
1549 \cs_set_nopar:Npn \exp_args:Nnf {\:n\:f\:::}
1550 \cs_set_nopar:Npn \exp_args:Nno {\:n\:o\:::}
1551 \cs_set_nopar:Npn \exp_args:Nnx {\:n\:x\:::}
1552
1553 \cs_set_nopar:Npn \exp_args:Noc {\:o\:c\:::}
1554 \cs_set_nopar:Npn \exp_args:Noo {\:o\:o\:::}
1555 \cs_set_nopar:Npn \exp_args:Nox {\:o\:x\:::}
1556
1557 \cs_set_nopar:Npn \exp_args:NVV {\:V\:V\:::}
1558
1559 \cs_set_nopar:Npn \exp_args:Nxo {\:x\:o\:::}
1560 \cs_set_nopar:Npn \exp_args:Nxx {\:x\:x\:::}

```

```

\exp_args:Ncco
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Ncnc
\exp_args:Ncno
\exp_args:NNno
\exp_args:NNNV
\exp_args:Nnno
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nooo
\exp_args:Noox
\exp_args:Nnnc
\exp_args:NNnx

```

```

1561 \cs_set_nopar:Npn \exp_args:NNNV {\:N\:N\:V\:::}
1562

```

```

1563 \cs_set_nopar:Npn \exp_args:NNno {\::N\::n\::o\:::}
1564 \cs_set_nopar:Npn \exp_args:NNnx {\::N\::n\::x\:::}
1565 \cs_set_nopar:Npn \exp_args:NNoo {\::N\::o\::o\:::}
1566 \cs_set_nopar:Npn \exp_args:NNox {\::N\::o\::x\:::}
1567
1568 \cs_set_nopar:Npn \exp_args:Nnnc {\::n\::n\::c\:::}
1569 \cs_set_nopar:Npn \exp_args:Nnno {\::n\::n\::o\:::}
1570 \cs_set_nopar:Npn \exp_args:Nnnx {\::n\::n\::x\:::}
1571 \cs_set_nopar:Npn \exp_args:Nnox {\::n\::o\::x\:::}
1572
1573 \cs_set_nopar:Npn \exp_args:NcNc {\::c\::N\::c\:::}
1574 \cs_set_nopar:Npn \exp_args:NcNo {\::c\::N\::o\:::}
1575 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\:::}
1576 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\:::}
1577 \cs_set_nopar:Npn \exp_args:Nccx {\::c\::c\::x\:::}
1578 \cs_set_nopar:Npn \exp_args:Ncnx {\::c\::n\::x\:::}
1579
1580 \cs_set_nopar:Npn \exp_args:Noox {\::o\::o\::x\:::}
1581 \cs_set_nopar:Npn \exp_args:Nooo {\::o\::o\::o\:::}

```

## 97.6 Preventing expansion

```

\exp_not:o
\exp_not:d
\exp_not:f
\exp_not:v
\exp_not:V
1582 \cs_new:Npn\exp_not:o#1{\exp_not:n\exp_after:wN{#1}}
1583 \cs_new:Npn\exp_not:d#1{
1584   \exp_not:n\exp_after:wN\exp_after:wN\exp_after:wN{#1}
1585 }
1586 \cs_new:Npn\exp_not:f#1{
1587   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 #1}
1588 }
1589 \cs_new:Npn\exp_not:v#1{
1590   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:c {#1}}
1591 }
1592 \cs_new:Npn\exp_not:V#1{
1593   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:N #1}
1594 }

```

`\exp_not:c` A helper function.

```

1595 \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`.

```

1596 \cs_new:Npn \cs_generate_variant:Nn #1 {
1597   \chk_if_exist_cs:N #1
1598   \cs_split_function:NN #1 \cs_generate_variant_aux:nnNn
1599 }

```

We discard the boolean and then set off a loop through the desired variant forms.

```

1600 \cs_set:Npn \cs_generate_variant_aux:nnNn #1#2#3#4{
1601   \cs_generate_variant_aux:nnw {#1}{#2} #4,?,\q_recursion_stop
1602 }

```

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.

```

1603 \cs_set:Npn \cs_generate_variant_aux:nnw #1 #2 #3, {
1604   \if:w ? #3
1605   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1606   \fi:

```

Then check if the variant form has already been defined.

```

1607   \cs_if_free:cTF {
1608     #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1609   }
1610   {

```

If not, then define it and then additionally check if the `\exp_args:N` form needed is defined.

```

1611     \cs_new_nopar:cpx {
1612       #1:#3 \use:c{use_none:\cs_generate_variant_aux:N #3 ?}#2
1613     }
1614     {
1615       \exp_not:c { exp_args:N #3} \exp_not:c {#1:#2}
1616     }
1617     \cs_generate_internal_variant:n {#3}
1618   }

```

Otherwise tell that it was already defined.

```

1619   {
1620     \iow_log:x{
1621       Variant~\token_to_str:c {
1622         #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1623       }~already~defined;~ not~ changing~ it~on~line~
1624       \tex_the:D \tex_inputlineno:D
1625     }
1626   }

```



Recurse.

```
1627 \cs_generate_variant_aux:nnw{#1}{#2}
1628 }
```

The small loop for defining the required number of ns. Break when seeing a ?.

```
1629 \cs_set:Npn \cs_generate_variant_aux:N #1{
1630   \if:w ?#1 \exp_after:wN\use_none:nn \fi: n \cs_generate_variant_aux:N
1631 }
```

`\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`

```
1632 \cs_new:Npn \cs_generate_internal_variant:n #1 {
1633   \cs_if_free:cT { \exp_args:N #1 }{
```

We use `new` to log the definition if we have to make one.

```
1634     \cs_new:cpx { \exp_args:N #1 }
1635                 { \cs_generate_internal_variant_aux:n #1 : }
1636   }
1637 }
```

`\cs_generate_internal_variant_aux:n` This command grabs char by char outputting `\::#1` (not expanded further) until we see a `::`. That colon is in fact also turned into `\:::` so that the required structure for `\exp_args...` commands is correctly terminated.

```
1638 \cs_new:Npn \cs_generate_internal_variant_aux:n #1 {
1639   \exp_not:c{::#1}
1640   \if_meaning:w #1 :
1641     \exp_after:wN \use_none:n
1642   \fi:
1643   \cs_generate_internal_variant_aux:n
1644 }
```

## 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
```

```
1645 \cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
1646 \cs_new:Npn \::f_unbraced \:::#1#2 {
1647   \exp_after:wN \exp_arg_last_unbraced:nn
1648   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1649 }
1650 \cs_new:Npn \::o_unbraced \:::#1#2 {
1651   \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2 }{#1}
1652 }
1653 \cs_new:Npn \::V_unbraced \:::#1#2 {
1654   \exp_after:wN \exp_arg_last_unbraced:nn
1655   \exp_after:wN { \tex_romannumeral:D -'0 \exp_eval_register:N #2 } {#1}
1656 }
```

```

1657 \cs_new:Npn \::v_unbraced \:::#1#2 {
1658   \exp_after:wN \exp_arg_last_unbraced:nn
1659   \exp_after:wN {
1660     \tex_romannumeral:D -'0 \exp_eval_register:c {#2}
1661   } {#1}
1662 }

```

\exp\_last\_unbraced:NV Now the business end.

```

\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:NcV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNV
\exp_last_unbraced:NNNo
1663 \cs_new_nopar:Npn \exp_last_unbraced:Nf { \::f_unbraced \::: }
1664 \cs_new_nopar:Npn \exp_last_unbraced:NV { \::V_unbraced \::: }
1665 \cs_new_nopar:Npn \exp_last_unbraced:Nv { \::v_unbraced \::: }
1666 \cs_new_nopar:Npn \exp_last_unbraced:NcV {
1667   \::c \::V_unbraced \:::
1668 }
1669 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3 {
1670   \exp_after:wN #1 \exp_after:wN #2 #3
1671 }
1672 \cs_new_nopar:Npn \exp_last_unbraced:NNV {
1673   \::N \::V_unbraced \:::
1674 }
1675 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4 {
1676   \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4
1677 }

1678 </initex | package>

```

Show token usage:

```

1679 <*showmemory>
1680 \showMemUsage
1681 </showmemory>

```

## 98 l3prg implementation

### 98.1 Variables

\l_tmpa_bool	Reserved booleans.
\g_tmpa_bool	

\g_prg_inline_level_int	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.

```

1682 <*package>
1683 \ProvidesExplPackage

```

```

1684     {\filename}{\filedate}{\fileversion}{\filedescription}
1685 \package_check_loaded_expl:
1686 </package>
1687 <*initex | package>

```

### 98.3 Choosing modes

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```

1688 \prg_set_conditional:Npnn \mode_if_vertical: {p,TF,T,F}{
1689   \if_mode_vertical:
1690   \prg_return_true: \else: \prg_return_false: \fi:
1691 }

```

`\mode_if_horizontal_p:` For testing horizontal mode.

```

\mode_if_horizontal:TF
1692 \prg_set_conditional:Npnn \mode_if_horizontal: {p,TF,T,F}{
1693   \if_mode_horizontal:
1694   \prg_return_true: \else: \prg_return_false: \fi:
1695 }

```

`\mode_if_inner_p:` For testing inner mode.

```

\mode_if_inner:TF
1696 \prg_set_conditional:Npnn \mode_if_inner: {p,TF,T,F}{
1697   \if_mode_inner:
1698   \prg_return_true: \else: \prg_return_false: \fi:
1699 }

```

`\mode_if_math_p:` For testing math mode. Uses the kern-save `\scan_align_safe_stop:`.

```

\mode_if_math:TF
1700 \prg_set_conditional:Npnn \mode_if_math: {p,TF,T,F}{
1701   \scan_align_safe_stop: \if_mode_math:
1702   \prg_return_true: \else: \prg_return_false: \fi:
1703 }

```

### 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*...

```

1704 \cs_new_nopar:Npn \group_align_safe_begin: {
1705   \if_false:{\fi:\if_num:w'=\c_zero\fi:}
1706 \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>10</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.

```

1707 \cs_new_nopar:Npn \scan_align_safe_stop: {
1708   \intexpr_compare:nNnT \etex_currentgrouptype:D = \c_six
1709   {
1710     \intexpr_compare:nNnF \etex_lastnodetype:D = \c_zero
1711     {
1712       \intexpr_compare:nNnF \etex_lastnodetype:D = \c_seven
1713       \scan_stop:
1714     }
1715   }
1716 }

```

## 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 T<sub>E</sub>X 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

<sup>10</sup>Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

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.

```

1717 \cs_new_nopar:Npn \prg_replicate:nn #1{
1718   \cs:w prg_do_nothing:
1719   \exp_after:wN\prg_replicate_first_aux:N
1720   \tex_romannumeral:D -'\q \intexpr_eval:n{#1} \cs_end:
1721   \cs_end:
1722 }
1723 \cs_new_nopar:Npn \prg_replicate_aux:N#1{
1724   \cs:w prg_replicate_#1:n\prg_replicate_aux:N
1725 }
1726 \cs_new_nopar:Npn \prg_replicate_first_aux:N#1{
1727   \cs:w prg_replicate_first_#1:n\prg_replicate_aux:N
1728 }

```

Then comes all the functions that do the hard work of inserting all the copies.

```

1729 \cs_new_nopar:Npn \prg_replicate_ :n #1{}% no, this is not a typo!
1730 \cs_new:cpn {prg_replicate_0:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}}
1731 \cs_new:cpn {prg_replicate_1:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}#1}
1732 \cs_new:cpn {prg_replicate_2:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}#1#1}
1733 \cs_new:cpn {prg_replicate_3:n}#1{
1734   \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1}
1735 \cs_new:cpn {prg_replicate_4:n}#1{
1736   \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1}
1737 \cs_new:cpn {prg_replicate_5:n}#1{
1738   \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1}
1739 \cs_new:cpn {prg_replicate_6:n}#1{
1740   \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1}
1741 \cs_new:cpn {prg_replicate_7:n}#1{
1742   \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1}
1743 \cs_new:cpn {prg_replicate_8:n}#1{
1744   \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1}
1745 \cs_new:cpn {prg_replicate_9:n}#1{
1746   \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...

```

1747 \cs_new:cpn {prg_replicate_first_0:n}#1{\cs_end: }
1748 \cs_new:cpn {prg_replicate_first_1:n}#1{\cs_end: #1}
1749 \cs_new:cpn {prg_replicate_first_2:n}#1{\cs_end: #1#1}
1750 \cs_new:cpn {prg_replicate_first_3:n}#1{\cs_end: #1#1#1}
1751 \cs_new:cpn {prg_replicate_first_4:n}#1{\cs_end: #1#1#1#1}
1752 \cs_new:cpn {prg_replicate_first_5:n}#1{\cs_end: #1#1#1#1#1}
1753 \cs_new:cpn {prg_replicate_first_6:n}#1{\cs_end: #1#1#1#1#1#1}
1754 \cs_new:cpn {prg_replicate_first_7:n}#1{\cs_end: #1#1#1#1#1#1#1}
1755 \cs_new:cpn {prg_replicate_first_8:n}#1{\cs_end: #1#1#1#1#1#1#1#1}
1756 \cs_new:cpn {prg_replicate_first_9:n}#1{\cs_end: #1#1#1#1#1#1#1#1#1}

```

```

\prg_stepwise_function:nnnN
g_stepwise_function_incr:nnnN
g_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.

```

1757 \cs_new:Npn \prg_stepwise_function:nnnN #1#2{
1758   \intexpr_compare:nNnTF{#2}<\c_zero
1759   {\exp_args:Nf\prg_stepwise_function_decr:nnnN }
1760   {\exp_args:Nf\prg_stepwise_function_incr:nnnN }
1761   {\intexpr_eval:n{#1}}{#2}
1762 }
1763 \cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4{
1764   \intexpr_compare:nNnF {#1}>{#3}
1765   {
1766     #4{#1}
1767     \exp_args:Nf \prg_stepwise_function_incr:nnnN
1768     {\intexpr_eval:n{#1 + #2}}
1769     {#2}{#3}{#4}
1770   }
1771 }
1772 \cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4{
1773   \intexpr_compare:nNnF {#1}<{#3}
1774   {
1775     #4{#1}
1776     \exp_args:Nf \prg_stepwise_function_decr:nnnN
1777     {\intexpr_eval:n{#1 + #2}}
1778     {#2}{#3}{#4}
1779   }
1780 }

```

\g\_prg\_inline\_level\_int  
 \prg\_stepwise\_inline:nnnn  
 prg\_stepwise\_inline\_decr:nnnn  
 prg\_stepwise\_inline\_incr:nnnn

This function uses the same approach as for instance `\clist_map_inline:Nn` to allow arbitrary nesting. First construct the special function and then call an auxiliary one which just carries the newly constructed csname. Must make assignments global when we maintain our own stack.

```

1781 \int_new:N\g_prg_inline_level_int
1782 \cs_new:Npn\prg_stepwise_inline:nnnn #1#2#3#4{
1783   \int_gincr:N \g_prg_inline_level_int
1784   \cs_gset_nopar:cpn{prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}##1{#4}
1785   \intexpr_compare:nNnTF {#2}<\c_zero
1786   {\exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
1787   {\exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
1788   {prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}
1789   {\intexpr_eval:n{#1}} {#2} {#3}
1790   \int_gdecr:N \g_prg_inline_level_int
1791 }
1792 \cs_new:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4{
1793   \intexpr_compare:nNnF {#2}>{#4}
1794   {
1795     #1{#2}
1796     \exp_args:Nnf \prg_stepwise_inline_incr:Nnnn #1
1797     {\intexpr_eval:n{#2 + #3}} {#3}{#4}
1798   }
1799 }
1800 \cs_new:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4{
1801   \intexpr_compare:nNnF {#2}<{#4}

```

```

1802 {
1803   #1{#2}
1804   \exp_args:Nf \prg_stepwise_inline_decr:Nnnn #1
1805   {\intexpr_eval:n{#2 + #3}} {#3}{#4}
1806 }
1807 }

```

`\prg_stepwise_variable:nnnNn` Almost the same as above. Just store the value in #4 and execute #5.

```

\stepwise_variable_decr:nnnNn
\stepwise_variable_incr:nnnNn
1808 \cs_new:Npn \prg_stepwise_variable:nnnNn #1#2 {
1809   \intexpr_compare:nNnTF {#2}<\c_zero
1810   {\exp_args:Nf \prg_stepwise_variable_decr:nnnNn}
1811   {\exp_args:Nf \prg_stepwise_variable_incr:nnnNn}
1812   {\intexpr_eval:n{#1}}{#2}
1813 }
1814 \cs_new:Npn \prg_stepwise_variable_incr:nnnNn #1#2#3#4#5 {
1815   \intexpr_compare:nNnF {#1}>{#3}
1816   {
1817     \cs_set_nopar:Npn #4{#1} #5
1818     \exp_args:Nf \prg_stepwise_variable_incr:nnnNn
1819     {\intexpr_eval:n{#1 + #2}}{#2}{#3}#4{#5}
1820   }
1821 }
1822 \cs_new:Npn \prg_stepwise_variable_decr:nnnNn #1#2#3#4#5 {
1823   \intexpr_compare:nNnF {#1}<{#3}
1824   {
1825     \cs_set_nopar:Npn #4{#1} #5
1826     \exp_args:Nf \prg_stepwise_variable_decr:nnnNn
1827     {\intexpr_eval:n{#1 + #2}}{#2}{#3}#4{#5}
1828   }
1829 }

```

## 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 1830 \cs_new_nopar:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\bool_set_true:c 1831 \cs_new_nopar:Npn \bool_new:c #1 { \cs_new_eq:cN {#1} \c_false_bool }
\bool_set_false:N 1832 \cs_new_nopar:Npn \bool_set_true:N #1 { \cs_set_eq:NN #1 \c_true_bool }
\bool_set_false:c 1833 \cs_new_nopar:Npn \bool_set_true:c #1 { \cs_set_eq:cN {#1} \c_true_bool }
\bool_gset_true:N 1834 \cs_new_nopar:Npn \bool_set_false:N #1 { \cs_set_eq:NN #1 \c_false_bool }
\bool_gset_true:c 1835 \cs_new_nopar:Npn \bool_set_false:c #1 { \cs_set_eq:cN {#1} \c_false_bool }
\bool_gset_false:N 1836 \cs_new_nopar:Npn \bool_gset_true:N #1 { \cs_gset_eq:NN #1 \c_true_bool }
\bool_gset_false:c 1837 \cs_new_nopar:Npn \bool_gset_true:c #1 { \cs_gset_eq:cN {#1} \c_true_bool }
\bool_gset_false:c 1838 \cs_new_nopar:Npn \bool_gset_false:N #1 { \cs_gset_eq:NN #1 \c_false_bool }
1839 \cs_new_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 1840 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:cc 1841 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_gset_eq:NN 1842 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:Nc 1843 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\bool_gset_eq:cN 1844 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\bool_gset_eq:cc 1845 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
1846 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
1847 \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
1848 \bool_new:N \l_tmpa_bool
1849 \bool_new:N \g_tmpa_bool

\bool_if_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just
\bool_if_p:c be input directly.
\bool_if:NTF
\bool_if:cTF 1850 \prg_set_conditional:Npnn \bool_if:N #1 {p,TF,T,F}{
1851 \if_bool:N #1 \prg_return_true: \else: \prg_return_false: \fi:
1852 }
1853 \cs_generate_variant:Nn \bool_if_p:N {c}
1854 \cs_generate_variant:Nn \bool_if:NTF {c}
1855 \cs_generate_variant:Nn \bool_if:NT {c}
1856 \cs_generate_variant:Nn \bool_if:NF {c}

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The ‘while’
\bool_while_do:cn version executes the code as long as the boolean is true; the ‘until’ version executes the
\bool_until_do:Nn code as long as the boolean is false.
\bool_until_do:cn
1857 \cs_new:Npn \bool_while_do:Nn #1 #2 {
1858 \bool_if:NT #1 {#2 \bool_while_do:Nn #1 {#2}}
1859 }
1860 \cs_generate_variant:Nn \bool_while_do:Nn {c}

1861 \cs_new:Npn \bool_until_do:Nn #1 #2 {
1862 \bool_if:NF #1 {#2 \bool_until_do:Nn #1 {#2}}
1863 }
1864 \cs_generate_variant:Nn \bool_until_do:Nn {c}

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested
\bool_do_while:cn after executing the body. Otherwise identical to the above functions.
\bool_do_until:Nn
\bool_do_until:cn 1865 \cs_new:Npn \bool_do_while:Nn #1 #2 {
1866 #2 \bool_if:NT #1 {\bool_do_while:Nn #1 {#2}}
1867 }
1868 \cs_generate_variant:Nn \bool_do_while:Nn {c}

1869 \cs_new:Npn \bool_do_until:Nn #1 #2 {
1870 #2 \bool_if:NF #1 {\bool_do_until:Nn #1 {#2}}
1871 }
1872 \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_(:w</code>	
<code>\bool_p:w</code>	• If an Open is seen, start evaluating a new expression using the Eval function and
<code>\bool_8_1:w</code>	call GetNext again.
<code>\bool_I_1:w</code>	
<code>\bool_8_0:w</code>	• If a Not is seen, insert a negating function (if-even in this case) and call GetNext.
<code>\bool_I_0:w</code>	
<code>\bool_)_0:w</code>	• If none of the above, start evaluating a new expression by reinserting the token
<code>\bool_)_1:w</code>	found (this is supposed to be a predicate function) in front of Eval.
<code>\bool_S_0:w</code>	
<code>\bool_S_1:w</code>	

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

**$\langle true \rangle$ And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

**$\langle false \rangle$ And** Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return  $\langle false \rangle$ .

**$\langle true \rangle$ Or** Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return  $\langle true \rangle$ .

**$\langle false \rangle$ Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

**$\langle true \rangle$ Close** Current truth value is true, Close seen, return  $\langle true \rangle$ .

**$\langle false \rangle$ Close** Current truth value is false, Close seen, return  $\langle false \rangle$ .

We introduce an additional Stop operation with the following semantics:

**$\langle true \rangle$ Stop** Current truth value is true, return  $\langle true \rangle$ .

**$\langle false \rangle$ Stop** Current truth value is false, return  $\langle false \rangle$ .

The reasons for this follow below.

Now for how these works in practice. The canonical true and false values have numerical values 1 and 0 respectively. We evaluate this using the primitive `\tex_number:D` operation. First we issue a `\group_align_safe_begin:` as we are using && as syntax shorthand for the And operation and we need to hide it for T<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.

```

1873 \cs_set:Npn \bool_if_p:n #1{
1874   \group_align_safe_begin:
1875   \bool_get_next:N ( #1 )S
1876 }

```

The GetNext operation. We make it a switch: If not a ! or (, we assume it is a predicate.

```

1877 \cs_set:Npn \bool_get_next:N #1{
1878   \use:c {
1879     bool_
1880     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
1881     :w
1882   } #1
1883 }

```

The Not operation. Discard the token read and reverse the truth value of the next expression using \intexpr\_if\_even\_p:n.

```

1884 \cs_set:cpn {bool_!:w}#1{
1885   \exp_after:wN \intexpr_if_even_p:n \tex_number:D \bool_get_next:N
1886 }

```

The Open operation. Discard the token read and start a sub-expression.

```

1887 \cs_set:cpn {bool_(w}#1{
1888   \exp_after:wN \bool_cleanup:N \tex_number:D \bool_get_next:N
1889 }

```

Otherwise just evaluate the predicate and look for And, Or or Close afterward.

```

1890 \cs_set:cpn {bool_p:w}{\exp_after:wN \bool_cleanup:N \tex_number:D }

```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```

1891 \cs_new_nopar:Npn \bool_cleanup:N #1{
1892   \exp_after:wN \bool_choose:NN \exp_after:wN #1
1893   \int_to_roman:w-'\q
1894 }

```

Branching the six way switch.

```

1895 \cs_new_nopar:Npn \bool_choose:NN #1#2{ \use:c{bool_#2_#1:w} }

```

Continues scanning. Must remove the second & or |.

```

1896 \cs_new_nopar:cpn{bool_&_1:w}{\bool_get_next:N}
1897 \cs_new_nopar:cpn{bool_|_0: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.

```

1898 \cs_new_nopar:cpn{bool_)_0:w}{ \c_false_bool }
1899 \cs_new_nopar:cpn{bool_)_1:w}{ \c_true_bool }
1900 \cs_new_nopar:cpn{bool_S_0:w}{\group_align_safe_end: \c_false_bool }
1901 \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.

```
1902 \cs_set:cpn{bool_&_0:w}&{\bool_eval_skip_to_end:Nw \c_false_bool}
1903 \cs_set:cpn{bool_|_1: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.

```
1904 \cs_set:Npn \bool_eval_skip_to_end:Nw #1#2#{
1905   \bool_eval_skip_to_end_aux:Nw #1 #2(\q_no_value\q_nil{#2}
1906 }
```

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

```
1907 \cs_set:Npn \bool_eval_skip_to_end_aux:Nw #1#2(#3#4\q_nil#5{
1908   \quark_if_no_value:NTF #3
1909   { #1 }
1910   { \bool_eval_skip_to_end_auxii:Nw #1 #5 }
1911 }
```

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!

```
1912 \cs_set:Npn \bool_eval_skip_to_end_auxii:Nw #1#2(#3){
1913   \bool_eval_skip_to_end:Nw #1#3 )
1914 }
```

`\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 1915 \cs_new:Npn \bool_set:Nn #1#2 {\tex_chardef:D #1 = \bool_if_p:n {#2}}
\bool_gset:cn 1916 \cs_new:Npn \bool_gset:Nn #1#2 {
1917   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
1918 }
1919 \cs_generate_variant:Nn \bool_set:Nn {c}
1920 \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.

```
1921 \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.

```
1922 \cs_new:Npn \bool_xor_p:nn #1#2 {
1923   \intexpr_compare:nNnTF {\bool_if_p:n { #1 }} = {\bool_if_p:n { #2 }}
1924   {\c_false_bool}{\c_true_bool}
1925 }
```

```

1926 \prg_set_conditional:Npnn \bool_if:n #1 {TF,T,F}{
1927   \if_predicate:w \bool_if_p:n{#1}
1928   \prg_return_true: \else: \prg_return_false: \fi:
1929 }

```

```

\bool_while_do:nn #1 : Predicate test
\bool_until_do:nn #2 : Code to execute
\bool_do_while:nn
\bool_do_until:nn
1930 \cs_new:Npn \bool_while_do:nn #1#2 {
1931   \bool_if:nT {#1} { #2 \bool_while_do:nn {#1}{#2} }
1932 }
1933 \cs_new:Npn \bool_until_do:nn #1#2 {
1934   \bool_if:nF {#1} { #2 \bool_until_do:nn {#1}{#2} }
1935 }
1936 \cs_new:Npn \bool_do_while:nn #1#2 {
1937   #2 \bool_if:nT {#1} { \bool_do_while:nn {#1}{#2} }
1938 }
1939 \cs_new:Npn \bool_do_until:nn #1#2 {
1940   #2 \bool_if:nF {#1} { \bool_do_until:nn {#1}{#2} }
1941 }

```

## 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.

```

1942 \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.

```

1943   \exp_args:Nf \prg_case_int_aux:nnn { \intexpr_eval:n{#1}} #2

```

The ? below is just so there are enough arguments when we reach the end. And it made you look. ;-) )

```

1944   \q_recursion_tail ? \q_recursion_stop
1945 }
1946 \cs_new:Npn \prg_case_int_aux:nnn #1#2#3{

```

If we reach the end, return the else case. We just remove braces.

```

1947   \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}

```

Otherwise we compare (which evaluates #2 for us)

```

1948   \intexpr_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.

```

1949 { \prg_end_case:nw {#3} }
1950 { \prg_case_int_aux:nnn {#1}}
1951 }

```

`\prg_case_dim:nnn` Same as `\prg_case_dim:nnn` except it is for  $\langle dim \rangle$  registers.  
`\prg_case_dim_aux:nnn`

```

1952 \cs_new:Npn \prg_case_dim:nnn #1 #2 {
1953   \exp_args:No \prg_case_dim_aux:nnn {\dim_use:N \dim_eval:n{#1}} #2
1954   \q_recursion_tail ? \q_recursion_stop
1955 }
1956 \cs_new:Npn \prg_case_dim_aux:nnn #1#2#3{
1957   \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
1958   \dim_compare:nNnTF{#1}={#2}
1959   { \prg_end_case:nw {#3} }
1960   { \prg_case_dim_aux:nnn {#1}}
1961 }

```

`\prg_case_str:nnn` Same as `\prg_case_dim:nnn` except it is for strings.  
`\prg_case_str_aux:nnn`

```

1962 \cs_new:Npn \prg_case_str:nnn #1 #2 {
1963   \prg_case_str_aux:nnn {#1} #2
1964   \q_recursion_tail ? \q_recursion_stop
1965 }
1966 \cs_new:Npn \prg_case_str_aux:nnn #1#2#3{
1967   \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
1968   \tl_if_eq:xxTF{#1}{#2}
1969   { \prg_end_case:nw {#3} }
1970   { \prg_case_str_aux:nnn {#1}}
1971 }

```

`\prg_case_tl:Nnn` Same as `\prg_case_dim:nnn` except it is for token list variables.  
`\prg_case_tl_aux:NNn`

```

1972 \cs_new:Npn \prg_case_tl:Nnn #1 #2 {
1973   \prg_case_tl_aux:NNn #1 #2
1974   \q_recursion_tail ? \q_recursion_stop
1975 }
1976 \cs_new:Npn \prg_case_tl_aux:NNn #1#2#3{
1977   \quark_if_recursion_tail_stop_do:Nn #2{\use:n}
1978   \tl_if_eq:NNTF #1 #2
1979   { \prg_end_case:nw {#3} }
1980   { \prg_case_tl_aux:NNn #1}
1981 }

```

`\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.

```

1982 \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.

```
1983 \cs_new_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
1984   \cs_set:cpx{#1_quicksort:n}##1{
1985     \exp_not:c{#1_quicksort_start_partition:w} ##1
1986     \exp_not:n{#2\q_nil#3\q_stop}
1987   }
1988   \cs_set:cpx{#1_quicksort_braced:n}##1{
1989     \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
1990     \exp_not:N\q_nil\exp_not:N\q_stop
1991   }
1992   \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
1993     \exp_not:N \quark_if_nil:nTF {##1}\exp_not:N \use_none_delimit_by_q_stop:w
1994     \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{-}{-}
1995   }
1996   \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
1997     \exp_not:N \quark_if_nil:nTF {##1}\exp_not:N \use_none_delimit_by_q_stop:w
1998     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{-}{-}
1999   }
```

Now for doing the partitions.

```
2000 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2001   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2002   {
2003     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2004     \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2005     \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2006   }
2007   {##1}{##2}{##3}{##4}
2008 }
2009 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2010   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
```

```

2011 {
2012     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2013     \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2014     \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2015 }
2016 {##1}{##2}{##3}{##4}
2017 }
2018 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2019     \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2020     {
2021         \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2022         \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2023         \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2024     }
2025     {##1}{##2}{##3}{##4}
2026 }
2027 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2028     \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2029     {
2030         \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2031         \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2032         \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2033     }
2034     {##1}{##2}{##3}{##4}
2035 }

```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```

2036 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2037     \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}{##3}}
2038 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2039     \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3}{##4}}
2040 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2041     \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}{##2}{##3}}
2042 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2043     \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##4}{##3}}
2044 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2045     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##4}{##3}}
2046 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2047     \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3}{##4}}
2048 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2049     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}{##2}{##3}}
2050 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2051     \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.

```

2052 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2053     \exp_not:c{#1_quicksort_braced:n}{##2}
2054     \exp_not:c{#1_quicksort_function:n}{##1}
2055     \exp_not:c{#1_quicksort_braced:n}{##3}
2056 }
2057 }

```



`\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.

2058 `\prg_define_quicksort:nnn {prg}{\}{\}`

`\prg_quicksort_function:n`  
`\prg_quicksort_compare:nnTF`

2059 `\cs_set:Npn \prg_quicksort_function:n {\ERROR}`  
 2060 `\cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}`

That's it (for now).

2061 `</initex | package>`

2062 `<*showmemory>`

2063 `\showMemUsage`

2064 `</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.

2065 `<*package>`  
 2066 `\ProvidesExplPackage`  
 2067 `{\filename}{\filedate}{\fileversion}{\filedescription}`  
 2068 `\package_check_loaded_expl:`  
 2069 `</package>`  
 2070 `<*initex | package>`

`\quark_new:N` Allocate a new quark.

2071 `\cs_new_nopar:Npn \quark_new:N #1{\tl_new: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.

`\q_nil`

2072 `\quark_new:N \q_stop`  
 2073 `\quark_new:N \q_no_value`  
 2074 `\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`.

`\q_mark`

2075 `\quark_new:N \q_error`  
 2076 `\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.

```
2077 \quark_new:N\q_recursion_tail
2078 \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 the recursion end marker is found, we wrap things up and finish.

```
2079 \cs_new:Npn \quark_if_recursion_tail_stop:n #1 {
2080   \exp_after:wN\if_meaning:w
2081     \quark_if_recursion_tail_aux:w #1?\q_nil\q_recursion_tail\q_recursion_tail
2082     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2083   \fi:
2084 }
2085 \cs_new:Npn \quark_if_recursion_tail_stop:N #1 {
2086   \if_meaning:w#1\q_recursion_tail
2087     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2088   \fi:
2089 }
2090 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n {o}
```

`\quark_if_recursion_tail_stop_do:nn`  
`\quark_if_recursion_tail_stop_do:Nn`  
`\quark_if_recursion_tail_stop_do:on`

```
2091 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2 {
2092   \exp_after:wN\if_meaning:w
2093     \quark_if_recursion_tail_aux:w #1?\q_nil\q_recursion_tail\q_recursion_tail
2094     \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2095   \else:
2096     \exp_after:wN\use_none:n
2097   \fi:
2098   {#2}
2099 }
2100 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2 {
2101   \if_meaning:w #1\q_recursion_tail
2102     \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2103   \else:
2104     \exp_after:wN\use_none:n
2105   \fi:
2106   {#2}
2107 }
2108 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn {on}
```

`\quark_if_recursion_tail_aux:w`

```
2109 \cs_new:Npn \quark_if_recursion_tail_aux:w #1#2 \q_nil \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` `\quark_no_value` as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like `aabc` instead of a single token.<sup>11</sup>  
`\quark_if_no_value:NTF`  
`\quark_if_no_value:nTF`

---

<sup>11</sup>It may still loop in special circumstances however!

```

2110 \prg_new_conditional:Nnn \quark_if_no_value:N {p,TF,T,F} {
2111   \if_meaning:w \q_no_value #1
2112   \prg_return_true: \else: \prg_return_false: \fi:
2113 }

```

We also provide an `n` type. If run under a sufficiently new pdf $\epsilon$ -TeX, it uses a built-in primitive for string comparisons, otherwise it uses the slower `\str_if_eq_var_p:nf` function. In the latter case it would be faster to use a temporary token list variable but it would render the function non-expandable. Using the pdf $\epsilon$ -TeX primitive is the preferred approach. Note that we have to add a manual space token in the first part of the comparison, otherwise it is gobbled by `\str_if_eq_var_p:nf`. The reason for using this function instead of `\str_if_eq_p:nn` is that a sequence like `\q_no_value` will test equal to `\q_no_value` using the latter test function and unfortunately this example turned up in one application.

```

2114 \cs_if_exist:cTF {pdf_strcmp:D}
2115 {
2116   \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
2117     \if_num:w \pdf_strcmp:D
2118     {\exp_not:N \q_no_value}
2119     {\exp_not:n{#1}} = \c_zero
2120     \prg_return_true: \else: \prg_return_false:
2121     \fi:
2122   }
2123 }
2124 {
2125   \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
2126     \exp_args:NNo
2127     \if_predicate:w \str_if_eq_var_p:nf
2128     {\token_to_str:N\q_no_value\iow_space:}
2129     {\tl_to_str:n{#1}}
2130     \prg_return_true: \else: \prg_return_false:
2131     \fi:
2132   }
2133 }

```

`\quark_if_nil_p:N` A function to check for the presence of `\q_nil`.

`\quark_if_nil:N $\underline{TF}$`

```

2134 \prg_new_conditional:Nnn \quark_if_nil:N {p,TF,T,F} {
2135   \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi:
2136 }

```

`\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 $\underline{TF}$`

```

2137 \cs_if_exist:cTF {pdf_strcmp:D} {
2138   \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
2139     \if_num:w \pdf_strcmp:D
2140     {\exp_not:N \q_nil}
2141     {\exp_not:n{#1}} = \c_zero
2142     \prg_return_true: \else: \prg_return_false:
2143     \fi:
2144   }
2145 }

```

```

2146 {
2147 \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
2148   \exp_args:NNo
2149   \if_predicate:w \str_if_eq_var_p:nf
2150     {\token_to_str:N\q_nil\iow_space:}
2151     {\tl_to_str:n{#1}}
2152   \prg_return_true: \else: \prg_return_false:
2153   \fi:
2154 }
2155 }
2156 \cs_generate_variant:Nn \quark_if_nil_p:n {V}
2157 \cs_generate_variant:Nn \quark_if_nil:nTF {V}
2158 \cs_generate_variant:Nn \quark_if_nil:nT {V}
2159 \cs_generate_variant:Nn \quark_if_nil:nF {V}
2160 \cs_generate_variant:Nn \quark_if_nil_p:n {o}
2161 \cs_generate_variant:Nn \quark_if_nil:nTF {o}
2162 \cs_generate_variant:Nn \quark_if_nil:nT {o}
2163 \cs_generate_variant:Nn \quark_if_nil:nF {o}

```

Show token usage:

```

2164
2165 <*showmemory>
2166 \showMemUsage
2167 </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.

```

2168 \*package>
2169 \ProvidesExplPackage
2170   {\filename}{\filedate}{\fileversion}{\filedescription}
2171 \package_check_loaded_expl:
2172 \</package>
2173 \*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
2174 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
2175 \cs_new_nopar:Npn \char_set_catcode:nn #1#2 {
2176   \char_set_catcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
2177 }
2178 \cs_new_nopar:Npn \char_value_catcode:w { \int_use:N \tex_catcode:D }
2179 \cs_new_nopar:Npn \char_value_catcode:n #1 {
2180   \char_value_catcode:w \intexpr_eval:w #1\intexpr_eval_end:
2181 }
2182 \cs_new_nopar:Npn \char_show_value_catcode:w {
2183   \tex_showthe:D \tex_catcode:D
2184 }
2185 \cs_new_nopar:Npn \char_show_value_catcode:n #1 {
2186   \char_show_value_catcode:w \intexpr_eval:w #1\intexpr_eval_end:
2187 }

\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
2188 \cs_new_nopar:Npn \char_make_escape:N #1 { \char_set_catcode:nn {'#1} {\c_zero}
2189 \cs_new_nopar:Npn \char_make_begin_group:N #1 { \char_set_catcode:nn {'#1} {\c_one}
2190 \cs_new_nopar:Npn \char_make_end_group:N #1 { \char_set_catcode:nn {'#1} {\c_two}
2191 \cs_new_nopar:Npn \char_make_math_shift:N #1 { \char_set_catcode:nn {'#1} {\c_three}
2192 \cs_new_nopar:Npn \char_make_alignment:N #1 { \char_set_catcode:nn {'#1} {\c_four}
2193 \cs_new_nopar:Npn \char_make_end_line:N #1 { \char_set_catcode:nn {'#1} {\c_five}
2194 \cs_new_nopar:Npn \char_make_parameter:N #1 { \char_set_catcode:nn {'#1} {\c_six}
2195 \cs_new_nopar:Npn \char_make_math_superscript:N #1 { \char_set_catcode:nn {'#1} {\c_seven}
2196 \cs_new_nopar:Npn \char_make_math_subscript:N #1 { \char_set_catcode:nn {'#1} {\c_eight}
2197 \cs_new_nopar:Npn \char_make_ignore:N #1 { \char_set_catcode:nn {'#1} {\c_nine}
2198 \cs_new_nopar:Npn \char_make_space:N #1 { \char_set_catcode:nn {'#1} {\c_ten}
2199 \cs_new_nopar:Npn \char_make_letter:N #1 { \char_set_catcode:nn {'#1} {\c_eleven}
2200 \cs_new_nopar:Npn \char_make_other:N #1 { \char_set_catcode:nn {'#1} {\c_twelve}
2201 \cs_new_nopar:Npn \char_make_active:N #1 { \char_set_catcode:nn {'#1} {\c_thirteen}
2202 \cs_new_nopar:Npn \char_make_comment:N #1 { \char_set_catcode:nn {'#1} {\c_fourteen}
2203 \cs_new_nopar:Npn \char_make_invalid:N #1 { \char_set_catcode:nn {'#1} {\c_fifteen}

\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
2204 \cs_new_nopar:Npn \char_make_escape:n #1 { \char_set_catcode:nn {#1} {\c_zero} }
2205 \cs_new_nopar:Npn \char_make_begin_group:n #1 { \char_set_catcode:nn {#1} {\c_one} }

```

```

2206 \cs_new_nopar:Npn \char_make_end_group:n      #1 { \char_set_catcode:nn {#1} {\c_two} }
2207 \cs_new_nopar:Npn \char_make_math_shift:n     #1 { \char_set_catcode:nn {#1} {\c_three} }
2208 \cs_new_nopar:Npn \char_make_alignment:n      #1 { \char_set_catcode:nn {#1} {\c_four} }
2209 \cs_new_nopar:Npn \char_make_end_line:n       #1 { \char_set_catcode:nn {#1} {\c_five} }
2210 \cs_new_nopar:Npn \char_make_parameter:n      #1 { \char_set_catcode:nn {#1} {\c_six} }
2211 \cs_new_nopar:Npn \char_make_math_superscript:n #1 { \char_set_catcode:nn {#1} {\c_seven} }
2212 \cs_new_nopar:Npn \char_make_math_subscript:n #1 { \char_set_catcode:nn {#1} {\c_eight} }
2213 \cs_new_nopar:Npn \char_make_ignore:n        #1 { \char_set_catcode:nn {#1} {\c_nine} }
2214 \cs_new_nopar:Npn \char_make_space:n         #1 { \char_set_catcode:nn {#1} {\c_ten} }
2215 \cs_new_nopar:Npn \char_make_letter:n        #1 { \char_set_catcode:nn {#1} {\c_eleven} }
2216 \cs_new_nopar:Npn \char_make_other:n        #1 { \char_set_catcode:nn {#1} {\c_twelve} }
2217 \cs_new_nopar:Npn \char_make_active:n        #1 { \char_set_catcode:nn {#1} {\c_thirteen} }
2218 \cs_new_nopar:Npn \char_make_comment:n       #1 { \char_set_catcode:nn {#1} {\c_fourteen} }
2219 \cs_new_nopar:Npn \char_make_invalid:n       #1 { \char_set_catcode:nn {#1} {\c_fifteen} }

```

\char\_set\_mathcode:w Math codes.

```

\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
2220 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
2221 \cs_new_nopar:Npn \char_set_mathcode:nn #1#2 {
2222   \char_set_mathcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
2223 }
2224 \cs_new_protected_nopar:Npn \char_gset_mathcode:w { \pref_global:D \tex_mathcode:D }
2225 \cs_new_nopar:Npn \char_gset_mathcode:nn #1#2 {
2226   \char_gset_mathcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
2227 }
2228 \cs_new_nopar:Npn \char_value_mathcode:w { \int_use:N \tex_mathcode:D }
2229 \cs_new_nopar:Npn \char_value_mathcode:n #1 {
2230   \char_value_mathcode:w \intexpr_eval:w #1\intexpr_eval_end:
2231 }
2232 \cs_new_nopar:Npn \char_show_value_mathcode:w { \tex_showthe:D \tex_mathcode:D }
2233 \cs_new_nopar:Npn \char_show_value_mathcode:n #1 {
2234   \char_show_value_mathcode:w \intexpr_eval:w #1\intexpr_eval_end:
2235 }

```

```

\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
2236 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
2237 \cs_new_nopar:Npn \char_set_lccode:nn #1#2{
2238   \char_set_lccode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
2239 }
2240 \cs_new_nopar:Npn \char_value_lccode:w {\int_use:N\tex_lccode:D}
2241 \cs_new_nopar:Npn \char_value_lccode:n #1{\char_value_lccode:w
2242   \intexpr_eval:w #1\intexpr_eval_end:}
2243 \cs_new_nopar:Npn \char_show_value_lccode:w {\tex_showthe:D\tex_lccode:D}
2244 \cs_new_nopar:Npn \char_show_value_lccode:n #1{
2245   \char_show_value_lccode:w \intexpr_eval:w #1\intexpr_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
2246 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
2247 \cs_new_nopar:Npn \char_set_uccode:nn #1#2{
2248   \char_set_uccode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
2249 }
2250 \cs_new_nopar:Npn \char_value_uccode:w {\int_use:N\tex_uccode:D}

```

```

2251 \cs_new_nopar:Npn \char_value_uccode:n #1{\char_value_uccode:w
2252 \intexpr_eval:w #1\intexpr_eval_end:}
2253 \cs_new_nopar:Npn \char_show_value_uccode:w {\tex_showthe:D\tex_uccode:D}
2254 \cs_new_nopar:Npn \char_show_value_uccode:n #1{
2255 \char_show_value_uccode:w \intexpr_eval:w #1\intexpr_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
2256 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D
2257 \cs_new_nopar:Npn \char_set_sfcode:nn #1#2 {
2258 \char_set_sfcode:w #1 = \intexpr_eval:w #2\intexpr_eval_end:
2259 }
2260 \cs_new_nopar:Npn \char_value_sfcode:w { \int_use:N \tex_sfcode:D }
2261 \cs_new_nopar:Npn \char_value_sfcode:n #1 {
2262 \char_value_sfcode:w \intexpr_eval:w #1\intexpr_eval_end:
2263 }
2264 \cs_new_nopar:Npn \char_show_value_sfcode:w { \tex_showthe:D \tex_sfcode:D }
2265 \cs_new_nopar:Npn \char_show_value_sfcode:n #1 {
2266 \char_show_value_sfcode:w \intexpr_eval:w #1\intexpr_eval_end:
2267 }

```

## 100.4 Generic tokens

`\token_new:Nn` Creates a new token. (Will: why can't this just be `\cs_new_eq:NN \token_new:Nn \cs_gnew_eq:NN`? Seriously, that doesn't work!)

```

2268 \cs_new_nopar:Npn \token_new:Nn #1#2 {\cs_gnew_eq:NN #1#2}

```

`\c_group_begin_token` `\c_group_end_token` `\c_math_shift_token` `\c_alignment_tab_token` `\c_parameter_token` `\c_math_superscript_token` `\c_math_subscript_token` `\c_space_token` `\c_letter_token` `\c_other_char_token` `\c_active_char_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```

2269 \cs_new_eq:NN \c_group_begin_token {
2270 \cs_new_eq:NN \c_group_end_token }
2271 \group_begin:
2272 \char_set_catcode:nn{'\*}{3}
2273 \token_new:Nn \c_math_shift_token {*}
2274 \char_set_catcode:nn{'\*}{4}
2275 \token_new:Nn \c_alignment_tab_token {#}
2276 \token_new:Nn \c_parameter_token {#}
2277 \token_new:Nn \c_math_superscript_token {^}
2278 \char_set_catcode:nn{'\*}{8}
2279 \token_new:Nn \c_math_subscript_token {*}
2280 \token_new:Nn \c_space_token {~}
2281 \token_new:Nn \c_letter_token {a}
2282 \token_new:Nn \c_other_char_token {1}
2283 \char_set_catcode:nn{'\*}{13}
2284 \cs_gset_nopar:Npn \c_active_char_token {\exp_not:N*}
2285 \group_end:

```

`\token_if_group_begin_p:N` `\token_if_group_begin:NTF` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```

2286 \prg_new_conditional:Nnn \token_if_group_begin:N {p,TF,T,F} {
2287   \if_catcode:w \exp_not:N #1\c_group_begin_token
2288   \prg_return_true: \else: \prg_return_false: \fi:
2289 }

```

`\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`

```

2290 \prg_new_conditional:Nnn \token_if_group_end:N {p,TF,T,F} {
2291   \if_catcode:w \exp_not:N #1\c_group_end_token
2292   \prg_return_true: \else: \prg_return_false: \fi:
2293 }

```

`\token_if_math_shift_p:N` Check if token is a math shift token. We use the constant `\c_math_shift_token` for this.

`\token_if_math_shift:NTF`

```

2294 \prg_new_conditional:Nnn \token_if_math_shift:N {p,TF,T,F} {
2295   \if_catcode:w \exp_not:N #1\c_math_shift_token
2296   \prg_return_true: \else: \prg_return_false: \fi:
2297 }

```

`\token_if_alignment_tab_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token` for this.

`\token_if_alignment_tab:NTF`

```

2298 \prg_new_conditional:Nnn \token_if_alignment_tab:N {p,TF,T,F} {
2299   \if_catcode:w \exp_not:N #1\c_alignment_tab_token
2300   \prg_return_true: \else: \prg_return_false: \fi:
2301 }

```

`\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.

```

2302 \prg_new_conditional:Nnn \token_if_parameter:N {p,TF,T,F} {
2303   \exp_after:wN\if_catcode:w \cs:w c_parameter_token\cs_end:\exp_not:N #1
2304   \prg_return_true: \else: \prg_return_false: \fi:
2305 }

```

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

`\token_if_math_superscript:NTF`

```

2306 \prg_new_conditional:Nnn \token_if_math_superscript:N {p,TF,T,F} {
2307   \if_catcode:w \exp_not:N #1\c_math_superscript_token
2308   \prg_return_true: \else: \prg_return_false: \fi:
2309 }

```

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

`\token_if_math_subscript:NTF`

```

2310 \prg_new_conditional:Nnn \token_if_math_subscript:N {p,TF,T,F} {
2311   \if_catcode:w \exp_not:N #1\c_math_subscript_token
2312   \prg_return_true: \else: \prg_return_false: \fi:
2313 }

```



`\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 $\underline{TF}$`

```
2314 \prg_new_conditional:Nnn \token_if_space:N {p,TF,T,F} {
2315   \if_catcode:w \exp_not:N #1\c_space_token
2316   \prg_return_true: \else: \prg_return_false: \fi:
2317 }
```

`\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 $\underline{TF}$`

```
2318 \prg_new_conditional:Nnn \token_if_letter:N {p,TF,T,F} {
2319   \if_catcode:w \exp_not:N #1\c_letter_token
2320   \prg_return_true: \else: \prg_return_false: \fi:
2321 }
```

`\token_if_other_char_p:N` Check if token is an other char token. We use the constant `\c_other_char_token` for this.  
`\token_if_other_char:N $\underline{TF}$`

```
2322 \prg_new_conditional:Nnn \token_if_other_char:N {p,TF,T,F} {
2323   \if_catcode:w \exp_not:N #1\c_other_char_token
2324   \prg_return_true: \else: \prg_return_false: \fi:
2325 }
```

`\token_if_active_char_p:N` Check if token is an active char token. We use the constant `\c_active_char_token` for this.  
`\token_if_active_char:N $\underline{TF}$`

```
2326 \prg_new_conditional:Nnn \token_if_active_char:N {p,TF,T,F} {
2327   \if_catcode:w \exp_not:N #1\c_active_char_token
2328   \prg_return_true: \else: \prg_return_false: \fi:
2329 }
```

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.  
`\token_if_eq_meaning:NN $\underline{TF}$`

```
2330 \prg_new_conditional:Nnn \token_if_eq_meaning:NN {p,TF,T,F} {
2331   \if_meaning:w #1 #2
2332   \prg_return_true: \else: \prg_return_false: \fi:
2333 }
```

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.  
`\token_if_eq_catcode:NN $\underline{TF}$`

```
2334 \prg_new_conditional:Nnn \token_if_eq_catcode:NN {p,TF,T,F} {
2335   \if_catcode:w \exp_not:N #1 \exp_not:N #2
2336   \prg_return_true: \else: \prg_return_false: \fi:
2337 }
```

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.  
`\token_if_eq_charcode:NN $\underline{TF}$`

```
2338 \prg_new_conditional:Nnn \token_if_eq_charcode:NN {p,TF,T,F} {
2339   \if_charcode:w \exp_not:N #1 \exp_not:N #2
2340   \prg_return_true: \else: \prg_return_false: \fi:
2341 }
```

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like  
`\token_if_macro:NTF` `\long macro:#1->#1` so we simply check to see if the meaning contains `->`. Argument  
`\token_if_macro_p_aux:w` #2 in the code below will be empty if the string `->` isn't present, proof that the token was  
not a macro (which is why we reverse the emptiness test). However this function will fail  
on its own auxiliary function (and a few other private functions as well) but that should  
certainly never be a problem!

```

2342 \prg_new_conditional:Nnn \token_if_macro:N {p,TF,T,F} {
2343   \exp_after:wN \token_if_macro_p_aux:w \token_to_meaning:N #1 -> \q_nil
2344 }
2345 \cs_new_nopar:Npn \token_if_macro_p_aux:w #1 -> #2 \q_nil{
2346   \if_predicate:w \tl_if_empty_p:n{#2}
2347     \prg_return_false: \else: \prg_return_true: \fi:
2348 }

```

`\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`

```

2349 \prg_new_conditional:Nnn \token_if_cs:N {p,TF,T,F} {
2350   \if_predicate:w \token_if_eq_catcode_p:NN \scan_stop: #1
2351     \prg_return_true: \else: \prg_return_false: \fi:

```

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that  $\text{\TeX}$  will temporarily convert  
`\token_if_expandable:NTF` `\exp_not:N`  $\langle token \rangle$  into `\scan_stop:` if  $\langle token \rangle$  is expandable.

```

2352 \prg_new_conditional:Nnn \token_if_expandable:N {p,TF,T,F} {
2353   \cs_if_exist:NTF #1 {
2354     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2355     \prg_return_false: \else: \prg_return_true: \fi:
2356   } {
2357     \prg_return_false:
2358   }
2359 }

```

`\token_if_chardef_p:N` Most of these functions have to check the meaning of the token in question so we need to  
`\token_if_mathchardef_p:N` do some checkups on which characters are output by `\token_to_meaning:N`. As usual,  
`\token_if_int_register_p:N` these characters have catcode 12 so we must do some serious substitutions in the code  
`\token_if_skip_register_p:N` below...

```

2360 \group_begin:
2361   \char_set_lccode:nn {'\T}{'\T}
2362   \char_set_lccode:nn {'\F}{'\F}
2363   \char_set_lccode:nn {'\X}{'\n}
2364   \char_set_lccode:nn {'\Y}{'\t}
2365   \char_set_lccode:nn {'\Z}{'\d}
2366   \char_set_lccode:nn {'\?}{'\}
2367   \tl_map_inline:nn{\X\Y\Z\M\C\H\A\R\O\U\S\K\I\P\L\G\P\E}
2368     {\char_set_catcode:nn {'#1}{12}}

```

`\token_if_protected_macro_p:N` We convert the token list to lowercase and restore the catcode and lowercase code changes.  
`\token_if_protected_long_macro_p:N`

```

2369 \tl_to_lowercase:n{
2370   \group_end:

```

`\token_if_dim_register_p:N`  
`\token_if_skip_register_p:N`  
`\token_if_int_register_p:N`  
`\token_if_toks_register_p:N`  
`\token_if_chardef_p_aux:w`  
`\token_if_mathchardef_p_aux:w`  
`\token_if_int_register_p_aux:w`  
`\token_if_skip_register_p_aux:w`  
`\token_if_dim_register_p_aux:w`  
`\token_if_toks_register_p_aux:w`  
`\token_if_protected_macro_p_aux:w`  
`\token_if_long_macro_p_aux:w`  
`\token_if_protected_long_macro_p_aux:w`

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since T<sub>E</sub>X thinks of such tokens as hexadecimal so it stores them as `\char"<hex number>` or `\mathchar"<hex number>`.

```

2371 \prg_new_conditional:Nnn \token_if_chardef:N {p,TF,T,F} {
2372   \exp_after:wN \token_if_chardef_aux:w
2373   \token_to_meaning:N #1?CHAR"\q_nil
2374 }
2375 \cs_new_nopar:Npn \token_if_chardef_aux:w #1?CHAR"#2\q_nil{
2376   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2377 }

2378 \prg_new_conditional:Nnn \token_if_mathchardef:N {p,TF,T,F} {
2379   \exp_after:wN \token_if_mathchardef_aux:w
2380   \token_to_meaning:N #1?MAYHCHAR"\q_nil
2381 }
2382 \cs_new_nopar:Npn \token_if_mathchardef_aux:w #1?MAYHCHAR"#2\q_nil{
2383   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2384 }

```

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.

```

2385 \prg_new_conditional:Nnn \token_if_int_register:N {p,TF,T,F} {
2386   \if_meaning:w \tex_countdef:D #1
2387   \prg_return_false:
2388   \else:
2389     \exp_after:wN \token_if_int_register_aux:w
2390     \token_to_meaning:N #1?COUXY\q_nil
2391   \fi:
2392 }
2393 \cs_new_nopar:Npn \token_if_int_register_aux:w #1?COUXY#2\q_nil{
2394   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2395 }

```

Skip registers are done the same way as the integer registers.

```

2396 \prg_new_conditional:Nnn \token_if_skip_register:N {p,TF,T,F} {
2397   \if_meaning:w \tex_skipdef:D #1
2398   \prg_return_false:
2399   \else:
2400     \exp_after:wN \token_if_skip_register_aux:w
2401     \token_to_meaning:N #1?SKIP\q_nil
2402   \fi:
2403 }
2404 \cs_new_nopar:Npn \token_if_skip_register_aux:w #1?SKIP#2\q_nil{
2405   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2406 }

```

Dim registers. No news here

```

2407 \prg_new_conditional:Nnn \token_if_dim_register:N {p,TF,T,F} {
2408   \if_meaning:w \tex_dimendef:D #1
2409   \c_false_bool
2410   \else:

```

```

2411 \exp_after:wN \token_if_dim_register_aux:w
2412 \token_to_meaning:N #1?ZIMEX\q_nil
2413 \fi:
2414 }
2415 \cs_new_nopar:Npn \token_if_dim_register_aux:w #1?ZIMEX#2\q_nil{
2416 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2417 }

```

Toks registers.

```

2418 \prg_new_conditional:Nnn \token_if_toks_register:N {p,TF,T,F} {
2419 \if_meaning:w \tex_toksdef:D #1
2420 \prg_return_false:
2421 \else:
2422 \exp_after:wN \token_if_toks_register_aux:w
2423 \token_to_meaning:N #1?YOKS\q_nil
2424 \fi:
2425 }
2426 \cs_new_nopar:Npn \token_if_toks_register_aux:w #1?YOKS#2\q_nil{
2427 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2428 }

```

Protected macros.

```

2429 \prg_new_conditional:Nnn \token_if_protected_macro:N {p,TF,T,F} {
2430 \exp_after:wN \token_if_protected_macro_aux:w
2431 \token_to_meaning:N #1?PROYECY EZ~MACRO\q_nil
2432 }
2433 \cs_new_nopar:Npn \token_if_protected_macro_aux:w #1?PROYECY EZ~MACRO#2\q_nil{
2434 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2435 }

```

Long macros.

```

2436 \prg_new_conditional:Nnn \token_if_long_macro:N {p,TF,T,F} {
2437 \exp_after:wN \token_if_long_macro_aux:w
2438 \token_to_meaning:N #1?LOXG~MACRO\q_nil
2439 }
2440 \cs_new_nopar:Npn \token_if_long_macro_aux:w #1?LOXG~MACRO#2\q_nil{
2441 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2442 }

```

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.

```

2443 \prg_new_conditional:Nnn \token_if_protected_long_macro:N {p,TF,T,F} {
2444 \exp_after:wN \token_if_protected_long_macro_aux:w
2445 \token_to_meaning:N #1?PROYECY EZ?LOXG~MACRO\q_nil
2446 }
2447 \cs_new_nopar:Npn \token_if_protected_long_macro_aux:w #1
2448 ?PROYECY EZ?LOXG~MACRO#2\q_nil{
2449 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2450 }

```

Finally the \tl\_to\_lowercase:n ends!

```

2451 }

```

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.

`\prefix_arg_replacement_aux:w` 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.

```

2452 \group_begin:
2453 \char_set_lccode:nn {'\?}{'\:}
2454 \char_set_catcode:nn{'\M}{12}
2455 \char_set_catcode:nn{'\A}{12}
2456 \char_set_catcode:nn{'\C}{12}
2457 \char_set_catcode:nn{'\R}{12}
2458 \char_set_catcode:nn{'\O}{12}
2459 \tl_to_lowercase:n{
2460   \group_end:
2461   \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:w #1MACRO?#2->#3\q_nil#4{
2462     #4{#1}{#2}{#3}
2463   }
2464   \cs_new_nopar:Npn\token_get_prefix_spec:N #1{
2465     \token_if_macro:NTF #1{
2466       \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2467       \token_to_meaning:N #1\q_nil\use_i:nnn
2468     }{\scan_stop:}
2469   }
2470   \cs_new_nopar:Npn\token_get_arg_spec:N #1{
2471     \token_if_macro:NTF #1{
2472       \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2473       \token_to_meaning:N #1\q_nil\use_ii:nnn
2474     }{\scan_stop:}
2475   }
2476   \cs_new_nopar:Npn\token_get_replacement_spec:N #1{
2477     \token_if_macro:NTF #1{
2478       \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2479       \token_to_meaning:N #1\q_nil\use_iii:nnn
2480     }{\scan_stop:}
2481   }
2482 }

```

**Useless code: because we can!**

`\token_if_primitive_p:N` 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.

```

2483 \prg_new_conditional:Nnn \token_if_primitive:N {p,TF,T,F} {
2484   \if_predicate:w \token_if_cs_p:N #1
2485   \if_predicate:w \token_if_macro_p:N #1

```

```

2486     \prg_return_false:
2487   \else:
2488     \token_if_primitive_p_aux:N #1
2489   \fi:
2490 \else:
2491   \if_predicate:w \token_if_active_char_p:N #1
2492   \if_predicate:w \token_if_macro_p:N #1
2493     \prg_return_false:
2494   \else:
2495     \token_if_primitive_p_aux:N #1
2496   \fi:
2497 \else:
2498   \prg_return_false:
2499 \fi:
2500 \fi:
2501 }
2502 \cs_new_nopar:Npn \token_if_primitive_p_aux:N #1{
2503   \if_predicate:w \token_if_chardef_p:N #1 \c_false_bool
2504 \else:
2505   \if_predicate:w \token_if_mathchardef_p:N #1 \prg_return_false:
2506 \else:
2507   \if_predicate:w \token_if_int_register_p:N #1 \prg_return_false:
2508 \else:
2509   \if_predicate:w \token_if_skip_register_p:N #1 \prg_return_false:
2510 \else:
2511   \if_predicate:w \token_if_dim_register_p:N #1 \prg_return_false:
2512 \else:
2513   \if_predicate:w \token_if_toks_register_p:N #1 \prg_return_false:
2514 \else:

```

We made it!

```

2515     \prg_return_true:
2516   \fi:
2517 \fi:
2518 \fi:
2519 \fi:
2520 \fi:
2521 \fi:
2522 }

```

## 100.5 Peeking ahead at the next token

```

\l_peek_token We define some other tokens which will initially be the character ?.
\g_peek_token
\l_peek_search_token
2523 \token_new:Nn \l_peek_token {?}
2524 \token_new:Nn \g_peek_token {?}
2525 \token_new:Nn \l_peek_search_token {?}

```

`\peek_after: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`.

```

2526 \cs_new_nopar:Npn \peek_after:NN {\tex_futurelet:D \l_peek_token }
2527 \cs_new_nopar:Npn \peek_gafter:NN {
2528   \pref_global:D \tex_futurelet:D \g_peek_token
2529 }

```

For normal purposes there are four main cases:

1. peek at the next token.
2. peek at the next non-space token.
3. peek at the next token and remove it.
4. peek at the next non-space token and remove it.

The generic functions will take four arguments: The token to search for, the test function to run on it and the true/false cases. The general algorithm is this:

1. Store the token to search for in `\l_peek_search_token`.
2. In order to avoid doubling of hash marks where it seems unnatural we put the `\langle true \rangle` and `\langle false \rangle` cases through an `x` type expansion but using `\exp_not:n` to avoid any expansion. This has the same effect as putting it through a `\langle toks \rangle` register but is faster. Also put in a special alignment safe group end.
3. Put in an alignment safe group begin.
4. Peek ahead and call the function which will act on the next token in the input stream.

`\l_peek_true_tl` Two dedicated token list variables that store the true and false cases.

```

\l_peek_false_tl
2530 \tl_new:Nn \l_peek_true_tl {}
2531 \tl_new:Nn \l_peek_false_tl {}

```

`\peek_tmp:w` Scratch function used for storing the token to be removed if found.

```

2532 \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.

```

2533 \tl_new:Nn \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.

```

2534 \cs_new:Npn \peek_token_generic:NNTF #1#2#3#4 {
2535   \cs_set_eq:NN \l_peek_search_token #2
2536   \tl_set:Nn \l_peek_search_tl {#2}
2537   \tl_set:Nx \l_peek_true_tl {\exp_not:n{\group_align_safe_end: #3}}
2538   \tl_set:Nx \l_peek_false_tl {\exp_not:n{\group_align_safe_end: #4}}
2539   \group_align_safe_begin:

```

```

2540 \peek_after:NN #1
2541 }
2542 \cs_new:Npn \peek_token_generic:NNT #1#2#3 {
2543 \peek_token_generic:NNTF #1#2 {#3} {}
2544 }
2545 \cs_new:Npn \peek_token_generic:NNF #1#2#3 {
2546 \peek_token_generic:NNTF #1#2 {} {#3}
2547 }

```

`\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.

```

2548 \cs_new:Npn \peek_token_remove_generic:NNTF #1#2#3#4 {
2549 \cs_set_eq:NN \l_peek_search_token #2
2550 \tl_set:Nn \l_peek_search_tl {#2}
2551 \tl_set:Nx \l_peek_true_aux_tl { \exp_not:n{ #3 } }
2552 \tl_set_eq:NN \l_peek_true_tl \c_peek_true_remove_next_tl
2553 \tl_set:Nx \l_peek_false_tl { \exp_not:n{ \group_align_safe_end: #4 } }
2554 \group_align_safe_begin:
2555 \peek_after:NN #1
2556 }
2557 \cs_new:Npn \peek_token_remove_generic:NNT #1#2#3 {
2558 \peek_token_remove_generic:NNTF #1#2 {#3} {}
2559 }
2560 \cs_new:Npn \peek_token_remove_generic:NNF #1#2#3 {
2561 \peek_token_remove_generic:NNTF #1#2 {} {#3}
2562 }

```

`\l_peek_true_aux_tl` Two token list variables to help with removing the character from the input stream.  
`\c_peek_true_remove_next_tl`

```

2563 \tl_new:Nn \l_peek_true_aux_tl {}
2564 \tl_new:Nn \c_peek_true_remove_next_tl { \group_align_safe_end:
2565 \tex_afterassignment:D \l_peek_true_aux_tl \cs_set_eq:NN \peek_tmp:w
2566 }

```

`\peek_execute_branches_meaning:` There are three major tests between tokens in TeX: meaning, catcode and charcode.  
`\peek_execute_branches_catcode:` Hence we define three basic test functions that set in after the ignoring phase is over and  
`\peek_execute_branches_charcode:` done with.

`\peek_execute_branches_charcode_aux:NN`

```

2567 \cs_new_nopar:Npn \peek_execute_branches_meaning: {
2568 \if_meaning:w \l_peek_token \l_peek_search_token
2569 \exp_after:wN \l_peek_true_tl
2570 \else:
2571 \exp_after:wN \l_peek_false_tl
2572 \fi:
2573 }
2574 \cs_new_nopar:Npn \peek_execute_branches_catcode: {
2575 \if_catcode:w \exp_not:N \l_peek_token \exp_not:N \l_peek_search_token
2576 \exp_after:wN \l_peek_true_tl
2577 \else:
2578 \exp_after:wN \l_peek_false_tl
2579 \fi:
2580 }

```



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`.

```

2581 \cs_new_nopar:Npn \peek_execute_branches_charcode: {
2582   \bool_if:nTF {
2583     \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
2584     \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
2585   }
2586   { \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.

```

2587   { \exp_after:wN \peek_execute_branches_charcode_aux:NN \l_peek_search_tl }
2588 }

```

Then we just do the usual `\if_charcode:w` comparison. We also remember to insert `#2` again after executing the true or false branches.

```

2589 \cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2{
2590   \if_charcode:w \exp_not:N #1\exp_not:N#2
2591     \exp_after:wN \l_peek_true_tl
2592   \else:
2593     \exp_after:wN \l_peek_false_tl
2594   \fi:
2595   #2
2596 }

```

```

\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.

```

2597 \cs_new_nopar:Npn \peek_def_aux:nnnn #1#2#3#4 {
2598   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { TF }
2599   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { T }
2600   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { F }
2601 }
2602 \cs_new_nopar:Npn \peek_def_aux_ii:nnnnn #1#2#3#4#5 {
2603   \cs_new_nopar:cpx { #1 #5 } {
2604     \tl_if_empty:nF {#2} {
2605       \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 }
2606     }
2607     \exp_not:c { #3 #5 }
2608     \exp_not:n { #4 }
2609   }
2610 }

```

`\peek_meaning:NTF` Here we use meaning comparison with `\if_meaning:w`.

```
2611 \peek_def_aux:nnnn
2612 { peek_meaning:N }
2613 {}
2614 { peek_token_generic:NN }
2615 { \peek_execute_branches_meaning: }
```

`peek_meaning_ignore_spaces:NTF`

```
2616 \peek_def_aux:nnnn
2617 { peek_meaning_ignore_spaces:N }
2618 { \peek_execute_branches_meaning: }
2619 { peek_token_generic:NN }
2620 { \peek_ignore_spaces_execute_branches: }
```

`\peek_meaning_remove:NTF`

```
2621 \peek_def_aux:nnnn
2622 { peek_meaning_remove:N }
2623 {}
2624 { peek_token_remove_generic:NN }
2625 { \peek_execute_branches_meaning: }
```

`meaning_remove_ignore_spaces:NTF`

```
2626 \peek_def_aux:nnnn
2627 { peek_meaning_remove_ignore_spaces:N }
2628 { \peek_execute_branches_meaning: }
2629 { peek_token_remove_generic:NN }
2630 { \peek_ignore_spaces_execute_branches: }
```

`\peek_catcode:NTF` Here we use catcode comparison with `\if_catcode:w`.

```
2631 \peek_def_aux:nnnn
2632 { peek_catcode:N }
2633 {}
2634 { peek_token_generic:NN }
2635 { \peek_execute_branches_catcode: }
```

`peek_catcode_ignore_spaces:NTF`

```
2636 \peek_def_aux:nnnn
2637 { peek_catcode_ignore_spaces:N }
2638 { \peek_execute_branches_catcode: }
2639 { peek_token_generic:NN }
2640 { \peek_ignore_spaces_execute_branches: }
```

`\peek_catcode_remove:NTF`

```
2641 \peek_def_aux:nnnn
2642 { peek_catcode_remove:N }
2643 {}
2644 { peek_token_remove_generic:NN }
2645 { \peek_execute_branches_catcode: }
```

code\_remove\_ignore\_spaces:NTF

```
2646 \peek_def_aux:nnnn
2647 { peek_catcode_remove_ignore_spaces:N }
2648 { \peek_execute_branches_catcode: }
2649 { peek_token_remove_generic:NN }
2650 { \peek_ignore_spaces_execute_branches: }
```

\peek\_charcode:NTF Here we use charcode comparison with \if\_charcode:w.

```
2651 \peek_def_aux:nnnn
2652 { peek_charcode:N }
2653 {}
2654 { peek_token_generic:NN }
2655 { \peek_execute_branches_charcode: }
```

ek\_charcode\_ignore\_spaces:NTF

```
2656 \peek_def_aux:nnnn
2657 { peek_charcode_ignore_spaces:N }
2658 { \peek_execute_branches_charcode: }
2659 { peek_token_generic:NN }
2660 { \peek_ignore_spaces_execute_branches: }
```

\peek\_charcode\_remove:NTF

```
2661 \peek_def_aux:nnnn
2662 { peek_charcode_remove:N }
2663 {}
2664 { peek_token_remove_generic:NN }
2665 { \peek_execute_branches_charcode: }
```

code\_remove\_ignore\_spaces:NTF

```
2666 \peek_def_aux:nnnn
2667 { peek_charcode_remove_ignore_spaces:N }
2668 { \peek_execute_branches_charcode: }
2669 { peek_token_remove_generic:NN }
2670 { \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  
more\_spaces\_execute\_branches: 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.

```
2671 \cs_new_nopar:Npn \peek_ignore_spaces_aux: {
2672   \peek_after:NN \peek_ignore_spaces_execute_branches:
2673 }
2674 \cs_new_nopar:Npn \peek_ignore_spaces_execute_branches: {
2675   \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
```

```

2676 { \tex_afterassignment:D \peek_ignore_spaces_aux:
2677 \cs_set_eq:NN \peek_tmp:w
2678 }
2679 \peek_execute_branches:
2680 }

2681 </initex | package>

2682 <*showmemory>
2683 \showMemUsage
2684 </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 &lt;function&gt; {&lt;integer expression&gt;}</code> <code>\int_pre_eval_one_arg:Nnn &lt;function&gt; {&lt;int expr<sub>1</sub>&gt;}</code> <code>{&lt;int expr<sub>2</sub>&gt;}</code>
--	--

These are expansion helpers; they evaluate their integer expressions before handing them off to the specified *<function>*.

<code>\int_get_sign_and_digits:n *</code> <code>\int_get_sign:n *</code> <code>\int_get_digits:n *</code>	<code>\int_get_sign_and_digits:n {&lt;number&gt;}</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.

```

2685 <*package>
2686 \ProvidesExplPackage

```

```

2687   {\filename}{\filedate}{\fileversion}{\filedescription}
2688   \package_check_loaded_expl:
2689   </package>
2690   <*initex | package>

```

```

\int_to_roman:w A new name for the primitives.
\int_to_number:w
\int_advance:w
2691   \cs_new_eq:NN \int_to_roman:w \tex_romannumeral:D
2692   \cs_new_eq:NN \int_to_number:w \tex_number:D
2693   \cs_new_eq:NN \int_advance:w \tex_advance:D

```

Functions that support L<sup>A</sup>T<sub>E</sub>X's user accessible counters should be added here, too. But first the internal counters.

### 101.3 Allocation and setting

`\int_new:N` Allocation of a new internal counter is already done above. Here we define the next likely variant.

