new primitives

in luametatex

Introduction

Here I will discuss some of the new primitives in LuaTEX and LuaMetaTEX, the later being a successor that permits the ConTEXt folks to experiment with new features. The order is arbitrary. When you compare LuaTEX with pdfTEX, there are actually quite some differences. Some primitives that pdfTEX introduced have been dropped in LuaTEX because they can be done better in Lua. Others have been promoted to core primitives that no longer have a `pdf` prefix. Then there are lots of new primitives, some introduce new concepts, some are a side effect of for instance new math font technologies, and then there are those that are handy extensions to the macro language. The LuaMetaTEX engine drops quite some primitives, like those related to pdfTEX specific f(r)ont or backend features. It also adds some new primitives, mostly concerning the macro language.

We also discuss the primitives that fit into the macro programming scope that are present in traditional TEX and ε-TEX but there are for sure better of explanations out there already. Primitives that relate to typesetting, like those controlling math, fonts, boxes, attributes, directions, catcodes, Lua (functions) etc are not discussed or discussed in less detail here.

There are for instance primitives to create aliases to low level registers like counters and dimensions, as well as other (semi-numeric) quantities like characters, but normally these are wrapped into high level macros so that definitions can't clash too much. Numbers, dimensions etc can be advanced, multiplied and divided and there is a simple expression mechanism to deal with them. We don't go into these details here: it's mostly an overview of what the engine provides. If you are new to TEX, you need to play a while with its mixed bag of typesetting and programming features in order to understand the difference between this macro language and other languages you might be familiar with.

In this document the section titles that discuss the original TEX and ε-TEX primitives have a different color those explaining the LuaTEX and LuaMetaTEX primitives.

Primitives that extend typesetting related functionality, provide control over subsystems (like math), allocate additional data types and resources, deal with fonts and languages, manipulate boxes and glyphs, etc. are hardly discussed here. Math for instance is a topic of its own. In this document we concentrate on the programming aspects.

Primitives

1 \

This original TEX primitive is equivalent to the more verbose `\explicitspace`.

2 \-

This original TEX primitive is equivalent to the more verbose `\explicitdiscretionary`.

3 \/

This original TEX primitive is equivalent to the more verbose `\explicititaliccorrection`.

4 \advance

Advances the given register by an also given value:

```
\advance\scratchdimen      10pt
\advance\scratchdimen      by 3pt
\advance\scratchcounterone \zerocount
\advance\scratchcounterone \scratchcountertwo
```

The by keyword is optional.

5 \advanceby

This is slightly more efficient variant of \advance that doesn't look for by and therefore, if one is missing, doesn't need to push back the last seen token. Using \advance with by is nearly as efficient but takes more tokens.

6 \afterassigned

The \afterassignment primitive stores a token to be injected (and thereby expanded) after an assignment has happened. Unlike \aftergroup, multiple calls are not accumulated, and changing that would be too incompatible. This is why we have \afterassigned, which can be used to inject a bunch of tokens. But in order to be consistent this one is also not accumulative.

```
\afterassigned{done}%
\afterassigned{{\bf done}}%
\scratchcounter=123
```

results in: **done** being typeset.

7 \afterassignment

The token following \afterassignment, a traditional TEX primitive, is saved and gets injected (and then expanded) after a following assignment took place.

```
\afterassignment !\def\MyMacro {}\quad
\afterassignment !\let\MyMacro ?\quad
\afterassignment !\scratchcounter 123\quad
\afterassignment !%
\afterassignment ?\advance\scratchcounter by 1
```

The \afterassignments are not accumulated, the last one wins:

! ! ! ?

8 \aftergroup

The traditional TEX \aftergroup primitive stores the next token and expands that after the group has been closed.

Multiple \aftergroups are combined:

```
before{ ! \aftergroup a\aftergroup f\aftergroup t\aftergroup e\aftergroup r}
```

before ! after

9 \aftergrouped

The in itself powerful `\aftergroup` primitives works quite well, even if you need to do more than one thing: you can either use it multiple times, or you can define a macro that does multiple things and apply that after the group. However, you can avoid that by using this primitive which takes a list of tokens.

```
regular
\bgroup
\aftergrouped{regular}%
\bf bold
\egroup
```

Because it happens after the group, we're no longer typesetting in bold.

regular **bold** regular

10 \aliased

This primitive is part of the overload protection subsystem where control sequences can be tagged.

```
\permanent\def\foo{FOO}
          \let\ofo\foo
\aliased  \let\oof\foo
```

```
\meaningasis\foo
\meaningasis\ofo
\meaningasis\oof
```

gives:

```
\permanent \def \foo {FOO}
\def \ofo {FOO}
\permanent \def \oof {FOO}
```

When a something is `\let` the ‘permanent’, ‘primitive’ and ‘immutable’ flags are removed but the `\aliased` prefix retains them.

```
\let\relaxed\relax
```

```
\meaningasis\relax
\meaningasis\relaxed
```

So in this example the `\relaxed` alias is not flagged as primitive:

```
\primitive \relax
\relax
```

11 \alignmark

When you have the # not set up as macro parameter character cq. align mark, you can use this primitive instead. The same rules apply with respect to multiple such tokens in (nested) macros and alignments.

12 \aligntab

When you have the & not set up as align tab, you can use this primitive instead. The same rules apply with respect to multiple such tokens in (nested) macros and alignments.

13 \associateunit

The TEX engine comes with some build in units, like `pt` (fixed) and `em` (adaptive). On top of that a macro package can add additional units, which is what we do in ConTEXt. In figure 1 we show the current repertoire.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
b																bp										
c			cc	cd				ch					cm										cw	cx		
d				dd							dk															
e													em						es		eu			ex		
f	fa		fc	fd				fh	fi						fo				fs	ft			fw			
h																			hs							
i														in												
l																							lw			
m	ma												mm				mq				mu			mx		
p			pc						pi											pt				px		
s				sd				sh								sp				st						
t								th											ts				tw			
u																					uu					
v																			vs							

tex pdftex luametatex context

Figure 1 Available units

When this primitive is used in a context where a number is expected it returns the origin of the unit (in the color legend running from 1 upto 4). A new unit is defined as:

```
\newdimen\MyDimenZA  \MyDimenZA=10pt
```

```
\protected\def\MyDimenAB{\dimexpr\hsize/2\relax}
```

```
\associateunit za \MyDimenZA
\associateunit zb \MyMacroZB
```

Possible associations are: macros that expand to a dimension, internal dimension registers, register dimensions (`\dimendef`, direct dimensions (`\dimensiondef`) and Lua functions that return a dimension.

One can run into scanning ahead issues where TEX expects a unit and a user unit gets expanded. This is why for instance in ConTEXt we define the `ma` unit as:

```
\protected\def\mathaxisunit{\scaledmathaxis\mathstyle\norelax}
```

```
\associateunit ma \mathaxisunit % or \newuserunit \mathaxisunit ma
```

So that it can be used in rule specifications that themselves look ahead for keywords and therefore are normally terminated by a `\relax`. Adding the extra `\norelax` will make the scanner see one that doesn't get fed back into the input. Of course a macro package has to manage extra units in order to avoid conflicts.

14 \atendoffile

The `\everyeof` primitive is kind of useless because you don't know if a file (which can be a tokenlist processed as pseudo file) itself includes a file, which then results in nested application of this token register. One way around this is:

```
\atendoffile\SomeCommand
```

This acts on files the same way as `\atendofgroup` does. Multiple calls will be accumulated and are bound to the current file.

15 \atendoffiled

This is the multi token variant of `\atendoffile`. Multiple invocations are accumulated and by default prepended to the existing list. As with grouping this permits proper nesting. You can force an append by the optional keyword reverse.

16 \atendofgroup

The token provided will be injected just before the group ends. Because these tokens are collected, you need to be aware of possible interference between them. However, normally this is managed by the macro package.

```
\bgroup
\atendofgroup\unskip
\atendofgroup )%
(but it works okay
\egroup
```

Of course these effects can also be achieved by combining (extra) grouping with `\aftergroup` calls, so this is more a convenience primitives than a real necessity: (but it works okay), as proven here.

17 \atendofgrouped

This is the multi token variant of `\atendofgroup`. Of course the next example is somewhat naive when it comes to spacing and so, but it shows the purpose.

```
\bgroup
\atendofgrouped{\bf QED}%
\atendofgrouped{ (indeed)}%
This sometimes looks nicer.
\egroup
```

Multiple invocations are accumulated: This sometimes looks nicer. **QED (indeed)**.

18 \attribute

The following sets an attribute(register) value:

```
\attribute 999 = 123
```

An attribute is unset by assigning -2147483647 to it. A user needs to be aware of attributes being used now and in the future of a macro package and setting them this way is very likely going to interfere.

19 \attributedef

This primitive can be used to relate a control sequence to an attribute register and can be used to implement a mechanism for defining unique ones that won't interfere. As with other registers: leave management to the macro package in order to avoid unwanted side effects!

20 \batchmode

This command disables (error) messages which can safe some runtime in situations where TEX's character-by-character log output impacts runtime. It only makes sense in automated workflows where one doesn't look at the log anyway.

21 \begincsname

The next code creates a control sequence token from the given serialized tokens:

```
\csname mymacro\endcsname
```

When `\mymacro` is not defined a control sequence will be created with the meaning `\relax`. A side effect is that a test for its existence might fail because it now exists. The next sequence will *not* create an controil sequence:

```
\begincsname mymacro\endcsname
```

This actually is kind of equivalent to:

```
\ifcsname mymacro\endcsname
    \csname mymacro\endcsname
\fi
```

22 \begingroup

This primitive starts a group and has to be ended with `\endgroup`. See `\beginsimplegroup` for more info.

23 \beginlocalcontrol

Once TEX is initialized it will enter the main loop. In there certain commands trigger a function that itself can trigger further scanning and functions. In LuaMetaTEX we can have local main loops and we can either enter it from the Lua end (which we don't discuss here) or at the TEX end using this primitive.

```
\scratchcounter100

\edef\whatever{
    a
    \beginlocalcontrol
        \advance\scratchcounter 10
        b
    \endlocalcontrol
    \beginlocalcontrol
        c
    \endlocalcontrol
    d
    \advance\scratchcounter 10
}

\the\scratchcounter
\whatever
\the\scratchcounter
```

A bit of close reading probably gives an impression of what happens here:

b c

110 a d 120

The local loop can actually result in material being injected in the current node list. However, where normally assignments are not taking place in an `\edef`, here they are applied just fine. Basically we have a local TEX job, be it that it shares all variables with the parent loop.

24 \beginmathgroup

In math mode grouping with `\begingroup` and `\endgroup` in some cases works as expected, but because the math input is converted in a list that gets processed later some settings can become persistent, like changes in style or family. The engine therefore provides the alternatives `\beginmathgroup` and `\endmathgroup` that restore some properties.

25 \beginsimplegroup

The original TEX engine distinguishes two kind of grouping that at the user end show up as:

```
\begingroup \endgroup
\bgroup \egroup { }
```

where the last two pairs are equivalent unless the scanner explicitly wants to see a left and/or right brace and not an equivalent. For the sake of simplify we use the aliases here. It is not possible to mix these pairs, so:

```
\bgroup xxx\endgroup
\begingroup xxx\egroup
```

will in both cases issue an error. This can make it somewhat hard to write generic grouping macros without somewhat dirty trickery. The way out is to use the generic group opener `\beginsimplegroup`.

Internally LuaMetaTEX is aware of what group it currently is dealing with and there we distinguish:

simple group	`\bgroup`	`\egroup`
semi simple group	`\begingroup`	`\endgroup \endsimplegroup`
also simple group	`\beginsimplegroup`	`\egroup \endgroup \endsimplegroup`
math simple group	`\beginmathgroup`	`\endmathgroup`

This means that you can say:

```
\beginsimplegroup xxx\endsimplegroup
\beginsimplegroup xxx\endgroup
\beginsimplegroup xxx\egroup
```

So a group started with `\beginsimplegroup` can be finished in three ways which means that the user (or calling macro) doesn't have take into account what kind of grouping was used to start with. Normally usage of this primitive is hidden in macros and not something the user has to be aware of.

26 \catcode

Every character can be put in a category, but this is typically something that the macro package manages because changes can affect behavior. Also, once passed as an argument, the catcode of a character is frozen. There are 16 different values:

`\escapecatcode`	0	`\begingroupcatcode`	1
`\endgroupcatcode`	2	`\mathshiftcatcode`	3
`\alignmentcatcode`	4	`\endoflinecatcode`	5
`\parametercatcode`	6	`\superscriptcatcode`	7
`\subscriptcatcode`	8	`\ignorecatcode`	9
`\spacecatcode`	10	`\lettercatcode`	11
`\othercatcode`	12	`\activecatcode`	13
`\commentcatcode`	14	`\invalidcatcode`	15

The first column shows the constant that ConTEXt provides and the name indicates the purpose. Here are two examples:

```
\catcode123=\begingroupcatcode
\catcode125=\endgroupcatcode
```

27 \catcodetable

The catcode table with the given index will become active.

28 \cdef

This primitive is like `\edef` but in some usage scenarios is slightly more efficient because (delayed) expansion is ignored which in turn saves building a temporary token list.

```
\edef\FooA{this is foo} \meaningfull\FooA\crlf
\cdef\FooB{this is foo} \meaningfull\FooB\par
```

```
macro:this is foo
constant macro:this is foo
```

29 \cdefcsname

This primitive is like `\edefcsame` but in some usage scenarios is slightly more efficient because (delayed) expansion is ignored which in turn saves building a temporary token list.

```
\edefcsname FooA\endcsname{this is foo} \meaningasis\FooA\crlf
\cdefcsname FooB\endcsname{this is foo} \meaningasis\FooB\par
```

```
\def \FooA {this is foo}
\constant \def \FooB {this is foo}
```

30 \chardef

The following definition relates a control sequence to a specific character:

```
\chardef\copyrightsign"A9
```

However, because in a context where a number is expected, such a `\chardef` is seen as valid number, there was a time when this primitive was used to define constants without overflowing the by then limited pool of count registers. In ε-TEX aware engines this was less needed, and in LuaMetaTEX we have `\integerdef` as a more natural candidate.

31 \constant

This prefix tags a macro (without arguments) as being constant. The main consequence is that in some cases expansion gets delayed which gives a little performance boost and less (temporary) memory usage, for instance in **`\csname`** like scenarios.

32 \constrained

See previous section about `\retained`.

33 \count

This accesses a count register by index. This is kind of ‘not done’ unless you do it local and make sure that it doesn't influence macros that you call.

```
\count4023=10
```

In standard TEX the first 10 counters are special because they get reported to the console, and **`\count`**`0` is then assumed to be the page counter.

34 \countdef

This primitive relates a control sequence to a count register. Compare this to the example in the previous section.

```
\countdef\MyCounter4023
\MyCounter=10
```

However, this is also ‘not done’. Instead one should use the allocator that the macro package provides.

```
\newcount\MyCounter
\MyCounter=10
```

In LuaMetaTEX we also have integers that don't rely on registers. These are assigned by the primitive `\integerdef`:

```
\integerdef\MyCounterA 10
```

Or better `\newinteger`.

```
\newinteger\MyCounterB
\MyCounterN10
```

There is a lowlevel manual on registers.

35 \csactive

Because LuaTEX (and LuaMetaTEX) are Unicode engines active characters are implemented a bit differently. They don't occupy a eight bit range of characters but are stored as control sequence with a special prefix `U+FFFF` which never shows up in documents. The `\csstring` primitive injects the name of a control sequence without leading escape character, the `\csactive` injects the internal name of the following (either of not active) character. As we cannot display the prefix: `\csactive~` will inject the utf sequences for `U+FFFF` and `U+007E`, so here we get the bytes `EFBFBF7E`. Basically the next token is preceded by `\string`, so when you don't provide a character you are in for a surprise.

36 \csname

This original TEX primitive starts the construction of a control sequence reference. It does a lookup and when no sequence with than name is found, it will create a hash entry and defaults its meaning to `\relax`.

```
\csname letters and other characters\endcsname
```

37 \csstring

This primitive returns the name of the control sequence given without the leading escape character (normally a backslash). Of course you could strip that character with a simple helper but this is more natural.

```
\csstring\mymacro
```

We get the name, not the meaning: `mymacro`.