For the L<sup>A</sup>T<sub>E</sub>X3 format:

```

2694   <*initex>
2695   \alloc_setup_type:nnn {int} {11} \c_max_register_num
2696   \cs_new_nopar:Npn \int_new:N #1 {\alloc_reg:NnNN g {int} \tex_countdef:D#1}
2697   \cs_new_nopar:Npn \int_new_l:N #1 {\alloc_reg:NnNN l {int} \tex_countdef:D#1}
2698   </initex>

```

For 'l3in2e':

```

2699   <*package>
2700   \cs_new_nopar:Npn \int_new:N #1 {
2701     \chk_if_free_cs:N #1
2702     \newcount #1
2703   }
2704   </package>

2705   \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 get a better overview.

```

\int_set:cn
\int_gset:Nn
\int_gset:cn
2706   \cs_new_nopar:Npn \int_set:Nn #1#2{#1 \intexpr_eval:w #2\intexpr_eval_end:
2707   <*check>
2708   \chk_local_or_pref_global:N #1
2709   </check>
2710   }
2711   \cs_new_nopar:Npn \int_gset:Nn {
2712   <*check>
2713     \pref_global_chk:
2714   </check>
2715   <-check> \pref_global:D
2716     \int_set:Nn }
2717   \cs_generate_variant:Nn\int_set:Nn {cn}
2718   \cs_generate_variant:Nn\int_gset:Nn {cn}

```

\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
2719 \cs_new_nopar:Npn \int_incr:N #1{\int_advance:w#1\c_one
2720 <*check>
2721 \chk_local_or_pref_global:N #1
2722 </check>
2723 }
2724 \cs_new_nopar:Npn \int_decr:N #1{\int_advance:w#1\c_minus_one
2725 <*check>
2726 \chk_local_or_pref_global:N #1
2727 </check>
2728 }
2729 \cs_new_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.

```

2730 <*check>
2731 \pref_global_chk:
2732 </check>
2733 <-check> \pref_global:D
2734 \int_incr:N}
2735 \cs_new_nopar:Npn \int_gdecr:N {
2736 <*check>
2737 \pref_global_chk:
2738 </check>
2739 <-check> \pref_global:D
2740 \int_decr:N}

```

With the \int\_add:Nn functions we can shorten the above code. If this makes it too slow ...

```

2741 \cs_set_nopar:Npn \int_incr:N #1{\int_add:Nn#1\c_one}
2742 \cs_set_nopar:Npn \int_decr:N #1{\int_add:Nn#1\c_minus_one}
2743 \cs_set_nopar:Npn \int_gincr:N #1{\int_gadd:Nn#1\c_one}
2744 \cs_set_nopar:Npn \int_gdecr:N #1{\int_gadd:Nn#1\c_minus_one}

2745 \cs_generate_variant:Nn \int_incr:N {c}
2746 \cs_generate_variant:Nn \int_decr:N {c}
2747 \cs_generate_variant:Nn \int_gincr:N {c}
2748 \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
2749 \cs_new_nopar:Npn \int_zero:N #1 {#1=\c_zero}
2750 \cs_generate_variant:Nn \int_zero:N {c}

2751 \cs_new_nopar:Npn \int_gzero:N #1 {\pref_global:D #1=\c_zero}
2752 \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
2753 \cs_new_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...

```

2754 \int_advance:w #1 by \intexpr_eval:w #2\intexpr_eval_end:
2755 <*check>
2756 \chk_local_or_pref_global:N #1
2757 </check>
2758 }
2759 \cs_new_nopar:Npn \int_sub:Nn #1#2{
2760 \int_advance:w #1-\intexpr_eval:w #2\intexpr_eval_end:
2761 <*check>
2762 \chk_local_or_pref_global:N #1
2763 </check>
2764 }
2765 \cs_new_nopar:Npn \int_gadd:Nn {
2766 <*check>
2767 \pref_global_chk:
2768 </check>
2769 <-check> \pref_global:D
2770 \int_add:Nn }
2771 \cs_new_nopar:Npn \int_gsub:Nn {
2772 <*check>
2773 \pref_global_chk:
2774 </check>
2775 <-check> \pref_global:D
2776 \int_sub:Nn }
2777 \cs_generate_variant:Nn \int_add:Nn {cn}
2778 \cs_generate_variant:Nn \int_gadd:Nn {cn}
2779 \cs_generate_variant:Nn \int_sub:Nn {cn}
2780 \cs_generate_variant:Nn \int_gsub:Nn {cn}

```

`\int_use:N` Here is how counters are accessed:

```

\int_use:c
2781 \cs_new_eq:NN \int_use:N \tex_the:D
2782 \cs_new_nopar:Npn \int_use:c #1{\int_use:N \cs:w#1\cs_end:}

```

`\int_show:N` Diagnostics.

```

\int_show:c
2783 \cs_new_eq:NN \int_show:N \tex_showthe:D
2784 \cs_new_nopar:Npn \int_show:c {\exp_args:Nc \int_show:N }

```

`\int_to_arabic:n` Nothing exciting here.

```

2785 \cs_new_nopar:Npn \int_to_arabic:n #1{ \intexpr_eval:n{#1}}

```

`\int_roman_lcuc_mapping:Nnn` Using TeX'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.

```

2786 \cs_new_nopar:Npn \int_roman_lcuc_mapping:Nnn #1#2#3{
2787 \cs_set_nopar:cpn {int_to_lc_roman_#1:}{#2}
2788 \cs_set_nopar:cpn {int_to_uc_roman_#1:}{#3}
2789 }

```

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.

```
2790 \int_roman_lcuc_mapping:Nnn i i I
2791 \int_roman_lcuc_mapping:Nnn v v V
2792 \int_roman_lcuc_mapping:Nnn x x X
2793 \int_roman_lcuc_mapping:Nnn l l L
2794 \int_roman_lcuc_mapping:Nnn c c C
2795 \int_roman_lcuc_mapping:Nnn d d D
2796 \int_roman_lcuc_mapping:Nnn m m M
```

For the delimiter we cheat and let it gobble its arguments instead.

```
2797 \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 `\intexpr_eval:n` first which is safer and more flexible.

```
2798 \cs_new_nopar:Npn \int_to_roman:n #1 {
2799   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN l
2800   \int_to_roman:w \intexpr_eval:n {#1} Q
2801 }
2802 \cs_new_nopar:Npn \int_to_Roman:n #1 {
2803   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN u
2804   \int_to_roman:w \intexpr_eval:n {#1} Q
2805 }
2806 \cs_new_nopar:Npn \int_to_roman_lcuc:NN #1#2{
2807   \use:c {int_to_#1c_roman_#2:}
2808   \int_to_roman_lcuc:NN #1
2809 }
```

`\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.

```
2810 \cs_set_nopar:Npn \int_convert_number_with_rule:nnN #1#2#3{
2811   \intexpr_compare:nNnTF {#1}>{#2}
2812   {
2813     \exp_args:Nf \int_convert_number_with_rule:nnN
```



```

2814 { \intexpr_div_truncate:nn {#1-1}{#2} }{#2}
2815 #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.

```

2816 \exp_args:Nf #3 { \intexpr_eval:n{1+\intexpr_mod:nn {#1-1}{#2}} }
2817 }
2818 { \exp_args:Nf #3{ \intexpr_eval:n{#1} } }
2819 }

```

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.

`\alph_default_conversion_rule:n` Now we just set up a default conversion rule. Ideally every language should have one such rule, as say in Danish there are 29 letters in the alphabet.

```

2820 \cs_new_nopar:Npn \int_alph_default_conversion_rule:n #1{
2821   \if_case:w #1
2822     \or: a\or: b\or: c\or: d\or: e\or: f
2823     \or: g\or: h\or: i\or: j\or: k\or: l
2824     \or: m\or: n\or: o\or: p\or: q\or: r
2825     \or: s\or: t\or: u\or: v\or: w\or: x
2826     \or: y\or: z
2827   \fi:
2828 }
2829 \cs_new_nopar:Npn \int_Alph_default_conversion_rule:n #1{
2830   \if_case:w #1
2831     \or: A\or: B\or: C\or: D\or: E\or: F
2832     \or: G\or: H\or: I\or: J\or: K\or: L
2833     \or: M\or: N\or: O\or: P\or: Q\or: R
2834     \or: S\or: T\or: U\or: V\or: W\or: X
2835     \or: Y\or: Z
2836   \fi:
2837 }

```

`\int_to_alph:n` The actual functions are just instances of the generic function. The second argument of `\int_convert_number_with_rule:nnN` should of course match the number of `\or:s` in the conversion rule.

```

2838 \cs_new_nopar:Npn \int_to_alph:n #1{
2839   \int_convert_number_with_rule:nnN {#1}{26}
2840   \int_alph_default_conversion_rule:n
2841 }
2842 \cs_new_nopar:Npn \int_to_Alph:n #1{
2843   \int_convert_number_with_rule:nnN {#1}{26}
2844   \int_Alph_default_conversion_rule:n
2845 }

```

`\int_to_symbol:n` Turning a number into a symbol is also easy enough.

```

2846 \cs_new_nopar:Npn \int_to_symbol:n #1{
2847   \mode_if_math:TF
2848   {

```

```

2849 \int_convert_number_with_rule:nnN {#1}{9}
2850 \int_symbol_math_conversion_rule:n
2851 }
2852 {
2853 \int_convert_number_with_rule:nnN {#1}{9}
2854 \int_symbol_text_conversion_rule:n
2855 }
2856 }

```

symbol\_math\_conversion\_rule:n Nothing spectacular here.

symbol\_text\_conversion\_rule:n

```

2857 \cs_new_nopar:Npn \int_symbol_math_conversion_rule:n #1 {
2858 \if_case:w #1
2859 \or: *
2860 \or: \dagger
2861 \or: \ddagger
2862 \or: \mathsection
2863 \or: \mathparagraph
2864 \or: \ /
2865 \or: **
2866 \or: \dagger\dagger
2867 \or: \ddagger\ddagger
2868 \fi:
2869 }
2870 \cs_new_nopar:Npn \int_symbol_text_conversion_rule:n #1 {
2871 \if_case:w #1
2872 \or: \textasteriskcentered
2873 \or: \textdagger
2874 \or: \textdaggerdbl
2875 \or: \textsection
2876 \or: \textparagraph
2877 \or: \textbardbl
2878 \or: \textasteriskcentered\textasteriskcentered
2879 \or: \textdagger\textdagger
2880 \or: \textdaggerdbl\textdaggerdbl
2881 \fi:
2882 }

```

\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

```

2883 \int_new:N \l_tmpa_int
2884 \int_new:N \l_tmpb_int
2885 \int_new:N \l_tmpc_int
2886 \int_new:N \g_tmpa_int
2887 \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

```

2888 \cs_set_nopar:Npn \int_pre_eval_one_arg:Nn #1#2{
2889 \exp_args:Nf#1{\intexpr_eval:n{#2}}
2890 \cs_set_nopar:Npn \int_pre_eval_two_args:Nnn #1#2#3{
2891 \exp_args:Nff#1{\intexpr_eval:n{#2}}{\intexpr_eval:n{#3}}
2892 }

```

## 101.4 Defining constants

`\int_const:Nn` As stated, most constants can be defined as `\tex_chardef:D` or `\tex_mathchardef:D` but that's engine dependent.

```

2893 \cs_new_nopar:Npn \int_const:Nn #1#2 {
2894   \intexpr_compare:nNnTF {#2} > \c_minus_one {
2895     \intexpr_compare:nNnTF {#2} > \c_max_register_num {
2896       \int_new:N #1 \int_set:Nn #1{#2}
2897     } {
2898       \chk_if_free_cs:N #1 \tex_mathchardef:D #1 = \intexpr_eval:n{#2}
2899     }
2900   } {
2901     \int_new:N #1 \int_set:Nn #1{#2}
2902   }
2903 }

```

`\c_minus_one` 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.

```

\c_zero
\c_one
\c_two
\c_three 2904 %% \tex_countdef:D \c_minus_one = 10 \scan_stop:
\c_four   2905 %% \c_minus_one = -1 \scan_stop:          %% in l3basics
\c_five   2906 %\int_const:Nn \c_zero {0}
\c_six     2907 \int_const:Nn \c_one {1}
\c_seven   2908 \int_const:Nn \c_two {2}
\c_eight   2909 \int_const:Nn \c_three {3}
\c_nine    2910 \int_const:Nn \c_four {4}
\c_ten      2911 \int_const:Nn \c_five {5}
\c_eleven   2912 \int_const:Nn \c_six {6}
\c_twelve   2913 \int_const:Nn \c_seven {7}
\c_thirteen 2914 \int_const:Nn \c_eight {8}
\c_fourteen 2915 \int_const:Nn \c_nine {9}
\c_fifteen  2916 \int_const:Nn \c_ten {10}
\c_sixteen  2917 \int_const:Nn \c_eleven {11}
\c_seventeen 2918 \int_const:Nn \c_twelve {12}
\c_eighteen 2919 \int_const:Nn \c_thirteen {13}
\c_nineteen 2920 \int_const:Nn \c_fourteen {14}
\c_twenty    2921 \int_const:Nn \c_fifteen {15}
\c_hundred_one 2922 %% \tex_chardef:D \c_sixteen = 16\scan_stop: %% in l3basics
\c_twohundred_fifty_five 2923 \int_const:Nn \c_thirty_two {32}
\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

```

The next one may seem a little odd (obviously!) but is useful when dealing with logical operators.

```

2924 \int_const:Nn \c_hundred_one {101}
2925 \int_const:Nn \c_twohundred_fifty_five{255}
2926 \int_const:Nn \c_twohundred_fifty_six {256}
2927 \int_const:Nn \c_thousand {1000}
2928 \int_const:Nn \c_ten_thousand {10000}
2929 \int_const:Nn \c_ten_thousand_one {10001}
2930 \int_const:Nn \c_ten_thousand_two {10002}
2931 \int_const:Nn \c_ten_thousand_three {10003}
2932 \int_const:Nn \c_ten_thousand_four {10004}
2933 \int_const:Nn \c_twenty_thousand {20000}

```

`\c_max_int` The largest number allowed is  $2^{31} - 1$

2934 `\int_const:Nn \c_max_int {2147483647}`

## 101.5 Scanning and conversion

Conversion between different numbering schemes requires meticulous work. A number can be preceded by any number of + and/or -. We define a generic function which will return the sign and/or the remainder.

```

\int_get_sign_and_digits:n A number may be preceded by any number of +s and -s. Start out by assuming we have
    \int_get_sign:n a positive number.
    \int_get_digits:n
get_sign_and_digits_aux:nNNN 2935 \cs_new_nopar:Npn \int_get_sign_and_digits:n #1{
get_sign_and_digits_aux:oNNN 2936   \int_get_sign_and_digits_aux:nNNN {#1} \c_true_bool \c_true_bool \c_true_bool
2937 }
2938 \cs_new_nopar:Npn \int_get_sign:n #1{
2939   \int_get_sign_and_digits_aux:nNNN {#1} \c_true_bool \c_true_bool \c_false_bool
2940 }
2941 \cs_new_nopar:Npn \int_get_digits:n #1{
2942   \int_get_sign_and_digits_aux:nNNN {#1} \c_true_bool \c_false_bool \c_true_bool
2943 }

```

Now check the first character in the string. Only a - can change if a number is positive or negative, hence we reverse the boolean governing this. Then gobble the - and start over.

```

2944 \cs_new_nopar:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4{
2945   \tl_if_head_eq_charcode:fNTF {#1} -
2946   {
2947     \bool_if:NTF #2
2948     { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} \c_false_bool #3#4 }
2949     { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} \c_true_bool #3#4 }
2950   }

```

The other cases are much simpler since we either just have to gobble the + or exit immediately and insert the correct sign.

```

2951 {
2952   \tl_if_head_eq_charcode:fNTF {#1} +
2953   { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} #2#3#4}
2954   {

```

The boolean #3 is for printing the sign while #4 is for printing the digits.

```

2955     \bool_if:NT #3 { \bool_if:NF #2 - }
2956     \bool_if:NT #4 {#1}
2957   }
2958 }
2959 }
2960 \cs_generate_variant:Nn \int_get_sign_and_digits_aux:nNNN {oNNN}

```

`\int_convert_from_base_ten:nn` #1 is the base 10 number to be converted to base #2. We split off the sign first, print if there and then convert only the number. Since this is supposedly a base 10 number we can let T<sub>E</sub>X do the reading of + and -.

```

2961 \cs_set_nopar:Npn \int_convert_from_base_ten:nn#1#2{
2962   \intexpr_compare:nNnTF {#1}<\c_zero
2963   {
2964     - \int_convert_from_base_ten_aux:nfn {}
2965     { \intexpr_eval:n {-#1} }
2966   }
2967   {
2968     \int_convert_from_base_ten_aux:nfn {}
2969     { \intexpr_eval:n {#1} }
2970   }
2971   {#2}
2972 }

```

The algorithm runs like this:

1. If the number  $\langle num \rangle$  is greater than  $\langle base \rangle$ , calculate modulus of  $\langle num \rangle$  and  $\langle base \rangle$  and carry that over for next round. The remainder is calculated as a truncated division of  $\langle num \rangle$  and  $\langle base \rangle$ . Start over with these new values.
2. If  $\langle num \rangle$  is less than or equal to  $\langle base \rangle$  convert it to the correct symbol, print the previously calculated digits and exit.

#1 is the carried over result, #2 the remainder and #3 the base number.

```

2973 \cs_new_nopar:Npn \int_convert_from_base_ten_aux:nnn#1#2#3{
2974   \intexpr_compare:nNnTF {#2}<{#3}
2975   { \int_convert_number_to_letter:n{#2} #1 }
2976   {
2977     \int_convert_from_base_ten_aux:ffn
2978     {
2979       \int_convert_number_to_letter:n {\intexpr_mod:nn {#2}{#3}}
2980       #1
2981     }
2982     { \intexpr_div_truncate:nn{#2}{#3}}
2983     {#3}
2984   }
2985 }
2986 \cs_generate_variant:Nn \int_convert_from_base_ten_aux:nnn {nfn}
2987 \cs_generate_variant:Nn \int_convert_from_base_ten_aux:nnn {ffn}

```

`\int_convert_number_to_letter:n` Turning a number for a different base into a letter or digit.

```

2988 \cs_set_nopar:Npn \int_convert_number_to_letter:n #1{
2989   \if_case:w \intexpr_eval:w #1-10\intexpr_eval_end:
2990   \exp_after:wN A \or: \exp_after:wN B \or:
2991   \exp_after:wN C \or: \exp_after:wN D \or: \exp_after:wN E \or:
2992   \exp_after:wN F \or: \exp_after:wN G \or: \exp_after:wN H \or:
2993   \exp_after:wN I \or: \exp_after:wN J \or: \exp_after:wN K \or:
2994   \exp_after:wN L \or: \exp_after:wN M \or: \exp_after:wN N \or:
2995   \exp_after:wN O \or: \exp_after:wN P \or: \exp_after:wN Q \or:

```

```

2996 \exp_after:wN R \or: \exp_after:wN S \or: \exp_after:wN T \or:
2997 \exp_after:wN U \or: \exp_after:wN V \or: \exp_after:wN W \or:
2998 \exp_after:wN X \or: \exp_after:wN Y \or: \exp_after:wN Z \else:
2999 \use_i_after_fi:nw{ #1 }\fi: }

```

`\int_convert_to_base_ten:nn` #1 is the number, #2 is its base. First we get the sign, then use only the digits/letters from it and pass that onto a new function.

```

3000 \cs_set_nopar:Npn \int_convert_to_base_ten:nn #1#2 {
3001   \intexpr_eval:n{
3002     \int_get_sign:n{#1}
3003     \exp_args:Nf\int_convert_to_base_ten_aux:nn {\int_get_digits:n{#1}}{#2}
3004   }
3005 }

```

This is an intermediate function to get things started.

```

3006 \cs_new_nopar:Npn \int_convert_to_base_ten_aux:nn #1#2{
3007   \int_convert_to_base_ten_auxi:nnN {0}{#2} #1 \q_nil
3008 }

```

Here we check each letter/digit and calculate the next number. #1 is the previously calculated result (to be multiplied by the base), #2 is the base and #3 is the next letter/digit to be added.

```

3009 \cs_new_nopar:Npn \int_convert_to_base_ten_auxi:nnN#1#2#3{
3010   \quark_if_nil:NTF #3
3011   {#1}
3012   {\exp_args:Nf\int_convert_to_base_ten_auxi:nnN
3013     {\intexpr_eval:n{ #1*#2+\int_convert_letter_to_number:N #3} }
3014     {#2}
3015   }
3016 }

```

This is for turning a letter or digit into a number. This function also takes care of handling lowercase and uppercase letters. Hence a is turned into 11 and so is A.

```

3017 \cs_set_nopar:Npn \int_convert_letter_to_number:N #1{
3018   \intexpr_compare:nNnTF{‘#1}<{58}{#1}
3019   {
3020     \intexpr_eval:n{ ‘#1 -
3021       \intexpr_compare:nNnTF{‘#1}<{91}{ 55 }{ 87 }
3022     }
3023   }
3024 }

3025 </initex | package>

```

Show token usage:

```

3026 <*showmemory>
3027 \showMemUsage
3028 </showmemory>

```

## 102 l3num implementation

We start by ensuring that the required packages are loaded.

```

3029 <*package>
3030 \ProvidesExplPackage
3031   {\filename}{\filedate}{\fileversion}{\filedescription}
3032 \package_check_loaded_expl:
3033 </package>
3034 <*initex | package>

```

`\if_num:w` Here are the remaining primitives for number comparisons and expressions.

```

\if_case:w
3035 \cs_new_eq:NN \if_num:w \tex_ifnum:D
3036 \cs_new_eq:NN \if_case:w \tex_ifcase:D

```

Functions that support L<sup>A</sup>T<sub>E</sub>X's user accessible counters should be added here, too. But first the internal counters.

`\num_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\num_decr:N
\num_gincr:N 3037 \cs_set_nopar:Npn \num_incr:N #1{\num_add:Nn#1 1}
\num_gdecr:N 3038 \cs_set_nopar:Npn \num_decr:N #1{\num_add:Nn#1 \c_minus_one}
3039 \cs_set_nopar:Npn \num_gincr:N #1{\num_gadd:Nn#1 1}
3040 \cs_set_nopar:Npn \num_gdecr:N #1{\num_gadd:Nn#1 \c_minus_one}

```

`\num_incr:c` We also need ...

```

\num_decr:c
\num_gincr:c 3041 \cs_generate_variant:Nn \num_incr:N {c}
\num_gdecr:c 3042 \cs_generate_variant:Nn \num_decr:N {c}
3043 \cs_generate_variant:Nn \num_gincr:N {c}
3044 \cs_generate_variant:Nn \num_gdecr:N {c}

```

`\num_zero:N` We also need ...

```

\num_zero:c
\num_gzero:N 3045 \cs_new_nopar:Npn \num_zero:N #1 {\num_set:Nn #1 0}
\num_gzero:c 3046 \cs_new_nopar:Npn \num_gzero:N #1 {\num_gset:Nn #1 0}

3047 \cs_generate_variant:Nn \num_zero:N {c}
3048 \cs_generate_variant:Nn \num_gzero:N {c}

```

`\num_new:N` Allocate a new `<num>` variable and initialize it with zero.

```

\num_new:c
3049 \cs_new_nopar:Npn \num_new:N #1{\tl_new:Nn #1{0}}
3050 \cs_generate_variant:Nn \num_new:N {c}

```

`\num_set:Nn` Assigning values to `<num>` registers.

```

\num_set:cn
\num_gset:Nn 3051 \cs_new_nopar:Npn \num_set:Nn #1#2{
\num_gset:cn 3052   \tl_set:Nn #1{\tex_number:D \intexpr_eval:n {#2}}
3053 }
3054 \cs_generate_variant:Nn \num_set:Nn {c}

```

```

3055 \cs_new_nopar:Npn \num_gset:Nn {\pref_global:D \num_set:Nn}
3056 \cs_generate_variant:Nn\num_gset:Nn {c}

```

`\num_set_eq:NN` Setting  $\langle num \rangle$  registers equal to each other.

```

\num_set_eq:cN
\num_set_eq:Nc
\num_set_eq:cc
3057 \cs_new_eq:NN \num_set_eq:NN \tl_set_eq:NN
3058 \cs_generate_variant:Nn\num_set_eq:NN {c,Nc,cc}

```

`\num_gset_eq:NN` Setting  $\langle num \rangle$  registers equal to each other.

```

\num_gset_eq:cN
\num_gset_eq:Nc
\num_gset_eq:cc
3059 \cs_new_eq:NN \num_gset_eq:NN \tl_gset_eq:NN
3060 \cs_generate_variant:Nn\num_gset_eq:NN {c,Nc,cc}

```

`\num_add:Nn` Adding is easily done as the second argument goes through `\intexpr_eval:n`.

```

\num_add:cN
\num_gadd:Nn
\num_gadd:cn
3061 \cs_new_nopar:Npn \num_add:Nn #1#2 {\num_set:Nn #1{#1+#2}}
3062 \cs_generate_variant:Nn\num_add:Nn {c}

3063 \cs_new_nopar:Npn \num_gadd:Nn {\pref_global:D \num_add:Nn}
3064 \cs_generate_variant:Nn\num_gadd:Nn {c}

```

`\num_use:N` Here is how num macros are accessed:

```

\num_use:c
3065 \cs_new_eq:NN\num_use:N \use:n
3066 \cs_new_eq:NN\num_use:c \use:c

```

`\num_show:N` Here is how num macros are diagnosed:

```

\num_show:c
3067 \cs_new_eq:NN\num_show:N \cs_show:N
3068 \cs_new_eq:NN\num_show:c \cs_show:c

```

`\num_elt_count:n` Helper function for counting elements in a list.

```

\num_elt_count_prop:Nn
3069 \cs_new:Npn \num_elt_count:n #1 { + 1 }
3070 \cs_new:Npn \num_elt_count_prop:Nn #1#2 { + 1 }

```

`\l_tmpa_num` We provide an number local and two global  $\langle num \rangle$ s, maybe we need more or less.

```

\l_tmpb_num
\l_tmpc_num
3071 \num_new:N \l_tmpa_num
3072 \num_new:N \l_tmpb_num
3073 \num_new:N \l_tmpc_num
\g_tmpa_num
3074 \num_new:N \g_tmpa_num
\g_tmpb_num
3075 \num_new:N \g_tmpb_num

```

`\c_max_register_num`

```

3076 \tex_mathchardef:D \c_max_register_num = 32767 \scan_stop:

3077 \</initex | package>

```



## 103 l3intexpr implementation

We start by ensuring that the required packages are loaded.

```

3078 <*package>
3079 \ProvidesExplPackage
3080   {\filename}{\filedate}{\fileversion}{\filedescription}
3081   \package_check_loaded_expl:
3082 </package>
3083 <*initex | package>

```

```

\intexpr_value:w Here are the remaining primitives for number comparisons and expressions.
\intexpr_eval:n
\intexpr_eval:w
\intexpr_eval_end:
\if_intexpr_compare:w
\if_intexpr_odd:w
\if_intexpr_case:w
3084 \cs_set_eq:NN \intexpr_value:w \tex_number:D
3085 \cs_set_eq:NN \intexpr_eval:w \etex_numexpr:D
3086 \cs_set_protected:Npn \intexpr_eval_end: {\tex_relax:D}
3087 \cs_set_eq:NN \if_intexpr_compare:w \tex_ifnum:D
3088 \cs_set_eq:NN \if_intexpr_odd:w \tex_ifodd:D
3089 \cs_set_eq:NN \if_intexpr_case:w \tex_ifcase:D
3090 \cs_set:Npn \intexpr_eval:n #1{
3091   \intexpr_value:w \intexpr_eval:w #1\intexpr_eval_end:
3092 }

```

```

\intexpr_compare_p:n Comparison tests using a simple syntax where only one set of braces is required and
\intexpr_compare:nTF additional operators such as != and >= are supported. First some notes on the idea
behind this. We wish to support writing code like

```

```

\intexpr_compare_p:n { 5 + \l_tmpa_int != 4 - \l_tmpb_int }

```

In other words, we want to somehow add the missing `\intexpr_eval:w` where required. We can start evaluating from the left using `\intexpr: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 T<sub>E</sub>X evaluate this left hand side of the (in)equality.

```

3093 \prg_set_conditional:Npnn \intexpr_compare:n #1{p,TF,T,F}{
3094   \exp_after:wN \intexpr_compare_auxi:w \intexpr_value:w
3095   \intexpr_eval:w #1\q_stop
3096 }

```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is `<null>`. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3097 \cs_set:Npn \intexpr_compare_auxi:w #1#2\q_stop{
3098   \exp_after:wN \intexpr_compare_auxii:w \tex_romannumeral:D
3099   \if:w #1- \else: -\fi: #1#2 \q_stop #1#2 \q_nil
3100 }

```

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.

```

3101 \cs_set:Npn \intexpr_compare_auxii:w #1#2#3\q_stop{
3102   \use:c{
3103     intexpr_compare_
3104     #1 \if_meaning:w =#2 = \fi:
3105     :w}
3106 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3107 \cs_set:cpn {intexpr_compare_=:w} #1=#2\q_nil{
3108   \if_intexpr_compare:w #1=\intexpr_eval:w #2 \intexpr_eval_end:
3109   \prg_return_true: \else: \prg_return_false: \fi:
3110 }

```

So is the one using  $==$  – we just have to use  $==$  in the parameter text.

```

3111 \cs_set:cpn {intexpr_compare_==:w} #1==#2\q_nil{
3112   \if_intexpr_compare:w #1=\intexpr_eval:w #2 \intexpr_eval_end:
3113   \prg_return_true: \else: \prg_return_false: \fi:
3114 }

```

Not equal is just about reversing the truth value.

```

3115 \cs_set:cpn {intexpr_compare_!=:w} #1!=#2\q_nil{
3116   \if_intexpr_compare:w #1=\intexpr_eval:w #2 \intexpr_eval_end:
3117   \prg_return_false: \else: \prg_return_true: \fi:
3118 }

```

Less than and greater than are also straight forward.

```

3119 \cs_set:cpn {intexpr_compare_<:w} #1<#2\q_nil{
3120   \if_intexpr_compare:w #1<\intexpr_eval:w #2 \intexpr_eval_end:
3121   \prg_return_true: \else: \prg_return_false: \fi:
3122 }
3123 \cs_set:cpn {intexpr_compare_>:w} #1>#2\q_nil{
3124   \if_intexpr_compare:w #1>\intexpr_eval:w #2 \intexpr_eval_end:
3125   \prg_return_true: \else: \prg_return_false: \fi:
3126 }

```

For the less than or equal operation, we simply add 1 to the right hand side and then use a less than test. A similar trick for the greater than or equal test except there we subtract 1.

```

3127 \cs_set:cpn {intexpr_compare_<=:w} #1<=#2\q_nil{
3128   \if_intexpr_compare:w #1<\intexpr_eval:w #2 +\c_one \intexpr_eval_end:
3129   \prg_return_true: \else: \prg_return_false: \fi:
3130 }
3131 \cs_set:cpn {intexpr_compare_>=:w} #1>=#2\q_nil{
3132   \if_intexpr_compare:w #1>\intexpr_eval:w #2 - \c_one \intexpr_eval_end:
3133   \prg_return_true: \else: \prg_return_false: \fi:
3134 }

```

`\intexpr_compare_p:nNn` More efficient but less natural in typing.

```
\intexpr_compare:nNnTF
3135 \prg_set_conditional:Npnn \intexpr_compare:nNn #1#2#3{p,TF,T,F}{
3136   \if_intexpr_compare:w \intexpr_eval:w #1 #2 \intexpr_eval:w #3 \intexpr_eval_end:
3137   \prg_return_true: \else: \prg_return_false: \fi:
3138 }
```

`\intexpr_max:nn` Functions for min, max, and absolute value.

```
\intexpr_min:nn
\intexpr_abs:n
3139 \cs_set:Npn \intexpr_abs:n #1{
3140   \intexpr_value:w
3141   \if_intexpr_compare:w \intexpr_eval:w #1<\c_zero
3142   -
3143   \fi:
3144   \intexpr_eval:w #1\intexpr_eval_end:
3145 }
3146 \cs_set:Npn \intexpr_max:nn #1#2{
3147   \intexpr_value:w \intexpr_eval:w
3148   \if_intexpr_compare:w
3149     \intexpr_eval:w #1>\intexpr_eval:w #2\intexpr_eval_end:
3150     #1
3151   \else:
3152     #2
3153   \fi:
3154   \intexpr_eval_end:
3155 }
3156 \cs_set:Npn \intexpr_min:nn #1#2{
3157   \intexpr_value:w \intexpr_eval:w
3158   \if_intexpr_compare:w
3159     \intexpr_eval:w #1<\intexpr_eval:w #2\intexpr_eval_end:
3160     #1
3161   \else:
3162     #2
3163   \fi:
3164   \intexpr_eval_end:
3165 }
```

`\intexpr_div_truncate:nn` As `\intexpr_eval:w` rounds the result of a division we also provide a version that truncates the result.

`\intexpr_div_round:nn`

`\intexpr_mod:nn`

Initial version didn't work correctly with eTeX's implementation.

```
3166 %\cs_set:Npn \intexpr_div_truncate_raw:nn #1#2 {
3167 % \intexpr_eval:n{ (2*#1 - #2) / (2* #2) }
3168 %}
```

New version by Heiko:

```
3169 \cs_set:Npn \intexpr_div_truncate:nn #1#2 {
3170   \intexpr_value:w \intexpr_eval:w
3171   \if_intexpr_compare:w \intexpr_eval:w #1 = \c_zero
3172   0
3173   \else:
3174     (#1
```

```

3175 \if_intexpr_compare:w \intexpr_eval:w #1 < \c_zero
3176 \if_intexpr_compare:w \intexpr_eval:w #2 < \c_zero
3177 -( #2 +
3178 \else:
3179 +( #2 -
3180 \fi:
3181 \else:
3182 \if_intexpr_compare:w \intexpr_eval:w #2 < \c_zero
3183 +( #2 +
3184 \else:
3185 -( #2 -
3186 \fi:
3187 \fi:
3188 1)/2)
3189 \fi:
3190 /(#2)
3191 \intexpr_eval_end:
3192 }

```

For the sake of completeness:

```

3193 \cs_set:Npn \intexpr_div_round:nn #1#2 {\intexpr_eval:n{(#1)/(#2)}}

```

Finally there's the modulus operation.

```

3194 \cs_set:Npn \intexpr_mod:nn #1#2 {
3195 \intexpr_value:w
3196 \intexpr_eval:w
3197 #1 - \intexpr_div_truncate:nn {#1}{#2} * (#2)
3198 \intexpr_eval_end:
3199 }

```

`\intexpr_if_odd_p:n` A predicate function.

```

\intexpr_if_odd:nTF
\intexpr_if_even_p:n
\intexpr_if_even:nTF
3200 \prg_set_conditional:Npnn \intexpr_if_odd:n #1 {p,TF,T,F} {
3201 \if_intexpr_odd:w \intexpr_eval:w #1\intexpr_eval_end:
3202 \prg_return_true: \else: \prg_return_false: \fi:
3203 }
3204 \prg_set_conditional:Npnn \intexpr_if_even:n #1 {p,TF,T,F} {
3205 \if_intexpr_odd:w \intexpr_eval:w #1\intexpr_eval_end:
3206 \prg_return_false: \else: \prg_return_true: \fi:
3207 }

```

`\intexpr_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.

```

\intexpr_until_do:nn
\intexpr_do_while:nn
\intexpr_do_until:nn
3208 \cs_set:Npn \intexpr_while_do:nn #1#2{
3209 \intexpr_compare:nT {#1}{#2} \intexpr_while_do:nn {#1}{#2}}
3210 }
3211 \cs_set:Npn \intexpr_until_do:nn #1#2{
3212 \intexpr_compare:nF {#1}{#2} \intexpr_until_do:nn {#1}{#2}}
3213 }
3214 \cs_set:Npn \intexpr_do_while:nn #1#2{
3215 #2 \intexpr_compare:nT {#1}{\intexpr_do_while:nNnn {#1}{#2}}

```

```

3216 }
3217 \cs_set:Npn \intexpr_do_until:nn #1#2{
3218   #2 \intexpr_compare:nF {#1}{\intexpr_do_until:nn {#1}{#2}}
3219 }

```

\intexpr\_while\_do:nNnn As above but not using the more natural syntax.

```

\intexpr_until_do:nNnn
\intexpr_do_while:nNnn
\intexpr_do_until:nNnn
3220 \cs_set:Npn \intexpr_while_do:nNnn #1#2#3#4{
3221   \intexpr_compare:nNnT {#1}#2{#3}{#4 \intexpr_while_do:nNnn {#1}#2{#3}{#4}}
3222 }
3223 \cs_set:Npn \intexpr_until_do:nNnn #1#2#3#4{
3224   \intexpr_compare:nNnF {#1}#2{#3}{#4 \intexpr_until_do:nNnn {#1}#2{#3}{#4}}
3225 }
3226 \cs_set:Npn \intexpr_do_while:nNnn #1#2#3#4{
3227   #4 \intexpr_compare:nNnT {#1}#2{#3}{\intexpr_do_while:nNnn {#1}#2{#3}{#4}}
3228 }
3229 \cs_set:Npn \intexpr_do_until:nNnn #1#2#3#4{
3230   #4 \intexpr_compare:nNnF {#1}#2{#3}{\intexpr_do_until:nNnn {#1}#2{#3}{#4}}
3231 }
3232 </initex | package>

```

## 104 l3skip implementation

We start by ensuring that the required packages are loaded.

```

3233 <*package>
3234 \ProvidesExplPackage
3235   {\filename}{\filedate}{\fileversion}{\filedescription}
3236 \package_check_loaded_expl:
3237 </package>
3238 <*initex | package>

```

### 104.1 Skip registers

\skip\_new:N Allocation of a new internal registers.

```

\skip_new:c
3239 <*initex>
3240 \alloc_setup_type:nnn {skip} \c_zero \c_max_register_num
3241 \cs_new_nopar:Npn\skip_new:N #1 { \alloc_reg:NnNN g {skip} \tex_skipdef:D #1 }
3242 \cs_new_nopar:Npn\skip_new_l:N #1 { \alloc_reg:NnNN l {skip} \tex_skipdef:D #1 }
3243 </initex>
3244 <*package>
3245 \cs_new_nopar:Npn \skip_new:N #1 {
3246   \chk_if_free_cs:N #1
3247   \newskip #1
3248 }
3249 </package>
3250 \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
3251 \cs_new_nopar:Npn \skip_set:Nn #1#2 {
3252   #1\skip_eval:n{#2}
3253   \<check>
3254   \chk_local_or_pref_global:N #1
3255   \</check>
3256 }
3257 \cs_new_nopar:Npn \skip_gset:Nn {
3258   \<check>
3259   \pref_global_chk:
3260   \</check>
3261   \<-check> \pref_global:D
3262   \skip_set:Nn
3263 }
3264 \cs_generate_variant:Nn \skip_set:Nn {cn}
3265 \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
3266 \cs_new_nopar:Npn \skip_zero:N #1{
3267   #1\c_zero_skip \scan_stop:
3268   \<check>
3269   \chk_local_or_pref_global:N #1
3270   \</check>
3271 }
3272 \cs_new_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.

```

3273 \<check>
3274 \pref_global_chk:
3275 \</check>
3276 \<-check> \pref_global:D
3277   \skip_zero:N
3278 }
3279 \cs_generate_variant:Nn \skip_zero:N {c}
3280 \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
3281 \cs_new_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
3282   \tex_advance:D#1 by \skip_eval:n{#2}
3283   \<check>
3284   \chk_local_or_pref_global:N #1

```

```

3285 \</check>
3286 }
3287 \cs_generate_variant:Nn \skip_add:Nn {cn}

3288 \cs_new_nopar:Npn \skip_sub:Nn #1#2{
3289   \tex_advance:D#1-\skip_eval:n{#2}
3290 \< *check>
3291   \chk_local_or_pref_global:N #1
3292 \< /check>
3293 }

3294 \cs_new_nopar:Npn \skip_gadd:Nn {
3295 \< *check>
3296   \pref_global_chk:
3297 \< /check>
3298 \< -check> \pref_global:D
3299   \skip_add:Nn
3300 }
3301 \cs_generate_variant:Nn \skip_gadd:Nn {cn}

3302 \cs_new_nopar:Npn \skip_gsub:Nn {
3303 \< *check>
3304   \pref_global_chk:
3305 \< /check>
3306 \< -check> \pref_global:D
3307   \skip_sub:Nn
3308 }

```

\skip\_horizontal:N Inserting skips.

```

\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n
3309 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
3310 \cs_generate_variant:Nn \skip_horizontal:N {c}

3311 \cs_new_nopar:Npn \skip_horizontal:n #1 { \skip_horizontal:N \skip_eval:n{#1} }
3312 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
3313 \cs_generate_variant:Nn \skip_vertical:N {c}
3314 \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
3315 \cs_new_eq:NN \skip_use:N \tex_the:D
3316 \cs_generate_variant:Nn \skip_use:N {c}

```

\skip\_show:N Diagnostics.

```

\skip_show:c
3317 \cs_new_eq:NN \skip_show:N \tex_showthe:D
3318 \cs_new_nopar:Npn \skip_show:c #1 { \skip_show:N \cs:w #1 \cs_end: }

```

\skip\_eval:n Evaluating a calc expression.

```

3319 \cs_new_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
3320 %%\chk_if_free_cs:N \l_tmpa_skip
3321 %%\tex_skipdef:D\l_tmpa_skip 255 %currently taken up by \skip@
3322 \skip_new:N \l_tmpa_skip
3323 \skip_new:N \l_tmpb_skip
3324 \skip_new:N \l_tmpc_skip
3325 \skip_new:N \g_tmpa_skip
3326 \skip_new:N \g_tmpb_skip

```

```

\c_zero_skip
\c_max_skip
3327 <!\package>
3328 \skip_new:N \c_zero_skip
3329 \skip_set:Nn \c_zero_skip {0pt}
3330 \skip_new:N \c_max_skip
3331 \skip_set:Nn \c_max_skip {16383.99999pt}
3332 </!\package>
3333 <!\linitex>
3334 \cs_set_eq:NN \c_zero_skip \z@
3335 \cs_set_eq:NN \c_max_skip \maxdimen
3336 </!\linitex>

```

`\skip_if_infinite_glue_p:n` 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.

```

3337 \prg_new_conditional:Nnn \skip_if_infinite_glue:n {p,TF,T,F} {
3338   \bool_if:nTF {
3339     \intexpr_compare_p:nNn {\etex_gluestretchorder:D #1 } > \c_zero ||
3340     \intexpr_compare_p:nNn {\etex_glueshrinkorder:D #1 } > \c_zero
3341   } {\prg_return_true:} {\prg_return_false:}
3342 }

```

`\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.

```

3343 \cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4{
3344   \skip_if_infinite_glue:nTF {#1}
3345   {
3346     #3 = \c_zero_skip
3347     #4 = \c_zero_skip
3348     #2
3349   }
3350   {
3351     #3 = \etex_gluestretch:D #1 \scan_stop:
3352     #4 = \etex_glueshrink:D #1 \scan_stop:
3353   }
3354 }

```



## 104.2 Dimen registers

`\dim_new:N` Allocating  $\langle dim \rangle$  registers...

```

\dim_new:c
3355 \<initex>
3356 \alloc_setup_type:nnn {dimen} \c_zero \c_max_register_num
3357 \cs_new_nopar:Npn \dim_new:N #1 {\alloc_reg:NnNN g {dimen} \tex_dimendef:D #1 }
3358 \cs_new_nopar:Npn \dim_new_l:N #1 {\alloc_reg:NnNN l {dimen} \tex_dimendef:D #1 }
3359 \</initex>
3360 \<*package>
3361 \cs_new_nopar:Npn \dim_new:N #1 {
3362   \chk_if_free_cs:N #1
3363   \newdimen #1
3364 }
3365 \</package>
3366 \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 `\dimen1` or alike. See OR!

```

\dim_set:cn
\dim_set:Nc
3367 \cs_new_nopar:Npn \dim_set:Nn #1#2 { #1~ \dim_eval:n{#2} }
3368 \cs_generate_variant:Nn \dim_set:Nn {cn,Nc}
\dim_gset:Nn
\dim_gset:cn
3369 \cs_new_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn }
\dim_gset:Nc
3370 \cs_generate_variant:Nn \dim_gset:Nn {cn,Nc,cc}

```

`\dim_zero:N` Resetting.

```

\dim_gzero:N
\dim_zero:c
3371 \cs_new_nopar:Npn \dim_zero:N #1 { #1\c_zero_skip }
\dim_gzero:c
3372 \cs_generate_variant:Nn \dim_zero:N {c}

3373 \cs_new_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N }
3374 \cs_generate_variant:Nn \dim_gzero:N {c}

```

`\dim_add:Nn` Addition.

```

\dim_add:cn
3375 \cs_new_nopar:Npn \dim_add:Nn #1#2{
\dim_add:Nc
\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.

```

```

3376   \tex_advance:D#1 by \dim_eval:n{#2}\scan_stop:
3377 }
3378 \cs_generate_variant:Nn \dim_add:Nn {cn,Nc}

3379 \cs_new_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
3380 \cs_generate_variant:Nn \dim_gadd:Nn {cn}

```

`\dim_sub:Nn` Subtracting.

```

\dim_sub:cn
3381 \cs_new_nopar:Npn \dim_sub:Nn #1#2 { \tex_advance:D#1-#2\scan_stop: }
\dim_sub:Nc
3382 \cs_generate_variant:Nn \dim_sub:Nn {cn,Nc}
\dim_gsub:Nn
\dim_gsub:cn

```

```

3383 \cs_new_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn }
3384 \cs_generate_variant:Nn \dim_gsub:Nn {cn}

\dim_use:N Accessing a  $\langle dim \rangle$ .
\dim_use:c
3385 \cs_new_eq:NN \dim_use:N \tex_the:D
3386 \cs_generate_variant:Nn \dim_use:N {c}

\dim_show:N Diagnostics.
\dim_show:c
3387 \cs_new_eq:NN \dim_show:N \tex_showthe:D
3388 \cs_new_nopar:Npn \dim_show:c #1 { \dim_show:N \cs:w #1 \cs_end: }

\l_tmpa_dim Some scratch registers.
\l_tmpb_dim
\l_tmpc_dim
\l_tmpd_dim
\g_tmpa_dim
\g_tmpb_dim
3389 \dim_new:N \l_tmpa_dim
3390 \dim_new:N \l_tmpb_dim
3391 \dim_new:N \l_tmpc_dim
3392 \dim_new:N \l_tmpd_dim
3393 \dim_new:N \g_tmpa_dim
3394 \dim_new:N \g_tmpb_dim

\c_zero_dim Just aliases.
\c_max_dim
3395 \cs_new_eq:NN \c_zero_dim \c_zero_skip
3396 \cs_new_eq:NN \c_max_dim \c_max_skip

\dim_eval:n Evaluating a calc expression.
3397 \cs_new_nopar:Npn \dim_eval:n #1 { \etex_dimexpr:D #1 \scan_stop: }

\if_dim:w The comparison primitive.
3398 \cs_new_eq:NN \if_dim:w \tex_ifdim:D

\dim_compare_p:nNn
\dim_compare:nNnTF
3399 \prg_new_conditional:Nnn \dim_compare:nNn {p,TF,T,F} {
3400   \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
3401   \prg_return_true: \else: \prg_return_false: \fi:
3402 }

\dim_while_do:nNnn while_do and do_while functions for dimensions. Same as for the int type only the
\dim_until_do:nNnn names have changed.
\dim_do_while:nNnn
\dim_do_until:nNnn
3403 \cs_new_nopar:Npn \dim_while_do:nNnn #1#2#3#4{
3404   \dim_compare:nNnT {#1}#2{#3}{#4} \dim_while_do:nNnn {#1}#2{#3}{#4}}
3405 }
3406 \cs_new_nopar:Npn \dim_until_do:nNnn #1#2#3#4{
3407   \dim_compare:nNnF {#1}#2{#3}{#4} \dim_until_do:nNnn {#1}#2{#3}{#4}}
3408 }
3409 \cs_new_nopar:Npn \dim_do_while:nNnn #1#2#3#4{
3410   #4 \dim_compare:nNnT {#1}#2{#3}{\dim_do_while:nNnn {#1}#2{#3}{#4}}
3411 }
3412 \cs_new_nopar:Npn \dim_do_until:nNnn #1#2#3#4{
3413   #4 \dim_compare:nNnF {#1}#2{#3}{\dim_do_until:nNnn {#1}#2{#3}{#4}}
3414 }

```

## 104.3 Muskips

`\muskip_new:N` And then we add muskips.

```
3415 <*initex>
3416 \alloc_setup_type:nnn {muskip} \c_zero \c_max_register_num
3417 \cs_new_nopar:Npn \muskip_new:N #1{\alloc_reg:NnNN g {muskip} \tex_muskipdef:D #1}
3418 \cs_new_nopar:Npn \muskip_new_l:N #1{\alloc_reg:NnNN l {muskip} \tex_muskipdef:D #1}
3419 </initex>
3420 <*package>
3421 \cs_new_nopar:Npn \muskip_new:N #1 {
3422   \chk_if_free_cs:N #1
3423   \newmuskip #1
3424 }
3425 </package>
```

`\muskip_set:Nn` Simple functions for muskips.

```
\muskip_gset:Nn
\muskip_add:Nn
\muskip_gadd:Nn
\muskip_sub:Nn
\muskip_gsub:Nn
3426 \cs_new_nopar:Npn \muskip_set:Nn#1#2{#1\etex_muexpr:D#2\scan_stop:}
3427 \cs_new_nopar:Npn \muskip_gset:Nn{\pref_global:D\muskip_set:Nn}
3428 \cs_new_nopar:Npn \muskip_add:Nn#1#2{\tex_advance:D#1\etex_muexpr:D#2\scan_stop:}
3429 \cs_new_nopar:Npn \muskip_gadd:Nn{\pref_global:D\muskip_add:Nn}
3430 \cs_new_nopar:Npn \muskip_sub:Nn#1#2{\tex_advance:D#1-\etex_muexpr:D#2\scan_stop:}
3431 \cs_new_nopar:Npn \muskip_gsub:Nn{\pref_global:D\muskip_sub:Nn}
```

`\muskip_use:N` Accessing a `<muskip>`.

```
3432 \cs_new_eq:NN \muskip_use:N \tex_the:D
3433 </initex | package>
```

## 105 l3tl implementation

We start by ensuring that the required packages are loaded.

```
3434 <*package>
3435 \ProvidesExplPackage
3436   {\filename}{\filedate}{\fileversion}{\filedescription}
3437   \package_check_loaded_expl:
3438 </package>
3439 <*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.)

## 105.1 Functions

`\tl_new:N` We provide one allocation function (which checks that the name is not used) and two clear functions that locally or globally clear the token list. The allocation function has two arguments to specify an initial value. This is the only way to give values to constants.

`\tl_new:c`  
`\tl_new:Nn`  
`\tl_new:cn`  
`\tl_new:Nx`

```

3440 \cs_new:Npn \tl_new:Nn #1#2{
3441   \chk_if_free_cs:N #1

```

If checking we don't allow constants to be defined.

```

3442 <*check>
3443   \chk_var_or_const:N #1
3444 </check>

```

Otherwise any variable type is allowed.

```

3445   \cs_gset_nopar:Npn #1{#2}
3446 }
3447 \cs_generate_variant:Nn \tl_new:Nn {cn}
3448 \cs_new:Npn \tl_new:Nx #1#2{
3449   \chk_if_free_cs:N #1
3450 <check> \chk_var_or_const:N #1
3451   \cs_gset_nopar:Npx #1{#2}
3452 }
3453 \cs_new_nopar:Npn \tl_new:N #1{\tl_new:Nn #1{}}
3454 \cs_new_nopar:Npn \tl_new:c #1{\tl_new:cn {#1}{}}

```

`\tl_use:N` Perhaps this should just be enabled when checking?  
`\tl_use:c`

```

3455 \cs_new_nopar:Npn \tl_use:N #1 {
3456   \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.

```

3457   \msg_kernel_bug:x {Token-list-variable~ '\token_to_str:N #1'~
3458                     has~ an~ erroneous~ structure!}
3459   \else:
3460     \exp_after:wN #1
3461   \fi:
3462 }
3463 \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`

```

3464 \cs_new_nopar:Npn \tl_show:N #1 { \cs_show:N #1 }
3465 \cs_generate_variant:Nn \tl_show:N {c}
3466 \cs_set_eq:NN \tl_show:n \etex_showtokens:D

```

`\tl_set:Nn` To set token lists to a specific value to type of functions are available: `\tl_set_eq:NN` takes two token-lists as its arguments assign the first the contents of the second;  
`\tl_set:Nv` `\tl_set:Nn` has as its second argument a 'real' list of tokens. One can view  
`\tl_set:Nv`  
`\tl_set:Nf`  
`\tl_set:Nx`  
`\tl_set:cn`  
`\tl_set:cV`  
`\tl_set:co`  
`\tl_set:cx`  
`\tl_gset:Nn`  
`\tl_gset:Nv`  
`\tl_gset:No`

\tl\_set\_eq:NN as a special form of \tl\_set:No. Both functions have global counterparts.

During development we check if the token list that is being assigned to exists. If not, a warning will be issued.

```

3467 <*check>
3468 \cs_new:Npn \tl_set:Nn #1#2{
3469   \chk_exist_cs:N #1 \cs_set_nopar:Npn #1{#2}

```

We use \chk\_local\_or\_pref\_global:N after the assignment to allow constructs with \pref\_global\_chk:. But one should note that this is less efficient then using the real global variant since they are built-in.

```

3470   \chk_local_or_pref_global:N #1
3471 }
3472 \cs_new:Npn \tl_set:Nx #1#2{
3473   \chk_exist_cs:N #1 \cs_set_nopar:Npx #1{#2} \chk_local:N #1
3474 }

```

The the global versions.

```

3475 \cs_new:Npn \tl_gset:Nn #1#2{
3476   \chk_exist_cs:N #1 \cs_gset_nopar:Npn #1{#2} \chk_global:N #1
3477 }
3478 \cs_new:Npn \tl_gset:Nx #1#2{
3479   \chk_exist_cs:N #1 \cs_gset_nopar:Npx #1{#2} \chk_global:N #1
3480 }
3481 </check>

```

For some functions like \tl\_set:Nn we need to define the ‘non-check’ version with arguments since we want to allow constructions like \tl\_set:Nn\l\_tmpa\_tl\foo and so we can’t use the primitive  $\TeX$  command.

```

3482 <!*check>
3483 \cs_new:Npn \tl_set:Nn#1#2{\cs_set_nopar:Npn#1{#2}}
3484 \cs_new:Npn \tl_set:Nx#1#2{\cs_set_nopar:Npx#1{#2}}
3485 \cs_new:Npn \tl_gset:Nn#1#2{\cs_gset_nopar:Npn#1{#2}}
3486 \cs_new:Npn \tl_gset:Nx#1#2{\cs_gset_nopar:Npx#1{#2}}
3487 </!check>

```

The remaining functions can just be defined with help from the expansion module.

```

3488 \cs_generate_variant:Nn \tl_set:Nn {NV,No,Nv,Nf,cn,cV,co,cx}
3489 \cs_generate_variant:Nn \tl_gset:Nn {NV,No,Nv,cn,cx}

```

\ tl _ set _ eq : NN \ tl _ set _ eq : Nc \ tl _ set _ eq : cN \ tl _ set _ eq : cc \ tl _ gset _ eq : NN \ tl _ gset _ eq : Nc \ tl _ gset _ eq : cN \ tl _ gset _ eq : cc	For setting token list variables equal to each other. First checking: <pre> 3490 &lt;*check&gt; 3491 \cs_new_nopar:Npn \tl_set_eq:NN #1#2{ 3492   \chk_exist_cs:N #1 \cs_set_eq:NN #1#2 3493   \chk_local_or_pref_global:N #1 \chk_var_or_const:N #2 3494 } 3495 \cs_new_nopar:Npn \tl_gset_eq:NN #1#2{ 3496   \chk_exist_cs:N #1 \cs_gset_eq:NN #1#2 </pre>
--	--

```

3497 \chk_global:N #1 \chk_var_or_const:N #2
3498 }
3499 </check>

```

Non-checking versions are easy.

```

3500 <!*check>
3501 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
3502 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
3503 </!check>

```

The rest again with the expansion module.

```

3504 \cs_generate_variant:Nn \tl_set_eq:NN {Nc,c,cc}
3505 \cs_generate_variant:Nn \tl_gset_eq:NN {Nc,c,cc}

```

<pre> \tl_clear:N \tl_clear:c \tl_gclear:N \tl_gclear:c </pre>	<p>Clearing a token list variable.</p> <pre> 3506 \cs_new_nopar:Npn \tl_clear:N #1{\tl_set_eq:NN #1\c_empty_tl} 3507 \cs_generate_variant:Nn \tl_clear:N {c} 3508 \cs_new_nopar:Npn \tl_gclear:N #1{\tl_gset_eq:NN #1\c_empty_tl} 3509 \cs_generate_variant:Nn \tl_gclear:N {c} </pre>
--	--

<pre> \tl_clear_new:N \tl_clear_new:c </pre>	<p>These macros check whether a token list exists. If it does it is cleared, if it doesn't it is allocated.</p>
--	---

```

3510 <*check>
3511 \cs_new_nopar:Npn \tl_clear_new:N #1{
3512   \chk_var_or_const:N #1
3513   \if_predicate:w \cs_if_exist_p:N #1
3514     \tl_clear:N #1
3515   \else:
3516     \tl_new:Nn #1{}
3517   \fi:
3518 }
3519 </check>
3520 <-check>\cs_new_eq:NN \tl_clear_new:N \tl_clear:N
3521 \cs_generate_variant:Nn \tl_clear_new:N {c}

```

<pre> \tl_gclear_new:N \tl_gclear_new:c </pre>	<p>These are the global versions of the above.</p>
--	--

```

3522 <*check>
3523 \cs_new_nopar:Npn \tl_gclear_new:N #1{
3524   \chk_var_or_const:N #1
3525   \if_predicate:w \cs_if_exist_p:N #1
3526     \tl_gclear:N #1
3527   \else:
3528     \tl_new:Nn #1{}
3529   \fi:}
3530 </check>
3531 <-check>\cs_new_eq:NN \tl_gclear_new:N \tl_gclear:N
3532 \cs_generate_variant:Nn \tl_gclear_new:N {c}

```

$\backslash$ tl_put_left:Nn $\backslash$ tl_put_left:NV $\backslash$ tl_put_left:No $\backslash$ tl_put_left:Nx $\backslash$ tl_put_left:cn $\backslash$ tl_put_left:cV $\backslash$ tl_put_left:co $\backslash$ tl_gput_left:Nn $\backslash$ tl_gput_left:NV $\backslash$ tl_gput_left:No $\backslash$ tl_gput_left:Nx $\backslash$ tl_gput_left:cn $\backslash$ tl_gput_left:cV $\backslash$ tl_gput_left:co	<p>We can add tokens to the left (either globally or locally). It is not quite as easy as we would like because we have to ensure the assignments</p> <pre> \l_tpa_tl{##1abc##2def} \l_tpb_tl{##1abc} \l_tpb_tl{##2def} </pre> <p>cause <math>\backslash</math>l_tpa_tl and <math>\backslash</math>l_tpb_tl to be identical. The old code did not succeed in doing this (it gave an error) and so we use a different technique where the item(s) to be added are first stored in a temporary variable and then added using an x type expansion combined with the appropriate level of non-expansion. Putting the tokens directly into one assignment does not work unless we want full expansion. Note (according to the warning earlier) T<sub>E</sub>X does not allow us to treat #s the same in all cases. Tough.</p>
---	--

```

3533 \cs_new:Npn \tl_put_left:Nn #1#2 {
3534   \tl_set:Nn \l_exp_tl {#2}
3535   \tl_set:Nx #1 { \exp_not:V \l_exp_tl \exp_not:V #1 }
3536   <check> \chk_local_or_pref_global:N #1
3537 }
3538 \cs_new:Npn \tl_put_left:NV #1#2 {
3539   \tl_set:Nx #1 { \exp_not:V #2 \exp_not:V #1 }
3540 }
3541 \cs_new:Npn \tl_put_left:No #1#2{
3542   \tl_set:No \l_exp_tl {#2}
3543   \tl_set:Nx #1 { \exp_not:V \l_exp_tl \exp_not:V #1 }
3544   <check> \chk_local_or_pref_global:N #1
3545 }
3546 \cs_new:Npn \tl_put_left:Nx #1#2{
3547   \tl_set:Nx #1 { #2 \exp_not:V #1 }
3548   <check> \chk_local_or_pref_global:N #1
3549 }
3550 \cs_new:Npn \tl_gput_left:Nn #1#2{
3551   \tl_set:Nn \l_exp_tl{#2}
3552   \tl_gset:Nx #1 { \exp_not:V \l_exp_tl \exp_not:V #1 }
3553   <check> \chk_local_or_pref_global:N #1
3554 }
3555 \cs_new:Npn \tl_gput_left:NV #1#2 {
3556   \tl_gset:Nx #1 { \exp_not:V #2 \exp_not:V #1 }
3557 }
3558 \cs_new:Npn \tl_gput_left:No #1#2{
3559   \tl_set:No \l_exp_tl {#2}
3560   \tl_gset:Nx #1 { \exp_not:V \l_exp_tl \exp_not:V #1 }
3561   <check> \chk_local_or_pref_global:N #1
3562 }
3563 \cs_new:Npn \tl_gput_left:Nx #1#2{
3564   \tl_gset:Nx #1 { #2 \exp_not:V #1 }
3565   <check> \chk_local_or_pref_global:N #1
3566 }
3567 \cs_generate_variant:Nn \tl_put_left:Nn {cn,co,cV}

3568 \cs_generate_variant:Nn \tl_gput_left:Nn {cn,co}
3569 \cs_generate_variant:Nn \tl_gput_left:NV {cV}

```

`\tl_put_right:Nn` These are variants of the functions above, but for adding tokens to the right.  
`\tl_put_right:NV`  
`\tl_put_right:No`  
`\tl_put_right:Nx`  
`\tl_put_right:cn`  
`\tl_put_right:cV`  
`\tl_put_right:co`  
`\tl_gput_right:Nn`  
`\tl_gput_right:NV`  
`\tl_gput_right:No`  
`\tl_gput_right:Nx`  
`\tl_gput_right:cn`  
`\tl_gput_right:cV`  
`\tl_gput_right:co`

```

3570 \cs_new:Npn \tl_put_right:Nn #1#2 {
3571   \tl_set:Nn \l_exp_tl {#2}
3572   \tl_set:Nx #1 { \exp_not:V #1 \exp_not:V \l_exp_tl }
3573   <check> \chk_local_or_pref_global:N #1
3574 }
3575 \cs_new:Npn \tl_gput_right:Nn #1#2{
3576   \tl_set:Nn \l_exp_tl {#2}
3577   \tl_gset:Nx #1 { \exp_not:V #1 \exp_not:V \l_exp_tl }
3578   <check> \chk_local_or_pref_global:N #1
3579 }
3580 \cs_new:Npn \tl_put_right:NV #1#2 {
3581   \tl_set:Nx #1 { \exp_not:V #1 \exp_not:V #2 }
3582 }
3583 \cs_new:Npn \tl_put_right:No #1#2 {
3584   \tl_set:No \l_exp_tl {#2}
3585   \tl_set:Nx #1 { \exp_not:V #1 \exp_not:V \l_exp_tl }
3586   <check> \chk_local_or_pref_global:N #1
3587 }
3588 \cs_new:Npn \tl_gput_right:NV #1#2 {
3589   \tl_gset:Nx #1 { \exp_not:V #1 \exp_not:V #2 }
3590 }
3591 \cs_new:Npn \tl_gput_right:No #1#2 {
3592   \tl_set:No \l_exp_tl {#2}
3593   \tl_gset:Nx #1 { \exp_not:V #1 \exp_not:V \l_exp_tl }
3594   <check> \chk_local_or_pref_global:N #1
3595 }
3596 \cs_set:Npn \tl_put_right:Nx #1#2 {
3597   \tl_set:Nx #1 { \exp_not:V #1 #2 }
3598   <check> \chk_local_or_pref_global:N #1
3599 }
3600 \cs_set:Npn \tl_gput_right:Nx #1#2 {
3601   \tl_gset:Nx #1 { \exp_not:V #1 #2 }
3602   <check> \chk_local_or_pref_global:N #1
3603 }
3604 \cs_generate_variant:Nn \tl_put_right:Nn {cn,co}
3605 \cs_generate_variant:Nn \tl_put_right:NV {cV}
3606 \cs_generate_variant:Nn \tl_gput_right:Nn {cn,co,cV}

```

`\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.

```

3607 \cs_new_nopar:Npn \tl_gset:Nc {
3608   <*check>
3609   \pref_global_chk:
3610   </check>
3611   <-check> \pref_global:D
3612   \tl_set:Nc}

```

`\pref_global_chk:` will turn the variable check in `\tl_set:No` into a global check.



```
3613 \cs_new_nopar:Npn \tl_set:Nc #1#2{\tl_set:No #1{\cs:w#2\cs_end:}}
```

## 105.2 Variables and constants

`\c_job_name_tl` Inherited from the `expl3` name for the primitive.

```
3614 \tl_new:Nn \c_job_name_tl {\tex_jobname:D}
```

`\c_empty_tl` Two constants which are often used.

```
3615 \tl_new:Nn \c_empty_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.

```
3616 \tl_new:Nn \g_tmpa_tl {}
```

```
3617 \tl_new:Nn \g_tmpb_tl {}
```

`\l_testa_tl` Global and local temporaries. These are the ones for test routines. This means that one can safely use other temporaries when calling test routines.

`\l_testb_tl`

`\g_testa_tl`

`\g_testb_tl`

```
3618 \tl_new:Nn \l_testa_tl {}
```

```
3619 \tl_new:Nn \l_testb_tl {}
```

```
3620 \tl_new:Nn \g_testa_tl {}
```

```
3621 \tl_new:Nn \g_testb_tl {}
```

`\l_tmpa_tl` These are local temporary token list variables.

`\l_tmpb_tl`

```
3622 \tl_new:Nn \l_tmpa_tl {}
```

```
3623 \tl_new:Nn \l_tmpb_tl {}
```

## 105.3 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:NTF`

`\tl_if_empty:cTF`

```
3624 \prg_set_conditional:Npnn \tl_if_empty:N #1 {p,TF,T,F} {
```

```
3625   \if_meaning:w #1 \c_empty_tl
```

```
3626   \prg_return_true: \else: \prg_return_false: \fi:
```

```
3627 }
```

```
3628 \cs_generate_variant:Nn \tl_if_empty_p:N {c}
```

```
3629 \cs_generate_variant:Nn \tl_if_empty:N {c}
```

```
3630 \cs_generate_variant:Nn \tl_if_empty:NTF {c}
```

```
3631 \cs_generate_variant:Nn \tl_if_empty:NF {c}
```

\tl\_if\_eq\_p:NN Returns \c\_true\_bool iff the two token list variables are equal.

```

\tl_if_eq_p:Nc 3632 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 {p,TF,T,F} {
\tl_if_eq_p:cN 3633   \if_meaning:w #1 #2 \prg_return_true: \else: \prg_return_false: \fi:
\tl_if_eq_p:cc 3634 }
\tl_if_eq:NNTF 3635 \cs_generate_variant:Nn \tl_if_eq_p:NN {Nc,c,cc}
\tl_if_eq:NcTF 3636 \cs_generate_variant:Nn \tl_if_eq:NNTF {Nc,c,cc}
\tl_if_eq:cNTF 3637 \cs_generate_variant:Nn \tl_if_eq:NNT   {Nc,c,cc}
\tl_if_eq:ccTF 3638 \cs_generate_variant:Nn \tl_if_eq:NNF   {Nc,c,cc}

```

\tl\_if\_empty\_p:n It would be tempting to just use \if\_meaning:w\q\_nil#1\q\_nil as a test since this works really well. However it fails on a token list starting with \q\_nil of course but more troubling is the case where argument is a complete conditional such as \if\_true: a \else: b \fi: because then \if\_true: is used by \if\_meaning:w, the test turns out false, the \else: executes the false branch, the \fi: ends it and the \q\_nil at the end starts executing... A safer route is to convert the entire token list into harmless characters first and then compare that. This way the test will even accept \q\_nil as the first token.

```

3639 \prg_new_conditional:Npnn \tl_if_empty:n #1 {p,TF,T,F} {
3640   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
3641   \prg_return_true: \else: \prg_return_false: \fi:
3642 }
3643 \cs_generate_variant:Nn \tl_if_empty_p:n {V}
3644 \cs_generate_variant:Nn \tl_if_empty:nTF {V}
3645 \cs_generate_variant:Nn \tl_if_empty:nT   {V}
3646 \cs_generate_variant:Nn \tl_if_empty:nF   {V}
3647 \cs_generate_variant:Nn \tl_if_empty_p:n {o}
3648 \cs_generate_variant:Nn \tl_if_empty:nTF {o}
3649 \cs_generate_variant:Nn \tl_if_empty:nT   {o}
3650 \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 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”.

```

3651 \prg_new_conditional:Npnn \tl_if_blank:n #1 {p,TF,T,F} {
3652   \exp_after:wN \tl_if_blank_p_aux:w \tl_to_str:n {#1} aa..\q_nil
3653 }
3654 \cs_new:Npn \tl_if_blank_p_aux:w #1#2 a #3#4 \q_nil {
3655   \if_meaning:w #3 #4 \prg_return_true: \else: \prg_return_false: \fi:
3656 }
3657 \cs_generate_variant:Nn \tl_if_blank_p:n {V}
3658 \cs_generate_variant:Nn \tl_if_blank:nTF {V}
3659 \cs_generate_variant:Nn \tl_if_blank:nT   {V}
3660 \cs_generate_variant:Nn \tl_if_blank:nF   {V}
3661 \cs_generate_variant:Nn \tl_if_blank_p:n {o}
3662 \cs_generate_variant:Nn \tl_if_blank:nTF {o}
3663 \cs_generate_variant:Nn \tl_if_blank:nT   {o}
3664 \cs_generate_variant:Nn \tl_if_blank:nF   {o}

```

`\tl_if_eq:xxTF` Test if two token lists are identical. pdfTeX contains a most interesting primitive for expandable string comparison so we make use of it if available. Presumably it will be in the final version.

`\tl_if_eq:nnTF` Firstly we give it an appropriate name. Note that this primitive actually performs an x type expansion but it is still expandable! Hence we must program these functions backwards to add `\exp_not:n`. We provide the combinations for the types n, o and x.

```

\tl_if_eq:ooTF
\tl_if_eq:xnTF
\tl_if_eq:nxTF
\tl_if_eq:onTF
\tl_if_eq:noTF
3665 \cs_new_eq:NN \tl_compare:xx \pdf_strcmp:D
\tl_if_eq:VnTF
3666 \cs_new:Npn \tl_compare:nn #1#2{
\tl_if_eq:nVTF
3667 \tl_compare:xx{\exp_not:n{#1}}{\exp_not:n{#2}}
\tl_if_eq:xVTF
3668 }
\tl_if_eq:xoTF
3669 \cs_new:Npn \tl_compare:nx #1{
\tl_if_eq:VxTF
3670 \tl_compare:xx{\exp_not:n{#1}}
\tl_if_eq:oxTF
3671 }
\tl_if_eq_p:xx
3672 \cs_new:Npn \tl_compare:xn #1#2{
\tl_if_eq_p:nn
3673 \tl_compare:xx{#1}{\exp_not:n{#2}}
\tl_if_eq_p:VV
3674 }
\tl_if_eq_p:oo
3675 \cs_new:Npn \tl_compare:nV #1#2 {
\tl_if_eq_p:ox
3676 \tl_compare:xx { \exp_not:n {#1} } { \exp_not:V #2 }
\tl_if_eq_p:xn
3677 }
\tl_if_eq_p:nx
3678 \cs_new:Npn \tl_compare:no #1#2{
\tl_if_eq_p:Vn
3679 \tl_compare:xx{\exp_not:n{#1}}{\exp_not:n\exp_after:wN{#2}}
\tl_if_eq_p:on
3680 }
\tl_if_eq_p:nV
3681 \cs_new:Npn \tl_compare:Vn #1#2 {
\tl_if_eq_p:no
3682 \tl_compare:xx { \exp_not:V #1 } { \exp_not:n {#2} }
\tl_if_eq_p:xV
3683 }
\tl_if_eq_p:Vx
3684 \cs_new:Npn \tl_compare:on #1#2{
\tl_if_eq_p:ox
3685 \tl_compare:xx{\exp_not:n\exp_after:wN{#1}}{\exp_not:n{#2}}
\tl_if_eq_p:oo
3686 }
\tl_if_eq_p:ox
3687 \cs_new:Npn \tl_compare:VV #1#2 {
\tl_if_eq_p:ox
3688 \tl_compare:xx { \exp_not:V #1 } { \exp_not:V #2 }
\tl_if_eq_p:oo
3689 }
\tl_if_eq_p:oo
3690 \cs_new:Npn \tl_compare:oo #1#2{
\tl_if_eq_p:ox
3691 \tl_compare:xx{\exp_not:n\exp_after:wN{#1}}{\exp_not:n\exp_after:wN{#2}}
\tl_if_eq_p:ox
3692 }
\tl_if_eq_p:ox
3693 \cs_new:Npn \tl_compare:xV #1#2 {
\tl_if_eq_p:ox
3694 \tl_compare:xx {#1} { \exp_not:V #2 }
\tl_if_eq_p:ox
3695 }
\tl_if_eq_p:ox
3696 \cs_new:Npn \tl_compare:ox #1#2{
\tl_if_eq_p:ox
3697 \tl_compare:xx{#1}{\exp_not:n\exp_after:wN{#2}}
\tl_if_eq_p:ox
3698 }
\tl_if_eq_p:ox
3699 \cs_new:Npn \tl_compare:Vx #1#2 {
\tl_if_eq_p:ox
3700 \tl_compare:xx { \exp_not:V #1 } {#2}
\tl_if_eq_p:ox
3701 }
\tl_if_eq_p:ox
3702 \cs_new:Npn \tl_compare:ox #1#2{
\tl_if_eq_p:ox
3703 \tl_compare:xx{\exp_not:n\exp_after:wN{#1}}{#2}
\tl_if_eq_p:ox
3704 }

```

Since we have a lot of basically identical functions to define we define one to define the rest. Unfortunately we aren't quite set up to use the new `\tl_map_inline:nn` function yet.

```

3705 \cs_set_nopar:Npn \tl_tmp:w #1 {
3706 \tl_set:Nx \l_tmpa_tl {

```

```

3707 \exp_not:N \prg_new_conditional:Npnn \exp_not:c {tl_if_eq:#1}
3708 #####1 #####2 {p,TF,T,F} {
3709   \exp_not:N \tex_ifnum:D
3710   \exp_not:c {tl_compare:#1} {#####1}{#####2}
3711   \exp_not:n{=\c_zero \prg_return_true: \else: \prg_return_false: \fi: }
3712 }
3713 }
3714 \l_tmpa_tl
3715 }
3716 \tl_tmp:w{xx} \tl_tmp:w{nx} \tl_tmp:w{ox} \tl_tmp:w{Vx}
3717 \tl_tmp:w{xn} \tl_tmp:w{nn} \tl_tmp:w{on} \tl_tmp:w{Vn}
3718 \tl_tmp:w{xox} \tl_tmp:w{no} \tl_tmp:w{oo}
3719 \tl_tmp:w{xV} \tl_tmp:w{nV} \tl_tmp:w{VV}

```

However all of this only makes sense if we actually have that primitive. Therefore we disable it again if it is not there and define `\tl_if_eq:nn` the old fashioned (and unexpandable) way.

In some cases below, since arbitrary token lists could be being used in this function, you can't assume (as token list variables usually do) that there won't be any `#` tokens. Therefore, `\tl_set:Nx` and `\exp_not:n` is used instead of plain `\tl_set:Nn`.

```

3720 \cs_if_exist:cF{pdf_strcmp:D}{
3721   \prg_set_protected_conditional:Npnn \tl_if_eq:nn #1#2 {TF,T,F} {
3722     \tl_set:Nx \l_testa_tl {\exp_not:n{#1}}
3723     \tl_set:Nx \l_testb_tl {\exp_not:n{#2}}
3724     \if_meaning:w\l_testa_tl \l_testb_tl
3725     \prg_return_true: \else: \prg_return_false:
3726     \fi:
3727   }
3728   \prg_set_protected_conditional:Npnn \tl_if_eq:nV #1#2 {TF,T,F} {
3729     \tl_set:Nx \l_testa_tl { \exp_not:n {#1} }
3730     \tl_set:Nx \l_testb_tl { \exp_not:V #2 }
3731     \if_meaning:w \l_testa_tl \l_testb_tl
3732     \prg_return_true: \else: \prg_return_false:
3733     \fi:
3734   }
3735   \prg_set_protected_conditional:Npnn \tl_if_eq:no #1#2 {TF,T,F} {
3736     \tl_set:Nx \l_testa_tl {\exp_not:n{#1}}
3737     \tl_set:Nx \l_testb_tl {\exp_not:o{#2}}
3738     \if_meaning:w\l_testa_tl \l_testb_tl
3739     \prg_return_true: \else: \prg_return_false:
3740     \fi:
3741   }
3742   \prg_set_protected_conditional:Npnn \tl_if_eq:nx #1#2 {TF,T,F} {
3743     \tl_set:Nx \l_testa_tl {\exp_not:n{#1}}
3744     \tl_set:Nx \l_testb_tl {#2}
3745     \if_meaning:w\l_testa_tl \l_testb_tl
3746     \prg_return_true: \else: \prg_return_false:
3747     \fi:
3748   }
3749   \prg_set_protected_conditional:Npnn \tl_if_eq:Vn #1#2 {TF,T,F} {
3750     \tl_set:Nx \l_testa_tl { \exp_not:V #1 }
3751     \tl_set:Nx \l_testb_tl { \exp_not:n{#2} }
3752     \if_meaning:w \l_testa_tl \l_testb_tl

```

```

3753     \prg_return_true: \else: \prg_return_false:
3754     \fi:
3755 }
3756 \prg_set_protected_conditional:Npnn \tl_if_eq:on #1#2 {TF,T,F} {
3757     \tl_set:Nx \l_testa_tl {\exp_not:o{#1}}
3758     \tl_set:Nx \l_testb_tl {\exp_not:n{#2}}
3759     \if_meaning:w\l_testa_tl \l_testb_tl
3760     \prg_return_true: \else: \prg_return_false:
3761     \fi:
3762 }
3763 \prg_set_protected_conditional:Npnn \tl_if_eq:VV #1#2 {TF,T,F} {
3764     \tl_set:Nx \l_testa_tl { \exp_not:V #1 }
3765     \tl_set:Nx \l_testb_tl { \exp_not:V #2 }
3766     \if_meaning:w \l_testa_tl \l_testb_tl
3767     \prg_return_true: \else: \prg_return_false:
3768     \fi:
3769 }
3770 \prg_set_protected_conditional:Npnn \tl_if_eq:oo #1#2 {TF,T,F} {
3771     \tl_set:Nx \l_testa_tl {\exp_not:o{#1}}
3772     \tl_set:Nx \l_testb_tl {\exp_not:o{#2}}
3773     \if_meaning:w\l_testa_tl \l_testb_tl
3774     \prg_return_true: \else: \prg_return_false:
3775     \fi:
3776 }
3777 \prg_set_protected_conditional:Npnn \tl_if_eq:Vx #1#2 {TF,T,F} {
3778     \tl_set:Nx \l_testa_tl { \exp_not:V #1 }
3779     \tl_set:Nx \l_testb_tl {#2}
3780     \if_meaning:w \l_testa_tl \l_testb_tl
3781     \prg_return_true: \else: \prg_return_false:
3782     \fi:
3783 }
3784 \prg_set_protected_conditional:Npnn \tl_if_eq:ox #1#2 {TF,T,F} {
3785     \tl_set:Nx \l_testa_tl {\exp_not:o{#1}}
3786     \tl_set:Nx \l_testb_tl {#2}
3787     \if_meaning:w\l_testa_tl \l_testb_tl
3788     \prg_return_true: \else: \prg_return_false:
3789     \fi:
3790 }
3791 \prg_set_protected_conditional:Npnn \tl_if_eq:xn #1#2 {TF,T,F} {
3792     \tl_set:Nx \l_testa_tl {#1}
3793     \tl_set:Nx \l_testb_tl {\exp_not:n{#2}}
3794     \if_meaning:w\l_testa_tl \l_testb_tl
3795     \prg_return_true: \else: \prg_return_false:
3796     \fi:
3797 }
3798 \prg_set_protected_conditional:Npnn \tl_if_eq:xV #1#2 {TF,T,F} {
3799     \tl_set:Nx \l_testa_tl {#1}
3800     \tl_set:Nx \l_testb_tl { \exp_not:V #2 }
3801     \if_meaning:w \l_testa_tl \l_testb_tl
3802     \prg_return_true: \else: \prg_return_false:
3803     \fi:
3804 }
3805 \prg_set_protected_conditional:Npnn \tl_if_eq:xo #1#2 {TF,T,F} {
3806     \tl_set:Nx \l_testa_tl {#1}

```

```

3807 \tl_set:Nx \l_testb_tl {\exp_not:o{#2}}
3808 \if_meaning:w\l_testa_tl \l_testb_tl
3809 \prg_return_true: \else: \prg_return_false:
3810 \fi:
3811 }
3812 \prg_set_protected_conditional:Npnn \tl_if_eq:xx #1#2 {TF,T,F} {
3813 \tl_set:Nx \l_testa_tl {#1}
3814 \tl_set:Nx \l_testb_tl {#2}
3815 \if_meaning:w\l_testa_tl \l_testb_tl
3816 \prg_return_true: \else: \prg_return_false:
3817 \fi:
3818 }
3819 }

```

## 105.4 Working with the contents of token lists

`\tl_to_lowercase:n` Just some names for a few primitives.

```

\tl_to_uppercase:n
3820 \cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
3821 \cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D

```

`\tl_to_str:n` Another name for a primitive.

```

3822 \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
3823 \cs_new_nopar:Npn \tl_to_str:N {\exp_after:wN\tl_to_str_aux:w
3824 \token_to_meaning:N}
3825 \cs_new_nopar:Npn \tl_to_str_aux:w #1>{\
3826 \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
3827 \cs_new:Npn \tl_map_function:nN #1#2{
3828 \tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop
3829 }
3830 \cs_new_nopar:Npn \tl_map_function:NN #1#2{
3831 \exp_after:wN \tl_map_function_aux:Nn
3832 \exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop
3833 }
3834 \cs_new:Npn \tl_map_function_aux:Nn #1#2{
3835 \quark_if_recursion_tail_stop:n{#2}
3836 #1{#2} \tl_map_function_aux:Nn #1
3837 }
3838 \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 fake counter `\g_tl_inline_level_num` to make them nestable.<sup>12</sup> We can also make use of `\tl_map_function:Nn` from before.

```

\tl_map_inline:aux:n
\g_tl_inline_level_num
3839 \cs_new:Npn \tl_map_inline:nn #1#2{
3840   \num_gincr:N \g_tl_inline_level_num
3841   \cs_gset:cpn {tl_map_inline_ \num_use:N \g_tl_inline_level_num :n}
3842   ##1{#2}
3843   \exp_args:Nc \tl_map_function_aux:Nn
3844   {tl_map_inline_ \num_use:N \g_tl_inline_level_num :n}
3845   #1 \q_recursion_tail\q_recursion_stop
3846   \num_gdecr:N \g_tl_inline_level_num
3847 }
3848 \cs_new:Npn \tl_map_inline:Nn #1#2{
3849   \num_gincr:N \g_tl_inline_level_num
3850   \cs_gset:cpn {tl_map_inline_ \num_use:N \g_tl_inline_level_num :n}
3851   ##1{#2}
3852   \exp_last_unbraced:NcV \tl_map_function_aux:Nn
3853   {tl_map_inline_ \num_use:N \g_tl_inline_level_num :n}
3854   #1 \q_recursion_tail\q_recursion_stop
3855   \num_gdecr:N \g_tl_inline_level_num
3856 }
3857 \cs_generate_variant:Nn \tl_map_inline:Nn {c}
3858 \tl_new:Nn \g_tl_inline_level_num{0}

```

`\tl_map_variable:nNn` `\tl_map_variable:nNn`  $\langle token\ list \rangle$   $\langle temp \rangle$   $\langle action \rangle$  assigns  $\langle temp \rangle$  to each element and executes  $\langle action \rangle$ .

```

\tl_map_variable:NNn
\tl_map_variable:cNn
3859 \cs_new:Npn \tl_map_variable:nNn #1#2#3{
3860   \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop
3861 }

```

Next really has to be v/V args

```

3862 \cs_new_nopar:Npn \tl_map_variable:NNn {\exp_args:No \tl_map_variable:nNn}
3863 \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.

```

3864 \cs_new:Npn \tl_map_variable_aux:Nnn #1#2#3{
3865   \tl_set:Nn #1{#3}
3866   \quark_if_recursion_tail_stop:N #1
3867   #2 \tl_map_variable_aux:Nnn #1{#2}
3868 }

```

`\tl_map_break:` The break statement.

```

3869 \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` <sup>12</sup>This should be a proper integer, but I don't want to mess with the dependencies right now...

```

3870 \cs_new:Npn \tl_reverse:n #1{
3871   \tl_reverse_aux:nN {} #1 \q_recursion_tail\q_recursion_stop
3872 }
3873 \cs_new:Npn \tl_reverse_aux:nN #1#2{
3874   \quark_if_recursion_tail_stop_do:nn {#2}{#1 }
3875   \tl_reverse_aux:nN {#2#1}
3876 }
3877 \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.

```

3878 \cs_new_nopar:Npn \tl_reverse:N #1 {
3879   \tl_set:Nf #1 { \tl_reverse:o { #1 \exp_stop_f: } }
3880 }

```

`\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. `\num_elt_count: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`

```

3881 \cs_new:Npn \tl_elt_count:n #1{
3882   \intexpr_eval:n {
3883     \tl_map_function:nN {#1}\num_elt_count:n
3884   }
3885 }
3886 \cs_generate_variant:Nn \tl_elt_count:n {V,o}
3887 \cs_new_nopar:Npn \tl_elt_count:N #1{
3888   \intexpr_eval:n {
3889     \tl_map_function:NN #1 \num_elt_count:n
3890   }
3891 }

```

`\tl_set_rescan:Nnn` These functions store the `{\langle token list \rangle}` in `\langle tl var. \rangle` after redefining catcodes, etc., in argument #2.

`\tl_gset_rescan:Nnn`

```

#1 : \langle tl var. \rangle
#2 : {\langle catcode setup, etc. \rangle}
#3 : {\langle token list \rangle}

```

```

3892 \cs_new:Npn \tl_set_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_set:Nn }
3893 \cs_new:Npn \tl_gset_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_gset:Nn }

```

`\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.

`_two_ats_with_two_catcodes_tl`

A tl with two @ characters with two different catcodes. Used as a special marker for delimited text.

```

3894 \group_begin:
3895 \tex_lccode:D '\A = '\@ \scan_stop:
3896 \tex_lccode:D '\B = '\@ \scan_stop:
3897 \tex_catcode:D '\A = 8 \scan_stop:
3898 \tex_catcode:D '\B = 3 \scan_stop:
3899 \tl_to_lowercase:n {
3900 \group_end:
3901 \tl_new:Nn \c_two_ats_with_two_catcodes_tl {AB}
3902 }

```

#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!

```

3903 \cs_new:Npn \tl_set_rescan_aux:NNnn #1#2#3#4 {
3904 \group_begin:
3905 \toks_set:NV \etex_everyeof:D \c_two_ats_with_two_catcodes_tl
3906 \tex_endlinechar:D = \c_minus_one
3907 #3
3908 \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
3909 \exp_args:NNNV
3910 \group_end:
3911 #1 #2 \l_tmpa_toks
3912 }

```

`\tl_rescan_aux:w`

```

3913 \exp_after:wN \cs_set:Npn
3914 \exp_after:wN \tl_rescan_aux:w
3915 \exp_after:wN #
3916 \exp_after:wN 1 \c_two_ats_with_two_catcodes_tl {
3917 \tl_set:Nn \l_tmpa_toks {#1}
3918 }

```

`\tl_set_rescan:Nnx` These functions store the full expansion of `{<token list>}` in `<tl var.>` after redefining  
`\tl_gset_rescan:Nnx` catcodes, etc., in argument #2.

```
#1 : <tl var.>
#2 : {\catcode setup, etc.}
#3 : {\token list}
```

The expanded versions are much simpler because the `\etex_scantokens:D` can occur within the expansion.

```
3919 \cs_new:Npn \tl_set_rescan:Nnx #1#2#3 {
3920   \group_begin:
3921     \etex_everyeof:D { \exp_not:N }
3922     \tex_endlinechar:D = \c_minus_one
3923     #2
3924     \tl_set:Nx \l_tmpa_tl { \etex_scantokens:D {#3} }
3925     \exp_args:NNNV
3926   \group_end:
3927   \tl_set:Nn #1 \l_tmpa_tl
3928 }
```

Globally is easier again:

```
3929 \cs_new:Npn \tl_gset_rescan:Nnx #1#2#3 {
3930   \group_begin:
3931     \etex_everyeof:D { \exp_not:N }
3932     \tex_endlinechar:D = \c_minus_one
3933     #2
3934     \tl_gset:Nx #1 { \etex_scantokens:D {#3} }
3935   \group_end:
3936 }
```

`\tl_rescan:nn` The inline wrapper for `\etex_scantokens:D`.

```
#1 : Catcode changes (etc.)
#2 : Token list to re-tokenise
```

```
3937 \cs_new:Npn \tl_rescan:nn #1#2 {
3938   \group_begin:
3939     \toks_set:NV \etex_everyeof:D \c_two_at_with_two_catcodes_tl
3940     \tex_endlinechar:D = \c_minus_one
3941     #1
3942     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
3943   \exp_args:NV \group_end:
3944   \l_tmpa_toks
3945 }
```

## 105.5 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.

```
3946 \prg_new_protected_conditional:Npnn \tl_if_in:Nn #1#2 {TF,T,F} {
3947   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
3948     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
```

```

3949 }
3950 \exp_after:wN \tl_tmp:w #1 #2 \q_no_value \q_stop
3951 }
3952 \cs_generate_variant:Nn \tl_if_in:NnTF {c}
3953 \cs_generate_variant:Nn \tl_if_in:NnT {c}
3954 \cs_generate_variant:Nn \tl_if_in:NnF {c}

\tl_if_in:nnTF
\tl_if_in:VnTF
\tl_if_in:onTF 3955 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 {TF,T,F} {
3956   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
3957     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
3958   }
3959   \tl_tmp:w #1 #2 \q_no_value \q_stop
3960 }
3961 \cs_generate_variant:Nn \tl_if_in:nnTF {V}
3962 \cs_generate_variant:Nn \tl_if_in:nnT {V}
3963 \cs_generate_variant:Nn \tl_if_in:nnF {V}
3964 \cs_generate_variant:Nn \tl_if_in:nnTF {o}
3965 \cs_generate_variant:Nn \tl_if_in:nnT {o}
3966 \cs_generate_variant:Nn \tl_if_in:nnF {o}

```

`\l_tl_replace_toks` A temp variable for the replace operations.

```

3967 %%\l_tl_replace_toks % moved to l3token as alloc not set up yet.

```

`\tl_replace_in:Nnn` Replacing the first item in a token list variable goes like this: Define a temporary function with delimited arguments containing the search term and take a closer look at what is left. We append the expansion of the token list with the search term plus the quark `\q_no_value`. If the search term isn't present this last one is found and the following token is the quark, so we test for that. If the search term is present we will have to split off the `#3\q_no_value` we had, so we define yet another function with delimited arguments to do this. The advantage here is that now we have a special end sequence so there is no problem if the search term appears more than once. Only problem left is to prevent brace stripping in both ends, so we prepend the expansion of the token list with `\prg_do_nothing:` later to be expanded and also prepend the remainder of the first split operation with `\prg_do_nothing:` also to be expanded again later on.

`\tl_replace_in:cnn`

`\tl_greplace_in:Nnn`

`\tl_greplace_in:cnn`

```

\tl_replace_in_aux:NNnn #1 : \tl_set:Nx or \tl_gset:Nx
#2 : <tl var.>
#3 : item to find
#4 : replacement text

3968 \cs_new:Npn \tl_replace_in_aux:NNnn #1#2#3#4{
3969   \cs_set:Npn \tl_tmp:w ##1#3##2\q_stop{
3970     \quark_if_no_value:nF{##2}
3971     {

```

At this point `##1` starts with a `\prg_do_nothing:` so we expand it to remove it.

```

3972   \toks_set:No\l_tl_replace_toks{##1#4}
3973   \cs_set:Npn \tl_tmp:w ####1#3\q_no_value{

```

```

3974         \toks_put_right:No \l_tl_replace_toks { ###1 }
3975     }
3976     \tl_tmp:w \prg_do_nothing: ##2

```

Now all that is done is setting the token list variable equal to the expansion of the token register.

```

3977         #1#2{\toks_use:N\l_tl_replace_toks}
3978     }
3979 }

```

Here is where we start the process. Note that the token list might start with a space token so we use this little trick with \use:n to prevent it from being removed.

```

3980 \use:n{\exp_after:wN \tl_tmp:w\exp_after:wN\prg_do_nothing:}
3981 #2#3 \q_no_value\q_stop
3982 }

```

Now the various versions doing the replacement either globally or locally.

```

3983 \cs_new_nopar:Npn \tl_replace_in:Nnn {\tl_replace_in_aux:NNnn \tl_set:Nx}
3984 \cs_generate_variant:Nn\tl_replace_in:Nnn {cnn}

3985 \cs_new_nopar:Npn \tl_greplace_in:Nnn {\tl_replace_in_aux:NNnn \tl_gset:Nx}
3986 \cs_generate_variant:Nn\tl_greplace_in:Nnn {cnn}

```

```

\tl_replace_all_in:Nnn
\tl_replace_all_in:cnn
\tl_greplace_all_in:Nnn
\tl_greplace_all_in:cnn
\tl_replace_all_in_aux:NNnn

```

The version for replacing *all* occurrences of the search term is fairly easy since we just have to keep doing the replacement on the split-off part until all are replaced. Otherwise it is pretty much the same as above.

```

3987 \cs_set:Npn \tl_replace_all_in_aux:NNnn #1#2#3#4{
3988     \toks_clear:N \l_tl_replace_toks
3989     \cs_set:Npn \tl_tmp:w ##1#3##2\q_stop{
3990         \quark_if_no_value:nTF{##2}
3991         {
3992             \toks_put_right:No \l_tl_replace_toks {##1}
3993         }
3994         {
3995             \toks_put_right:No \l_tl_replace_toks {##1 #4}
3996             \tl_tmp:w \prg_do_nothing: ##2 \q_stop
3997         }
3998     }
3999     \use:n{\exp_after:wN \tl_tmp:w \exp_after:wN \prg_do_nothing:}
4000     #2#3 \q_no_value\q_stop
4001     #1#2{\toks_use:N\l_tl_replace_toks}
4002 }

```

Now the various forms.

```

4003 \cs_new_nopar:Npn \tl_replace_all_in:Nnn {
4004     \tl_replace_all_in_aux:NNnn \tl_set:Nx}
4005 \cs_generate_variant:Nn \tl_replace_all_in:Nnn {cnn}

4006 \cs_new_nopar:Npn \tl_greplace_all_in:Nnn {
4007     \tl_replace_all_in_aux:NNnn \tl_gset:Nx}
4008 \cs_generate_variant:Nn \tl_greplace_all_in:Nnn {cnn}

```

<pre> \tl_remove_in:Nn \tl_remove_in:cn \tl_gremove_in:Nn \tl_gremove_in:cn </pre>	<p>Next comes a series of removal functions. I have just implemented them as subcases of the replace functions for now (I'm lazy).</p> <pre> 4009 \cs_new:Npn \tl_remove_in:Nn #1#2{\tl_replace_in:Nnn #1{#2}{}} 4010 \cs_new:Npn \tl_gremove_in:Nn #1#2{\tl_greplace_in:Nnn #1{#2}{}} 4011 \cs_generate_variant:Nn \tl_remove_in:Nn {cn} 4012 \cs_generate_variant:Nn \tl_gremove_in:Nn {cn} </pre>
<pre> \tl_remove_all_in:Nn \tl_remove_all_in:cn \tl_gremove_all_in:Nn \tl_gremove_all_in:cn </pre>	<p>Same old, same old.</p> <pre> 4013 \cs_new:Npn \tl_remove_all_in:Nn #1#2{ 4014   \tl_replace_all_in:Nnn #1{#2}{}} 4015 } 4016 \cs_new:Npn \tl_gremove_all_in:Nn #1#2{ 4017   \tl_greplace_all_in:Nnn #1{#2}{}} 4018 } 4019 \cs_generate_variant:Nn \tl_remove_all_in:Nn {cn} 4020 \cs_generate_variant:Nn \tl_gremove_all_in:Nn {cn} </pre>

## 105.6 Heads or tails?

<pre> \tl_head:n \tl_head_i:n \tl_tail:n \tl_tail:f \tl_head_iii:n \tl_head_iii:f \tl_head:w \tl_head_i:w \tl_tail:w \tl_head_iii:w </pre>	<p>These functions pick up either the head or the tail of a list. <code>\tl_head_iii:n</code> returns the first three items on a list.</p> <pre> 4021 \cs_new:Npn \tl_head:n #1{\tl_head:w #1\q_nil} 4022 \cs_new_eq:NN \tl_head_i:n \tl_head:n 4023 \cs_new:Npn \tl_tail:n #1{\tl_tail:w #1\q_nil} 4024 \cs_generate_variant:Nn \tl_tail:n {f} 4025 \cs_new:Npn \tl_head_iii:n #1{\tl_head_iii:w #1\q_nil} 4026 \cs_generate_variant:Nn \tl_head_iii:n {f} 4027 \cs_new_eq:NN \tl_head:w \use_i_delimit_by_q_nil:nw 4028 \cs_new_eq:NN \tl_head_i:w \tl_head:w 4029 \cs_new:Npn \tl_tail:w #1#2\q_nil{#2} 4030 \cs_new:Npn \tl_head_iii:w #1#2#3#4\q_nil{#1#2#3} </pre>
--	--

<pre> \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 </pre>	<p>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. <code>\tl_if_head_meaning_eq:nNTF</code> uses <code>\if_meaning:w</code> and will consider the tokens <code>b<sub>11</sub></code> and <code>b<sub>12</sub></code> different. <code>\tl_if_head_char_eq:nNTF</code> on the other hand only compares character codes so would regard <code>b<sub>11</sub></code> and <code>b<sub>12</sub></code> as equal but would also regard two primitives as equal.</p> <pre> 4031 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 {p,TF,T,F} { 4032   \exp_after:wN \if_meaning:w \tl_head:w #1 \q_nil #2 4033   \prg_return_true: \else: \prg_return_false: \fi: 4034 } </pre>
--	---

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.

```

4035 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 {p,TF,T,F} {

```

```

4036 \exp_after:wN \if:w \exp_after:wN \exp_not:N
4037 \tl_head:w #1 \q_nil \exp_not:N #2
4038 \prg_return_true: \else: \prg_return_false: \fi:
4039 }

```

Actually the default is already an f type expansion.

```

4040 %% \cs_new:Npn \tl_if_head_eq_charcode_p:fN #1#2{
4041 %% \exp_after:wN \if_charcode:w \tl_head:w #1 \q_nil \exp_not:N #2
4042 %% \c_true_bool
4043 %% \else:
4044 %% \c_false_bool
4045 %% \fi:
4046 %% }
4047 %% \def_long_test_function_new:npn {tl_if_head_eq_charcode:fN}#1#2{
4048 %% \if_predicate:w \tl_if_head_eq_charcode_p:fN {#1}#2}

```

These :fN variants are broken; temporary patch:

```

4049 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN {f}
4050 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF {f}
4051 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT {f}
4052 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF {f}

```

And now catcodes:

```

4053 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1#2 {p,TF,T,F} {
4054 \exp_after:wN \if_catcode:w \exp_after:wN \exp_not:N
4055 \tl_head:w #1 \q_nil \exp_not:N #2
4056 \prg_return_true: \else: \prg_return_false: \fi:
4057 }

```

Show token usage:

```

4058 <*showmemory>
4059 \showMemUsage
4060 </showmemory>

```

## 106 l3toks implementation

We start by ensuring that the required packages are loaded.

```

4061 <*package>
4062 \ProvidesExplPackage
4063 {\filename}{\filedate}{\fileversion}{\filedescription}
4064 \package_check_loaded_expl:
4065 </package>
4066 <*initex | package>

```

## 106.1 Allocation and use

`\toks_new:N` Allocates a new token register.

```
\toks_new:c
4067 <*initex>
4068 \alloc_setup_type:nnn {toks} \c_zero \c_max_register_num
4069 \cs_new_nopar:Npn \toks_new:N #1 { \alloc_reg:NnNN g {toks} \tex_toksdef:D #1 }
4070 \cs_new_nopar:Npn \toks_new_l:N #1 { \alloc_reg:NnNN l {toks} \tex_toksdef:D #1 }
4071 </initex>
4072 <package>\cs_set_eq:NN \toks_new:N \newtoks % nick from LaTeX for the moment
4073 \cs_generate_variant:Nn \toks_new:N {c}
```

`\toks_use:N` This function returns the contents of a token register.

```
\toks_use:c
4074 \cs_new_eq:NN \toks_use:N \tex_the:D
4075 \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` `\toks_set:Nx` expand `<stuff>` once and fully.

```
\toks_set:Nv
4076 <*check>
4077 \cs_new_nopar:Npn \toks_set:Nn #1 { \chk_local:N #1 #1 }
4078 \cs_generate_variant:Nn \toks_set:Nn {No,Nf}
4079 </check>
\toks_set:cn
```

`\toks_set:co` If we don't check if `<toks>` is a local register then the `\toks_set:Nn` function has nothing to do. We implement `\toks_set:No/d/f` by hand when not checking because this is going to be used *extensively* in keyval processing! TODO: (Will) Can we get some numbers published on how necessary this is? On the other hand I'm happy to believe Morten :)

```
\toks_set:cV
\toks_set:cv
\toks_set:cx
\toks_set:cf
4080 <!*check>
4081 \cs_new_eq:NN \toks_set:Nn \prg_do_nothing:
4082 \cs_new:Npn \toks_set:Nv #1#2 {
4083 #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:N #2 }
4084 }
4085 \cs_new:Npn \toks_set:Nv #1#2 {
4086 #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:c {#2} }
4087 }
4088 \cs_new:Npn \toks_set:No #1#2 { #1 \exp_after:wN {#2} }
4089 \cs_new:Npn \toks_set:Nf #1#2 {
4090 #1 \exp_after:wN { \int_to_roman:w -'0#2 }
4091 }
4092 </!check>
4093 \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
4094 <check>\cs_new_nopar:Npn \toks_gset:Nn #1 { \chk_global:N #1 \pref_global:D #1 }
4095 <!check>\cs_new_eq:NN \toks_gset:Nn \pref_global:D
4096 \cs_generate_variant:Nn \toks_gset:Nn {NV,No,Nx,cn,cV,co,cx}
\toks_gset:cn
\toks_gset:cV
\toks_gset:co
\toks_gset:cx
```

\toks\_set\_eq:NN \toks\_set\_eq:NN<toks1><toks2> copies the contents of <toks2> in <toks1>.

```

\toks_set_eq:Nc
\toks_set_eq:cN
\toks_set_eq:cc
\toks_gset_eq:NN
\toks_gset_eq:Nc
\toks_gset_eq:cN
\toks_gset_eq:cc
4097 <*check>
4098 \cs_new_nopar:Npn\toks_set_eq:NN #1#2 {
4099   \chk_local:N #1
4100   \chk_var_or_const:N #2
4101   #1 #2
4102 }
4103 \cs_new_nopar:Npn\toks_gset_eq:NN #1#2 {
4104   \chk_global:N #1
4105   \chk_var_or_const:N #2
4106   \pref_global:D #1 #2
4107 }
4108 </check>
4109 <!*check>
4110 \cs_new_eq:NN \toks_set_eq:NN \prg_do_nothing:
4111 \cs_new_eq:NN \toks_gset_eq:NN \pref_global:D
4112 </!check>
4113 \cs_generate_variant:Nn \toks_set_eq:NN {Nc,cN,cc}
4114 \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
4115 \cs_new_nopar:Npn \toks_clear:N #1 {
4116   #1\c_empty_toks
4117 <check>\chk_local_or_pref_global:N #1
4118 }
4119 \cs_new_nopar:Npn \toks_gclear:N {
4120 <check> \pref_global_chk:
4121 <!check> \pref_global:D
4122   \toks_clear:N
4123 }
4124 \cs_generate_variant:Nn \toks_clear:N {c}
4125 \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.

```

4126 \cs_new_nopar:Npn \toks_use_clear:N #1 {
4127   \exp_last_unbraced:NNV \toks_clear:N #1 #1
4128 }
4129 \cs_new_nopar:Npn \toks_use_gclear:N {
4130 <check> \pref_global_chk:
4131 <!check> \pref_global:D
4132   \toks_use_clear:N
4133 }
4134 \cs_generate_variant:Nn \toks_use_clear:N {c}
4135 \cs_generate_variant:Nn \toks_use_gclear:N {c}

```



`\toks_show:N` This function shows the contents of a token register on the terminal. TODO: this is not pretty when the argument is a control sequence that doesn't exist!

```
4136 \cs_new_eq:NN \toks_show:N \tex_showthe:D
4137 \cs_generate_variant:Nn \toks_show:N {c}
```

## 106.2 Adding to token registers' contents

`\toks_put_left:Nn` `\toks_put_left:Nn <toks><stuff>` adds the tokens of *stuff* on the 'left-side' of the token register *<toks>*. `\toks_put_left:No` does the same, but expands the tokens once. We need to look out for brace stripping so we add a token, which is then later removed.

```

\toks_put_left:Nn 4138 \cs_new_nopar:Npn \toks_put_left:Nn #1 {
\toks_put_left:Nv 4139 \exp_after:wN \toks_put_left_aux:w \exp_after:wN \q_mark
\toks_put_left:No 4140 \toks_use:N #1 \q_stop #1
\toks_put_left:Nx 4141 }
\toks_gput_left:Nn 4142 \cs_generate_variant:Nn \toks_put_left:Nn {NV,No,Nx,cn,co,cV}
\toks_gput_left:Nv 4143 \cs_new_nopar:Npn \toks_gput_left:Nn {
\toks_gput_left:No 4144 <check> \pref_global_chk:
\toks_gput_left:Nx 4145 <!check> \pref_global:D
\toks_gput_left:cn 4146 \toks_put_left:Nn
\toks_gput_left:cV 4147 }
\toks_put_left_aux:w 4148 \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.

```
4149 \cs_new:Npn \toks_put_left_aux:w #1\q_stop #2#3 {
4150 #2 \exp_after:wN { \use_i:nn {#3} #1 }
4151 <check> \chk_local_or_pref_global:N #2
4152 }
```

`\toks_put_right:Nn` These macros add a list of tokens to the right of a token register.

```

\toks_put_right:Nn 4153 \cs_new:Npn \toks_put_right:Nn #1#2 {
\toks_put_right:Nv 4154 #1 \exp_after:wN { \toks_use:N #1 #2 }
\toks_put_right:No 4155 <check> \chk_local_or_pref_global:N #1
\toks_put_right:Nx 4156 }
\toks_gput_right:Nn 4157 \cs_new_nopar:Npn \toks_gput_right:Nn {
\toks_gput_right:Nv 4158 <check> \pref_global_chk:
\toks_gput_right:Nv 4159 <!check> \pref_global:D
\toks_gput_right:No 4160 \toks_put_right:Nn
\toks_gput_right:Nx 4161 }
\toks_gput_right:cn 4162 <check>\cs_generate_variant:Nn \toks_put_right:Nn {No}
\toks_gput_right:cV 4163 <!*check>
```

A couple done by hand for speed.

```

4164 \cs_new:Npn \toks_put_right:NV #1#2 {
4165   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4166     \exp_after:wN \toks_use:N \exp_after:wN #1
4167     \int_to_roman:w -'0 \exp_eval_register:N #2
4168   }
4169 }
4170 \cs_new:Npn \toks_put_right:No #1#2 {
4171   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4172     \exp_after:wN \toks_use:N \exp_after:wN #1 #2
4173   }
4174 }
4175 </!check>
4176 \cs_generate_variant:Nn \toks_put_right:Nn {Nx,cn,cV,co}
4177 \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.

```

4178 <check>\cs_generate_variant:Nn \toks_put_right:Nn {Nf}
4179 <!*check>
4180 \cs_new:Npn \toks_put_right:Nf #1#2 {
4181   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4182     \exp_after:wN \toks_use:N \exp_after:wN #1 \int_to_roman:w -'0#2
4183   }
4184 }
4185 </!check>

```

### 106.3 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.

```

4186 \prg_new_conditional:Nnn \toks_if_empty:N {p,TF,T,F} {
4187   \tl_if_empty:VTF #1 {\prg_return_true:} {\prg_return_false:}
4188 }
4189 \cs_generate_variant:Nn \toks_if_empty_p:N {c}
4190 \cs_generate_variant:Nn \toks_if_empty:NTF {c}
4191 \cs_generate_variant:Nn \toks_if_empty:NT {c}
4192 \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
4193 \prg_new_conditional:Nnn \toks_if_eq:NN {p,TF,T,F} {
4194   \tl_if_eq:xxTF {\toks_use:N #1} {\toks_use:N #2}
4195   {\prg_return_true:} {\prg_return_false:}
4196 }
4197 \cs_generate_variant:Nn \toks_if_eq_p:NN {Nc,c,cc}
4198 \cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
4199 \cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
4200 \cs_generate_variant:Nn \toks_if_eq:NNF {Nc,c,cc}

```

## 106.4 Variables and constants

`\l_tmpa_toks` Some scratch registers ...

```

\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks
4201 \tex_toksdef:D \l_tmpa_toks = 255\scan_stop:
4202 \initex\seq_put_right:Nn \g_toks_allocation_seq {255}
4203 \toks_new:N \l_tmpb_toks
4204 \toks_new:N \l_tmpc_toks
4205 \toks_new:N \g_tmpa_toks
4206 \toks_new:N \g_tmpb_toks
4207 \toks_new:N \g_tmpc_toks

```

`\c_empty_toks` And here is a constant, which is a (permanently) empty token register.

```
4208 \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.

```
4209 \toks_new:N \l_tl_replace_toks
```

```
4210 \</initex | package>
```

Show token usage:

```

4211 <*showmemory>
4212 \showMemUsage
4213 </showmemory>

```

## 107 l3seq implementation

```

4214 <*package>
4215 \ProvidesExplPackage
4216 { \filename } { \filedate } { \fileversion } { \filedescription }
4217 \package_check_loaded_expl:
4218 </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:`

```

4219 <*initex | package>
4220 \cs_new:Npn \seq_elt:w { \ERROR }
4221 \cs_new:Npn \seq_elt_end: { \ERROR }

```

### 107.1 Allocating and initialisation

`\seq_new:N` Sequences are implemented using token lists.

```

\seq_new:c
4222 \cs_new_eq:NN \seq_new:N \tl_new:N
4223 \cs_new_eq:NN \seq_new:c \tl_new:c

```

`\seq_clear:N` Clearing a sequence is the same as clearing a token list.  
`\seq_clear:c`  
`\seq_gclear:N` 4224 `\cs_new_eq:NN \seq_clear:N \tl_clear:N`  
`\seq_gclear:c` 4225 `\cs_new_eq:NN \seq_clear:c \tl_clear:c`  
4226 `\cs_new_eq:NN \seq_gclear:N \tl_gclear:N`  
4227 `\cs_new_eq:NN \seq_gclear:c \tl_gclear:c`

`\seq_clear_new:N` Clearing a sequence is the same as clearing a token list.  
`\seq_clear_new:c`  
`\seq_gclear_new:N` 4228 `\cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N`  
`\seq_gclear_new:c` 4229 `\cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c`  
4230 `\cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N`  
4231 `\cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c`

`\seq_set_eq:NN` We can set one seq equal to another.  
`\seq_set_eq:Nc`  
`\seq_set_eq:cN` 4232 `\cs_new_eq:NN \seq_set_eq:NN \cs_set_eq:NN`  
`\seq_set_eq:cc` 4233 `\cs_new_eq:NN \seq_set_eq:cN \cs_set_eq:cN`  
4234 `\cs_new_eq:NN \seq_set_eq:Nc \cs_set_eq:Nc`  
4235 `\cs_new_eq:NN \seq_set_eq:cc \cs_set_eq:cc`

`\seq_gset_eq:NN` And of course globally which seems to be needed far more often.<sup>13</sup>  
`\seq_gset_eq:cN` 4236 `\cs_new_eq:NN \seq_gset_eq:NN \cs_gset_eq:NN`  
`\seq_gset_eq:Nc` 4237 `\cs_new_eq:NN \seq_gset_eq:cN \cs_gset_eq:cN`  
`\seq_gset_eq:cc` 4238 `\cs_new_eq:NN \seq_gset_eq:Nc \cs_gset_eq:Nc`  
4239 `\cs_new_eq:NN \seq_gset_eq:cc \cs_gset_eq:cc`

`\seq_gconcat:NNN` `\seq_gconcat:NNN`  $\langle seq\ 1 \rangle \langle seq\ 2 \rangle \langle seq\ 3 \rangle$  will globally assign  $\langle seq\ 1 \rangle$  the concatenation  
`\seq_gconcat:ccc` of  $\langle seq\ 2 \rangle$  and  $\langle seq\ 3 \rangle$ .

4240 `\cs_new_nopar:Npn \seq_gconcat:NNN #1#2#3 {`  
4241 `\tl_gset:Nx #1 { \exp_not:V #2 \exp_not:V #3 }`  
4242 `}`  
4243 `\cs_generate_variant:Nn \seq_gconcat:NNN {ccc}`

## 107.2 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 $\underline{TF}$`  4244 `\prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N {p,TF,T,F}`  
`\seq_if_empty:c $\underline{TF}$`  4245 `\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.

4246 `\cs_new_nopar:Npn \seq_if_empty_err:N #1 {`  
4247 `\if_meaning:w #1 \c_empty_tl`

---

<sup>13</sup>To save a bit of space these functions could be made identical to those from the `tl` or `clist` module.

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.)

```

4248      \tl_clear:N \l_testa_tl % catch prefixes
4249      \msg_kernel_bug:x {Empty~sequence~'\token_to_str:N#1'}
4250      \fi:
4251    }

```

`\seq_if_in:NnTF` `\seq_if_in:NnTF`  $\langle seq \rangle \langle item \rangle \langle true\ case \rangle \langle false\ case \rangle$  will check whether  $\langle item \rangle$  is in  $\langle seq \rangle$  and then either execute the  $\langle true\ case \rangle$  or the  $\langle false\ case \rangle$ .  $\langle true\ case \rangle$  and  $\langle false\ case \rangle$  may contain incomplete `\if_charcode:w` statements.

`\seq_if_in:cnTF`

`\seq_if_in:cVTF`

`\seq_if_in:coTF`

`\seq_if_in:cxTF`

Note that ##2 in the definition below for `\seq_tmp:w` contains exactly one token which we can compare with `\q_no_value`.

```

4252 \prg_new_conditional:Nnn \seq_if_in:Nn {TF,T,F} {
4253   \cs_set:Npn \seq_tmp:w ##1 \seq_elt:w #2 \seq_elt_end: ##2##3 \q_stop {
4254     \if_meaning:w \q_no_value ##2
4255     \prg_return_false: \else: \prg_return_true: \fi:
4256   }
4257   \exp_after:wN \seq_tmp:w #1 \seq_elt:w #2 \seq_elt_end: \q_no_value \q_stop
4258 }

4259 \cs_generate_variant:Nn \seq_if_in:NnTF {cV,co,c,cx}
4260 \cs_generate_variant:Nn \seq_if_in:NnT  {cV,co,c,cx}
4261 \cs_generate_variant:Nn \seq_if_in:NnF   {cV,co,c,cx}

```

### 107.3 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`

```

4262 \cs_new_nopar:Npn \seq_get:NN #1 {
4263   \seq_if_empty_err:N #1
4264   \exp_after:wN \seq_get_aux:w #1 \q_stop
4265 }
4266 \cs_new:Npn \seq_get_aux:w \seq_elt:w #1 \seq_elt_end: #2 \q_stop #3 {
4267   \tl_set:Nn #3 {#1}
4268 }
4269 \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  $\langle def_1 \rangle$ .

`\seq_pop_aux:w`

```

4270 \cs_new:Npn \seq_pop_aux:nnNN #1#2#3 {
4271   \seq_if_empty_err:N #3
4272   \exp_after:wN \seq_pop_aux:w #3 \q_stop #1#2#3
4273 }
4274 \cs_new:Npn \seq_pop_aux:w
4275   \seq_elt:w #1 \seq_elt_end: #2 \q_stop #3#4#5#6 {
4276   #3 #5 {#2}

```

```

4277   #4 #6 {#1}
4278 }

\seq_show:N
\seq_show:c
4279 \cs_new_eq:NN \seq_show:N \tl_show:N
4280 \cs_new_eq:NN \seq_show:c \tl_show:c

\seq_display:N
\seq_display:c
4281 \cs_new_nopar:Npn \seq_display:N #1 {
4282   \iow_term:x { Sequence~\token_to_str:N #1~contains~
4283     the~elements~(without~outer~braces): }
4284   \toks_clear:N \l_tmpa_toks
4285   \seq_map_inline:Nn #1 {
4286     \toks_if_empty:NF \l_tmpa_toks {
4287       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
4288     }
4289     \toks_put_right:Nx \l_tmpa_toks {
4290       \iow_space: \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
4291     }
4292   }
4293   \toks_show:N \l_tmpa_toks
4294 }
4295 \cs_generate_variant:Nn \seq_display:N {c}

```

## 107.4 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.

```

4296 \cs_new:Npn \seq_put_aux:Nnn #1 {
4297   \exp_after:wN \seq_put_aux:w #1 \q_stop #1
4298 }
4299 \cs_new: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

```

```

4300 \cs_new:Npn \seq_put_left:Nn #1#2 {
4301   \seq_put_aux:Nnn #1 {\seq_elt:w #2\seq_elt_end:} {}
4302 }

```

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).

```

4303 \cs_generate_variant:Nn \seq_put_left:Nn {NV,No,Nx,c,cV,co}

4304 \cs_new:Npn \seq_put_right:Nn #1#2{
4305   \seq_put_aux:Nnn #1{\seq_elt:w #2\seq_elt_end:}}

4306 \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

4307 \cs_new:Npn \seq_gput_left:Nn {
4308   \*check
4309   \pref_global_chk:
4310   \</check>
4311   \*!check
4312   \pref_global:D
4313   \</!check>
4314   \seq_put_left:Nn
4315 }

4316 \cs_new:Npn \seq_gput_right:Nn {
4317   \*check
4318   \pref_global_chk:
4319   \</check>
4320   \*!check
4321   \pref_global:D
4322   \</!check>
4323   \seq_put_right:Nn
4324 }

4325 \cs_generate_variant:Nn \seq_gput_left:Nn {NV,No,Nx,c,cV,co}
4326 \cs_generate_variant:Nn \seq_gput_right:Nn {NV,No,Nx,c,cV,co}

\seq_gput_right:Nc TODO: move to xor (Sep 2008)

4327 \cs_generate_variant:Nn \seq_gput_right:Nn {Nc}

```

## 107.5 Mapping

```

\seq_map_variable:NNn Nothing spectacular here.
\seq_map_variable:cNn
\seq_map_variable_aux:Nnw
  \seq_map_break:
  \seq_map_break:n

4328 \cs_new:Npn \seq_map_variable_aux:Nnw #1#2 \seq_elt:w #3 \seq_elt_end: {
4329   \tl_set:Nx #1{\exp_not:n{#3}}
4330   \quark_if_nil:NT #1 \seq_map_break:
4331   #2
4332   \seq_map_variable_aux:Nnw #1{#2}
4333 }
4334 \cs_new:Npn \seq_map_variable:NNn #1#2#3 {
4335   \tl_set:Nx #2 {\exp_not:n{\seq_map_variable_aux:Nnw #2{#3}}}
4336   \exp_after:wN #2 #1 \seq_elt:w \q_nil\seq_elt_end: \q_stop
4337 }
4338 \cs_generate_variant:Nn \seq_map_variable:NNn {c}
4339
4340 \cs_new_eq:NN \seq_map_break: \use_none_delimit_by_q_stop:w
4341 \cs_new_eq:NN \seq_map_break:n \use_i_delimit_by_q_stop:nw

\seq_map_function:NN \seq_map_function:NN <sequence> <cmd> applies <cmd> to each element of <sequence>,
\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 <cmd> must be a function with one argument that is delimited
by \seq_elt_end:.

```

```

4342 \cs_new_nopar:Npn \seq_map_function:NN #1#2 {
4343   \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2{##1}}
4344   #1
4345   \cs_set_eq:NN \seq_elt:w \ERROR
4346 }
4347 \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

```

4348 \cs_new_nopar:Npn \seq_map_inline:Nn #1#2 {
4349   \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2}
4350   #1
4351   \cs_set_eq:NN \seq_elt:w \ERROR
4352 }
4353 \cs_generate_variant:Nn \seq_map_inline:Nn {c}

```

## 107.6 Manipulation

\l\_clist\_remove\_clist A common scratch space for the removal routines.

```

4354 \seq_new:N \l_seq_remove_seq

```

\seq\_remove\_duplicates\_aux:NN Copied from \clist\_remove\_duplicates.

```

\seq_remove_duplicates_aux:n
\seq_remove_duplicates:N
\seq_gremove_duplicates:N
4355 \cs_new:Nn \seq_remove_duplicates_aux:NN {
4356   \seq_clear:N \l_seq_remove_seq
4357   \seq_map_function:NN #2 \seq_remove_duplicates_aux:n
4358   #1 #2 \l_seq_remove_seq
4359 }
4360 \cs_new:Nn \seq_remove_duplicates_aux:n {
4361   \seq_if_in:NnF \l_seq_remove_seq {#1} {
4362     \seq_put_right:Nn \l_seq_remove_seq {#1}
4363   }
4364 }
4365 \cs_new_nopar:Npn \seq_remove_duplicates:N {
4366   \seq_remove_duplicates_aux:NN \seq_set_eq:NN
4367 }
4368 \cs_new_nopar:Npn \seq_gremove_duplicates:N {
4369   \seq_remove_duplicates_aux:NN \seq_gset_eq:NN
4370 }

```

## 107.7 Sequence stacks

\seq\_push:Nn 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.

```

\seq_push:NV
\seq_push:No
\seq_push:cn
\seq_pop:NN
\seq_pop:cn
4371 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
4372 \cs_new_eq:NN \seq_push:NV \seq_put_left:NV
4373 \cs_new_eq:NN \seq_push:No \seq_put_left:No
4374 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
4375 \cs_new_nopar:Npn \seq_pop:NN { \seq_pop_aux:nnNN \tl_set:Nn \tl_set:Nn }
4376 \cs_generate_variant:Nn \seq_pop:NN {c}

```



`\seq_gpush:Nn` I don't agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of `\seq_gpop:NN` the value is nevertheless returned locally.

`\seq_gpush:Nv`  
`\seq_gpop:NN`  
`\seq_gpop:cN`

```

4377 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
4378 \cs_new_nopar:Npn \seq_gpop:NN { \seq_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn }
4379 \cs_generate_variant:Nn \seq_gpush:Nn {NV,No,c,Nv}
4380 \cs_generate_variant:Nn \seq_gpop:NN {c}

```

`\seq_top:NN` Looking at the top element of the stack without removing it is done with this operation.

`\seq_top:cN`

```

4381 \cs_new_eq:NN \seq_top:NN \seq_get:NN
4382 \cs_new_eq:NN \seq_top:cN \seq_get:cN

```

```

4383 </initex | package>

```

Show token usage:

```

4384 <*showmemory>
4385 %\showMemUsage
4386 </showmemory>

```

## 108 l3clist implementation

We start by ensuring that the required packages are loaded.

```

4387 <*package>
4388 \ProvidesExplPackage
4389 { \filename } { \filedate } { \fileversion } { \filedescription }
4390 \package_check_loaded_expl:
4391 </package>
4392 <*initex | package>

```

### 108.1 Allocation and initialisation

`\clist_new:N` Comma-Lists are implemented using token lists.

`\clist_new:c`

```

4393 \cs_new_eq:NN \clist_new:N \tl_new:N
4394 \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`

```

4395 \cs_new_eq:NN \clist_clear:N \tl_clear:N
4396 \cs_generate_variant:Nn \clist_clear:N {c}
4397 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
4398 \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`

```

4399 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
4400 \cs_generate_variant:Nn \clist_clear_new:N {c}
4401 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
4402 \cs_generate_variant:Nn \clist_gclear_new:N {c}

```



### 108.3 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.

```

4433 \cs_new_nopar:Npn \clist_use:N #1 {
4434   \if_meaning:w #1 \scan_stop:
4435     \msg_kernel_bug:x {
4436       Comma~list~ '\token_to_str:N #1'~ has~ an~ erroneous~ structure!}
4437   \else:
4438     \exp_after:wN #1
4439   \fi:
4440 }
4441 \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`

```

4442 \cs_new_nopar:Npn \clist_get:NN #1 {
4443   \clist_if_empty_err:N #1
4444   \exp_after:wN \clist_get_aux:w #1,\q_stop
4445 }
4446 \cs_new:Npn \clist_get_aux:w #1,#2\q_stop #3 { \tl_set:Nn #3{#1} }
4447 \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`

```

4448 \cs_new:Npn \clist_pop_aux:nnNN #1#2#3 {
4449   \clist_if_empty_err:N #3
4450   \exp_after:wN \clist_pop_aux:w #3,\q_nil\q_stop #1#2#3
4451 }

```

After the assignments below, if there was only one element in the original `clist`, it now contains only `\q_nil`.

```

4452 \cs_new:Npn \clist_pop_aux:w #1,#2\q_stop #3#4#5#6 {
4453   #4 #6 {#1}
4454   #3 #5 {#2}
4455   \quark_if_nil:NTF #5 { #3 #5 {} }{ \clist_pop_auxi:w #2 #3#5 }
4456 }
4457 \cs_new:Npn \clist_pop_auxi:w #1,\q_nil #2#3 { #2#3{#1} }

```

`\clist_show:N`  
`\clist_show:c`

```

4458 \cs_new_eq:NN \clist_show:N \tl_show:N
4459 \cs_new_eq:NN \clist_show:c \tl_show:c

```

```

\clist_display:N
\clist_display:c
4460 \cs_new_nopar:Npn \clist_display:N #1 {
4461   \iow_term:x { Comma-list~\token_to_str:N #1~contains~
4462     the~elements~(without~outer~braces): }
4463   \toks_clear:N \l_tmpa_toks
4464   \clist_map_inline:Nn #1 {
4465     \toks_if_empty:NF \l_tmpa_toks {
4466       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
4467     }
4468     \toks_put_right:Nx \l_tmpa_toks {
4469       \iow_space: \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
4470     }
4471   }
4472   \toks_show:N \l_tmpa_toks
4473 }
4474 \cs_generate_variant:Nn \clist_display:N {c}

```

## 108.4 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.

```

4475 \cs_new:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6 {
4476   \clist_if_empty:NTF #5 { #1 #5 {#6} } {
4477     \tl_if_empty:nF {#6} { #2 #5{#3#6#4} }
4478   }
4479 }

```

`\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`

`\clist_gput_left:NV`

`\clist_gput_left:No`

`\clist_gput_left:Nx`

`\clist_gput_left:cn`

`\clist_gput_left:cV`

`\clist_gput_left:co`

Global versions.

```

4484 \cs_new_nopar:Npn \clist_gput_left:Nn {
4485   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn {} ,
4486 }
4487 \cs_generate_variant:Nn \clist_gput_left:Nn {NV,No,Nx,cn,cV,co}

```

`\clist_put_right:Nn` Adding something to the right side is almost the same.

`\clist_put_right:NV`

`\clist_put_right:No`

`\clist_put_right:Nx`

`\clist_put_right:cn`

`\clist_put_right:cV`

`\clist_put_right:co`

```

4488 \cs_new_nopar:Npn \clist_put_right:Nn {
4489   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn {} ,
4490 }
4491 \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` 4492 `\cs_new_nopar:Npn \clist_gput_right:Nn {`  
`\clist_gput_right:Nx` 4493 `\clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , {}`  
`\clist_gput_right:cn` 4494 `}`  
`\clist_gput_right:cV` 4495 `\cs_generate_variant:Nn \clist_gput_right:Nn {NV,No,Nx,cn,cV,co}`  
`\clist_gput_right:co`

## 108.5 Mapping

`\clist_map_function:NN` `\clist_map_function:NN`  $\langle comma-list \rangle$   $\langle cmd \rangle$  applies  $\langle cmd \rangle$  to each element of  $\langle comma-list \rangle$ ,  
`\clist_map_function:cn` from left to right.  
`\clist_map_function:nN`

```

4496 \cs_new_nopar:Npn \clist_map_function:NN #1#2 {
4497   \clist_if_empty:NF #1 {
4498     \exp_after:wN \clist_map_function_aux:Nw
4499     \exp_after:wN #2 #1 , \q_recursion_tail , \q_recursion_stop
4500   }
4501 }
4502 \cs_generate_variant:Nn \clist_map_function:NN {cN}

4503 \cs_new:Npn \clist_map_function:nN #1#2 {
4504   \tl_if_blank:nF {#1} {
4505     \clist_map_function_aux:Nw #2 #1 , \q_recursion_tail , \q_recursion_stop
4506   }
4507 }

```

`\clist_map_function_aux:Nw` The general loop. Tests if we hit the first stop marker and exits if we did. If we didn't, place the function #1 in front of the element #2, which is surrounded by braces.

```

4508 \cs_new:Npn \clist_map_function_aux:Nw #1#2,{
4509   \quark_if_recursion_tail_stop:n{#2}
4510   #1{#2}
4511   \clist_map_function_aux:Nw #1
4512 }

```

`\clist_map_break:` The break statement is easy. Same as in other modules, gobble everything up to the special recursion stop marker.

```

4513 \cs_new_eq:NN \clist_map_break: \use_none_delimit_by_q_recursion_stop:w

```

`\clist_map_inline:Nn` The inline type is faster but not expandable. In order to make it nestable, we use a  
`\clist_map_inline:cn` counter to keep track of the nesting level so that all of the functions called have distinct  
`\clist_map_inline:nn` names. A simpler approach would of course be to use grouping and thus the save stack  
 but then you lose the ability to do things locally.

A funny little thing occurred in one document: The command setting up the first call of `\clist_map_inline:Nn` was used in a tabular cell and the inline code used `\` so the loop broke as soon as this happened. Lesson to be learned from this: If you wish to have group like structure but not using the groupings of  $\text{\TeX}$ , then do every operation globally.

```

4514 \int_new:N \g_clist_inline_level_int

```

```

4515 \cs_new:Npn \clist_map_inline:Nn #1#2 {
4516   \clist_if_empty:NF #1 {
4517     \int_gincr:N \g_clist_inline_level_int
4518     \cs_gset:cpn {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
4519     ##1{#2}

```

```

4520 \exp_last_unbraced:NcV \clist_map_function_aux:Nw
4521 {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
4522 #1 , \q_recursion_tail , \q_recursion_stop
4523 \int_gdecr:N \g_clist_inline_level_int
4524 }
4525 }
4526 \cs_generate_variant:Nn \clist_map_inline:Nn {c}
4527 \cs_new:Npn \clist_map_inline:nn #1#2 {
4528   \tl_if_empty:nF {#1} {
4529     \int_gincr:N \g_clist_inline_level_int
4530     \cs_gset:cpn {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
4531     ##1{#2}
4532     \exp_args:Nc \clist_map_function_aux:Nw
4533     {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
4534     #1 , \q_recursion_tail , \q_recursion_stop
4535     \int_gdecr:N \g_clist_inline_level_int
4536   }
4537 }

```

```

4538 \cs_new:Npn \clist_map_variable:nNn #1#2#3 {
4539   \tl_if_empty:nF {#1} {
4540     \clist_map_variable_aux:Nnw #2 {#3} #1
4541     , \q_recursion_tail , \q_recursion_stop
4542   }
4543 }

```

```
4544 \cs_new_nopar:Npn \clist_map_variable:NNn {\exp_args:No \clist_map_variable:nNn}
4545 \cs_generate_variant:Nn\clist_map_variable:NNn {cNn}
```

```

4546 \cs_new:Npn \clist_map_variable_aux:Nnw #1#2#3,{
4547   \cs_set_nopar:Npn #1{#3}
4548   \quark_if_recursion_tail_stop:N #1
4549   #2 \clist_map_variable_aux:Nnw #1{#2}
4550 }

```

## 108.6 Higher level functions

`\clist_concat_aux:NNNN` `\clist_gconcat: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:NNN`  
`\clist_concat:ccc`  
`\clist_gconcat:NNN`  
`\clist_gconcat:ccc`

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.

```

4551 \cs_new_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4 {
4552   \toks_set:No \l_tmpa_toks {#3}
4553   \toks_set:No \l_tmpb_toks {#4}
4554   #1 #2 {
4555     \toks_use:N \l_tmpa_toks
4556     \toks_if_empty:NF \l_tmpa_toks { \toks_if_empty:NF \l_tmpb_toks , }
4557     \toks_use:N \l_tmpb_toks
4558   }
4559 }
4560 \cs_new_nopar:Npn \clist_concat:NNN { \clist_concat_aux:NNNN \tl_set:Nx }
4561 \cs_new_nopar:Npn \clist_gconcat:NNN { \clist_concat_aux:NNNN \tl_gset:Nx }
4562 \cs_generate_variant:Nn \clist_concat:NNN {ccc}
4563 \cs_generate_variant:Nn \clist_gconcat:NNN {ccc}

```

`\l_clist_remove_clist` A common scratch space for the removal routines.

```

4564 \clist_new:N \l_clist_remove_clist

```

`\list_remove_duplicates_aux:NN` `\clist_remove_duplicates_aux:n` `\clist_remove_duplicates:N` `\clist_gremove_duplicates:N`

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 element is already present in the list.

```

4565 \cs_new:Nn \clist_remove_duplicates_aux:NN {
4566   \clist_clear:N \l_clist_remove_clist
4567   \clist_map_function:NN #2 \clist_remove_duplicates_aux:n
4568   #1 #2 \l_clist_remove_clist
4569 }
4570 \cs_new:Nn \clist_remove_duplicates_aux:n {
4571   \clist_if_in:NnF \l_clist_remove_clist {#1} {
4572     \clist_put_right:Nn \l_clist_remove_clist {#1}
4573   }
4574 }

```

The high level functions are just for telling if it should be a local or global setting.

```

4575 \cs_new_nopar:Npn \clist_remove_duplicates:N {
4576   \clist_remove_duplicates_aux:NN \clist_set_eq:NN
4577 }
4578 \cs_new_nopar:Npn \clist_gremove_duplicates:N {
4579   \clist_remove_duplicates_aux:NN \clist_gset_eq:NN
4580 }

```

`\clist_remove_element:Nn` `\clist_gremove_element:Nn`

The same general idea is used for removing elements: the parent functions just set things up for the internal ones.

`\clist_remove_element_aux:NNn` `\clist_remove_element_aux:n`

```

4581 \cs_new_nopar:Npn \clist_remove_element:Nn {

```

```

4582 \clist_remove_element_aux:NNn \clist_set_eq:NN
4583 }
4584 \cs_new_nopar:Npn \clist_gremove_element:Nn {
4585   \clist_remove_element_aux:NNn \clist_gset_eq:NN
4586 }
4587 \cs_new:Nn \clist_remove_element_aux:NNn {
4588   \clist_clear:N \l_clist_remove_clist
4589   \cs_set:Nn \clist_remove_element_aux:n {
4590     \tl_if_eq:nnF {#3} {##1} {
4591       \clist_put_right:Nn \l_clist_remove_clist {##1}
4592     }
4593   }
4594   \clist_map_function:NN #2 \clist_remove_element_aux:n
4595   #1 #2 \l_clist_remove_clist
4596 }
4597 \cs_new:Nn \clist_remove_element_aux:n { }

```

## 108.7 Stack operations

We build stacks from comma-lists, but here we put the specific functions together.

<pre> \clist_push:Nn \clist_push:No \clist_push:NV \clist_push:cn \clist_pop:NN \clist_pop:cn </pre>	<p>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.</p> <pre> 4598 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn 4599 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV 4600 \cs_new_eq:NN \clist_push:No \clist_put_left:No 4601 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn 4602 \cs_new_nopar:Npn \clist_pop:NN {\clist_pop_aux:nnNN \tl_set:Nn \tl_set:Nn} 4603 \cs_generate_variant:Nn \clist_pop:NN {cn} </pre>
<pre> \clist_gpush:Nn \clist_gpush:No \clist_gpush:NV \clist_gpush:cn \clist_gpop:NN \clist_gpop:cn </pre>	<p>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 <code>\clist_gpop:NN</code> the value is nevertheless returned locally.</p> <pre> 4604 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn 4605 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV 4606 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No 4607 \cs_generate_variant:Nn \clist_gpush:Nn {cn}  4608 \cs_new_nopar:Npn \clist_gpop:NN {\clist_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn} 4609 \cs_generate_variant:Nn \clist_gpop:NN {cn} </pre>
<pre> \clist_top:NN \clist_top:cn </pre>	<p>Looking at the top element of the stack without removing it is done with this operation.</p> <pre> 4610 \cs_new_eq:NN \clist_top:NN \clist_get:NN 4611 \cs_new_eq:NN \clist_top:cn \clist_get:cn </pre> <p style="margin-top: 10px;">4612 <code>&lt;/initex   package&gt;</code></p>

Show token usage:

```

4613 <*showmemory>
4614 %\showMemUsage
4615 </showmemory>

```



## 109 l3prop implementation

A property list is a token register whose contents is of the form

`\q_prop <key1> \q_prop {<info1>} ... \q_prop <keyn> \q_prop {<infon>}`

The property `<key>`s and `<info>`s might be arbitrary token lists; each `<info>` is surrounded by braces.

We start by ensuring that the required packages are loaded.

```

4616 <*package>
4617 \ProvidesExplPackage
4618   {\filename}{\filedate}{\fileversion}{\filedescription}
4619 \package_check_loaded_expl:
4620 </package>
4621 <*initex | package>

```

`\q_prop` The separator between `<key>`s and `<info>`s and `<key>`s.

```

4622 \quark_new:N\q_prop

```

To get values from property-lists, token lists should be passed to the appropriate functions.

### 109.1 Functions

`\prop_new:N` Property lists are implemented as token registers.

```

\prop_new:c
4623 \cs_new_eq:NN \prop_new:N \toks_new:N
4624 \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
4625 \cs_new_eq:NN \prop_clear:N \toks_clear:N
4626 \cs_new_eq:NN \prop_clear:c \toks_clear:c
\prop_gclear:c
4627 \cs_new_eq:NN \prop_gclear:N \toks_gclear:N
4628 \cs_new_eq:NN \prop_gclear:c \toks_gclear:c

```

`\prop_set_eq:NN` This makes two `<prop>`s have the same contents.

```

\prop_set_eq:Nc
\prop_set_eq:cN
4629 \cs_new_eq:NN \prop_set_eq:NN \toks_set_eq:NN
4630 \cs_new_eq:NN \prop_set_eq:Nc \toks_set_eq:Nc
\prop_set_eq:cc
4631 \cs_new_eq:NN \prop_set_eq:cN \toks_set_eq:cN
\prop_gset_eq:NN
4632 \cs_new_eq:NN \prop_set_eq:cc \toks_set_eq:cc
\prop_gset_eq:Nc
4633 \cs_new_eq:NN \prop_gset_eq:NN \toks_gset_eq:NN
\prop_gset_eq:cN
4634 \cs_new_eq:NN \prop_gset_eq:Nc \toks_gset_eq:Nc
\prop_gset_eq:cc
4635 \cs_new_eq:NN \prop_gset_eq:cN \toks_gset_eq:cN
4636 \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
4637 \cs_new_eq:NN \prop_show:N \toks_show:N
4638 \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`

```

4639 \cs_new_nopar:Npn \prop_display:N #1 {
4640   \iow_term:x { Property-list~\token_to_str:N #1~contains~
4641     the~pairs~(without~outer~braces): }
4642   \toks_clear:N \l_tmpa_toks
4643   \prop_map_inline:Nn #1 {
4644     \toks_if_empty:NF \l_tmpa_toks {
4645       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
4646     }
4647     \toks_put_right:Nx \l_tmpa_toks {
4648       \iow_space: \iow_char:N \{ \exp_not:n {##1} \iow_char:N \} \iow_space:
4649       \iow_space: => \iow_space:
4650       \iow_space: \iow_char:N \{ \exp_not:n {##2} \iow_char:N \}
4651     }
4652   }
4653   \toks_show:N \l_tmpa_toks
4654 }
4655 \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.

```

4656 \cs_new:Npn \prop_split_aux:Nnn #1#2#3{
4657   \cs_set:Npn \prop_tmp:w ##1 \q_prop #2 \q_prop ##2##3 \q_stop {
4658     #3 {##1}{##2}{##3}
4659   }
4660   \exp_after:wN \prop_tmp:w \toks_use:N #1 \q_prop #2 \q_prop \q_no_value \q_stop
4661 }

```

`\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:cnN`

`\prop_get_aux:w`

```

4662 \cs_new:Npn \prop_get:NnN #1#2 {
4663   \prop_split_aux:Nnn #1{#2}\prop_get_aux:w
4664 }
4665 \cs_new:Npn \prop_get_aux:w #1#2#3#4 { \tl_set:Nx #4 {\exp_not:n{#2}} }
4666 \cs_generate_variant:Nn \prop_get:NnN {cnN}

```

`\prop_gget:NnN`

The global version of the previous function.

`\prop_gget:cnN`

`\prop_gget_aux:w`

```

4667 \cs_new:Npn \prop_gget:NnN #1#2{
4668   \prop_split_aux:Nnn #1{#2}\prop_gget_aux:w}
4669 \cs_new:Npn \prop_gget_aux:w #1#2#3#4{\tl_gset:Nx#4{\exp_not:n{#2}}}
4670 \cs_generate_variant:Nn \prop_gget:NnN {cnN}

```

`\prop_get_gdel:NnN`

`\prop_get_del_aux:w`

`\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.

```

4671 \cs_new:Npn \prop_get_gdel:NnN #1#2#3{
4672   \prop_split_aux:Nnn #1{#2}{\prop_get_del_aux:w #3{\toks_gset:Nn #1}{#2}}}
4673 \cs_new:Npn \prop_get_del_aux:w #1#2#3#4#5#6{
4674   \tl_set:Nx #1{\exp_not:n{#5}}
4675   \quark_if_no_value:NF #1 {
4676     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#3\q_prop\q_no_value {#2{#4##1}}
4677     \prop_tmp:w #6}
4678 }

```

\prop\_put:Nnn \prop\_put:Nnn {<prop>} {<key>} {<info>} adds/changes the value associated with <key> in <prop> to <info>.

```

\prop_put:cnn
\prop_gput:Nnn
\prop_gput:NnV
\prop_gput:Nno
\prop_gput:Nnx
\prop_gput:cnn
\prop_gput:ccx
\prop_put_aux:w
4679 \cs_new:Npn \prop_put:Nnn #1#2{
4680   \prop_split_aux:Nnn #1{#2}{
4681     \prop_clear:N #1
4682     \prop_put_aux:w {\toks_put_right:Nn #1}{#2}
4683   }
4684 }
4685 \cs_new:Npn \prop_gput:Nnn #1#2{
4686   \prop_split_aux:Nnn #1{#2}{
4687     \prop_gclear:N #1
4688     \prop_put_aux:w {\toks_gput_right:Nn #1}{#2}
4689   }
4690 }

```

```

4691 \cs_new:Npn \prop_put_aux:w #1#2#3#4#5#6{
4692   #1{\q_prop#2\q_prop{#6}#3}
4693   \tl_if_empty:nF{#5}
4694   {
4695     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
4696     \prop_tmp:w #5
4697   }
4698 }

```

```

4699 \cs_generate_variant:Nn \prop_put:Nnn { NnV, cnn }

```

```

4700 \cs_generate_variant:Nn \prop_gput:Nnn { NnV, Nno, Nnx, Nox, cnn, ccx }

```

\prop\_del:Nn \prop\_del:Nn <prop> <key> deletes the entry for <key> in <prop>, if any.

```

\prop_gdel:Nn
\prop_del_aux:w
4701 \cs_new:Npn \prop_del:Nn #1#2{
4702   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_set:Nn #1}{#2}}}
4703 \cs_new:Npn \prop_gdel:Nn #1#2{
4704   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_gset:Nn #1}{#2}}}
4705 \cs_new:Npn \prop_del_aux:w #1#2#3#4#5{
4706   \cs_set_nopar:Npn \prop_tmp:w {#4}
4707   \quark_if_no_value:NF \prop_tmp:w {
4708     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{#3##1}}
4709     \prop_tmp:w #5
4710   }
4711 }

```

\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):

```

4712 \cs_new:Npn \prop_gput_if_new:Nnn #1#2{
4713   \prop_split_aux:Nnn #1{#2}{\prop_put_if_new_aux:w #1{#2}}
4714 \cs_new:Npn \prop_put_if_new_aux:w #1#2#3#4#5#6{
4715   \tl_if_empty:nT {#5}{#1{\q_prop#2\q_prop{#6}#3}}}

```

## 109.2 Predicates and conditionals

`\prop_if_empty_p:N` This conditional takes a `<prop>` as its argument and evaluates either the true or the false case, depending on whether or not `<prop>` contains any properties.

```

\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF
4716 \prg_new_eq_conditional:NNn \prop_if_empty:N \toks_if_empty:N {p,TF,T,F}
4717 \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
4718 \prg_new_eq_conditional:NNn \prop_if_eq:NN \toks_if_eq:NN {p,TF,T,F}
4719 \prg_new_eq_conditional:NNn \prop_if_eq:cN \toks_if_eq:cN {p,TF,T,F}
\prop_if_eq_p:cc
4720 \prg_new_eq_conditional:NNn \prop_if_eq:Nc \toks_if_eq:Nc {p,TF,T,F}
\prop_if_eq:NNTF
4721 \prg_new_eq_conditional:NNn \prop_if_eq:cc \toks_if_eq:cc {p,TF,T,F}
\prop_if_eq:NcTF

```

`\prop_if_in:NnTF` `<property_list>` `<key>` `<true_case>` `<false_case>` will check whether or not `<key>` is on the `<property_list>` and then select either the true or false case.

```

\prop_if_in:NcTF
\prop_if_in:cnTF
\prop_if_in:ccTF
\prop_if_in_aux:w
4722 \prg_new_conditional:Nnn \prop_if_in:Nn {TF,T,F} {
4723   \prop_split_aux:Nnn #1 {#2} {\prop_if_in_aux:w}
4724 }
4725 \cs_new_nopar:Npn \prop_if_in_aux:w #1#2#3 {
4726   \quark_if_no_value:nTF {#2} {\prg_return_false:} {\prg_return_true:}
4727 }
4728 \cs_generate_variant:Nn \prop_if_in:NnTF {NV,No,cn,cc}
4729 \cs_generate_variant:Nn \prop_if_in:NnT {NV,No,cn,cc}
4730 \cs_generate_variant:Nn \prop_if_in:NnF {NV,No,cn,cc}

```

## 109.3 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:

```

4731 \cs_set_nopar:Npn \prop_map_function:NN #1#2 {
4732   \exp_after:wN \prop_map_function_aux:w
4733   \exp_after:wN #2 \toks_use:N #1 \q_prop \q_nil \q_prop \q_no_value \q_stop
4734 }
4735 \cs_set:Npn \prop_map_function_aux:w #1 \q_prop #2 \q_prop #3 {
4736   \if_meaning:w \q_nil #2
4737   \exp_after:wN \prop_map_break:
4738   \fi:
4739   #1{#2}{#3}
4740   \prop_map_function_aux:w #1
4741 }

```

(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

```

4742 \cs_generate_variant:Nn \prop_map_function:NN {c,Nc,cc}

```

`\prop_map_inline:Nn` 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.

`\prop_map_inline:cn`

`\g_prop_inline_level_num`

```

4743 \num_new:N \g_prop_inline_level_num
4744 \cs_new_nopar:Npn \prop_map_inline:Nn #1#2 {
4745   \num_gincr:N \g_prop_inline_level_num
4746   \cs_gset:cpn {prop_map_inline_ \num_use:N \g_prop_inline_level_num :n}
4747     ##1##2{#2}
4748   \prop_map_function:Nc #1
4749     {prop_map_inline_ \num_use:N \g_prop_inline_level_num :n}
4750   \num_gdecr:N \g_prop_inline_level_num
4751 }

4752 \cs_generate_variant:Nn\prop_map_inline:Nn {cn}

```

`\prop_map_break:` The break statement.

```

4753 \cs_new_eq:NN \prop_map_break: \use_none_delimit_by_q_stop:w
4754 </initex | package>

```

Show token usage:

```

4755 <*showmemory>
4756 %\showMemUsage
4757 </showmemory>

```

## 110 l3io implementation

We start by ensuring that the required packages are loaded.