38 \currentgrouplevel

The next example gives: [1] [2] [3] [2] [1].

```
[\the\currentgrouplevel] \bgroup
    [\the\currentgrouplevel] \bgroup
        [\the\currentgrouplevel]
    \egroup [\the\currentgrouplevel]
\egroup [\the\currentgrouplevel]
```

39 \currentgrouptype

The next example gives: [22] [1] [22] [1] [1] [23] [1] [1].

```
[\the\currentgrouptype] \bgroup
    [\the\currentgrouptype] \begingroup
        [\the\currentgrouptype]
    \endgroup [\the\currentgrouptype]
    [\the\currentgrouptype] \beginmathgroup
        [\the\currentgrouptype]
    \endmathgroup [\the\currentgrouptype]
[\the\currentgrouptype] \egroup
```

The possible values depend in the engine and for LuaMetaTEX they are:

0	bottomlevel	9	output	18	mathoperator	27	mathnumber
1	simple	10	mathsubformula	19	mathradical	28	localbox
2	hbox	11	mathstack	20	mathchoice	29	splitoff
3	adjustedhbox	12	mathcomponent	21	alsosimple	30	splitkeep
4	vbox	13	discretionary	22	semisimple	31	preamble
5	vtop	14	insert	23	mathsimple	32	alignset
6	dbox	15	vadjust	24	mathfence	33	finishrow
7	align	16	vcenter	25	mathinline	34	lua
8	noalign	17	mathfraction	26	mathdisplay		

40 \currentifbranch

The next example gives: [0] [1] [-1] [1] [0].

```
[\the\currentifbranch] \iftrue
    [\the\currentifbranch] \iffalse
        [\the\currentifbranch]
    \else
        [\the\currentifbranch]
    \fi [\the\currentifbranch]
\fi [\the\currentifbranch]
```

So when in the ‘then’ branch we get plus one and when in the ‘else’ branch we end up with a minus one.

41 \currentiflevel

The next example gives: [0] [1][2] [3] [2] [1] [0].

```
[\the\currentiflevel] \iftrue
    [\the\currentiflevel]\iftrue
        [\the\currentiflevel] \iftrue
            [\the\currentiflevel]
        \fi [\the\currentiflevel]
    \fi [\the\currentiflevel]
\fi [\the\currentiflevel]
```

42 \currentiftype

The next example gives: [-1] [25][25] [25] [25] [25] [-1].

```
[\the\currentiftype] \iftrue
    [\the\currentiftype]\iftrue
        [\the\currentiftype] \iftrue
            [\the\currentiftype]
        \fi [\the\currentiftype]
    \fi [\the\currentiftype]
\fi [\the\currentiftype]
```

The values are engine dependent:

0	char	7	absfloat	14	odd	21	vbox	28	chknunber
1	cat	8	zerofloat	15	vmode	22	tok	29	numval
2	num	9	intervalfloat	16	hmode	23	cstoken	30	cmpnum
3	absnum	10	dim	17	mmode	24	x	31	chkdim
4	zeronum	11	absdim	18	inner	25	true	32	chkdimension
5	intervalnum	12	zerodim	19	void	26	false	33	dimval
6	float	13	intervaldim	20	hbox	27	chknum	34	cmpdim

43 \currentloopiterator

Here we show the different expanded loop variants:

```
\edef\testA{\expandedloop  1 10 1{!}}
\edef\testB{\expandedrepeat  10  {!}}
\edef\testC{\expandedendless     {\ifnum\currentloopiterator>10 \quitloop\else !\fi}}
\edef\testD{\expandedendless     {\ifnum#I>10 \quitloop\else !\fi}}
```

All these give the same result:

```
\def \testA {!!!!!!!!!!}
\def \testB {!!!!!!!!!!}
\def \testC {!!!!!!!!!!}
\def \testD {!!!!!!!!!!}
```

The `#I` is a shortcut to the current loop iterator; other shortcuts are `#P` for the parent iterator value and `#G` for the grand parent.

44 \currentloopnesting

This integer reports how many nested loops are currently active. Of course in practice the value only has meaning when you know at what outer level your nested loop started.

45 \currentstacksize

This is more diagnostic feature than a useful one but we show it anyway. There is some basic overhead when we enter a group:

```
\bgroup [\the\currentstacksize]
    \bgroup [\the\currentstacksize]
        \bgroup [\the\currentstacksize]
        [\the\currentstacksize] \egroup
    [\the\currentstacksize] \egroup
[\the\currentstacksize] \egroup
```

[57] [58] [59] [59] [58] [57]

As soon as we define something or change a value, the stack gets populated by information needed for recovery after the group ends.

```
\bgroup [\the\currentstacksize]
    \scratchcounter 1
    \bgroup [\the\currentstacksize]
        \scratchdimen 1pt
        \scratchdimen 2pt
        \bgroup [\the\currentstacksize]
            \scratchcounter 2
            \scratchcounter 3
        [\the\currentstacksize] \egroup
    [\the\currentstacksize] \egroup
[\the\currentstacksize] \egroup
```

[57] [59] [61] [62] [60] [58]

The stack also keeps some state information, for instance when a box is being built. In LuaMetaTEX that is is quite a bit more than in other engines but it is compensated by more efficient save stack handling elsewhere.

```
\hbox \bgroup [\the\currentstacksize]
    \hbox \bgroup [\the\currentstacksize]
        \hbox \bgroup [\the\currentstacksize]
        [\the\currentstacksize] \egroup
    [\the\currentstacksize] \egroup
[\the\currentstacksize] \egroup
```

[65] [74] [83] [83] [74] [65]

46 \def

This is the main definition command, as in:

```
\def\foo{l me}
```

with companions like `\gdef`, `\edef`, `\xdef`, etc. and variants like:

```
\def\foo#1{... #1...}
```

where the hash is used in the preamble and for referencing. More about that can be found in the low level manual about macros.

47 \defcsname

We now get a series of log clutter avoidance primitives. It's fine if you argue that they are not really needed, just don't use them.

```
\expandafter\def\csname MyMacro:1\endcsname{...}
             \defcsname MyMacro:1\endcsname{...}
```

The fact that TEX has three (expanded and global) companions can be seen as a signal that less verbosity makes sense. It's just that macro packages use plenty of `\csname`'s.

48 \deferred

This is mostly a compatibility prefix and it can be checked at the Lua end when there is a Lua based assignment going on. It is the counterpart of `\immediate`. In the traditional engines a `\write` is normally deferred (turned into a node) and can be handled `\immediate`, while a `\special` does the opposite.

49 \detokened

The following token will be serialized into characters with category 'other'.

```
\toks0{123}
\def\foo{let's be \relax'd}
\def\oof#1{let's see #1}
\detokened\toks0
\detokened\foo
\detokened\oof
\detokened\setbox
\detokened X
```

Gives:

```
123
let's be \relax 'd
\oof
\setbox
X
```

Macros with arguments are not shown.

50 \detokenize

This ε-TEX primitive turns the content of the provides list will become characters, kind of verbatim.

```
\expandafter\let\expandafter\temp\detokenize{1} \meaning\temp
\expandafter\let\expandafter\temp\detokenize{A} \meaning\temp
```

```
the character U+0031 1
the character U+0041 A
```

51 \detokenized

The following (single) token will be serialized into characters with category ‘other’.

```
\toks0{123}
\def\foo{let's be \relax'd}
\def\oof#1{let's see #1}
\detokenized\toks0
\detokenized\foo
\detokenized\oof
\detokenized\setbox
\detokenized X
```

Gives:

```
\toks 0
\foo
\oof
\setbox
X
```

It is one of these new primitives that complement others like `\detokened` and such, and they are often mostly useful in experiments of some low level magic, which made them stay.

52 \dimen

Like `\count` this is a register accessor which is described in more detail in a low level manual.

```
\dimen0=10pt
```

While TEX has some assumptions with respect to the first ten count registers (as well as the one that holds the output, normally 255), all dimension registers are treated equal. However, you need to be aware of clashes with other usage. Therefore you can best use the predefined scratch registers or define dedicate ones with the **`\newdimen`** macro.

53 \dimendef

This primitive is used by the **`\newdimen`** macro when it relates a control sequence with a specific register. Only use it when you know what you're doing.

54 \dimensiondef

A variant of `\integerdef` is:

```
\dimensiondef\MyDimen = 1234pt
```

The properties are comparable to the ones described in the section `\integerdef`.

55 \dimexpr

This primitive is similar to of `\numexpr` but operates on dimensions instead. Integer quantities are interpreted as dimensions in scaled points.

```
\the\dimexpr (1pt + 2pt - 5pt) * 10 / 2 \relax
```

gives: -10.0pt. You can mix in symbolic integers and dimensions. This doesn't work:

because the engine scans for a dimension and only for an integer (or equivalent) after a `*` or `/`.

56 \dimexpression

This command is like `\numexpression` but results in a dimension instead of an integer. Where `\dimexpr` doesn't like `2 * 10pt` this expression primitive is quite happy with it.

57 \directlua

This is the low level interface to Lua:

Gives: “Greetings from the lua end!” as expected. In Lua we have access to all kind of internals of the engine. In LuaMetaTEX the interfaces have been polished and extended compared to LuaTEX. Although many primitives and mechanisms were added to the TEX frontend, the main extension interface remains Lua. More information can be found in documents that come with ConTEXt, in presentations and in articles.

58 \divide

The `\divide` operation can be applied to integers, dimensions, float, attribute and glue quantities. There are subtle rounding differences between the divisions in expressions and `\divide`:

```
\scratchcounter 1049 \numexpr\scratchcounter / 10\relax : 105
\scratchcounter 1049 \numexpr\scratchcounter : 10\relax : 104
\scratchcounter 1049 \divide\scratchcounter by 10       : 104
```

The `:` divider in `\dimexpr` is something that we introduced in LuaTEX.

59 \divideby

This is slightly more efficient variant of `\divide` that doesn't look for `by`. See previous section.

60 \edef

This is the expanded version of \def.

```
\def \foo{foo}       \meaning\foo
\def \ofo{\foo\foo} \meaning\ofo
\edef\oof{\foo\foo} \meaning\oof
```

Because \foo is unprotected it will expand inside the body definition:

```
macro:foo
macro:\foo \foo
macro:foofoo
```

61 \edefcsname

This is the companion of \edef:

```
\expandafter\edef\csname MyMacro:1\endcsname{...}
             \edefcsname MyMacro:1\endcsname{...}
```

62 \edivide

When expressions were introduced the decision was made to round the divisions which is incompatible with the way \divide works. The expression scanners in LuaMetaTEX compensates that by providing a : for integer division. The \ edivide does the opposite: it rounds the way expressions do.

```
\the\dimexpr .4999pt                     : 2 \relax           =.24994pt
\the\dimexpr .4999pt                     / 2 \relax           =.24995pt
\scratchdimen.4999pt \divide \scratchdimen 2 \the\scratchdimen=.24994pt
\scratchdimen.4999pt \edivide\scratchdimen 2 \the\scratchdimen=.24995pt

\the\numexpr   1001                        : 2 \relax             =500
\the\numexpr   1001                        / 2 \relax             =501
\scratchcounter1001  \divide \scratchcounter 2 \the\scratchcounter=500
\scratchcounter1001  \edivide\scratchcounter 2 \the\scratchcounter=501
```

Keep in mind that with dimensions we have a fractional part so we actually rounding applies to the fraction. For that reason we also provide \rdivide.

0.24994pt=.24994pt
0.24995pt=.24995pt
0.24994pt=.24994pt
0.24995pt=.24995pt

500=500
501=501
500=500
501=501

63 \edivideby

This the by-less variant of `\edivide`.

64 \else

This traditional primitive is part of the condition testing mechanism. When a condition matches, TEX will continue till it sees an `\else` or `\or` or `\orelse` (to be discussed later). It will then do a fast skipping pass till it sees an `\fi`.

65 \end

This ends a TEX run, unless of course this primitive is redefined.

66 \endcsname

This primitive is used in combination with `\csname`, `\ifcsname` and `\begincsname` where its end the scanning for the to be constructed control sequence token.

67 \endgroup

This is the companion of the `\begingroup` primitive that opens a group. See `\beginsimplegroup` for more info.

68 \endinput

The engine can be in different input modes: reading from file, reading from a token list, expanding a macro, processing something that comes back from Lua, etc. This primitive quits reading from file:

```
this is seen
\endinput
here we're already quit
```

There is a catch. This is what the above gives:

this is seen

but how about this:

```
this is seen
before \endinput after
here we're already quit
```

Here we get:

this is seen before after

Because a token list is one line, the following works okay:

```
\def\quitrun{\ifsomething \endinput \fi}
```

but in a file you'd have to do this when you quit in a conditional:

```
\ifsomething
    \expandafter \endinput
\fi
```

While the one-liner works as expected:

```
\ifsomething \endinput \fi
```

69 \endlinechar

This is an internal integer register. When set to positive value the character with that code point will be appended to the line. The current value is 13. Here is an example:

```
\endlinechar\hyphenasciicode
line 1
line 2
```

line 1-line 2-

If the character is active, the property is honored and the command kicks in. The maximum value is 127 (the maximum character code a single byte utf character can carry.)

70 \endlocalcontrol

See `\beginlocalcontrol`.

71 \endmathgroup

This primitive is the counterpart of `\beginmathgroup`.

72 \endsimplegroup

This one ends a simple group, see `\beginsimplegroup` for an explanation about grouping primitives.

73 \enforced

The engine can be set up to prevent overloading of primitives and macros defined as `\permanent` or `\immutable`. However, a macro package might want to get around this in controlled situations, which is why we have a `\enforced` prefix. This prefix in interpreted differently in so called ‘ini’ mode when macro definitions can be dumped in the format. Internally they get an `always` flag as indicator that in these places an overload is possible.

```
\permanent\def\foo{original}

\def\oof          {\def\foo{fails}}
\def\oof{\enforced\def\foo{succeeds}}
```

Of course this only has an effect when overload protection is enabled.

74 \eofinput

This is a variant on `\input` that takes a token list as first argument. That list is expanded when the file ends. It has companion primitives `\atendoffile` (single token) and `\atendoffiled` (multiple tokens).

75 \errmessage

This primitive expects a token list and shows its expansion on the console and/or in the log file, depending on how TEX is configured. After that it will enter the error state and either goes on or waits for input, again depending on how TEX is configured. For the record: we don't use this primitive in ConTEXt.

76 \errorstopmode

This directive stops at every opportunity to interact. In ConTEXt we overload the actions in a callback and quit the run because we can assume that a successful outcome is unlikely.

77 \escapechar

This internal integer has the code point of the character that get prepended to a control sequence when it is serialized (for instance in tracing or messages).

78 \etoks

This assigns an expanded token list to a token register:

```
\def\temp{less stuff}
\etoks\scratchtoks{a bit \temp}
```

The orginal value of the register is lost.

79 \etoksapp

A variant of `\toksapp` is the following: it expands the to be appended content.

```
\def\temp{more stuff}
\etoksapp\scratchtoks{some \temp}
```

80 \etokspre

A variant of `\tokspre` is the following: it expands the to be prepended content.

```
\def\temp{less stuff}
\etokspre\scratchtoks{a bit \temp}
```

81 \eufactor

When we introduced the `es` (2.5cm) and `ts` (2.5mm) units as metric variants of the `in` we also added the `eu` factor. One `eu` equals one tenth of a `es` times the `\eufactor`. The `ts` is a convenient offset in test files, the `es` a convenient ones for layouts and image dimensions and the `eu` permits definitions that scale nicely without the need for dimensions. They also were a prelude to what later became possible with `\associateunit`.

82 `\everyeof`

The content of this token list is injected when a file ends but it can only be used reliably when one is really sure that no other file is loaded in the process. So in the end it is of no real use in a more complex macro package.

83 `\everyjob`

This token list register is injected at the start of a job, or more precisely, just before the main control loop starts.