```

4758 <*package>
4759 \ProvidesExplPackage
4760   {\filename}{\filedate}{\fileversion}{\filedescription}
4761   \package_check_loaded_expl:
4762 </package>
4763 <*initex | package>

```

This section is primarily concerned with input and output streams. The naming conventions for i/o streams is `ior` (for read) and `iow` (for write) as module names. e.g. `\c_ior_test_stream` is an input stream variable called ‘test’.

### 110.1 Output streams

`\iow_new:N` Allocation of new output streams is done by these functions. As we currently do not distribute a new allocation module we nick the `\newwrite` function.

`\iow_new:c`

```

4764 <*initex>
4765 \alloc_setup_type:nnn {iow} \c_zero \c_sixteen
4766 \cs_new_nopar:Npn \iow_new:N #1 {\alloc_reg:NnNN g {iow} \tex_chardef:D #1}
4767 </initex>

```

```

4768 <*package>
4769 \cs_set_eq:NN \iow_new:N \newwrite
4770 </package>
4771 \cs_generate_variant:Nn \iow_new:N {c}

```

**\iow\_open:Nn** To open streams for reading or writing the following two functions are provided. The  
**\iow\_open:cn** streams are opened immediately.

From some bad experiences on the mainframe, I learned that it is better to force the close before opening a dataset for writing. We have to check whether this is also necessary in case of **\tex\_openin:D**.

```

4772 \cs_new_nopar:Npn \iow_open:Nn #1#2 {
4773   \iow_close:N #1
4774   \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
4775 }
4776 \cs_generate_variant:Nn \iow_open:Nn {c}

```

**\iow\_close:N** Since we close output streams prior to opening, a separate closing operation is probably not necessary. But here it is, just in case.... Actually you will need this if you intend to write and then read in the same pass from a stream.

```

4777 \cs_new_nopar:Npn \iow_close:N { \tex_immediate:D \tex_closeout:D }

```

**\c\_iow\_term\_stream** Here we allocate two output streams for writing to the transcript file only (**\c\_iow\_log\_stream**)  
**\c\_ior\_term\_stream** and to both the terminal and transcript file (**\c\_iow\_term\_stream**). Both can be used  
**\c\_iow\_log\_stream** to read from and have equivalent **\c\_ior** versions.  
**\c\_ior\_log\_stream**

```

4778 \cs_new_eq:NN \c_iow_term_stream \c_sixteen
4779 \cs_new_eq:NN \c_ior_term_stream \c_sixteen
4780 \cs_new_eq:NN \c_iow_log_stream \c_minus_one
4781 \cs_new_eq:NN \c_ior_log_stream \c_minus_one

```

## Immediate writing

**\iow\_now:Nx** An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```

4782 \cs_new_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.

```

4783 \cs_new_nopar:Npn \iow_now:Nn #1#2 {
4784   \iow_now:Nx #1 { \exp_not:n {#2} }
4785 }

```

**\iow\_log:n** Now we redefine two functions for which we needed a definition very early on.

```

\iow_log:x
\iow_term:n
\iow_term:x
4786 \cs_set_nopar:Npn \iow_log:x { \iow_now:Nx \c_iow_log_stream }
4787 \cs_new_nopar:Npn \iow_log:n { \iow_now:Nn \c_iow_log_stream }
4788 \cs_set_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
4789 \cs_new_nopar:Npn \iow_term:n { \iow_now:Nn \c_iow_term_stream }

```

`\iow_term:x` isn't exactly equivalent to the old `\typeout` since we need to control expansion in the function we provide for the user.

`\iow_now_when_avail:Nn` `\iow_now_when_avail:Nn`  $\langle stream \rangle$   $\langle code \rangle$ . This routine writes its second argument unexpanded to the stream given by the first argument, provided that this stream was opened for writing. Note, that `#` characters get doubled within  $\langle code \rangle$ .

In this routine we have to check whether or not the output stream that was requested is defined at all. So we check if the name is still free.

```

4790 \cs_new_nopar:Npn \iow_now_when_avail:Nn #1 {
4791   \cs_if_free:NTF #1 {\use_none:n} {\iow_now:Nn #1}
4792 }
4793 \cs_generate_variant:Nn \iow_now_when_avail:Nn {c}

```

`\iow_now_buffer_safe:Nn` `\iow_now_buffer_safe:Nx` `\iow_now_buffer_safe:Nx` 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.

```

4794 \cs_new_nopar:Npn \iow_now_buffer_safe_aux:w #1#2#3 {
4795   \group_begin: \tex_newlinechar:D'\ #1#2{#3} \group_end:
4796 }
4797 \cs_new_nopar:Npn \iow_now_buffer_safe:Nx {
4798   \iow_now_buffer_safe_aux:w \iow_now:Nx
4799 }
4800 \cs_new_nopar:Npn \iow_now_buffer_safe:Nn {
4801   \iow_now_buffer_safe_aux:w \iow_now:Nn
4802 }

```

## Deferred writing

`\iow_shipout_x:Nn` `\iow_shipout_x:Nx` First the easy part, this is the primitive.

```

4803 \cs_set_eq:NN \iow_shipout_x:Nn \tex_write:D
4804 \cs_generate_variant:Nn \iow_shipout_x:Nn {Nx}

```

`\iow_shipout:Nn` `\iow_shipout:Nx` With  $\varepsilon$ -T<sub>E</sub>X available deferred writing is easy.

```

4805 \cs_new_nopar:Npn \iow_shipout:Nn #1#2{
4806   \iow_shipout_x:Nn #1 { \exp_not:n {#2} }
4807 }
4808 \cs_generate_variant:Nn \iow_shipout:Nn {Nx}

```

'Buffer safe' forms of these functions are not possible since the deferred writing will restore the value of `\tex_newlinechar:D` before it will have a chance to act. But on the other hand it is nevertheless possible to make all deferred writes long by setting the `\tex_newlinechar:D` inside the output routine just before the `\tex_shipout:D`. The only disadvantage of this method is the fact that messages to the terminal during this time will also then break at spaces. But we should consider this.



## Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```
4809 \cs_new_nopar:Npn \iow_newline: {^^J}
```

`\iow_space:` Global variable holding the character that inserts a space char when writing to an output stream.

```
4810 \cs_new_nopar:Npn \iow_space: {~}
```

`\iow_char:N` Function to write any escaped char to an output stream.

```
4811 \cs_new:Nn \iow_char:N { \cs_to_str:N #1 }
```

`\c_iow_comment_char` TODO: remove these in favour of `\iow_char:N` (Will, Apr 2009)

`\c_iow_lbrace_char` We also need to be able to write braces and the comment character. We achieve  
`\c_iow_rbrace_char` this by defining global constants to expand into a version of these characters with  
`\tex_catcode:D = 12`.

```
4812 \tl_new:Nx \c_iow_comment_char { \cs_to_str:N \% }
```

To avoid another allocation function which is probably only necessary here we use the `\cs_set_nopar:Npx` command directly.

```
4813 \tl_new:Nx \c_iow_lbrace_char { \cs_to_str:N \{ }
```

```
4814 \tl_new:Nx \c_iow_rbrace_char { \cs_to_str:N \} }
```

## 110.2 Input streams

`\ior_new:N` Allocation of new input streams is done by this function. As we currently do not distribute a new allocation module we nick the `\newread` function.

```
4815 <*initex>
4816 \alloc_setup_type:nnn {ior} \c_zero \c_sixteen
4817 \cs_new_nopar:Npn \ior_new:N #1 { \alloc_reg:NnNN g {ior} \tex_chardef:D #1 }
4818 </initex>
4819 <package> \cs_set_eq:NN \ior_new:N \newread
```

`\ior_open:Nn` Processing of input-streams (via `\tex_openin:D` and `\closein`) is always ‘immediate’ as  
`\ior_close:N` far as T<sub>E</sub>X is concerned. An extra `\tex_immediate:D` is silently ignored.

```
4820 \cs_set_eq:NN \ior_close:N \tex_closein:D
4821 \cs_new_nopar:Npn \ior_open:Nn #1#2 {
4822   \ior_close:N #1 \scan_stop:
4823   \tex_openin:D #1#2 \scan_stop:
4824 }
```

`\if_eof:w`

```
4825 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

`\ior_if_eof_p:N` `\ior_if_eof:NTF`  $\langle stream \rangle$   $\langle true\ case \rangle$   $\langle false\ case \rangle$ . To test if some particular input stream is exhausted the following conditional is provided:

```
4826 \prg_new_conditional:Nnn \ior_if_eof:N {p,TF,T,F} {
4827   \tex_ifeof:D #1 \prg_return_true: \else: \prg_return_false: \fi:
4828 }
```

`\ior_to:NN` And here we read from files.  
`\ior_gto:NN`

```
4829 \cs_new_nopar:Npn \ior_to:NN #1#2 {
4830   \tex_read:D #1 to #2
4831   \check \chk_local_or_pref_global:N #2
4832 }
4833 \cs_new_nopar:Npn \ior_gto:NN {
4834   \+check \pref_global_chk:
4835   \-check \pref_global:D
4836   \ior_to:NN
4837 }
4838 \</initex | package>
```

Show token usage:

```
4839 \*showmemory>
4840 \showMemUsage
4841 \</showmemory>
```

## 111 l3msg implementation

The usual lead-off.

```
4842 \*package>
4843 \ProvidesExplPackage
4844   {\filename}{\filedate}{\fileversion}{\filedescription}
4845   \package_check_loaded_expl:
4846 \</package>
4847 \*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.

```
4848 \int_set:Nn \tex_errorcontextlines:D { \c_minus_one }
```

### 111.1 Variables and constants

`\c_msg_fatal_tl` Header information.

```
\c_msg_error_tl
\c_msg_warning_tl
\c_msg_info_tl
4849 \tl_new:Nn \c_msg_fatal_tl { Fatal~Error }
4850 \tl_new:Nn \c_msg_error_tl { Error }
4851 \tl_new:Nn \c_msg_warning_tl { Warning }
4852 \tl_new:Nn \c_msg_info_tl { Info }
```

<code>\c_msg_fatal_text_tl</code>	Simple pieces of text for messages.
<code>\c_msg_help_text_tl</code>	
<code>\c_msg_kernel_bug_text_tl</code>	4853 <code>\tl_new:Nn \c_msg_fatal_text_tl {</code>
<code>\c_msg_kernel_bug_more_text_tl</code>	4854 <code>  This-is-a-fatal-error:~LaTeX-will~abort</code>
<code>\c_msg_no_info_text_tl</code>	4855 <code>}</code>
<code>\c_msg_return_text_tl</code>	4856 <code>\tl_new:Nn \c_msg_help_text_tl {</code>
	4857 <code>  For~immediate~help~type~H~&lt;return&gt;</code>
	4858 <code>}</code>
	4859 <code>\tl_new:Nn \c_msg_kernel_bug_text_tl {</code>
	4860 <code>  This-is-a-LaTeX-bug:~check~coding!</code>
	4861 <code>}</code>
	4862 <code>\tl_new:Nn \c_msg_kernel_bug_more_text_tl {</code>
	4863 <code>  There-is-a-coding-bug-somewhere-around-here.</code>
	4864 <code>  \msg_newline:</code>
	4865 <code>  This-probably-needs-examining-by-an-expert.</code>
	4866 <code>  \c_msg_return_text_tl</code>
	4867 <code>}</code>
	4868 <code>\tl_new:Nn \c_msg_no_info_text_tl {</code>
	4869 <code>  LaTeX-does-not-know-anything-more-about-this-error,~sorry.</code>
	4870 <code>  \c_msg_return_text_tl</code>
	4871 <code>}</code>
	4872 <code>\tl_new:Nn \c_msg_return_text_tl {</code>
	4873 <code>  \msg_two_newlines:</code>
	4874 <code>  Try~typing~&lt;return&gt;~to~proceed.</code>
	4875 <code>  \msg_newline:</code>
	4876 <code>  If~that~doesn't~work,~type~X~&lt;return&gt;~to~quit</code>
	4877 <code>}</code>
<code>\c_msg_hide_tl&lt;spaces&gt;</code>	An empty variable with a number of (category code 11) spaces at the end of its name. This is used to push the $\TeX$ part of an error message “off the screen”. No indentation here as <code>_</code> is a letter!
	4878 <code>\group_begin:</code>
	4879 <code>\char_make_letter:N\ %</code>
	4880 <code>\tl_to_lowercase:nf%</code>
	4881 <code>\group_end:%</code>
	4882 <code>\tl_new:Nn%</code>
	4883 <code>\c_msg_hide_tl</code> %
	4884 <code>{}%</code>
	4885 <code>}%</code>
<code>\c_msg_on_line_tl</code>	“On line”.
	4886 <code>\tl_new:Nn \c_msg_on_line_tl { on~line }</code>
<code>\c_msg_text_prefix_tl</code>	Prefixes for storage areas.
<code>\c_msg_more_text_prefix_tl</code>	4887 <code>\tl_new:Nn \c_msg_text_prefix_tl { msg_text // }</code>
<code>\c_msg_code_prefix_tl</code>	4888 <code>\tl_new:Nn \c_msg_more_text_prefix_tl { msg_text_more // }</code>
	4889 <code>\tl_new:Nn \c_msg_code_prefix_tl { msg_code // }</code>
<code>\l_msg_class_tl</code>	For holding the current message method and that for redirection.
<code>\l_msg_current_class_tl</code>	4890 <code>\tl_new:N \l_msg_class_tl</code>
	4891 <code>\tl_new:N \l_msg_current_class_tl</code>

`\l_msg_names_clist` Lists used for filtering.

```
4892 \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`

```
4893 \prop_new:N \l_msg_redirect_classes_prop
```

```
4894 \prop_new:N \l_msg_redirect_names_prop
```

`\l_msg_redirect_classes_clist` To prevent an infinite loop.

```
4895 \clist_new:N \l_msg_redirect_classes_clist
```

## 111.2 Output helper functions

`\msg_line_number:` For writing the line number nicely.

`\msg_line_context:`

```
4896 \cs_new_nopar:Nn { \msg_line_number: } {
```

```
4897   \toks_use:N \tex_inputlineno:D
```

```
4898 }
```

```
4899 \cs_new_nopar:Nn { \msg_line_context: } {
```

```
4900   \msg_space:
```

```
4901   \c_msg_on_line_tl
```

```
4902   \msg_space:
```

```
4903   \msg_line_number:
```

```
4904 }
```

`\msg_newline:` Always forces a new line.

`\msg_two_newlines:`

```
4905 \cs_new_nopar:Nn \msg_newline: { ^^J }
```

```
4906 \cs_new_nopar:Nn \msg_two_newlines: { ^^J ^^J }
```

`\msg_space:` For printing spaces, some very simple functions.

`\msg_two_spaces:`

`\msg_four_spaces:`

```
4907 \cs_new_nopar:Nn \msg_space: { ~ }
```

```
4908 \cs_new_nopar:Nn \msg_two_spaces: { \msg_space: \msg_space: }
```

```
4909 \cs_new_nopar:Nn \msg_four_spaces: { \msg_two_spaces: \msg_two_spaces: }
```

## 111.3 Generic functions

The lowest level functions make no assumptions about modules, *etc.*

`\msg_generic_new:nnnn` Creating a new message is basically the same as the non-checking version, and so after a check everything hands over.

`\msg_generic_new:nnn`

`\msg_generic_new:nn`

```
4910 \cs_new_nopar:Npn \msg_generic_new:nnnn #1 {
```

```
4911   \exp_args:Nc \chk_if_free_cs:N { \c_msg_text_prefix_tl #1 :nn }
```

```
4912   \msg_generic_set:nnnn {#1}
```

```
4913 }
```

```
4914 \cs_new_nopar:Npn \msg_generic_new:nnn #1 {
```

```

4915 \exp_args:Nc \chk_if_free_cs:N { \c_msg_text_prefix_tl #1 :nn }
4916 \msg_generic_set:nnn {#1}
4917 }
4918 \cs_new_nopar:Npn \msg_generic_new:nn #1 {
4919 \exp_args:Nc \chk_if_free_cs:N { \c_msg_text_prefix_tl #1 :nn }
4920 \msg_generic_set:nn {#1}
4921 }

```

\msg\_generic\_set:nnnn Creating a message is quite simple. There must be a short text part, while the other parts  
 \msg\_generic\_set:nnn may not exist. To avoid filling up the hash table with empty functions, only non-empty  
 \msg\_generic\_set:nn arguments are stored. The various auxiliary functions are used to allow spaces in the  
 \msg\_generic\_set\_clist:n text arguments.  
 \msg\_generic\_set\_text:n  
 \msg\_generic\_set\_more\_text:n  
 \msg\_generic\_set\_code:n

```

4922 \cs_new_nopar:Npn \msg_generic_set:nnnn #1 {
4923 \msg_generic_set_clist:n {#1}
4924 \char_make_space:N \ %
4925 \msg_generic_set_code:nnnn{#1}%
4926 }
4927 \cs_new_nopar:Npn \msg_generic_set:nnn #1 {
4928 \msg_generic_set_clist:n {#1}
4929 \char_make_space:N \ %
4930 \msg_generic_set_more_text:nnn{#1}%
4931 }
4932 \cs_new_nopar:Npn \msg_generic_set:nn #1 {
4933 \msg_generic_set_clist:n {#1}
4934 \char_make_space:N \ %
4935 \msg_generic_set_text:nn{#1}%
4936 }
4937 \cs_new_nopar:Npn \msg_generic_set_clist:n #1 {
4938 \clist_if_in:NnF \l_msg_names_clist { // #1 / } {
4939 \clist_put_right:Nn \l_msg_names_clist { // #1 / }
4940 }
4941 }
4942 \cs_new:Nn \msg_generic_set_text:nn {
4943 \cs_set:cn { \c_msg_text_prefix_tl #1 :nn } {#2}
4944 \char_make_ignore:N \
4945 }
4946 \cs_new:Nn \msg_generic_set_more_text:nnn {
4947 \cs_set:cn { \c_msg_text_prefix_tl #1 :nn } {#2}
4948 \tl_if_empty:nTF {#3} {
4949 \cs_set_eq:cn { \c_msg_more_text_prefix_tl #1 } \c_undefined
4950 }{
4951 \cs_set:cn { \c_msg_more_text_prefix_tl #1 :nn } {#3}
4952 }
4953 \char_make_ignore:N \
4954 }
4955 \cs_new:Npn \msg_generic_set_code:nnnn #1#2#3 {
4956 \cs_set:cn { \c_msg_text_prefix_tl #1 :nn } {#2}
4957 \tl_if_empty:nTF {#3} {
4958 \cs_set_eq:cn { \c_msg_more_text_prefix_tl #1 } \c_undefined
4959 }{
4960 \cs_set:cn { \c_msg_more_text_prefix_tl #1 :nn } {#3}
4961 }
4962 \char_make_ignore:N \

```

```

4963 \msg_generic_set_code:nn {#1}
4964 }
4965 \cs_new:Nn \msg_generic_set_code:nn {
4966   \tl_if_empty:nTF {#2} {
4967     \cs_set_eq:cN { \c_msg_code_prefix_tl #1 : } \c_undefined
4968   }{
4969     \cs_set:cn { \c_msg_code_prefix_tl #1 : } {#2}
4970   }
4971 }

```

`\msg_direct_interrupt:xxxxn` The low-level interruption macro is rather opaque, unfortunately. The idea here is to create a message which hides all of T<sub>E</sub>X's own information by filling the output up with spaces. To achieve this, spaces have to be letters: hence no indentation. The odd `\c_msg_hide_tl` actually does the hiding: it is the large run of spaces 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.

```

4972 \group_begin:
4973 \char_set_lccode:nn {'\&} {'\ } % {
4974 \char_set_lccode:w '\} = '\ \scan_stop:
4975 \char_make_active:N \&
4976 \char_make_letter:N\ %
4977 \tl_to_lowercase:n{%
4978 \group_end:%
4979 \cs_new_protected:Nn\msg_direct_interrupt:xxxxn{%
4980 \group_begin:%
4981 \cs_set_eq:NN\ \msg_newline:%
4982 \cs_set_eq:NN\ \msg_space:%
4983 \msg_direct_interrupt_aux:n{#4}%
4984 \cs_set_nopar:Npn\ \{ \msg_newline:#3}%
4985 \tex_errhelp:D\l_msg_tmp_tl%
4986 \cs_set:Npn&{%
4987 \tex_errmessage:D{%
4988 #1\msg_newline:%
4989 #2\msg_two_newlines:%
4990 \c_msg_help_text_tl%
4991 \c_msg_hide_tl                                     %
4992 }%
4993 }%
4994 &%
4995 \group_end:%
4996 #5%
4997 }%
4998 }%
4999 \cs_new:Nn \msg_direct_interrupt_aux:n {
5000   \tl_if_empty:nTF {#1} {
5001     \tl_set:Nx \l_msg_tmp_tl { { \c_msg_no_info_text_tl } }
5002   }{
5003     \tl_set:Nx \l_msg_tmp_tl { {#1 } }
5004   }
5005 }

```

`\msg_direct_log:xx` Printing to the log or terminal without a stop is rather easier.  
`\msg_direct_term:xx`

```

5006 \cs_new_protected:Nn \msg_direct_log:xx {
5007   \group_begin:
5008     \cs_set:Npn \ { \msg_newline: #2 }
5009     \cs_set_eq:NN \ \msg_space:
5010     \iow_log:x { #1 \msg_newline: }
5011   \group_end:
5012 }
5013 \cs_new_protected:Nn \msg_direct_term:xx {
5014   \group_begin:
5015     \cs_set:Npn \ { \msg_newline: #2 }
5016     \cs_set_eq:NN \ \msg_space:
5017     \iow_term:x { #1 \msg_newline: }
5018   \group_end:
5019 }

```

## 111.4 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:nnnnn  For making messages.
\msg_new:nnnn
\msg_new:nnn
\msg_set:nnnnn
\msg_set:nnnn
\msg_set:nnn
5020 \cs_new_nopar:Npn \msg_new:nnnnn #1#2 {
5021   \msg_generic_new:nnnn { #1 / #2 }
5022 }
5023 \cs_new_nopar:Npn \msg_new:nnnn #1#2 {
5024   \msg_generic_new:nnn { #1 / #2 }
5025 }
5026 \cs_new_nopar:Npn \msg_new:nnn #1#2 {
5027   \msg_generic_new:nn { #1 / #2 }
5028 }
5029 \cs_new_nopar:Npn \msg_set:nnnnn #1#2 {
5030   \msg_generic_set:nnnn { #1 / #2 }
5031 }
5032 \cs_new_nopar:Npn \msg_set:nnnn #1#2 {
5033   \msg_generic_set:nnn { #1 / #2 }
5034 }
5035 \cs_new_nopar:Npn \msg_set:nnn #1#2 {
5036   \msg_generic_set:nn { #1 / #2 }
5037 }

```

\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.

```

5038 \cs_new_nopar:Npn \msg_class_new:nn #1 {
5039   \exp_args:Nc \chk_if_free_cs:N { msg_ #1 :nnxx }
5040   \prop_new:c { l_msg_redirect_ #1 _prop }
5041   \msg_class_set:nn {#1}
5042 }
5043 \cs_new_nopar:Nn \msg_class_set:nn {
5044   \prop_clear:c { l_msg_redirect_ #1 _prop }
5045   \cs_set_protected:cn { msg_ #1 :nnxx } {

```

```

5046     \msg_use:nnnnxx {#1} {#2} {##1} {##2} {##3} {##4}
5047 }
5048 \cs_set_protected:cn { msg_ #1 :nnx } {
5049     \use:c { msg_ #1 :nnxx } {##1} {##2} {##3} { }
5050 }
5051 \cs_set_protected:cn { msg_ #1 :nn } {
5052     \use:c { msg_ #1 :nnxx } {##1} {##2} { } { }
5053 }
5054 }

```

`\msg_use:nnnnxx` 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.

```

5055 \cs_new:Nn \msg_use:nnnnxx {
5056     \cs_set:Nn \msg_use_code: {
5057         \clist_clear:N \l_msg_redirect_classes_clist
5058         #2
5059     }
5060     \cs_set:Nn \msg_use_loop:n {
5061         \clist_if_in:NnTF \l_msg_redirect_classes_clist {#1} {
5062             \msg_kernel_error:n { message~loop }
5063         }{
5064             \clist_put_right:Nn \l_msg_redirect_classes_clist {#1}
5065             \cs_if_exist:cTF { msg_ ##1 :nnxx } {
5066                 \use:c { msg_ ##1 :nnxx } {#3} {#4} {#5} {#6}
5067             }{
5068                 \msg_kernel_error:nx { message~class~unknown } { ##1 }
5069             }
5070         }
5071     }
5072     \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 :nn } {
5073         \msg_use_aux:nnn {#1} {#3} {#4}
5074     }{
5075         \msg_kernel_error:nxx { message~unknown } { #3 } { #4 }
5076     }
5077 }

```

`\msg_use_code:` Blank definitions are initially created for these functions.

`\msg_use_loop:`

```

5078 \cs_new_nopar:Nn \msg_use_code: { }
5079 \cs_new_nopar:Nn \msg_use_loop:n { }

```

`\msg_use_aux:nnn` The first auxiliary macro looks for a match by name: the most restrictive check.

```

5080 \cs_new_nopar:Nn \msg_use_aux:nnn {
5081     \tl_set:Nn \l_msg_current_class_tl {#1}
5082     \tl_set:Nn \l_msg_current_module_tl {#2}
5083     \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / } {
5084         \msg_use_loop_check:nn { names } { // #2 / #3 / }
5085     }{
5086         \msg_use_aux:nn {#1} {#2}
5087     }
5088 }

```



`\msg_use_aux:nn` The second function checks for general matches by module or for all modules.

```

5089 \cs_new_nopar:Nn \msg_use_aux:nn {
5090   \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2} {
5091     \msg_use_loop_check:nn {#1} {#2}
5092   }{
5093     \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {*} {
5094       \msg_use_loop_check:nn {#1} {*}
5095     }{
5096       \msg_use_code:
5097     }
5098   }
5099 }

```

`\msg_use_loop_check:nn` When checking whether to loop, the same code is needed in a few places.

```

5100 \cs_new:Nn \msg_use_loop_check:nn {
5101   \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
5102   \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl {
5103     \msg_use_code:
5104   }{
5105     \msg_use_loop:n { \l_msg_class_tl }
5106   }
5107 }

```

`\msg_fatal:nxxx` For fatal errors, after the error message `TeX` bails out.

```

\msg_fatal:nnx
\msg_fatal:nn
5108 \msg_class_new:nn { fatal } {
5109   \msg_direct_interrupt:xxxxn
5110   { \c_msg_fatal_tl \msg_two_newlines: }
5111   {
5112     ( \c_msg_fatal_tl ) \msg_space:
5113     \use:c { \c_msg_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5114   }
5115   { ( \c_msg_fatal_tl ) \msg_space: }
5116   { \c_msg_fatal_text_tl }
5117   { \tex_end:D }
5118 }

```

`\msg_error:nxxx` For an error, the interrupt routine is called, then any recovery code is tried.

```

\msg_error:nnx
\msg_error:nn
5119 \msg_class_new:nn { error } {
5120   \msg_direct_interrupt:xxxxn
5121   { #1~\c_msg_error_tl \msg_newline: }
5122   {
5123     ( #1 ) \msg_space:
5124     \use:c { \c_msg_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5125   }
5126   { ( #1 ) \msg_space: }
5127   {
5128     \cs_if_exist:cTF { \c_msg_more_text_prefix_tl #1 / #2 :nn } {
5129       \use:c { \c_msg_more_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5130     }{
5131       \c_msg_no_info_text_tl

```

```

5132     }
5133   }
5134   {
5135     \cs_if_exist:cT { \c_msg_code_prefix_tl #1 /#2 :nn } {
5136       \use:c { \c_msg_code_prefix_tl #1 / #2 :nn} {#3} {#4}
5137     }
5138   }
5139 }

```

\msg\_warning:nnxx Warnings are printed to the terminal.

```

\msg_warning:nnx
\msg_warning:nn
5140 \msg_class_new:nn { warning } {
5141   \msg_direct_term:xx {
5142     \msg_space: #1~\c_msg_warning_tl :~
5143     \use:c { \c_msg_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5144   }
5145   { ( #1 ) \msg_two_spaces: }
5146 }

```

\msg\_info:nnxx Information only goes into the log.

```

\msg_info:nnx
\msg_info:nn
5147 \msg_class_new:nn { info } {
5148   \msg_direct_log:xx {
5149     \msg_space: #1~\c_msg_info_tl :~
5150     \use:c { \c_msg_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5151   }
5152   { ( #1 ) \msg_two_spaces: }
5153 }

```

\msg\_log:nnxx “Log” data is very similar to information, but with no extras added.

```

\msg_log:nnx
\msg_log:nn
5154 \msg_class_new:nn { log } {
5155   \msg_direct_log:xx {
5156     \use:c { \c_msg_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5157   }
5158   { }
5159 }

```

\msg\_trace:nnxx Trace data is the same as log data, more or less

```

\msg_trace:nnx
\msg_trace:nn
5160 \msg_class_new:nn { trace } {
5161   \msg_direct_log:xx {
5162     \use:c { \c_msg_text_prefix_tl #1 / #2 :nn } {#3} {#4}
5163   }
5164   { }
5165 }

```

\msg\_none:nnxx The none message type is needed so that input can be gobbled.

```

\msg_none:nnx
\msg_none:nn
5166 \msg_class_new:nn { none } { }

```

## 111.5 Redirection functions

`\msg_redirect_class:nn` Converts class one into class two.

```
5167 \cs_new_nopar:Nn \msg_redirect_class:nn {  
5168   \prop_put:cnn { l_msg_redirect_ #1 _prop } {*} {#2}  
5169 }
```

`\msg_redirect_module:nnn` For when all messages of a class should be altered for a given module.

```
5170 \cs_new_nopar:Nn \msg_redirect_module:nnn {  
5171   \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3}  
5172 }
```

`\msg_redirect_name:nnn` Named message will always use the given class.

```
5173 \cs_new_nopar:Nn \msg_redirect_name:nnn {  
5174   \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3}  
5175 }
```

## 111.6 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_new:nn` part.

```
\msg_kernel_set:nnnn  
\msg_kernel_set:nnn  
\msg_kernel_set:nn  
5176 \cs_new_nopar:Npn \msg_kernel_new:nnnn #1 {  
5177   \msg_new:nnnnn { LaTeX } {#1}  
5178 }  
5179 \cs_new_nopar:Npn \msg_kernel_new:nnn #1 {  
5180   \msg_new:nnnn { LaTeX } {#1}  
5181 }  
5182 \cs_new_nopar:Npn \msg_kernel_new:nn #1 {  
5183   \msg_new:nnn { LaTeX } {#1}  
5184 }  
5185 \cs_new_nopar:Npn \msg_kernel_set:nnnn #1 {  
5186   \msg_set:nnnnn { LaTeX } {#1}  
5187 }  
5188 \cs_new_nopar:Npn \msg_kernel_set:nnn #1 {  
5189   \msg_set:nnnn { LaTeX } {#1}  
5190 }  
5191 \cs_new_nopar:Npn \msg_kernel_set:nn #1 {  
5192   \msg_set:nnn { LaTeX } {#1}  
5193 }
```

`\msg_kernel_classes_new:n` Quickly make the fewer-arguments versions.

```
5194 \cs_new_nopar:Nn \msg_kernel_classes_new:n {  
5195   \cs_new_protected:cn { msg_kernel_ #1 :nx } {  
5196     \use:c { msg_kernel_ #1 :nxx } {##1} {##2} { }  
5197   }  
5198   \cs_new_protected:cn { msg_kernel_ #1 :n } {
```

```

5199     \use:c { msg_kernel_ #1 :nxx } {##1} { } { }
5200   }
5201 }

```

`\msg_kernel_fatal:nxx` Fatal kernel errors cannot be re-defined.

```

\msg_kernel_fatal:nx
\msg_kernel_fatal:n
5202 \cs_new_protected:Nn \msg_kernel_fatal:nxx {
5203   \msg_direct_interrupt:xxxxn
5204   { \c_msg_fatal_tl \msg_two_newlines: }
5205   {
5206     ( LaTeX ) \msg_space:
5207     \use:c { \c_msg_text_prefix_tl LaTeX / #1 :nn } {#2} {#3}
5208   }
5209   { ( LaTeX ) \msg_space: }
5210   { \c_msg_fatal_text_tl }
5211   { \tex_end:D }
5212 }
5213 \msg_kernel_classes_new:n { fatal }

```

`\msg_kernel_error:nxx` Neither can kernel errors.

```

\msg_kernel_error:nx
\msg_kernel_error:n
5214 \cs_new_protected:Nn \msg_kernel_error:nxx {
5215   \msg_direct_interrupt:xxxxn
5216   { LaTeX~\c_msg_error_tl \msg_newline: }
5217   {
5218     ( LaTeX ) \msg_space:
5219     \use:c { \c_msg_text_prefix_tl LaTeX / #1 :nn } {#2} {#3}
5220   }
5221   { ( LaTeX ) \msg_space: }
5222   {
5223     \cs_if_exist:cTF { \c_msg_more_text_prefix_tl LaTeX / #1 :nn } {
5224       \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 :nn } {#2} {#3}
5225     }{
5226       \c_msg_no_info_text_tl
5227     }
5228   }
5229   {
5230     \cs_if_exist:cT { \c_msg_code_prefix_tl LaTeX / #1 :nn } {
5231       \use:c { \c_msg_code_prefix_tl LaTeX / #1 :nn } {#2} {#3}
5232     }
5233   }
5234 }
5235 \msg_kernel_classes_new:n { error }

```

`\msg_kernel_warning:nxx` Life is much more simple for warnings and information messages, as these are just short-cuts to the standard classes.

```

\msg_kernel_warning:nx
\msg_kernel_warning:n
\msg_kernel_info:nxx
\msg_kernel_info:nx
\msg_kernel_info:n
5236 \cs_new_protected_nopar:Npn \msg_kernel_warning:nxx {
5237   \msg_warning:nnxx { LaTeX }
5238 }
5239 \msg_kernel_classes_new:n { warning }
5240 \cs_new_protected_nopar:Npn \msg_kernel_info:nxx {
5241   \msg_info:nnxx { LaTeX }
5242 }
5243 \msg_kernel_classes_new:n { info }

```

Some very basic error messages.

```

5244 \msg_kernel_new:nnn
5245 { coding~bug }
5246 {This is a LaTeX bug: check coding!\#1}
5247 {#2}
5248 \msg_kernel_new:nnn
5249 { message~unknown }
5250 {Unknown message ‘#2’ for module ‘#1’..}
5251 {LaTeX was asked to display a message by the ‘#1’ module.\
5252   The message was supposed to be called ‘#2’, but I can’t\
5253   find a message with that name.
5254   \c_msg_return_text_tl}
5255 \msg_kernel_new:nnn
5256 { message~class~unknown }
5257 {Unknown message class ‘#1’..}
5258 {You have asked for a message to be redirected to class ‘#1’\
5259   but this class is unknown.
5260   \c_msg_return_text_tl}
5261 \msg_kernel_new:nnn
5262 { message~loop }
5263 {Message redirection loop.}
5264 {You have asked for a message to be redirected,\
5265   but the redirection instructions form a loop:\
5266   you’ve lost the message.
5267   \c_msg_return_text_tl}

```

`\msg_kernel_bug:x` The L<sup>A</sup>T<sub>E</sub>X coding bug error gets re-visited here.

```

5268 \cs_set_protected:Nn \msg_kernel_bug:x {
5269   \msg_direct_interrupt:xxxxn
5270   { \c_msg_kernel_bug_text_tl }
5271   { !~#1 }
5272   { ! }
5273   { \c_msg_kernel_bug_more_text_tl }
5274   { }
5275 }
5276 </initex | package>

```

## 112 l3box implementation

Announce and ensure that the required packages are loaded.

```

5277 <*package>
5278 \ProvidesExplPackage
5279   {\filename}{\filedate}{\fileversion}{\filedescription}
5280 \package_check_loaded_expl:
5281 </package>
5282 <*initex | package>

```

The code in this module is very straight forward so I’m not going to comment it very extensively.

## 112.1 Generic boxes

`\box_new:N` Defining a new  $\langle box \rangle$  register.

`\box_new:c`

```
5283 \*initex)
5284 \alloc_setup_type:nnn {box} \c_zero \c_max_register_num
```

Now, remember that `\box255` has a special role in T<sub>E</sub>X, it shouldn't be allocated. . .

```
5285 \seq_put_right:Nn \g_box_allocation_seq {255}
5286 \cs_new_nopar:Npn \box_new:N #1 {\alloc_reg:NnNN g {box} \tex_mathchardef:D #1}
5287 \cs_new_nopar:Npn \box_new_l:N #1 {\alloc_reg:NnNN l {box} \tex_mathchardef:D #1}
5288 \/initex)
```

When we run on top of L<sup>A</sup>T<sub>E</sub>X, we just use its allocation mechanism.

```
5289 \*package)
5290 \cs_new:Npn \box_new:N #1 {
5291   \chk_if_free_cs:N #1
5292   \newbox #1
5293 }
5294 \package)

5295 \cs_generate_variant:Nn \box_new:N {c}
```

`\if_hbox:N` The primitives for testing if a  $\langle box \rangle$  is empty/void or which type of box it is.

`\if_vbox:N`

`\if_box_empty:N`

```
5296 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
5297 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
5298 \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`

```
5299 \prg_new_conditional:Nnn \box_if_horizontal:N {p,TF,T,F} {
5300   \tex_ifhbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
5301 }
5302 \prg_new_conditional:Nnn \box_if_vertical:N {p,TF,T,F} {
5303   \tex_ifvbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
5304 }
5305 \cs_generate_variant:Nn \box_if_horizontal_p:N {c}
5306 \cs_generate_variant:Nn \box_if_horizontal:N {NTF} {c}
5307 \cs_generate_variant:Nn \box_if_horizontal:NT {c}
5308 \cs_generate_variant:Nn \box_if_horizontal:NF {c}
5309 \cs_generate_variant:Nn \box_if_vertical_p:N {c}
5310 \cs_generate_variant:Nn \box_if_vertical:N {NTF} {c}
5311 \cs_generate_variant:Nn \box_if_vertical:NT {c}
5312 \cs_generate_variant:Nn \box_if_vertical:NF {c}
```

`\box_if_empty_p:N`

`\box_if_empty_p:c`

`\box_if_empty:NTF`

`\box_if_empty:cTF`

Testing if a  $\langle box \rangle$  is empty/void.

```
5313 \prg_new_conditional:Nnn \box_if_empty:N {p,TF,T,F} {
5314   \tex_ifvoid:D #1 \prg_return_true: \else: \prg_return_false: \fi:
5315 }
5316 \cs_generate_variant:Nn \box_if_empty_p:N {c}
5317 \cs_generate_variant:Nn \box_if_empty:N {NTF} {c}
5318 \cs_generate_variant:Nn \box_if_empty:NT {c}
5319 \cs_generate_variant:Nn \box_if_empty:NF {c}
```

<code>\box_set_eq:NN</code> <code>\box_set_eq:cN</code> <code>\box_set_eq:Nc</code> <code>\box_set_eq:cc</code>	Assigning the contents of a box to be another box. This clears the second box globally (that's how $\text{\TeX}$ does it).  <small>5320</small> <code>\cs_new_nopar:Npn \box_set_eq:NN #1#2 {\tex_setbox:D #1 \tex_box:D #2}</code> <small>5321</small> <code>\cs_generate_variant:Nn \box_set_eq:NN {cN,Nc,cc}</code>
<code>\box_gset_eq:NN</code> <code>\box_gset_eq:cN</code> <code>\box_gset_eq:Nc</code> <code>\box_gset_eq:cc</code>	Global version of the above.  <small>5322</small> <code>\cs_new_nopar:Npn \box_gset_eq:NN {\pref_global:D\box_set_eq:NN}</code> <small>5323</small> <code>\cs_generate_variant:Nn \box_gset_eq:NN {cN,Nc,cc}</code>
<code>\l_last_box</code>	A different name for this read-only primitive.  <small>5324</small> <code>\cs_new_eq:NN \l_last_box \tex_lastbox:D</code>
<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>	Set a box to the previous box.  <small>5325</small> <code>\cs_new_nopar:Npn \box_set_to_last:N #1{\tex_setbox:D#1\l_last_box}</code> <small>5326</small> <code>\cs_generate_variant:Nn \box_set_to_last:N {c}</code> <small>5327</small> <code>\cs_new_nopar:Npn \box_gset_to_last:N {\pref_global:D \box_set_to_last:N}</code> <small>5328</small> <code>\cs_generate_variant:Nn \box_gset_to_last:N {c}</code>
<code>\box_move_left:nn</code> <code>\box_move_right:nn</code> <code>\box_move_up:nn</code> <code>\box_move_down:nn</code>	Move box material in different directions.  <small>5329</small> <code>\cs_new:Npn \box_move_left:nn #1#2{\tex_moveleft:D\dim_eval:n{#1}{#2}}</code> <small>5330</small> <code>\cs_new:Npn \box_move_right:nn #1#2{\tex_moveright:D\dim_eval:n{#1}{#2}}</code> <small>5331</small> <code>\cs_new:Npn \box_move_up:nn #1#2{\tex_raise:D\dim_eval:n{#1}{#2}}</code> <small>5332</small> <code>\cs_new:Npn \box_move_down:nn #1#2{\tex_lower:D\dim_eval:n{#1}{#2}}</code>
<code>\box_clear:N</code> <code>\box_clear:c</code> <code>\box_gclear:N</code> <code>\box_gclear:c</code>	Clear a $\langle box \rangle$ register.  <small>5333</small> <code>\cs_new_nopar:Npn \box_clear:N #1{\box_set_eq:NN #1 \c_empty_box }</code> <small>5334</small> <code>\cs_generate_variant:Nn \box_clear:N {c}</code> <small>5335</small> <code>\cs_new_nopar:Npn \box_gclear:N {\pref_global:D\box_clear:N}</code> <small>5336</small> <code>\cs_generate_variant:Nn \box_gclear:N {c}</code>
<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>	Accessing the height, depth, and width of a $\langle box \rangle$ register.  <small>5337</small> <code>\cs_new_eq:NN \box_ht:N \tex_ht:D</code> <small>5338</small> <code>\cs_new_eq:NN \box_dp:N \tex_dp:D</code> <small>5339</small> <code>\cs_new_eq:NN \box_wd:N \tex_wd:D</code> <small>5340</small> <code>\cs_generate_variant:Nn \box_ht:N {c}</code> <small>5341</small> <code>\cs_generate_variant:Nn \box_dp:N {c}</code> <small>5342</small> <code>\cs_generate_variant:Nn \box_wd:N {c}</code>
<code>\box_use_clear:N</code> <code>\box_use_clear:c</code> <code>\box_use:N</code> <code>\box_use:c</code>	Using a $\langle box \rangle$ . These are just $\text{\TeX}$ primitives with meaningful names.  <small>5343</small> <code>\cs_new_eq:NN \box_use_clear:N \tex_box:D</code> <small>5344</small> <code>\cs_generate_variant:Nn \box_use_clear:N {c}</code> <small>5345</small> <code>\cs_new_eq:NN \box_use:N \tex_copy:D</code> <small>5346</small> <code>\cs_generate_variant:Nn \box_use:N {c}</code>

`\box_show:N` Show the contents of a box and write it into the log file.

`\box_show:c`

```

5347 \cs_set_eq:NN \box_show:N \tex_showbox:D
5348 \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`

```

5349 <package>\cs_set_eq:NN \c_empty_box \voidb@x
5350 <package>\cs_new_eq:NN \l_tmpa_box \@tempboxa
5351 <initex>\box_new:N \c_empty_box
5352 <initex>\box_new:N \l_tmpa_box
5353 \box_new:N \l_tmpb_box

```

## 112.2 Vertical boxes

`\vbox:n` Put a vertical box directly into the input stream.

```

5354 \cs_new_nopar:Npn \vbox:n {\tex_vbox: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`

```

5355 \cs_new:Npn \vbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_vbox:D {#2}}
5356 \cs_generate_variant:Nn \vbox_set:Nn {cn}
5357 \cs_new_nopar:Npn \vbox_gset:Nn {\pref_global:D \vbox_set:Nn}
5358 \cs_generate_variant:Nn \vbox_gset:Nn {cn}

```

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.

`\vbox_set_to_ht:cnn`

`\vbox_gset_to_ht:Nnn`

`\vbox_gset_to_ht:cnn`

`\vbox_gset_to_ht:ccn`

```

5359 \cs_new:Npn \vbox_set_to_ht:Nnn #1#2#3 {
5360   \tex_setbox:D #1 \tex_vbox:D to #2 {#3}
5361 }
5362 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn {cnn}
5363 \cs_new_nopar:Npn \vbox_gset_to_ht:Nnn {\pref_global:D \vbox_set_to_ht:Nnn }
5364 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn {cnn,ccn}

```

`\vbox_set_inline_begin:N` Storing material in a vertical box. This type is useful in environment definitions.

`\vbox_set_inline_end:`

`\vbox_gset_inline_begin:N`

`\vbox_gset_inline_end:`

```

5365 \cs_new_nopar:Npn \vbox_set_inline_begin:N #1 {
5366   \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
5367 \cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token
5368 \cs_new_nopar:Npn \vbox_gset_inline_begin:N {
5369   \pref_global:D \vbox_set_inline_begin:N }
5370 \cs_new_eq:NN \vbox_gset_inline_end: \c_group_end_token

```

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.

`\vbox_to_zero:n`

```

5371 \cs_new:Npn \vbox_to_ht:nn #1#2{\tex_vbox:D to \dim_eval:n{#1}{#2}}
5372 \cs_new:Npn \vbox_to_zero:n #1 {\tex_vbox:D to \c_zero_dim {#1}}

```

`\vbox_set_split_to_ht:Nnn` Splitting a vertical box in two.

```

5373 \cs_new_nopar:Npn \vbox_set_split_to_ht:Nnn #1#2#3{
5374   \tex_setbox:D #1 \tex_vsplit:D #2 to #3
5375 }

```



<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>	5376 <code>\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D</code>
<code>\vbox_unpack_clear:c</code>	5377 <code>\cs_generate_variant:Nn \vbox_unpack:N {c}</code>
	5378 <code>\cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D</code>
	5379 <code>\cs_generate_variant:Nn \vbox_unpack_clear:N {c}</code>

## 112.3 Horizontal boxes

`\hbox:n` Put a horizontal box directly into the input stream.

5380 `\cs_new_nopar:Npn \hbox:n {\tex_hbox:D \scan_stop:}`

`\hbox_set:Nn` 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).

`\hbox_set:cn`

`\hbox_gset:Nn`

`\hbox_gset:cn`

5381 `\cs_new:Npn \hbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_hbox:D {#2}}`  
5382 `\cs_generate_variant:Nn \hbox_set:Nn {cn}`  
5383 `\cs_new_nopar:Npn \hbox_gset:Nn {\pref_global:D \hbox_set:Nn}`  
5384 `\cs_generate_variant:Nn \hbox_gset:Nn {cn}`

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width.

`\hbox_set_to_wd:cnn`

`\hbox_gset_to_wd:Nnn`

`\hbox_gset_to_wd:cnn`

5385 `\cs_new:Npn \hbox_set_to_wd:Nnn #1#2#3 {`  
5386 `\tex_setbox:D #1 \tex_hbox:D to \dim_eval:n{#2} {#3}`  
5387 `}`  
5388 `\cs_generate_variant:Nn \hbox_set_to_wd:Nnn {cnn}`  
5389 `\cs_new_nopar:Npn \hbox_gset_to_wd:Nnn {\pref_global:D \hbox_set_to_wd:Nnn }`  
5390 `\cs_generate_variant:Nn \hbox_gset_to_wd:Nnn {cnn}`

`\hbox_set_inline_begin:N` Storing material in a horizontal box. This type is useful in environment definitions.

`\hbox_set_inline_begin:c`

`\hbox_set_inline_end:`

`\hbox_gset_inline_begin:N`

`\hbox_gset_inline_begin:c`

`\hbox_gset_inline_end:`

5391 `\cs_new_nopar:Npn \hbox_set_inline_begin:N #1 {`  
5392 `\tex_setbox:D #1 \tex_hbox:D \c_group_begin_token`  
5393 `}`  
5394 `\cs_generate_variant:Nn \hbox_set_inline_begin:N {c}`  
5395 `\cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token`  
5396 `\cs_new_nopar:Npn \hbox_gset_inline_begin:N {`  
5397 `\pref_global:D \hbox_set_inline_begin:N`  
5398 `}`  
5399 `\cs_generate_variant:Nn \hbox_gset_inline_begin:N {c}`  
5400 `\cs_new_eq:NN \hbox_gset_inline_end: \c_group_end_token`

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

`\hbox_to_zero:n`

5401 `\cs_new:Npn \hbox_to_wd:nn #1#2 {\tex_hbox:D to #1 {#2}}`  
5402 `\cs_new:Npn \hbox_to_zero:n #1 {\tex_hbox:D to \c_zero_skip {#1}}`

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

`\hbox_unpack:c`

`\hbox_unpack_clear:N`

`\hbox_unpack_clear:c`

5403 `\cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D`  
5404 `\cs_generate_variant:Nn \hbox_unpack:N {c}`  
5405 `\cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D`  
5406 `\cs_generate_variant:Nn \hbox_unpack_clear:N {c}`

```

5407 </initex | package>

5408 <*showmemory>
5409 \showMemUsage
5410 </showmemory>

```

## 113 l3xref implementation

### 113.1 Internal functions and variables

<pre> \g_xref_all_curr_immediate_fields_prop \g_xref_all_curr_deferred_fields_prop </pre>	What they say they are :)
---	---------------------------

<code>\xref_write</code>	A stream for writing cross references, although they are not required to be in a separate file.
--------------------------	---

<code>\xref_define_label:nn</code>	<code>\xref_define_label:nn {&lt;name&gt;} {&lt;plist contents&gt;}</code>
------------------------------------	--

Define the property list for each label; used internally by `\xref_set_label:n`.

### 113.2 Module code

We start by ensuring that the required packages are loaded.

```

5411 <*package>
5412 \ProvidesExplPackage
5413   {\filename}{\filedate}{\fileversion}{\filedescription}
5414 \package_check_loaded_expl:
5415 </package>
5416 <*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>_t1}` 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` The two main property lists for storing information. They contain key-info pairs for all  
`all_curr_deferred_fields_prop` known types.

```
5417 \prop_new:N \g_xref_all_curr_immediate_fields_prop
5418 \prop_new:N \g_xref_all_curr_deferred_fields_prop
```

`\xref_new:nn` Setting up a new cross reference type is fairly straight forward when we follow the game  
`\xref_deferred_new:nn` plan mentioned earlier.

```
\xref_new_aux:nnn
5419 \cs_new_nopar:Npn \xref_new:nn {\xref_new_aux:nnn{immediate}}
5420 \cs_new_nopar:Npn \xref_deferred_new:nn {\xref_new_aux:nnn{deferred}}
5421 \cs_new_nopar:Npn \xref_new_aux:nnn #1#2#3{
```

First put the new type in the relevant property list.

```
5422 \prop_gput:ccx {g_xref_all_curr_ #1 _fields_prop}
5423 { xref_ #2 _key }
5424 { \exp_not:c {l_xref_curr_#2_tl} }
```

Then define the key to be a protected macro.<sup>14</sup>

```
5425 \cs_set_protected_nopar:cpn { xref_#2_key }{}
5426 \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}
```

```
5427 \toks_set:Nx \l_tmpa_toks {
5428   \exp_not:n { \cs_set_nopar:cpn {xref_get_value_#2_aux:w} ##1 }
5429   \exp_not:N \q_prop
5430   \exp_not:c { xref_#2_key }
5431   \exp_not:N \q_prop
5432 }
5433 \toks_use:N \l_tmpa_toks ##2 ##3\q_nil {##2}
5434 }
```

`\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.

```
5435 \cs_new_nopar:Npn \xref_get_value:nn #1#2 {
5436   \cs_if_exist:cTF{g_xref_#2_prop}
5437   {
```

---

<sup>14</sup>We could also set it equal to `\scan_stop:` but this just feels “cleaner”.

This next expansion may look a little weird but it isn't if you think about it!

```
5438 \exp_args:NcNc \exp_after:wN {xref_get_value_#1_aux:w}
5439 \toks_use:N {g_xref_#2_prop}
```

Better put in the stop marker.

```
5440 \q_nil
5441 }
5442 {??}
5443 }
```

Temporary! We expand the property list and so we can't have the `\q_prop` marker just expand!

```
5444 \cs_set_nopar:Npn \exp_after:cc #1#2 {
5445 \exp_after:wN \exp_after:wN
5446 \cs:w #1\exp_after:wN\cs_end: \cs:w #2\cs_end:
5447 }
5448 \cs_set_protected:Npn \q_prop {\q_prop}
```

`\xref_define_label:nn`  
`\xref_define_label_aux: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`.

```
5449 \cs_new_protected_nopar:Npn \xref_define_label:nn {
5450 \group_begin:
5451 \char_set_catcode:nn {'\ }\c_ten
5452 \xref_define_label_aux:nn
5453 }
```

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.

```
5454 \cs_new_nopar:Npn \xref_define_label_aux:nn #1#2 {
5455 \cs_if_free:cTF{g_xref_#1_prop}
5456 {\prop_new:c{g_xref_#1_prop}}{\WARNING}
5457 \toks_gset:cn{g_xref_#1_prop}{#2}
5458 \group_end:
5459 }
```

`\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.

```
5460 \cs_set_nopar:Npn \xref_set_label:n #1{
5461 \cs_set_nopar:Npx \xref_tmp:w{\toks_use:N\g_xref_all_curr_immediate_fields_prop}
5462 \exp_args:NNx\iow_shipout_x:Nn \xref_write{
5463 \xref_define_label:nn {#1} {
5464 \xref_tmp:w
5465 \toks_use:N \g_xref_all_curr_deferred_fields_prop
```

```

5466     }
5467   }
5468 }

```

`\xref_write` A stream for writing cross references although they do not require to be in a separate file.

```

5469 \iow_new:N \xref_write

```

That's it (for now).

```

5470 </initex | package>

```

```

5471 <*showmemory>
5472 \showMemUsage
5473 </showmemory>

```

## 114 l3xref test file

```

5474 <*testfile>
5475 \documentclass{article}
5476 \usepackage{l3xref}
5477 \ExplSyntaxOn
5478 \cs_set_nopar:Npn \startrecording {\iow_open:Nn \xref_write {\jobname.xref}}
5479 \cs_set_nopar:Npn \DefineCrossReferences {
5480   \group_begin:
5481     \ExplSyntaxNamesOn
5482     \InputIfFileExists{\jobname.xref}{\}{}
5483   \group_end:
5484 }
5485 \AtBeginDocument{\DefineCrossReferences\startrecording}
5486
5487 \xref_new:nn {name}{}
5488 \cs_set_nopar:Npn \setname{\tl_set:Nn\l_xref_curr_name_tl}
5489 \cs_set_nopar:Npn \getname{\xref_get_value:nn{name}}
5490
5491 \xref_deferred_new:nn {page}{\thepage}
5492 \cs_set_nopar:Npn \getpage{\xref_get_value:nn{page}}
5493
5494 \xref_deferred_new:nn {valuepage}{\number\value{page}}
5495 \cs_set_nopar:Npn \getvaluepage{\xref_get_value:nn{valuepage}}
5496
5497 \cs_set_eq:NN \setlabel \xref_set_label:n
5498
5499 \ExplSyntaxOff
5500 \begin{document}
5501 \pagenumbering{roman}
5502
5503 Text\setname{This is a name}\setlabel{testlabel1}. More
5504 text\setname{This is another name}\setlabel{testlabel2}. \clearpage
5505
5506 Text\setname{This is a third name}\setlabel{testlabel3}. More

```

```

5507 text\setname{Hello World!}\setlabel{testlabel4}. \clearpage
5508
5509 \pagenumbering{arabic}
5510
5511 Text\setname{Name 5}\setlabel{testlabel5}. More text\setname{Name
5512 6}\setlabel{testlabel6}. \clearpage
5513
5514 Text\setname{Name 7}\setlabel{testlabel 7}. More text\setname{Name
5515 8}\setlabel{testlabel8}. \clearpage
5516
5517 Now let's extract some values. \getname{testlabel1} on page
5518 \getpage{testlabel1} with value \getvaluepage{testlabel1}.
5519
5520 Now let's extract some values. \getname{testlabel 7} on page
5521 \getpage{testlabel 7} with value \getvaluepage{testlabel 7}.
5522 \end{document}
5523 </testfile>

```

## 115 l3keyval implementation

### 115.1 Internal functions and variables

```

\l_KV_tmpa_tl
\l_KV_tmpp_tl
\c_KV_single_equal_sign_tl
\l_KV_parse_toks
\l_KV_currkey_toks
\l_KV_currval_toks

```

Token list variables and token registers used internally.

```

\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  $\langle tl\ var.\rangle$  with active characters.

```

\KV_remove_surrounding_spaces:nw
\KV_remove_surrounding_spaces_auxi:w ★ \KV_remove_surrounding_spaces:nw <toks> <token list> \
\KV_remove_surrounding_spaces_auxi:w <token list> \Q3

```

Removes a possible leading space plus a possible ending space from a  $\langle token\ list\rangle$ . The first version (which is not used in the code) stores it in  $\langle toks\rangle$ .

```

\KV_add_value_element:w \KV_set_key_element:w <token list> \q_nil
\KV_set_key_element:w \KV_add_value_element:w \q_stop <token list> \q_nil

```

Specialised functions to strip spaces from their input and set the token registers  $\backslash l\_KV\_currkey\_toks$  or  $\backslash l\_KV\_currval\_toks$  respectively.

```

\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

```

\KV\_split\_key\_value\_current:w ...

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.

## 115.2 Module code

We start by ensuring that the required packages are loaded.

```

5524 <*package>
5525 \ProvidesExplPackage
5526   {\filename}{\filedate}{\fileversion}{\filedescription}
5527 \package_check_loaded_expl:
5528 </package>
5529 <*initex | package>

```

\l\_KV\_tmpa\_tl Various useful things.

\l\_KV\_tmpl\_tl

\c\_KV\_single\_equal\_sign\_tl

```

5530 \tl_new:N \l_KV_tmpa_tl
5531 \tl_new:N \l_KV_tmpl_tl
5532 \tl_new:Nn \c_KV_single_equal_sign_tl{=}

```

\l\_KV\_parse\_toks Some more useful things.

\l\_KV\_currkey\_toks

\l\_KV\_currval\_toks

```

5533 \toks_new:N \l_KV_parse_toks
5534 \toks_new:N \l_KV_currkey_toks
5535 \toks_new:N \l_KV_currval_toks

```

move\_one\_level\_of\_braces\_bool A boolean to control

```

5536 \bool_new:N \l_KV_remove_one_level_of_braces_bool
5537 \bool_set_true:N \l_KV_remove_one_level_of_braces_bool

```

ze\_outerlevel\_active\_equals:N

ze\_outerlevel\_active\_commas:N

Some functions for sanitizing top level equals and commas. Replace =<sub>13</sub> and ,<sub>13</sub> with =<sub>12</sub> and ,<sub>12</sub> resp.

```

5538 \group_begin:
5539 \char_set_catcode:nn{'\={13}
5540 \char_set_catcode:nn{'\,}{13}
5541 \char_set_lccode:nn{'\8}{'\={}
5542 \char_set_lccode:nn{'\9}{'\,}
5543 \tl_to_lowercase:n{\group_end:
5544 \cs_new_nopar:Npn \KV_sanitize_outerlevel_active_equals:N #1{
5545   \tl_replace_all_in:Nnn #1 = 8
5546 }
5547 \cs_new_nopar:Npn \KV_sanitize_outerlevel_active_commas:N #1{
5548   \tl_replace_all_in:Nnn #1 , 9
5549 }
5550 }

```

```

\remove_surrounding_spaces:nw
\ove_surrounding_spaces_auxi:w
\ve_surrounding_spaces_auxii:w
\KV_set_key_element:w
\KV_add_value_element:w

```

The macro `\KV_remove_surrounding_spaces:nw` removes a possible leading space plus a possible ending space from its second argument and stores it in the token register `#1`.

Based on Around the Bend No. 15 but with some enhancements. For instance, this definition is purely expandable.

We use a funny token `Q3` as a delimiter.

```

5551 \group_begin:
5552 \char_set_catcode:nn{'\Q}{3}

5553 \cs_gnew: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 `\toks_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.

```

5554 #1{\KV_remove_surrounding_spaces_auxi:w \exp_not:N#2Q~Q}
5555 }

```

`\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.

```

5556 \cs_gnew:Npn\KV_remove_surrounding_spaces_auxi:w#1~Q{
5557   \KV_remove_surrounding_spaces_auxii:w #1 Q
5558 }

```

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`.

```

5559 \cs_gnew: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_toks`.



```

5560 \cs_gnew:Npn\KV_add_value_element:w\q_stop#1\q_nil{
5561   \toks_set:Nf\l_KV_currval_toks {
5562     \KV_remove_surrounding_spaces_auxi:w \exp_not:N#1Q~Q
5563   }
5564   \toks_put_right:No\l_KV_parse_toks{
5565     \exp_after:wN {\toks_use:N \l_KV_currval_toks}
5566   }
5567 }

```

When storing the key we firstly remove spaces plus the prepended \q\_no\_value.

```

5568 \cs_gnew:Npn\KV_set_key_element:w#1\q_nil{
5569   \toks_set:Nf\l_KV_currkey_toks
5570   {
5571     \exp_last_unbraced:NNo \KV_remove_surrounding_spaces_auxi:w
5572     \exp_not:N \use_none:n #1Q~Q
5573   }

```

Afterwards we gobble an extra level of braces if that's what we are asked to do.

```

5574   \bool_if:NT \l_KV_remove_one_level_of_braces_bool
5575   {
5576     \exp_args:NNo \toks_set:No \l_KV_currkey_toks {
5577       \exp_after:wN \KV_add_element_aux:w
5578       \toks_use:N \l_KV_currkey_toks \q_nil
5579     }
5580   }
5581 }
5582 \group_end:

```

\KV\_add\_element\_aux:w A helper function for fixing braces around keys and values.

```

5583 \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.

```

5584 \cs_new:Npn \KV_parse_sanitize_aux:n #1 {
5585   \group_begin:
5586   \toks_clear:N \l_KV_parse_toks
5587   \tl_set:Nx \l_KV_tmpa_tl { \exp_not:n {#1} }
5588   \KV_sanitize_outerlevel_active_equals:N \l_KV_tmpa_tl
5589   \KV_sanitize_outerlevel_active_commas:N \l_KV_tmpa_tl
5590   \exp_last_unbraced:NNV \KV_parse_elt:w \q_no_value
5591   \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.

```

5592   \exp_last_unbraced:NV \group_end:
5593   \l_KV_parse_toks
5594 }

```

`\KV_parse_no_sanitiz`: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!

```

5595 \cs_new:Npn \KV_parse_no_sanitiz_aux:n #1{
5596   \group_begin:
5597     \toks_clear:N \l_KV_parse_toks
5598     \KV_parse_elt:w \q_no_value #1 , \q_nil ,
5599     \exp_last_unbraced:N \group_end:
5600     \l_KV_parse_toks
5601 }

```

`\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.

```

5602 \cs_set:Npn \KV_parse_elt:w #1,{
5603   \tl_if_blank:oTF{\use_none:n #1}
5604   { \KV_parse_elt:w \q_no_value }
5605   {
5606     \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.

```

5607   {
5608     \KV_split_key_value_current:w #1==\q_nil
5609     \KV_parse_elt:w \q_no_value
5610   }
5611 }
5612 }

```

`\KV_split_key_value_current:w` The function called to split the keys and values.

```

5613 \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.

`\KV_split_key_value_space_removal:w` The code below removes extraneous spaces around the keys and values plus one set of braces around the entire value.

`\KV_split_key_value_space_removal_detect_error:wTF`

`\KV_split_key_value_space_removal_aux:w`

Unlike the version to be used when spaces are ignored, this one only grabs the key which is everything up to the first = and save the rest for closer inspection. Reason is that if a user has entered `mykey={{myval}}`, then the outer braces have already been removed before we even look at what might come after the key. So this is slightly more tedious (but only slightly) but at least it always removes only one level of braces.

```

5614 \cs_new:Npn \KV_split_key_value_space_removal:w #1 = #2\q_nil{

```

First grab the key.

```
5615 \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.

```
5616 \tl_set:Nx\l_KV_tmpa_tl{\exp_not:n{#2}}
5617 \tl_if_eq:NNTF\l_KV_tmpa_tl\c_KV_single_equal_sign_tl
```

Then we just insert the default key.

```
5618 {
5619   \toks_put_right:No\l_KV_parse_toks{
5620     \exp_after:wN \KV_key_no_value_elt:n
5621     \exp_after:wN {\toks_use:N\l_KV_currkey_toks}
5622   }
5623 }
```

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.

```
5624 {
5625   \KV_split_key_value_space_removal_detect_error:wTF#2\q_no_value\q_nil
5626   {\KV_split_key_value_space_removal_aux:w \q_stop #2}
5627   {\ERROR}
5628 }
5629 }
```

The error test.

```
5630 \cs_new:Npn
5631 \KV_split_key_value_space_removal_detect_error:wTF#1=#2#3\q_nil{
5632   \tl_if_head_eq_meaning:nNTF{#3}\q_no_value
5633 }
```

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.

```
5634 \cs_new:Npn \KV_val_preserve_braces:NnN #1#2#3{{#2}}
5635 \cs_new:Npn \KV_split_key_value_space_removal_aux:w #1=={
5636   \tl_set:Nx\l_KV_tmpa_tl{\exp_not:o{\use_none:nnn#1\q_nil\q_nil}}
5637   \toks_put_right:No\l_KV_parse_toks{
5638     \exp_after:wN \KV_key_value_elt:nn
5639     \exp_after:wN {\toks_use:N\l_KV_currkey_toks}
5640   }
```

If there a blank space or nothing at all, \l\_KV\_tmpa\_tl is now completely empty.

```
5641 \tl_if_empty:NNTF\l_KV_tmpa_tl
```

We just put an empty value on the stack.

```

5642 { \toks_put_right:Nn\l_KV_parse_toks{}} }
5643 {

```

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.

```

5644 \quark_if_nil:NTF\l_KV_tmpa_tl
5645 {
5646   \bool_if:NTF \l_KV_remove_one_level_of_braces_bool
5647   {
5648     \toks_put_right:No\l_KV_parse_toks{
5649       \exp_after:wN{\use_ii:nnn #1\q_nil}
5650     }
5651   }
5652   {
5653     \toks_put_right:No\l_KV_parse_toks{
5654       \exp_after:wN{\KV_val_preserve_braces:NnN #1\q_nil}
5655     }
5656   }
5657 }

```

Otherwise we grab the value.

```

5658 { \KV_add_value_element:w #1\q_nil }
5659 }
5660 }

```

`_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  $\text{\TeX}$ .

```

5661 \cs_new:Npn \KV_split_key_value_no_space_removal:w #1#2=#3=#4\q_nil{
5662   \tl_set:Nn\l_KV_tmpa_tl{#4}
5663   \tl_if_empty:NTF \l_KV_tmpa_tl
5664   {
5665     \toks_put_right:Nn\l_KV_parse_toks{\KV_key_no_value_elt:n{#2}}
5666   }
5667   {
5668     \tl_if_eq:NNTF\c_KV_single_equal_sign_tl\l_KV_tmpa_tl
5669     {
5670       \toks_put_right:Nn\l_KV_parse_toks{\KV_key_value_elt:nn{#2}{#3}}
5671     }
5672     {\ERROR}
5673   }
5674 }

```

```

\KV_key_no_value_elt:n
\KV_key_value_elt:nn

```

```

5675 \cs_new:Npn \KV_key_no_value_elt:n #1{\ERROR}
5676 \cs_new:Npn \KV_key_value_elt:nn #1#2{\ERROR}

```

o\_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.

```

5677 \cs_new_nopar:Npn \KV_parse_no_space_removal_no_sanitize:n {
5678   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_no_space_removal:w
5679   \KV_parse_no_sanitize_aux:n
5680 }

```

arise\_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.

```

5681 \cs_new_nopar:Npn \KV_parse_space_removal_sanitize:n {
5682   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
5683   \KV_parse_sanitize_aux:n
5684 }

```

For preamble use by the non-programmer this is probably best.

```

5685 \cs_new_nopar:Npn \KV_parse_space_removal_no_sanitize:n {
5686   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
5687   \KV_parse_no_sanitize_aux:n
5688 }

5689 </initex | package>

5690 <*showmemory>
5691 \showMemUsage
5692 </showmemory>

```

## 116 l3calc implementation

### 116.1 Variables

```

\l_calc_expression_tl
\g_calc_A_register
\l_calc_B_register
\l_calc_current_type_int
\g_calc_A_int
\l_calc_B_int
\l_calc_C_int
\g_calc_A_dim
\l_calc_B_dim
\l_calc_C_dim
\g_calc_A_skip
\l_calc_B_skip
\l_calc_C_skip
\g_calc_A_muskip
\l_calc_B_muskip
\l_calc_C_muskip

```

Internal registers.

## 116.2 Internal functions

```
\calc_assign_generic:NNNNnn
\calc_pre_scan:N
\calc_open:w
\calc_init_B:
\calc_numeric:
\calc_close:
\calc_post_scan:N
\calc_multiply:N
\calc_divide:N
\calc_generic_add_or_subtract:N
\calc_add:
\calc_subtract:
\calc_add_A_to_B:
\calc_subtract_A_from_B:
\calc_generic_multiply_or_divide:N
\calc_multiply_B_by_A:
\calc_divide_B_by_A:
\calc_multiply:
\calc_divide:
\calc_textsize:Nn
\calc_ratio_multiply:nn
\calc_ratio_divide:nn
\calc_real_evaluate:nn
\calc_real_multiply:n
\calc_real_divide:n
\calc_maxmin_operation:Nnn
\calc_maxmin_generic:Nnn
\calc_maxmin_div_or_mul:NNnn
\calc_maxmin_multiply:
\calc_maxmin_multiply:
\calc_error:N
\calc_chk_document_counter:nn
```

Awaiting better documentation :)

## 116.3 Module code

Since this is basically a re-worked version of the `calc` package, I haven't bothered with too many comments except for in the places where this package differs. This may (and should) change at some point.

We start by ensuring that the required packages are loaded.

```
5693 <*package>
5694 \ProvidesExplPackage
5695   {\filename}{\filedate}{\fileversion}{\filedescription}
5696   \package_check_loaded_expl:
5697 </package>
5698 <*initex | package>
```

```

\l_calc_expression_tl Here we define some registers and pointers we will need.
\g_calc_A_register
\l_calc_B_register
\l_calc_current_type_int
5699 \tl_new:Nn\l_calc_expression_tl{}
5700 \cs_new_nopar:Npn \g_calc_A_register{}
5701 \cs_new_nopar:Npn \l_calc_B_register{}
5702 \int_new:N \l_calc_current_type_int

```

```

\g_calc_A_int For each type of register we will need three registers to do our manipulations.
\l_calc_B_int
\l_calc_C_int
5703 \int_new:N \g_calc_A_int
5704 \int_new:N \l_calc_B_int
\g_calc_A_dim
5705 \int_new:N \l_calc_C_int
\l_calc_B_dim
5706 \dim_new:N \g_calc_A_dim
\l_calc_C_dim
5707 \dim_new:N \l_calc_B_dim
\g_calc_A_skip
5708 \dim_new:N \l_calc_C_dim
\l_calc_B_skip
5709 \skip_new:N \g_calc_A_skip
\l_calc_C_skip
5710 \skip_new:N \l_calc_B_skip
\g_calc_A_muskip
5711 \skip_new:N \l_calc_C_skip
\l_calc_B_muskip
5712 \muskip_new:N \g_calc_A_muskip
\l_calc_C_muskip
5713 \muskip_new:N \l_calc_B_muskip
5714 \muskip_new:N \l_calc_C_muskip

```

```

\calc_assign_generic:NNNNnn The generic function. #1 is a number denoting which type we are doing. (0=int, 1=dim,
2=skip, 3=muskip), #2 = temp register A, #3 = temp register B, #4 is a function acting
on #5 which is the register to be set. #6 is the calc expression. We do a little extra work
so that \real and \ratio can still be used by the user.

```

```

5715 \cs_new:Npn \calc_assign_generic:NNNNnn#1#2#3#4#5#6{
5716   \cs_set_eq:NN\g_calc_A_register#2
5717   \cs_set_eq:NN\l_calc_B_register#3
5718   \int_set:Nn \l_calc_current_type_int {#1}
5719   \group_begin:
5720     \cs_set_eq:NN \real \calc_real:n
5721     \cs_set_eq:NN \ratio\calc_ratio:nn
5722     \tl_set:Nx\l_calc_expression_tl{#6}
5723     \exp_after:wN
5724   \group_end:
5725   \exp_after:wN\calc_open:w\exp_after:wN(\l_calc_expression_tl !
5726   \pref_global:D\g_calc_A_register\l_calc_B_register
5727   \group_end:
5728   #4{#5}\l_calc_B_register
5729 }

```

A simpler version relying on \real and \ratio having our definition is

```

\cs_new:Npn \calc_assign_generic:NNNNnn#1#2#3#4#5#6{
  \cs_set_eq:NN\g_calc_A_register#2\cs_set_eq:NN\l_calc_B_register#3
  \int_set:Nn \l_calc_current_type_int {#1}
  \tl_set:Nx\l_calc_expression_tl{#6}
  \exp_after:wN\calc_open:w\exp_after:wN(\l_calc_expression_tl !
  \pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:
  #4{#5}\l_calc_B_register
}

```

\calc\_int\_set:Nn Here are the individual versions for the different register types. First integer registers.

\calc\_int\_gset:Nn

\calc\_int\_add:Nn

\calc\_int\_gadd:Nn

\calc\_int\_sub:Nn

\calc\_int\_gsub:Nn

```

5730 \cs_new_nopar:Npn\calc_int_set:Nn{
5731   \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_set:Nn
5732 }
5733 \cs_new_nopar:Npn\calc_int_gset:Nn{
5734   \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gset:Nn
5735 }
5736 \cs_new_nopar:Npn\calc_int_add:Nn{
5737   \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_add:Nn
5738 }
5739 \cs_new_nopar:Npn\calc_int_gadd:Nn{
5740   \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gadd:Nn
5741 }
5742 \cs_new_nopar:Npn\calc_int_sub:Nn{
5743   \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_sub:Nn
5744 }
5745 \cs_new_nopar:Npn\calc_int_gsub:Nn{
5746   \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gsub:Nn
5747 }

```

\calc\_dim\_set:Nn Dimens.

\calc\_dim\_gset:Nn

\calc\_dim\_add:Nn

\calc\_dim\_gadd:Nn

\calc\_dim\_sub:Nn

\calc\_dim\_gsub:Nn

```

5748 \cs_new_nopar:Npn\calc_dim_set:Nn{
5749   \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_set:Nn
5750 }
5751 \cs_new_nopar:Npn\calc_dim_gset:Nn{
5752   \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_gset:Nn
5753 }
5754 \cs_new_nopar:Npn\calc_dim_add:Nn{
5755   \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_add:Nn
5756 }
5757 \cs_new_nopar:Npn\calc_dim_gadd:Nn{
5758   \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_gadd:Nn
5759 }
5760 \cs_new_nopar:Npn\calc_dim_sub:Nn{
5761   \calc_assign_generic:NNNNnn\c_one\g_calc_A_int\l_calc_B_int\dim_sub:Nn
5762 }
5763 \cs_new_nopar:Npn\calc_dim_gsub:Nn{
5764   \calc_assign_generic:NNNNnn\c_one\g_calc_A_int\l_calc_B_int\dim_gsub:Nn
5765 }

```

\calc\_skip\_set:Nn Skips.

\calc\_skip\_gset:Nn

\calc\_skip\_add:Nn

\calc\_skip\_gadd:Nn

\calc\_skip\_sub:Nn

\calc\_skip\_gsub:Nn

```

5766 \cs_new_nopar:Npn\calc_skip_set:Nn{
5767   \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_set:Nn
5768 }
5769 \cs_new_nopar:Npn\calc_skip_gset:Nn{
5770   \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gset:Nn
5771 }
5772 \cs_new_nopar:Npn\calc_skip_add:Nn{
5773   \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_add:Nn
5774 }
5775 \cs_new_nopar:Npn\calc_skip_gadd:Nn{

```



```

5776 \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gadd:Nn
5777 }
5778 \cs_new_nopar:Npn\calc_skip_sub:Nn{
5779 \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_sub:Nn
5780 }
5781 \cs_new_nopar:Npn\calc_skip_gsub:Nn{
5782 \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gsub:Nn
5783 }

```

\calc\_muskip\_set:Nn Muskip.

```

\calc_muskip_gset:Nn
\calc_muskip_add:Nn
\calc_muskip_gadd:Nn
\calc_muskip_sub:Nn
\calc_muskip_gsub:Nn
5784 \cs_new_nopar:Npn\calc_muskip_set:Nn{
5785 \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
5786 \muskip_set:Nn
5787 }
5788 \cs_new_nopar:Npn\calc_muskip_gset:Nn{
5789 \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
5790 \muskip_gset:Nn
5791 }
5792 \cs_new_nopar:Npn\calc_muskip_add:Nn{
5793 \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
5794 \muskip_add:Nn
5795 }
5796 \cs_new_nopar:Npn\calc_muskip_gadd:Nn{
5797 \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
5798 \muskip_gadd:Nn
5799 }
5800 \cs_new_nopar:Npn\calc_muskip_sub:Nn{
5801 \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
5802 \muskip_add:Nn
5803 }
5804 \cs_new_nopar:Npn\calc_muskip_gsub:Nn{
5805 \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
5806 \muskip_gadd:Nn
5807 }

```

\calc\_pre\_scan:N In case we found one of the special operations, this should just be executed.

```

5808 \cs_new_nopar:Npn \calc_pre_scan:N #1{
5809 \if_meaning:w(#1
5810 \exp_after:wN\calc_open:w
5811 \else:
5812 \if_meaning:w \calc_textsize:Nn #1
5813 \else:
5814 \if_meaning:w \calc_maxmin_operation:Nnn #1
5815 \else:

```

\calc\_numeric: uses a primitive assignment so doesn't care about these dangling \fi:s.

```

5816 \calc_numeric:
5817 \fi:
5818 \fi:
5819 \fi:
5820 #1}

```

\calc\_open:w

```
5821 \cs_new_nopar:Npn \calc_open:w({
5822   \group_begin:\group_execute_after:N\calc_init_B:
5823   \group_begin:\group_execute_after:N\calc_init_B:
5824   \calc_pre_scan:N
5825 }
```

\calc\_init\_B:

\calc\_numeric:

\calc\_close:

```
5826 \cs_new_nopar:Npn\calc_init_B:{\l_calc_B_register\g_calc_A_register}
5827 \cs_new_nopar:Npn\calc_numeric:{
5828   \tex_afterassignment:D\calc_post_scan:N
5829   \pref_global:D\g_calc_A_register
5830 }
5831 \cs_new_nopar:Npn\calc_close:{
5832   \group_end:\pref_global:D\g_calc_A_register\l_calc_B_register
5833   \group_end:\pref_global:D\g_calc_A_register\l_calc_B_register
5834   \calc_post_scan:N}
```

\calc\_post\_scan:N Look at what token we have and decide where to go.

```
5835 \cs_new_nopar:Npn\calc_post_scan:N#1{
5836   \if_meaning:w#1!\cs_set_eq:NN\calc_next:w\group_end: \else:
5837     \if_meaning:w#1+\cs_set_eq:NN\calc_next:w\calc_add: \else:
5838       \if_meaning:w#1-\cs_set_eq:NN\calc_next:w\calc_subtract: \else:
5839         \if_meaning:w#1*\cs_set_eq:NN\calc_next:w\calc_multiply:N \else:
5840           \if_meaning:w#1/\cs_set_eq:NN\calc_next:w\calc_divide:N \else:
5841             \if_meaning:w#1)\cs_set_eq:NN\calc_next:w\calc_close: \else:
5842               \if_meaning:w#1\scan_stop:\cs_set_eq:NN\calc_next:w\calc_post_scan:N
5843             \else:
```

If we get here, there is an error but let's also disable \calc\_next:w since it is otherwise undefined. No need to give extra errors just for that.

```
5844           \cs_set_eq:NN \calc_next:w \prg_do_nothing:
5845           \calc_error:N#1
5846         \fi:
5847       \fi:
5848     \fi:
5849   \fi:
5850 \fi:
5851 \fi:
5852 \fi:
5853 \calc_next:w}
```

\calc\_multiply:N The switches for multiplication and division.

\calc\_divide:N

```
5854 \cs_new_nopar:Npn \calc_multiply:N #1{
5855   \if_meaning:w \calc_maxmin_operation:Nnn #1
5856   \cs_set_eq:NN \calc_next:w \calc_maxmin_multiply:
5857 \else:
5858   \if_meaning:w \calc_ratio_multiply:nn #1
5859   \cs_set_eq:NN \calc_next:w \calc_ratio_multiply:nn
```

```

5860     \else:
5861         \if_meaning:w \calc_real_evaluate:nn #1
5862         \cs_set_eq:NN \calc_next:w \calc_real_multiply:n
5863     \else:
5864         \cs_set_nopar:Npn \calc_next:w{\calc_multiply: #1}
5865     \fi:
5866 \fi:
5867 \fi:
5868 \calc_next:w
5869 }
5870 \cs_new_nopar:Npn \calc_divide:N #1{
5871     \if_meaning:w \calc_maxmin_operation:Nnn #1
5872     \cs_set_eq:NN \calc_next:w \calc_maxmin_divide:
5873 \else:
5874     \if_meaning:w \calc_ratio_multiply:nn #1
5875     \cs_set_eq:NN \calc_next:w \calc_ratio_divide:nn
5876 \else:
5877     \if_meaning:w \calc_real_evaluate:nn #1
5878     \cs_set_eq:NN \calc_next:w \calc_real_divide:n
5879 \else:
5880     \cs_set_nopar:Npn \calc_next:w{\calc_divide: #1}
5881 \fi:
5882 \fi:
5883 \fi:
5884 \calc_next:w
5885 }

```

\calc\_generic\_add\_or\_subtract:N Here is how we add and subtract.

```

\calc_add:
\calc_subtract:
\calc_add_A_to_B:
\calc_subtract_A_from_B:
5886 \cs_new_nopar:Npn \calc_generic_add_or_subtract:N#1{
5887     \group_end:
5888     \pref_global:D\g_calc_A_register\l_calc_B_register\group_end:
5889     \group_begin:\group_execute_after:N#1\group_begin:
5890     \group_execute_after:N\calc_init_B:
5891     \calc_pre_scan:N}
5892 \cs_new_nopar:Npn \calc_add:{\calc_generic_add_or_subtract:N\calc_add_A_to_B:}
5893 \cs_new_nopar:Npn \calc_subtract:{
5894     \calc_generic_add_or_subtract:N\calc_subtract_A_from_B:}

```

Don't use \tex\_advance:D since it allows overflows.

```

5895 \cs_new_nopar:Npn \calc_add_A_to_B:{
5896     \l_calc_B_register
5897     \if_case:w\l_calc_current_type_int
5898     \etex_numexpr:D\or:
5899     \etex_dimexpr:D\or:
5900     \etex_glueexpr:D\or:
5901     \etex_muexpr:D\fi:
5902     \l_calc_B_register + \g_calc_A_register\scan_stop:
5903 }
5904 \cs_new_nopar:Npn \calc_subtract_A_from_B:{
5905     \l_calc_B_register
5906     \if_case:w\l_calc_current_type_int
5907     \etex_numexpr:D\or:

```

```

5908 \etex_dimexpr:D\or:
5909 \etex_glueexpr:D\or:
5910 \etex_muexpr:D\fi:
5911 \l_calc_B_register - \g_calc_A_register\scan_stop:
5912 }

```

`\generic_multiply_or_divide:N` And here is how we multiply and divide. Note that we do not use the primitive TeX operations but the expandable operations provided by  $\epsilon$ -TeX. This means that all results are rounded not truncated!

```

\calc_multiply_B_by_A:
\calc_divide_B_by_A:
\calc_multiply:
\calc_divide:
5913 \cs_new_nopar:Npn\calc_generic_multiply_or_divide:N#1{
5914 \group_end:
5915 \group_begin:
5916 \cs_set_eq:NN\g_calc_A_register\g_calc_A_int
5917 \cs_set_eq:NN\l_calc_B_register\l_calc_B_int
5918 \int_zero:N \l_calc_current_type_int
5919 \group_execute_after:N#1\calc_pre_scan:N
5920 }
5921 \cs_new_nopar:Npn\calc_multiply_B_by_A:{
5922 \l_calc_B_register
5923 \if_case:w\l_calc_current_type_int
5924 \etex_numexpr:D\or:
5925 \etex_dimexpr:D\or:
5926 \etex_glueexpr:D\or:
5927 \etex_muexpr:D\fi:
5928 \l_calc_B_register*\g_calc_A_int\scan_stop:
5929 }
5930 \cs_new_nopar:Npn\calc_divide_B_by_A:{
5931 \l_calc_B_register
5932 \if_case:w\l_calc_current_type_int
5933 \etex_numexpr:D\or:
5934 \etex_dimexpr:D\or:
5935 \etex_glueexpr:D\or:
5936 \etex_muexpr:D\fi:
5937 \l_calc_B_register/\g_calc_A_int\scan_stop:
5938 }
5939 \cs_new_nopar:Npn\calc_multiply:{
5940 \calc_generic_multiply_or_divide:N\calc_multiply_B_by_A:}
5941 \cs_new_nopar:Npn\calc_divide:{
5942 \calc_generic_multiply_or_divide:N\calc_divide_B_by_A:}

```

`\calc_calculate_box_size:nnn` Put something in a box and measure it. #1 is a list of `\box_ht:N` etc., #2 should be `\dim_set:Nn<dim register>` or `\dim_gset:Nn<dim register>` and #3 is the contents.

```

\calc_calculate_box_size_aux:n
5943 \cs_new:Npn \calc_calculate_box_size:nnn #1#2#3{
5944 \hbox_set:Nn \l_tmpa_box {{#3}}
5945 #2{\c_zero_dim \tl_map_function:nN{#1}\calc_calculate_box_size_aux:n}
5946 }

```

Helper for calculating the final dimension.

```

5947 \cs_set_nopar:Npn \calc_calculate_box_size_aux:n#1{ + #1\l_tmpa_box}

```

`\calc_textsize:Nn` Now we can define `\calc_textsize:Nn`.

```

5948 \cs_set_protected:Npn \calc_textsize:Nn#1#2{
5949   \group_begin:
5950   \cs_set_eq:NN\calc_widthof_aux:n\box_wd:N
5951   \cs_set_eq:NN\calc_heightof_aux:n\box_ht:N
5952   \cs_set_eq:NN\calc_depthof_aux:n\box_dp:N
5953   \cs_set_nopar:Npn\calc_totalheightof_aux:n{\box_ht:N\box_dp:N}
5954   \exp_args:No\calc_calculate_box_size:nnn{#1}
5955   {\dim_gset:Nn\g_calc_A_register}

```

Restore the four user commands here since there might be a recursive call.

```

5956 {
5957   \cs_set_eq:NN \calc_depthof_aux:n \calc_depthof_auxi:n
5958   \cs_set_eq:NN \calc_widthof_aux:n \calc_widthof_auxi:n
5959   \cs_set_eq:NN \calc_heightof_aux:n \calc_heightof_auxi:n
5960   \cs_set_eq:NN \calc_totalheightof_aux:n \calc_totalheightof_auxi:n
5961   #2
5962 }
5963 \group_end:
5964 \calc_post_scan:N
5965 }

```

`\calc_ratio_multiply:nn` Evaluate a ratio. If we were already evaluation a *muskip* register, the ratio is probably also done with this type and we'll have to convert them to regular points.

`\calc_ratio_divide:nn`

```

5966 \cs_set_protected:Npn\calc_ratio_multiply:nn#1#2{
5967   \group_end:\group_begin:
5968   \if_num:w\l_calc_current_type_int < \c_three
5969     \calc_dim_set:Nn\l_calc_B_int{#1}
5970     \calc_dim_set:Nn\l_calc_C_int{#2}
5971   \else:
5972     \calc_dim_muskip:Nn{\l_calc_B_int\etex_mutogluue:D}{#1}
5973     \calc_dim_muskip:Nn{\l_calc_C_int\etex_mutogluue:D}{#2}
5974   \fi:

```

Then store the ratio as a fraction, which we just pass on.

```

5975   \cs_gset_nopar:Npx\calc_calculated_ratio:{
5976     \int_use:N\l_calc_B_int/\int_use:N\l_calc_C_int
5977   }
5978   \group_end:

```

Here we set the new value of `\l_calc_B_register` and remember to evaluate it as the correct type. Note that the intermediate calculation is a scaled operation (meaning the intermediate value is 64-bit) so we don't get into trouble when first multiplying by a large number and then dividing.

```

5979   \l_calc_B_register
5980   \if_case:w\l_calc_current_type_int
5981   \etex_numexpr:D\or:
5982   \etex_dimexpr:D\or:
5983   \etex_glueexpr:D\or:
5984   \etex_muexpr:D\fi:
5985   \l_calc_B_register*\calc_calculated_ratio:\scan_stop:
5986   \group_begin:
5987   \calc_post_scan:N}

```

Division is just flipping the arguments around.

```
5988 \cs_new:Npn \calc_ratio_divide:nn#1#2{\calc_ratio_multiply:nn{#2}{#1}}
```

\calc\_real\_evaluate:nn Although we could define the \real function as a subcase of \ratio, this is horribly inefficient since we just want to convert the decimal to a fraction.  
 \calc\_real\_multiply:n  
 \calc\_real\_divide:n

```
5989 \cs_new_protected_nopar:Npn\calc_real_evaluate:nn #1#2{
5990   \group_end:
5991   \l_calc_B_register
5992   \if_case:w\l_calc_current_type_int
5993   \etex_numexpr:D\or:
5994   \etex_dimexpr:D\or:
5995   \etex_glueexpr:D\or:
5996   \etex_muexpr:D\fi:
5997   \l_calc_B_register *
5998   \tex_number:D \dim_eval:n{#1pt}/
5999   \tex_number:D\dim_eval:n{#2pt}
6000   \scan_stop:
6001   \group_begin:
6002   \calc_post_scan:N}
6003 \cs_new_nopar:Npn \calc_real_multiply:n #1{\calc_real_evaluate:nn{#1}{1}}
6004 \cs_new_nopar:Npn \calc_real_divide:n {\calc_real_evaluate:nn{1}}
```

\calc\_maxmin\_operation:Nnn The max and min functions.

```
\calc_maxmin_generic:Nnn
\calc_maxmin_div_or_mul:NNnn
\calc_maxmin_multiply:
\calc_maxmin_multiply:
6005 \cs_set_protected:Npn\calc_maxmin_operation:Nnn#1#2#3{
6006   \group_begin:
6007   \calc_maxmin_generic:Nnn#1{#2}{#3}
6008   \group_end:
6009   \calc_post_scan:N
6010 }
```

#1 is either > or < and was expanded into this initially.

```
6011 \cs_new_protected:Npn \calc_maxmin_generic:Nnn#1#2#3{
6012   \group_begin:
6013   \if_case:w\l_calc_current_type_int
6014   \calc_int_set:Nn\l_calc_C_int{#2}%
6015   \calc_int_set:Nn\l_calc_B_int{#3}%
6016   \pref_global:D\g_calc_A_register
6017   \if_num:w\l_calc_C_int#1\l_calc_B_int
6018   \l_calc_C_int\else:\l_calc_B_int\fi:
6019   \or:
6020   \calc_dim_set:Nn\l_calc_C_dim{#2}%
6021   \calc_dim_set:Nn\l_calc_B_dim{#3}%
6022   \pref_global:D\g_calc_A_register
6023   \if_dim:w\l_calc_C_dim#1\l_calc_B_dim
6024   \l_calc_C_dim\else:\l_calc_B_dim\fi:
6025   \or:
6026   \calc_skip_set:Nn\l_calc_C_skip{#2}%
6027   \calc_skip_set:Nn\l_calc_B_skip{#3}%
6028   \pref_global:D\g_calc_A_register
6029   \if_dim:w\l_calc_C_skip#1\l_calc_B_skip
```

```

6030 \l_calc_C_skip\else:\l_calc_B_skip\fi:
6031 \else:
6032 \calc_muskip_set:Nn\l_calc_C_muskip{#2}%
6033 \calc_muskip_set:Nn\l_calc_B_muskip{#3}%
6034 \pref_global:D\g_calc_A_register
6035 \if_dim:w\l_calc_C_muskip#1\l_calc_B_muskip
6036 \l_calc_C_muskip\else:\l_calc_B_muskip\fi:
6037 \fi:
6038 \group_end:
6039 }
6040 \cs_new:Npn\calc_maxmin_div_or_mul:NNnn#1#2#3#4{
6041 \group_end:
6042 \group_begin:
6043 \int_zero:N\l_calc_current_type_int
6044 \group_execute_after:N#1
6045 \calc_maxmin_generic:Nnn#2{#3}{#4}
6046 \group_end:
6047 \group_begin:
6048 \calc_post_scan:N
6049 }
6050 \cs_new_nopar:Npn\calc_maxmin_multiply:{
6051 \calc_maxmin_div_or_mul:NNnn\calc_multiply_B_by_A:}
6052 \cs_new_nopar:Npn\calc_maxmin_divide: {
6053 \calc_maxmin_div_or_mul:NNnn\calc_divide_B_by_A:}

```

`\calc_error:N` The error message.

```

6054 \cs_new_nopar:Npn\calc_error:N#1{
6055 \PackageError{calc}
6056 {'\token_to_str:N#1'~ invalid~ at~ this~ point}
6057 {I~ expected~ to~ see~ one~ of:~ +- -- *~ /~ )}
6058 }

```

## 116.4 Higher level commands

The various operations allowed.

`\calc_maxof:nn` Max and min operations

`\calc_minof:nn`

`\maxof`

`\minof`

```

6059 \cs_new:Npn \calc_maxof:nn#1#2{
6060 \calc_maxmin_operation:Nnn > \exp_not:n{{#1}{#2}}
6061 }
6062 \cs_new:Npn \calc_minof:nn#1#2{
6063 \calc_maxmin_operation:Nnn < \exp_not:n{{#1}{#2}}
6064 }
6065 \cs_set_eq:NN \maxof \calc_maxof:nn
6066 \cs_set_eq:NN \minof \calc_minof:nn

```

`\calc_widthof:n` Text dimension commands.

`\calc_widthof_aux:n`

`\calc_widthof_auxi:n`

`\calc_heightof:n`

`\calc_heightof_aux:n`

`\calc_heightof_auxi:n`

`\calc_depthof:n`

`\calc_depthof_aux:n`

`\calc_depthof_auxi:n`

`\calc_totalheightof:n`

`\calc_totalheightof_aux:n`

`\calc_totalheightof_auxi:n`

```

6069 }
6070 \cs_new:Npn \calc_heightof:n#1{
6071   \calc_textsize:Nn \exp_not:N\calc_heightof_aux:n\exp_not:n{{#1}}
6072 }
6073 \cs_new:Npn \calc_depthof:n#1{
6074   \calc_textsize:Nn \exp_not:N\calc_depthof_aux:n\exp_not:n{{#1}}
6075 }
6076 \cs_new:Npn \calc_totalheightof:n#1{
6077   \calc_textsize:Nn \exp_not:N\calc_totalheightof_aux:n \exp_not:n{{#1}}
6078 }
6079 \cs_new:Npn \calc_widthof_aux:n #1{
6080   \exp_not:N\calc_widthof_aux:n\exp_not:n{{#1}}
6081 }
6082 \cs_new_eq:NN \calc_widthof_auxi:n \calc_widthof_aux:n
6083 \cs_new:Npn \calc_depthof_aux:n #1{
6084   \exp_not:N\calc_depthof_aux:n\exp_not:n{{#1}}
6085 }
6086 \cs_new_eq:NN \calc_depthof_auxi:n \calc_depthof_aux:n
6087 \cs_new:Npn \calc_heightof_aux:n #1{
6088   \exp_not:N\calc_heightof_aux:n\exp_not:n{{#1}}
6089 }
6090 \cs_new_eq:NN \calc_heightof_auxi:n \calc_heightof_aux:n
6091 \cs_new:Npn \calc_totalheightof_aux:n #1{
6092   \exp_not:N\calc_totalheightof_aux:n\exp_not:n{{#1}}
6093 }
6094 \cs_new_eq:NN \calc_totalheightof_auxi:n \calc_totalheightof_aux:n

```

\calc\_ratio:nn Ratio and real.

```

\calc_real:n
6095 \cs_new:Npn \calc_ratio:nn#1#2{
6096   \calc_ratio_multiply:nn\exp_not:n{{#1}}{#2}}
6097 \cs_new_nopar:Npn \calc_real:n {\calc_real_evaluate:nn}

```

We can implement real and ratio without actually using these names. We'll see.

```

\widthof User commands.
\heightof
\depthof
\totalheightof
\ratio
\real
6098 \cs_set_eq:NN \depthof\calc_depthof:n
6099 \cs_set_eq:NN \widthof\calc_widthof:n
6100 \cs_set_eq:NN \heightof\calc_heightof:n
6101 \cs_set_eq:NN \totalheightof\calc_totalheightof:n
6102 %%\cs_set_eq:NN \ratio\calc_ratio:nn
6103 %%\cs_set_eq:NN \real\calc_real:n

```

```

\setlength
\gsetlength
\addtolength
\gaddtolength
6104 \cs_set_protected_nopar:Npn \setlength{\calc_skip_set:Nn}
6105 \cs_set_protected_nopar:Npn \gsetlength{\calc_skip_gset:Nn}
6106 \cs_set_protected_nopar:Npn \addtolength{\calc_skip_add:Nn}
6107 \cs_set_protected_nopar:Npn \gaddtolength{\calc_skip_gadd:Nn}

```

\calc\_setcounter:nn Document commands for L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> counters. Also add support for amstext. Note that when l3breqn is used, \mathchoice will no longer need this switch as the argument is only executed once.

```

\setcounter
\addtocounter
\stepcounter

```

```

\calc_chk_document_counter:nn

```



```

6108 </initex | package>
6109 <*package>
6110 \newif\iffirstchoice@ \firstchoice@true
6111 </package>
6112 <*initex | package>
6113 \cs_set_protected_nopar:Npn \calc_setcounter:nn#1#2{
6114   \calc_chk_document_counter:nn{#1}{
6115     \exp_args:Nc\calc_int_gset:Nn {c@#1}{#2}
6116   }
6117 }
6118 \cs_set_protected_nopar:Npn \calc_addtocounter:nn#1#2{
6119 </initex | package>
6120 <*package>
6121 \iffirstchoice@
6122 </package>
6123 <*initex | package>
6124 \calc_chk_document_counter:nn{#1}{
6125   \exp_args:Nc\calc_int_gadd:Nn {c@#1}{#2}
6126 }
6127 </initex | package>
6128 <*package>
6129 \fi:
6130 </package>
6131 <*initex | package>
6132 }
6133 \cs_set_protected_nopar:Npn \calc_stepcounter:n#1{
6134 </initex | package>
6135 <*package>
6136 \iffirstchoice@
6137 </package>
6138 <*initex | package>
6139 \calc_chk_document_counter:nn{#1}{
6140   \int_gincr:c {c@#1}
6141   \group_begin:
6142     \cs_set_eq:NN \@elt\@stpelt \use:c{cl@#1}
6143   \group_end:
6144 }
6145 </initex | package>
6146 <*package>
6147 \fi:
6148 </package>
6149 <*initex | package>
6150 }
6151 \cs_new_nopar:Npn \calc_chk_document_counter:nn#1{
6152   \cs_if_free:cTF{c@#1}{\@nocounterr {#1}}
6153 }
6154 \cs_set_eq:NN \setcounter \calc_setcounter:nn
6155 \cs_set_eq:NN \addtocounter \calc_addtocounter:nn
6156 \cs_set_eq:NN \stepcounter \calc_stepcounter:n
6157 </initex | package>
6158 <*package>
6159 \AtBeginDocument{
6160   \cs_set_eq:NN \setcounter \calc_setcounter:nn
6161   \cs_set_eq:NN \addtocounter \calc_addtocounter:nn

```

```

6162 \cs_set_eq:NN \stepcounter \calc_stepcounter:n
6163 }

```

Prevent the usual calc from loading.

```

6164 \cs_set_nopar:cpn{ver@calc.sty}{2005/08/06}
6165 \</package>

```

```

6166 <*showmemory>
6167 \showMemUsage
6168 </showmemory>

```

## 117 l3file implementation

The usual lead-off.

```

6169 <*package>
6170 \ProvidesExplPackage
6171   {\filename}{\filedate}{\fileversion}{\filedescription}
6172 \package_check_loaded_expl:
6173 </package>
6174 <*initex | package>

```

`\g_file_record_clist` When files are read with logging, the names are added here. There are two lists, one for everything and one for only those items which might later be listed (as in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>'s `\listfiles`).

```

6175 \clist_new:N \g_file_record_clist
6176 \clist_new:N \g_file_record_full_clist

```

`\l_file_search_path_clist` Checking input needs a stream to work with

```

6177 \clist_new:N \l_file_search_path_clist

```

`\l_file_test_read_stream` Checking input needs a stream to work with

```

6178 \ior_new:N \l_file_test_read_stream

```

`\l_file_tmp_bool` A flag is needed for internal purposes.

```

6179 \bool_new:N \l_file_tmp_bool

```

`\l_file_tmp_tl` A scratch token list variable.

```

6180 \tl_new:N \l_file_tmp_tl

```

`\file_if_exist_p:n` Checking if a file exists takes place in two parts. First, there is a simple check “here”. If  
`\file_if_exist:nTF` that fails, then there is a loop over the current search path.

```

\file_if_exist_path:n
\file_if_exist_aux:n
6181 \prg_new_conditional:Nnn \file_if_exist:n {p,TF,T,F} {
6182   \ior_open:Nn \l_file_test_read_stream {#1}

```

```

6183 \ior_if_eof:NTF \l_file_test_read_stream {
6184   \file_if_exist_path:n {#1}
6185 }{
6186   \ior_close:N \l_file_test_read_stream
6187   \prg_return_true:
6188 }
6189 }
6190 \cs_new_nopar:Nn \file_if_exist_path:n {
6191   \bool_set_false:N \l_file_tmp_bool
6192   \cs_set_nopar:Nn \file_if_exist_aux:n {
6193     \ior_open:Nn \l_file_test_read_stream { #1 ##1 }
6194     \ior_if_eof:NF \l_file_test_read_stream {
6195       \bool_set_true:N \l_file_tmp_bool
6196       \clist_map_break:
6197     }
6198   }
6199 </initex|package>
6200 <*package>
6201   \cs_if_exist:NT \input@path {
6202     \cs_set_eq:NN \l_file_search_path_clist \input@path
6203   }
6204 </package>
6205 <*initex|package>
6206   \clist_map_function:NN \l_file_search_path_clist \file_if_exist_aux:n
6207   \ior_close:N \l_file_test_read_stream
6208   \bool_if:NTF \l_file_tmp_bool {
6209     \prg_return_true:
6210   }{
6211     \prg_return_false:
6212   }
6213 }
6214 \cs_new_nopar:Nn \file_if_exist_aux:n { }

```

\file\_add\_path:nN  
 \file\_add\_path\_search:n  
 \file\_add\_path\_aux:n

Checking if a file exists takes place in two parts. First, there is a simple check “here”. If that fails, then there is a loop over the current search path.

```

6215 \cs_new_nopar:Nn \file_add_path:nN {
6216   \tl_clear:N #2
6217   \ior_open:Nn \l_file_test_read_stream {#1}
6218   \ior_if_eof:NTF \l_file_test_read_stream {
6219     \file_add_path_search:nN {#1} #2
6220   }{
6221     \tl_set:Nn #2 {#1}
6222   }
6223   \ior_close:N \l_file_test_read_stream
6224 }
6225 \cs_new_nopar:Nn \file_add_path_search:nN {
6226   \cs_set_nopar:Nn \file_add_path_aux:n {
6227     \ior_open:Nn \l_file_test_read_stream { ##1 #1 }
6228     \ior_if_eof:NF \l_file_test_read_stream {
6229       \tl_set:Nn #2 { ##1 #1 }
6230       \clist_map_break:
6231     }
6232   }

```

```

6233 </initex | package>
6234 <*package>
6235   \cs_if_exist:NT \input@path {
6236     \cs_set_eq:NN \l_file_search_path_clist \input@path
6237   }
6238 </package>
6239 <*initex | package>
6240   \clist_map_function:NN \l_file_search_path_clist \file_add_path_aux:n
6241 }
6242 \cs_new_nopar:Nn \file_add_path_aux:n { }

```

`\file_input:n` `\file_add_path:n` will return an empty token list variable if the file is not found. This is used rather than `\file_if_exist:nT` here as it saves running the same loop twice.

```

6243 \cs_new:Nn \file_input:n {
6244   \file_add_path:nN {#1} \l_file_tmp_tl
6245   \tl_if_empty:NF \l_file_tmp_tl {
6246     \file_input_no_check:n \l_file_tmp_tl
6247   }
6248 }
6249 \cs_new:Nn \file_input_no_record:n {
6250   \file_add_path:nN {#1} \l_file_tmp_tl
6251   \tl_if_empty:NF \l_file_tmp_tl {
6252     \file_input_no_check_no_record:n \l_file_tmp_tl
6253   }
6254 }

```

`\file_input_no_check:n` File input records what is going on before setting to work.

```

6255 \cs_new_nopar:Nn \file_input_no_check:n {
6256   \clist_gput_right:Nx \g_file_record_clist {#1}
6257   \wlog{ADDING: #1}
6258 </initex | package>
6259 <*package>
6260   \@addtofilelist {#1}
6261 </package>
6262 <*initex | package>
6263   \clist_gput_right:Nx \g_file_record_full_clist {#1}
6264   \tex_input:D #1 ~
6265 }

```

`\file_input_no_check_no_record:n` Inputting a file without adding to the main record is basically the same: even in this case the file goes on the full log.

```

6266 \cs_new_nopar:Nn \file_input_no_check_no_record:n {
6267   \clist_gput_right:Nx \g_file_record_full_clist {#1}
6268   \tex_input:D #1 ~
6269 }

```

`\file_list:` Two functions to list all files used to the log: the full version includes everything whereas `\file_list_full:` the standard version uses the shorter record.

```

\file_list:N
6270 \cs_new_nopar:Nn \file_list: {

```

```

6271 \file_list:N \g_file_record_clist
6272 }
6273 \cs_new_nopar:Nn \file_list_full: {
6274   \file_list:N \g_file_record_full_clist
6275 }
6276 \cs_new_nopar:Nn \file_list:N {
6277   \clist_remove_duplicates:N #1
6278   \iow_log:x { *~File~List~* }
6279   \clist_map_function:NN #1 \file_list_aux:n
6280   \iow_log:x { ***** }
6281 }
6282 \cs_new_nopar:Nn \file_list_aux:n {
6283   \iow_log:x { #1 }
6284 }