84 `\expand`

Beware, this is not a prefix but a directive to ignore the protected characters of the following macro.

```
\protected \def \testa{\the\scratchcounter}
           \edef\testb{\testa}
           \edef\testc{\expand\testa}
```

The meaning of the three macros is:

```
protected macro:\the \scratchcounter
macro:\testa
macro:123
```

85 `\expandactive`

This a bit of an outlier and mostly there for completeness.

```
                           \meaningasis~
\edef\foo{~}               \meaningasis\foo
\edef\foo{\expandactive~}  \meaningasis\foo
```

There seems to be no difference but the real meaning of the first `\foo` is ‘active character 126’ while the second `\foo` ‘protected call ’ is.

```
\protected \def ~ {\nobreakspace }
\def \foo {~}
\def \foo {~}
```

Of course the definition of the active tilde is ConTEXt specific and situation dependent.

86 `\expandafter`

This original TEX primitive stores the next token, does a one level expansion of what follows it, which actually can be an not expandable token, and reinjects the stored token in the input. Like:

```
\expandafter\let\csname my weird macro name\endcsname{m w m n}
```

Without `\expandafter` the `\csname` primitive would have been let to the left brace (effectively then a begin group). Actually in this particular case the control sequence with the weird name is injected and when it didn't yet exist it will get the meaning `\relax` so we sort of have two assignments in a row then.

87 \expandafterpars

Here is another gobbler: the next token is reinjected after following spaces and par tokens have been read. So:

```
[\expandafterpars 1 2]
[\expandafterpars 3
4]
[\expandafterpars 5

6]
```

gives us: [12] [34] [56], because empty lines are like `\par` and therefore ignored.

88 \expandafterspaces

This is a gobbler: the next token is reinjected after following spaces have been read. Here is a simple example:

```
[\expandafterspaces 1 2]
[\expandafterspaces 3
4]
[\expandafterspaces 5

6]
```

We get this typeset: [12] [34] [5

6], because a newline normally is configured to be a space (and leading spaces in a line are normally being ingored anyway).

89 \expandcstoken

The rationale behind this primitive is that when we `\let` a single token like a character it is hard to compare that with something similar, stored in a macro. This primitive pushes back a single token alias created by `\let` into the input.

```
\let\tempA + \meaning\tempA

\let\tempB X \meaning\tempB \crlf
\let\tempC $ \meaning\tempC \par

\edef\temp         {\tempA} \doifelse{\temp}{+}{Y}{N} \meaning\temp \crlf
\edef\temp         {\tempB} \doifelse{\temp}{X}{Y}{N} \meaning\temp \crlf
\edef\temp         {\tempC} \doifelse{\temp}{X}{Y}{N} \meaning\temp \par
```

```
\edef\temp{\expandcstoken\tempA} \doifelse{\temp}{+}{Y}{N} \meaning\temp \crlf
\edef\temp{\expandcstoken\tempB} \doifelse{\temp}{X}{Y}{N} \meaning\temp \crlf
\edef\temp{\expandcstoken\tempC} \doifelse{\temp}{$}{Y}{N} \meaning\temp \par

\doifelse{\expandcstoken\tempA}{+}{Y}{N}
\doifelse{\expandcstoken\tempB}{X}{Y}{N}
\doifelse{\expandcstoken\tempC}{$}{Y}{N} \par
```

The meaning of the `\let` macros shows that we have a shortcut to a character with (in this case) catcode letter, other (here 'other character' gets abbreviated to 'character'), math shift etc.

```
the character U+002B 'plus sign'

the letter U+0058 X
math shift character U+0024 'dollar sign'

N macro:\tempA
N macro:\tempB
N macro:\tempC

Y macro:+
Y macro:X
Y macro:$

Y Y Y
```

Here we use the ConTEXt macro **`\doifelse`** which can be implemented in different ways, but the only property relevant to the user is that the expanded content of the two arguments is compared.

90 \expanded

This primitive complements the two expansion related primitives mentioned in the previous two sections. This time the content will be expanded and then pushed back into the input. Protected macros will not be expanded, so you can use this primitive to expand the arguments in a call. In ConTEXt you need to use **`\normalexpanded`** because we already had a macro with that name. We give some examples:

```
\def\A{!}
          \def\B#1{\string#1}                          \B{\A}
          \def\B#1{\string#1} \normalexpanded{\noexpand\B{\A}}
\protected\def\B#1{\string#1}                          \B{\A}
```

\A
!
\A

91 \expandedafter

The following two lines are equivalent:

```
\def\foo{123}
\expandafter[\expandafter[\expandafter\secondofthreearguments\foo]]
```

```
\expandedafter{[[\secondofthreearguments}\foo]]
```

In ConTEXt MkIV the number of times that one has multiple `\expandafters` is much larger than in ConTEXt LMTX thanks to some of the new features in LuaMetaTEX, and this primitive is not really used yet in the core code.

[[2]]
[[2]]

92 \expandeddetokenize

This is a companion to `\detokenize` that expands its argument:

```
\def\foo{12#H3}
\def\oof{\foo}
\detokenize        {\foo} \detokenize        {\oof}
\expandeddetokenize{\foo} \expandeddetokenize{\oof}
\edef\ofo{\expandeddetokenize{\foo}} \meaningless\ofo
\edef\ofo{\expandeddetokenize{\oof}} \meaningless\ofo
```

This is a bit more convenient than

```
\detokenize \expandafter {\normalexpanded {\foo}}
```

kind of solutions. We get:

\foo \oof
12#3 12#3
12#3
12#3

93 \expandedendless

This one loops forever but because the loop counter is not set you need to find a way to quit it.

94 \expandedloop

This variant of the previously introduced `\localcontrolledloop` doesn't enter a local branch but immediately does its work. This means that it can be used inside an expansion context like `\edef`.

```
\edef\whatever
  {\expandedloop 1 10 1
     {\scratchcounter=\the\currentloopiterator\relax}}

\meaningasis\whatever
```

```
\def \whatever {\scratchcounter =1\relax \scratchcounter =2\relax \scratchcounter =3\relax \scratchcounter
=4\relax \scratchcounter =5\relax \scratchcounter =6\relax \scratchcounter =7\relax \scratchcounter =8\relax
\scratchcounter =9\relax \scratchcounter =10\relax }
```

95 \expandedrepeat

This one takes one instead of three arguments which is sometimes more convenient.

96 \expandparameter

This primitive is a predecessor of `\parameterdef` so we stick to a simple example.

```
\def\foo#1#2%
  {\integerdef\MyIndexOne\parameterindex\plusone % 1
   \integerdef\MyIndexTwo\parameterindex\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}

\def\oof#1%
  {<1:\expandparameter\MyIndexOne><1:\expandparameter\MyIndexOne>%
   #1%
   <2:\expandparameter\MyIndexTwo><2:\expandparameter\MyIndexTwo>}

\foo{A}{B}
```

In principle the whole parameter stack can be accessed but often one never knows if a specific macro is called nested. The original idea behind this primitive was tracing but it can also be used to avoid passing parameters along a chain of calls.

<1:A><1:A>P<2:B><2:B><1:A><1:A>Q<2:B><2:B><1:A><1:A>R<2:B><2:B>

97 \expandtoken

This primitive creates a token with a specific combination of catcode and character code. Because it assumes some knowledge of TEX we can show it using some `\expandafter` magic:

```
\expandafter\let\expandafter\temp\expandtoken 11 `X \meaning\temp
\expandafter\let\expandafter\temp\expandtoken 12 `X \meaning\temp
```

The meanings are:

```
the letter U+0058 X
the character U+0058 X
```

Using other catcodes is possible but the results of injecting them into the input directly (or here by injecting `\temp`) can be unexpected because of what TEX expects. You can get messages you normally won't get, for instance about unexpected alignment interference, which is a side effect of TEX using some catcode/character combinations as signals and there is no reason to change those internals. That said:

```
\xdef\tempA{\expandtoken  9 `X} \meaning\tempA
\xdef\tempB{\expandtoken 10 `X} \meaning\tempB
\xdef\tempC{\expandtoken 11 `X} \meaning\tempC
\xdef\tempD{\expandtoken 12 `X} \meaning\tempD
```

are all valid and from the meaning you cannot really deduce what's in there:

```
macro:X
```

```
macro:X
macro:X
macro:X
```

But you can be assured that:

```
[AB: \ifx\tempA\tempB Y\else N\fi]
[AC: \ifx\tempA\tempC Y\else N\fi]
[AD: \ifx\tempA\tempD Y\else N\fi]
[BC: \ifx\tempB\tempC Y\else N\fi]
[BD: \ifx\tempB\tempD Y\else N\fi]
[CD: \ifx\tempC\tempD Y\else N\fi]
```

makes clear that they're different: [AB: N] [AC: N] [AD: N] [BC: N] [BD: N] [CD: N], and in case you wonder, the characters with catcode 10 are spaces, while those with code 9 are ignored.

98 \expandtoks

This is a more efficient equivalent of `\the` applied to a token register, so:

```
\scratchtoks{just some tokens}
\edef\TestA{[\the       \scratchtoks]}
\edef\TestB{[\expandtoks\scratchtoks]}
[\the       \scratchtoks] [\TestA] \meaning\TestA
[\expandtoks\scratchtoks] [\TestB] \meaning\TestB
```

does the expected:

[just some tokens] [[just some tokens]] macro:[just some tokens]
[just some tokens] [[just some tokens]] macro:[just some tokens]

The `\expandtoken` primitive avoid a copy into the input when there is no need for it.

99 \explicitdiscretionary

This is the verbose alias for one of TEX's single character control sequences: `\-`.

100 \explicititaliccorrection

This is the verbose alias for one of TEX's single character control sequences: `\/`. Italic correction is a character property specific to TEX and the concept is not present in modern font technologies. There is a callback that hooks into this command so that a macro package can provide its own solution to this (or alternatively it can assign values to the italic correction field.

101 \explicitspace

This is the verbose alias for one of TEX's single character control sequences: `\`. A space is inserted with properties according the space related variables. There is look-back involved in order to deal with space factors.

When `\nospaces` is set to 1 no spaces are inserted, when its value is 2 a zero space is inserted.

102 \fi

This traditional primitive is part of the condition testing mechanism and ends a test. So, we have:

```
\ifsomething ... \else ... \fi
\ifsomething ... \or ... \or ... \else ... \fi
\ifsomething ... \orelse \ifsometing  ... \else ... \fi
\ifsomething ... \or ... \orelse \ifsometing  ... \else ... \fi
```

The `\orelse` is new in LuaMetaTEX and a continuation like we find in other programming languages (see later section).

103 \float

In addition to integers and dimensions, which are fixed 16.16 integer floats we also have ‘native’ floats, based on 32 bit posit unums.

```
\float0 = 123.456           \the\float0
\float2 = 123.456           \the\float0
\advance \float0 by 123.456 \the\float0
\advance \float0 by \float2 \the\float0
\divideby\float0 3          \the\float0
```

They come with the same kind of support as the other numeric data types:

```
123.45600032806396484
123.45600032806396484
246.91200065612792969
370.36800384521484375
123.45600128173828125
```

We leave the subtle differences between floats and dimensions to the user to investigate:

```
\dimen00 = 123.456pt          \the\dimen0
\dimen02 = 123.456pt          \the\dimen0
\advance \dimen0 by 123.456pt \the\dimen0
\advance \dimen0 by \dimen2   \the\dimen0
\divideby\dimen0 3            \the\dimen0
```

The nature of posits is that they are more accurate around zero (or smaller numbers in general).

```
123.456pt
123.456pt
246.91199pt
370.36798pt
123.456pt
```

This also works:

```
\float0=123.456e4
\float2=123.456    \multiply\float2 by 10000
\the\float0
```

```
\the\float2
```

The values are (as expected) the same:

```
1234560
1234560
```

104 \floatdef

This primitive defines a symbolic (macro) alias to a float register, just like `\countdef` and friends do.

105 \floatexpr

This is the companion of `\numexpr`, `\dimexpr` etc.

```
\scratchcounter 200
\the   \floatexpr 123.456/456.123     \relax
\the   \floatexpr 1.2*\scratchcounter \relax
\the   \floatexpr \scratchcounter/3   \relax
\number\floatexpr \scratchcounter/3   \relax
```

Watch the difference between `\the` and `\number`:

```
0.2706638324563503265
240
66.666669845581054 69
67
```

106 \font

This primitive is either a symbolic reference to the current font or in the perspective of an assignment is used to trigger a font definitions with a given name (`cs`) and specification. In LuaMetaTEX the assignment will trigger a callback that then handles the definition; in addition to the filename an optional size specifier is checked (`at` or `scaled`).

In LuaMetaTEX *all* font loading is delegated to Lua, and there is no loading code built in the engine. Also, instead of `\font` in ConTEXt one uses dedicated and more advanced font definition commands.

107 \formatname

It is in the name: `cont-en`, but we cheat here by only showing the filename and not the full path, which in a ConTEXt setup can span more than a line in this paragraph.

108 \frozen

You can define a macro as being frozen:

```
\frozen\def\MyMacro{...}
```

When you redefine this macro you get an error:

```
! You can't redefine a frozen macro.
```

This is a prefix like `\global` and it can be combined with other prefixes.[1]

109 \futurecsname

In order to make the repertoire of `def`, `let` and `futurelet` primitives complete we also have:

```
\futurecsname MyMacro:1\endcsname\MyAction
```

110 \futuredef

We elaborate on the example of using `\futurelet` in the previous section. Compare that one with the next:

```
\def\MySpecialToken{[}
\def\DoWhatever{\ifx\NextToken\MySpecialToken YES\else NOP\fi : }
\futurelet\NextToken\DoWhatever [A]\crlf
\futurelet\NextToken\DoWhatever (A)\par
```

This time we get:

NOP: [A]
NOP: (A)

It is for that reason that we now also have `\futuredef`:

```
\def\MySpecialToken{[}
\def\DoWhatever{\ifx\NextToken\MySpecialToken YES\else NOP\fi : }
\futuredef\NextToken\DoWhatever [A]\crlf
\futuredef\NextToken\DoWhatever (A)\par
```

So we're back to what we want:

YES: [A]
NOP: (A)

111 \futureexpand

This primitive can be used as an alternative to a `\futurelet` approach, which is where the name comes from.[2]

```
\def\variantone<#1>{(#1)}
\def\varianttwo#1{[#1]}
\futureexpand<\variantone\varianttwo<one>
\futureexpand<\variantone\varianttwo{two}
```

So, the next token determines which of the two variants is taken:

[1] The `\outer` and `\long` prefixes are no-ops in LuaMetaTeX and LuaTeX can be configured to ignore them.
[2] In the engine primitives that have similar behavior are grouped in commands that are then dealt with together, code wise.

(one) [two]

Because we look ahead there is some magic involved: spaces are ignored but when we have no match they are pushed back into the input. The next variant demonstrates this:

```
\def\variantone<#1>{(#1)}
\def\varianttwo{}
\def\temp{\futureexpand<\variantone\varianttwo}
[\temp <one>]
[\temp {two}]
[\expandafter\temp\space <one>]
[\expandafter\temp\space {two}]
```

This gives us:

[(one)] [two] [(one)] [two]

112 \futureexpandis

We assume that the previous section is read. This variant will not push back spaces, which permits a consistent approach i.e. the user can assume that macro always gobbles the spaces.

```
\def\variantone<#1>{(#1)}
\def\varianttwo{}
\def\temp{\futureexpandis<\variantone\varianttwo}
[\temp <one>]
[\temp {two}]
[\expandafter\temp\space <one>]
[\expandafter\temp\space {two}]
```

So, here no spaces are pushed back. This `is` in the name of this primitive means ‘ignore spaces’, but having that added to the name would have made the primitive even more verbose (after all, we also don't have `\expandeddef` but `\edef` and no `\globalexpandeddef` but `\xdef`.

[(one)] [two] [(one)] [two]

113 \futureexpandisap

This primitive is like the one in the previous section but also ignores par tokens, so `isap` means ‘ignore spaces and paragraphs’.