```

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```

6285 </initex | package>
6286 <*package>
6287 % \begin{macrocode}
6288 \AtBeginDocument{
6289   \clist_put_right:NV \g_file_record_clist \@filelist
6290   \clist_put_right:NV \g_file_record_full_clist \@filelist
6291 }
6292 </package>

```

# Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols			
<code>\!</code>	1515	<code>\@declaredoptions</code>	709
<code>\%</code>	4812	<code>\@elt</code>	6142
<code>\&amp;</code>	4973, 4975	<code>\@empty</code>	708, 709
<code>\*</code>	2272, 2274, 2278, 2283	<code>\@filelist</code>	6289, 6290
<code>\,</code>	5540, 5542	<code>\@gobble</code>	692
<code>\-</code>	62	<code>\@ifpackageloaded</code>	726
<code>\/</code>	61	<code>\@nil</code>	706, 710
<code>\:</code>	587, 591, 985, 2453	<code>\@nocounterr</code>	6152
<code>\::</code>	157, 1449, 1452, <u>1454</u> , 1454–1456, 1458, 1459, 1461, 1464, 1467, 1475, 1480, 1485, 1493, 1536– 1543, 1545–1551, 1553–1555, 1557, 1559–1561, 1563–1566, 1568–1571, 1573–1578, 1580, 1581, 1646, 1650, 1653, 1657, 1663–1665, 1667, 1673	<code>\@p@pfilename</code>	706, 710
<code>\::N</code>	157, <u>1458</u> , 1458, 1540–1543, 1561, 1563–1566, 1573, 1574, 1673	<code>\@popfilename</code>	115, <u>659</u> , 666, 667
<code>\::V</code>	<u>157</u> , <u>1485</u> , 1485, 1538, 1542, 1557, 1561	<code>\@pushfilename</code>	115, <u>659</u> , 659, 664
<code>\::V_unbraced</code>	<u>1645</u> , 1653, 1664, 1667, 1673	<code>\@stpelt</code>	6142
<code>\::c</code>	<u>157</u> , <u>1461</u> , 1461, 1545, 1548, 1553, 1568, 1573–1578, 1667	<code>\@tempboxa</code>	5350
<code>\::f</code>	<u>157</u> , <u>1467</u> , 1467, 1536, 1540, 1546–1549	<code>\@unknownoptionerror</code>	707
<code>\::f_unbraced</code>	<u>1645</u> , 1646, 1663	<code>\@</code>	1109, 1111, 1115, 1123, 1124, 2366, 4981, 4984, 5008, 5015, 5246, 5251, 5252, 5258, 5264, 5265
<code>\::n</code>	<u>157</u> , <u>1455</u> , 1455, 1549– 1551, 1563, 1564, 1568–1571, 1578	<code>\{</code>	4290, 4469, 4648, 4650, 4813
<code>\::o</code>	<u>157</u> , <u>1464</u> , 1464, 1546, 1550, 1553–1555, 1559, 1563, 1565, 1566, 1569, 1571, 1574–1576, 1580, 1581	<code>\}</code>	4290, 4469, 4648, 4650, 4814, 4974
<code>\::o_unbraced</code>	<u>1645</u> , 1650	<code>^^A\box_new_l:N</code>	103
<code>\::v</code>	<u>157</u> , <u>1485</u> , 1493, 1537, 1541	<code>^^A\dim_new_l:N</code>	61
<code>\::v_unbraced</code>	<u>1645</u> , 1657, 1665	<code>^^A\int_new_l:N</code>	48
<code>\::x</code>	<u>157</u> , <u>1473</u> , 1475, 1480, 1539, 1543, 1545, 1551, 1555, 1559, 1560, 1564, 1566, 1570, 1571, 1577, 1578, 1580	<code>^^A\muskip_new_l:N</code>	64
<code>\=</code>	5539, 5541	<code>^^A\skip_new_l:N</code>	59
<code>\?</code>	2366, 2453	<code>^^A\toks_new_l:N</code>	74
<code>\@</code>	713, 985, 986, 3895, 3896	<code>\_</code>	586, 590
<code>\@end</code>	686	<code>\ </code>	2864
<code>\@hyph</code>	690	Numbers	
<code>\@input</code>	684	<code>\8</code>	5541
<code>\@italiccorr</code>	689	<code>\9</code>	5542
<code>\@underline</code>	685	<code>\_</code>	60, 1516, 4795, 4879, 4924, 4929, 4934, 4944, 4953, 4962, 4973, 4974, 4976, 4982, 5009, 5016, 5451
<code>\@addtofilelist</code>	6260	A	
<code>\@current</code>	712	<code>\A</code>	2367, 2455, 3895, 3897
<code>\@currname</code>	711	<code>\a</code>	977
<code>\@currnamestack</code>	706, 714	<code>\above</code>	181
		<code>\abovedisplayshortskip</code>	194
		<code>\abovedisplayskip</code>	195
		<code>\abovewithdelims</code>	182
		<code>\accent</code>	232
		<code>\addtocounter</code>	111, <u>6108</u> , 6155, 6161
		<code>\addtolength</code>	111, <u>6104</u> , 6106
		<code>\adjdemerits</code>	269

- \advance ..... 76
- \afterassignment ..... 86
- \aftergroup ..... 87
- \alloc\_reg:NnNN ..... 2696, 2697,  
3241, 3242, 3357, 3358, 3417, 3418,  
4069, 4070, 4766, 4817, 5286, 5287
- \alloc\_setup\_type:nnn ... 2695, 3240,  
3356, 3416, 4068, 4765, 4816, 5284
- \AtBeginDocument ..... 5485, 6159, 6288
- \atop ..... 183
- \atopwithdelims ..... 184
- B**
- \B ..... 3896, 3898
- \badness ..... 331
- \baselineskip ..... 259
- \batchmode ..... 152
- \begin ..... 5500, 6287
- \begingroup ..... 2, 90
- \beginL ..... 444
- \beginR ..... 446
- \belowdisplaysshortskip ..... 196
- \belowdisplayskip ..... 197
- \binoppenalty ..... 220
- \bool(:w ..... 1873
- \bool\_)\_0:w ..... 1873
- \bool\_)\_1:w ..... 1873
- \bool\_8\_0:w ..... 1873
- \bool\_8\_1:w ..... 1873
- \bool\_:w ..... 1873
- \bool\_choose:NN ..... 1873, 1892, 1895
- \bool\_cleanup:N .. 1873, 1888, 1890, 1891
- \bool\_do\_until:cn ..... 32, 1865
- \bool\_do\_until:Nn 32, 1865, 1869, 1870, 1872
- \bool\_do\_until:nn .. 35, 1930, 1939, 1940
- \bool\_do\_while:cn ..... 32, 1865
- \bool\_do\_while:Nn 32, 1865, 1865, 1866, 1868
- \bool\_do\_while:nn .. 35, 1930, 1936, 1937
- \bool\_eval\_skip\_to\_end:Nw .....  
..... 1902, 1903, 1904, 1904, 1913
- \bool\_eval\_skip\_to\_end\_aux:Nw .....  
..... 1904, 1905, 1907
- \bool\_eval\_skip\_to\_end\_auxii:Nw .....  
..... 1904, 1910, 1912
- \bool\_get\_next:N ..... 1873,  
1875, 1877, 1885, 1888, 1896, 1897
- \bool\_gset:cn ..... 33, 1915
- \bool\_gset:Nn ..... 33, 1915, 1916, 1920
- \bool\_gset\_eq:cc ..... 32, 1840, 1847
- \bool\_gset\_eq:cN ..... 32, 1840, 1846
- \bool\_gset\_eq:Nc ..... 32, 1840, 1845
- \bool\_gset\_eq:NN ..... 32, 1840, 1844
- \bool\_gset\_false:c ..... 31, 1830, 1839
- \bool\_gset\_false:N ..... 31, 1830, 1838
- \bool\_gset\_true:c ..... 31, 1830, 1837
- \bool\_gset\_true:N ..... 31, 1830, 1836
- \bool\_I\_0:w ..... 1873
- \bool\_I\_1:w ..... 1873
- \bool\_if:cTF ..... 32, 1850
- \bool\_if:N ..... 1850
- \bool\_if:n ..... 1926
- \bool\_if:NF ..... 1856, 1862, 1870, 2955
- \bool\_if:nF ..... 1934, 1940
- \bool\_if:NT .....  
.. 1855, 1858, 1866, 2955, 2956, 5574
- \bool\_if:nT ..... 1931, 1937
- \bool\_if:NTF .....  
32, 1257, 1850, 1854, 2947, 5646, 6208
- \bool\_if:nTF ..... 33, 1873, 2582, 3338
- \bool\_if\_p:c ..... 32, 1850
- \bool\_if\_p:N ..... 32, 1850, 1853
- \bool\_if\_p:n ..... 33, 1873,  
1873, 1915, 1917, 1921, 1923, 1927
- \bool\_new:c ..... 31, 1830, 1831
- \bool\_new:N .....  
31, 1830, 1830, 1848, 1849, 5536, 6179
- \bool\_not\_p:n ..... 33, 1921, 1921
- \bool\_p:w ..... 1873
- \bool\_S\_0:w ..... 1873
- \bool\_S\_1:w ..... 1873
- \bool\_set:cn ..... 33, 1915
- \bool\_set:Nn ..... 33, 1915, 1915, 1919
- \bool\_set\_eq:cc ..... 32, 1840, 1843
- \bool\_set\_eq:cN ..... 32, 1840, 1842
- \bool\_set\_eq:Nc ..... 32, 1840, 1841
- \bool\_set\_eq:NN ..... 32, 1840, 1840
- \bool\_set\_false:c ..... 31, 1830, 1835
- \bool\_set\_false:N .. 31, 1830, 1834, 6191
- \bool\_set\_true:c ..... 31, 1830, 1833
- \bool\_set\_true:N 31, 1830, 1832, 5537, 6195
- \bool\_until\_do:cn ..... 32, 1857
- \bool\_until\_do:Nn 32, 1857, 1861, 1862, 1864
- \bool\_until\_do:nn .. 35, 1930, 1933, 1934
- \bool\_while\_do:cn ..... 32, 1857
- \bool\_while\_do:Nn 32, 1857, 1857, 1858, 1860
- \bool\_while\_do:nn .. 35, 1930, 1930, 1931
- \bool\_xor\_p:nn ..... 33, 1922, 1922
- \botmark ..... 167
- \botmarks ..... 393
- \box ..... 375
- \box\_clear:c ..... 105, 5333
- \box\_clear:N ..... 105, 5333, 5333–5335
- \box\_dp:c ..... 105, 5337
- \box\_dp:N 105, 5337, 5338, 5341, 5952, 5953
- \box\_gclear:c ..... 105, 5333
- \box\_gclear:N ..... 105, 5333, 5335, 5336
- \box\_gset\_eq:cc ..... 104, 5322
- \box\_gset\_eq:cN ..... 104, 5322

- \box\_gset\_eq:Nc ..... 104, 5322
- \box\_gset\_eq:NN ... 104, 5322, 5322, 5323
- \box\_gset\_to\_last:c ..... 104, 5325
- \box\_gset\_to\_last:N 104, 5325, 5327, 5328
- \box\_ht:c ..... 105, 5337
- \box\_ht:N 105, 5337, 5337, 5340, 5951, 5953
- \box\_if\_empty:cTF ..... 104, 5313
- \box\_if\_empty:N ..... 5313
- \box\_if\_empty:NF ..... 5319
- \box\_if\_empty:NT ..... 5318
- \box\_if\_empty:NTF ..... 104, 5313, 5317
- \box\_if\_empty\_p:c ..... 104, 5313
- \box\_if\_empty\_p:N ..... 104, 5313, 5316
- \box\_if\_horizontal:cTF ..... 104, 5299
- \box\_if\_horizontal:N ..... 5299
- \box\_if\_horizontal:NF ..... 5308
- \box\_if\_horizontal:NT ..... 5307
- \box\_if\_horizontal:NTF . 104, 5299, 5306
- \box\_if\_horizontal\_p:c ..... 104, 5299
- \box\_if\_horizontal\_p:N . 104, 5299, 5305
- \box\_if\_vertical:cTF ..... 104, 5299
- \box\_if\_vertical:N ..... 5302
- \box\_if\_vertical:NF ..... 5312
- \box\_if\_vertical:NT ..... 5311
- \box\_if\_vertical:NTF ... 104, 5299, 5310
- \box\_if\_vertical\_p:c ..... 104, 5299
- \box\_if\_vertical\_p:N ... 104, 5299, 5309
- \box\_move\_down:nn ..... 105, 5329, 5332
- \box\_move\_left:nn ..... 105, 5329, 5329
- \box\_move\_right:nn ..... 105, 5329, 5330
- \box\_move\_up:nn ..... 105, 5329, 5331
- \box\_new:c ..... 103, 5283
- \box\_new:N ..... 103, 5283, 5286, 5290, 5295, 5351–5353
- \box\_new\_l:N ..... 5287
- \box\_set\_eq:cc ..... 104, 5320
- \box\_set\_eq:cN ..... 104, 5320
- \box\_set\_eq:Nc ..... 104, 5320
- \box\_set\_eq:NN 104, 5320, 5320–5322, 5333
- \box\_set\_to\_last:c ..... 104, 5325
- \box\_set\_to\_last:N . 104, 5325, 5325–5327
- \box\_show:c ..... 105, 5347
- \box\_show:N ..... 105, 5347, 5347, 5348
- \box\_use:c ..... 105, 5343
- \box\_use:N ..... 105, 5343, 5345, 5346
- \box\_use\_clear:c ..... 105, 5343
- \box\_use\_clear:N .. 105, 5343, 5343, 5344
- \box\_wd:c ..... 105, 5337
- \box\_wd:N .... 105, 5337, 5339, 5342, 5950
- \boxmaxdepth ..... 337
- \brokenpenalty ..... 294
- \c\_active\_char\_token 43, 2269, 2284, 2327
- \c\_alignment\_tab\_token 43, 2269, 2275, 2299
- \c\_eight ..... 52, 2196, 2212, 2904, 2914
- \c\_eleven ..... 52, 2199, 2215, 2904, 2917
- \c\_empty\_box .. 105, 5333, 5349, 5349, 5351
- \c\_empty\_tl ..... 71, 3506, 3508, 3615, 3615, 3625, 4247, 4414
- \c\_empty\_toks ..... 78, 4116, 4208, 4208
- \c\_false\_bool 9, 936, 956, 974, 975, 999, 1233, 1242, 1830, 1831, 1834, 1835, 1838, 1839, 1898, 1900, 1902, 1924, 2409, 2503, 2939, 2942, 2948, 4044
- \c\_fifteen .... 52, 2203, 2219, 2904, 2921
- \c\_five ..... 52, 2193, 2209, 2904, 2911
- \c\_four ..... 52, 2192, 2208, 2904, 2910
- \c\_fourteen .... 52, 2202, 2218, 2904, 2920
- \c\_group\_begin\_token ..... 43, 2269, 2269, 2287, 2583, 5366, 5392
- \c\_group\_end\_token ..... 43, 2269, 2270, 2291, 5367, 5370, 5395, 5400
- \c\_hundred\_one ..... 52, 2904, 2924
- \c\_ior\_log\_stream ..... 97, 4778, 4781
- \c\_ior\_term\_stream ..... 97, 4778, 4779
- \c\_iow\_comment\_char ..... 97, 4812, 4812
- \c\_iow\_lbrace\_char ..... 97, 4812, 4813
- \c\_iow\_log\_stream 97, 4778, 4780, 4786, 4787
- \c\_iow\_rbrace\_char ..... 97, 4812, 4814
- \c\_iow\_term\_stream ..... 97, 4778, 4778, 4788, 4789
- \c\_job\_name\_tl ..... 71, 3614, 3614
- \c\_KV\_single\_equal\_sign\_tl ..... 290, 5530, 5532, 5617, 5668
- \c\_letter\_token .... 43, 2269, 2281, 2319
- \c\_luatex\_is\_engine\_bool ..... 23, 1230, 1240, 1242, 1249
- \c\_math\_shift\_token . 43, 2269, 2273, 2295
- \c\_math\_subscript\_token ..... 43, 2269, 2279, 2311
- \c\_math\_superscript\_token ..... 43, 2269, 2277, 2307
- \c\_max\_dim ..... 63, 3395, 3396
- \c\_max\_int ..... 52, 2934, 2934
- \c\_max\_register\_num ..... 55, 2695, 2895, 3076, 3076, 3240, 3356, 3416, 4068, 5284
- \c\_max\_skip 61, 3327, 3330, 3331, 3335, 3396
- \c\_minus\_one . 13, 52, 1063, 1064, 1067, 1068, 1073, 2724, 2742, 2744, 2894, 2904, 2904, 2905, 3038, 3040, 3906, 3922, 3932, 3940, 4780, 4781, 4848
- \c\_msg\_code\_prefix\_tl 103, 4887, 4889, 4967, 4969, 5135, 5136, 5230, 5231
- \c\_msg\_error\_tl 102, 4849, 4850, 5121, 5216

## C

\C ..... 2367, 2456



- \c\_msg\_fatal\_text\_tl ..... 102, 4853, 4853, 5116, 5210
- \c\_msg\_fatal\_tl ..... 102, 4849, 4849, 5110, 5112, 5115, 5204
- \c\_msg\_help\_text\_tl 102, 4853, 4856, 4990
- \c\_msg\_hide\_tl ..... 4883, 4991
- \c\_msg\_info\_tl .... 102, 4849, 4852, 5149
- \c\_msg\_kernel\_bug\_more\_text\_tl .... 102, 4853, 4862, 5273
- \c\_msg\_kernel\_bug\_text\_tl ..... 102, 4853, 4859, 5270
- \c\_msg\_more\_text\_prefix\_tl ..... 103, 4887, 4888, 4949, 4951, 4958, 4960, 5128, 5129, 5223, 5224
- \c\_msg\_no\_info\_text\_tl ..... 102, 4853, 4868, 5001, 5131, 5226
- \c\_msg\_on\_line\_tl . 103, 4886, 4886, 4901
- \c\_msg\_return\_text\_tl .... 102, 4853, 4866, 4870, 4872, 5254, 5260, 5267
- \c\_msg\_text\_prefix\_tl ..... 103, 4887, 4887, 4911, 4915, 4919, 4943, 4947, 4956, 5072, 5113, 5124, 5143, 5150, 5156, 5162, 5207, 5219
- \c\_msg\_warning\_tl . 102, 4849, 4851, 5142
- \c\_nine ..... 52, 2197, 2213, 2904, 2915
- \c\_one ..... 52, 2189, 2205, 2719, 2741, 2743, 2904, 2907, 3128, 3132, 5749, 5752, 5755, 5758, 5761, 5764
- \c\_other\_char\_token . 43, 2269, 2282, 2323
- \c\_parameter\_token ..... 43, 2269, 2276
- \c\_peek\_true\_remove\_next\_tl ..... 184, 2552, 2563, 2564
- \c\_seven . 52, 1712, 2195, 2211, 2904, 2913
- \c\_six ... 52, 1708, 2194, 2210, 2904, 2912
- \c\_sixteen .. 13, 52, 1063, 1070, 1076, 2904, 2922, 4765, 4778, 4779, 4816
- \c\_space\_token ..... 43, 2269, 2280, 2315, 2584, 2675
- \c\_ten ... 52, 2198, 2214, 2904, 2916, 5451
- \c\_ten\_thousand ..... 52, 2904, 2928
- \c\_ten\_thousand\_four .... 52, 2904, 2932
- \c\_ten\_thousand\_one ..... 52, 2904, 2929
- \c\_ten\_thousand\_three ... 52, 2904, 2931
- \c\_ten\_thousand\_two ..... 52, 2904, 2930
- \c\_thirteen .... 52, 2201, 2217, 2904, 2919
- \c\_thirty\_two ..... 52, 2904, 2923
- \c\_thousand ..... 52, 2904, 2927
- \c\_three 52, 2191, 2207, 2904, 2909, 5785, 5789, 5793, 5797, 5801, 5805, 5968
- \c\_true\_bool ..... 9, 936, 956, 974, 974, 1003, 1231, 1240, 1832, 1833, 1836, 1837, 1899, 1901, 1903, 1924, 2936, 2939, 2942, 2949, 4042
- \c\_twelve ..... 52, 2200, 2216, 2904, 2918
- \c\_twenty\_thousand ..... 52, 2904, 2933
- \c\_two .... 52, 2190, 2206, 2904, 2908, 5767, 5770, 5773, 5776, 5779, 5782
- \c\_two\_ats\_with\_two\_catcodes\_tl .... 3894, 3901, 3905, 3916, 3939
- \c\_twohundred\_fifty\_five . 52, 2904, 2925
- \c\_twohundred\_fifty\_six . 52, 2904, 2926
- \c\_undefined .... 44, 49, 4949, 4958, 4967
- \c\_undefined:D ..... 1226, 1228
- \c\_xetex\_is\_engine\_bool ..... 23, 1230, 1231, 1233, 1236, 1245
- \c\_zero ..... 13, 52, 935, 940, 945, 950, 955, 960, 965, 970, 1045, 1051, 1063, 1071, 1705, 1706, 1710, 1758, 1785, 1809, 2119, 2141, 2188, 2204, 2749, 2751, 2904, 2906, 2962, 3141, 3171, 3175, 3176, 3182, 3240, 3339, 3340, 3356, 3416, 3711, 4068, 4765, 4816, 5284, 5731, 5734, 5737, 5740, 5743, 5746
- \c\_zero\_dim .... 63, 3395, 3395, 5372, 5945
- \c\_zero\_skip 61, 3267, 3327, 3328, 3329, 3334, 3346, 3347, 3371, 3395, 5402
- \calc\_add: ..... 298, 5837, 5886, 5892
- \calc\_add\_A\_to\_B: . 298, 5886, 5892, 5895
- \calc\_addtocounter:n ..... 111, 6108, 6118, 6155, 6161
- \calc\_assign\_generic:NNNNnn ..... 298, 5715, 5715, 5731, 5734, 5737, 5740, 5743, 5746, 5749, 5752, 5755, 5758, 5761, 5764, 5767, 5770, 5773, 5776, 5779, 5782, 5785, 5789, 5793, 5797, 5801, 5805
- \calc\_calculate\_box\_size:n ..... 112, 5943, 5943, 5954
- \calc\_calculate\_box\_size\_aux:n .... 5943, 5945, 5947
- \calc\_calculated\_ratio: .... 5975, 5985
- \calc\_chk\_document\_counter:n ..... 298, 6108, 6114, 6124, 6139, 6151
- \calc\_close: ..... 298, 5826, 5831, 5841
- \calc\_depthof:n ... 111, 6067, 6073, 6098
- \calc\_depthof\_aux:n ..... 5952, 5957, 6067, 6074, 6083, 6084, 6086
- \calc\_depthof\_auxi:n ... 5957, 6067, 6086
- \calc\_dim\_add:Nn ..... 112, 5748, 5754
- \calc\_dim\_gadd:Nn ..... 112, 5748, 5757
- \calc\_dim\_gset:Nn ..... 112, 5748, 5751
- \calc\_dim\_gsub:Nn ..... 112, 5748, 5763
- \calc\_dim\_muskip:Nn ..... 5972, 5973
- \calc\_dim\_set:Nn ..... 112, 5748, 5748, 5969, 5970, 6020, 6021
- \calc\_dim\_sub:Nn ..... 112, 5748, 5760
- \calc\_divide: ..... 298, 5880, 5913, 5941

`\calc_divide:N` . . . . . 298, 5840, 5854, 5870  
`\calc_divide_B_by_A:` . . . . .  
. . . . . 298, 5913, 5930, 5942, 6053  
`\calc_error:N` . . . . . 298, 5845, 6054, 6054  
`\calc_generic_add_or_subtract:N` . . . . .  
. . . . . 298, 5886, 5886, 5892, 5894  
`\calc_generic_multiply_or_divide:N` . . . . .  
. . . . . 298, 5913, 5913, 5940, 5942  
`\calc_heightof:n` . . . 111, 6067, 6070, 6100  
`\calc_heightof_aux:n` . . . . . 5951,  
5959, 6067, 6071, 6087, 6088, 6090  
`\calc_heightof_auxi:n` . . . 5959, 6067, 6090  
`\calc_init_B:` . . . . .  
. . . 298, 5822, 5823, 5826, 5826, 5890  
`\calc_int_add:Nn` . . . . . 111, 5730, 5736  
`\calc_int_gadd:Nn` . . . 111, 5730, 5739, 6125  
`\calc_int_gset:Nn` . . . 111, 5730, 5733, 6115  
`\calc_int_gsub:Nn` . . . . . 111, 5730, 5745  
`\calc_int_set:Nn` 111, 5730, 5730, 6014, 6015  
`\calc_int_sub:Nn` . . . . . 111, 5730, 5742  
`\calc_maxmin_div_or_mul:NNnn` . . . . .  
. . . . . 298, 6005, 6040, 6051, 6053  
`\calc_maxmin_divide:` . . . . . 5872, 6052  
`\calc_maxmin_generic:Nnn` . . . . .  
. . . . . 298, 6005, 6007, 6011, 6045  
`\calc_maxmin_multiply:` . . . . .  
. . . . . 298, 5856, 6005, 6050  
`\calc_maxmin_operation:Nnn` 298, 5814,  
5855, 5871, 6005, 6005, 6060, 6063  
`\calc_maxof:nn` . . . . . 111, 6059, 6059, 6065  
`\calc_minof:nn` . . . . . 111, 6059, 6062, 6066  
`\calc_multiply:` . . . 298, 5864, 5913, 5939  
`\calc_multiply:N` . . . 298, 5839, 5854, 5854  
`\calc_multiply_B_by_A:` . . . . .  
. . . . . 298, 5913, 5921, 5940, 6051  
`\calc_muskip_add:Nn` . . . . . 112, 5784, 5792  
`\calc_muskip_gadd:Nn` . . . . . 112, 5784, 5796  
`\calc_muskip_gset:Nn` . . . . . 112, 5784, 5788  
`\calc_muskip_gsub:Nn` . . . . . 112, 5784, 5804  
`\calc_muskip_set:Nn` . . . . .  
. . . . . 112, 5784, 5784, 6032, 6033  
`\calc_muskip_sub:Nn` . . . . . 112, 5784, 5800  
`\calc_next:w` . . . . . 5836–5842,  
5844, 5853, 5856, 5859, 5862, 5864,  
5868, 5872, 5875, 5878, 5880, 5884  
`\calc_numeric:` . . . . . 298, 5816, 5826, 5827  
`\calc_open:w` . . . 298, 5725, 5810, 5821, 5821  
`\calc_post_scan:N` . . . . .  
. . . . . 298, 5828, 5834, 5835, 5835,  
5842, 5964, 5987, 6002, 6009, 6048  
`\calc_pre_scan:N` . . . . .  
. . . 298, 5808, 5808, 5824, 5891, 5919  
`\calc_ratio:nn` 111, 5721, 6095, 6095, 6102  
`\calc_ratio_divide:nn` 298, 5875, 5966, 5988  
`\calc_ratio_multiply:nn` . . . 298, 5858,  
5859, 5874, 5966, 5966, 5988, 6096  
`\calc_real:n` . . . 111, 5720, 6095, 6097, 6103  
`\calc_real_divide:n` 298, 5878, 5989, 6004  
`\calc_real_evaluate:nn` . . . 298, 5861,  
5877, 5989, 5989, 6003, 6004, 6097  
`\calc_real_multiply:n` 298, 5862, 5989, 6003  
`\calc_setcounter:nn` . . . . .  
. . . . . 111, 6108, 6113, 6154, 6160  
`\calc_skip_add:Nn` . . . 112, 5766, 5772, 6106  
`\calc_skip_gadd:Nn` . . . 112, 5766, 5775, 6107  
`\calc_skip_gset:Nn` . . . 112, 5766, 5769, 6105  
`\calc_skip_gsub:Nn` . . . . . 112, 5766, 5781  
`\calc_skip_set:Nn` . . . . .  
. . . 112, 5766, 5766, 6026, 6027, 6104  
`\calc_skip_sub:Nn` . . . . . 112, 5766, 5778  
`\calc_stepcounter:n` . . . . .  
. . . . . 111, 6108, 6133, 6156, 6162  
`\calc_subtract:` . . . 298, 5838, 5886, 5893  
`\calc_subtract_A_from_B:` . . . . .  
. . . . . 298, 5886, 5894, 5904  
`\calc_textsize:Nn` . . . . . 298, 5812,  
5948, 5948, 6068, 6071, 6074, 6077  
`\calc_totalheightof:n` 111, 6067, 6076, 6101  
`\calc_totalheightof_aux:n` . . . . . 5953,  
5960, 6067, 6077, 6091, 6092, 6094  
`\calc_totalheightof_auxi:n` . . . . .  
. . . . . 5960, 6067, 6094  
`\calc_widthof:n` . . . . . 111, 6067, 6067, 6099  
`\calc_widthof_aux:n` . . . . . 5950,  
5958, 6067, 6068, 6079, 6080, 6082  
`\calc_widthof_auxi:n` . . . . . 5958, 6067, 6082  
`\catcode` . . . 26–28, 30, 31, 34–36, 38, 39, 379  
`\char` . . . . . 233  
`\char_gset_mathcode:nn` . . . 42, 2220, 2225  
`\char_gset_mathcode:w` 42, 2220, 2224, 2226  
`\char_make_active:N` . . . 41, 2188, 2201, 4975  
`\char_make_active:n` . . . . . 41, 2204, 2217  
`\char_make_alignment:N` . . . . . 41, 2188, 2192  
`\char_make_alignment:n` . . . . . 41, 2204, 2208  
`\char_make_begin_group:N` . . . . . 41, 2188, 2189  
`\char_make_begin_group:n` . . . . . 41, 2204, 2205  
`\char_make_comment:N` . . . . . 41, 2188, 2202  
`\char_make_comment:n` . . . . . 41, 2204, 2218  
`\char_make_end_group:N` . . . . . 41, 2188, 2190  
`\char_make_end_group:n` . . . . . 41, 2204, 2206  
`\char_make_end_line:N` . . . . . 41, 2188, 2193  
`\char_make_end_line:n` . . . . . 41, 2204, 2209  
`\char_make_escape:N` . . . . . 41, 2188, 2188  
`\char_make_escape:n` . . . . . 41, 2204, 2204  
`\char_make_ignore:N` . . . . .  
. . . . . 41, 2188, 2197, 4944, 4953, 4962  
`\char_make_ignore:n` . . . . . 41, 2204, 2213  
`\char_make_invalid:N` . . . . . 41, 2188, 2203

- \char\_make\_invalid:n . . . . 41, 2204, 2219
- \char\_make\_letter:N . . . . . 41, 2188, 2199, 4879, 4976
- \char\_make\_letter:n . . . . . 41, 2204, 2215
- \char\_make\_math\_shift:N . . . . 41, 2188, 2191
- \char\_make\_math\_shift:n . . . . 41, 2204, 2207
- \char\_make\_math\_subscript:N . . . . . 41, 2188, 2196
- \char\_make\_math\_subscript:n . . . . . 41, 2204, 2212
- \char\_make\_math\_superscript:N . . . . . 41, 2188, 2195
- \char\_make\_math\_superscript:n . . . . . 41, 2204, 2211
- \char\_make\_other:N . . . . . 41, 2188, 2200
- \char\_make\_other:n . . . . . 41, 2204, 2216
- \char\_make\_parameter:N . . . . 41, 2188, 2194
- \char\_make\_parameter:n . . . . 41, 2204, 2210
- \char\_make\_space:N . . . . . 41, 2188, 2198, 4924, 4929, 4934
- \char\_make\_space:n . . . . . 41, 2204, 2214
- \char\_set\_catcode:nn . . . . . 40, 2174, 2175, 2188–2219, 2272, 2274, 2278, 2283, 2368, 2454–2458, 5451, 5539, 5540, 5552
- \char\_set\_catcode:w . . . . 40, 2174, 2174, 2176
- \char\_set\_lccode:nn . . . . 41, 2236, 2237, 2361–2366, 2453, 4973, 5541, 5542
- \char\_set\_lccode:w . . . . . 41, 2236, 2236, 2238, 4974
- \char\_set\_mathcode:nn . . . . 42, 2220, 2221
- \char\_set\_mathcode:w . . . . 42, 2220, 2220, 2222
- \char\_set\_sfcode:nn . . . . . 42, 2256, 2257
- \char\_set\_sfcode:w . . . . 42, 2256, 2256, 2258
- \char\_set\_uccode:nn . . . . . 42, 2246, 2247
- \char\_set\_uccode:w . . . . 42, 2246, 2246, 2248
- \char\_show\_value\_catcode:n . . . . 40, 2174, 2185
- \char\_show\_value\_catcode:w . . . . . 40, 2174, 2182, 2186
- \char\_show\_value\_lccode:n . . . . 41, 2236, 2244
- \char\_show\_value\_lccode:w . . . . . 41, 2236, 2243, 2245
- \char\_show\_value\_mathcode:n . . . . . 42, 2220, 2233
- \char\_show\_value\_mathcode:w . . . . . 42, 2220, 2232, 2234
- \char\_show\_value\_sfcode:n . . . . 42, 2256, 2265
- \char\_show\_value\_sfcode:w . . . . . 42, 2256, 2264, 2266
- \char\_show\_value\_uccode:n . . . . 42, 2246, 2254
- \char\_show\_value\_uccode:w . . . . . 42, 2246, 2253, 2255
- \char\_value\_catcode:n . . . . 40, 2174, 2179
- \char\_value\_catcode:w . . . . 40, 2174, 2178, 2180
- \char\_value\_lccode:n . . . . . 41, 2236, 2241
- \char\_value\_lccode:w . . . . 41, 2236, 2240, 2241
- \char\_value\_mathcode:n . . . . 42, 2220, 2229
- \char\_value\_mathcode:w . . . . 42, 2220, 2228, 2230
- \char\_value\_sfcode:n . . . . . 42, 2256, 2261
- \char\_value\_sfcode:w . . . . 42, 2256, 2260, 2262
- \char\_value\_uccode:n . . . . . 42, 2246, 2251
- \char\_value\_uccode:w . . . . 42, 2246, 2250, 2251
- \chardef . . . . . 68
- \chk\_exist\_cs:N . . . . . 3469, 3473, 3476, 3479, 3492, 3496
- \chk\_global:N . . . . . 3476, 3479, 3497, 4094, 4104
- \chk\_if\_exist\_cs:c . . . . . 9, 1100, 1107
- \chk\_if\_exist\_cs:N . . . . 9, 1100, 1100, 1107, 1597
- \chk\_if\_free\_cs:N . . . . . 9, 1085, 1085, 1137, 1209, 1220, 2701, 2898, 3246, 3320, 3362, 3422, 3441, 3449, 4911, 4915, 4919, 5039, 5291
- \chk\_local:N . . . . . 3473, 4077, 4099
- \chk\_local\_or\_pref\_global:N . . . . . 2708, 2721, 2726, 2756, 2762, 3254, 3269, 3284, 3291, 3470, 3493, 3536, 3544, 3548, 3553, 3561, 3565, 3573, 3578, 3586, 3594, 3598, 3602, 4117, 4151, 4155, 4831
- \chk\_var\_or\_const:N . . . . . 3443, 3450, 3493, 3497, 3512, 3524, 4100, 4105
- \cleaders . . . . . 251
- \clearpage . . . . . 5504, 5507, 5512, 5515
- \clist\_clear:c . . . . . 84, 4395
- \clist\_clear:N . . . . . 84, 4395, 4395, 4396, 4566, 4588, 5057
- \clist\_clear\_new:c . . . . . 84, 4399
- \clist\_clear\_new:N . . . . 84, 4399, 4399, 4400
- \clist\_concat:ccc . . . . . 88, 4551
- \clist\_concat:NNN . . . . 88, 4551, 4560, 4562
- \clist\_concat\_aux:NNNN . . . . . 4551, 4551, 4560, 4561
- \clist\_display:c . . . . . 86, 4460
- \clist\_display:N . . . . 86, 4460, 4460, 4474
- \clist\_gclear:c . . . . . 84, 4395
- \clist\_gclear:N . . . . . 84, 4395, 4397, 4398
- \clist\_gclear\_new:c . . . . . 84, 4399
- \clist\_gclear\_new:N . . . . 84, 4399, 4401, 4402
- \clist\_gconcat:ccc . . . . . 88, 4551
- \clist\_gconcat:NNN . . . . 88, 4551, 4561, 4563
- \clist\_get:cN . . . . . 86, 4442, 4611
- \clist\_get:NN . . . . 86, 4442, 4442, 4447, 4610
- \clist\_get\_aux:w . . . . 90, 4442, 4444, 4446
- \clist\_gpop:cN . . . . . 89, 4604
- \clist\_gpop:NN . . . . . 89, 4604, 4608, 4609
- \clist\_gpush:cn . . . . . 89, 4604
- \clist\_gpush:Nn . . . . . 89, 4604, 4604, 4607
- \clist\_gpush:No . . . . . 89, 4604, 4606

\clist\_gpush:NV ..... 89, [4604](#), 4605  
\clist\_gput\_left:cn ..... 85, [4484](#)  
\clist\_gput\_left:co ..... 85, [4484](#)  
\clist\_gput\_left:cV ..... 85, [4484](#)  
\clist\_gput\_left:Nn .....  
..... 85, [4484](#), [4484](#), [4487](#), 4604  
\clist\_gput\_left:No ..... 85, [4484](#), 4606  
\clist\_gput\_left:NV ..... 85, [4484](#), 4605  
\clist\_gput\_left:Nx ..... 85, [4484](#)  
\clist\_gput\_right:cn ..... 86, [4492](#)  
\clist\_gput\_right:co ..... 86, [4492](#)  
\clist\_gput\_right:cV ..... 86, [4492](#)  
\clist\_gput\_right:Nn 86, [4492](#), [4492](#), [4495](#)  
\clist\_gput\_right:No ..... 86, [4492](#)  
\clist\_gput\_right:NV ..... 86, [4492](#)  
\clist\_gput\_right:Nx .....  
..... 86, [4492](#), 6256, 6263, 6267  
\clist\_gremove\_duplicates:N .....  
..... 89, [4565](#), 4578  
\clist\_gremove\_element:Nn 89, [4581](#), 4584  
\clist\_gset\_eq:cc ..... 85, [4407](#), 4410  
\clist\_gset\_eq:cN ..... 85, [4407](#), 4408  
\clist\_gset\_eq:Nc ..... 85, [4407](#), 4409  
\clist\_gset\_eq:NN 85, [4407](#), [4407](#), [4579](#), [4585](#)  
\clist\_if\_empty:c ..... 4412  
\clist\_if\_empty:cTF ..... 88, [4411](#)  
\clist\_if\_empty:N ..... 4411  
\clist\_if\_empty:Nf ..... 4497, 4516  
\clist\_if\_empty:NTF ..... 88, [4411](#), 4476  
\clist\_if\_empty\_err:N .....  
..... 90, [4413](#), [4413](#), [4443](#), 4449  
\clist\_if\_empty\_p:c ..... 88, [4411](#)  
\clist\_if\_empty\_p:N ..... 88, [4411](#)  
\clist\_if\_eq:cc ..... 4422  
\clist\_if\_eq:ccTF ..... 88, [4419](#)  
\clist\_if\_eq:cN ..... 4420  
\clist\_if\_eq:cNTF ..... 88, [4419](#)  
\clist\_if\_eq:Nc ..... 4421  
\clist\_if\_eq:NcTF ..... 88, [4419](#)  
\clist\_if\_eq:NN ..... 4419  
\clist\_if\_eq:NNTF ..... 88, [4419](#)  
\clist\_if\_eq\_p:cc ..... 88, [4419](#)  
\clist\_if\_eq\_p:cN ..... 88, [4419](#)  
\clist\_if\_eq\_p:Nc ..... 88, [4419](#)  
\clist\_if\_eq\_p:NN ..... 88, [4419](#)  
\clist\_if\_in:cnTF ..... 88, [4423](#)  
\clist\_if\_in:coTF ..... 88, [4423](#)  
\clist\_if\_in:cVTF ..... 88, [4423](#)  
\clist\_if\_in:Nn ..... 4423  
\clist\_if\_in:NnF ..... 4432, 4571, 4938  
\clist\_if\_in:NnT ..... 4431  
\clist\_if\_in:NnTF .. 88, [4423](#), 4430, 5061  
\clist\_if\_in:NoTF ..... 88, [4423](#)  
\clist\_if\_in:NVTF ..... 88, [4423](#)  
\clist\_map\_break: 87, [4513](#), [4513](#), 6196, 6230  
\clist\_map\_function:cN ..... 87, [4496](#)  
\clist\_map\_function:NN 87, [4496](#), [4496](#),  
4502, 4567, 4594, 6206, 6240, 6279  
\clist\_map\_function:nN .. 87, [4496](#), 4503  
\clist\_map\_function\_aux:Nw ... 4498,  
4505, 4508, 4508, 4511, 4520, 4532  
\clist\_map\_inline:cn ..... 87, [4514](#)  
\clist\_map\_inline:Nn .....  
..... 87, [4464](#), [4514](#), 4515, 4526  
\clist\_map\_inline:nn .... 87, [4514](#), 4527  
\clist\_map\_variable:cNn ..... 87, [4538](#)  
\clist\_map\_variable:NNn .....  
..... 87, [4538](#), 4544, 4545  
\clist\_map\_variable:nNn .....  
..... 87, [4538](#), 4538, 4544  
\clist\_map\_variable\_aux:Nnw .....  
..... 4540, [4546](#), 4546, 4549  
\clist\_new:c ..... 84, [4393](#)  
\clist\_new:N ..... 84, [4393](#), 4393,  
4394, 4564, 4892, 4895, 6175–6177  
\clist\_pop:cN ..... 89, [4598](#)  
\clist\_pop:NN ..... 89, [4598](#), 4602, 4603  
\clist\_pop\_aux:nnNN .....  
..... 90, [4448](#), [4448](#), 4602, 4608  
\clist\_pop\_aux:w ... 90, [4448](#), 4450, 4452  
\clist\_pop\_auxi:w .. 90, [4448](#), 4455, 4457  
\clist\_push:cn ..... 89, [4598](#), 4601  
\clist\_push:Nn ..... 89, [4598](#), 4598  
\clist\_push:No ..... 89, [4598](#), 4600  
\clist\_push:NV ..... 89, [4598](#), 4599  
\clist\_put\_aux:NNnnNn .....  
90, [4475](#), [4475](#), [4481](#), [4485](#), [4489](#), 4493  
\clist\_put\_left:cn ..... 85, [4480](#), 4601  
\clist\_put\_left:co ..... 85, [4480](#)  
\clist\_put\_left:cV ..... 85, [4480](#)  
\clist\_put\_left:Nn .....  
..... 85, [4480](#), [4480](#), [4483](#), 4598  
\clist\_put\_left:No ..... 85, [4480](#), 4600  
\clist\_put\_left:NV ..... 85, [4480](#), 4599  
\clist\_put\_left:Nx ..... 85, [4480](#)  
\clist\_put\_right:cn ..... 85, [4488](#)  
\clist\_put\_right:co ..... 85, [4488](#)  
\clist\_put\_right:cV ..... 85, [4488](#)  
\clist\_put\_right:Nn ..... 85, [4488](#),  
4488, 4491, 4572, 4591, 4939, 5064  
\clist\_put\_right:No ..... 85, [4488](#)  
\clist\_put\_right:NV . 85, [4488](#), 6289, 6290  
\clist\_put\_right:Nx ..... 85, [4488](#)  
\clist\_remove\_duplicates:N .....  
..... 89, [4565](#), 4575, 6277  
\clist\_remove\_duplicates\_aux:n ....  
..... 4565, 4567, 4570

- \clist\_remove\_duplicates\_aux:NN . . . . . 2986, 2987, 3041–3044, 3047, 3048, 3050, 3054, 3056, 3058, 3060, 3062, 3064, 3250, 3264, 3265, 3279, 3280, 3287, 3301, 3310, 3313, 3316, 3366, 3368, 3370, 3372, 3374, 3378, 3380, 3382, 3384, 3386, 3447, 3463, 3465, 3488, 3489, 3504, 3505, 3507, 3509, 3521, 3532, 3567–3569, 3604–3606, 3628–3631, 3635–3638, 3643–3650, 3657–3664, 3826, 3838, 3857, 3863, 3877, 3886, 3952–3954, 3961–3966, 3984, 3986, 4005, 4008, 4011, 4012, 4019, 4020, 4024, 4026, 4049–4052, 4073, 4075, 4078, 4093, 4096, 4113, 4114, 4124, 4125, 4134, 4135, 4137, 4142, 4148, 4162, 4176–4178, 4189–4192, 4197–4200, 4243, 4259–4261, 4269, 4295, 4303, 4306, 4325–4327, 4338, 4347, 4353, 4376, 4379, 4380, 4394, 4396, 4398, 4400, 4402, 4430–4432, 4441, 4447, 4474, 4483, 4487, 4491, 4495, 4502, 4526, 4545, 4562, 4563, 4603, 4607, 4609, 4655, 4666, 4670, 4699, 4700, 4728–4730, 4742, 4752, 4771, 4776, 4793, 4804, 4808, 5295, 5305–5312, 5316–5319, 5321, 5323, 5326, 5328, 5334, 5336, 5340–5342, 5344, 5346, 5348, 5356, 5358, 5362, 5364, 5377, 5379, 5382, 5384, 5388, 5390, 5394, 5399, 5404, 5406
- \clist\_remove\_element:Nn . . . . . 89, 4581, 4581
- \clist\_remove\_element\_aux:n . . . . . 4581, 4589, 4594, 4597
- \clist\_remove\_element\_aux:NNn . . . . . 4581, 4582, 4585, 4587
- \clist\_set\_eq:cc . . . . . 85, 4403, 4406
- \clist\_set\_eq:cN . . . . . 85, 4403, 4404
- \clist\_set\_eq:Nc . . . . . 85, 4403, 4405
- \clist\_set\_eq:NN 85, 4403, 4403, 4576, 4582
- \clist\_show:c . . . . . 86, 4458, 4459
- \clist\_show:N . . . . . 86, 4458, 4458
- \clist\_tmp:w . . . . . 4424, 4428
- \clist\_top:cN . . . . . 89, 4610, 4611
- \clist\_top:NN . . . . . 89, 4610, 4610
- \clist\_use:c . . . . . 86, 4433
- \clist\_use:N . . . . . 86, 4433, 4433, 4441
- \closein . . . . . 127
- \closeout . . . . . 122
- \clubpenalties . . . . . 435
- \clubpenalty . . . . . 262
- \copy . . . . . 319
- \count . . . . . 370
- \countdef . . . . . 69
- \cr . . . . . 94
- \crrcr . . . . . 95
- \cs:w . . . . . 10, 776, 779, 833, 867, 1030, 1316, 1355, 1462, 1512, 1525, 1526, 1528, 1530, 1532–1534, 1595, 1718, 1724, 1727, 2303, 2782, 3318, 3388, 3613, 5446
- \cs\_end: . . . . . 10, 776, 780, 833, 867, 1024, 1030, 1316, 1355, 1462, 1512, 1525, 1526, 1528, 1530, 1532, 1533, 1535, 1595, 1720, 1721, 1730–1732, 1734, 1736, 1738, 1740, 1742, 1744, 1746–1756, 2303, 2782, 3318, 3388, 3613, 5446
- \cs\_generate\_from\_arg\_count:cNnn . . . . . 955, 960, 965, 970, 1301, 1354
- \cs\_generate\_from\_arg\_count:NNnn . . . . . 1272, 1272, 1302, 1315
- \cs\_generate\_from\_arg\_count\_error\_msg:Nn . . . . . 1272, 1295, 1304
- \cs\_generate\_internal\_variant:n . . . . . 25, 1617, 1632, 1632
- \cs\_generate\_internal\_variant\_aux:n . . . . . 1635, 1638, 1638, 1643
- \cs\_generate\_variant:Nn . . . . . 24, 1596, 1596, 1853–1856, 1860, 1864, 1868, 1872, 1919, 1920, 2090, 2108, 2156–2163, 2705, 2717, 2718, 2745–2748, 2750, 2752, 2777–2780, 2960, 2986, 2987, 3041–3044, 3047, 3048, 3050, 3054, 3056, 3058, 3060, 3062, 3064, 3250, 3264, 3265, 3279, 3280, 3287, 3301, 3310, 3313, 3316, 3366, 3368, 3370, 3372, 3374, 3378, 3380, 3382, 3384, 3386, 3447, 3463, 3465, 3488, 3489, 3504, 3505, 3507, 3509, 3521, 3532, 3567–3569, 3604–3606, 3628–3631, 3635–3638, 3643–3650, 3657–3664, 3826, 3838, 3857, 3863, 3877, 3886, 3952–3954, 3961–3966, 3984, 3986, 4005, 4008, 4011, 4012, 4019, 4020, 4024, 4026, 4049–4052, 4073, 4075, 4078, 4093, 4096, 4113, 4114, 4124, 4125, 4134, 4135, 4137, 4142, 4148, 4162, 4176–4178, 4189–4192, 4197–4200, 4243, 4259–4261, 4269, 4295, 4303, 4306, 4325–4327, 4338, 4347, 4353, 4376, 4379, 4380, 4394, 4396, 4398, 4400, 4402, 4430–4432, 4441, 4447, 4474, 4483, 4487, 4491, 4495, 4502, 4526, 4545, 4562, 4563, 4603, 4607, 4609, 4655, 4666, 4670, 4699, 4700, 4728–4730, 4742, 4752, 4771, 4776, 4793, 4804, 4808, 5295, 5305–5312, 5316–5319, 5321, 5323, 5326, 5328, 5334, 5336, 5340–5342, 5344, 5346, 5348, 5356, 5358, 5362, 5364, 5377, 5379, 5382, 5384, 5388, 5390, 5394, 5399, 5404, 5406
- \cs\_generate\_variant\_aux:N . . . . . 1596, 1608, 1612, 1622, 1629, 1630
- \cs\_generate\_variant\_aux:nnNn . . . . . 1596, 1598, 1600
- \cs\_generate\_variant\_aux:nnw . . . . . 1596, 1601, 1603, 1627
- \cs\_get\_arg\_count\_from\_signature:c . . . . . 1269, 1356
- \cs\_get\_arg\_count\_from\_signature:N . . . . . 22, 898, 904, 911, 918, 1253, 1253, 1270, 1317
- \cs\_get\_arg\_count\_from\_signature\_aux:nnN . . . . . 1253, 1254, 1256
- \cs\_get\_arg\_count\_from\_signature\_auxii:w . . . . . 1253, 1263, 1268
- \cs\_get\_function\_name:N . . . . . 22, 1006, 1006
- \cs\_get\_function\_signature:N . . . . . 22, 1006, 1009
- \cs\_gnew:cn . . . . . 16, 1375
- \cs\_gnew:cpn . . . . . 15, 1168, 1174
- \cs\_gnew:cpx . . . . . 15, 1168, 1175
- \cs\_gnew:cx . . . . . 16, 1375
- \cs\_gnew:Nn . . . . . 16, 1336

- \cs\_gnew:Npn ..... [15](#), [1149](#), [1151](#),  
[1174](#), [5553](#), [5556](#), [5559](#), [5560](#), [5568](#)
- \cs\_gnew:Npx ..... [15](#), [1149](#), [1152](#), [1175](#)
- \cs\_gnew:Nx ..... [16](#), [1336](#)
- \cs\_gnew\_eq:cc ..... [20](#), [1219](#), [1225](#)
- \cs\_gnew\_eq:cN ..... [20](#), [1219](#), [1223](#)
- \cs\_gnew\_eq:Nc ..... [20](#), [1219](#), [1224](#)
- \cs\_gnew\_eq:NN .....  
..... [20](#), [1219](#), [1219](#), [1223–1225](#), [2268](#)
- \cs\_gnew\_nopar:cn ..... [16](#), [1375](#)
- \cs\_gnew\_nopar:cpn ..... [15](#), [1157](#), [1166](#)
- \cs\_gnew\_nopar:cpx ..... [15](#), [1157](#), [1167](#)
- \cs\_gnew\_nopar:cx ..... [16](#), [1375](#)
- \cs\_gnew\_nopar:Nn ..... [16](#), [1336](#)
- \cs\_gnew\_nopar:Npn .. [15](#), [1149](#), [1149](#), [1166](#)
- \cs\_gnew\_nopar:Npx .. [15](#), [1149](#), [1150](#), [1167](#)
- \cs\_gnew\_nopar:Nx ..... [16](#), [1336](#)
- \cs\_gnew\_protected:cn ..... [17](#), [1375](#)
- \cs\_gnew\_protected:cpn .. [15](#), [1184](#), [1190](#)
- \cs\_gnew\_protected:cpx .. [15](#), [1184](#), [1191](#)
- \cs\_gnew\_protected:cx ..... [17](#), [1375](#)
- \cs\_gnew\_protected:Nn ..... [17](#), [1336](#)
- \cs\_gnew\_protected:Npn [15](#), [1149](#), [1155](#), [1190](#)
- \cs\_gnew\_protected:Npx [15](#), [1149](#), [1156](#), [1191](#)
- \cs\_gnew\_protected:Nx ..... [17](#), [1336](#)
- \cs\_gnew\_protected\_nopar:cn ... [17](#), [1375](#)
- \cs\_gnew\_protected\_nopar:cpn .....  
..... [16](#), [1176](#), [1182](#)
- \cs\_gnew\_protected\_nopar:cpx .....  
..... [16](#), [1176](#), [1183](#)
- \cs\_gnew\_protected\_nopar:cx ... [17](#), [1375](#)
- \cs\_gnew\_protected\_nopar:Nn ... [17](#), [1336](#)
- \cs\_gnew\_protected\_nopar:Npn .....  
..... [16](#), [1149](#), [1153](#), [1182](#)
- \cs\_gnew\_protected\_nopar:Npx .....  
..... [16](#), [1149](#), [1154](#), [1183](#)
- \cs\_gnew\_protected\_nopar:Nx ... [17](#), [1336](#)
- \cs\_gset:cn ..... [19](#), [1359](#)
- \cs\_gset:cpn ..... [18](#), [1168](#),  
[1170](#), [3841](#), [3850](#), [4518](#), [4530](#), [4746](#)
- \cs\_gset:cpx ..... [18](#), [1168](#), [1171](#)
- \cs\_gset:cx ..... [19](#), [1359](#)
- \cs\_gset:Nn ..... [19](#), [1313](#)
- \cs\_gset:Npn [18](#), [813](#), [815](#), [1151](#), [1170](#), [1517](#)
- \cs\_gset:Npx .... [18](#), [813](#), [818](#), [1152](#), [1171](#)
- \cs\_gset:Nx ..... [19](#), [1313](#)
- \cs\_gset\_eq:cc .....  
..... [21](#), [1215](#), [1218](#), [1847](#), [4239](#), [4410](#)
- \cs\_gset\_eq:cN ..... [21](#), [1215](#), [1217](#),  
[1228](#), [1837](#), [1839](#), [1846](#), [4237](#), [4408](#)
- \cs\_gset\_eq:Nc .....  
..... [21](#), [1215](#), [1216](#), [1845](#), [4238](#), [4409](#)
- \cs\_gset\_eq:NN .....  
..... [21](#), [1215](#), [1215–1218](#), [1226](#), [1836](#),  
[1838](#), [1844](#), [3496](#), [3502](#), [4236](#), [4407](#)
- \cs\_gset\_nopar:cn ..... [19](#), [1359](#)
- \cs\_gset\_nopar:cpn .. [18](#), [1157](#), [1162](#), [1784](#)
- \cs\_gset\_nopar:cpx ..... [18](#), [1157](#), [1163](#)
- \cs\_gset\_nopar:cx ..... [19](#), [1359](#)
- \cs\_gset\_nopar:Nn ..... [19](#), [1313](#)
- \cs\_gset\_nopar:Npn .....  
..... [18](#), [813](#), [813](#), [816](#), [822](#),  
[828](#), [1149](#), [1162](#), [2284](#), [3445](#), [3476](#), [3485](#)
- \cs\_gset\_nopar:Npx .....  
..... [18](#), [813](#), [814](#), [819](#), [825](#),  
[831](#), [1150](#), [1163](#), [3451](#), [3479](#), [3486](#), [5975](#)
- \cs\_gset\_nopar:Nx ..... [19](#), [1313](#)
- \cs\_gset\_protected:cn ..... [20](#), [1359](#)
- \cs\_gset\_protected:cpn .. [18](#), [1184](#), [1186](#)
- \cs\_gset\_protected:cpx .. [18](#), [1184](#), [1187](#)
- \cs\_gset\_protected:cx ..... [20](#), [1359](#)
- \cs\_gset\_protected:Nn ..... [20](#), [1313](#)
- \cs\_gset\_protected:Npn .....  
..... [18](#), [813](#), [827](#), [1155](#), [1186](#)
- \cs\_gset\_protected:Npx .....  
..... [18](#), [813](#), [830](#), [1156](#), [1187](#)
- \cs\_gset\_protected:Nx ..... [20](#), [1313](#)
- \cs\_gset\_protected\_nopar:cn ... [20](#), [1359](#)
- \cs\_gset\_protected\_nopar:cpn .....  
..... [19](#), [1176](#), [1178](#)
- \cs\_gset\_protected\_nopar:cpx .....  
..... [19](#), [1176](#), [1179](#)
- \cs\_gset\_protected\_nopar:cx ... [20](#), [1359](#)
- \cs\_gset\_protected\_nopar:Nn ... [20](#), [1313](#)
- \cs\_gset\_protected\_nopar:Npn .....  
..... [19](#), [813](#), [821](#), [1153](#), [1178](#)
- \cs\_gset\_protected\_nopar:Npx .....  
..... [19](#), [813](#), [824](#), [1154](#), [1179](#)
- \cs\_gset\_protected\_nopar:Nx ... [20](#), [1313](#)
- \cs\_gundefine:c ..... [20](#), [1226](#), [1227](#)
- \cs\_gundefine:N ..... [20](#), [1226](#), [1226](#)
- \cs\_if\_do\_not\_use\_aux:nnN [1038](#), [1039](#), [1041](#)
- \cs\_if\_do\_not\_use\_p:N [9](#), [1038](#), [1038](#), [1051](#)
- \cs\_if\_eq:ccF ..... [1406](#)
- \cs\_if\_eq:ccT ..... [1405](#)
- \cs\_if\_eq:ccTF ..... [9](#), [1391](#), [1404](#)
- \cs\_if\_eq:cNF ..... [1398](#)
- \cs\_if\_eq:cNT ..... [1397](#)
- \cs\_if\_eq:cNTF ..... [9](#), [1391](#), [1396](#)
- \cs\_if\_eq:NcF ..... [1402](#)
- \cs\_if\_eq:NcT ..... [1401](#)
- \cs\_if\_eq:NcTF ..... [9](#), [1391](#), [1400](#)
- \cs\_if\_eq:NN ..... [1391](#)
- \cs\_if\_eq:NNF ..... [1398](#), [1402](#), [1406](#)
- \cs\_if\_eq:NNT ..... [1397](#), [1401](#), [1405](#)
- \cs\_if\_eq:NNTF . [9](#), [1391](#), [1396](#), [1400](#), [1404](#)



- \cs\_if\_eq\_name:NN ..... 1117  
 \cs\_if\_eq\_name\_p:NN ..... 8, 1117  
 \cs\_if\_eq\_p:cc ..... 9, 1391, 1403  
 \cs\_if\_eq\_p:cN ..... 9, 1391, 1395  
 \cs\_if\_eq\_p:Nc ..... 9, 1391, 1399  
 \cs\_if\_eq\_p:NN . 9, 1391, 1395, 1399, 1403  
 \cs\_if\_exist:c ..... 1023  
 \cs\_if\_exist:cF ..... 3720  
 \cs\_if\_exist:cT ..... 5135, 5230  
 \cs\_if\_exist:cTF ..... 9, 1012, 2114,  
 2137, 5065, 5072, 5128, 5223, 5436  
 \cs\_if\_exist:N ..... 1012  
 \cs\_if\_exist:Nf ..... 1101  
 \cs\_if\_exist:NT ..... 6201, 6235  
 \cs\_if\_exist:NTF ..... 9, 1012, 2353  
 \cs\_if\_exist\_p:c ..... 9, 1012  
 \cs\_if\_exist\_p:N 9, 1012, 1045, 3513, 3525  
 \cs\_if\_free:cF ..... 1062  
 \cs\_if\_free:cT ..... 1061, 1633  
 \cs\_if\_free:cTF .....  
 ..... 9, 1044, 1060, 1607, 5455, 6152  
 \cs\_if\_free:N ..... 1044  
 \cs\_if\_free:Nf ..... 1062, 1086  
 \cs\_if\_free:NT ..... 1061  
 \cs\_if\_free:NTF 9, 1044, 1060, 1474, 4791  
 \cs\_if\_free\_p:c ..... 9, 1044, 1059  
 \cs\_if\_free\_p:N ..... 9, 1044, 1059  
 \cs\_meaning:c ..... 10, 776, 782  
 \cs\_meaning:N ..... 10, 776, 781, 782  
 \cs\_new:cn ..... 16, 1375  
 \cs\_new:cpn ..... 14,  
1168, 1172, 1730–1733, 1735, 1737,  
 1739, 1741, 1743, 1745, 1747–1756  
 \cs\_new:cpx ..... 14, 1168, 1173, 1634  
 \cs\_new:cx ..... 16, 1375  
 \cs\_new:Nn 16, 1336, 4355, 4360, 4565,  
 4570, 4587, 4597, 4811, 4942, 4946,  
 4965, 4999, 5055, 5100, 6243, 6249  
 \cs\_new:Npn .....  
 .. 14, 879, 903, 1135, 1143, 1172,  
 1448, 1451, 1454, 1455, 1458, 1461,  
 1464, 1467, 1475, 1480, 1485, 1493,  
 1520, 1521, 1523, 1526, 1527, 1529,  
 1531, 1534, 1582, 1583, 1586, 1589,  
 1592, 1595, 1596, 1632, 1638, 1645,  
 1646, 1650, 1653, 1657, 1669, 1675,  
 1757, 1763, 1772, 1782, 1792, 1800,  
 1808, 1814, 1822, 1857, 1861, 1865,  
 1869, 1915, 1916, 1921, 1922, 1930,  
 1933, 1936, 1939, 1942, 1946, 1952,  
 1956, 1962, 1966, 1972, 1976, 1982,  
 2079, 2085, 2091, 2100, 2109, 2534,  
 2542, 2545, 2548, 2557, 2560, 2589,  
 3069, 3070, 3440, 3448, 3468, 3472,  
 3475, 3478, 3483–3486, 3533, 3538,  
 3541, 3546, 3550, 3555, 3558, 3563,  
 3570, 3575, 3580, 3583, 3588, 3591,  
 3654, 3666, 3669, 3672, 3675, 3678,  
 3681, 3684, 3687, 3690, 3693, 3696,  
 3699, 3702, 3827, 3834, 3839, 3848,  
 3859, 3864, 3870, 3873, 3881, 3892,  
 3893, 3903, 3919, 3929, 3937, 3968,  
 4009, 4010, 4013, 4016, 4021, 4023,  
 4025, 4029, 4030, 4040, 4082, 4085,  
 4088, 4089, 4149, 4153, 4164, 4170,  
 4180, 4220, 4221, 4266, 4270, 4274,  
 4296, 4299, 4300, 4304, 4307, 4316,  
 4328, 4334, 4446, 4448, 4452, 4457,  
 4475, 4503, 4508, 4515, 4527, 4538,  
 4546, 4656, 4662, 4665, 4667, 4669,  
 4671, 4673, 4679, 4685, 4691, 4701,  
 4703, 4705, 4712, 4714, 4955, 5290,  
 5329–5332, 5355, 5359, 5371, 5372,  
 5381, 5385, 5401, 5402, 5583, 5584,  
 5595, 5613, 5614, 5630, 5634, 5635,  
 5661, 5675, 5676, 5715, 5943, 5988,  
 6040, 6059, 6062, 6067, 6070, 6073,  
 6076, 6079, 6083, 6087, 6091, 6095  
 \cs\_new:Npx ..... 14, 1135, 1144, 1173  
 \cs\_new:Nx ..... 16, 1336  
 \cs\_new\_eq:cc ..... 20, 1208, 1214, 1412  
 \cs\_new\_eq:cN ..... 20, 1208, 1212, 1831  
 \cs\_new\_eq:Nc ..... 20, 1208, 1213  
 \cs\_new\_eq:NN ..... 20, 1208,  
 1208, 1212–1214, 1231, 1233, 1240,  
 1242, 1473, 1830, 1840–1847, 2174,  
 2220, 2236, 2246, 2256, 2269, 2270,  
 2691–2693, 2781, 2783, 3035, 3036,  
 3057, 3059, 3065–3068, 3309, 3312,  
 3315, 3317, 3385, 3387, 3395, 3396,  
 3398, 3432, 3501, 3502, 3520, 3531,  
 3665, 3820–3822, 3869, 4022, 4027,  
 4028, 4074, 4081, 4095, 4110, 4111,  
 4136, 4222–4239, 4279, 4280, 4340,  
 4341, 4371–4374, 4377, 4381, 4382,  
 4393, 4395, 4397, 4399, 4401, 4403–  
 4410, 4458, 4459, 4513, 4598–4601,  
 4604–4606, 4610, 4611, 4623–4638,  
 4753, 4778–4781, 4825, 5296–5298,  
 5324, 5337–5339, 5343, 5345, 5350,  
 5367, 5370, 5376, 5378, 5395, 5400,  
 5403, 5405, 6082, 6086, 6090, 6094  
 \cs\_new\_nopar:cn ..... 16, 1375  
 \cs\_new\_nopar:cpn 15, 1157, 1164, 1896–1901  
 \cs\_new\_nopar:cpx 15, 1157, 1165, 1611, 2603  
 \cs\_new\_nopar:cx ..... 16, 1375  
 \cs\_new\_nopar:Nn ..... 16,  
1336, 4896, 4899, 4905–4909, 5043,

- 5078–5080, 5089, 5167, 5170, 5173,  
5194, 6190, 6214, 6215, 6225, 6242,  
6255, 6266, 6270, 6273, 6276, 6282
- \cs\_new\_nopar:Npn . . . . . 15, 1135,  
1141, 1158, 1164, 1212–1214, 1216–  
1218, 1223–1227, 1252, 1395–1407,  
1447, 1472, 1663–1666, 1672, 1704,  
1706, 1707, 1717, 1723, 1726, 1729,  
1830–1839, 1891, 1895, 1983, 2071,  
2175, 2178, 2179, 2182, 2185, 2188–  
2219, 2221, 2225, 2228, 2229, 2232,  
2233, 2237, 2240, 2241, 2243, 2244,  
2247, 2250, 2251, 2253, 2254, 2257,  
2260, 2261, 2264, 2265, 2268, 2345,  
2375, 2382, 2393, 2404, 2415, 2426,  
2433, 2440, 2447, 2461, 2464, 2470,  
2476, 2502, 2526, 2527, 2532, 2567,  
2574, 2581, 2597, 2602, 2671, 2674,  
2696, 2697, 2700, 2706, 2711, 2719,  
2724, 2729, 2735, 2749, 2751, 2753,  
2759, 2765, 2771, 2782, 2784–2786,  
2798, 2802, 2806, 2820, 2829, 2838,  
2842, 2846, 2857, 2870, 2893, 2935,  
2938, 2941, 2944, 2973, 3006, 3009,  
3045, 3046, 3049, 3051, 3055, 3061,  
3063, 3241, 3242, 3245, 3251, 3257,  
3266, 3272, 3281, 3288, 3294, 3302,  
3311, 3314, 3318, 3319, 3343, 3357,  
3358, 3361, 3367, 3369, 3371, 3373,  
3375, 3379, 3381, 3383, 3388, 3397,  
3403, 3406, 3409, 3412, 3417, 3418,  
3421, 3426–3431, 3453–3455, 3464,  
3491, 3495, 3506, 3508, 3511, 3523,  
3607, 3613, 3823, 3825, 3830, 3862,  
3878, 3887, 3983, 3985, 4003, 4006,  
4069, 4070, 4077, 4094, 4098, 4103,  
4115, 4119, 4126, 4129, 4138, 4143,  
4157, 4240, 4246, 4262, 4281, 4342,  
4348, 4365, 4368, 4375, 4378, 4413,  
4433, 4442, 4460, 4480, 4484, 4488,  
4492, 4496, 4544, 4551, 4560, 4561,  
4575, 4578, 4581, 4584, 4602, 4608,  
4639, 4725, 4744, 4766, 4772, 4777,  
4782, 4783, 4787, 4789, 4790, 4794,  
4797, 4800, 4805, 4809, 4810, 4817,  
4821, 4829, 4833, 4910, 4914, 4918,  
4922, 4927, 4932, 4937, 5020, 5023,  
5026, 5029, 5032, 5035, 5038, 5176,  
5179, 5182, 5185, 5188, 5191, 5286,  
5287, 5320, 5322, 5325, 5327, 5333,  
5335, 5354, 5357, 5363, 5365, 5368,  
5373, 5380, 5383, 5389, 5391, 5396,  
5419–5421, 5435, 5454, 5544, 5547,  
5677, 5681, 5685, 5700, 5701, 5730,  
5733, 5736, 5739, 5742, 5745, 5748,  
5751, 5754, 5757, 5760, 5763, 5766,  
5769, 5772, 5775, 5778, 5781, 5784,  
5788, 5792, 5796, 5800, 5804, 5808,  
5821, 5826, 5827, 5831, 5835, 5854,  
5870, 5886, 5892, 5893, 5895, 5904,  
5913, 5921, 5930, 5939, 5941, 6003,  
6004, 6050, 6052, 6054, 6097, 6151
- \cs\_new\_nopar:Npx . . . 15, 1135, 1142, 1165  
\cs\_new\_nopar:Nx . . . . . 16, 1336  
\cs\_new\_protected:cn 16, 1375, 5195, 5198  
\cs\_new\_protected:cpn . . . 15, 1184, 1188  
\cs\_new\_protected:cpx . . . 15, 1184, 1189  
\cs\_new\_protected:cx . . . . . 16, 1375  
\cs\_new\_protected:Nn . . . . .  
16, 1336, 4979, 5006, 5013, 5202, 5214  
\cs\_new\_protected:Npn . . . . .  
. . . . . 15, 891, 917, 1135,  
1147, 1188, 1208, 1215, 1219, 6011  
\cs\_new\_protected:Npx 15, 1135, 1148, 1189  
\cs\_new\_protected:Nx . . . . . 16, 1336  
\cs\_new\_protected\_nopar:cn . . . . 17, 1375  
\cs\_new\_protected\_nopar:cpn . . . . .  
. . . . . 15, 1176, 1180  
\cs\_new\_protected\_nopar:cpx . . . . .  
. . . . . 15, 1176, 1181  
\cs\_new\_protected\_nopar:cx . . . . 17, 1375  
\cs\_new\_protected\_nopar:Nn . . . . 17, 1336  
\cs\_new\_protected\_nopar:Npn . . . . .  
. . . . . 15, 1135, 1145,  
1180, 2224, 5236, 5240, 5449, 5989  
\cs\_new\_protected\_nopar:Npx . . . . .  
. . . . . 15, 1135, 1146, 1181  
\cs\_new\_protected\_nopar:Nx . . . . 17, 1336  
\cs\_record\_meaning:N 13, 1082, 1083, 1094  
\cs\_set:cn . . . . . 19, 1359,  
4943, 4947, 4951, 4956, 4960, 4969  
\cs\_set:cpn . . . . . 17, 1168, 1168,  
1884, 1887, 1890, 1902, 1903, 3107,  
3111, 3115, 3119, 3123, 3127, 3131  
\cs\_set:cpx . . . . . 17, 1168,  
1169, 1984, 1988, 1992, 1996, 2000,  
2009, 2018, 2027, 2036, 2038, 2040,  
2042, 2044, 2046, 2048, 2050, 2052  
\cs\_set:cx . . . . . 19, 1359  
\cs\_set:Nn . . . . 19, 1313, 4589, 5056, 5060  
\cs\_set:Npn . . . . .  
. . . 17, 793, 795, 833–870, 873, 876,  
882, 888, 894, 897, 900, 907, 914,  
920–922, 927, 934, 939, 944, 949,  
954, 959, 964, 969, 983, 989, 993,  
1002, 1006, 1009, 1038, 1041, 1083,  
1132, 1135, 1143, 1157, 1168, 1253,  
1256, 1268, 1272, 1304, 1313, 1352,



- 1408, 1411, 1414, 1417, 1428–1431,  
1525, 1600, 1603, 1629, 1873, 1877,  
1904, 1907, 1912, 2059, 2060, 3090,  
3097, 3101, 3139, 3146, 3156, 3166,  
3169, 3193, 3194, 3208, 3211, 3214,  
3217, 3220, 3223, 3226, 3229, 3596,  
3600, 3913, 3947, 3956, 3969, 3973,  
3987, 3989, 4253, 4343, 4349, 4424,  
4657, 4735, 4986, 5008, 5015, 5602  
\cs\_set:Npx . . . . . 17, 793, 798, 1144, 1169  
\cs\_set:Nx . . . . . 19, 1313  
\cs\_set\_eq:cc . . . . .  
21, 1204, 1207, 1409, 1843, 4235, 4406  
\cs\_set\_eq:cN 21, 1204, 1205, 1833, 1835,  
1842, 4233, 4404, 4949, 4958, 4967  
\cs\_set\_eq:Nc . . . . .  
. . . . . 21, 1204, 1206, 1841, 4234, 4405  
\cs\_set\_eq:NN . . . . . 21,  
1204, 1204–1207, 1210, 1215, 1221,  
1832, 1834, 1840, 2535, 2549, 2565,  
2605, 2677, 3084, 3085, 3087–3089,  
3334, 3335, 3466, 3492, 3501, 4072,  
4232, 4345, 4351, 4403, 4769, 4803,  
4819, 4820, 4981, 4982, 5009, 5016,  
5347, 5349, 5497, 5678, 5682, 5686,  
5716, 5717, 5720, 5721, 5836–5842,  
5844, 5856, 5859, 5862, 5872, 5875,  
5878, 5916, 5917, 5950–5952, 5957–  
5960, 6065, 6066, 6098–6103, 6142,  
6154–6156, 6160–6162, 6202, 6236  
\cs\_set\_eq:NwN . . . . .  
. . . . . 21, 748, 754–781, 783, 786–  
794, 813, 814, 1064, 1204, 1443–1445  
\cs\_set\_nopar:cn . . . . . 19, 1359  
\cs\_set\_nopar:cpn 18, 1157, 1160, 1192–  
1200, 1202, 2787, 2788, 5428, 6164  
\cs\_set\_nopar:cpx . . . . . 18, 1157, 1161  
\cs\_set\_nopar:cx . . . . . 19, 1359  
\cs\_set\_nopar:Nn . . . . . 19, 1313, 6192, 6226  
\cs\_set\_nopar:Npn 18, 737, 793, 793, 795,  
796, 798, 801, 802, 804, 808, 976,  
1059–1062, 1072, 1075, 1078, 1085,  
1100, 1107, 1111, 1128, 1141, 1160,  
1205–1207, 1269, 1301, 1501, 1511,  
1536–1543, 1545–1551, 1553–1555,  
1557, 1559–1561, 1563–1566, 1568–  
1571, 1573–1578, 1580, 1581, 1817,  
1825, 2741–2744, 2810, 2888, 2890,  
2961, 2988, 3000, 3017, 3037–3040,  
3469, 3483, 3705, 4547, 4676, 4695,  
4706, 4708, 4731, 4786, 4788, 4984,  
5444, 5460, 5478, 5479, 5488, 5489,  
5492, 5495, 5864, 5880, 5947, 5953  
\cs\_set\_nopar:Npx 18, 793, 794, 799, 805,  
811, 1142, 1161, 1476, 3473, 3484, 5461  
\cs\_set\_nopar:Nx . . . . . 19, 1313  
\cs\_set\_protected:cn . . . . .  
. . . . . 19, 1359, 5045, 5048, 5051  
\cs\_set\_protected:cpn . . . . . 18, 1184, 1184  
\cs\_set\_protected:cpx . . . . .  
. . . . . 18, 1184, 1185, 1314, 1353  
\cs\_set\_protected:cx . . . . . 19, 1359  
\cs\_set\_protected:Nn . . . . . 19, 1313, 5268  
\cs\_set\_protected:Npn . . . . .  
18, 793, 807, 885, 910, 1147, 1184,  
1204, 3086, 5448, 5948, 5966, 6005  
\cs\_set\_protected:Npx . . . . .  
. . . . . 18, 793, 810, 1148, 1185  
\cs\_set\_protected:Nx . . . . . 19, 1313  
\cs\_set\_protected\_nopar:cn . . . . . 20, 1359  
\cs\_set\_protected\_nopar:cpn . . . . .  
. . . . . 18, 1176, 1176, 5425  
\cs\_set\_protected\_nopar:cpx . . . . .  
. . . . . 18, 1176, 1177  
\cs\_set\_protected\_nopar:cx . . . . . 20, 1359  
\cs\_set\_protected\_nopar:Nn . . . . . 20, 1313  
\cs\_set\_protected\_nopar:Npn . . . . .  
. . . . . 18, 793, 801, 807, 810, 815,  
818, 821, 824, 827, 830, 1136, 1145,  
1176, 6104–6107, 6113, 6118, 6133  
\cs\_set\_protected\_nopar:Npx . . . . .  
. . . . . 18, 793, 804, 1146, 1177  
\cs\_set\_protected\_nopar:Nx . . . . . 20, 1313  
\cs\_show:c . . . . . 10, 776, 784, 3068  
\cs\_show:N . . . . . 10, 776, 783, 784, 3067, 3464  
\cs\_split\_function:NN . . . . .  
. . . . . 22, 872, 878, 884, 890,  
896, 902, 909, 916, 984, 989, 1007,  
1010, 1039, 1254, 1422, 1424, 1598  
\cs\_split\_function\_aux:w . . . . . 984, 990, 993  
\cs\_split\_function\_auxii:w 984, 1000, 1002  
\cs\_str\_aux:w . . . . . 977, 983  
\cs\_tmp:w . . . . . 1135, 1141–1157, 1160–  
1191, 1313, 1320–1352, 1359–1390  
\cs\_to\_str:N 3, 21, 976, 976, 991, 4811–4814  
\cs\_to\_str\_aux:w . . . . . 976  
\csname . . . . . 11, 157  
\currentgrouplevel . . . . . 409  
\currentgrouptype . . . . . 410  
\currentifbranch . . . . . 406  
\currentiflevel . . . . . 405  
\currentifttype . . . . . 407
- D**
- \d . . . . . 2365  
\dagger . . . . . 2860, 2866  
\day . . . . . 365

- \ddagger ..... 2861, 2867
  - \deadcycles ..... 299
  - \DeclareOption ..... 47, 50
  - \def ..... 3–6, 24, 43, 48, 51, 56, 64
  - \def\_long\_test\_function\_new:npn .. 4047
  - \default@ds ..... 707
  - \defaultthyphenchar ..... 349
  - \defaultskewchar ..... 350
  - \DefineCrossReferences ..... 5479, 5485
  - \delcode ..... 380
  - \delimiter ..... 174
  - \delimiterfactor ..... 223
  - \delimitershortfall ..... 222
  - \depthof ..... 111, 6098, 6098
  - \detokenize ..... 397
  - \dim\_add:cn ..... 62, 3375
  - \dim\_add:Nc ..... 62, 3375
  - \dim\_add:Nn 62, 3375, 3375, 3378, 3379, 5755
  - \dim\_compare:nNn ..... 3399
  - \dim\_compare:nNnF ..... 3407, 3413
  - \dim\_compare:nNnT ..... 3404, 3410
  - \dim\_compare:nNnTF ..... 63, 1958, 3399
  - \dim\_compare\_p:nNn ..... 63, 3399
  - \dim\_do\_until:nNnn .. 63, 3403, 3412, 3413
  - \dim\_do\_while:nNnn .. 63, 3403, 3409, 3410
  - \dim\_eval:n ..... 63,  
1953, 3367, 3376, 3397, 3397, 3400,  
5329–5332, 5371, 5386, 5998, 5999
  - \dim\_gadd:cn ..... 62, 3375
  - \dim\_gadd:Nn ... 62, 3375, 3379, 3380, 5758
  - \dim\_gset:cc ..... 62, 3367
  - \dim\_gset:cn ..... 62, 3367
  - \dim\_gset:Nc ..... 62, 3367
  - \dim\_gset:Nn 62, 3367, 3369, 3370, 5752, 5955
  - \dim\_gsub:cn ..... 62, 3381
  - \dim\_gsub:Nn ... 62, 3381, 3383, 3384, 5764
  - \dim\_gzero:c ..... 61, 3371
  - \dim\_gzero:N ..... 61, 3371, 3373, 3374
  - \dim\_new:c ..... 61, 3355
  - \dim\_new:N ..... 61, 3355, 3357,  
3361, 3366, 3389–3394, 5706–5708
  - \dim\_new\_l:N ..... 3358
  - \dim\_set:cn ..... 62, 3367
  - \dim\_set:Nc ..... 62, 3367
  - \dim\_set:Nn .... 62, 3367, 3367–3369, 5749
  - \dim\_show:c ..... 62, 3387, 3388
  - \dim\_show:N ..... 62, 3387, 3387, 3388
  - \dim\_sub:cn ..... 62, 3381
  - \dim\_sub:Nc ..... 62, 3381
  - \dim\_sub:Nn .... 62, 3381, 3381–3383, 5761
  - \dim\_until\_do:nNnn .. 63, 3403, 3406, 3407
  - \dim\_use:c ..... 62, 3385
  - \dim\_use:N .... 62, 1953, 3385, 3385, 3386
  - \dim\_while\_do:nNnn .. 63, 3403, 3403, 3404
  - \dim\_zero:c ..... 61, 3371
  - \dim\_zero:N ..... 61, 3371, 3371–3373
  - \dimen ..... 371
  - \dimendef ..... 70
  - \dimexpr ..... 424
  - \directlua ..... 553
  - \discretionary ..... 234
  - \displayindent ..... 199
  - \displaylimits ..... 209
  - \displaystyle ..... 187
  - \displaywidowpenalties ..... 437
  - \displaywidowpenalty ..... 198
  - \displaywidth ..... 200
  - \divide ..... 77
  - \documentclass ..... 5475
  - \doublehyphendemerits ..... 267
  - \dp ..... 378
  - \ds@ ..... 708
  - \dump ..... 361
- E**
- \E ..... 2367
  - \edef ..... 22, 65
  - \efcode ..... 462
  - \else ..... 12, 118
  - \else: ..... 7, 741, 755, 758, 855, 936,  
941, 946, 956, 961, 966, 978, 996,  
1015, 1018, 1026, 1032, 1047, 1053,  
1113, 1114, 1116, 1133, 1232, 1237,  
1241, 1246, 1250, 1259, 1293, 1393,  
1506, 1690, 1694, 1698, 1702, 1851,  
1880, 1928, 2095, 2103, 2112, 2120,  
2130, 2135, 2142, 2152, 2288, 2292,  
2296, 2300, 2304, 2308, 2312, 2316,  
2320, 2324, 2328, 2332, 2336, 2340,  
2347, 2351, 2355, 2388, 2399, 2410,  
2421, 2487, 2490, 2494, 2497, 2504,  
2506, 2508, 2510, 2512, 2514, 2570,  
2577, 2592, 2998, 3099, 3109, 3113,  
3117, 3121, 3125, 3129, 3133, 3137,  
3151, 3161, 3173, 3178, 3181, 3184,  
3202, 3206, 3401, 3459, 3515, 3527,  
3626, 3633, 3641, 3655, 3711, 3725,  
3732, 3739, 3746, 3753, 3760, 3767,  
3774, 3781, 3788, 3795, 3802, 3809,  
3816, 4033, 4038, 4043, 4056, 4255,  
4426, 4437, 4827, 5300, 5303, 5314,  
5811, 5813, 5815, 5836–5841, 5843,  
5857, 5860, 5863, 5873, 5876, 5879,  
5971, 6018, 6024, 6030, 6031, 6036
  - \emergencystretch ..... 282
  - \end ..... 156, 5522
  - \endcsname ..... 11, 158
  - \endgroup ..... 13, 17, 91

- `\endinput` ..... 18, 130
- `\endL` ..... 445
- `\endlinechar` ..... 29, 37, 172
- `\endR` ..... 447
- `\eqno` ..... 192
- `\errhelp` ..... 14, 138
- `\errmessage` ..... 15, 132
- `\erroneous` ..... 1518
- `\ERROR` ... 2059, 2060, 4220, 4221, 4345,  
4351, 5613, 5627, 5672, 5675, 5676
- `\errorcontextlines` ..... 139
- `\errorstopmode` ..... 153
- `\escapechar` ..... 171
- `\etex_beginL:D` ..... 444
- `\etex_beginR:D` ..... 446
- `\etex_botmarks:D` ..... 393
- `\etex_clubpenalties:D` ..... 435
- `\etex_currentgrouplevel:D` ..... 409
- `\etex_currentgrouptype:D` ..... 410, 1708
- `\etex_currentifbranch:D` ..... 406
- `\etex_currentiflevel:D` ..... 405
- `\etex_currentifttype:D` ..... 407
- `\etex_detokenize:D` ..... 397, 3822
- `\etex_dimexpr:D` ..... 424, 3397,  
5899, 5908, 5925, 5934, 5982, 5994
- `\etex_displaywidowpenalties:D` ..... 437
- `\etex_endL:D` ..... 445
- `\etex_endR:D` ..... 447
- `\etex_eTeXrevision:D` ..... 389
- `\etex_eTeXversion:D` ..... 388
- `\etex_everyeof:D` 449, 3905, 3921, 3931, 3939
- `\etex_firstmarks:D` ..... 392
- `\etex_fontcharhp:D` ..... 417
- `\etex_fontcharht:D` ..... 416
- `\etex_fontcharic:D` ..... 419
- `\etex_fontcharwd:D` ..... 418
- `\etex_glueexpr:D` ..... 425, 3319,  
5900, 5909, 5926, 5935, 5983, 5995
- `\etex_glueshrink:D` ..... 428, 3352
- `\etex_glueshrinkorder:D` ..... 430, 3340
- `\etex_gluestretch:D` ..... 427, 3351
- `\etex_gluestretchorder:D` ..... 429, 3339
- `\etex_gluetomu:D` ..... 431
- `\etex_ifcsname:D` ..... 386, 772
- `\etex_ifdefined:D` .... 385, 554, 555, 771
- `\etex_iffontchar:D` ..... 415
- `\etex_interactionmode:D` ..... 413
- `\etex_interlinepenalties:D` ..... 434
- `\etex_lastlinefit:D` ..... 433
- `\etex_lastnodetype:D` .... 414, 1710, 1712
- `\etex_marks:D` ..... 390
- `\etex_middle:D` ..... 438
- `\etex_muexpr:D` . 426, 3426, 3428, 3430,  
5901, 5910, 5927, 5936, 5984, 5996
- `\etex_mutoglu:D` ..... 432, 5972, 5973
- `\etex_numexpr:D` 423, 1273, 1307, 3085,  
5898, 5907, 5924, 5933, 5981, 5993
- `\etex_pagediscards:D` ..... 441
- `\etex_parshapedimen:D` ..... 422
- `\etex_parshapeindent:D` ..... 420
- `\etex_parshapelength:D` ..... 421
- `\etex_predisplaydirection:D` ..... 448
- `\etex_protected:D` ..... 450, 792
- `\etex_readline:D` ..... 400
- `\etex_savinghyphcodes:D` ..... 439
- `\etex_savingvdiscards:D` ..... 440
- `\etex_scantokens:D` .....  
398, 3908, 3924, 3934, 3942
- `\etex_showgroups:D` ..... 411
- `\etex_showifs:D` ..... 412
- `\etex_showtokens:D` ..... 399, 3466
- `\etex_splitbotmarks:D` ..... 395
- `\etex_splitdiscards:D` ..... 442
- `\etex_splitfirstmarks:D` ..... 394
- `\etex_TeXxETstate:D` ..... 443
- `\etex_topmarks:D` ..... 391
- `\etex_tracingassigns:D` ..... 401
- `\etex_tracinggroups:D` ..... 408
- `\etex_tracingifs:D` ..... 404
- `\etex_tracingnesting:D` ..... 403
- `\etex_tracingscantokens:D` ..... 402
- `\etex_unexpanded:D` .....  
396, 563, 660, 664, 667, 775, 1445
- `\etex_unless:D` ..... 387, 760
- `\etex_widowpenalties:D` ..... 436
- `\etexmissingerror` ..... 5, 15, 16
- `\etexmissinghelp` ..... 6, 14, 16
- `\eTeXrevision` ..... 389
- `\eTeXversion` ..... 388
- `\everycr` ..... 100
- `\everydisplay` ..... 201
- `\everyeof` ..... 449
- `\everyhbox` ..... 340
- `\everyjob` ..... 369
- `\everymath` ..... 225
- `\everypar` ..... 288
- `\everyvbox` ..... 341
- `\ExecuteOptions` ..... 52
- `\exhyphenpenalty` ..... 264
- `\exp_after:cc` ..... 5444
- `\exp_after:wN` . 13, 156, 773, 773, 867–  
869, 929, 936, 941, 946, 951, 956,  
961, 966, 971, 977, 979, 981, 990,  
995, 997, 1025, 1027, 1030, 1042,  
1046, 1048, 1118–1123, 1129, 1258,  
1260, 1263, 1316, 1355, 1419, 1442,  
1443, 1462, 1465, 1468, 1469, 1477,  
1481, 1482, 1486, 1487, 1494, 1495,

- 1502, 1507, 1509, 1512, 1520–1527,  
 1529–1532, 1534, 1535, 1582, 1584,  
 1587, 1590, 1593, 1595, 1605, 1630,  
 1641, 1647, 1648, 1651, 1654, 1655,  
 1658, 1659, 1670, 1676, 1719, 1885,  
 1888, 1890, 1892, 2080, 2082, 2087,  
 2092, 2094, 2096, 2102, 2104, 2303,  
 2343, 2354, 2372, 2379, 2389, 2400,  
 2411, 2422, 2430, 2437, 2444, 2466,  
 2472, 2478, 2569, 2571, 2576, 2578,  
 2587, 2591, 2593, 2799, 2803, 2990–  
 2998, 3094, 3098, 3460, 3640, 3652,  
 3679, 3685, 3691, 3697, 3703, 3823,  