114 \futurelet

The original TEX primitive `\futurelet` can be used to create an alias to a next token, push it back into the input and then expand a given token.

```
\let\MySpecialTokenL[
\let\MySpecialTokenR] % nicer for checker
\def\DoWhatever{\ifx\NextToken\MySpecialTokenL YES\else NOP\fi : }
\futurelet\NextToken\DoWhatever [A]\crlf
\futurelet\NextToken\DoWhatever (A)\par
```

This is typically the kind of primitive that most users will never use because it expects a sane follow up handler (here `\DoWhatever`) and therefore is related to user interfacing.

YES: [A]
NOP: (A)

115 \gdef

The is the global companion of `\def`.

116 \gdefcsname

As with standard TEX we also define global ones:

```
\expandafter\gdef\csname MyMacro:1\endcsname{...}
             \gdefcsname MyMacro:1\endcsname{...}
```

117 \glet

This is the global companion of `\let`. The fact that it is not an original primitive is probably due to the expectation for it not it not being used (as) often (as in ConTEXt).

118 \gletcsname

Naturally LuaMetaTEX also provides a global variant:

```
\expandafter\global\expandafter\let\csname MyMacro:1\endcsname\relax
\expandafter                  \glet\csname MyMacro:1\endcsname\relax
                               \gletcsname MyMacro:1\endcsname\relax
```

So, here we save even more.

119 \glettonothing

This is the global companion of `\lettonothing`.

120 \global

This is one of the original prefixes that can be used when we define a macro of change some register.

```
\bgroup
       \def\MyMacroA{a}
\global\def\MyMacroB{a}
      \gdef\MyMacroC{a}
\egroup
```

The macro defined in the first line is forgotten when the groups is left. The second and third definition are both global and these definitions are retained.

121 \globaldefs

When set to a positive value, this internal integer will force all definitions to be global, and in a complex macro package that is not something a user will do unless it is very controlled.

122 \glueexpr

This is a more extensive variant of `\dimexpr` that also handles the optional stretch and shrink components.

123 \glueshrink

This returns the shrink component of a glue quantity. The result is a dimension so you need to apply `\the` when applicable.

124 \glueshrinkorder

This returns the shrink order of a glue quantity. The result is a integer so you need to apply `\the` when applicable.

125 \gluespecdef

A variant of `\integerdef` and `\dimensiondef` is:

```
\gluespecdef\MyGlue = 3pt plus 2pt minus 1pt
```

The properties are comparable to the ones described in the previous sections.

126 \gluestretch

This returns the stretch component of a glue quantity. The result is a dimension so you need to apply `\the` when applicable.

127 \gluestretchorder

This returns the stretch order of a glue quantity. The result is a integer so you need to apply `\the` when applicable.

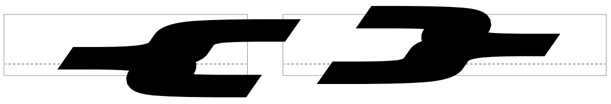
128 \glyph

This is a more extensive variant of `\char` that permits setting some properties if the injected character node.

```
\ruledhbox{\glyph
    scale 2000 xscale 9000 yscale 1200
    slant 700 weight 200
    xoffset 10pt yoffset -5pt left 10pt right 20pt
    123}
\quad
```

```
\ruledhbox{\glyph
    scale 2000 xscale 9000 yscale 1200
    slant 700 weight 200
    125}
```

In addition one can specify `font` (symbol), `id` (valid font id number), an `options` (bit set) and `raise`.



When no parameters are set, the current ones are used. More details and examples of usage can be found in the ConTEXt distribution.

129 \gtoksapp

This is the global variant of `\toksapp`.

130 \gtokspre

This is the global variant of `\tokspre`.

131 \if

This traditional TEX conditional checks if two character codes are the same. In order to understand unexpanded results it is good to know that internally TEX groups primitives in a way that serves the implementation. Each primitive has a command code and a character code, but only for real characters the name character code makes sense. This condition only really tests for character codes when we have a character, in all other cases, the result is true.

```
\def\A{A}\def\B{B} \chardef\C=`C \chardef\D=`D \def\AA{AA}

[\if AA   YES \else NOP \fi] [\if AB   YES \else NOP \fi]
[\if \A\B YES \else NOP \fi] [\if \A\A YES \else NOP \fi]
[\if \C\D YES \else NOP \fi] [\if \C\C YES \else NOP \fi]
[\if \count\dimen YES \else NOP \fi] [\if \AA\A YES \else NOP \fi]
```

The last example demonstrates that the tokens get expanded, which is why we get the extra A:

[YES] [NOP] [NOP] [YES] [YES] [YES] [YES] [AYES]

132 \ifabsdim

This test will negate negative dimensions before comparison, as in:

```
\def\TestA#1{\ifdim   #1<2pt too small\orelse\ifdim   #1>4pt too large\else okay\fi}
\def\TestB#1{\ifabsdim#1<2pt too small\orelse\ifabsdim#1>4pt too large\else okay\fi}

\TestA {1pt}\quad\TestA {3pt}\quad\TestA {5pt}\crlf
\TestB {1pt}\quad\TestB {3pt}\quad\TestB {5pt}\crlf
\TestB{-1pt}\quad\TestB{-3pt}\quad\TestB{-5pt}\par
```

So we get this:

```
too small   okay   too large
too small   okay   too large
too small   okay   too large
```

133 \ifabsfloat

This test will negate negative floats before comparison, as in:

```
\def\TestA#1{\iffloat   #1<2.46 small\orelse\iffloat   #1>4.68 large\else medium\fi}
\def\TestB#1{\ifabsfloat#1<2.46 small\orelse\ifabsfloat#1>4.68 large\else medium\fi}
```

```
\TestA {1.23}\quad\TestA {3.45}\quad\TestA {5.67}\crlf
\TestB {1.23}\quad\TestB {3.45}\quad\TestB {5.67}\crlf
\TestB{-1.23}\quad\TestB{-3.45}\quad\TestB{-5.67}\par
```

So we get this:

```
small   medium   large
small   medium   large
small   medium   large
```

134 \ifabsnum

This test will negate negative numbers before comparison, as in:

```
\def\TestA#1{\ifnum   #1<100 too small\orelse\ifnum   #1>200 too large\else okay\fi}
\def\TestB#1{\ifabsnum#1<100 too small\orelse\ifabsnum#1>200 too large\else okay\fi}
```

```
\TestA {10}\quad\TestA {150}\quad\TestA {210}\crlf
\TestB {10}\quad\TestB {150}\quad\TestB {210}\crlf
\TestB{-10}\quad\TestB{-150}\quad\TestB{-210}\par
```

Here we get the same result each time:

```
too small   okay   too large
too small   okay   too large
too small   okay   too large
```

135 \ifarguments

This is a variant of `\ifcase` were the selector is the number of arguments picked up. For example:

```
\def\MyMacro#1#2#3{\ifarguments\0\or1\or2\or3\else ?\fi} \MyMacro{A}{B}{C}
\def\MyMacro#1#0#3{\ifarguments\0\or1\or2\or3\else ?\fi} \MyMacro{A}{B}{C}
\def\MyMacro#1#-#2{\ifarguments\0\or1\or2\or3\else ?\fi} \MyMacro{A}{B}{C}\par
```

Watch the non counted, ignored, argument in the last case. Normally this test will be used in combination with `\ignorearguments`.

```
3  3  2
```

136 \ifboolean

This tests a number (register or equivalent) and any nonzero value represents `true`, which is nicer than using an **\unless\ifcase**.

137 \ifcase

This numeric TEX conditional takes a counter (literal, register, shortcut to a character, internal quantity) and goes to the branch that matches.

```
\ifcase 3 zero\or one\or two\or three\or four\else five or more\fi
```

Indeed: three equals three. In later sections we will see some LuaMetaTEX primitives that behave like an `\ifcase`.

138 \ifcat

Another traditional TEX primitive: what happens with what gets read in depends on the catcode of a character, think of characters marked to start math mode, or alphabetic characters (letters) versus other characters (like punctuation).

```
\def\A{A}\def\B{,} \chardef\C=`C \chardef\D=`, \def\AA{AA}
```

```
[\ifcat $!   YES \else NOP \fi] [\ifcat ()   YES \else NOP \fi]
[\ifcat AA   YES \else NOP \fi] [\ifcat AB   YES \else NOP \fi]
[\ifcat \A\B YES \else NOP \fi] [\ifcat \A\A YES \else NOP \fi]
[\ifcat \C\D YES \else NOP \fi] [\ifcat \C\C YES \else NOP \fi]
[\ifcat \count\dimen YES \else NOP \fi] [\ifcat \AA\A YES \else NOP \fi]
```

Close reading is needed here:

[NOP] [YES] [YES] [YES] [NOP] [YES] [YES] [YES] [YES] [AYES]

This traditional TEX condition as a well as the one in the previous section are hardly used in ConTEXt, if only because they expand what follows and we seldom need to compare characters.

139 \ifchkdim

A variant on the checker in the previous section is a dimension checker:

```
\ifchkdim oeps        \or okay\else error\fi\quad
\ifchkdim 12          \or okay\else error\fi\quad
\ifchkdim 12pt        \or okay\else error\fi\quad
\ifchkdim 12pt or more\or okay\else error\fi
```

We get:

error error okay okay

140 \ifchkdimension

COntrary to `\ifchkdim` this test doesn't accept trailing crap:

```
\ifchkdimension oeps        \or okay\else error\fi\quad
\ifchkdimension 12          \or okay\else error\fi\quad
\ifchkdimension 12pt        \or okay\else error\fi\quad
\ifchkdimension 12pt or more\or okay\else error\fi
```

reports:

error error okay error

141 \ifchknum

In ConTEXt there are quite some cases where a variable can have a number or a keyword indicating a symbolic name of a number or maybe even some special treatment. Checking if a valid number is given is possible to some extend, but a native checker makes much sense too. So here is one:

```
\ifchknum oeps        \or okay\else error\fi\quad
\ifchknum 12          \or okay\else error\fi\quad
\ifchknum 12pt        \or okay\else error\fi\quad
\ifchknum 12pt or more\or okay\else error\fi
```

The result is as expected:

error okay okay okay

142 \ifchknumber

This check is more restrictive than `\ifchknum` discussed in the previous section:

```
\ifchknumber oeps        \or okay\else error\fi\quad
\ifchknumber 12          \or okay\else error\fi\quad
\ifchknumber 12pt        \or okay\else error\fi\quad
\ifchknumber 12pt or more\or okay\else error\fi
```

Here we get:

error okay error error

143 \ifcmpdim

This conditional compares two dimensions and the resulting `\ifcase` reflects their relation:

```
[1pt 2pt : \ifcmpdim 1pt 2pt less\or equal\or more\fi]\quad
[1pt 1pt : \ifcmpdim 1pt 1pt less\or equal\or more\fi]\quad
[2pt 1pt : \ifcmpdim 2pt 1pt less\or equal\or more\fi]
```

This gives:

[1pt 2pt : less] [1pt 1pt : equal] [2pt 1pt : more]

144 \ifcmpnum

This conditional compares two numbers and the resulting `\ifcase` reflects their relation:

```
[1 2 : \ifcmpnum 1 2 less\or equal\or more\fi]\quad
[1 1 : \ifcmpnum 1 1 less\or equal\or more\fi]\quad
[2 1 : \ifcmpnum 2 1 less\or equal\or more\fi]
```

This gives:

[1 2 : less] [1 1 : equal] [2 1 : more]

145 \ifcondition

The conditionals in TEX are hard coded as primitives and although it might look like `\newif` creates one, it actually just defined three macros.

```
\newif\ifMyTest
\meaning\MyTesttrue  \crlf
\meaning\MyTestfalse \crlf
\meaning\ifMyTest    \crlf \MyTesttrue
\meaning\ifMyTest    \par
```

```
protected macro:\always \let \ifMyTest \iftrue
protected macro:\always \let \ifMyTest \iffalse
\iffalse
\iftrue
```

This means that when you say:

```
\ifMytest ... \else ... \fi
```

You actually have one of:

```
\iftrue  ... \else ... \fi
\iffalse ... \else ... \fi
```

and because these are proper conditions nesting them like:

```
\ifnum\scratchcounter > 0 \ifMyTest A\else B\fi \fi
```

will work out well too. This is not true for macros, so for instance:

```
\scratchcounter = 1
\unexpanded\def\ifMyTest{\iftrue}
\ifnum\scratchcounter > 0 \ifMyTest A\else B\fi \fi
```

will make a run fail with an error (or simply loop forever, depending on your code). This is where `\ifcondition` enters the picture:

```
\def\MyTest{\iftrue} \scratchcounter0
\ifnum\scratchcounter > 0
    \ifcondition\MyTest A\else B\fi
\else
    x
\fi
```

This primitive is seen as a proper condition when TEX is in “fast skipping unused branches” mode but when it is expanding a branch, it checks if the next expanded token is a proper tests and if so, it deals with that test, otherwise it fails. The main condition here is that the `\MyTest` macro expands to a proper true or false test, so, a definition like:

```
\def\MyTest{\ifnum\scratchcounter<10 }
```

is also okay. Now, is that neat or not?

146 \ifcsname

This is an ε-TEX conditional that complements the one on the previous section:

```
\expandafter\ifx\csname MyMacro\endcsname\relax ... \else ... \fi
            \ifcsname   MyMacro\endcsname       ... \else ... \fi
```

Here the first one has the side effect of defining the macro and defaulting it to `\relax`, while the second one doesn't do that. Juts think of checking a few million different names: the first one will deplete the hash table and probably string space too.

In LuaMetaTEX the construction stops when there is no letter or other character seen (TEX expands on the go so expandable macros are dealt with). Instead of an error message, the match is simply false and all tokens till the `\endcsname` are gobbled.

147 \ifcstok

A variant on the primitive mentioned in the previous section is one that operates on lists and macros:

```
\def\a{a} \def\b{b} \def\c{a}
```

This:

```
\ifcstok\a\b   Y\else N\fi\space
\ifcstok\a\c   Y\else N\fi\space
\ifcstok{\a}\c Y\else N\fi\space
\ifcstok{a}\c  Y\else N\fi
```

will give us: N Y Y Y.

148 \ifdefined

In traditional TEX checking for a macro to exist was a bit tricky and therefore ε-TEX introduced a convenient conditional. We can do this:

```
\ifx\MyMacro\undefined ... \else ... \fi
```

but that assumes that `\undefined` is indeed undefined. Another test often seen was this:

```
\expandafter\ifx\csname MyMacro\endcsname\relax ... \else ... \fi
```

Instead of comparing with `\undefined` we need to check with `\relax` because the control sequence is defined when not yet present and defaults to `\relax`. This is not pretty.

149 \ifdim

Dimensions can be compared with this traditional TEX primitive.

```
\scratchdimen=1pt \scratchcounter=65536
```

```
\ifdim\scratchdimen=\scratchcounter sp YES \else NOP\fi
\ifdim\scratchdimen=1               pt YES \else NOP\fi
```

The units are mandate:

YES YES

150 \ifdimexpression

The companion of the previous primitive is:

This matches when the result is non zero, and you can mix calculations and tests as with normal expressions. Contrary to the number variant units can be used and precision kicks in.

151 \ifdimval

This conditional is a variant on `\ifchkdim` and provides some more detailed information about the value:

```
[-12pt : \ifdimval-12pt\or negative\or zero\or positive\else error\fi]\quad
[0pt   : \ifdimval  0pt\or negative\or zero\or positive\else error\fi]\quad
[12pt  : \ifdimval 12pt\or negative\or zero\or positive\else error\fi]\quad
[oeps  : \ifdimval oeps\or negative\or zero\or positive\else error\fi]
```

This gives:

[-12pt : negative] [0pt : zero] [12pt : positive] [oeps : error]

152 \ifempty

This conditional checks if a control sequence is empty:

```
is \ifempty\MyMacro \else not \fi empty
```

It is basically a shortcut of:

```
is \ifx\MyMacro\empty \else not \fi empty
```

with:

```
\def\empty{}
```

Of course this is not empty at all:

```
\def\notempty#1{}
```

153 \iffalse

Here we have a traditional TEX conditional that is always false (therefore the same is true for any macro that is `\let` to this primitive).

154 \ifflags

This test primitive relates to the various flags that one can set on a control sequence in the perspective of overload protection and classification.

```
\protected\untraced\tolerant\def\foo[#1]{...#1...}
\permanent\constant         \def\oof{okay}
```

flag	`\foo`	`\oof`	flag	`\foo`	`\oof`
frozen	N	N	permanent	N	Y
immutable	N	N	mutable	N	N
noaligned	N	N	instance	N	N
untraced	Y	N	global	N	N
tolerant	Y	N	constant	N	Y
protected	Y	N	semiprotected	N	N

Instead of checking against a prefix you can test against a bitset made from:

0x1	frozen	0x2	permanent	0x4	immutable	0x8	primitive
0x10	mutable	0x20	noaligned	0x40	instance	0x80	untraced
0x100	global	0x200	tolerant	0x400	protected	0x800	overloaded
0x1000	aliased	0x2000	immediate	0x4000	conditional	0x8000	value
0x10000	semiprotected	0x20000	inherited	0x40000	constant	0x80000	deferred

155 \iffloat

This test does for floats what `\ifnum`, `\ifdim` do for numbers and dimensions: comparing two of them.

156 \iffontchar

This is an ε-TEX conditional. It takes a font identifier and a character number. In modern fonts simply checking could not be enough because complex font features can swap in other ones and their index can be anything. Also, a font mechanism can provide fallback fonts and characters, so don't rely on this one too much. It just reports true when the font passed to the frontend has a slot filled.

157 \ifhaschar

This one is a simplified variant of the above:

```
\ifhaschar !{this ! works} yes \else no \fi
```

and indeed we get: yes! Of course the spaces in this this example code are normally not present in such a test.

158 \ifhastok

This conditional looks for occurrences in token lists where each argument has to be a proper list.

```
\def\scratchtoks{x}

\ifhastoks{yz}         {xyz} Y\else N\fi\quad
\ifhastoks\scratchtoks {xyz} Y\else N\fi
```

We get:

Y   Y

159 \ifhastoks

This test compares two token lists. When a macro is passed it's meaning gets used.

```
\def\x  {x}
\def\xyz{xyz}

(\ifhastoks  {x}  {xyz}Y\else N\fi)\quad
(\ifhastoks {\x}  {xyz}Y\else N\fi)\quad
(\ifhastoks  \x   {xyz}Y\else N\fi)\quad
(\ifhastoks  {y}  {xyz}Y\else N\fi)\quad
(\ifhastoks {yz}  {xyz}Y\else N\fi)\quad
(\ifhastoks {yz} {\xyz}Y\else N\fi)
```

(Y)   (N)   (Y)   (Y)   (Y)   (N)

160 \ifhasxtoks

This primitive is like the one in the previous section but this time the given lists are expanded.

```
\def\x  {x}
\def\xyz{\x yz}

(\ifhasxtoks  {x}  {xyz}Y\else N\fi)\quad
(\ifhasxtoks {\x}  {xyz}Y\else N\fi)\quad
(\ifhastoks   \x   {xyz}Y\else N\fi)\quad
(\ifhasxtoks  {y}  {xyz}Y\else N\fi)\quad
(\ifhasxtoks {yz}  {xyz}Y\else N\fi)\quad
(\ifhasxtoks {yz} {\xyz}Y\else N\fi)
```

(Y)   (Y)   (Y)   (Y)   (Y)   (Y)

This primitive has some special properties.

```
\edef\+{\expandtoken 9 `+}

\ifhasxtoks {xy}   {xyz}Y\else N\fi\quad
\ifhasxtoks {x\+y} {xyz}Y\else N\fi
```

Here the first argument has a token that has category code ‘ignore’ which means that such a character will be skipped when seen. So the result is:

Y Y

This permits checks like these:

```
\edef\,{\expandtoken 9 `,}

\ifhasxtoks{\,x\,} {,x,y,z,}Y\else N\fi\quad
\ifhasxtoks{\,y\,} {,x,y,z,}Y\else N\fi\quad
\ifhasxtoks{\,z\,} {,x,y,z,}Y\else N\fi\quad
\ifhasxtoks{\,x\,}  {,xy,z,}Y\else N\fi
```

I admit that it needs a bit of a twisted mind to come up with this, but it works ok:

Y Y Y N

161 \ifhbox

This traditional conditional checks if a given box register or internal box variable represents a horizontal box,

162 \ifhmode

This traditional conditional checks we are in (restricted) horizontal mode.

163 \ifinalignment

As the name indicates, this primitive tests for being in an alignment. Roughly spoken, the engine is either in a state of align, handling text or dealing with math.

164 \ifincsname

This conditional is sort of obsolete and can be used to check if we're inside a `\csname` or `\ifcsname` construction. It's not used in ConTEXt.

165 \ifinner

This traditional one can be confusing. It is true when we are in restricted horizontal mode (a box), internal vertical mode (a box), or inline math mode.

```
test \ifhmode \ifinner INNER\fi HMODE\fi\crlf
\hbox{test \ifhmode \ifinner INNER \fi HMODE\fi} \par

\ifvmode \ifinner INNER\fi VMODE \fi\crlf
\vbox{\ifvmode \ifinner INNER \fi VMODE\fi} \crlf
\vbox{\ifinner INNER \ifvmode VMODE \fi \fi} \par
```

Watch the last line: because we typeset `INNER` we enter horizontal mode:

test HMODE
test INNER HMODE

VMODE
INNER VMODE
INNER

166 \ifintervaldim

This conditional is true when the intervals around the values of two dimensions overlap. The first dimension determines the interval.

```
[\ifintervaldim1pt 20pt 21pt \else no \fi overlap]
[\ifintervaldim1pt 18pt 20pt \else no \fi overlap]
```

So here: [overlap] [no overlap]

167 \ifintervalfloat

This one does with floats what we described under `\ifintervaldim`.

168 \ifintervalnum

This one does with integers what we described under `\ifintervaldim`.

169 \iflastnamedcs

When a `\csname` is constructed and succeeds the last one is remembered and can be accessed with `\lastnamedcs`. It can however be an undefined one. That state can be checked with this primitive. Of course it also works with the `\ifcsname` and `\begincsname` variants.

170 \ifmathparameter

This is an `\ifcase` where the value depends on if the given math parameter is zero, (`0`), set (`1`), or unset (`2`).

```
\ifmathparameter\Umathpunctclosespacing\displaystyle
    zero    \or
    nonzero \or
    unset   \fi
```

171 \ifmathstyle

This is a variant of `\ifcase` were the number is one of the seven possible styles: display, text, cramped text, script, cramped script, script script, cramped script script.

```
\ifmathstyle
  display
\or
  text
\or
  cramped text
```

```
\else
  normally smaller than text
\fi
```

172 \ifmmode

This traditional conditional checks we are in (inline or display) math mode mode.

173 \ifnum

This is a frequently used conditional: it compares two numbers where a number is anything that can be seen as such.

```
\scratchcounter=65 \chardef\A=65

\ifnum65=`A              YES \else NOP\fi
\ifnum\scratchcounter=65 YES \else NOP\fi
\ifnum\scratchcounter=\A YES \else NOP\fi
```

Unless a number is an unexpandable token it ends with a space or `\relax`, so when you end up in the true branch, you'd better check if TEX could determine where the number ends.

YES YES YES

On top of these ascii combinations, the engine also accepts some Unicode characters. This brings the full repertoire to:

character			operation
0x003C	<		less
0x003D	=		equal
0x003E	>		more
0x2208	∈		element of
0x2209	∉		not element of
0x2260	≠	!=	not equal
0x2264	≤	!>	less equal
0x2265	≥	!<	greater equal
0x2270	≰		not less equal
0x2271	≱		not greater equal

This also applied to `\ifdim` although in the case of element we discard the fractional part (read: divide the numeric representation by 65536).

174 \ifnumexpression

Here is an example of a conditional using expressions:

This matches when the result is non zero, and you can mix calculations and tests as with normal expressions.

175 \ifnumval

This conditional is a variant on `\ifchknum`. This time we get some more detail about the value:

```
[-12  : \ifnumval  -12\or negative\or zero\or positive\else error\fi]\quad
[0    : \ifnumval    0\or negative\or zero\or positive\else error\fi]\quad
[12   : \ifnumval   12\or negative\or zero\or positive\else error\fi]\quad
[oeps : \ifnumval oeps\or negative\or zero\or positive\else error\fi]
```

This gives:

[-12 : negative]   [0 : zero]   [12 : positive]   [oeps : error]

176 \ifodd

One reason for this condition to be around is that in a double sided layout we need test for being on an odd or even page. It scans for a number the same was as other primitives,

```
\ifodd65 YES \else NO\fi &
\ifodd`B YES \else NO\fi .
```

So: YES & NO.

177 \ifparameter

In a macro body `#1` is a reference to a parameter. You can check if one is set using a dedicated parameter condition:

```
\tolerant\def\foo[#1]#*[#2]%
  {\ifparameter#1\or one\else no one\fi\enspace
   \ifparameter#2\or two\else no two\fi\emspace}

\foo
\foo[1]
\foo[1][2]
```

We get:

no one no two   one no two   one two

178 \ifrelax

This is a convenient shortcut for `\ifx\relax` and the motivation for adding this one is (as with some others) to get less tracing.

179 \iftok

When you want to compare two arguments, the usual way to do this is the following:

```
\edef\tempA{#1}
\edef\tempb{#2}
```

```
\ifx\tempA\tempB
    the same
\else
    different
\fi
```

This works quite well but the fact that we need to define two macros can be considered a bit of a nuisance. It also makes macros that use this method to be not so called 'fully expandable'. The next one avoids both issues:

```
\iftok{#1}{#2}
    the same
\else
    different
\fi
```

Instead of direct list you can also pass registers, so given:

```
\scratchtoks{a}%
\toks0{a}%
```

This:

```
\iftok 0 \scratchtoks          Y\else N\fi\space
\iftok{a}\scratchtoks          Y\else N\fi\space
\iftok\scratchtoks\scratchtoks Y\else N\fi
```

gives: Y Y Y.

180 \iftrue

Here we have a traditional TEX conditional that is always true (therefore the same is true for any macro that is `\let` to this primitive).

181 \ifvbox

This traditional conditional checks if a given box register or internal box variable represents a vertical box,

182 \ifvmode

This traditional conditional checks we are in (internal) vertical mode.

183 \ifvoid

This traditional conditional checks if a given box register or internal box variable has any content.

184 \ifx

We use this traditional TEX conditional a lot in ConTEXt. Contrary to `\if` the two tokens that are compared are not expanded. This makes it possible to compare the meaning of two macros. Depending on the need, these macros can have their content expanded or not. A different number of parameters results in false.

Control sequences are identical when they have the same command code and character code. Because a `\let` macro is just a reference, both let macros are the same and equal to `\relax`:

```
\let\one\relax \let\two\relax
```

The same is true for other definitions that result in the same (primitive) or meaning encoded in the character field (think of `\chardefs` and so).

185 \ifzerodim

This tests for a dimen (dimension) being zero so we have:

```
\ifdim<dimension>=0pt
\ifzerodim<dimension>
\ifcase<dimension register>
```

186 \ifzerofloat

As the name indicated, this tests for a zero float value.

```
[\scratchfloat\zerofloat \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat\plusone   \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat 0.01      \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat 0.0e0     \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat \zeropoint\ifzerofloat\scratchfloat \else not \fi zero]
```

So: [zero] [not zero] [not zero] [zero] [zero]

187 \ifzeronum

This tests for a number (integer) being zero so we have these variants now:

```
\ifnum<integer or equivalent>=0
\ifzeronum<integer or equivalent>
\ifcase<integer or equivalent>
```

188 \ignorearguments

This primitive will quit argument scanning and start expansion of the body of a macro. The number of grabbed arguments can be tested as follows:

```
\def\MyMacro[#1][#2][#3]%
 {\ifarguments zero\or one\or two\or three \else hm\fi}

\MyMacro          \ignorearguments \quad
\MyMacro       [1]\ignorearguments \quad
\MyMacro    [1][2]\ignorearguments \quad
\MyMacro [1][2][3]\ignorearguments \par
```

zero one two three

Todo: explain optional delimiters.

189 \ignorenestedupto

This primitive gobbles following tokens and can deal with nested ‘environments’, for example:

```
\def\startfoo{\ignorenestedupto\startfoo\stopfoo}
```

```
(before
\startfoo
    test \startfoo test \stopfoo
   {test \startfoo test \stopfoo}
\stopfoo
after)
```

delivers:

(before after)

190 \ignorepars

This is a variant of `\ignorespaces`: following spaces *and* **`\par`** equivalent tokens are ignored, so for instance:

```
one + \ignorepars

two = \ignorepars \par
three
```

renders as: one + two = three. Traditionally TEX has been sensitive to `\par` tokens in some of its building blocks. This has to do with the fact that it could indicate a runaway argument which in the times of slower machines and terminals was best to catch early. In LuaMetaTEX we no longer have long macros and the mechanisms that are sensitive can be told to accept `\par` tokens (and ConTEXt set them such that this is the case).

191 \ignorerest

An example shows what this primitive does:

```
\tolerant\def\foo[#1]#*[#2]%
  {1234
   \ifparameter#1\or\else
     \expandafter\ignorerest
   \fi
   /#1/
   \ifparameter#2\or\else
     \expandafter\ignorerest
   \fi
   /#2/ }
```

```
\foo test \foo[456] test \foo[456][789] test
```

As this likely makes most sense in conditionals you need to make sure the current state is properly finished. Because `\expandafter` bumps the input state, here we actually quit two levels; this is because so called 'backed up text' is intercepted by this primitive.

1234 test 1234 /456/ test 1234 /456/ /789/ test

192 \ignorespaces

This traditional TEX primitive signals the scanner to ignore the following spaces, if any. We mention it because we show a companion in the next section.

193 \ignoreupto

This ignores everything upto the given token, so

```
\ignoreupto \foo not this but\foo only this
```

will give: only this.

194 \immediate

This one has no effect unless you intercept it at the Lua end and act upon it. In original TEX immediate is used in combination with read from and write to file operations. So, this is an old primitive with a new meaning.

195 \immutable

This prefix flags what follows as being frozen and is usually applied to for instance `\integerdef`'d control sequences. In that respect is is like `\permanent` but it makes it possible to distinguish quantities from macros.

196 \initcatcodetable

This initializes the catcode table with the given index.

197 \input

There are several ways to use this primitive:

```
\input  test
\input {test}
\input "test"
\input 'test'
```

When no suffix is given, TEX will assume the suffix is `.tex`. The second one is normally used.

198 \inputlineno

This integer holds the current linenumber but it is not always reliable.

199 \instance

This prefix flags a macro as an instance which is mostly relevant when a macro package want to categorize macros.

200 \integerdef

You can alias to a count (integer) register with `\countdef`:

```
\countdef\MyCount134
```

Afterwards the next two are equivalent:

```
\MyCount   = 99
\count1234 = 99
```

where `\MyCount` can be a bit more efficient because no index needs to be scanned. However, in terms of storage the value (here 99) is always in the register so `\MyCount` has to get there. This indirectness has the benefit that directly setting the value is reflected in the indirect accessor.

```
\integerdef\MyCount = 99
```

This primitive also defines a numeric equivalent but this time the number is stored with the equivalent. This means that:

```
\let\MyCopyOfCount = \MyCount
```

will store the *current* value of `\MyCount` in `\MyCopyOfCount` and changing either of them is not reflected in the other.

The usual `\advance`, `\multiply` and `\divide` can be used with these integers and they behave like any number. But compared to registers they are actually more a constant.

201 \interactionmode

This internal integer can be used to set or query the current interaction mode:

\batchmode	0	omits all stops and terminal output
\nonstopmode	1	omits all stops
\scrollmode	2	omits error stops
\errorstopmode	3	stops at every opportunity to interact

202 \jobname

This gives the current job name without suffix: `primitives`.