 3831, 3832, 3908, 3913–3916, 3942,  
 3950, 3980, 3999, 4032, 4036, 4041,  
 4054, 4083, 4086, 4088, 4090, 4139,  
 4150, 4154, 4165, 4166, 4171, 4172,  
 4181, 4182, 4257, 4264, 4272, 4297,  
 4336, 4438, 4444, 4450, 4498, 4499,  
 4660, 4732, 4733, 4737, 5438, 5445,  
 5446, 5565, 5577, 5620, 5621, 5638,  
 5639, 5649, 5654, 5723, 5725, 5810  
 \exp\_arg:x . . . . . 26, 1473, 1473, 1474, 1482  
 \exp\_arg\_last\_unbraced:nn . . . . .  
 . . . 1645, 1645, 1647, 1651, 1654, 1658  
 \exp\_arg\_next:nnn . . . . . 1448, 1448,  
 1465, 1468, 1477, 1481, 1486, 1494  
 \exp\_arg\_next\_nobrace:nnn 1448, 1451, 1462  
 \exp\_args:cc . . . . . 26, 1525, 1526  
 \exp\_args:Nc . . . . . 26, 782,  
 784, 785, 867, 867, 935, 940, 945,  
 950, 1059–1062, 1107, 1158, 1205,  
 1212, 1217, 1223, 1270, 1302, 1395–  
 1398, 1525, 1525, 2784, 3843, 4532,  
 4911, 4915, 4919, 5039, 6115, 6125  
 \exp\_args:Ncc . . . . . 27, 1207, 1214,  
 1218, 1225, 1403–1406, 1525, 1529  
 \exp\_args:Nccc . . . . . 28, 1525, 1531  
 \exp\_args:Ncco . . . . . 28, 1561, 1575, 1576  
 \exp\_args:Nccx . . . . . 28, 1561, 1577  
 \exp\_args:Ncf . . . 27, 1540, 1548, 1786, 1787  
 \exp\_args:NcNc . . . . . 28, 1561, 1573, 5438  
 \exp\_args:NcNo . . . . . 28, 1561, 1574  
 \exp\_args:Ncnx . . . . . 28, 1561, 1578  
 \exp\_args:Nco . . . . . 27, 1534, 1534, 1540  
 \exp\_args:Ncx . . . . . 27, 1540, 1545  
 \exp\_args:Nf . . . . .  
 . 27, 1536, 1536, 1759, 1760, 1767,  
 1776, 1810, 1811, 1818, 1826, 1943,  
 2813, 2816, 2818, 2889, 3003, 3012  
 \exp\_args:Nff . . . . . 27, 1540, 1547, 2891  
 \exp\_args:Nfo . . . . . 27, 1540, 1546  
 \exp\_args:NNc 27, 1206, 1213, 1216, 1224,  
 1399–1402, 1422, 1424, 1525, 1527  
 \exp\_args:NNf . . . 27, 1540, 1540, 1796, 1804  
 \exp\_args:Nnf . . . . .  
 . . . . . 27, 895, 901, 908, 915, 1540, 1549  
 \exp\_args:Nnnc . . . . . 28, 1561, 1568  
 \exp\_args:NNNo . . . . . 28, 1520, 1523  
 \exp\_args:NNno . . . . . 28, 1561, 1563  
 \exp\_args:Nnno . . . . . 28, 1561, 1569  
 \exp\_args:NNNV . . 28, 1561, 1561, 3909, 3925  
 \exp\_args:NNnx . . . . . 28, 1561, 1564  
 \exp\_args:Nnnx . . . . . 28, 1561, 1570  
 \exp\_args:NNo . . . . .  
 . . . . . 27, 1520, 1521, 2126, 2148, 5576  
 \exp\_args:Nno . . . . . 27, 1540, 1550  
 \exp\_args:NNoo . . . . . 28, 1561, 1565  
 \exp\_args:NNox . . . . . 28, 1561, 1566  
 \exp\_args:Nnox . . . . . 28, 1561, 1571  
 \exp\_args:NNV . . . . . 27, 1540, 1542  
 \exp\_args:NNv . . . . . 27, 1540, 1541  
 \exp\_args:NNx . . . . . 27, 1540, 1543, 5462  
 \exp\_args:Nnx . . . . . 27, 1540, 1551  
 \exp\_args:No . . . . .  
 26, 1520, 1520, 1953, 3862, 4544, 5954  
 \exp\_args:Noc . . . . . 27, 1540, 1553  
 \exp\_args:Noo . . . . . 27, 1540, 1554  
 \exp\_args:Nooo . . . . . 28, 1561, 1581  
 \exp\_args:Noox . . . . . 28, 1561, 1580  
 \exp\_args:Nox . . . . . 27, 1540, 1555  
 \exp\_args:NV . . . . . 26, 1536, 1538, 3943  
 \exp\_args:Nv . . . . . 26, 1536, 1537  
 \exp\_args:NVV . . . . . 27, 1540, 1557  
 \exp\_args:Nx . . . . . 27, 1536, 1539  
 \exp\_args:Nxo . . . . . 27, 1540, 1559  
 \exp\_args:Nxx . . . . . 27, 1540, 1560  
 \exp\_eval\_error\_msg:w . . 1501, 1504, 1517  
 \exp\_eval\_register:c . . . . .  
 157, 1497, 1501, 1511, 1590, 1660, 4086  
 \exp\_eval\_register:N 157, 1489, 1501,  
 1501, 1512, 1593, 1655, 4083, 4167  
 \exp\_last\_unbraced:NcV . . . . .  
 . . . . . 29, 1663, 1666, 3852, 4520  
 \exp\_last\_unbraced:Nf . . . 29, 1663, 1663  
 \exp\_last\_unbraced:NNNo . . 29, 1663, 1675  
 \exp\_last\_unbraced:NNo . . . . .  
 . . . . . 29, 1663, 1669, 4428, 5571  
 \exp\_last\_unbraced:NNV . . . . .  
 . . . . . 29, 1663, 1672, 4127, 5590  
 \exp\_last\_unbraced:NV . . . . .  
 . . . . . 29, 1663, 1664, 5592, 5599  
 \exp\_last\_unbraced:Nv . . . 29, 1663, 1665  
 \exp\_not:c . . . . .  
 . 28, 1595, 1595, 1615, 1639, 1985,  
 1989, 1994, 1998, 2001, 2003–2005,  
 2010, 2012–2014, 2019, 2021–2023,  
 2028, 2030–2032, 2037, 2039, 2041,

- 2043, 2045, 2047, 2049, 2051, 2053–  
2055, 2607, 3707, 3710, 5424, 5430
- \exp\_not:d ..... 28, [1582](#), 1583
- \exp\_not:f ..... 28, [1582](#), 1586
- \exp\_not:N [13](#), [28](#), [773](#), 774, 1315–1317,  
1354–1356, [1442](#), 1444, 1502, 1595,  
1990, 1993, 1997, 2001, 2010, 2019,  
2028, 2118, 2140, 2284, 2287, 2291,  
2295, 2299, 2303, 2307, 2311, 2315,  
2319, 2323, 2327, 2335, 2339, 2354,  
2575, 2590, 3707, 3709, 3921, 3931,  
4036, 4037, 4041, 4054, 4055, 5429,  
5431, 5554, 5562, 5572, 6068, 6071,  
6074, 6077, 6080, 6084, 6088, 6092
- \exp\_not:n ..... [13](#), [28](#), [773](#), 775,  
[1442](#), 1445, 1582, 1584, 1587, 1590,  
1593, 1986, 2119, 2141, 2537, 2538,  
2551, 2553, 2605, 2608, 3667, 3670,  
3673, 3676, 3679, 3682, 3685, 3691,  
3697, 3703, 3711, 3722, 3723, 3729,  
3736, 3743, 3751, 3758, 3793, 4290,  
4329, 4335, 4469, 4648, 4650, 4665,  
4669, 4674, 4784, 4806, 5428, 5587,  
5616, 6060, 6063, 6068, 6071, 6074,  
6077, 6080, 6084, 6088, 6092, 6096
- \exp\_not:o ..... 28, [1582](#), 1582, 3737,  
3757, 3771, 3772, 3785, 3807, 5636
- \exp\_not:V ..... 28,  
[1582](#), 1592, 3535, 3539, 3543, 3547,  
3552, 3556, 3560, 3564, 3572, 3577,  
3581, 3585, 3589, 3593, 3597, 3601,  
3676, 3682, 3688, 3694, 3700, 3730,  
3750, 3764, 3765, 3778, 3800, 4241
- \exp\_not:v ..... 28, [1582](#), 1589
- \exp\_stop\_f: ... [29](#), [1467](#), 1472, 1509, 3879
- \expandafter ..... [11](#), [12](#), [14](#), [15](#), [88](#)
- \expanded ..... [1473](#)
- \ExplSyntaxNamesOff ..... [5](#), [585](#), 589
- \ExplSyntaxNamesOn ..... [5](#), [585](#), 585, 5481
- \ExplSyntaxOff ..... [3](#), [5](#), [22](#),  
[559](#), 562, 662, 669, 680, 697, 705, 5499
- \ExplSyntaxOn ..... [3](#),  
[5](#), [559](#), 559, 653, 657, 678, 719, 5477
- \ExplSyntaxPopStack ... [115](#), 671, [675](#), 675
- \ExplSyntaxStack .....  
.... [115](#), 661, 668, 671, [675](#), 676, 683
- \ExplSyntaxStatus ..... [23](#), [24](#),  
[115](#), [559](#), 560, 564, 565, 575, 584, 661
- F**
- \F ..... [2362](#)
- \fam ..... [80](#)
- \fi ..... [12](#), [32](#), [119](#)
- \fi: ..... [7](#), [743](#), [755](#), 759, 854–  
857, 930, 936, 941, 946, 951, 956,  
961, 966, 971, 980, 998, 1020, 1021,  
1028, 1034, 1049, 1055, 1115, 1116,  
1129, 1130, 1133, 1234, 1237, 1243,  
1246, 1250, 1261, 1298, 1393, 1420,  
1505, 1508, 1518, 1606, 1630, 1642,  
1690, 1694, 1698, 1702, 1705, 1706,  
1851, 1880, 1928, 2083, 2088, 2097,  
2105, 2112, 2121, 2131, 2135, 2143,  
2153, 2288, 2292, 2296, 2300, 2304,  
2308, 2312, 2316, 2320, 2324, 2328,  
2332, 2336, 2340, 2347, 2351, 2355,  
2391, 2402, 2413, 2424, 2489, 2496,  
2499, 2500, 2516–2521, 2572, 2579,  
2594, 2827, 2836, 2868, 2881, 2999,  
3099, 3104, 3109, 3113, 3117, 3121,  
3125, 3129, 3133, 3137, 3143, 3153,  
3163, 3180, 3186, 3187, 3189, 3202,  
3206, 3401, 3461, 3517, 3529, 3626,  
3633, 3641, 3655, 3711, 3726, 3733,  
3740, 3747, 3754, 3761, 3768, 3775,  
3782, 3789, 3796, 3803, 3810, 3817,  
4033, 4038, 4045, 4056, 4250, 4255,  
4417, 4426, 4439, 4738, 4827, 5300,  
5303, 5314, 5817–5819, 5846–5852,  
5865–5867, 5881–5883, 5901, 5910,  
5927, 5936, 5974, 5984, 5996, 6018,  
6024, 6030, 6036, 6037, 6129, 6147
- \file\_add\_path:n .....  
..... [113](#), [6215](#), 6215, 6244, 6250
- \file\_add\_path\_aux:n [6215](#), 6226, 6240, 6242
- \file\_add\_path\_search:n ..... [6215](#)
- \file\_add\_path\_search:nN .... [6219](#), 6225
- \file\_if\_exist:n ..... [6181](#)
- \file\_if\_exist:nTF ..... [113](#), [6181](#)
- \file\_if\_exist\_aux:n [6181](#), 6192, 6206, 6214
- \file\_if\_exist\_p:n ..... [113](#), [6181](#)
- \file\_if\_exist\_path:n .. [6181](#), 6184, 6190
- \file\_input:n ..... [113](#), [6243](#), 6243
- \file\_input\_no\_check:n .....  
..... [113](#), [6246](#), [6255](#), 6255
- \file\_input\_no\_check\_no\_record:n ...  
..... [113](#), 6252, [6266](#), 6266
- \file\_input\_no\_record:n [113](#), [6243](#), 6249
- \file\_list: ..... [114](#), [6270](#), 6270
- \file\_list:N ..... [6270](#), 6271, 6274, 6276
- \file\_list\_aux:n ..... 6279, 6282
- \file\_list\_full: ..... [114](#), [6270](#), 6273
- \file\_not\_found:nTF ..... [1407](#)
- \fileauthor ..... [5](#), [593](#), 607, 622
- \filedate ..... [5](#), [593](#), 608,  
635, 639, 644, 647, 750, 1438, 1684,  
2067, 2170, 2687, 3031, 3080, 3235,

- 3436, 4063, 4216, 4389, 4618, 4760,  
 4844, 5279, 5413, 5526, 5695, 6171  
 \filedescription ..... 5, 593,  
 604, 621, 639, 647, 750, 1438, 1684,  
 2067, 2170, 2687, 3031, 3080, 3235,  
 3436, 4063, 4216, 4389, 4618, 4760,  
 4844, 5279, 5413, 5526, 5695, 6171  
 \filename ..... 5, 593,  
 605, 619, 639, 647, 750, 1438, 1684,  
 2067, 2170, 2687, 3031, 3080, 3235,  
 3436, 4063, 4216, 4389, 4618, 4760,  
 4844, 5279, 5413, 5526, 5695, 6171  
 \filenameext ..... 5, 593, 609, 636, 643  
 \filetimestamp ..... 5, 593, 610, 637, 645  
 \fileversion ..... 5, 593,  
 606, 620, 639, 647, 750, 1438, 1684,  
 2067, 2170, 2687, 3031, 3080, 3235,  
 3436, 4063, 4216, 4389, 4618, 4760,  
 4844, 5279, 5413, 5526, 5695, 6171  
 \finalhyphenemerits ..... 268  
 \firstchoice@true ..... 6110  
 \firstmark ..... 166  
 \firstmarks ..... 392  
 \firstoftwo ..... 3, 12  
 \floatingpenalty ..... 313  
 \font ..... 79  
 \fontchardp ..... 417  
 \fontcharht ..... 416  
 \fontcharic ..... 419  
 \fontcharwd ..... 418  
 \fontdimen ..... 346  
 \fontname ..... 170  
 \frozen@everydisplay ..... 688  
 \frozen@everymath ..... 687  
 \futurelet ..... 75
- G**
- \G ..... 2367  
 \g\_box\_allocation\_seq ..... 5285  
 \g\_calc\_A\_dim .....  
 297, 5703, 5706, 5749, 5752, 5755, 5758  
 \g\_calc\_A\_int ..... 297, 5703,  
 5703, 5731, 5734, 5737, 5740, 5743,  
 5746, 5761, 5764, 5916, 5928, 5937  
 \g\_calc\_A\_muskip ... 297, 5703, 5712,  
 5785, 5789, 5793, 5797, 5801, 5805  
 \g\_calc\_A\_register .....  
 297, 5699, 5700, 5716, 5726, 5826,  
 5829, 5832, 5833, 5888, 5902, 5911,  
 5916, 5955, 6016, 6022, 6028, 6034  
 \g\_calc\_A\_skip ..... 297, 5703, 5709,  
 5767, 5770, 5773, 5776, 5779, 5782  
 \g\_clist\_inline\_level\_int .....  
 ..... 4514, 4517, 4518,  
 4521, 4523, 4529, 4530, 4533, 4535  
 \g\_file\_record\_clist .....  
 ... 114, 6175, 6175, 6256, 6271, 6289  
 \g\_file\_record\_full\_clist .....  
 114, 6175, 6176, 6263, 6267, 6274, 6290  
 \g\_peek\_token ..... 47, 2523, 2524, 2528  
 \g\_prg\_inline\_level\_int .....  
 166, 1781, 1781, 1783, 1784, 1788, 1790  
 \g\_prop\_inline\_level\_num .....  
 93, 4743, 4743, 4745, 4746, 4749, 4750  
 \g\_testa\_tl ..... 72, 3618, 3620  
 \g\_testb\_tl ..... 72, 3618, 3621  
 \g\_tl\_inline\_level\_num .....  
 ..... 72, 3839, 3840, 3841, 3844,  
 3846, 3849, 3850, 3853, 3855, 3858  
 \g\_tmpa\_bool ..... 166, 1848, 1849  
 \g\_tmpa\_dim ..... 64, 3389, 3393  
 \g\_tmpa\_int ..... 52, 2883, 2886  
 \g\_tmpa\_num ..... 55, 3071, 3074  
 \g\_tmpa\_skip ..... 61, 3320, 3325  
 \g\_tmpa\_tl ..... 71, 3616, 3616  
 \g\_tmpa\_toks ..... 78, 4201, 4205  
 \g\_tmppb\_dim ..... 64, 3389, 3394  
 \g\_tmppb\_int ..... 52, 2883, 2887  
 \g\_tmppb\_num ..... 55, 3071, 3075  
 \g\_tmppb\_skip ..... 61, 3320, 3326  
 \g\_tmppb\_tl ..... 71, 3616, 3617  
 \g\_tmppb\_toks ..... 78, 4201, 4206  
 \g\_tmppc\_toks ..... 78, 4201, 4207  
 \g\_toks\_allocation\_seq ..... 4202  
 \g\_trace\_statistics\_status ..... 736  
 \g\_xref\_all\_curr\_deferred\_fields\_prop  
 ..... 286, 5417, 5418, 5465  
 \g\_xref\_all\_curr\_immediate\_fields\_prop  
 ..... 286, 5417, 5417, 5461  
 \gaddtolength ..... 111, 6104, 6107  
 \gdef ..... 66  
 \GetIdInfo ..... 5, 593, 593  
 \GetIdInfoAuxCVS:w ..... 593, 628, 633  
 \GetIdInfoAuxi:w ..... 593, 613, 617  
 \GetIdInfoAuxii:w ..... 593, 623, 626  
 \GetIdInfoAuxSVN:w ..... 593, 630, 641  
 \GetIdInfoMaybeMissing:w ..... 596, 598  
 \getname ..... 5489, 5517, 5520  
 \getpage ..... 5492, 5518, 5521  
 \getvaluepage ..... 5495, 5518, 5521  
 \global ..... 81  
 \globaldefs ..... 85  
 \glueexpr ..... 425  
 \glueshrink ..... 428  
 \glueshrinkorder ..... 430  
 \gluestretch ..... 427



- \gluestretchorder ..... 429
  - \gluetomu ..... 431
  - \group\_align\_safe\_begin: .....
    - .... 36, 1704, 1704, 1874, 2539, 2554
  - \group\_align\_safe\_end: 36, 1704, 1706,
    - 1900, 1901, 2537, 2538, 2553, 2564
  - \group\_begin: ..... 23, 786, 787,
    - 984, 1514, 2271, 2360, 2452, 3894,
    - 3904, 3920, 3930, 3938, 4795, 4878,
    - 4972, 4980, 5007, 5014, 5450, 5480,
    - 5538, 5551, 5585, 5596, 5719, 5822,
    - 5823, 5889, 5915, 5949, 5967, 5986,
    - 6001, 6006, 6012, 6042, 6047, 6141
  - \group\_end: 23, 786, 788, 988, 1519, 2285,
    - 2370, 2460, 3900, 3910, 3926, 3935,
    - 3943, 4795, 4881, 4978, 4995, 5011,
    - 5018, 5458, 5483, 5543, 5582, 5592,
    - 5599, 5724, 5727, 5832, 5833, 5836,
    - 5887, 5888, 5914, 5963, 5967, 5978,
    - 5990, 6008, 6038, 6041, 6046, 6143
  - \group\_execute\_after:N .... 23, 789,
    - 789, 5822, 5823, 5889, 5890, 5919, 6044
  - \gsetlength ..... 111, 6104, 6105
- H**
- \H ..... 2367
  - \halign ..... 92
  - \hangafter ..... 270
  - \hangindent ..... 271
  - \hbadness ..... 332
  - \hbox ..... 327
  - \hbox:n ..... 106, 5380, 5380
  - \hbox:w ..... 744
  - \hbox\_gset:cn ..... 106, 5381
  - \hbox\_gset:Nn ..... 106, 5381, 5383, 5384
  - \hbox\_gset\_inline\_begin:c .... 106, 5391
  - \hbox\_gset\_inline\_begin:N .....
    - .... 106, 5391, 5396, 5399
  - \hbox\_gset\_inline\_end: . 106, 5391, 5400
  - \hbox\_gset\_to\_wd:cnn ..... 106, 5385
  - \hbox\_gset\_to\_wd:Nnn 106, 5385, 5389, 5390
  - \hbox\_set:cn ..... 106, 5381
  - \hbox\_set:Nn .. 106, 5381, 5381–5383, 5944
  - \hbox\_set\_inline\_begin:c .... 106, 5391
  - \hbox\_set\_inline\_begin:N .....
    - .... 106, 5391, 5391, 5394, 5397
  - \hbox\_set\_inline\_end: .. 106, 5391, 5395
  - \hbox\_set\_to\_wd:cnn ..... 106, 5385
  - \hbox\_set\_to\_wd:Nnn .....
    - .... 106, 5385, 5385, 5388, 5389
  - \hbox\_to\_wd:nn ..... 106, 5401, 5401
  - \hbox\_to\_zero:n ..... 106, 5401, 5402
  - \hbox\_unpack:c ..... 106, 5403
  - \hbox\_unpack:N .... 106, 5403, 5403, 5404
  - \hbox\_unpack\_clear:c ..... 106, 5403
  - \hbox\_unpack\_clear:N 106, 5403, 5405, 5406
  - \heightof ..... 111, 6098, 6100
  - \hfil ..... 235
  - \hfill ..... 237
  - \hfilneg ..... 236
  - \hfuzz ..... 334
  - \hoffset ..... 309
  - \holdinginserts ..... 312
  - \hrule ..... 248
  - \hsize ..... 273
  - \hskip ..... 238
  - \hss ..... 239
  - \ht ..... 377
  - \hyphenation ..... 363
  - \hyphenchar ..... 347
  - \hyphenpenalty ..... 265
- I**
- \I ..... 2367
  - \if ..... 101
  - \if:w . 8, 755, 761, 928, 977, 1126, 1129,
    - 1130, 1418, 1604, 1630, 3099, 4036
  - \if\_bool:N 8, 755, 762, 1236, 1245, 1249, 1851
  - \if\_box\_empty:N ..... 103, 5296, 5298
  - \if\_case:w ..... 55, 2821, 2830,
    - 2858, 2871, 2989, 3035, 3036, 5897,
    - 5906, 5923, 5932, 5980, 5992, 6013
  - \if\_catcode:w . 8, 755, 765, 2287, 2291,
    - 2295, 2299, 2303, 2307, 2311, 2315,
    - 2319, 2323, 2327, 2335, 2575, 4054
  - \if\_charcode:w 8, 755, 764, 2339, 2590, 4041
  - \if\_cs\_exist:N 8, 771, 771, 1016, 1230, 1239
  - \if\_cs\_exist:w ..... 8, 771, 772, 1024
  - \if\_dim:w .....
    - 63, 3398, 3398, 3400, 6023, 6029, 6035
  - \if\_eof:w ..... 97, 4825, 4825
  - \if\_false: ..... 7, 755, 756, 869, 1705
  - \if\_hbox:N ..... 103, 5296, 5296
  - \if\_horizontal\_mode: ..... 738
  - \if\_intexpr\_case:w ..... 58, 3084, 3089
  - \if\_intexpr\_compare:w .....
    - . 58, 3084, 3087, 3108, 3112, 3116,
    - 3120, 3124, 3128, 3132, 3136, 3141,
    - 3148, 3158, 3171, 3175, 3176, 3182
  - \if\_intexpr\_odd:w 58, 3084, 3088, 3201, 3205
  - \if\_meaning:w ..... 7, 766, 766, 994,
    - 1013, 1030, 1112–1114, 1392, 1502,
    - 1503, 1640, 1880, 2080, 2086, 2092,
    - 2101, 2111, 2135, 2331, 2354, 2386,
    - 2397, 2408, 2419, 2568, 3104, 3456,
    - 3625, 3633, 3640, 3655, 3724, 3731,
    - 3738, 3745, 3752, 3759, 3766, 3773,
    - 3780, 3787, 3794, 3801, 3808, 3815,

- 4032, 4247, 4254, 4414, 4425, 4434,  
 4736, 5809, 5812, 5814, 5836–5842,  
 5855, 5858, 5861, 5871, 5874, 5877  
 \if\_mode\_horizontal: .. 8, 767, 768, 1693  
 \if\_mode\_inner: ..... 8, 767, 770, 1697  
 \if\_mode\_math: ..... 8, 767, 767, 1701  
 \if\_mode\_vertical: .... 8, 767, 769, 1689  
 \if\_num:w ..... 55, 1705, 1706,  
 2117, 2139, 3035, 3035, 5968, 6017  
 \if\_predicate:w ... 8, 755, 763, 1257,  
 1927, 2127, 2149, 2346, 2350, 2484,  
 2485, 2491, 2492, 2503, 2505, 2507,  
 2509, 2511, 2513, 3513, 3525, 4048  
 \if\_true: ..... 7, 755, 755, 868  
 \if\_vbox:N ..... 103, 5296, 5297  
 \ifcase ..... 102  
 \ifcat ..... 103  
 \ifcsname ..... 386  
 \ifdefined ..... 385  
 \ifdim ..... 106  
 \ifeof ..... 107  
 \iffalse ..... 112  
 \IfFileExists ..... 1407  
 \iffirstchoice@ ..... 6110, 6121, 6136  
 \iffontchar ..... 415  
 \ifhbox ..... 108  
 \ifhmode ..... 114  
 \ifinner ..... 117  
 \ifmmode ..... 115  
 \ifnum ..... 104  
 \ifodd ..... 23, 105  
 \iftrue ..... 113  
 \ifvbox ..... 109  
 \ifvmode ..... 116  
 \ifvoid ..... 110  
 \ifx ..... 11, 111  
 \ignorespaces ..... 159  
 \immediate ..... 121  
 \indent ..... 255  
 \input ..... 129  
 \input@path ..... 6201, 6202, 6235, 6236  
 \InputIfFileExists ..... 5482  
 \inputlineno ..... 131  
 \insert ..... 311  
 \insertpenalties ..... 314  
 \int\_add:cn ..... 49, 2753  
 \int\_add:Nn ..... 49, 2741,  
 2742, 2753, 2753, 2770, 2777, 5737  
 \int\_advance:w .....  
 200, 2691, 2693, 2719, 2724, 2754, 2760  
 \int\_Alph\_default\_conversion\_rule:n  
 ..... 51, 2820, 2829, 2844  
 \int\_alph\_default\_conversion\_rule:n  
 ..... 51, 2820, 2820, 2840  
 \int\_const:Nn .....  
 51, 2893, 2893, 2906–2921, 2923–2934  
 \int\_convert\_from\_base\_ten:nn .....  
 ..... 52, 2961, 2961  
 \int\_convert\_from\_base\_ten\_aux:ffn 2977  
 \int\_convert\_from\_base\_ten\_aux:fon 2961  
 \int\_convert\_from\_base\_ten\_aux:nfn .  
 ..... 2964, 2968  
 \int\_convert\_from\_base\_ten\_aux:nnn .  
 ..... 2961, 2973, 2986, 2987  
 \int\_convert\_from\_base\_ten\_aux:non 2961  
 \int\_convert\_letter\_to\_number:N ....  
 ..... 3013, 3017  
 \int\_convert\_number\_to\_letter:n ....  
 ..... 200, 2975, 2979, 2988, 2988  
 \int\_convert\_number\_with\_rule:nnN ..  
 ..... 51, 2810,  
 2810, 2813, 2839, 2843, 2849, 2853  
 \int\_convert\_to\_base\_ten:nn .....  
 ..... 53, 3000, 3000  
 \int\_convert\_to\_base\_ten\_aux:nn ....  
 ..... 3003, 3006  
 \int\_convert\_to\_base\_ten\_auxi:nnN ..  
 ..... 3007, 3009, 3012  
 \int\_decr:c ..... 49, 2719  
 \int\_decr:N 49, 2719, 2724, 2740, 2742, 2746  
 \int\_gadd:cn ..... 49, 2753  
 \int\_gadd:Nn .....  
 49, 2743, 2744, 2753, 2765, 2778, 5740  
 \int\_gdecr:c ..... 49, 2719  
 \int\_gdecr:N ..... 49, 1790,  
 2719, 2735, 2744, 2748, 4523, 4535  
 \int\_get\_digits:n . 200, 2935, 2941, 3003  
 \int\_get\_sign:n ... 200, 2935, 2938, 3002  
 \int\_get\_sign\_and\_digits:n .....  
 ..... 200, 2935, 2935  
 \int\_get\_sign\_and\_digits\_aux:nnnn ..  
 .. 2935, 2936, 2939, 2942, 2944, 2960  
 \int\_get\_sign\_and\_digits\_aux:onnn ..  
 ..... 2935, 2948, 2949, 2953  
 \int\_gincr:c ..... 49, 2719, 6140  
 \int\_gincr:N ..... 49, 1783,  
 2719, 2729, 2743, 2747, 4517, 4529  
 \int\_gset:cn ..... 49, 2706  
 \int\_gset:Nn ... 49, 2706, 2711, 2718, 5734  
 \int\_gsub:cn ..... 49, 2753  
 \int\_gsub:Nn ... 49, 2753, 2771, 2780, 5746  
 \int\_gzero:c ..... 49, 2749  
 \int\_gzero:N ..... 49, 2749, 2751, 2752  
 \int\_incr:c ..... 49, 2719  
 \int\_incr:N 49, 2719, 2719, 2734, 2741, 2745  
 \int\_new:c ..... 48, 2694



- \int\_new:N ..... 48,  
1781, 2694, 2696, 2700, 2705, 2883–  
2887, 2896, 2901, 4514, 5702–5705
- \int\_new\_l:N ..... 2697
- \int\_pre\_eval\_one\_arg:Nn 200, 2888, 2888
- \int\_pre\_eval\_two\_args:Nnn .....  
..... 200, 2888, 2890
- \int\_roman\_lcuc\_mapping:Nnn .....  
..... 51, 2786, 2786, 2790–2797
- \int\_set:cn ..... 49, 2706
- \int\_set:Nn ..... 49, 2706, 2706, 2716,  
2717, 2896, 2901, 4848, 5718, 5731
- \int\_show:c ..... 50, 2783, 2784
- \int\_show:N ..... 50, 2783, 2783, 2784
- \int\_sub:cn ..... 49, 2753
- \int\_sub:Nn 49, 2753, 2759, 2776, 2779, 5743
- \int\_symbol\_math\_conversion\_rule:n .  
..... 51, 2850, 2857, 2857
- \int\_symbol\_text\_conversion\_rule:n .  
..... 51, 2854, 2857, 2870
- \int\_to\_Alph:n ..... 50, 2838, 2842
- \int\_to\_alph:n ..... 50, 2838, 2838
- \int\_to\_arabic:n ..... 50, 2785, 2785
- \int\_to\_number:w ..... 51, 2691, 2692
- \int\_to\_Roman:n ..... 50, 2798, 2802
- \int\_to\_roman:n ..... 50, 2798, 2798
- \int\_to\_roman:w .....  
..... 50, 1893, 2691, 2691, 2800,  
2804, 4083, 4086, 4090, 4167, 4182
- \int\_to\_roman\_lcuc:NN .....  
..... 51, 2798, 2799, 2803, 2806, 2808
- \int\_to\_symbol:n ..... 50, 2846, 2846
- \int\_use:c ..... 49, 2781, 2782
- \int\_use:N ..... 49, 1784, 1788, 2178,  
2228, 2240, 2250, 2260, 2781, 2781,  
2782, 4518, 4521, 4530, 4533, 5976
- \int\_zero:c ..... 49, 2749
- \int\_zero:N 49, 2749, 2749, 2750, 5918, 6043
- \interactionmode ..... 413
- \interlinepenalties ..... 434
- \interlinepenalty ..... 293
- \intexpr\_abs:n ..... 57, 3139, 3139
- \intexpr\_compare:n ..... 3093
- \intexpr\_compare:nF ..... 3212, 3218
- \intexpr\_compare:nNn ..... 3135
- \intexpr\_compare:nNnF .....  
..... 1710, 1712, 1764, 1773,  
1793, 1801, 1815, 1823, 3224, 3230
- \intexpr\_compare:nNnT .. 1708, 3221, 3227
- \intexpr\_compare:nNnTF .... 56, 1758,  
1785, 1809, 1923, 1948, 2811, 2894,  
2895, 2962, 2974, 3018, 3021, 3135
- \intexpr\_compare:nT ..... 3209, 3215
- \intexpr\_compare:nTF ..... 56, 3093
- \intexpr\_compare\_auxi:w .... 3094, 3097
- \intexpr\_compare\_auxii:w .... 3098, 3101
- \intexpr\_compare\_p:n ..... 56, 3093
- \intexpr\_compare\_p:nNn 56, 3135, 3339, 3340
- \intexpr\_div\_round:nn ... 57, 3166, 3193
- \intexpr\_div\_truncate:nn .....  
..... 57, 2814, 2982, 3166, 3169, 3197
- \intexpr\_div\_truncate\_raw:nn .... 3166
- \intexpr\_do\_until:nn 58, 3208, 3217, 3218
- \intexpr\_do\_until:nNn 58, 3220, 3229, 3230
- \intexpr\_do\_while:nn .... 58, 3208, 3214
- \intexpr\_do\_while:nNn .....  
..... 58, 3215, 3220, 3226, 3227
- \intexpr\_eval:n ..... 56, 1720,  
1761, 1768, 1777, 1789, 1797, 1805,  
1812, 1819, 1827, 1943, 2785, 2800,  
2804, 2816, 2818, 2889, 2891, 2898,  
2965, 2969, 3001, 3013, 3020, 3052,  
3084, 3090, 3167, 3193, 3882, 3888
- \intexpr\_eval:w .....  
..... 57, 2176, 2180, 2186, 2222,  
2226, 2230, 2234, 2238, 2242, 2245,  
2248, 2252, 2255, 2258, 2262, 2266,  
2706, 2754, 2760, 2989, 3084, 3085,  
3091, 3095, 3108, 3112, 3116, 3120,  
3124, 3128, 3132, 3136, 3141, 3144,  
3147, 3149, 3157, 3159, 3170, 3171,  
3175, 3176, 3182, 3196, 3201, 3205
- \intexpr\_eval\_end: .....  
..... 57, 2176, 2180, 2186, 2222,  
2226, 2230, 2234, 2238, 2242, 2245,  
2248, 2252, 2255, 2258, 2262, 2266,  
2706, 2754, 2760, 2989, 3084, 3086,  
3091, 3108, 3112, 3116, 3120, 3124,  
3128, 3132, 3136, 3144, 3149, 3154,  
3159, 3164, 3191, 3198, 3201, 3205
- \intexpr\_if\_even:n ..... 3204
- \intexpr\_if\_even:nTF ..... 57, 3200
- \intexpr\_if\_even\_p:n .... 57, 1885, 3200
- \intexpr\_if\_odd:n ..... 3200
- \intexpr\_if\_odd:nTF ..... 57, 3200
- \intexpr\_if\_odd\_p:n ..... 57, 3200
- \intexpr\_max:nn ..... 57, 3139, 3146
- \intexpr\_min:nn ..... 57, 3139, 3156
- \intexpr\_mod:nn 57, 2816, 2979, 3166, 3194
- \intexpr\_until\_do:nn 58, 3208, 3211, 3212
- \intexpr\_until\_do:nNn 58, 3220, 3223, 3224
- \intexpr\_value:w 57, 3084, 3084, 3091,  
3094, 3140, 3147, 3157, 3170, 3195
- \intexpr\_while\_do:nn 58, 3208, 3208, 3209
- \intexpr\_while\_do:nNn 58, 3220, 3220, 3221
- \ior\_close:N .....  
..... 96, 4820, 4820, 4822, 6186, 6207, 6223
- \ior\_gto:NN ..... 97, 4829, 4833

- \ior\_if\_eof:N ..... 4826
  - \ior\_if\_eof:Nf ..... 6194, 6228
  - \ior\_if\_eof:Ntf ..... 96, 4826, 6183, 6218
  - \ior\_if\_eof\_p:N ..... 96, 4826
  - \ior\_new:N ..... 96, 4815, 4817, 4819, 6178
  - \ior\_open:Nn ..... 96, 4820, 4821, 6182, 6193, 6217, 6227
  - \ior\_to:NN ..... 97, 4829, 4829, 4836
  - \iow\_char:N ..... 96, 4290, 4469, 4648, 4650, 4811, 4811
  - \iow\_close:N ..... 94, 4773, 4777, 4777
  - \iow\_log:n ..... 95, 4786, 4787
  - \iow\_log:x 13, 95, 1072, 1072, 1096, 1620, 4786, 4786, 5010, 6278, 6280, 6283
  - \iow\_new:c ..... 94, 4764
  - \iow\_new:N 94, 4764, 4766, 4769, 4771, 5469
  - \iow\_newline: ..... 96, 4809, 4809
  - \iow\_now:Nn ..... 94, 4783, 4783, 4787, 4789, 4791, 4801
  - \iow\_now:Nx ..... 94, 4782, 4782, 4784, 4786, 4788, 4798
  - \iow\_now\_buffer\_safe:Nn . 95, 4794, 4800
  - \iow\_now\_buffer\_safe:Nx . 95, 4794, 4797
  - \iow\_now\_buffer\_safe\_aux:w ..... 4794, 4798, 4801
  - \iow\_now\_buffer\_safe\_expanded\_aux:w ..... 4794
  - \iow\_now\_when\_avail:cn ..... 95, 4790
  - \iow\_now\_when\_avail:Nn 95, 4790, 4790, 4793
  - \iow\_open:cn ..... 94, 4772
  - \iow\_open:Nn ... 94, 4772, 4772, 4776, 5478
  - \iow\_shipout:Nn ..... 95, 4805, 4805, 4808
  - \iow\_shipout:Nx ..... 95, 4805
  - \iow\_shipout\_x:Nn ..... 13, 95, 776, 776, 1073, 1076, 4782, 4803, 4803, 4804, 4806, 5462
  - \iow\_shipout\_x:Nx ..... 95, 4803
  - \iow\_space: ..... 96, 2128, 2150, 4290, 4469, 4648–4650, 4810, 4810
  - \iow\_term:n ..... 95, 4786, 4789
  - \iow\_term:x ..... 13, 95, 1072, 1075, 1079, 1095, 4282, 4461, 4640, 4786, 4788, 5017
- J**
- \jobname ..... 368, 5478, 5482
- K**
- \K ..... 2367
  - \kern ..... 246
  - \KV\_add\_element\_aux:w .. 5577, 5583, 5583
  - \KV\_add\_value\_element:w ..... 290, 5551, 5560, 5658
  - \KV\_key\_no\_value\_elt:n ..... 110, 5620, 5665, 5675, 5675
  - \KV\_key\_value\_elt:nn ..... 110, 5638, 5670, 5675, 5676
  - \KV\_parse\_elt:w ..... 5590, 5598, 5602, 5602, 5604, 5609
  - \KV\_parse\_no\_sanitization\_aux:n ..... 5595, 5595, 5679, 5687
  - \KV\_parse\_no\_space\_removal\_no\_sanitization:n ..... 109, 5677, 5677
  - \KV\_parse\_sanitization\_aux:n 5584, 5584, 5683
  - \KV\_parse\_space\_removal\_no\_sanitization:n ..... 110, 5681, 5685
  - \KV\_parse\_space\_removal\_sanitization:n ..... 110, 5681, 5681
  - \KV\_remove\_surrounding\_spaces:nw ... 290, 5551, 5553
  - \KV\_remove\_surrounding\_spaces\_auxi:w ... 290, 5551, 5554, 5556, 5562, 5571
  - \KV\_remove\_surrounding\_spaces\_auxii:w ..... 5551, 5557, 5559
  - \KV\_sanitization\_outerlevel\_active\_commas:N ..... 290, 5538, 5547, 5589
  - \KV\_sanitization\_outerlevel\_active\_equals:N ..... 290, 5538, 5544, 5588
  - \KV\_set\_key\_element:w 290, 5551, 5568, 5615
  - \KV\_split\_key\_value\_current:w ..... 291, 5608, 5613, 5613, 5678, 5682, 5686
  - \KV\_split\_key\_value\_no\_space\_removal:w ..... 291, 5661, 5661, 5678
  - \KV\_split\_key\_value\_space\_removal:w ..... 291, 5614, 5614, 5682, 5686
  - \KV\_split\_key\_value\_space\_removal\_aux:w ..... 5614, 5626, 5635
  - \KV\_split\_key\_value\_space\_removal\_detect\_error:wTF ..... 291, 5614, 5625, 5631
  - \KV\_val\_preserve\_braces:NnN . 5634, 5654
- L**
- \L ..... 2367
  - \l\_calc\_B\_dim .. 297, 5703, 5707, 5749, 5752, 5755, 5758, 6021, 6023, 6024
  - \l\_calc\_B\_int ..... 297, 5703, 5704, 5731, 5734, 5737, 5740, 5743, 5746, 5761, 5764, 5917, 5969, 5972, 5976, 6015, 6017, 6018
  - \l\_calc\_B\_muskip ..... 297, 5703, 5713, 5785, 5789, 5793, 5797, 5801, 5805, 6033, 6035, 6036
  - \l\_calc\_B\_register ..... 297, 5699, 5701, 5717, 5726, 5728, 5826, 5832, 5833, 5888, 5896, 5902, 5905, 5911, 5917, 5922, 5928, 5931, 5937, 5979, 5985, 5991, 5997

- \l\_calc\_B\_skip .....  
     297, 5703, 5710, 5767, 5770, 5773,  
     5776, 5779, 5782, 6027, 6029, 6030
- \l\_calc\_C\_dim .....  
     ... 297, 5703, 5708, 6020, 6023, 6024
- \l\_calc\_C\_int ..... 297, 5703, 5705,  
     5970, 5973, 5976, 6014, 6017, 6018
- \l\_calc\_C\_muskip .....  
     ... 297, 5703, 5714, 6032, 6035, 6036
- \l\_calc\_C\_skip .....  
     ... 297, 5703, 5711, 6026, 6029, 6030
- \l\_calc\_current\_type\_int .. 297, 5699,  
     5702, 5718, 5897, 5906, 5918, 5923,  
     5932, 5968, 5980, 5992, 6013, 6043
- \l\_calc\_expression\_tl .....  
     ..... 297, 5699, 5699, 5722, 5725
- \l\_clist\_remove\_clist .....  
     ..... 4354, 4564, 4564, 4566,  
     4568, 4571, 4572, 4588, 4591, 4595
- \l\_exp\_tl ..... 157, 1447, 1447,  
     1476, 1477, 3534, 3535, 3542, 3543,  
     3551, 3552, 3559, 3560, 3571, 3572,  
     3576, 3577, 3584, 3585, 3592, 3593
- \l\_file\_search\_path\_clist .....  
     114, 6177, 6177, 6202, 6206, 6236, 6240
- \l\_file\_test\_read\_stream .. 114, 6178,  
     6178, 6182, 6183, 6186, 6193, 6194,  
     6207, 6217, 6218, 6223, 6227, 6228
- \l\_file\_tmp\_bool .....  
     ... 114, 6179, 6179, 6191, 6195, 6208
- \l\_file\_tmp\_tl .....  
     114, 6180, 6180, 6244–6246, 6250–6252
- \l\_KV\_currkey\_toks ..... 290, 5533,  
     5534, 5569, 5576, 5578, 5621, 5639
- \l\_KV\_currval\_toks .....  
     ..... 290, 5533, 5535, 5561, 5565
- \l\_KV\_parse\_toks ... 290, 5533, 5533,  
     5564, 5586, 5593, 5597, 5600, 5619,  
     5637, 5642, 5648, 5653, 5665, 5670
- \l\_KV\_remove\_one\_level\_of\_braces\_bool  
     ... 110, 5536, 5536, 5537, 5574, 5646
- \l\_KV\_tmpa\_tl ..... 290, 5530,  
     5530, 5587–5589, 5591, 5616, 5617,  
     5636, 5641, 5644, 5662, 5663, 5668
- \l\_KV\_tmpb\_tl ..... 290, 5530, 5531
- \l\_last\_box ..... 106, 5324, 5324, 5325
- \l\_msg\_class\_tl .....  
     ... 103, 4890, 4890, 5101, 5102, 5105
- \l\_msg\_current\_class\_tl .....  
     ..... 103, 4890, 4891, 5081, 5102
- \l\_msg\_current\_module\_tl ..... 5082
- \l\_msg\_names\_clist .....  
     ..... 103, 4892, 4892, 4938, 4939
- \l\_msg\_redirect\_classes\_clist .....  
     ... 103, 4895, 4895, 5057, 5061, 5064
- \l\_msg\_redirect\_classes\_prop .....  
     ..... 103, 4893, 4893
- \l\_msg\_redirect\_names\_prop .....  
     ..... 103, 4893, 4894, 5083, 5174
- \l\_msg\_tmp\_tl ..... 4985, 5001, 5003
- \l\_peek\_false\_tl ... 184, 2530, 2531,  
     2538, 2553, 2571, 2578, 2586, 2593
- \l\_peek\_search\_tl .....  
     ... 184, 2533, 2533, 2536, 2550, 2587
- \l\_peek\_search\_token .....  
     47, 2523, 2525, 2535, 2549, 2568, 2575
- \l\_peek\_token ..... 47, 2523, 2523,  
     2526, 2568, 2575, 2583, 2584, 2675
- \l\_peek\_true\_aux\_tl .....  
     ..... 184, 2551, 2563, 2563, 2565
- \l\_peek\_true\_tl ..... 184, 2530,  
     2530, 2537, 2552, 2569, 2576, 2591
- \l\_seq\_remove\_seq .....  
     ..... 4354, 4356, 4358, 4361, 4362
- \l\_testa\_tl ..... 72, 3618, 3618,  
     3722, 3724, 3729, 3731, 3736, 3738,  
     3743, 3745, 3750, 3752, 3757, 3759,  
     3764, 3766, 3771, 3773, 3778, 3780,  
     3785, 3787, 3792, 3794, 3799, 3801,  
     3806, 3808, 3813, 3815, 4248, 4415
- \l\_testb\_tl ..... 72,  
     3618, 3619, 3723, 3724, 3730, 3731,  
     3737, 3738, 3744, 3745, 3751, 3752,  
     3758, 3759, 3765, 3766, 3772, 3773,  
     3779, 3780, 3786, 3787, 3793, 3794,  
     3800, 3801, 3807, 3808, 3814, 3815
- \l\_tl\_replace\_toks ..... 71,  
     78, 3967, 3967, 3972, 3974, 3977,  
     3988, 3992, 3995, 4001, 4209, 4209
- \l\_tmpa\_bool ..... 166, 1848, 1848
- \l\_tmpa\_box 105, 5349, 5350, 5352, 5944, 5947
- \l\_tmpa\_dim ..... 64, 3389, 3389
- \l\_tmpa\_int ..... 52, 2883, 2883
- \l\_tmpa\_num ..... 55, 3071, 3071
- \l\_tmpa\_skip ..... 61, 3320, 3320–3322
- \l\_tmpa\_tl .. 4, 71, 599, 601, 602, 613,  
     615, 3622, 3622, 3706, 3714, 3924, 3927
- \l\_tmpa\_toks ..... 78, 3911,  
     3917, 3944, 4201, 4201, 4284, 4286,  
     4287, 4289, 4293, 4463, 4465, 4466,  
     4468, 4472, 4552, 4555, 4556, 4642,  
     4644, 4645, 4647, 4653, 5427, 5433
- \l\_tmpb\_box ..... 105, 5349, 5353
- \l\_tmpb\_dim ..... 64, 3389, 3390
- \l\_tmpb\_int ..... 52, 2883, 2884
- \l\_tmpb\_num ..... 55, 3071, 3072
- \l\_tmpb\_skip ..... 61, 3320, 3323

<code>\l_tmpb_tl</code> . . . . .	71, 600, 601, 3622, 3623	<code>\mathparagraph</code> . . . . .	2863
<code>\l_tmpb_toks</code> 78, 4201, 4203, 4553, 4556, 4557		<code>\mathpunct</code> . . . . .	214
<code>\l_tmpc_dim</code> . . . . .	64, 3389, 3391	<code>\mathrel</code> . . . . .	215
<code>\l_tmpc_int</code> . . . . .	52, 2883, 2885	<code>\mathsection</code> . . . . .	2862
<code>\l_tmpc_num</code> . . . . .	55, 3071, 3073	<code>\mathsurround</code> . . . . .	226
<code>\l_tmpc_skip</code> . . . . .	61, 3320, 3324	<code>\maxdeadcycles</code> . . . . .	296
<code>\l_tmpc_toks</code> . . . . .	78, 4201, 4204	<code>\maxdepth</code> . . . . .	297
<code>\l_tmpd_dim</code> . . . . .	64, 3389, 3392	<code>\maxdimen</code> . . . . .	3335
<code>\l_xref_curr_name_tl</code> . . . . .	5488	<code>\maxof</code> . . . . .	111, 6059, 6065
<code>\language</code> . . . . .	163	<code>\meaning</code> . . . . .	356
<code>\lastbox</code> . . . . .	320	<code>\medmuskip</code> . . . . .	227
<code>\lastkern</code> . . . . .	253	<code>\message</code> . . . . .	133
<code>\lastlinefit</code> . . . . .	433	<code>\MessageBreak</code> . . . . .	728
<code>\lastnodetype</code> . . . . .	414	<code>\middle</code> . . . . .	438
<code>\lastpenalty</code> . . . . .	359	<code>\minof</code> . . . . .	111, 6059, 6066
<code>\lastskip</code> . . . . .	254	<code>\mkern</code> . . . . .	180
<code>\lccode</code> . . . . .	382	<code>\mode_if_horizontal:</code> . . . . .	1692
<code>\leaders</code> . . . . .	250	<code>\mode_if_horizontal:TF</code> . . . . .	35, 1692
<code>\left</code> . . . . .	218	<code>\mode_if_horizontal_p:</code> . . . . .	35, 1692
<code>\lefthyphenmin</code> . . . . .	274	<code>\mode_if_inner:</code> . . . . .	1696
<code>\leftmarginkern</code> . . . . .	496	<code>\mode_if_inner:TF</code> . . . . .	35, 1696
<code>\leftskip</code> . . . . .	276	<code>\mode_if_inner_p:</code> . . . . .	35, 1696
<code>\leqno</code> . . . . .	193	<code>\mode_if_math:</code> . . . . .	1700
<code>\let</code> . . . . .	40, 63	<code>\mode_if_math:TF</code> . . . . .	35, 1700, 2847
<code>\limits</code> . . . . .	210	<code>\mode_if_math_p:</code> . . . . .	35, 1700
<code>\linepenalty</code> . . . . .	266	<code>\mode_if_vertical:</code> . . . . .	1688
<code>\lineskip</code> . . . . .	260	<code>\mode_if_vertical:TF</code> . . . . .	35, 1688
<code>\lineskiplimit</code> . . . . .	261	<code>\mode_if_vertical_p:</code> . . . . .	35, 1688
<code>\long</code> . . . . .	43, 48, 51, 56, 82	<code>\month</code> . . . . .	366
<code>\looseness</code> . . . . .	278	<code>\moveleft</code> . . . . .	316
<code>\lower</code> . . . . .	315	<code>\moveright</code> . . . . .	317
<code>\lowercase</code> . . . . .	354	<code>\msg_class_new:nn</code> 98, 5038, 5038, 5108,	
<code>\lpcode</code> . . . . .	463	5119, 5140, 5147, 5154, 5160, 5166	
<code>\luatex_directlua:D</code> . . . . .	553, 1239	<code>\msg_class_set:nn</code> . . 98, 5038, 5041, 5043	
<code>\luatex_if_engine:</code> . . . . .	1248	<code>\msg_direct_interrupt:n</code> . . . . .	4972
<code>\luatex_if_engine:TF</code> . . . . .	23, 1230	<code>\msg_direct_interrupt:xxxxn</code> 101, 4972,	
		4979, 5109, 5120, 5203, 5215, 5269	
		<code>\msg_direct_interrupt_aux:n</code> . 4983, 4999	
		<code>\msg_direct_log:xx</code> . . . . .	
		101, 5006, 5006, 5148, 5155, 5161	
		<code>\msg_direct_term:xx</code> 101, 5006, 5013, 5141	
		<code>\msg_error:nn</code> . . . . .	99, 5119
		<code>\msg_error:nnx</code> . . . . .	99, 5119
		<code>\msg_error:nnxx</code> . . . . .	99, 5119
		<code>\msg_fatal:nn</code> . . . . .	98, 5108
		<code>\msg_fatal:nnx</code> . . . . .	98, 5108
		<code>\msg_fatal:nnxx</code> . . . . .	98, 5108
		<code>\msg_four_spaces:</code> . . . . .	100, 4907, 4909
		<code>\msg_generic_new:nn</code> 101, 4910, 4918, 5027	
		<code>\msg_generic_new:nnn</code> 101, 4910, 4914, 5024	
		<code>\msg_generic_new:nnnn</code> 101, 4910, 4910, 5021	
		<code>\msg_generic_set:nn</code> . . . . .	
		101, 4920, 4922, 4932, 5036	
M			
<code>\M</code> . . . . .	2367, 2454		
<code>\m@ne</code> . . . . .	1064		
<code>\mag</code> . . . . .	162		
<code>\mark</code> . . . . .	164		
<code>\marks</code> . . . . .	390		
<code>\mathaccent</code> . . . . .	175		
<code>\mathbin</code> . . . . .	205		
<code>\mathchar</code> . . . . .	176		
<code>\mathchardef</code> . . . . .	73		
<code>\mathchoice</code> . . . . .	173		
<code>\mathclose</code> . . . . .	206		
<code>\mathcode</code> . . . . .	384		
<code>\mathinner</code> . . . . .	207		
<code>\mathop</code> . . . . .	208		
<code>\mathopen</code> . . . . .	212		
<code>\mathord</code> . . . . .	213		

- \msg\_generic\_set:nnn ..... [101](#), [4916](#), [4922](#), [4927](#), [5033](#)
  - \msg\_generic\_set:nnnn ..... [101](#), [4912](#), [4922](#), [4922](#), [5030](#)
  - \msg\_generic\_set\_clist:n ..... [4922](#), [4923](#), [4928](#), [4933](#), [4937](#)
  - \msg\_generic\_set\_code:n ..... [4922](#)
  - \msg\_generic\_set\_code:nn .... [4963](#), [4965](#)
  - \msg\_generic\_set\_code:nnnn .. [4925](#), [4955](#)
  - \msg\_generic\_set\_more\_text:n .... [4922](#)
  - \msg\_generic\_set\_more\_text:nnn .... [4930](#), [4946](#)
  - \msg\_generic\_set\_text:n ..... [4922](#)
  - \msg\_generic\_set\_text:nn .... [4935](#), [4942](#)
  - \msg\_info:nn ..... [99](#), [5147](#)
  - \msg\_info:nnx ..... [99](#), [5147](#)
  - \msg\_info:nnxx ..... [99](#), [5147](#), [5241](#)
  - \msg\_kernel\_bug:x ..... [13](#),  
[102](#), [1078](#), [1078](#), [1088](#), [1103](#), [1305](#),  
[3457](#), [4249](#), [4416](#), [4435](#), [5268](#), [5268](#)
  - \msg\_kernel\_classes\_new:n .....  
.. [5194](#), [5194](#), [5213](#), [5235](#), [5239](#), [5243](#)
  - \msg\_kernel\_error:n .... [102](#), [5062](#), [5214](#)
  - \msg\_kernel\_error:nx ... [102](#), [5068](#), [5214](#)
  - \msg\_kernel\_error:nnx [102](#), [5075](#), [5214](#), [5214](#)
  - \msg\_kernel\_fatal:n ..... [102](#), [5202](#)
  - \msg\_kernel\_fatal:nx ..... [102](#), [5202](#)
  - \msg\_kernel\_fatal:nnx .. [102](#), [5202](#), [5202](#)
  - \msg\_kernel\_info:n ..... [102](#), [5236](#)
  - \msg\_kernel\_info:nx ..... [102](#), [5236](#)
  - \msg\_kernel\_info:nnx ... [102](#), [5236](#), [5240](#)
  - \msg\_kernel\_new:nn .... [101](#), [5176](#), [5182](#)
  - \msg\_kernel\_new:nnn .....  
[101](#), [5176](#), [5179](#), [5244](#), [5248](#), [5255](#), [5261](#)
  - \msg\_kernel\_new:nnnn ... [101](#), [5176](#), [5176](#)
  - \msg\_kernel\_set:nn .... [101](#), [5176](#), [5191](#)
  - \msg\_kernel\_set:nnn .... [101](#), [5176](#), [5188](#)
  - \msg\_kernel\_set:nnnn ... [101](#), [5176](#), [5185](#)
  - \msg\_kernel\_warning:n ..... [102](#), [5236](#)
  - \msg\_kernel\_warning:nx ..... [102](#), [5236](#)
  - \msg\_kernel\_warning:nnx [102](#), [5236](#), [5236](#)
  - \msg\_line\_context: .... [100](#), [4896](#), [4899](#)
  - \msg\_line\_number: . [100](#), [4896](#), [4896](#), [4903](#)
  - \msg\_log:nn ..... [99](#), [5154](#)
  - \msg\_log:nnx ..... [99](#), [5154](#)
  - \msg\_log:nnxx ..... [99](#), [5154](#)
  - \msg\_new:nnn ..... [98](#), [5020](#), [5026](#), [5183](#)
  - \msg\_new:nnnn ..... [98](#), [5020](#), [5023](#), [5180](#)
  - \msg\_new:nnnnn ..... [98](#), [5020](#), [5020](#), [5177](#)
  - \msg\_newline: ..... [100](#), [4864](#),  
[4875](#), [4905](#), [4905](#), [4981](#), [4984](#), [4988](#),  
[5008](#), [5010](#), [5015](#), [5017](#), [5121](#), [5216](#)
  - \msg\_none:nn ..... [99](#), [5166](#)
  - \msg\_none:nnx ..... [99](#), [5166](#)
  - \msg\_none:nnxx ..... [99](#), [5166](#)
  - \msg\_redirect\_class:nn . [100](#), [5167](#), [5167](#)
  - \msg\_redirect\_module:nnn [100](#), [5170](#), [5170](#)
  - \msg\_redirect\_name:nnn . [100](#), [5173](#), [5173](#)
  - \msg\_set:nnn ..... [98](#), [5020](#), [5035](#), [5192](#)
  - \msg\_set:nnnn ..... [98](#), [5020](#), [5032](#), [5189](#)
  - \msg\_set:nnnnn ..... [98](#), [5020](#), [5029](#), [5186](#)
  - \msg\_space: ..... [100](#),  
[4900](#), [4902](#), [4907](#), [4907](#), [4908](#), [4982](#),  
[5009](#), [5016](#), [5112](#), [5115](#), [5123](#), [5126](#),  
[5142](#), [5149](#), [5206](#), [5209](#), [5218](#), [5221](#)
  - \msg\_trace:nn ..... [99](#), [5160](#)
  - \msg\_trace:nnx ..... [99](#), [5160](#)
  - \msg\_trace:nnxx ..... [99](#), [5160](#)
  - \msg\_two\_newlines: .....  
[100](#), [4873](#), [4905](#), [4906](#), [4989](#), [5110](#), [5204](#)
  - \msg\_two\_spaces: .....  
... [100](#), [4907](#), [4908](#), [4909](#), [5145](#), [5152](#)
  - \msg\_use:nnnnxx ..... [5046](#), [5055](#), [5055](#)
  - \msg\_use\_aux:nn ..... [5086](#), [5089](#), [5089](#)
  - \msg\_use\_aux:nnn ..... [5073](#), [5080](#), [5080](#)
  - \msg\_use\_code: [5056](#), [5078](#), [5078](#), [5096](#), [5103](#)
  - \msg\_use\_loop: ..... [5078](#)
  - \msg\_use\_loop:n ..... [5060](#), [5079](#), [5105](#)
  - \msg\_use\_loop\_check:nn .....  
..... [5084](#), [5091](#), [5094](#), [5100](#), [5100](#)
  - \msg\_warning:nn ..... [99](#), [5140](#)
  - \msg\_warning:nnx ..... [99](#), [5140](#)
  - \msg\_warning:nnxx ..... [99](#), [5140](#), [5237](#)
  - \mskip ..... [177](#)
  - \muexpr ..... [426](#)
  - \multiply ..... [78](#)
  - \muskip ..... [374](#)
  - \muskip\_add:Nn .....  
.... [64](#), [3426](#), [3428](#), [3429](#), [5794](#), [5802](#)
  - \muskip\_gadd:Nn [64](#), [3426](#), [3429](#), [5798](#), [5806](#)
  - \muskip\_gset:Nn .... [64](#), [3426](#), [3427](#), [5790](#)
  - \muskip\_gsub:Nn ..... [64](#), [3426](#), [3431](#)
  - \muskip\_new:N .....  
.... [64](#), [3415](#), [3417](#), [3421](#), [5712](#)–[5714](#)
  - \muskip\_new\_l:N ..... [3418](#)
  - \muskip\_set:Nn . [64](#), [3426](#), [3426](#), [3427](#), [5786](#)
  - \muskip\_sub:Nn ..... [64](#), [3426](#), [3430](#), [3431](#)
  - \muskip\_use:N ..... [64](#), [3432](#), [3432](#)
  - \muskipdef ..... [72](#)
  - \mutoglu ..... [432](#)
- N
- \n ..... [2363](#)
  - \name\_pop\_stack:w ..... [693](#), [704](#), [723](#)
  - \name\_primitive:NN .....  
[56](#), [60](#)–[450](#), [452](#)–[472](#), [474](#)–[480](#), [482](#)–  
[485](#), [487](#)–[508](#), [510](#)–[521](#), [523](#)–[553](#), [556](#)
  - \name\_tmp: ..... [723](#)

- \name\_undefine:N . . . . 43, 48, 51, 58, 692
  - \NeedsTeXFormat . . . . . 715
  - \newbox . . . . . 5292
  - \newcount . . . . . 2702
  - \newdimen . . . . . 3363
  - \newif . . . . . 6110
  - \newlinechar . . . . . 128
  - \newmuskup . . . . . 3423
  - \newread . . . . . 4819
  - \newskip . . . . . 3247
  - \newtoks . . . . . 4072
  - \newwrite . . . . . 4769
  - \noalign . . . . . 96
  - \noboundary . . . . . 231
  - \noexpand . . . . . 32, 89
  - \noindent . . . . . 257
  - \nolimits . . . . . 211
  - \nonscript . . . . . 191
  - \nonstopmode . . . . . 154
  - \nulldelimiterspace . . . . . 224
  - \nullfont . . . . . 342
  - \num\_add:cn . . . . . 54, 3061
  - \num\_add:Nn 54, 3037, 3038, 3061, 3061–3063
  - \num\_decr:c . . . . . 53, 3041
  - \num\_decr:N . . . . . 53, 3037, 3038, 3042
  - \num\_elt\_count:n 55, 3069, 3069, 3883, 3889
  - \num\_elt\_count\_prop:Nn . . 55, 3069, 3070
  - \num\_gadd:cn . . . . . 54, 3061
  - \num\_gadd:Nn 54, 3039, 3040, 3061, 3063, 3064
  - \num\_gdecr:c . . . . . 53, 3041
  - \num\_gdecr:N . . . . . 53, 3037, 3040, 3044, 3846, 3855, 4750
  - \num\_gincr:c . . . . . 53, 3041
  - \num\_gincr:N . . . . . 53, 3037, 3039, 3043, 3840, 3849, 4745
  - \num\_gset:cn . . . . . 54, 3051
  - \num\_gset:Nn . . . 54, 3046, 3051, 3055, 3056
  - \num\_gset\_eq:cc . . . . . 54, 3059
  - \num\_gset\_eq:cN . . . . . 54, 3059
  - \num\_gset\_eq:Nc . . . . . 54, 3059
  - \num\_gset\_eq:NN . . . . 54, 3059, 3059, 3060
  - \num\_gzero:c . . . . . 54, 3045
  - \num\_gzero:N . . . . . 54, 3045, 3046, 3048
  - \num\_incr:c . . . . . 53, 3041
  - \num\_incr:N . . . . . 53, 3037, 3037, 3041
  - \num\_new:c . . . . . 53, 3049
  - \num\_new:N . . . . . 53, 3049, 3049, 3050, 3071–3075, 4743
  - \num\_set:cn . . . . . 54, 3051
  - \num\_set:Nn . . . . . 54, 3045, 3051, 3051, 3054, 3055, 3061
  - \num\_set\_eq:cc . . . . . 54, 3057
  - \num\_set\_eq:cN . . . . . 54, 3057
  - \num\_set\_eq:Nc . . . . . 54, 3057
  - \num\_set\_eq:NN . . . . . 54, 3057, 3057, 3058
  - \num\_show:c . . . . . 54, 3067, 3068
  - \num\_show:N . . . . . 54, 3067, 3067
  - \num\_use:c . . . . . 54, 3065, 3066
  - \num\_use:N . . . . . 54, 3065, 3065, 3841, 3844, 3850, 3853, 4746, 4749
  - \num\_zero:c . . . . . 54, 3045
  - \num\_zero:N . . . . . 54, 3045, 3045, 3047
  - \number . . . . . 351, 5494
  - \numexpr . . . . . 423
- O**
- \O . . . . . 2367, 2458
  - \omit . . . . . 97
  - \openin . . . . . 123
  - \openout . . . . . 124
  - \or . . . . . 120
  - \or: . . . . . 7, 55, 58, 755, 757, 856, 1275, 1277, 1279, 1281, 1283, 1285, 1287, 1289, 1291, 2822–2826, 2831–2835, 2859–2867, 2872–2880, 2990–2998, 5898–5900, 5907–5909, 5924–5926, 5933–5935, 5981–5983, 5993–5995, 6019, 6025
  - \outer . . . . . 83
  - \output . . . . . 298
  - \outputpenalty . . . . . 308
  - \over . . . . . 185
  - \overfullrule . . . . . 336
  - \overline . . . . . 216
  - \overwithdelims . . . . . 186
- P**
- \P . . . . . 2367
  - \package\_check\_loaded\_expl: . . . . . 725, 751, 1439, 1685, 2068, 2171, 2688, 3032, 3081, 3236, 3437, 4064, 4217, 4390, 4619, 4761, 4845, 5280, 5414, 5527, 5696, 6172
  - \package\_provides:w . . . . . 698, 699
  - \PackageError . . . . . 16, 727, 6055
  - \pagedepth . . . . . 300
  - \pagediscards . . . . . 441
  - \pagefilllstretch . . . . . 304
  - \pagefillstretch . . . . . 303
  - \pagefilstretch . . . . . 302
  - \pagegoal . . . . . 306
  - \pagenumbering . . . . . 5501, 5509
  - \pageshrink . . . . . 305
  - \pagestretch . . . . . 301
  - \pagetotal . . . . . 307
  - \par . . . . . 256, 691
  - \parfillskip . . . . . 287
  - \parindent . . . . . 280

\parshape .....	272	\pdf_mdffivesum:D .....	505
\parshapedimen .....	422	\pdf_minorversion:D .....	453
\parshapeindent .....	420	\pdf_names:D .....	540
\parshapelength .....	421	\pdf_noligatures:D .....	551
\parskip .....	279	\pdf_normaldeviate:D .....	504
\patterns .....	362	\pdf_obj:D .....	523
\pausing .....	149	\pdf_optionalalwaysusepdfpagebox:D ..	466
\pdf_adjustspacing:D .....	460	\pdf_optionpdfinclusionerrorlevel:D	468
\pdf_annot:D .....	529	\pdf_outline:D .....	532
\pdf_catalog:D .....	539	\pdf_output:D .....	452
\pdf_compresslevel:D .....	454	\pdf_pageattr:D .....	483
\pdf_creationdate:D .....	489	\pdf_pageheight:D .....	477
\pdf_decimaldigits:D .....	455	\pdf_pageref:D .....	490
\pdf_dest:D .....	533	\pdf_pageresources:D .....	484
\pdf_destmargin:D .....	479	\pdf_pagesattr:D .....	482
\pdf_efcode:D .....	462	\pdf_pagewidth:D .....	476
\pdf_elapsedtime:D .....	519	\pdf_pkmode:D .....	485
\pdf_endlink:D .....	531	\pdf_pkresolution:D .....	457
\pdf_endthread:D .....	536	\pdf_protrudechars:D .....	461
\pdf_escapehex:D .....	500	\pdf_randomseed:D .....	520
\pdf_escapename:D .....	499	\pdf_refobj:D .....	524
\pdf_escapestring:D .....	498	\pdf_refxform:D .....	526
\pdf_filedump:D .....	508	\pdf_refximage:D .....	528
\pdf_filemoddate:D .....	506	\pdf_resettimer:D .....	549
\pdf_filesize:D .....	507	\pdf_rightmarginkern:D .....	497
\pdf_fontattr:D .....	543	\pdf_rpcode:D .....	464
\pdf_fontexpand:D .....	545	\pdf_savepos:D .....	537
\pdf_fontname:D .....	492	\pdf_setrandomseed:D .....	550
\pdf_fontobjnum:D .....	493	\pdf_shellesscape:D .....	521
\pdf_fontsize:D .....	494	\pdf_startlink:D .....	530
\pdf_forcepagebox:D .....	465	\pdf_startthread:D .....	535
\pdf_gamma:D .....	471	\pdf_strcmp:D .. 502, 556, 2117, 2139,	3665
\pdf_horigin:D .....	474	\pdf_textrbanner:D .....	488
\pdf_imageapplygamma:D .....	470	\pdf_texrevision:D .....	487
\pdf_imagegamma:D .....	472	\pdf_texversion:D .....	510
\pdf_imagehicolor:D .....	469	\pdf_thread:D .....	534
\pdf_imageresolution:D .....	456	\pdf_threadmargin:D .....	480
\pdf_includechars:D .....	495	\pdf_tracingfonts:D .....	458
\pdf_inclusionerrorlevel:D .....	467	\pdf_trailer:D .....	544
\pdf_info:D .....	538	\pdf_unescapehex:D .....	501
\pdf_lastannot:D .....	515	\pdf_uniformdeviate:D .....	503
\pdf_lastdemerits:D .....	518	\pdf_uniqueresname:D .....	459
\pdf_lastobj:D .....	511	\pdf_vorigin:D .....	475
\pdf_lastxform:D .....	512	\pdf_xform:D .....	525
\pdf_lastximage:D .....	513	\pdf_xformname:D .....	491
\pdf_lastximagepages:D .....	514	\pdf_ximage:D .....	527
\pdf_lastxpos:D .....	516	\pdfadjustspacing .....	460
\pdf_lastypos:D .....	517	\pdfannot .....	529
\pdf_leftmarginkern:D .....	496	\pdfcatalog .....	539
\pdf_linkmargin:D .....	478	\pdfcompresslevel .....	454
\pdf_literal:D .....	547	\pdfcreationdate .....	489
\pdf_lpcode:D .....	463	\pdfdecimaldigits .....	455
\pdf_mapfile:D .....	541	\pdfdest .....	533
\pdf_mapline:D .....	542	\pdfdestmargin .....	479



<code>\pdfelapsedtime</code>	519	<code>\pdfprotrudechars</code>	461
<code>\pdfendlink</code>	531	<code>\pdfrandomseed</code>	520
<code>\pdfendthread</code>	536	<code>\pdfrefobj</code>	524
<code>\pdfescapehex</code>	500	<code>\pdfrefxform</code>	526
<code>\pdfescapename</code>	499	<code>\pdfrefximage</code>	528
<code>\pdfescapestring</code>	498	<code>\pdfresettimer</code>	549
<code>\pdffiledump</code>	508	<code>\pdfsavepos</code>	537
<code>\pdffilemoddate</code>	506	<code>\pdfsetrandomseed</code>	550
<code>\pdffilesize</code>	507	<code>\pdfshellescape</code>	521
<code>\pdffontattr</code>	543	<code>\pdfstartlink</code>	530
<code>\pdffontexpand</code>	545	<code>\pdfstartthread</code>	535
<code>\pdffontname</code>	492	<code>\pdfstrcmp</code>	502
<code>\pdffontobjnum</code>	493	<code>\pdftexbanner</code>	488
<code>\pdffontsize</code>	494	<code>\pdftexrevision</code>	487
<code>\pdfforcepagebox</code>	465	<code>\pdftexversion</code>	510
<code>\pdfgamma</code>	471	<code>\pdfthread</code>	534
<code>\pdfhorigin</code>	474	<code>\pdfthreadmargin</code>	480
<code>\pdfimageapplygamma</code>	470	<code>\pdftracingfonts</code>	458
<code>\pdfimagegamma</code>	472	<code>\pdftrailer</code>	544
<code>\pdfimageghicolor</code>	469	<code>\pdfunescapehex</code>	501
<code>\pdfimageresolution</code>	456	<code>\pdfuniformdeviate</code>	503
<code>\pdfincludechars</code>	495	<code>\pdfuniqueresname</code>	459
<code>\pdfinclusionerrorlevel</code>	467	<code>\pdfvorigin</code>	475
<code>\pdfinfo</code>	538	<code>\pdfxform</code>	525
<code>\pdflastannot</code>	515	<code>\pdfxformname</code>	491
<code>\pdflastdemerits</code>	518	<code>\pdfximage</code>	527
<code>\pdflastobj</code>	511	<code>\peek_after:NN</code>	47, 2526, 2526, 2540, 2555, 2672
<code>\pdflastxform</code>	512	<code>\peek_catcode:NTF</code>	47, 2631
<code>\pdflastximage</code>	513	<code>\peek_catcode_ignore_spaces:NTF</code>	47, 2636
<code>\pdflastximagepages</code>	514	<code>\peek_catcode_remove:NTF</code>	47, 2641
<code>\pdflastxpos</code>	516	<code>\peek_catcode_remove_ignore_spaces:NTF</code>	47, 2646
<code>\pdflastypos</code>	517	<code>\peek_charcode:NTF</code>	47, 2651
<code>\pdflinkmargin</code>	478	<code>\peek_charcode_ignore_spaces:NTF</code>	47, 2656
<code>\pdfliteral</code>	547	<code>\peek_charcode_remove:NTF</code>	47, 2661
<code>\pdfmapfile</code>	541	<code>\peek_charcode_remove_ignore_spaces:NTF</code>	47, 2666
<code>\pdfmapline</code>	542	<code>\peek_def_aux:nnnn</code>	2597, 2597, 2611, 2616, 2621, 2626, 2631, 2636, 2641, 2646, 2651, 2656, 2661, 2666
<code>\pdfmdfivesum</code>	505	<code>\peek_def_aux_ii:nnnnn</code>	2597, 2598–2600, 2602
<code>\pdfminorversion</code>	453	<code>\peek_execute_branches:</code>	2605, 2679
<code>\pdfnames</code>	540	<code>\peek_execute_branches_catcode:</code>	48, 2567, 2574, 2635, 2638, 2645, 2648
<code>\pdfnoligatures</code>	551	<code>\peek_execute_branches_charcode:</code>	48, 2567, 2581, 2655, 2658, 2665, 2668
<code>\pdfnormaldeviate</code>	504	<code>\peek_execute_branches_charcode_aux:NN</code>	2567, 2587, 2589
<code>\pdfobj</code>	523	<code>\peek_execute_branches_meaning:</code>	48, 2567, 2567, 2615, 2618, 2625, 2628
<code>\pdfoptionalalwaysusepdfpagebox</code>	466	<code>\peek_gafter:NN</code>	47, 2526, 2527
<code>\pdfoptionpdfinclusionerrorlevel</code>	468		
<code>\pdfoutline</code>	532		
<code>\pdfoutput</code>	452		
<code>\pdfpageattr</code>	483		
<code>\pdfpageheight</code>	477		
<code>\pdfpageref</code>	490		
<code>\pdfpageresources</code>	484		
<code>\pdfpagesattr</code>	482		
<code>\pdfpagewidth</code>	476		
<code>\pdfpkmode</code>	485		
<code>\pdfpkresolution</code>	457		



- \peek\_ignore\_spaces\_aux: ..... 184, 2671, 2671, 2676
- \peek\_ignore\_spaces\_execute\_branches: ..... 184, 2620, 2630, 2640, 2650, 2660, 2670, 2671, 2672, 2674
- \peek\_meaning:NTF ..... 47, 2611
- \peek\_meaning\_ignore\_spaces:NTF 47, 2616
- \peek\_meaning\_remove:NTF ..... 47, 2621
- \peek\_meaning\_remove\_ignore\_spaces:NTF ..... 47, 2626
- \peek\_tmp:w ... 184, 2532, 2532, 2565, 2677
- \peek\_token\_generic:NNF ..... 2545
- \peek\_token\_generic:NNT ..... 2542
- \peek\_token\_generic:NNTF ..... 48, 2534, 2534, 2543, 2546
- \peek\_token\_remove\_generic:NNF .. 2560
- \peek\_token\_remove\_generic:NNT .. 2557
- \peek\_token\_remove\_generic:NNTF .... 48, 2548, 2548, 2558, 2561
- \penalty ..... 357
- \postdisplaypenalty ..... 204
- \predisplaydirection ..... 448
- \predisdisplaypenalty ..... 203
- \predisplaysize ..... 202
- \pref\_global:D ..... 21, 790, 790, 1215, 1221, 2224, 2528, 2715, 2733, 2739, 2751, 2769, 2775, 3055, 3063, 3261, 3276, 3298, 3306, 3369, 3373, 3379, 3383, 3427, 3429, 3431, 3611, 4094, 4095, 4106, 4111, 4121, 4131, 4145, 4159, 4312, 4321, 4835, 5322, 5327, 5335, 5357, 5363, 5369, 5383, 5389, 5397, 5726, 5829, 5832, 5833, 5888, 6016, 6022, 6028, 6034
- \pref\_global\_chk: ..... 2713, 2731, 2737, 2767, 2773, 3259, 3274, 3296, 3304, 3609, 4120, 4130, 4144, 4158, 4309, 4318, 4834
- \pref\_long:D ..... 21, 790, 791, 796, 799, 808, 811, 816, 819, 828, 831
- \pref\_protected:D ..... 21, 790, 792, 795, 798, 801, 802, 804, 805, 808, 811, 822, 825, 828, 831
- \pretolerance ..... 283
- \prevdepth ..... 330
- \prevgraf ..... 289
- \prg\_case\_dim:nnn ..... 34, 1952, 1952
- \prg\_case\_dim\_aux:nnn ..... 1952, 1953, 1956, 1960
- \prg\_case\_int:nnn ..... 34, 1942, 1942
- \prg\_case\_int\_aux:nnn ..... 1942, 1943, 1946, 1950
- \prg\_case\_str:nnn ..... 34, 1962, 1962
- \prg\_case\_str\_aux:nnn ..... 1962, 1963, 1966, 1970
- \prg\_case\_tl:Nnn ..... 34, 1972, 1972
- \prg\_case\_tl\_aux:NNn 1972, 1973, 1976, 1980
- \prg\_conditional\_form\_F:nnn ..... 1431
- \prg\_conditional\_form\_p:nnn ..... 1428
- \prg\_conditional\_form\_T:nnn ..... 1430
- \prg\_conditional\_form\_TF:nnn .... 1429
- \prg\_define\_quicksort:nnn 1983, 1983, 2058
- \prg\_do\_nothing: 13, 1252, 1252, 3976, 3980, 3996, 3999, 4081, 4110, 5844
- \prg\_end\_case:nw ..... 1949, 1959, 1969, 1979, 1982, 1982
- \prg\_generate\_conditional\_aux:nnNNnnnn ..... 872, 878, 884, 890, 896, 902, 909, 916, 922
- \prg\_generate\_conditional\_aux:nnw .. 923, 927, 932
- \prg\_generate\_conditional\_parm\_aux:nnNNnnnn ..... 922
- \prg\_generate\_conditional\_parm\_aux:nw ..... 922
- \prg\_generate\_F\_form\_count:Nnnnn ... 954, 969
- \prg\_generate\_F\_form\_parm:Nnnnn 934, 949
- \prg\_generate\_p\_form\_count:Nnnnn ... 954, 954
- \prg\_generate\_p\_form\_parm:Nnnnn 934, 934
- \prg\_generate\_T\_form\_count:Nnnnn ... 954, 964
- \prg\_generate\_T\_form\_parm:Nnnnn 934, 944
- \prg\_generate\_TF\_form\_count:Nnnnn .. 954, 959
- \prg\_generate\_TF\_form\_parm:Nnnnn ... 934, 939
- \prg\_get\_count\_aux:nn ..... 895, 901, 908, 915, 920, 920
- \prg\_get\_parm\_aux:nw ..... 871, 877, 883, 889, 920, 921
- \prg\_new\_conditional:Nnn ..... 30, 894, 900, 2110, 2116, 2125, 2134, 2138, 2147, 2286, 2290, 2294, 2298, 2302, 2306, 2310, 2314, 2318, 2322, 2326, 2330, 2334, 2338, 2342, 2349, 2352, 2371, 2378, 2385, 2396, 2407, 2418, 2429, 2436, 2443, 2483, 3337, 3399, 4186, 4193, 4252, 4423, 4722, 4826, 5299, 5302, 5313, 6181
- \prg\_new\_conditional:Npnn ..... 30, 870, 876, 1235, 3632, 3639, 3651, 3707, 4031, 4035, 4053
- \prg\_new\_eq\_conditional:NNn ..... 1408, 1411, 4244, 4245, 4411, 4412, 4419-4422, 4716-4721

- \prg\_new\_protected\_conditional:Nnn . 1926, 3093, 3135, 3200, 3204, 3624
- ..... 30, 894, 914
- \prg\_new\_protected\_conditional:Npnn
- ..... 30, 870, 888, 3946, 3955
- \prg\_quicksort:n ..... 37, 2058
- \prg\_quicksort\_compare:nnTF .....
- ..... 37, 2059, 2060
- \prg\_quicksort\_function:n 37, 2059, 2059
- \prg\_replicate:nn ..... 36, 1717, 1717
- \prg\_replicate\_ ..... 1729
- \prg\_replicate\_aux:N 1717, 1723, 1724, 1727
- \prg\_replicate\_first\_aux:N .....
- ..... 1717, 1719, 1726
- \prg\_return\_false: 30, 868, 869, 1014,
- 1019, 1031, 1036, 1054, 1057, 1114,
- 1116, 1133, 1237, 1246, 1250, 1393,
- 1690, 1694, 1698, 1702, 1851, 1928,
- 2112, 2120, 2130, 2135, 2142, 2152,
- 2288, 2292, 2296, 2300, 2304, 2308,
- 2312, 2316, 2320, 2324, 2328, 2332,
- 2336, 2340, 2347, 2351, 2355, 2357,
- 2376, 2383, 2387, 2394, 2398, 2405,
- 2416, 2420, 2427, 2434, 2441, 2449,
- 2486, 2493, 2498, 2505, 2507, 2509,
- 2511, 2513, 3109, 3113, 3117, 3121,
- 3125, 3129, 3133, 3137, 3202, 3206,
- 3341, 3401, 3626, 3633, 3641, 3655,
- 3711, 3725, 3732, 3739, 3746, 3753,
- 3760, 3767, 3774, 3781, 3788, 3795,
- 3802, 3809, 3816, 3948, 3957, 4033,
- 4038, 4056, 4187, 4195, 4255, 4426,
- 4726, 4827, 5300, 5303, 5314, 6211
- \prg\_return\_true: ..... 30, 868,
- 868, 1017, 1033, 1052, 1113, 1133,
- 1237, 1245, 1249, 1393, 1690, 1694,
- 1698, 1702, 1851, 1928, 2112, 2120,
- 2130, 2135, 2142, 2152, 2288, 2292,
- 2296, 2300, 2304, 2308, 2312, 2316,
- 2320, 2324, 2328, 2332, 2336, 2340,
- 2347, 2351, 2355, 2376, 2383, 2394,
- 2405, 2416, 2427, 2434, 2441, 2449,
- 2515, 3109, 3113, 3117, 3121, 3125,
- 3129, 3133, 3137, 3202, 3206, 3341,
- 3401, 3626, 3633, 3641, 3655, 3711,
- 3725, 3732, 3739, 3746, 3753, 3760,
- 3767, 3774, 3781, 3788, 3795, 3802,
- 3809, 3816, 3948, 3957, 4033, 4038,
- 4056, 4187, 4195, 4255, 4426, 4726,
- 4827, 5300, 5303, 5314, 6187, 6209
- \prg\_set\_conditional:Nnn ... 30, 894, 894
- \prg\_set\_conditional:Npnn .....
- ..... 30, 870, 870, 1012, 1023,
- 1044, 1108, 1117, 1125, 1244, 1248,
- 1391, 1688, 1692, 1696, 1700, 1850,
- \prg\_set\_eq\_conditional:NNn . 1408, 1408
- \prg\_set\_eq\_conditional\_aux:NNNn ...
- ..... 1409, 1412, 1414, 1414
- \prg\_set\_eq\_conditional\_aux:NNNw ...
- ..... 1414, 1415, 1417, 1426
- \prg\_set\_protected\_conditional:Nnn .
- ..... 30, 894, 907
- \prg\_set\_protected\_conditional:Npnn
- ..... 30, 870, 882, 3721, 3728,
- 3735, 3742, 3749, 3756, 3763, 3770,
- 3777, 3784, 3791, 3798, 3805, 3812
- \prg\_stepwise\_function:nnnN .....
- ..... 36, 1757, 1757
- \prg\_stepwise\_function\_decr:nnnN ...
- ..... 1757, 1759, 1772, 1776
- \prg\_stepwise\_function\_incr:nnnN ...
- ..... 1757, 1760, 1763, 1767
- \prg\_stepwise\_inline:nnnn 36, 1781, 1782
- \prg\_stepwise\_inline\_decr:Nnnn ....
- ..... 1786, 1800, 1804
- \prg\_stepwise\_inline\_decr:nnnn .. 1781
- \prg\_stepwise\_inline\_incr:Nnnn ....
- ..... 1787, 1792, 1796
- \prg\_stepwise\_inline\_incr:nnnn .. 1781
- \prg\_stepwise\_variable:nnnN .....
- ..... 36, 1808, 1808
- \prg\_stepwise\_variable\_decr:nnnN ..
- ..... 1808, 1810, 1822, 1826
- \prg\_stepwise\_variable\_incr:nnnN ..
- ..... 1808, 1811, 1814, 1818
- \ProcessOptions ..... 53
- \prop\_clear:c ..... 90, 4625, 4626, 5044
- \prop\_clear:N ..... 90, 4625, 4625, 4681
- \prop\_del:Nn ..... 91, 4701, 4701
- \prop\_del\_aux:w 93, 4701, 4702, 4704, 4705
- \prop\_display:c ..... 92, 4639
- \prop\_display:N .... 92, 4639, 4639, 4655
- \prop\_gclear:c ..... 90, 4625, 4628
- \prop\_gclear:N ..... 90, 4625, 4627, 4687
- \prop\_gdel:Nn ..... 91, 4701, 4703
- \prop\_get:cnN ..... 91, 4662, 5101
- \prop\_get:NnN ..... 91, 4662, 4662, 4666
- \prop\_get\_aux:w .... 93, 4662, 4663, 4665
- \prop\_get\_del\_aux:w . 93, 4671, 4672, 4673
- \prop\_get\_gdel:NnN ..... 91, 4671, 4671
- \prop\_gget:cnN ..... 91, 4667
- \prop\_gget:NnN ..... 91, 4667, 4667, 4670
- \prop\_gget\_aux:w ... 93, 4667, 4668, 4669
- \prop\_gput:ccx ..... 91, 4679, 5422
- \prop\_gput:cnn ..... 91, 4679
- \prop\_gput:Nnn ..... 91, 4679, 4685, 4700
- \prop\_gput:Nno ..... 91, 4679
- \prop\_gput:NnV ..... 91, 4679

- \prop\_gput:Nnx ..... 91, 4679
  - \prop\_gput\_if\_new:Nnn ... 91, 4712, 4712
  - \prop\_gset\_eq:cc ..... 91, 4629, 4636
  - \prop\_gset\_eq:cN ..... 91, 4629, 4635
  - \prop\_gset\_eq:Nc ..... 91, 4629, 4634