203 \lastarguments

```
\def\MyMacro    #1{\the\lastarguments (#1) }          \MyMacro{1}       \crlf
\def\MyMacro  #1#2{\the\lastarguments (#1) (#2)}      \MyMacro{1}{2}    \crlf
\def\MyMacro#1#2#3{\the\lastarguments (#1) (#2) (#3)} \MyMacro{1}{2}{3} \par

\def\MyMacro    #1{(#1)           \the\lastarguments} \MyMacro{1}       \crlf
\def\MyMacro  #1#2{(#1) (#2)      \the\lastarguments} \MyMacro{1}{2}    \crlf
\def\MyMacro#1#2#3{(#1) (#2) (#3) \the\lastarguments} \MyMacro{1}{2}{3} \par
```

The value of `\lastarguments` can only be trusted in the expansion until another macro is seen and expanded. For instance in these examples, as soon as a character (like the left parenthesis) is seen, horizontal mode is entered and `\everypar` is expanded which in turn can involve macros. You can see that in the second block (that is: unless we changed `\everypar` in the meantime).

```
1(1)
2(1) (2)
3(1) (2) (3)

(1) 0
(1) (2) 2
(1) (2) (3) 3
```

204 \lastloopiterator

In addition to `\currentloopiterator` we have a variant that stores the value in case an unexpanded loop is used:

```
\localcontrolledrepeat 8 { [\the\currentloopiterator\eq\the\lastloopiterator] }
\expandedrepeat        8 { [\the\currentloopiterator\eq\the\lastloopiterator] }
\unexpandedrepeat      8 { [\the\currentloopiterator\ne\the\lastloopiterator] }
```

```
[1=1] [2=2] [3=3] [4=4] [5=5] [6=6] [7=7] [8=8]
[1=1] [2=2] [3=3] [4=4] [5=5] [6=6] [7=7] [8=8]
[0≠1] [0≠2] [0≠3] [0≠4] [0≠5] [0≠6] [0≠7] [0≠8]
```

205 \lastnamedcs

The example code in the previous section has some redundancy, in the sense that there to be looked up control sequence name `mymacro` is assembled twice. This is no big deal in a traditional eight bit TEX but in a Unicode engine multi-byte sequences demand some more processing (although it is unlikely that control sequences have many multi-byte utf8 characters).

```
\ifcsname mymacro\endcsname
    \csname mymacro\endcsname
\fi
```

Instead we can say:

```
\ifcsname mymacro\endcsname
    \lastnamedcs
```

```
\fi
```

Although there can be some performance benefits another advantage is that it uses less tokens and parsing. It might even look nicer.

206 \letcharcode

Assigning a meaning to an active character can sometimes be a bit cumbersome; think of using some documented uppercase magic that one tends to forget as it's used only a few times and then never looked at again. So we have this:

```
{\letcharcode 65 1 \catcode 65 13 A : \meaning A}\crlf
{\letcharcode 65 2 \catcode 65 13 A : \meaning A}\par
```

here we define `A` as an active charcter with meaning `1` in the first line and `2` in the second.

```
1 : the character U+0031 1
2 : the character U+0032 2
```

Normally one will assign a control sequence:

```
{\letcharcode 66 \bf \catcode 66 13 {B   bold}: \meaning B}\crlf
{\letcharcode 73 \it \catcode 73 13 {I italic}: \meaning I}\par
```

Of course `\bf` and `\it` are ConTEXt specific commands:

```
bold: protected macro:\ifmmode \expandafter \mathbf \else \expandafter \normalbf \fi
italic: protected macro:\ifmmode \expandafter \mathit \else \expandafter \normalit \fi
```

207 \letcsname

It is easy to see that we save two tokens when we use this primitive. As with the `..defcs..` variants it also saves a push back of the composed macro name.

```
\expandafter\let\csname MyMacro:1\endcsname\relax
             \letcsname MyMacro:1\endcsname\relax
```

208 \letfrozen

You can explicitly freeze an unfrozen macro:

```
\def\MyMacro{...}
\letfrozen\MyMacro
```

A redefinition will now give:

```
! You can't redefine a frozen macro.
```

209 \letprotected

Say that you have these definitions:

```
             \def  \MyMacroA{alpha}
\protected   \def  \MyMacroB{beta}
             \edef \MyMacroC{\MyMacroA\MyMacroB}
\letprotected      \MyMacroA
             \edef \MyMacroD{\MyMacroA\MyMacroB}
\meaning           \MyMacroC\crlf
\meaning           \MyMacroD\par
```

The typeset meaning in this example is:

```
macro:alpha\MyMacroB
macro:\MyMacroA \MyMacroB
```

210 \lettolastnamedcs

The `\lastnamedcs` primitive is somewhat special as it is a (possible) reference to a control sequence which is why we have a dedicated variant of `\let`.

```
\csname relax\endcsname\let                        \foo\lastnamedcs \meaning\foo
\csname relax\endcsname\expandafter\let\expandafter \oof\lastnamedcs \meaning\oof
\csname relax\endcsname\lettolastnamedcs            \ofo             \meaning\ofo
```

These give the following where the first one obviously is not doing what we want and the second one is kind of cumbersome.

\lastnamedcs
\relax
\relax

211 \lettonothing

This one let's a control sequence to nothing. Assuming that **\empty** is indeed empty, these two lines are equivalent.

```
\let         \foo\empty
\lettonothing\oof
```

212 \localcontrol

This primitive takes a single token:

```
\edef\testa{\scratchcounter123 \the\scratchcounter}
\edef\testc{\testa \the\scratchcounter}
\edef\testd{\localcontrol\testa \the\scratchcounter}
```

The three meanings are:

123

```
\testa  macro:\scratchcounter 123 123
\testc  macro:\scratchcounter 123 123123
\testd  macro:123
```

The `\localcontrol` makes that the following token gets expanded so we don't see the yet to be expanded assignment show up in the macro body.

213 \localcontrolled

The previously described local control feature comes with two extra helpers. The `\localcontrolled` primitive takes a token list and wraps this into a local control sidetrack. For example:

```
\edef\testa{\scratchcounter123 \the\scratchcounter}
\edef\testb{\localcontrolled{\scratchcounter123}\the\scratchcounter}
```

The two meanings are:

```
\testa  macro:\scratchcounter 123 123
\testb  macro:123
```

The assignment is applied immediately in the expanded definition.

214 \localcontrolledendless

As the name indicates this will loop forever. You need to explicitly quit the loop with `\quitloop` or `\quitloopnow`. The first quitter aborts the loop at the start of a next iteration, the second one tries to exit immediately, but is sensitive for interference with for instance nested conditionals.

215 \localcontrolledloop

As with more of the primitives discussed here, there is a manual in the ‘lowlevel’ subset that goes into more detail. So, here a simple example has to do:

```
\localcontrolledloop 1 100 1 {%
    \ifnum\currentloopiterator>6\relax
        \quitloop
    \else
        [\number\currentloopnesting:\number\currentloopiterator]
        \localcontrolledloop 1 8 1 {%
            (\number\currentloopnesting:\number\currentloopiterator)
        }\par
    \fi
}
```

Here we see the main loop primitive being used nested. The code shows how we can `\quitloop` and have access to the `\currentloopiterator` as well as the nesting depth `\currentloopnesting`.

```
[1:1] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:2] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:3] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:4] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:5] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:6] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
```

Be aware of the fact that `\quitloop` will end the loop at the *next* iteration so any content after it will show up. Normally this one will be issued in a condition and we want to end that properly. Also keep in mind that because we use local control (a nested TEX expansion loop) anything you feed back can be injected out of order.

The three numbers can be separated by an equal sign which is a trick to avoid look ahead issues that can result from multiple serialized numbers without spaces that indicate the end of sequence of digits.

216 \localcontrolledrepeat

This one takes one instead three arguments which looks a bit better in simple looping.

217 \long

This original prefix gave the macro being defined the property that it could not have `\par` (or the often equivalent empty lines) in its arguments. It was mostly a protection against a forgotten right curly brace, resulting in a so called run-away argument. That mattered on a paper terminal or slow system where such a situation should be catched early. In LuaTEX it was already optional, and in LuaMetaTEX we dropped this feature completely (so that we could introduce others).

218 \lowercase

This token processor converts character tokens to their lowercase counterparts as defined per `\lccode`. In order to permit dirty tricks active characters are also processed. We don't really use this primitive in ConTEXt, but for consistency we let it respond to `\expand`:[3]

```
\edef           \foo       {\lowercase{tex TeX \TEX}} \meaningless\foo
\lowercase{\edef\foo                  {tex TeX \TEX}} \meaningless\foo
\edef           \foo{\expand\lowercase{tex TeX \TEX}} \meaningless\foo
```

Watch how `\lowercase` is not expandable but can be forced to. Of course, as the logo macro is protected the TEX logo remains mixed case.

\lowercase {tex TeX \TEX }
tex tex \TEX
tex tex \TEX

219 \luaescapestring

This command converts the given (token) list into something that is acceptable for Lua. It is inherited from LuaTEX and not used in ConTEXt.

```
\directlua { tex.print ("\luaescapestring {{\tt This is a "test".}}") }
```

Results in: `This is a "test".` (Watch the grouping.)

220 \luatexbanner

This gives: `This is LuaMetaTeX, Version 2.10.11.`

[3] Instead of providing **\lowercased** and **\uppercased** primitives that would clash with macros anyway.

221 \luatexrevision

This is an integer. The current value is: `10`.

222 \luatexversion

This is an integer. The current value is: `210`.

223 \mathgroupingmode

Normally a `{}` or `\bgroup-\egroup` pair in math create a math list. However, users are accustomed to using it also for grouping and then a list being created might not be what a user wants. As an alternative to the more verbose `\begingroup-\endgroup` or even less sensitive `\beginmathgroup-\endmathgroup` you can set the math grouping mode to a non zero value which makes curly braces (and the aliases) behave as expected.

224 \meaning

We start with a primitive that will be used in the following sections. The reported meaning can look a bit different than the one reported by other engines which is a side effect of additional properties and more extensive argument parsing.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaning\foo
```

tolerant protected macro:[#1]#*[#2]->(#1)(#2)

225 \meaningasis

Although it is not really round trip with the original due to information being lost this primitive tries to return an equivalent definition.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningasis\foo
```

\permanent \tolerant \protected \def \foo [#1]#*[#2]{(#1)(#2)}

226 \meaningful

This one reports a bit less than `\meaningful`.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningful\foo
```

permanent tolerant protected macro

227 \meaningfull

This one reports a bit more than `\meaning`.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningfull\foo
```

permanent tolerant protected macro:[#1]#*[#2]->(#1)(#2)

228 \meaningles

This one reports a bit less than \meaningless.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningles\foo
```

[#1]#*[#2]

229 \meaningless

This one reports a bit less than \meaning.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningless\foo
```

[#1]#*[#2]->(#1)(#2)

230 \mugluespecdef

A variant of \gluespecdef that expects mu units is:

```
\mugluespecdef\MyGlue = 3mu plus 2mu minus 1mu
```

The properties are comparable to the ones described in the previous sections.

231 \multiply

The given quantity is multiplied by the given integer (that can be preceded by the keyword ‘by’, like:

```
\scratchdimen=10pt \multiply\scratchdimen by 3
```

232 \multiplyby

This is slightly more efficient variant of \multiply that doesn't look for by. See previous section.

233 \nonstopmode

This directive omits all stops.

234 \norelax

The rationale for this command can be shown by a few examples:

```
\dimen0 1pt \dimen2 1pt \dimen4 2pt
\edef\testa{\ifdim\dimen0=\dimen2\norelax N\else Y\fi}
\edef\testb{\ifdim\dimen0=\dimen2\relax   N\else Y\fi}
\edef\testc{\ifdim\dimen0=\dimen4\norelax N\else Y\fi}
\edef\testd{\ifdim\dimen0=\dimen4\relax   N\else Y\fi}
\edef\teste{\norelax}
```

The five meanings are:

```
\testa  macro:N
\testb  macro:\relax N
\testc  macro:Y
\testd  macro:Y
\teste  macro:
```

So, the `\norelax` acts like `\relax` but is not pushed back as usual (in some cases).

235 \nospaces

When `\nospaces` is set to 1 no spaces are inserted, when its value is 2 a zero space is inserted. The default value is 0 which means that spaces become glue with properties depending on the font, specific parameters and/or space factors determined preceding characters.

236 \number

This TEX primitive serializes the next token into a number, assuming that it is indeed a number, like

```
\number`A
\number65
\number\scratchcounter
```

For counters and such the `\the` primitive does the same, but when you're not sure if what follows is a verbose number or (for instance) a counter the `\number` primitive is a safer bet, because `\the 65` will not work.

237 \numericscale

This primitive can best be explained by a few examples:

```
\the\numericscale 1323
\the\numericscale 1323.0
\the\numericscale 1.323
\the\numericscale 13.23
```

In several places TEX uses a scale but due to the lack of floats it then uses 1000 as 1.0 replacement. This primitive can be used for ‘real’ scales:

1323000
1323000
1323
13230

238 \numericscaled

This is a variant if `\numericscale`:

```
\scratchcounter 1000
\the\numericscaled 1323   \scratchcounter
\the\numericscaled 1323.0 \scratchcounter
```

```
\the\numericscaled 1.323  \scratchcounter
\the\numericscaled 13.23  \scratchcounter
```

The second number gets multiplied by the first fraction:

1323000
1323000
1323
13230

239 \numexpr

This primitive was introduced by ε-TEX and supports a simple expression syntax:

```
\the\numexpr 10 * (1 + 2 - 5) / 2 \relax
```

gives: -10. You can mix in symbolic integers and dimensions.

240 \numexpression

The normal `\numexpr` primitive understands the `+`, `-`, `*` and `/` operators but in LuaMetaTEX we also can use `:` for a non rounded integer division (think of Lua's `//`). if you want more than that, you can use the new expression primitive where you can use the following operators.

add	`+`	
subtract	`-`	
multiply	`*`	
divide	`/ :`	
mod	`%`	`mod`
band	`&`	`band`
bxor	`^`	`bxor`
bor	`| v`	`bor`
and	`&&`	`and`
or	`||`	`or`
setbit	`<undecided>`	`bset`
resetbit	`<undecided>`	`breset`
left	`<<`	
right	`>>`	
less	`<`	
lessequal	`<=`	
equal	`= ==`	
moreequal	`>=`	
more	`>`	
unequal	`<> != ~=`	
not	`! ~`	`not`

An example of the verbose bitwise operators is:

```
\scratchcounter = \numexpression
    "00000 bor "00001 bor "00020 bor "00400 bor "08000 bor "F0000
\relax
```

In the table you might have notices that some operators have equivalents. This makes the scanner a bit less sensitive for catcode regimes.

When `\tracingexpressions` is set to one or higher the intermediate 'reverse polish notation' stack that is used for the calculation is shown, for instance:

```
4:8: {numexpression rpn: 2 5 > 4 5 > and}
```

When you want the output on your console, you need to say:

```
\tracingexpressions 1
\tracingonline      1
```

241 \or

This traditional primitive is part of the condition testing mechanism and relates to an `\ifcase` test (or a similar test to be introduced in later sections). Depending on the value, TEX will do a fast scanning till the right `\or` is seen, then it will continue expanding till it sees a `\or` or `\else` or `\orelse` (to be discussed later). It will then do a fast skipping pass till it sees an `\fi`.

242 \orelse

This primitive provides a convenient way to flatten your conditional tests. So instead of

```
\ifnum\scratchcounter<-10
    too small
\else\ifnum\scratchcounter>10
    too large
\else
    just right
\fi\fi
```

You can say this:

```
\ifnum\scratchcounter<-10
    too small
\orelse\ifnum\scratchcounter>10
    too large
\else
    just right
\fi
```

You can mix tests and even the case variants will work in most cases[4]

```
\ifcase\scratchcounter          zero
\or                             one
\or                             two
\orelse\ifnum\scratchcounter<10 less than ten
\else                           ten or more
```

[4] I just play safe because there are corner cases that might not work yet.

```
\fi
```

Performance wise there are no real benefits although in principle there is a bit less housekeeping involved than with nested checks. However you might like this:

```
\ifnum\scratchcounter<-10
    \expandafter\toosmall
\orelse\ifnum\scratchcounter>10
    \expandafter\toolarge
\else
    \expandafter\justright
\fi
```

over:

```
\ifnum\scratchcounter<-10
    \expandafter\toosmall
\else\ifnum\scratchcounter>10
    \expandafter\expandafter\expandafter\toolarge
\else
    \expandafter\expandafter\expandafter\justright
\fi\fi
```

or the more ConTEXt specific:

```
\ifnum\scratchcounter<-10
    \expandafter\toosmall
\else\ifnum\scratchcounter>10
    \doubleexpandafter\toolarge
\else
    \doubleexpandafter\justright
\fi\fi
```

But then, some TEXies like complex and obscure code and throwing away working old code that took ages to perfect and get working and also showed that one masters TEX might hurt.