  - \prop\_gset\_eq:NN ..... 91, 4629, 4633
  - \prop\_if\_empty:c ..... 4717
  - \prop\_if\_empty:cTF ..... 92, 4716
  - \prop\_if\_empty:N ..... 4716
  - \prop\_if\_empty:NTF ..... 92, 4716
  - \prop\_if\_empty\_p:c ..... 92, 4716
  - \prop\_if\_empty\_p:N ..... 92, 4716
  - \prop\_if\_eq:cc ..... 4721
  - \prop\_if\_eq:ccTF ..... 92, 4718
  - \prop\_if\_eq:cN ..... 4719
  - \prop\_if\_eq:cNTF ..... 92, 4718
  - \prop\_if\_eq:Nc ..... 4720
  - \prop\_if\_eq:NcTF ..... 92, 4718
  - \prop\_if\_eq:NN ..... 4718
  - \prop\_if\_eq:NNTF ..... 92, 4718
  - \prop\_if\_eq\_p:cc ..... 92, 4718
  - \prop\_if\_eq\_p:cN ..... 92, 4718
  - \prop\_if\_eq\_p:Nc ..... 92, 4718
  - \prop\_if\_eq\_p:NN ..... 92, 4718
  - \prop\_if\_in:ccTF ..... 93, 4722
  - \prop\_if\_in:cnTF ... 93, 4722, 5090, 5093
  - \prop\_if\_in:Nn ..... 4722
  - \prop\_if\_in:NnF ..... 4730
  - \prop\_if\_in:NnT ..... 4729
  - \prop\_if\_in:NnTF ... 93, 4722, 4728, 5083
  - \prop\_if\_in:NoTF ..... 93, 4722
  - \prop\_if\_in:NVTF ..... 93, 4722
  - \prop\_if\_in\_aux:w .. 93, 4722, 4723, 4725
  - \prop\_map\_break: ... 92, 4737, 4753, 4753
  - \prop\_map\_function:cc ..... 92, 4731
  - \prop\_map\_function:cN ..... 92, 4731
  - \prop\_map\_function:Nc ... 92, 4731, 4748
  - \prop\_map\_function:NN 92, 4731, 4731, 4742
  - \prop\_map\_function\_aux:w .....  
..... 93, 4731, 4732, 4735, 4740
  - \prop\_map\_inline:cn ..... 92, 4743
  - \prop\_map\_inline:Nn .....  
..... 92, 4643, 4743, 4744, 4752
  - \prop\_new:c .... 90, 4623, 4624, 5040, 5456
  - \prop\_new:N .....  
90, 4623, 4623, 4893, 4894, 5417, 5418
  - \prop\_put:cn ..... 91, 4679, 5168, 5171
  - \prop\_put:Nnn .. 91, 4679, 4679, 4699, 5174
  - \prop\_put:NnV ..... 91, 4679
  - \prop\_put\_aux:w 93, 4679, 4682, 4688, 4691
  - \prop\_put\_if\_new\_aux:w 93, 4712, 4713, 4714
  - \prop\_set\_eq:cc ..... 91, 4629, 4632
  - \prop\_set\_eq:cN ..... 91, 4629, 4631
  - \prop\_set\_eq:Nc ..... 91, 4629, 4630
  - \prop\_set\_eq:NN ..... 91, 4629, 4629
  - \prop\_show:c ..... 92, 4637, 4638
  - \prop\_show:N ..... 92, 4637, 4637
  - \prop\_split\_aux:Nnn .....  
. 93, 4656, 4656, 4663, 4668, 4672,  
4680, 4686, 4702, 4704, 4713, 4723
  - \prop\_tmp:w ..... 4657, 4660,  
4676, 4677, 4695, 4696, 4706–4709
  - \protect ..... 729
  - \protected ..... 450
  - \ProvidesClass ..... 656
  - \ProvidesExplClass ..... 5, 650, 655
  - \ProvidesExplPackage .....  
..... 5, 650, 651, 749, 1437, 1683,  
2066, 2169, 2686, 3030, 3079, 3234,  
3435, 4062, 4215, 4388, 4617, 4759,  
4843, 5278, 5412, 5525, 5694, 6170
  - \ProvidesPackage ..... 652, 695
- Q**
- \Q ..... 5552
  - \q ..... 991, 1720, 1893
  - \q\_error ..... 39, 2075, 2075
  - \q\_mark ..... 39, 2075, 2076, 4139
  - \q\_nil ..... 39, 671, 675,  
848, 851, 977, 983, 991, 993, 1000,  
1002, 1264, 1268, 1905, 1907, 1986,  
1990, 2072, 2074, 2081, 2093, 2109,  
2135, 2140, 2150, 2343, 2345, 2373,  
2375, 2380, 2382, 2390, 2393, 2401,  
2404, 2412, 2415, 2423, 2426, 2431,  
2433, 2438, 2440, 2445, 2448, 2461,  
2467, 2473, 2479, 3007, 3099, 3107,  
3111, 3115, 3119, 3123, 3127, 3131,  
3640, 3652, 3654, 4021, 4023, 4025,  
4029, 4030, 4032, 4037, 4041, 4055,  
4336, 4450, 4457, 4733, 4736, 5433,  
5440, 5553, 5560, 5568, 5578, 5583,  
5591, 5598, 5608, 5614, 5615, 5625,  
5631, 5636, 5649, 5654, 5658, 5661
  - \q\_no\_value ..... 39, 1905, 2072,  
2073, 2111, 2118, 2128, 3950, 3959,  
3973, 3981, 4000, 4254, 4257, 4425,  
4428, 4660, 4676, 4695, 4708, 4733,  
5590, 5598, 5604, 5609, 5625, 5632
  - \q\_prop ..... 93, 4622, 4622,  
4657, 4660, 4676, 4692, 4695, 4708,  
4715, 4733, 4735, 5429, 5431, 5448
  - \q\_recursion\_stop .....  
... 39, 850, 853, 925, 1415, 1601,  
1944, 1954, 1964, 1974, 1982, 2077,  
2078, 3828, 3832, 3845, 3854, 3860,  
3871, 4499, 4505, 4522, 4534, 4541

- `\q_recursion_tail` ..... 39, 1944, 1954, 1964, 1974,  
     2077, 2077, 2081, 2086, 2093, 2101,  
     2109, 3828, 3832, 3845, 3854, 3860,  
     3871, 4499, 4505, 4522, 4534, 4541  
`\q_stop` ..... 39,  
     849, 852, 1986, 1990, 2052, 2072,  
     2072, 3095, 3097, 3099, 3101, 3947,  
     3950, 3956, 3959, 3969, 3981, 3989,  
     3996, 4000, 4140, 4149, 4253, 4257,  
     4264, 4266, 4272, 4275, 4297, 4299,  
     4336, 4424, 4428, 4444, 4446, 4450,  
     4452, 4657, 4660, 4733, 5560, 5626  
`\quark_if_nil:N` ..... 2134  
`\quark_if_nil:n` ..... 2138, 2147  
`\quark_if_nil:nF` ..... 2159, 2163  
`\quark_if_nil:NT` ..... 4330  
`\quark_if_nil:nT` . 1993, 1997, 2158, 2162  
`\quark_if_nil:NTF` 38, 2134, 3010, 4455, 5644  
`\quark_if_nil:nTF` ..... 38, 2001,  
     2010, 2019, 2028, 2137, 2157, 2161  
`\quark_if_nil:oF` ..... 5606  
`\quark_if_nil:oTF` ..... 38, 2137  
`\quark_if_nil:VTF` ..... 38, 2137  
`\quark_if_nil_p:N` ..... 38, 2134  
`\quark_if_nil_p:n` .. 38, 2137, 2156, 2160  
`\quark_if_nil_p:o` ..... 38, 2137  
`\quark_if_nil_p:V` ..... 38, 2137  
`\quark_if_no_value:N` ..... 2110  
`\quark_if_no_value:n` ..... 2116, 2125  
`\quark_if_no_value:NF` ..... 4675, 4707  
`\quark_if_no_value:nF` ..... 3970  
`\quark_if_no_value:NTF` .. 38, 1908, 2110  
`\quark_if_no_value:nTF` .....  
     38, 2110, 3948, 3957, 3990, 4726  
`\quark_if_no_value_p:N` ..... 38, 2110  
`\quark_if_no_value_p:n` ..... 38, 2110  
`\quark_if_recursion_tail_aux:w` ....  
     2081, 2093, 2109, 2109  
`\quark_if_recursion_tail_stop:N` ....  
     39, 2079, 2085, 3866, 4548  
`\quark_if_recursion_tail_stop:n` ....  
     39, 2079, 2079, 2090, 3835, 4509  
`\quark_if_recursion_tail_stop:o` 39, 2079  
`\quark_if_recursion_tail_stop_do:Nn`  
     39, 1977, 2091, 2100  
`\quark_if_recursion_tail_stop_do:nn`  
     39, 1947,  
     1957, 1967, 2091, 2091, 2108, 3874  
`\quark_if_recursion_tail_stop_do:on`  
     39, 2091  
`\quark_new:N` ... 38, 2071, 2071–2078, 4622
- R**
- `\R` ..... 2367, 2457  
`\radical` ..... 178  
`\raise` ..... 318  
`\ratio` ..... 111, 5721, 6098, 6102  
`\read` ..... 125  
`\readline` ..... 400  
`\real` ..... 111, 5720, 6098, 6103  
`\relax` .... 11, 23, 26–31, 34–39, 160, 704  
`\relpenalty` ..... 221  
`\RequirePackage` ..... 716  
`\reverse_if:N` ..... 7, 755, 760  
`\right` ..... 219  
`\rightthyphenmin` ..... 275  
`\rightmarginkern` ..... 497  
`\rightskip` ..... 277  
`\romannumeral` ..... 352  
`\rPCODE` ..... 464
- S**
- `\S` ..... 2367  
`\savingshyphcodes` ..... 439  
`\savingsdiscards` ..... 440  
`\scan_align_safe_stop:` 36, 1701, 1707, 1707  
`\scan_stop:` ..... 22, 736,  
     786, 786, 985, 1109, 1113, 1114,  
     1122, 1124, 1126, 1129, 1133, 1418,  
     1713, 2350, 2468, 2474, 2480, 2904,  
     2905, 2922, 3076, 3267, 3319, 3351,  
     3352, 3376, 3381, 3397, 3426, 3428,  
     3430, 3895–3898, 4201, 4434, 4774,  
     4822, 4823, 4974, 5354, 5380, 5842,  
     5902, 5911, 5928, 5937, 5985, 6000  
`\scantokens` ..... 398  
`\scriptfont` ..... 344  
`\scriptscriptfont` ..... 345  
`\scriptscriptstyle` ..... 190  
`\scriptspace` ..... 230  
`\scriptstyle` ..... 189  
`\scrollmode` ..... 155  
`\secondoftwo` ..... 4, 12  
`\seq_clear:c` ..... 79, 4224, 4225  
`\seq_clear:N` ..... 79, 4224, 4224, 4356  
`\seq_clear_new:c` ..... 79, 4228, 4229  
`\seq_clear_new:N` ..... 79, 4228, 4228  
`\seq_display:c` ..... 82, 4281  
`\seq_display:N` ..... 82, 4281, 4281, 4295  
`\seq_elt:w` ..... 83, 4219, 4220,  
     4253, 4257, 4266, 4275, 4301, 4305,  
     4328, 4336, 4343, 4345, 4349, 4351  
`\seq_elt_end:` ..... 83,  
     4219, 4221, 4253, 4257, 4266, 4275,  
     4301, 4305, 4328, 4336, 4343, 4349  
`\seq_gclear:c` ..... 79, 4224, 4227

- \seq\_gclear:N ..... 79, [4224](#), [4226](#)
- \seq\_gclear\_new:c ..... 79, [4228](#), [4231](#)
- \seq\_gclear\_new:N ..... 79, [4228](#), [4230](#)
- \seq\_gconcat:ccc ..... 79, [4240](#)
- \seq\_gconcat:NNN ... 79, [4240](#), [4240](#), [4243](#)
- \seq\_get:cN ..... 81, [4262](#), [4382](#)
- \seq\_get:NN .... 81, [4262](#), [4262](#), [4269](#), [4381](#)
- \seq\_get\_aux:w ..... 83, [4262](#), [4264](#), [4266](#)
- \seq\_gpop:cN ..... 83, [4377](#)
- \seq\_gpop:NN ..... 83, [4377](#), [4378](#), [4380](#)
- \seq\_gpush:cn ..... 83, [4377](#)
- \seq\_gpush:Nn ..... 83, [4377](#), [4377](#), [4379](#)
- \seq\_gpush:No ..... 83, [4377](#)
- \seq\_gpush:Nv ..... 83, [4377](#)
- \seq\_gpush:Nv ..... 83, [4377](#)
- \seq\_gput\_left:cn ..... 80, [4307](#)
- \seq\_gput\_left:co ..... 80, [4307](#)
- \seq\_gput\_left:cV ..... 80, [4307](#)
- \seq\_gput\_left:Nn 80, [4307](#), [4307](#), [4325](#), [4377](#)
- \seq\_gput\_left:No ..... 80, [4307](#)
- \seq\_gput\_left:Nv ..... 80, [4307](#)
- \seq\_gput\_left:Nx ..... 80, [4307](#)
- \seq\_gput\_right:cn ..... 80, [4307](#)
- \seq\_gput\_right:co ..... 80, [4307](#)
- \seq\_gput\_right:cV ..... 80, [4307](#)
- \seq\_gput\_right:Nc ..... 80, [4327](#)
- \seq\_gput\_right:Nn ..... 80, [4307](#), [4316](#), [4326](#), [4327](#)
- \seq\_gput\_right:No ..... 80, [4307](#)
- \seq\_gput\_right:Nv ..... 80, [4307](#)
- \seq\_gput\_right:Nx ..... 80, [4307](#)
- \seq\_gremove\_duplicates:N 82, [4355](#), [4368](#)
- \seq\_gset\_eq:cc ..... 79, [4236](#), [4239](#)
- \seq\_gset\_eq:cN ..... 79, [4236](#), [4237](#)
- \seq\_gset\_eq:Nc ..... 79, [4236](#), [4238](#)
- \seq\_gset\_eq:NN .... 79, [4236](#), [4236](#), [4369](#)
- \seq\_if\_empty:c ..... [4245](#)
- \seq\_if\_empty:cTF ..... 82, [4244](#)
- \seq\_if\_empty:N ..... [4244](#)
- \seq\_if\_empty:NTF ..... 82, [4244](#)
- \seq\_if\_empty\_err:N ..... 82, [4246](#), [4246](#), [4263](#), [4271](#)
- \seq\_if\_empty\_p:c ..... 82, [4244](#)
- \seq\_if\_empty\_p:N ..... 82, [4244](#)
- \seq\_if\_in:cnTF ..... 82, [4252](#)
- \seq\_if\_in:coTF ..... 82, [4252](#)
- \seq\_if\_in:cVTF ..... 82, [4252](#)
- \seq\_if\_in:cxTF ..... 82, [4252](#)
- \seq\_if\_in:Nn ..... [4252](#)
- \seq\_if\_in:NnF ..... [4261](#), [4361](#)
- \seq\_if\_in:NnT ..... [4260](#)
- \seq\_if\_in:NnTF ..... 82, [4252](#), [4259](#)
- \seq\_map\_break: .... 83, [4328](#), [4330](#), [4340](#)
- \seq\_map\_break:n ..... 83, [4328](#), [4341](#)
- \seq\_map\_function:cN ..... 81, [4342](#)
- \seq\_map\_function:NN ..... 81, [4342](#), [4342](#), [4347](#), [4357](#)
- \seq\_map\_inline:cn ..... 81, [4348](#)
- \seq\_map\_inline:Nn ..... 81, [4285](#), [4348](#), [4348](#), [4353](#)
- \seq\_map\_variable:cNn ..... 81, [4328](#)
- \seq\_map\_variable:NNn 81, [4328](#), [4334](#), [4338](#)
- \seq\_map\_variable\_aux:Nnw ..... 4328, [4328](#), [4332](#), [4335](#)
- \seq\_new:c ..... 3, 79, [4222](#), [4223](#)
- \seq\_new:N ..... 3, 79, [4222](#), [4222](#), [4354](#)
- \seq\_pop:cN ..... 83, [4371](#)
- \seq\_pop:NN ..... 83, [4371](#), [4375](#), [4376](#)
- \seq\_pop\_aux:nnNN 82, [4270](#), [4270](#), [4375](#), [4378](#)
- \seq\_pop\_aux:w ..... 83, [4270](#), [4272](#), [4274](#)
- \seq\_push:cn ..... 83, [4371](#), [4374](#)
- \seq\_push:Nn ..... 83, [4371](#), [4371](#)
- \seq\_push:No ..... 83, [4371](#), [4373](#)
- \seq\_push:Nv ..... 83, [4371](#), [4372](#)
- \seq\_put\_aux:Nnn 83, [4296](#), [4296](#), [4301](#), [4305](#)
- \seq\_put\_aux:w ..... 83, [4296](#), [4297](#), [4299](#)
- \seq\_put\_left:cn ..... 80, [4300](#), [4374](#)
- \seq\_put\_left:co ..... 80, [4300](#)
- \seq\_put\_left:cV ..... 80, [4300](#)
- \seq\_put\_left:Nn ..... 80, [4300](#), [4300](#), [4303](#), [4314](#), [4371](#)
- \seq\_put\_left:No ..... 80, [4300](#), [4373](#)
- \seq\_put\_left:Nv ..... 80, [4300](#), [4372](#)
- \seq\_put\_left:Nx ..... 80, [4300](#)
- \seq\_put\_right:cn ..... 80, [4300](#)
- \seq\_put\_right:co ..... 80, [4300](#)
- \seq\_put\_right:cV ..... 80, [4300](#)
- \seq\_put\_right:Nn ..... 80, [4202](#), [4300](#), [4304](#), [4306](#), [4323](#), [4362](#), [5285](#)
- \seq\_put\_right:No ..... 80, [4300](#)
- \seq\_put\_right:Nv ..... 80, [4300](#)
- \seq\_put\_right:Nx ..... 80, [4300](#)
- \seq\_remove\_duplicates:N . 82, [4355](#), [4365](#)
- \seq\_remove\_duplicates\_aux:n ..... 4355, [4357](#), [4360](#)
- \seq\_remove\_duplicates\_aux:NN ..... 4355, [4355](#), [4366](#), [4369](#)
- \seq\_set\_eq:cc ..... 79, [4232](#), [4235](#)
- \seq\_set\_eq:cN ..... 79, [4232](#), [4233](#)
- \seq\_set\_eq:Nc ..... 79, [4232](#), [4234](#)
- \seq\_set\_eq:NN ..... 79, [4232](#), [4232](#), [4366](#)
- \seq\_show:c ..... 81, [4279](#), [4280](#)
- \seq\_show:N ..... 81, [4279](#), [4279](#)
- \seq\_tmp:w ..... [4253](#), [4257](#)
- \seq\_top:cN ..... 83, [4381](#), [4382](#)
- \seq\_top:NN ..... 83, [4381](#), [4381](#)
- \setbox ..... 326
- \setcounter ..... 111, [6108](#), [6154](#), [6160](#)

- \setlabel ..... 5497, 5503, 5504,  
5506, 5507, 5511, 5512, 5514, 5515
  - \setlanguage ..... 84
  - \setlength ..... 111, 6104, 6104
  - \setname ..... 5488,  
5503, 5504, 5506, 5507, 5511, 5514
  - \sfcode ..... 381
  - \shipout ..... 291
  - \show ..... 134
  - \showbox ..... 136
  - \showboxbreadth ..... 150
  - \showboxdepth ..... 151
  - \showgroups ..... 411
  - \showifs ..... 412
  - \showlists ..... 137
  - \showMemUsage .....  
. 737, 746, 1434, 1680, 2063, 2166,  
2683, 3027, 4059, 4212, 4385, 4614,  
4756, 4840, 5409, 5472, 5691, 6167
  - \showthe ..... 135
  - \showtokens ..... 399
  - \skewchar ..... 348
  - \skip ..... 372
  - \skip@ ..... 3321
  - \skip\_add:cn ..... 59, 3281
  - \skip\_add:Nn 59, 3281, 3281, 3287, 3299, 5773
  - \skip\_eval:n ..... 60, 3252,  
3282, 3289, 3311, 3314, 3319, 3319
  - \skip\_gadd:cn ..... 59, 3281
  - \skip\_gadd:Nn .. 59, 3281, 3294, 3301, 5776
  - \skip\_gset:cn ..... 59, 3251
  - \skip\_gset:Nn .. 59, 3251, 3257, 3265, 5770
  - \skip\_gsub:Nn ..... 59, 3281, 3302, 5782
  - \skip\_gzero:c ..... 59, 3266
  - \skip\_gzero:N ..... 59, 3266, 3272, 3280
  - \skip\_horizontal:c ..... 60, 3309
  - \skip\_horizontal:N .. 60, 3309, 3309–3311
  - \skip\_horizontal:n ..... 60, 3309, 3311
  - \skip\_if\_infinite\_glue:n ..... 3337
  - \skip\_if\_infinite\_glue:nTF 60, 3337, 3344
  - \skip\_if\_infinite\_glue\_p:n .... 60, 3337
  - \skip\_new:c ..... 59, 3239
  - \skip\_new:N 59, 3239, 3241, 3245, 3250,  
3322–3326, 3328, 3330, 5709–5711
  - \skip\_new\_l:N ..... 3242
  - \skip\_set:cn ..... 59, 3251
  - \skip\_set:Nn ..... 59, 3251,  
3251, 3262, 3264, 3329, 3331, 5767
  - \skip\_show:c ..... 60, 3317, 3318
  - \skip\_show:N ..... 60, 3317, 3317, 3318
  - \skip\_split\_finite\_else\_action:nnNN  
..... 60, 3343, 3343
  - \skip\_sub:Nn ... 59, 3281, 3288, 3307, 5779
  - \skip\_use:c ..... 60, 3315
  - \skip\_use:N ..... 60, 3315, 3315, 3316
  - \skip\_vertical:c ..... 60, 3309
  - \skip\_vertical:N ... 60, 3309, 3312–3314
  - \skip\_vertical:n ..... 60, 3309, 3314
  - \skip\_zero:c ..... 59, 3266
  - \skip\_zero:N ... 59, 3266, 3266, 3277, 3279
  - \skipdef ..... 71
  - \spacefactor ..... 290
  - \spaceskip ..... 285
  - \span ..... 98
  - \special ..... 360, 548
  - \splitbotmark ..... 169
  - \splitbotmarks ..... 395
  - \splitdiscards ..... 442
  - \splitfirstmark ..... 168
  - \splitfirstmarks ..... 394
  - \splitmaxdepth ..... 338
  - \splittopskip ..... 339
  - \startrecording ..... 5478, 5485
  - \stepcounter ..... 111, 6108, 6156, 6162
  - \str\_if\_eq:nn ..... 1108
  - \str\_if\_eq\_p:nn ..... 10, 1042, 1108
  - \str\_if\_eq\_p\_aux:w .....  
..... 1108, 1109, 1111, 1115, 1119
  - \str\_if\_eq\_var:nf ..... 1125
  - \str\_if\_eq\_var\_p:nf . 10, 1125, 2127, 2149
  - \str\_if\_eq\_var\_start:nnN .....  
..... 1125, 1126, 1128, 1130, 1132
  - \str\_if\_eq\_var\_stop:w .. 1125, 1129, 1132
  - \strcmp ..... 554, 556
  - \string ..... 353
- T**
- \T ..... 2361
  - \t ..... 2364
  - \tabskip ..... 99
  - TeX and L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> commands:
  - \@Roman ..... 50
  - \@alph ..... 50
  - \@arabic ..... 50
  - \@empty ..... 71
  - \@fnsymbol ..... 50
  - \@for ..... 87
  - \@ifnextchar ..... 47
  - \@namedef ..... 18
  - \@ne ..... 52
  - \@roman ..... 50
  - \@tfor ..... 70
  - \empty ..... 13
  - \advance ..... 200
  - \aftergroup ..... 23
  - \Alph ..... 50
  - \begingroup ..... 23
  - \box ..... 105

<code>\catcode</code>	40	<code>\thr@@</code>	52
<code>\closein</code>	96	<code>\tw@</code>	52
<code>\copy</code>	105	<code>\uccode</code>	42
<code>\csname</code>	10	<code>\unexpanded</code>	13, 28
<code>\def</code>	18	<code>\unhbox</code>	107
<code>\detokenize</code>	68	<code>\unhcopy</code>	107
<code>\dimexpr</code>	63	<code>\unvbox</code>	108
<code>\dp</code>	105	<code>\unvcopy</code>	108
<code>\edef</code>	18	<code>\uppercase</code>	70
<code>\empty</code>	71	<code>\vskip</code>	60
<code>\endcsname</code>	10	<code>\vsplit</code>	107
<code>\endgroup</code>	23	<code>\wd</code>	105
<code>\expandafter</code>	13, 157	<code>\write</code>	96
<code>\futurelet</code>	47	<code>\xdef</code>	18
<code>\gdef</code>	18	<code>\z@</code>	52
<code>\global</code>	21	<code>\tex_above:D</code>	181
<code>\glueexpr</code>	61	<code>\tex_abovedisplaysshortskip:D</code>	194
<code>\hskip</code>	60	<code>\tex_abovedisplayskip:D</code>	195
<code>\ht</code>	105	<code>\tex_abovewithdelims:D</code>	182
<code>\ifcase</code>	55, 58	<code>\tex_accent:D</code>	232
<code>\ifdim</code>	63	<code>\tex_adjdemerits:D</code>	269
<code>\ifeof</code>	97	<code>\tex_advance:D</code>	76, 2693, 3282, 3289, 3376, 3381, 3428, 3430
<code>\ifhbox</code>	104	<code>\tex_afterassignment:D</code>	86, 2565, 2676, 5828
<code>\ifnum</code>	55, 58	<code>\tex_aftergroup:D</code>	87, 789
<code>\ifodd</code>	58	<code>\tex_atop:D</code>	183
<code>\ifvbox</code>	104	<code>\tex_atopwithdelims:D</code>	184
<code>\ifvoid</code>	104	<code>\tex_badness:D</code>	331
<code>\jobname</code>	71	<code>\tex_baselineskip:D</code>	259
<code>\lccode</code>	42	<code>\tex_batchmode:D</code>	152
<code>\let</code>	21	<code>\tex_begingroup:D</code>	90, 594, 696, 787
<code>\long</code>	21	<code>\tex_belowdisplaysshortskip:D</code>	196
<code>\lowercase</code>	70	<code>\tex_belowdisplayskip:D</code>	197
<code>\m@ne</code>	52	<code>\tex_binoppenalty:D</code>	220
<code>\mathcode</code>	42	<code>\tex_botmark:D</code>	167
<code>\meaning</code>	22	<code>\tex_box:D</code>	375, 5320, 5343
<code>\newbox</code>	103	<code>\tex_boxmaxdepth:D</code>	337
<code>\newcount</code>	48	<code>\tex_brokenpenalty:D</code>	294
<code>\newdimen</code>	61	<code>\tex_catcode:D</code>	379, 567–569, 571, 572, 576–578, 580, 581, 586, 587, 590, 591, 595, 713, 986, 1515, 1516, 2174, 2178, 2183, 3897, 3898
<code>\newmuskip</code>	64	<code>\tex_char:D</code>	233
<code>\newread</code>	96	<code>\tex_chardef:D</code>	68, 974, 975, 1070, 1071, 1915, 1917, 2922, 4766, 4817
<code>\newskip</code>	59	<code>\tex_cleaders:D</code>	251
<code>\newtoks</code>	74	<code>\tex_closein:D</code>	127, 4820
<code>\newwrite</code>	94	<code>\tex_closeout:D</code>	122, 4777
<code>\noexpand</code>	13, 28	<code>\tex_clubpenalty:D</code>	262
<code>\number</code>	51, 57	<code>\tex_copy:D</code>	319, 5345
<code>\numexpr</code>	57	<code>\tex_count:D</code>	370
<code>\or</code>	55, 58	<code>\tex_countdef:D</code>	69, 1067, 2386, 2696, 2697, 2904
<code>\outer</code>	21	<code>\tex_cr:D</code>	94
<code>\protected</code>	21		
<code>\relax</code>	22		
<code>\romannumeral</code>	51		
<code>\sfcode</code>	42		
<code>\showbox</code>	105		
<code>\string</code>	22		



<code>\tex_crcr:D</code> .....	95	<code>\tex_fam:D</code> .....	80
<code>\tex_csname:D</code> .....	157, 703, 718, 779	<code>\tex_fi:D</code> .....	119, 557, 558, 573, 582, 614, 631, 672, 681, 721, 722, 759
<code>\tex_day:D</code> .....	365	<code>\tex_finalhyphendemerits:D</code> .....	268
<code>\tex_deadcycles:D</code> .....	299	<code>\tex_firstmark:D</code> .....	166
<code>\tex_def:D</code> ....	64, 559, 565, 575, 584, 585, 589, 593, 598–600, 602, 604– 610, 613, 617, 619–622, 626, 633, 635–637, 641, 643–645, 651, 655, 675, 676, 683, 693, 695, 699, 704, 710, 715, 716, 725, 782, 784, 785, 793	<code>\tex_floatingpenalty:D</code> .....	313
<code>\tex_defaulthyphenchar:D</code> .....	349	<code>\tex_font:D</code> .....	79
<code>\tex_defaultskewchar:D</code> .....	350	<code>\tex_fontdimen:D</code> .....	346
<code>\tex_delcode:D</code> .....	380	<code>\tex_fontname:D</code> .....	170
<code>\tex_delimiter:D</code> .....	174	<code>\tex_futurelet:D</code> ....	75, 723, 2526, 2528
<code>\tex_delimiterfactor:D</code> .....	223	<code>\tex_gdef:D</code> .....	66, 711, 712, 714, 813
<code>\tex_delimitershortfall:D</code> .....	222	<code>\tex_global:D</code> ....	81, 708, 709, 790, 1917
<code>\tex_dimen:D</code> .....	371	<code>\tex_globaldefs:D</code> .....	85
<code>\tex_dimendef:D</code> ....	70, 2408, 3357, 3358	<code>\tex_halign:D</code> .....	92
<code>\tex_discretionary:D</code> .....	234	<code>\tex_hangafter:D</code> .....	270
<code>\tex_displayindent:D</code> .....	199	<code>\tex_hangindent:D</code> .....	271
<code>\tex_displaylimits:D</code> .....	209	<code>\tex_hbadness:D</code> .....	332
<code>\tex_displaystyle:D</code> .....	187	<code>\tex_hbox:D</code> .....	327, 5380, 5381, 5386, 5392, 5401, 5402
<code>\tex_displaywidowpenalty:D</code> .....	198	<code>\tex_hfil:D</code> .....	235
<code>\tex_displaywidth:D</code> .....	200	<code>\tex_hfill:D</code> .....	237
<code>\tex_divide:D</code> .....	77	<code>\tex_hfilneg:D</code> .....	236
<code>\tex_doublehyphendemerits:D</code> .....	267	<code>\tex_hfuzz:D</code> .....	334
<code>\tex_dp:D</code> .....	378, 5338	<code>\tex_hoffset:D</code> .....	309
<code>\tex_dump:D</code> .....	361	<code>\tex_holdinginserts:D</code> .....	312
<code>\tex_edef:D</code> ...	65, 562, 659, 661, 666, 794	<code>\tex_hrule:D</code> .....	248
<code>\tex_else:D</code> .....	118, 561, 612, 629, 670, 679, 694, 758	<code>\tex_hsize:D</code> .....	273
<code>\tex_emergencystretch:D</code> .....	282	<code>\tex_hskip:D</code> .....	238, 3309
<code>\tex_end:D</code> .....	156, 686, 5117, 5211	<code>\tex_hss:D</code> .....	239
<code>\tex_endcsname:D</code> .....	158, 703, 718, 780	<code>\tex_ht:D</code> .....	377, 5337
<code>\tex_endgroup:D</code> ...	91, 603, 618, 700, 788	<code>\tex_hyphen:D</code> .....	62, 690
<code>\tex_endinput:D</code> .....	130	<code>\tex_hyphenation:D</code> .....	363
<code>\tex_endlinechar:D</code> .....	172, 570, 579, 3906, 3922, 3932, 3940	<code>\tex_hyphenchar:D</code> .....	347
<code>\tex_eqno:D</code> .....	192	<code>\tex_hyphenpenalty:D</code> .....	265
<code>\tex_errhelp:D</code> .....	138, 4985	<code>\tex_if:D</code> .....	101, 668, 761, 764
<code>\tex_errmessage:D</code> ..	132, 739, 1080, 4987	<code>\tex_ifcase:D</code> ....	102, 1273, 3036, 3089
<code>\tex_errorcontextlines:D</code> .....	139, 4848	<code>\tex_ifcat:D</code> .....	103, 765
<code>\tex_errorstopmode:D</code> .....	153	<code>\tex_ifdim:D</code> .....	106, 3398
<code>\tex_escapechar:D</code> .....	171	<code>\tex_ifeof:D</code> .....	107, 4825, 4827
<code>\tex_everycr:D</code> .....	100	<code>\tex_iffalse:D</code> .....	112, 756
<code>\tex_everydisplay:D</code> .....	201, 688	<code>\tex_ifhbox:D</code> .....	108, 5296, 5300
<code>\tex_everyhbox:D</code> .....	340	<code>\tex_ifhmode:D</code> .....	114, 768
<code>\tex_everyjob:D</code> .....	369	<code>\tex_ifinner:D</code> .....	117, 770
<code>\tex_everymath:D</code> .....	225, 687	<code>\tex_ifmmode:D</code> .....	115, 767
<code>\tex_everypar:D</code> .....	288	<code>\tex_ifnum:D</code> .....	104, 1045, 1051, 3035, 3087, 3709
<code>\tex_everyvbox:D</code> .....	341	<code>\tex_ifodd:D</code> .....	105, 560, 564, 677, 762, 763, 3088
<code>\tex_exhyphenpenalty:D</code> .....	264	<code>\tex_iftrue:D</code> .....	113, 755
<code>\tex_expandafter:D</code> ....	88, 628, 630, 664, 667, 671, 702, 706, 717, 773, 1443	<code>\tex_ifvbox:D</code> .....	109, 5297, 5303
		<code>\tex_ifvmode:D</code> .....	116, 769
		<code>\tex_ifvoid:D</code> .....	110, 5298, 5314
		<code>\tex_ifx:D</code> ...	111, 601, 627, 692, 717, 766
		<code>\tex_ignorespaces:D</code> .....	159



<code>\tex_immediate:D</code> . . . . .	121, 638, 646, 701, 1073, 1076, 4774, 4777, 4782	<code>\tex_message:D</code> . . . . .	133, 742
<code>\tex_indent:D</code> . . . . .	255	<code>\tex_mkern:D</code> . . . . .	180
<code>\tex_input:D</code> . . . . .	129, 684, 720, 6264, 6268	<code>\tex_month:D</code> . . . . .	366
<code>\tex_inputlineno:D</code> . . . . .	131, 1097, 1624, 4897	<code>\tex_moveleft:D</code> . . . . .	316, 5329
<code>\tex_insert:D</code> . . . . .	311	<code>\tex_moveright:D</code> . . . . .	317, 5330
<code>\tex_insertpenalties:D</code> . . . . .	314	<code>\tex_mskip:D</code> . . . . .	177
<code>\tex_interlinepenalty:D</code> . . . . .	293	<code>\tex_multiply:D</code> . . . . .	78
<code>\tex_italiccor:D</code> . . . . .	61, 689	<code>\tex_muskip:D</code> . . . . .	374
<code>\tex_jobname:D</code> . . . . .	368, 3614	<code>\tex_muskipdef:D</code> . . . . .	72, 3417, 3418
<code>\tex_kern:D</code> . . . . .	246	<code>\tex_newlinechar:D</code> . . . . .	128, 4795
<code>\tex_language:D</code> . . . . .	163	<code>\tex_noalign:D</code> . . . . .	96
<code>\tex_lastbox:D</code> . . . . .	320, 5324	<code>\tex_noboundary:D</code> . . . . .	231
<code>\tex_lastkern:D</code> . . . . .	253	<code>\tex_noexpand:D</code> . . . . .	89, 573, 774, 1444
<code>\tex_lastpenalty:D</code> . . . . .	359	<code>\tex_noindent:D</code> . . . . .	257
<code>\tex_lastskip:D</code> . . . . .	254	<code>\tex_nolimits:D</code> . . . . .	211
<code>\tex_lccode:D</code> . . . . .	382, 985, 2236, 2240, 2243, 3895, 3896	<code>\tex_nonscript:D</code> . . . . .	191
<code>\tex_leaders:D</code> . . . . .	250	<code>\tex_nonstopmode:D</code> . . . . .	154
<code>\tex_left:D</code> . . . . .	218	<code>\tex_nulldelimiterspace:D</code> . . . . .	224
<code>\tex_lefthyphenmin:D</code> . . . . .	274	<code>\tex_nullfont:D</code> . . . . .	342
<code>\tex_leftskip:D</code> . . . . .	276	<code>\tex_number:D</code> . . . . .	351, 1126, 1885, 1888, 1890, 2692, 3052, 3084, 5998, 5999
<code>\tex_leqno:D</code> . . . . .	193	<code>\tex_omit:D</code> . . . . .	97
<code>\tex_let:D</code> . . . . .	40, 44, 49, 57, 63, 684–691, 707–709, 754	<code>\tex_openin:D</code> . . . . .	123, 4823
<code>\tex_limits:D</code> . . . . .	210	<code>\tex_openout:D</code> . . . . .	124, 4774
<code>\tex_linepenalty:D</code> . . . . .	266	<code>\tex_or:D</code> . . . . .	120, 757
<code>\tex_lineskip:D</code> . . . . .	260	<code>\tex_outer:D</code> . . . . .	83
<code>\tex_lineskiplimit:D</code> . . . . .	261	<code>\tex_output:D</code> . . . . .	298
<code>\tex_long:D</code> . . . . .	82, 791	<code>\tex_outputpenalty:D</code> . . . . .	308
<code>\tex_looseness:D</code> . . . . .	278	<code>\tex_over:D</code> . . . . .	185
<code>\tex_lower:D</code> . . . . .	315, 5332	<code>\tex_overfullrule:D</code> . . . . .	336
<code>\tex_lowercase:D</code> . . . . .	354, 987, 3820	<code>\tex_overline:D</code> . . . . .	216
<code>\tex_mag:D</code> . . . . .	162	<code>\tex_overwithdelims:D</code> . . . . .	186
<code>\tex_mark:D</code> . . . . .	164	<code>\tex_pagedepth:D</code> . . . . .	300
<code>\tex_mathaccent:D</code> . . . . .	175	<code>\tex_pagefilllstretch:D</code> . . . . .	304
<code>\tex_mathbin:D</code> . . . . .	205	<code>\tex_pagefillstretch:D</code> . . . . .	303
<code>\tex_mathchar:D</code> . . . . .	176	<code>\tex_pagefilstretch:D</code> . . . . .	302
<code>\tex_mathchardef:D</code> . . . . .	73, 2898, 3076, 5286, 5287	<code>\tex_pagegoal:D</code> . . . . .	306
<code>\tex_mathchoice:D</code> . . . . .	173	<code>\tex_pageshrink:D</code> . . . . .	305
<code>\tex_mathclose:D</code> . . . . .	206	<code>\tex_pagestretch:D</code> . . . . .	301
<code>\tex_mathcode:D</code> 384, 2220, 2224, 2228, 2232		<code>\tex_pagetotal:D</code> . . . . .	307
<code>\tex_mathinner:D</code> . . . . .	207	<code>\tex_par:D</code> . . . . .	256, 691
<code>\tex_mathop:D</code> . . . . .	208	<code>\tex_parfillskip:D</code> . . . . .	287
<code>\tex_mathopen:D</code> . . . . .	212	<code>\tex_parindent:D</code> . . . . .	280
<code>\tex_mathord:D</code> . . . . .	213	<code>\tex_parshape:D</code> . . . . .	272
<code>\tex_mathpunct:D</code> . . . . .	214	<code>\tex_parskip:D</code> . . . . .	279
<code>\tex_mathrel:D</code> . . . . .	215	<code>\tex_patterns:D</code> . . . . .	362
<code>\tex_mathsurround:D</code> . . . . .	226	<code>\tex_pausing:D</code> . . . . .	149
<code>\tex_maxdeadcycles:D</code> . . . . .	296	<code>\tex_penalty:D</code> . . . . .	357
<code>\tex_maxdepth:D</code> . . . . .	297	<code>\tex_postdisplaypenalty:D</code> . . . . .	204
<code>\tex_meaning:D</code> . . . . .	356, 777, 781	<code>\tex_predisplaypenalty:D</code> . . . . .	203
<code>\tex_medmuskip:D</code> . . . . .	227	<code>\tex_predisplaysize:D</code> . . . . .	202
		<code>\tex_pretolerance:D</code> . . . . .	283
		<code>\tex_prevdepth:D</code> . . . . .	330
		<code>\tex_prevgraf:D</code> . . . . .	289

<code>\tex_radical:D</code> .....	178	<code>\tex_thinmuskip:D</code> .....	228
<code>\tex_raise:D</code> .....	318, 5331	<code>\tex_time:D</code> .....	364
<code>\tex_read:D</code> .....	125, 4830	<code>\tex_toks:D</code> .....	373
<code>\tex_relax:D</code> .....	160, 560, 564, 567–572, 576–581, 586, 587, 590, 591, 595, 623, 624, 626, 633, 634, 641, 642, 677, 718, 720, 786, 1013, 1030, 1273, 1307, 1503, 1515, 1516, 3086, 3456	<code>\tex_toksdef:D</code> .	74, 2419, 4069, 4070, 4201
<code>\tex_relpemalty:D</code> .....	221	<code>\tex_tolerance:D</code> .....	284
<code>\tex_right:D</code> .....	219	<code>\tex_topmark:D</code> .....	165
<code>\tex_righthyphenmin:D</code> .....	275	<code>\tex_topskip:D</code> .....	295
<code>\tex_rightskip:D</code> .....	277	<code>\tex_tracingcommands:D</code> .....	140
<code>\tex_romannumeral:D</code> .	352, 868, 869, 991, 1469, 1488, 1496, 1587, 1590, 1593, 1648, 1655, 1660, 1720, 2691, 3098	<code>\tex_tracinglostchars:D</code> .....	141
<code>\tex_scriptfont:D</code> .....	344	<code>\tex_tracingmacros:D</code> .....	142
<code>\tex_scriptscriptfont:D</code> .....	345	<code>\tex_tracingonline:D</code> .....	143
<code>\tex_scriptscriptstyle:D</code> .....	190	<code>\tex_tracingoutput:D</code> .....	144
<code>\tex_scriptspace:D</code> .....	230	<code>\tex_tracingpages:D</code> .....	145
<code>\tex_scriptstyle:D</code> .....	189	<code>\tex_tracingparagraphs:D</code> .....	146
<code>\tex_scrollmode:D</code> .....	155	<code>\tex_tracingrestores:D</code> .....	147
<code>\tex_setbox:D</code> .	326, 5320, 5325, 5355, 5360, 5366, 5374, 5381, 5386, 5392	<code>\tex_tracingstats:D</code> .....	148
<code>\tex_setlanguage:D</code> .....	84	<code>\tex_uccode:D</code> ....	383, 2246, 2250, 2253
<code>\tex_sfcode:D</code> ....	381, 2256, 2260, 2264	<code>\tex_uchyph:D</code> .....	281
<code>\tex_shipout:D</code> .....	291, 744	<code>\tex_underline:D</code> .....	217, 685
<code>\tex_show:D</code> .....	134, 783	<code>\tex_unhbox:D</code> .....	322, 5405
<code>\tex_showbox:D</code> .....	136, 5347	<code>\tex_unhcopy:D</code> .....	323, 5403
<code>\tex_showboxbreadth:D</code> .....	150	<code>\tex_unkern:D</code> .....	247
<code>\tex_showboxdepth:D</code> .....	151	<code>\tex_unpenalty:D</code> .....	358
<code>\tex_showlists:D</code> .....	137	<code>\tex_unskip:D</code> .....	245
<code>\tex_showthe:D</code> .	135, 2183, 2232, 2243, 2253, 2264, 2783, 3317, 3387, 4136	<code>\tex_unvbox:D</code> .....	324, 5378
<code>\tex_skewchar:D</code> .....	348	<code>\tex_unvcopy:D</code> .....	325, 5376
<code>\tex_skip:D</code> .....	372	<code>\tex_uppercase:D</code> .....	355, 3821
<code>\tex_skipdef:D</code> .	71, 2397, 3241, 3242, 3321	<code>\tex_vadjust:D</code> .....	258
<code>\tex_space:D</code> .....	60	<code>\tex_valign:D</code> .....	93
<code>\tex_spacefactor:D</code> .....	290	<code>\tex_vbadness:D</code> .....	333
<code>\tex_spaceskip:D</code> .....	285	<code>\tex_vbox:D</code> .....	328, 5354, 5355, 5360, 5366, 5371, 5372
<code>\tex_span:D</code> .....	98	<code>\tex_vcenter:D</code> .....	179
<code>\tex_special:D</code> .....	360	<code>\tex_vfil:D</code> .....	240
<code>\tex_splitbotmark:D</code> .....	169	<code>\tex_vfill:D</code> .....	242
<code>\tex_splitfirstmark:D</code> .....	168	<code>\tex_vfilneg:D</code> .....	241
<code>\tex_splitmaxdepth:D</code> .....	338	<code>\tex_vfuzz:D</code> .....	335
<code>\tex_splittopskip:D</code> .....	339	<code>\tex_voffset:D</code> .....	310
<code>\tex_string:D</code> .....	353, 778	<code>\tex_vrule:D</code> .....	249
<code>\tex_tabskip:D</code> .....	99	<code>\tex_vsize:D</code> .....	292
<code>\tex_textfont:D</code> .....	343	<code>\tex_vskip:D</code> .....	243, 3312
<code>\tex_textstyle:D</code> .....	188	<code>\tex_vsplitt:D</code> .....	321, 5374
<code>\tex_the:D</code> .....	161, 567–572, 1097, 1307, 1509, 1517, 1624, 2781, 3315, 3385, 3432, 4074	<code>\tex_vss:D</code> .....	244
<code>\tex_thickmuskip:D</code> .....	229	<code>\tex_vtop:D</code> .....	329
		<code>\tex_wd:D</code> .....	376, 5339
		<code>\tex_widowpenalty:D</code> .....	263
		<code>\tex_write:D</code> .	126, 638, 646, 701, 776, 4803
		<code>\tex_xdef:D</code> .....	67, 702, 814
		<code>\tex_xleaders:D</code> .....	252
		<code>\tex_xspaceskip:D</code> .....	286
		<code>\tex_year:D</code> .....	367
		<code>\textasteriskcentered</code> .....	2872, 2878
		<code>\textbardbl</code> .....	2877
		<code>\textdagger</code> .....	2873, 2879

- \textdaggerdbl ..... 2874, 2880
- \textfont ..... 343
- \textparagraph ..... 2876
- \textsection ..... 2875
- \textstyle ..... 188
- \TeXETstate ..... 443
- \the ..... 26–31, 161
- \thepage ..... 5491
- \thickmuskip ..... 229
- \thinmuskip ..... 228
- \time ..... 364
- \tl\_clear:c ..... 66, 3506, 4225
- \tl\_clear:N ..... 66, 3506, 3506, 3507, 3514,  
3520, 4224, 4248, 4395, 4415, 6216
- \tl\_clear\_new:c ..... 66, 3510, 4229
- \tl\_clear\_new:N ..... 66, 3510, 3511, 3520, 3521, 4228, 4399
- \tl\_compare:nn ..... 3666
- \tl\_compare:no ..... 3678
- \tl\_compare:nV ..... 3675
- \tl\_compare:nx ..... 3669
- \tl\_compare:on ..... 3684
- \tl\_compare:oo ..... 3690
- \tl\_compare:ox ..... 3702
- \tl\_compare:Vn ..... 3681
- \tl\_compare:VV ..... 3687
- \tl\_compare:Vx ..... 3699
- \tl\_compare:xn ..... 3672
- \tl\_compare:xo ..... 3696
- \tl\_compare:xV ..... 3693
- \tl\_compare:xx ..... 3665, 3667,  
3670, 3673, 3676, 3679, 3682, 3685,  
3688, 3691, 3694, 3697, 3700, 3703
- \tl\_elt\_count:N ..... 71, 3881, 3887
- \tl\_elt\_count:n ..... 71, 3881, 3881, 3886
- \tl\_elt\_count:o ..... 71, 3881
- \tl\_elt\_count:V ..... 71, 3881
- \tl\_gclear:c ..... 66, 3506, 4227
- \tl\_gclear:N ..... 66, 3506,  
3508, 3509, 3526, 3531, 4226, 4397
- \tl\_gclear\_new:c ..... 66, 3522, 4231
- \tl\_gclear\_new:N ..... 66, 3522, 3523, 3531, 3532, 4230, 4401
- \tl\_gput\_left:cn ..... 67, 3533
- \tl\_gput\_left:co ..... 67, 3533
- \tl\_gput\_left:cV ..... 67, 3533
- \tl\_gput\_left:Nn ..... 67, 3533, 3550, 3568, 4485
- \tl\_gput\_left:No ..... 67, 3533, 3558
- \tl\_gput\_left:NV ..... 67, 3533, 3555, 3569
- \tl\_gput\_left:Nx ..... 67, 3533, 3563
- \tl\_gput\_right:cn ..... 67, 3570
- \tl\_gput\_right:co ..... 67, 3570
- \tl\_gput\_right:cV ..... 67, 3570
- \tl\_gput\_right:Nn ..... 67, 3570, 3575, 3606, 4493
- \tl\_gput\_right:No ..... 67, 3570, 3591
- \tl\_gput\_right:NV ..... 67, 3570, 3588
- \tl\_gput\_right:Nx ..... 67, 3570, 3600
- \tl\_gremove\_all\_in:cn ..... 73, 4013
- \tl\_gremove\_all\_in:Nn ..... 73, 4013, 4016, 4020
- \tl\_gremove\_in:cn ..... 72, 4009
- \tl\_gremove\_in:Nn ..... 72, 4009, 4010, 4012
- \tl\_greplace\_all\_in:cn ..... 72, 3987
- \tl\_greplace\_all\_in:Nnn ..... 72, 3987, 4006, 4008, 4017
- \tl\_greplace\_in:cn ..... 72, 3968
- \tl\_greplace\_in:Nnn ..... 72, 3968, 3985, 3986, 4010
- \tl\_gset:cn ..... 66, 3467
- \tl\_gset:cx ..... 66, 3467
- \tl\_gset:Nc ..... 66, 3607, 3607
- \tl\_gset:Nn ..... 66, 3467, 3475, 3485,  
3489, 3893, 4378, 4485, 4493, 4608
- \tl\_gset:No ..... 66, 3467
- \tl\_gset:NV ..... 66, 3467
- \tl\_gset:Nv ..... 66, 3467
- \tl\_gset:Nx ..... 66, 3467, 3478, 3486, 3552, 3556,  
3560, 3564, 3577, 3589, 3593, 3601,  
3934, 3985, 4007, 4241, 4561, 4669
- \tl\_gset\_eq:cc ..... 67, 3490
- \tl\_gset\_eq:cN ..... 67, 3490
- \tl\_gset\_eq:Nc ..... 67, 3490
- \tl\_gset\_eq:NN ..... 67, 3059, 3490, 3495, 3502, 3505, 3508
- \tl\_gset\_rescan:Nnn ..... 68, 3892, 3893
- \tl\_gset\_rescan:Nnx ..... 68, 3919, 3929
- \tl\_head:n ..... 73, 4021, 4021, 4022
- \tl\_head:w ..... 73, 4021, 4021,  
4027, 4028, 4032, 4037, 4041, 4055
- \tl\_head\_i:n ..... 73, 4021, 4022
- \tl\_head\_i:w ..... 73, 4021, 4028
- \tl\_head\_iii:f ..... 73, 4021
- \tl\_head\_iii:n ..... 73, 4021, 4025, 4026
- \tl\_head\_iii:w ..... 73, 4021, 4025, 4030
- \tl\_if\_blank:n ..... 3651
- \tl\_if\_blank:nF ..... 3660, 3664, 4504
- \tl\_if\_blank:nT ..... 3659, 3663
- \tl\_if\_blank:nTF ..... 70, 3651, 3658, 3662
- \tl\_if\_blank:oTF ..... 70, 3651, 5603
- \tl\_if\_blank:VTF ..... 70, 3651
- \tl\_if\_blank\_p:n ..... 70, 3651, 3657, 3661
- \tl\_if\_blank\_p:o ..... 70, 3651
- \tl\_if\_blank\_p:V ..... 70, 3651
- \tl\_if\_blank\_p\_aux:w ..... 3651, 3652, 3654
- \tl\_if\_empty:c ..... 4245, 4412
- \tl\_if\_empty:cTF ..... 68, 3624
- \tl\_if\_empty:N ..... 3624, 4244, 4411
- \tl\_if\_empty:n ..... 3639

\tl_if_empty:NF	3631, 6245, 6251	\tl_if_eq:xVTF	69, 3665
\tl_if_empty:nF	2604,	\tl_if_eq:xx	3812
3646, 3650, 4477, 4528, 4539, 4693		\tl_if_eq:xxTF	69, 1968, 3665, 4194
\tl_if_empty:NT	3630	\tl_if_eq_p:cc	68, 3632
\tl_if_empty:nT	3645, 3649, 4715	\tl_if_eq_p:cN	68, 3632
\tl_if_empty:NTF	68, 3624, 3629, 5641, 5663	\tl_if_eq_p:Nc	68, 3632
\tl_if_empty:nTF	69, 2376, 2383, 2394, 2405,	\tl_if_eq_p:NN	68, 3632, 3635
2416, 2427, 2434, 2441, 2449, 3639,		\tl_if_eq_p:nn	69, 3665
3644, 3648, 4948, 4957, 4966, 5000		\tl_if_eq_p:no	69, 3665
\tl_if_empty:oTF	69, 3639	\tl_if_eq_p:nV	69, 3665
\tl_if_empty:VTF	69, 3639, 4187	\tl_if_eq_p:nx	69, 3665
\tl_if_empty_p:c	68, 3624	\tl_if_eq_p:on	69, 3665
\tl_if_empty_p:N	68, 3624, 3628	\tl_if_eq_p:oo	69, 3665
\tl_if_empty_p:n	69, 2346, 3639, 3643, 3647	\tl_if_eq_p:ox	69, 3665
\tl_if_empty_p:o	69, 3639	\tl_if_eq_p:Vn	69, 3665
\tl_if_empty_p:V	69, 3639	\tl_if_eq_p:VV	69, 3665
\tl_if_eq:cc	4422	\tl_if_eq_p:Vx	69, 3665
\tl_if_eq:ccTF	69, 3632	\tl_if_eq_p:xn	69, 3665
\tl_if_eq:cN	4420	\tl_if_eq_p:xo	69, 3665
\tl_if_eq:cNTF	69, 3632	\tl_if_eq_p:xV	69, 3665
\tl_if_eq:Nc	4421	\tl_if_eq_p:xx	69, 3665
\tl_if_eq:NcTF	69, 3632	\tl_if_head_eq_catcode:nN	4053
\tl_if_eq:NN	3632, 4419	\tl_if_head_eq_catcode:nNTF	74, 4031
\tl_if_eq:nn	3721	\tl_if_head_eq_catcode_p:nN	74, 4031
\tl_if_eq:NNF	3638	\tl_if_head_eq_charcode:fNTF	73, 2945, 2952, 4031
\tl_if_eq:nnF	4590	\tl_if_head_eq_charcode:nN	4035
\tl_if_eq:NNT	3637	\tl_if_head_eq_charcode:nNF	4052
\tl_if_eq:NNTF	69, 1978, 3632, 3636, 5102, 5617, 5668	\tl_if_head_eq_charcode:nNT	4051
\tl_if_eq:nnTF	69, 3665	\tl_if_head_eq_charcode:nNTF	73, 4031, 4050
\tl_if_eq:no	3735	\tl_if_head_eq_charcode_p:fN	73, 4031, 4040, 4048
\tl_if_eq:noTF	69, 3665	\tl_if_head_eq_charcode_p:nN	73, 4031, 4049
\tl_if_eq:nV	3728	\tl_if_head_eq_meaning:nN	4031
\tl_if_eq:nVTF	69, 3665	\tl_if_head_eq_meaning:nNTF	73, 4031, 5632
\tl_if_eq:nx	3742	\tl_if_head_eq_meaning_p:nN	73, 4031
\tl_if_eq:nxTF	69, 3665	\tl_if_in:cnTF	72, 3946
\tl_if_eq:on	3756	\tl_if_in:Nn	3946
\tl_if_eq:onTF	69, 3665	\tl_if_in:nn	3955
\tl_if_eq:oo	3770	\tl_if_in:NnF	3954
\tl_if_eq:ooTF	69, 3665	\tl_if_in:nnF	3963, 3966
\tl_if_eq:ox	3784	\tl_if_in:NnT	3953
\tl_if_eq:oxTF	69, 3665	\tl_if_in:nnT	3962, 3965
\tl_if_eq:Vn	3749	\tl_if_in:NnTF	72, 3946, 3952
\tl_if_eq:VnTF	69, 3665	\tl_if_in:nnTF	72, 3955, 3961, 3964
\tl_if_eq:VV	3763	\tl_if_in:onTF	72, 3955
\tl_if_eq:VVTf	69, 3665	\tl_if_in:VnTF	72, 3955
\tl_if_eq:Vx	3777	\tl_map_break:	71, 3869, 3869
\tl_if_eq:VxTF	69, 3665	\tl_map_function:cN	70, 3827
\tl_if_eq:xn	3791	\tl_map_function:NN	70, 3827, 3830, 3838, 3889
\tl_if_eq:xnTF	69, 3665		
\tl_if_eq:xo	3805		
\tl_if_eq:xoTF	69, 3665		
\tl_if_eq:xV	3798		

- \tl\_map\_function:nN ..... 70, [3827](#), [3827](#), [3883](#), [5945](#)
- \tl\_map\_function\_aux:NN ..... [3827](#)
- \tl\_map\_function\_aux:Nn ..... [3828](#), [3831](#), [3834](#), [3836](#), [3843](#), [3852](#)
- \tl\_map\_inline:cn ..... 70, [3839](#)
- \tl\_map\_inline:Nn .. 70, [3839](#), [3848](#), [3857](#)
- \tl\_map\_inline:nn .. 70, [2367](#), [3839](#), [3839](#)
- \tl\_map\_inline\_aux:n ..... [3839](#)
- \tl\_map\_variable:cNn ..... 70, [3859](#)
- \tl\_map\_variable:NNn 70, [3859](#), [3862](#), [3863](#)
- \tl\_map\_variable:nNn 70, [3859](#), [3859](#), [3862](#)
- \tl\_map\_variable\_aux:NnN ..... [3864](#)
- \tl\_map\_variable\_aux:Nnn 3860, [3864](#), [3867](#)
- \tl\_new:c ..... 65, [3440](#), [3454](#), [4223](#)
- \tl\_new:cn ..... 65, [3440](#), [3454](#), [5426](#)
- \tl\_new:N ..... 65, [3440](#), [3453](#), [4222](#), [4393](#), [4890](#), [4891](#), [5530](#), [5531](#), [6180](#)
- \tl\_new:Nn ..... 65, [2071](#), [2530](#), [2531](#), [2533](#), [2563](#), [2564](#), [3049](#), [3440](#), [3440](#), [3447](#), [3453](#), [3516](#), [3528](#), [3614](#)–[3623](#), [3858](#), [3901](#), [4849](#)–[4853](#), [4856](#), [4859](#), [4862](#), [4868](#), [4872](#), [4882](#), [4886](#)–[4889](#), [5532](#), [5699](#)
- \tl\_new:Nx .... 65, [3440](#), [3448](#), [4812](#)–[4814](#)
- \tl\_put\_left:cn ..... 66, [3533](#)
- \tl\_put\_left:co ..... 66, [3533](#)
- \tl\_put\_left:cV ..... 66, [3533](#)
- \tl\_put\_left:Nn 66, [3533](#), [3533](#), [3567](#), [4481](#)
- \tl\_put\_left:No ..... 66, [3533](#), [3541](#)
- \tl\_put\_left:NV ..... 66, [3533](#), [3538](#)
- \tl\_put\_left:Nx ..... 66, [3533](#), [3546](#)
- \tl\_put\_right:cn ..... 67, [3570](#)
- \tl\_put\_right:co ..... 67, [3570](#)
- \tl\_put\_right:cV ..... 67, [3570](#)
- \tl\_put\_right:Nn 67, [3570](#), [3570](#), [3604](#), [4489](#)
- \tl\_put\_right:No ..... 67, [3570](#), [3583](#)
- \tl\_put\_right:NV ... 67, [3570](#), [3580](#), [3605](#)
- \tl\_put\_right:Nx ..... 67, [3570](#), [3596](#)
- \tl\_remove\_all\_in:cn ..... 73, [4013](#)
- \tl\_remove\_all\_in:Nn 73, [4013](#), [4013](#), [4019](#)
- \tl\_remove\_in:cn ..... 72, [4009](#)
- \tl\_remove\_in:Nn ... 72, [4009](#), [4009](#), [4011](#)
- \tl\_replace\_all\_in:cnn ..... 72, [3987](#)
- \tl\_replace\_all\_in:Nnn ..... 72, [3987](#), [4003](#), [4005](#), [4014](#), [5545](#), [5548](#)
- \tl\_replace\_all\_in\_aux:NNnn ..... [3987](#), [3987](#), [4004](#), [4007](#)
- \tl\_replace\_in:cnn ..... 72, [3968](#)
- \tl\_replace\_in:Nnn ..... 72, [3968](#), [3983](#), [3984](#), [4009](#)
- \tl\_replace\_in\_aux:NNnn ..... [3968](#), [3968](#), [3983](#), [3985](#)
- \tl\_rescan:nn ..... 68, [3937](#), [3937](#)
- \tl\_rescan\_aux:w . 3908, [3913](#), [3914](#), [3942](#)
- \tl\_reverse:N ..... 71, [3878](#), [3878](#)
- \tl\_reverse:n ..... 71, [3870](#), [3870](#), [3877](#)
- \tl\_reverse:o ..... 71, [3870](#), [3879](#)
- \tl\_reverse:V ..... 71, [3870](#)
- \tl\_reverse\_aux:nN [3870](#), [3871](#), [3873](#), [3875](#)
- \tl\_set:cn ..... 66, [3467](#)
- \tl\_set:co ..... 66, [3467](#)
- \tl\_set:cV ..... 66, [3467](#)
- \tl\_set:cx ..... 66, [3467](#)
- \tl\_set:Nc ..... 66, [3607](#), [3612](#), [3613](#)
- \tl\_set:Nf ..... 66, [3467](#), [3879](#)
- \tl\_set:Nn ..... 66, [2536](#), [2550](#), [3467](#), [3468](#), [3483](#), [3488](#), [3534](#), [3551](#), [3571](#), [3576](#), [3865](#), [3892](#), [3917](#), [3927](#), [4267](#), [4299](#), [4375](#), [4378](#), [4446](#), [4481](#), [4489](#), [4602](#), [4608](#), [5081](#), [5082](#), [5488](#), [5662](#), [6221](#), [6229](#)
- \tl\_set:No ..... 66, [3052](#), [3467](#), [3542](#), [3559](#), [3584](#), [3592](#), [3613](#)
- \tl\_set:NV ..... 66, [3467](#)
- \tl\_set:Nv ..... 66, [3467](#)
- \tl\_set:Nx 66, [2537](#), [2538](#), [2551](#), [2553](#), [3467](#), [3472](#), [3484](#), [3535](#), [3539](#), [3543](#), [3547](#), [3572](#), [3581](#), [3585](#), [3597](#), [3706](#), [3722](#), [3723](#), [3729](#), [3730](#), [3736](#), [3737](#), [3743](#), [3744](#), [3750](#), [3751](#), [3757](#), [3758](#), [3764](#), [3765](#), [3771](#), [3772](#), [3778](#), [3779](#), [3785](#), [3786](#), [3792](#), [3793](#), [3799](#), [3800](#), [3806](#), [3807](#), [3813](#), [3814](#), [3924](#), [3983](#), [4004](#), [4329](#), [4335](#), [4560](#), [4665](#), [4674](#), [5001](#), [5003](#), [5587](#), [5616](#), [5636](#), [5722](#)
- \tl\_set\_eq:cc ..... 67, [3490](#)
- \tl\_set\_eq:cN ..... 67, [3490](#)
- \tl\_set\_eq:Nc ..... 67, [3490](#)
- \tl\_set\_eq:NN ..... 67, [2552](#), [3057](#), [3490](#), [3491](#), [3501](#), [3504](#), [3506](#)
- \tl\_set\_rescan:Nnn ..... 68, [3892](#), [3892](#)
- \tl\_set\_rescan:Nnx ..... 68, [3919](#), [3919](#)
- \tl\_set\_rescan\_aux:NNnn ..... 3892, [3893](#), [3894](#), [3903](#)
- \tl\_show:c ..... 65, [3464](#), [4280](#), [4459](#)
- \tl\_show:N 65, [3464](#), [3464](#), [3465](#), [4279](#), [4458](#)
- \tl\_show:n ..... 65, [3464](#), [3466](#)
- \tl\_tail:f ..... 73, [4021](#)
- \tl\_tail:n ..... 73, [4021](#), [4023](#), [4024](#)
- \tl\_tail:w ..... 73, [4021](#), [4023](#), [4029](#)
- \tl\_tmp:w ..... 3705, [3716](#)–[3719](#), [3947](#), [3950](#), [3956](#), [3959](#), [3969](#), [3973](#), [3976](#), [3980](#), [3989](#), [3996](#), [3999](#)
- \tl\_to\_lowercase:n ... 70, [2369](#), [2459](#), [3820](#), [3820](#), [3899](#), [4880](#), [4977](#), [5543](#)
- \tl\_to\_str:c ..... 68, [3823](#)
- \tl\_to\_str:N ..... 68, [3823](#), [3823](#), [3826](#)

\tl_to_str:n .....	\token_if_int_register_p_aux:w ..	2360
68, 2129, 2151, 3640, 3652, 3822, 3822	\token_if_letter:N .....	2318
\tl_to_str_aux:w .....	\token_if_letter:NTF .....	44, 2318
3823, 3823, 3825	\token_if_letter_p:N .....	44, 2318
\tl_to_uppercase:n .....	\token_if_long_macro:N .....	2436
70, 3820, 3821	\token_if_long_macro:NTF .....	45, 2360
\tl_use:c .....	\token_if_long_macro_aux:w ..	2437, 2440
65, 3455	\token_if_long_macro_p:N .....	45, 2360
\tl_use:N .....	\token_if_long_macro_p_aux:w .....	2360
65, 3455, 3455, 3463	\token_if_macro:N .....	2342
\token_get_arg_spec:N ...	\token_if_macro:NTF .....	45, 2342, 2465, 2471, 2477
46, 2452, 2470	\token_if_macro_p:N .	45, 2342, 2485, 2492
\token_get_prefix_arg_replacement_aux:w	\token_if_macro_p_aux:w	2342, 2343, 2345
.....	\token_if_math_shift:N .....	2294
2452, 2461, 2466, 2472, 2478	\token_if_math_shift:NTF .....	43, 2294
\token_get_prefix_spec:N .	\token_if_math_shift_p:N .....	43, 2294
46, 2452, 2464	\token_if_math_subscript:N .....	2310
\token_get_replacement_spec:N .....	\token_if_math_subscript:NTF ..	44, 2310
46, 2452, 2476	\token_if_math_subscript_p:N ..	44, 2310
\token_if_active_char:N .....	\token_if_math_superscript:N .....	2306
2326	\token_if_math_superscript:NTF	44, 2306
\token_if_active_char:NTF .....	\token_if_math_superscript_p:N	44, 2306
44, 2326	\token_if_mathchardef:N .....	2378
\token_if_active_char_p:N	\token_if_mathchardef:NTF .....	46, 2360
44, 2326, 2491	\token_if_mathchardef_aux:w .	2379, 2382
\token_if_alignment_tab:N .....	\token_if_mathchardef_p:N	46, 2360, 2505
2298	\token_if_mathchardef_p_aux:w ...	2360
\token_if_alignment_tab:NTF ...	\token_if_other_char:N .....	2322
43, 2298	\token_if_other_char:NTF .....	44, 2322
\token_if_alignment_tab_p:N ...	\token_if_other_char_p:N .....	44, 2322
43, 2298	\token_if_parameter:N .....	2302
\token_if_chardef:N .....	\token_if_parameter:NTF .....	43, 2302
2371	\token_if_parameter_p:N .....	43, 2302
\token_if_chardef:NTF .....	\token_if_primitive:N .....	2483
45, 2360	\token_if_primitive:NTF .....	46, 2483
\token_if_chardef_aux:w .....	\token_if_primitive_p:N .....	46, 2483
2372, 2375	\token_if_primitive_p_aux:N .....	2483, 2488, 2495, 2502
\token_if_chardef_p:N ...	\token_if_protected_long_macro:N .	2443
45, 2360, 2503	\token_if_protected_long_macro:NTF .	45, 2360
\token_if_chardef_p_aux:w .....	\token_if_protected_long_macro_aux:w	2444, 2447
2360	\token_if_protected_long_macro_p:N .	45, 2360
\token_if_cs:N .....	\token_if_protected_long_macro_p_aux:w	2360
2349	\token_if_protected_macro:N .....	2429
\token_if_cs:NTF .....	\token_if_protected_macro:NTF .	45, 2360
45, 2349	\token_if_protected_macro_aux:w ....	2430, 2433
\token_if_cs_p:N .....	\token_if_protected_macro_p:N .	45, 2360
45, 2349, 2484	\token_if_protected_macro_p_aux:w	2360
\token_if_dim_register:N .....		
2407		
\token_if_dim_register:NTF ....		
46, 2360		
\token_if_dim_register_aux:w		
2411, 2415		
\token_if_dim_register_p:N		
46, 2360, 2511		
\token_if_dim_register_p_aux:w ..		
2360		
\token_if_eq_catcode:NN .....		
2334		
\token_if_eq_catcode:NNTF .....		
44, 2334		
\token_if_eq_catcode_p:NN .....		
44, 2334, 2350, 2583		
\token_if_eq_charcode:NN .....		
2338		
\token_if_eq_charcode:NNTF ....		
44, 2338		
\token_if_eq_charcode_p:NN ....		
44, 2338		
\token_if_eq_meaning:NN .....		
2330		
\token_if_eq_meaning:NNTF		
44, 2330, 2675		
\token_if_eq_meaning_p:NN		
44, 2330, 2584		
\token_if_expandable:N .....		
2352		
\token_if_expandable:NTF .....		
45, 2352		
\token_if_expandable_p:N .....		
45, 2352		
\token_if_group_begin:N .....		
2286		
\token_if_group_begin:NTF .....		
43, 2286		
\token_if_group_begin_p:N .....		
43, 2286		
\token_if_group_end:N .....		
2290		
\token_if_group_end:NTF .....		
43, 2290		
\token_if_group_end_p:N .....		
43, 2290		
\token_if_int_register:N .....		
2385		
\token_if_int_register:NTF ....		
46, 2360		
\token_if_int_register_aux:w		
2389, 2393		
\token_if_int_register_p:N		
46, 2360, 2507		



- \token\_if\_skip\_register:N ..... 2396
- \token\_if\_skip\_register:NTF ... 46, 2360
- \token\_if\_skip\_register\_aux:w 2400, 2404
- \token\_if\_skip\_register\_p:N ..... 46, 2360, 2509
- \token\_if\_skip\_register\_p\_aux:w .. 2360
- \token\_if\_space:N ..... 2314
- \token\_if\_space:NTF ..... 44, 2314
- \token\_if\_space\_p:N ..... 44, 2314
- \token\_if\_toks\_register:N ..... 2418
- \token\_if\_toks\_register:NTF ... 46, 2360
- \token\_if\_toks\_register\_aux:w 2422, 2426
- \token\_if\_toks\_register\_p:N ..... 46, 2360, 2513
- \token\_if\_toks\_register\_p\_aux:w .. 2360
- \token\_new:Nn ... 43, 2268, 2268, 2273, 2275–2277, 2279–2282, 2523–2525
- \token\_to\_meaning:N ..... 22, 776, 777, 1090, 2343, 2373, 2380, 2390, 2401, 2412, 2423, 2431, 2438, 2445, 2467, 2473, 2479, 3824
- \token\_to\_str:c ..... 22, 776, 785, 1621
- \token\_to\_str:N ..... 4, 22, 776, 778, 785, 977, 981, 1042, 1088, 1095, 1096, 1103, 1120, 1124, 1306, 2128, 2150, 3457, 4249, 4282, 4416, 4436, 4461, 4640, 6056
- \toks ..... 373
- \toks\_clear:c ..... 76, 4115, 4626
- \toks\_clear:N ..... 76, 3988, 4115, 4115, 4122, 4124, 4127, 4284, 4463, 4625, 4642, 5586, 5597
- \toks\_gclear:c ..... 76, 4115, 4628
- \toks\_gclear:N . 76, 4115, 4119, 4125, 4627
- \toks\_gput\_left:cn ..... 77, 4138
- \toks\_gput\_left:co ..... 77, 4138
- \toks\_gput\_left:cV ..... 77, 4138
- \toks\_gput\_left:Nn .. 77, 4138, 4143, 4148
- \toks\_gput\_left:No ..... 77, 4138
- \toks\_gput\_left:NV ..... 77, 4138
- \toks\_gput\_left:Nx ..... 77, 4138
- \toks\_gput\_right:cn ..... 77, 4153
- \toks\_gput\_right:co ..... 77, 4153
- \toks\_gput\_right:cV ..... 77, 4153
- \toks\_gput\_right:Nn ..... 77, 4153, 4157, 4177, 4688
- \toks\_gput\_right:No ..... 77, 4153
- \toks\_gput\_right:NV ..... 77, 4153
- \toks\_gput\_right:Nx ..... 77, 4153
- \toks\_gset:cn ..... 75, 4094, 5457
- \toks\_gset:co ..... 75, 4094
- \toks\_gset:cV ..... 75, 4094
- \toks\_gset:cx ..... 75, 4094
- \toks\_gset:Nn ..... 75, 4094, 4094–4096, 4672, 4704
- \toks\_gset:No ..... 75, 4094
- \toks\_gset:NV ..... 75, 4094
- \toks\_gset:Nx ..... 75, 4094
- \toks\_gset\_eq:cc ..... 76, 4097, 4636
- \toks\_gset\_eq:cN ..... 76, 4097, 4635
- \toks\_gset\_eq:Nc ..... 76, 4097, 4634
- \toks\_gset\_eq:NN ..... 76, 4097, 4103, 4111, 4114, 4633
- \toks\_if\_empty:c ..... 4717
- \toks\_if\_empty:cTF ..... 77, 4186
- \toks\_if\_empty:N ..... 4186, 4716
- \toks\_if\_empty:NF ..... 4192, 4286, 4465, 4556, 4644
- \toks\_if\_empty:NT ..... 4191
- \toks\_if\_empty:NTF ..... 77, 4186, 4190
- \toks\_if\_empty\_p:c ..... 77, 4186
- \toks\_if\_empty\_p:N ..... 77, 4186, 4189
- \toks\_if\_eq:cc ..... 4721
- \toks\_if\_eq:ccTF ..... 78, 4193
- \toks\_if\_eq:cN ..... 4719
- \toks\_if\_eq:cNTF ..... 78, 4193
- \toks\_if\_eq:Nc ..... 4720
- \toks\_if\_eq:NcTF ..... 78, 4193
- \toks\_if\_eq:NN ..... 4193, 4718
- \toks\_if\_eq:NNF ..... 4200
- \toks\_if\_eq:NNT ..... 4199
- \toks\_if\_eq:NNTF ..... 78, 4193, 4198
- \toks\_if\_eq\_p:cc ..... 78, 4193
- \toks\_if\_eq\_p:cN ..... 78, 4193
- \toks\_if\_eq\_p:Nc ..... 78, 4193
- \toks\_if\_eq\_p:NN ..... 78, 4193, 4197
- \toks\_new:c ..... 74, 4067, 4624
- \toks\_new:N ..... 74, 4067, 4069, 4072, 4073, 4203–4209, 4623, 5533–5535
- \toks\_new\_l:N ..... 4070
- \toks\_put\_left:cn ..... 76, 4138
- \toks\_put\_left:co ..... 76, 4138
- \toks\_put\_left:cV ..... 76, 4138
- \toks\_put\_left:Nn 76, 4138, 4138, 4142, 4146
- \toks\_put\_left:No ..... 76, 4138
- \toks\_put\_left:NV ..... 76, 4138
- \toks\_put\_left:Nx ..... 76, 4138
- \toks\_put\_left\_aux:w ... 4138, 4139, 4149
- \toks\_put\_right:cn ..... 77, 4153
- \toks\_put\_right:co ..... 77, 4153
- \toks\_put\_right:cV ..... 77, 4153
- \toks\_put\_right:Nf ..... 77, 4178, 4180
- \toks\_put\_right:Nn ..... 77, 4153, 4153, 4160, 4162, 4176, 4178, 4682, 5642, 5665, 5670

\toks_put_right:No .....	146
..... 77, 3974, 3992, 3995, 4153,	
4170, 5564, 5619, 5637, 5648, 5653	
\toks_put_right:Nv .....	147
..... 77, 4153, 4164	
\toks_put_right:Nx .....	402
..... 77, 4153,	
4287, 4289, 4466, 4468, 4645, 4647	
\toks_set:cf .....	148
..... 75, 4076	
\toks_set:cn .....	
..... 75, 4076	
\toks_set:co .....	
..... 75, 4076	
\toks_set:cV .....	
..... 75, 4076	
\toks_set:cv .....	
..... 75, 4076	
\toks_set:cx .....	
..... 75, 4076	
\toks_set:Nf ... 75, 4076, 4089, 5561, 5569	
\toks_set:Nn .....	
..... 75, 4076, 4077, 4078, 4081, 4093, 4702	
\toks_set:No .....	
..... 75, 3972, 4076, 4088, 4552, 4553, 5576	
\toks_set:Nv ... 75, 3905, 3939, 4076, 4082	
\toks_set:Nv .....	
..... 75, 4076, 4085	
\toks_set:Nx .....	
..... 75, 4076, 5427	
\toks_set_eq:cc .....	
..... 75, 4097, 4632	
\toks_set_eq:cN .....	
..... 75, 4097, 4631	
\toks_set_eq:Nc .....	
..... 75, 4097, 4630	
\toks_set_eq:NN .....	
..... 75, 4097, 4098, 4110, 4113, 4629	
\toks_show:c .....	
..... 76, 4136, 4638	
\toks_show:N .....	
..... 76, 4136,	
4136, 4137, 4293, 4472, 4637, 4653	
\toks_use:c .....	
..... 75, 4074	
\toks_use:N .....	
..... 75,	
3977, 4001, 4074, 4074, 4075, 4140,	
4154, 4166, 4172, 4182, 4194, 4555,	
4557, 4660, 4733, 4897, 5433, 5439,	
5461, 5465, 5565, 5578, 5621, 5639	
\toks_use_clear:c .....	
..... 76, 4126	
\toks_use_clear:N 76, 4126, 4126, 4132, 4134	
\toks_use_gclear:c .....	
..... 76, 4126	
\toks_use_gclear:N .. 76, 4126, 4129, 4135	
\toksdef .....	
..... 74	
\tolerance .....	
..... 284	
\topmark .....	
..... 165	
\topmarks .....	
..... 391	
\topskip .....	
..... 295	
\totalheightof .....	
..... 111, 6098, 6101	
\tracingassigns .....	
..... 401	
\tracingcommands .....	
..... 140	
\tracinggroups .....	
..... 408	
\tracingifs .....	
..... 404	
\tracinglostchars .....	
..... 141	
\tracingmacros .....	
..... 142	
\tracingnesting .....	
..... 403	
\tracingonline .....	
..... 143	
\tracingoutput .....	
..... 144	
\tracingpages .....	
..... 145	
\tracingparagraphs .....	
..... 146	
\tracingrestores .....	
..... 147	
\tracingscantokens .....	
..... 402	
\tracingstats .....	
..... 148	
U	
\U .....	2367
\uccode .....	383
\uchyph .....	281
\underline .....	217
\unexpanded .....	23, 396
\unhbox .....	322
\unhcopy .....	323
\unkern .....	247
\unless .....	387
\unpenalty .....	358
\unskip .....	245
\unvbox .....	324
\unvcopy .....	325
\uppercase .....	355
\use:c ... 11, 833, 833, 931, 1608, 1612,	
1622, 1878, 1895, 2807, 3066, 3102,	
5049, 5052, 5066, 5113, 5124, 5129,	
5136, 5143, 5150, 5156, 5162, 5196,	
5199, 5207, 5219, 5224, 5231, 6142	
\use:n .....	11, 834,
834, 946, 951, 966, 971, 1307, 1947,	
1957, 1967, 1977, 3065, 3980, 3999	
\use:nn .....	11, 834, 835
\use:nnn .....	11, 834, 836
\use:nnnn .....	11, 834, 837
\use_0_parameter: .....	1192
\use_1_parameter: .....	1192
\use_2_parameter: .....	1192
\use_3_parameter: .....	1192
\use_4_parameter: .....	1192
\use_5_parameter: .....	1192
\use_6_parameter: .....	1192
\use_7_parameter: .....	1192
\use_8_parameter: .....	1192
\use_9_parameter: .....	1192
\use_i:nn .....	11, 838, 838,
941, 961, 995, 1025, 1046, 1258, 4150	
\use_i:nnn .....	11, 840, 840, 1007, 2467
\use_i:nnnn .....	12, 840, 843
\use_i_after_else:nw .....	12, 854, 855
\use_i_after_fi:nw 12, 854, 854, 1294, 2999	
\use_i_after_or:nw .....	12, 854, 856
\use_i_after_orelse:nw .....	12,
854, 857, 1274, 1276, 1278, 1280,	
1282, 1284, 1286, 1288, 1290, 1292	
\use_i_delimit_by_q_nil:nw .....	
..... 12, 851, 851, 4027	



- `\use_i_delimit_by_q_recursion_stop:nw` ..... 12, 851, 853, 2094, 2102  
`\use_i_delimit_by_q_stop:nw` ..... 12, 851, 852, 4341  
`\use_ii:nnn` ..... 12, 840, 847, 1507  
`\use_ii:nn` ..... 11, 838, 839, 941, 961, 979, 997, 1027, 1048, 1260, 5606  
`\use_ii:nnn` . 11, 840, 841, 1010, 2473, 5649  
`\use_ii:nnnn` ..... 12, 840, 844  
`\use_iii:nnn` ..... 11, 840, 842, 2479  
`\use_iii:nnnn` ..... 12, 840, 845  
`\use_iv:nnnn` ..... 12, 840, 846  
`\use_none:n` ..... 11, 858, 858, 981, 1296, 1641, 2096, 2104, 2948, 2949, 2953, 4791, 5572, 5603  
`\use_none:nn` ..... 11, 858, 859, 946, 951, 966, 971, 1630, 2797  
`\use_none:nnn` ..... 11, 858, 860, 5636  
`\use_none:nnnn` ..... 11, 858, 861  
`\use_none:nnnnn` ..... 11, 858, 862  
`\use_none:nnnnnn` ..... 11, 858, 863  
`\use_none:nnnnnnn` ..... 11, 858, 864  
`\use_none:nnnnnnnn` ..... 11, 858, 865  
`\use_none:nnnnnnnnn` . 11, 858, 866, 1264  
`\use_none_delimit_by_q_nil:w` 12, 848, 848  
`\use_none_delimit_by_q_recursion_stop:w` ..... 12, 848, 850, 929, 1419, 1605, 2082, 2087, 3869, 4513  
`\use_none_delimit_by_q_stop:w` ..... 12, 848, 849, 1993, 1997, 4340, 4753  
`\usepackage` ..... 729, 5476
- ### V
- `\vadjust` ..... 258, 546  
`\valign` ..... 93  
`\value` ..... 5494  
`\vbadness` ..... 333  
`\vbox` ..... 328  
`\vbox:n` ..... 107, 5354, 5354  
`\vbox_gset:cn` ..... 107, 5355  
`\vbox_gset:Nn` ..... 107, 5355, 5357, 5358  
`\vbox_gset_inline_begin:N` 107, 5365, 5368  
`\vbox_gset_inline_end:` . 107, 5365, 5370  
`\vbox_gset_to_ht:ccn` ..... 107, 5359  
`\vbox_gset_to_ht:cnn` ..... 107, 5359  
`\vbox_gset_to_ht:Nnn` 107, 5359, 5363, 5364  
`\vbox_set:cn` ..... 107, 5355  
`\vbox_set:Nn` ..... 107, 5355, 5355–5357  
`\vbox_set_inline_begin:N` ..... 107, 5365, 5365, 5369  
`\vbox_set_inline_end:` . 107, 5365, 5367  
`\vbox_set_split_to_ht:NNn` 107, 5373, 5373  
`\vbox_set_to_ht:cnn` ..... 107, 5359  
`\vbox_set_to_ht:Nnn` ..... 107, 5359, 5359, 5362, 5363  
`\vbox_to_ht:nn` ..... 107, 5371, 5371
- `\vbox_to_zero:n` ..... 107, 5371, 5372  
`\vbox_unpack:c` ..... 108, 5376  
`\vbox_unpack:N` .... 108, 5376, 5376, 5377  
`\vbox_unpack_clear:c` ..... 108, 5376  
`\vbox_unpack_clear:N` 108, 5376, 5378, 5379  
`\vcenter` ..... 179  
`\vfil` ..... 240  
`\vfill` ..... 242  
`\vfilneg` ..... 241  
`\vfuzz` ..... 335  
`\voffset` ..... 310  
`\voidb@x` ..... 5349  
`\vrule` ..... 249  
`\vsize` ..... 292  
`\vskip` ..... 243  
`\vsplit` ..... 321  
`\vss` ..... 244  
`\vtop` ..... 329
- ### W
- `\WARNING` ..... 5456  
`\wd` ..... 376  
`\widowpenalties` ..... 436  
`\widowpenalty` ..... 263  
`\widthof` ..... 111, 6098, 6099  
`\wlog` ..... 6257  
`\write` ..... 126
- ### X
- `\X` ..... 2363, 2367  
`\xdef` ..... 67  
`\xetex_if_engine:` ..... 1235, 1244  
`\xetex_if_engine:TF` ..... 3, 23, 1230  
`\xetex_version:D` ..... 552, 555, 1230  
`\XeTeXversion` ..... 552  
`\xleaders` ..... 252  
`\xref_deferred_new:nn` ..... 108, 5419, 5420, 5491, 5494  
`\xref_define_label:nn` 286, 5449, 5449, 5463  
`\xref_define_label_aux:nn` 5449, 5452, 5454  
`\xref_get_value:nn` ..... 108, 5435, 5435, 5489, 5492, 5495  
`\xref_new:nn` ..... 108, 5419, 5419, 5487  
`\xref_new_aux:nnn` ..... 5419, 5419–5421  
`\xref_set_label:n` . 108, 5460, 5460, 5497  
`\xref_tmp:w` ..... 5461, 5464  
`\xref_write` ... 286, 5462, 5469, 5469, 5478  
`\xspaceskip` ..... 286
- ### Y
- `\Y` ..... 2364, 2367  
`\year` ..... 367
- ### Z
- `\Z` ..... 2365, 2367  
`\z@` ..... 3334