243 \orunless

This is the negated variant of `\orelse` (prefixing that one with `\unless` doesn't work well.

244 \outer

An outer macro is one that can only be used at the outer level. This property is no longer supported. Like `\long`, the `\outer` prefix is now an no-op (and we don't expect this to have unfortunate side effects).

245 \parameterdef

Here is an example of binding a variable to a parameter. The alternative is of course to use an `\edef`.

```
\def\foo#1#2%
```

```
  {\parameterdef\MyIndexOne\plusone % 1
   \parameterdef\MyIndexTwo\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}

\def\oof#1%
  {<1:\MyIndexOne><1:\MyIndexOne>%
   #1%
   <2:\MyIndexTwo><2:\MyIndexTwo>}

\foo{A}{B}
```

The outcome is:

<1:A><1:A>P<2:B><2:B><1:A><1:A>Q<2:B><2:B><1:A><1:A>R<2:B><2:B>

246 \parameterindex

This gives the zero based position on the parameter stack. One reason for introducing `\parameterdef` is that the position remains abstract so there we don't need to use `\parameterindex`.

247 \parametermark

This is an equivalent for `#`.

248 \parametermode

Setting this internal integer to a positive value (best use 1 because future versions might use bit set) will enable the usage of `#` for escaped in the main text and body of macros.

249 \positdef

The engine uses 32 bit integers for various purposes and has no (real) concept of a floating point quantity. We get around this by providing a floating point data type based on 32 bit unums (posits). These have the advantage over native floats of more precision in the lower ranges but at the cost of a software implementation.

The `\positdef` primitive is the floating point variant of `\integerdef` and `\dimensiondef`: an efficient way to implement named quantities other than registers.

```
\positdef     \MyFloatA 5.678
\positdef     \MyFloatB 567.8
[\the\MyFloatA] [\todimension\MyFloatA] [\tointeger\MyFloatA]
[\the\MyFloatB] [\todimension\MyFloatB] [\tointeger\MyFloatB]
```

For practical reasons we can map posit (or float) onto an integer or dimension:

[5.6780000030994415283] [5.678pt] [6]
[567.8000030517578125] [567.80005pt] [568]

250 \previousloopiterator

```
\edef\testA{
    \expandedrepeat 2 {%
        \expandedrepeat 3 {%
            (\the\previousloopiterator1:\the\currentloopiterator)
        }%
    }%
}
\edef\testB{
    \expandedrepeat 2 {%
        \expandedrepeat 3 {%
            (#P:#I) % #G is two levels up
        }%
    }%
}
```

These give the same result:

```
\def \testA { (1:1) (1:2) (1:3) (2:1) (2:2) (2:3) }
\def \testB { (1:1) (1:2) (1:3) (2:1) (2:2) (2:3) }
```

The number indicates the number of levels we go up the loop chain.

251 \protected

A protected macro is one that doesn't get expanded unless it is time to do so. For instance, inside an `\edef` it just stays what it is. It often makes sense to pass macros as-is to (multi-pass) file (for tables of contents).

In ConTEXt we use either `\protected` or `\unexpanded` because the later was the command we used to achieve the same results before ε-TEX introduced this protection primitive. Originally the `\protected` macro was also defined but it has been dropped.

252 \protecteddetokenize

This is a variant of `\protecteddetokenize` that uses some escapes encoded as body parameters, like `#H` for a hash.

253 \protectedexpandeddetokenize

This is a variant of `\expandeddetokenize` that uses some escapes encoded as body parameters, like `#H` for a hash.

254 \pxdimen

The current numeric value of this dimension is 65781, 1.00374pt: one `bp`. We kept it around because it was introduced in pdfTEX and made it into LuaTEX, where it relates to the resolution of included images. In ConTEXt it is not used.

255 \quitloop

There are several loop primitives and they can be quit with `\quitloop` at the next the *next* iteration. An immediate quit is possible with `\quitloopnow`. An example is given with `\localcontrolledloop`.

256 \quitloopnow

There are several loop primitives and they can be quit with `\quitloopnow` at the spot.

257 \rdivide

This is variant of `\divide` that rounds the result. For integers the result is the same as `\edivide`.

```
\the\dimexpr .4999pt                      : 2 \relax            =.24994pt
\the\dimexpr .4999pt                      / 2 \relax            =.24995pt
\scratchdimen.4999pt \divide \scratchdimen 2 \the\scratchdimen =.24994pt
\scratchdimen.4999pt \edivide\scratchdimen 2 \the\scratchdimen =.24995pt
\scratchdimen 4999pt \rdivide\scratchdimen 2 \the\scratchdimen =2500.0pt
\scratchdimen 5000pt \rdivide\scratchdimen 2 \the\scratchdimen =2500.0pt
```

```
\the\numexpr   1001                        : 2 \relax             =500
\the\numexpr   1001                        / 2 \relax             =501
\scratchcounter1001 \divide \scratchcounter 2 \the\scratchcounter=500
\scratchcounter1001 \edivide\scratchcounter 2 \the\scratchcounter=501
\scratchcounter1001 \rdivide\scratchcounter 2 \the\scratchcounter=501
```

0.24994pt=.24994pt
0.24995pt=.24995pt
0.24994pt=.24994pt
0.24995pt=.24995pt
2500.0pt=2500.0pt
2500.0pt=2500.0pt

500=500
501=501
500=500
501=501
501=501

258 \rdivideby

This is the by-less companion to `\rdivide`.

259 \relax

This primitive does nothing and is often used to end a verbose number or dimension in a comparison, for example:

```
\ifnum \scratchcounter = 123\relax
```

which prevents a lookahead. A variant would be:

```
\ifnum \scratchcounter = 123 %
```

assuming that spaces are not ignored. Another application is finishing an expression like `\numexpr` or `\dimexpr`. I is also used to prevent lookahead in cases like:

```
\vrule height 3pt depth 2pt width 5pt\relax
\hskip 5pt plus 3pt minus 2pt\relax
```

Because `\relax` is not expandable the following:

```
\edef\foo{\relax}   \meaningfull\foo
\edef\oof{\norelax} \meaningfull\oof
```

gives this:

macro:\relax
macro:

A `\norelax` disappears here but in the previously mentioned scenarios it has the same function as `\relax`. It will not be pushed back either in cases where a lookahead demands that.

260 \retained

When a value is assigned inside a group TEX pushes the current value on the save stack in order to be able to restore the original value after the group has ended. You can reach over a group by using the `\global` prefix. A mix between local and global assignments can be achieved with the `\retained` primitive.

```
\MyDim 15pt \bgroup \the\MyDim \space
\bgroup
    \bgroup
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
    \bgroup
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
\egroup
\egroup \the\MyDim
```

```
\MyDim 15pt \bgroup \the\MyDim \space
\bgroup
    \bgroup
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
    \bgroup
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
\egroup
\egroup \the\MyDim

\MyDim 15pt \bgroup \the\MyDim \space
    \constrained\MyDim\zeropoint
    \bgroup
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
    \bgroup
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
\egroup \the\MyDim
```

These lines result in:

```
15.0pt 25.0pt 25.0pt 25.0pt 25.0pt 15.0pt
15.0pt 25.0pt 35.0pt 45.0pt 55.0pt 55.0pt
15.0pt 10.0pt 20.0pt 30.0pt 40.0pt 15.0pt
```

Because LuaMetaTEX avoids redundant stack entries and reassignments this mechanism is a bit fragile but the `\constrained` prefix makes sure that we do have a stack entry. If it is needed depends on the usage pattern.

261 \retokenized

This is a companion of `\tokenized` that accepts a catcode table, so the whole repertoire is:

```
\tokenized                            {test $x$ test: current}
\tokenized   catcodetable \ctxcatcodes {test $x$ test: context}
\tokenized   catcodetable \vrbcatcodes {test $x$ test: verbatim}
\retokenized              \ctxcatcodes {test $x$ test: context}
\retokenized              \vrbcatcodes {test $x$ test: verbatim}
```

Here we pass the numbers known to ConTEXt and get:

test x test: current
test x test: context
test \$x\$ test: verbatim
test x test: context
test \$x\$ test: verbatim

262 \romannumeral

This converts a number into a sequence of characters representing a roman numeral. Because the Romans had no zero, a zero will give no output, a fact that is sometimes used for hacks and showing off ones macro coding capabilities. A large number will for sure result in a long string because after thousand we start duplicating.

263 \savecatcodetable

This primitive stores the currently set catcodes in the current table.

264 \scantextokens

This primitive scans the input as if it comes from a file. In the next examples the \detokenize primitive turns tokenized code into verbatim code that is similar to what is read from a file.

```
\edef\whatever{\detokenize{This is {\bf bold} and this is not.}}
\detokenize   {This is {\bf bold} and this is not.}\crlf
\scantextokens{This is {\bf bold} and this is not.}\crlf
\scantextokens{\whatever}\crlf
\scantextokens\expandafter{\whatever}\par
```

This primitive does not have the end-of-file side effects of its precursor \scantokens.

This is {\bf bold} and this is not.
This is **bold** and this is not.
This is {\bf bold} and this is not.
This is **bold** and this is not.

265 \scantokens

Just forget about this ε-TEX primnitive, just take the one in the next section.

266 \scrollmode

This directive omits error stops.

267 \semiexpand

This command expands the next macro when it is protected with \semprotected. See that primitive there for an example.

268 \semiexpanded

This command expands the tokens in the given list including the macros protected by with \semprotected. See that primitive there for an example.

269 \semiprotected

The working of this prefix can best be explained with an example. We define a few macros first:

```
              \def\TestA{A}
\semiprotected\def\TestB{B}
    \protected\def\TestC{C}

\edef\TestD{\TestA           \TestB           \TestC}
```

```
\edef\TestE{\TestA\semiexpand\TestB\semiexpand\TestC}
\edef\TestF{\TestA\expand    \TestB\expand    \TestC}

\edef\TestG{\normalexpanded    {\TestA\TestB\TestC}}
\edef\TestH{\normalsemiexpanded{\TestA\TestB\TestC}}
```

The meaning of the macros that are made from the other three are:

Here we use the `\normal..` variants because (currently) we still have the macro with the `\expanded` in the ConTEXt core.

```
A\TestB \TestC
AB\TestC
ABC
A\TestB \TestC
AB\TestC
```

270 \skip

This is the accessor for an indexed glue register.

271 \string

We mention this original primitive because of the one in the next section. It expands the next token or control sequence as if it was just entered, so normally a control sequence becomes a backslash followed by characters and a space.

272 \swapcsvalues

Because we mention some `def` and `let` primitives here, it makes sense to also mention a primitive that will swap two values (meanings). This one has to be used with care. Of course that what gets swapped has to be of the same type (or at least similar enough not to cause issues). Registers for instance store their values in the token, but as soon as we are dealing with token lists we also need to keep an eye on reference counting. So, to some extend this is an experimental feature.

273 \the

The `\the` primitive serializes the following token, when applicable: integers, dimensions, token registers, special quantities, etc. The catcodes of the result will be according to the current settings, so in `\the\dimen0`, the `pt` will have catcode ‘letter’ and the number and period will become ‘other’.

274 \thewithoutunit

The `\the` primitive, when applied to a dimension variable, adds a `pt` unit. because dimensions are the only traditional unit with a fractional part they are sometimes used as pseudo floats in which case `\thewithoutunit` can be used to avoid the unit. This is more convenient than stripping it off afterwards (via an expandable macro).

275 \todimension

The following code gives this: 1234.0pt and like its numeric counterparts accepts anything that resembles a number this one goes beyond (user, internal or pseudo) registers values too.

```
\scratchdimen = 1234pt \todimension\scratchdimen
```

276 \tohexadecimal

The following code gives this: 4D2 with uppercase letters.

```
\scratchcounter = 1234 \tohexadecimal\scratchcounter
```

277 \tointeger

The following code gives this: 1234 and is equivalent to `\number`.

```
\scratchcounter = 1234 \tointeger\scratchcounter
```

278 \tokenized

Just as `\expanded` has a counterpart `\unexpanded`, it makes sense to give `\detokenize` a companion:

```
\edef\foo{\detokenize{\inframed{foo}}}
\edef\oof{\detokenize{\inframed{oof}}}

\meaning\foo \crlf \dontleavehmode\foo

\edef\foo{\tokenized{\foo\foo}}

\meaning\foo \crlf \dontleavehmode\foo

\dontleavehmode\tokenized{\foo\oof}
```

```
macro:\inframed {foo}
\inframed {foo}

macro:\inframed {foo}\inframed {foo}
|foo|foo|

|foo|foo|oof|
```

This primitive is similar to:

```
\def\tokenized#1{\scantextokens\expandafter{\normalexpanded{#1}}}
```

and should be more efficient, not that it matters much as we don't use it that much (if at all).

279 \toksapp

One way to append something to a token list is the following:

```
\scratchtoks\expandafter{\the\scratchtoks more stuff}
```

This works all right, but it involves a copy of what is already in `\scratchtoks`. This is seldom a real issue unless we have large token lists and many appends. This is why LuaTEX introduced:

```
\toksapp\scratchtoks{more stuff}
\toksapp\scratchtoksone\scratchtokstwo
```

At some point, when working on LuaMetaTEX, I realized that primitives like this one and the next appenders and prependers to be discussed were always on the radar of Taco and me. Some were even implemented in what we called `eetex`: extended ε-TEX, and we even found back the prototypes, dating from pre-pdfTEX times.

280 \tokspre

Where appending something is easy because of the possible `\expandafter` trickery a prepend would involve more work, either using temporary token registers and/or using a mixture of the (no)expansion added by ε-TEX, but all are kind of inefficient and cumbersome.

```
\tokspre\scratchtoks{less stuff}
\tokspre\scratchtoksone\scratchtokstwo
```

This prepends the token list that is provided.

281 \tolerant

This prefix tags the following macro as being tolerant with respect to the expected arguments. It only makes sense when delimited arguments are used or when braces are mandate.

```
\tolerant\def\foo[#1]#*[#2]{(#1)(#2)}
```

This definition makes `\foo` tolerant for various calls:

```
\foo \foo[1] \foo [1] \foo[1] [2] \foo [1] [2]
```

these give: ()()(1)()(1)()(1)(2) (1)(2). The spaces after the first call disappear because the macro name parser gobbles it, while in the second case the `#*` gobbles them. Here is a variant:

```
\tolerant\def\foo[#1]#,[#2]{!#1!#2!}
```

```
\foo[?] x
\foo[?] [?] x
```

```
\tolerant\def\foo[#1]#*[#2]{!#1!#2!}
```

```
\foo[?] x
\foo[?] [?] x
```

We now get the following:

!?!! x !?!?! x

!?!!x !?!?! x

Here the `#,` remembers that spaces were gobbles and they will be put back when there is no further match. These are just a few examples of this tolerant feature. More details can be found in the lowlevel manuals.

282 \toscaled

The following code gives this: 1234.0 is similar to `\todimension` but omits the `pt` so that we don't need to revert to some nasty stripping code.

```
\scratchdimen = 1234pt \toscaled\scratchdimen
```

283 \tosparsedimension

The following code gives this: 1234pt where ‘sparse’ indicates that redundant trailing zeros are not shown.

```
\scratchdimen = 1234pt \tosparsedimension\scratchdimen
```

284 \tosparsescaled

The following code gives this: 1234 where ‘sparse’ means that redundant trailing zeros are omitted.

```
\scratchdimen = 1234pt \tosparsescaled\scratchdimen
```

285 \unexpanded

This is an ε-TEX enhancement. The content will not be expanded in a context where expansion is happening, like in an `\edef`. In ConTEXt you need to use `\normalunexpanded` because we already had a macro with that name.

```
\def \A{!}                         \meaning\A
\def \B{?}                         \meaning\B
\edef\C{\A\B}                      \meaning\C
\edef\C{\normalunexpanded{\A}\B} \meaning\C
```

```
macro:!
macro:?
macro:!?
macro:\A ?
```

286 \unexpandedendless

This one loops forever so you need to quit explicitly.

287 \unexpandedloop

As follow up on `\expandedloop` we now show its counterpart:

```
\edef\whatever
```

```
  {\unexpandedloop 1 10 1
     {\scratchcounter=\the\currentloopiterator\relax}}
```

```
\meaningasis\whatever
```

```
\def \whatever {\scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter
=0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax
\scratchcounter =0\relax \scratchcounter =0\relax }
```

The difference between the (un)expanded loops and a local controlled one is shown here. Watch the out of order injection of A's.

```
\edef\TestA{\localcontrolledloop 1 5 1 {A}} % out of order
\edef\TestB{\expandedloop        1 5 1 {B}}
\edef\TestC{\unexpandedloop      1 5 1 {C\relax}}
```

AAAAA

We show the effective definition as well as the outcome of using them

```
\meaningasis\TestA
\meaningasis\TestB
\meaningasis\TestC

A: \TestA
B: \TestB
C: \TestC
```

```
\def \TestA {}
\def \TestB {BBBBB}
\def \TestC {C\relax C\relax C\relax C\relax C\relax }

A:
B: BBBBB
C: CCCCC
```

Watch how because it is empty `\TestA` has become a constant macro because that's what deep down empty boils down to.

288 \unexpandedrepeat

This one takes one instead of three arguments which looks better in simple loops.

289 \unless

This ε-TEX prefix will negate the test (when applicable).

```
       \ifx\one\two YES\else NO\fi
\unless\ifx\one\two NO\else YES\fi
```

This primitive is hardly used in ConTEXt and we probably could get rid of these few cases.

290 \unletfrozen

A frozen macro cannot be redefined: you get an error. But as nothing in TEX is set in stone, you can do this:

```
\frozen\def\MyMacro{...}
\unletfrozen\MyMacro
```

and `\MyMacro` is no longer protected from overloading. It is still undecided to what extend ConTEXt will use this feature.

291 \unletprotected

The complementary operation of `\letprotected` can be used to unprotect a macro, so that it gets expandable.

```
               \def  \MyMacroA{alpha}
\protected     \def  \MyMacroB{beta}
               \edef \MyMacroC{\MyMacroA\MyMacroB}
\unletprotected      \MyMacroB
               \edef \MyMacroD{\MyMacroA\MyMacroB}
\meaning             \MyMacroC\crlf
\meaning             \MyMacroD\par
```

Compare this with the example in the previous section:

```
macro:alpha\MyMacroB
macro:alphabeta
```

292 \untraced

Related to the meaning providers is the `\untraced` prefix. It marks a macro as to be reported by name only. It makes the macro look like a primitive.

```
         \def\foo{}
\untraced\def\oof{}

\scratchtoks{\foo\foo\oof\oof}

\tracingall \the\scratchtoks \tracingnone
```

This will show up in the log as follows:

```
1:4: {\the}
1:5: \foo ->
1:5: \foo ->
1:5: \oof
1:5: \oof
```

This is again a trick to avoid too much clutter in a log. Often it doesn't matter to users what the meaning of a macro is (if they trace at all).[5]

[5] An earlier variant could also hide the expansion completely but that was just confusing.

293 \uppercase

See its counterpart `\lowercase` for an explanation.

294 \xdef

This is an alternative for `\global\edef`:

```
\xdef\MyMacro{...}
```

295 \xdefcsname

This is the companion of `\xdef`:

```
\expandafter\xdef\csname MyMacro:1\endcsname{...}
             \xdefcsname MyMacro:1\endcsname{...}
```

296 \xtoks

This is the global variant of `\etoks`.

297 \xtoksapp

This is the global variant of `\etoksapp`.

298 \xtokspre

This is the global variant of `\etokspre`.

Obsolete

The LuaMetaTEX engine has more than its LuaTEX ancestor but it also has less. Because in the end the local control mechanism performed quite okay I decided to drop the `\immediateassignment` and `\immediateassigned` variants. They sort of used the same trick so there isn't much to gain and it was less generic (read: error prone).

Syntax

1 accent

t \accent
[xoffset *dimension*] [yoffset *dimension*] *integer character*

2 aftersomething

l \afterassigned
{ *tokens* }
t \afterassignment
token
t \aftergroup
token
l \aftergrouped
{ *tokens* }
l \atendoffile
token
l \atendoffiled
[reverse] { *tokens* }
l \atendofgroup
token
l \atendofgrouped
{ *tokens* }

3 alignmenttab

l \aligntab

4 arithmic

t \advance
quantity [by] *quantity*
l \advanceby
quantity quantity
t \divide
quantity [by] *quantity*
l \divideby
quantity quantity
l \edivide
quantity quantity
l \edivideby
quantity quantity
t \multiply
quantity [by] *quantity*
l \multiplyby
quantity quantity
l \rdivide
quantity quantity
l \rdivideby
quantity quantity

5 association

l \associateunit
\cs [=] *integer*
> \cs : *integer*

6 auxiliary

l \insertmode
integer
: *integer*
e \interactionmode
integer
: *integer*
t \prevdepth
dimension
: *dimension*
t \prevgraf
integer
: *integer*
t \spacefactor
integer
: *integer*

7 begingroup

t \begingroup
l \beginmathgroup
l \beginsimplegroup

8 beginlocal

l \beginlocalcontrol
l \expandedendless
{ *tokens* }
l \expandedloop
integer integer integer { *tokens* }
l \expandedrepeat
integer { *tokens* }

l \localcontrol
tokens\endlocalcontrol
l \localcontrolled
{ *tokens* }
l \localcontrolledendless
{ *tokens* }
l \localcontrolledloop
see \expandedloop
l \localcontrolledrepeat
integer { *tokens* }
l \unexpandedendless
{ *tokens* }
l \unexpandedloop
see \expandedloop
l \unexpandedrepeat
integer { *tokens* }

9 beginparagraph

t \indent
t \noindent
l \parattribute
integer [=] *integer*
l \quitvmode
l \snapshotpar
cardinal
: *integer*
l \undent
l \wrapuppar
[reverse] { *tokens* }

10 boundary

l \boundary
[=] *integer*
l \mathboundary
[=] *integer*
l \noboundary
l \optionalboundary
[=] *integer*
l \pageboundary
[=] *integer*
l \protrusionboundary
[=] *integer*
l \wordboundary

11 boxproperty

l \boxadapt
(index | *box*) [=] *integer*
> (index | *box*) : *dimension*
l \boxanchor
see \boxadapt
l \boxanchors
(index | *box*) [=] *integer integer*
> (index | *box*) : *integer*
l \boxattribute
(index | *box*) *integer* [=] *integer*
> (index | *box*) *integer* : *integer*
l \boxdirection
see \boxadapt
l \boxfreeze
(index | *box*)
> (index | *box*) : *integer*
l \boxgeometry
see \boxadapt
l \boxlimitate
see \boxfreeze
l \boxorientation
see \boxadapt
l \boxrepack
see \boxfreeze
l \boxshift
(index | *box*) [=] *dimension*
> (index | *box*) : *dimension*
l \boxshrink
see \boxfreeze
l \boxsource
see \boxadapt
l \boxstretch
see \boxfreeze
l \boxtarget
see \boxadapt
l \boxtotal
see \boxfreeze
l \boxvadjust
(index | *box*) { *tokens* }
> (index | *box*) : cardinal
l \boxxmove
see \boxshift
l \boxxoffset
see \boxshift
l \boxymove
see \boxshift

l \boxyoffset
see \boxshift

t \dp
see \boxshift

t \ht
see \boxshift

t \wd
see \boxshift

12 caseshift

t \lowercase
{ *tokens* }

t \uppercase
{ *tokens* }

13 catcodetable

l \initcatcodetable
integer

l \savecatcodetable
integer

14 charnumber

t \char
integer

l \glyph
[xoffset *dimension*] [yoffset *dimension*] [scale *integer*] [xscale *integer*] [yscale *integer*] [left *dimension*] [right *dimension*] [raise *dimension*] [options *integer*] [*font integer*] [id *integer*] *integer*

15 combinetoks

l \etoks
toks { *tokens* }

l \etoksapp
toks { *tokens* }

l \etokspre
toks { *tokens* }

l \gtoksapp
toks { *tokens* }

l \gtokspre
toks { *tokens* }

l \toksapp
toks { *tokens* }

l \tokspre
toks { *tokens* }

l \xtoks
toks { *tokens* }

l \xtoksapp
toks { *tokens* }

l \xtokspre
toks { *tokens* }

16 convert

l \csactive
> *token* : *tokens*

l \csstring
> *token* : *tokens*

l \detokened
> (\cs | { *tokens* } | *toks*) : *tokens*

l \detokenized
> { *tokens* } : *tokens*

l \directlua
> { *tokens* } : *tokens*

l \expanded
> { *tokens* } : *tokens*

t \fontname
> (*font* | *integer*) : *tokens*

l \fontspecifiedname
> (*font* | *integer*) : *tokens*

l \formatname
: *tokens*

t \jobname
: *tokens*

l \luabytecode
> *integer* : *tokens*

l \luaescapestring
> { *tokens* } : *tokens*

l \luafunction
> *integer* : *tokens*

l \luatexbanner
: *tokens*

t \meaning
> *token* : *tokens*

l \meaningasis
> *token* : *tokens*

l \meaningful
> *token* : *tokens*

l \meaningfull
> *token* : *tokens*

l \meaningles
  > *token* : *tokens*
l \meaningless
  > *token* : *tokens*
t \number
  > *integer* : *tokens*
t \romannumeral
  > *integer* : *tokens*
l \semiexpanded
  > { *tokens* } : *tokens*
t \string
  > *token* : *tokens*
l \tocharacter
  > *integer* : *tokens*
l \todimension
  > *dimension* : *tokens*
l \tohexadecimal
  > *integer* : *tokens*
l \tointeger
  > *integer* : *tokens*
l \tomathstyle
  > *mathstyle* : *tokens*
l \toscaled
  > *dimension* : *tokens*
l \tosparsedimension
  > *dimension* : *tokens*
l \tosparsescaled
  > *dimension* : *tokens*

17 csname

l \begincsname
  tokens\endcsname
t \csname
  tokens\endcsname
l \futurecsname
  tokens\endcsname
l \lastnamedcs

18 def

l \cdef
  \cs [*preamble*] { *tokens* }
l \cdefcsname
  tokens\endcsname [*preamble*] { *tokens* }
t \def
  \cs [*preamble*] { *tokens* }
l \defcsname
  tokens\endcsname [*preamble*] { *tokens* }
t \edef
  \cs [*preamble*] { *tokens* }
l \edefcsname
  tokens\endcsname [*preamble*] { *tokens* }
t \gdef
  \cs [*preamble*] { *tokens* }
l \gdefcsname
  tokens\endcsname [*preamble*] { *tokens* }
t \xdef
  \cs [*preamble*] { *tokens* }
l \xdefcsname
  tokens\endcsname [*preamble*] { *tokens* }

19 definecharcode

l \Udelcode
  integer [=] *integer*
  > *integer* : *integer*
l \Umathcode
  integer [=] *integer*
  > *integer* : *integer*
l \amcode
  integer [=] *integer*
  > *integer* : *integer*
t \catcode
  integer [=] *integer*
  > *integer* : *integer*
t \delcode
  integer [=] *integer*
  > *integer* : *integer*
l \hccode
  integer [=] *integer*
  > *integer* : *integer*
l \hmcode
  integer [=] *integer*
  > *integer* : *integer*
t \lccode
  integer [=] *integer*
  > *integer* : *integer*
t \mathcode
  integer [=] *integer*
  > *integer* : *integer*
t \sfcode
  integer [=] *integer*
  > *integer* : *integer*
t \uccode
  integer [=] *integer*
  > *integer* : *integer*

20 definefamily

t \scriptfont
family (*font* | *integer*)
> family : *integer*

t \scriptscriptfont
see \scriptfont

t \textfont
see \scriptfont

21 definefont

t \font
\cs ({ filename } | filename) [(at *dimension* | scaled *integer*)]
: *tokens*

22 delimiternumber

l \Udelimiter
integer integer integer

t \delimiter
integer

23 discretionary

t \-

l \automaticdiscretionary

t \discretionary
[penalty] [postword] [preword] [break] [nobreak] [options] [class] { *tokens* } { *tokens* } { *tokens* }

l \explicitdiscretionary

24 endcsname

t \endcsname

25 endgroup

t \endgroup

l \endmathgroup

l \endsimplegroup

26 endjob

t \dump

t \end

27 endlocal

l \endlocalcontrol

28 endparagraph

t \par

29 endtemplate

l \aligncontent

t \cr

t \crcr

t \noalign
{ *tokens* }

t \omit

t \span

30 equationnumber

t \eqno
{ *tokens* }

t \leqno
{ *tokens* }

31 expandafter

l \expand
token

l \expandactive
token

t \expandafter
token token

l \expandafterpars
token

l \expandafterspaces
token

l \expandcstoken
token

l \expandedafter
token { *tokens* }

l \expandparameter
integer

l \expandtoken
token

l \expandtoks
 { *tokens* }

l \futureexpand
 token token token

l \futureexpandis
 TODO

l \futureexpandisap
 TODO

l \semiexpand
 token

e \unless

32 explicitspace

**t **

l \explicitspace
 TODO

33 fontproperty

l \cfcode
 (*font* | *integer*) *integer* [=] *integer*
 > (*font* | *integer*) *integer* : *integer*

l \efcode
 see \cfcode

t \fontdimen
 (*font* | *integer*) *integer* [=] *dimension*
 > (*font* | *integer*) *integer* : *dimension*

t \hyphenchar
 (*font* | *integer*) [=] *integer*
 > (*font* | *integer*) : *integer*

l \lpcode
 see \fontdimen

l \rpcode
 see \fontdimen

l \scaledfontdimen
 see \hyphenchar

t \skewchar
 see \hyphenchar

34 getmark

t \botmark

e \botmarks
 integer

l \currentmarks
 integer

t \firstmark

e \firstmarks
 integer

t \splitbotmark

e \splitbotmarks
 integer

t \splitfirstmark

e \splitfirstmarks
 integer

t \topmark

e \topmarks
 integer

35 halign

t \halign
 [attr *integer integer*] [callback *integer*] [discard] [noskips] [reverse] [to *dimension*] [spread *dimension*] { *tokens* }

36 hmove

t \moveleft
 dimension box

t \moveright
 dimension box

37 hrule

t \hrule
 [attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [left *dimension*] [right *dimension*] [top *dimension*] [bottom *dimension*] [xoffset *dimension*] [yoffset *dimension*] [*font integer*] [fam *integer*] [char *integer*]

l \nohrule
 see \hrule

l \virtualhrule
 [attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [left *dimension*] [right *dimension*] [top *dimension*] [bottom *dimension*] [xoffset *dimension*] [yoffset *dimension*]

38 hskip

t \hfil

t \hfill

t \hfilneg

t \hskip

dimension [plus (*dimension* | fi [n*l])] [minus (*dimension* | fi [n*l])]

t \hss

39 hyphenation

l \hjcode

integer [=] *integer*

t \hyphenation

{ *tokens* }

l \hyphenationmin

[=] *integer*

t \patterns

{ *tokens* }

l \postexhyphenchar

[=] *integer*

l \posthyphenchar

[=] *integer*

l \preexhyphenchar

[=] *integer*

l \prehyphenchar

[=] *integer*

40 iftest

t \else

t \fi

t \if

l \ifabsdim

dimension (! | < | = | > | ∈ | ∉ | ≠ | ≤ | ≥ | ≰ | ≱) *dimension*

l \ifabsfloat

float (! | < | = | > | ∈ | ∉ | ≠ | ≤ | ≥ | ≰ | ≱) *float*

l \ifabsnum

integer (! | < | = | > | ∈ | ∉ | ≠ | ≤ | ≥ | ≰ | ≱) *integer*

l \ifarguments

l \ifboolean

integer

t \ifcase

integer

t \ifcat

token

l \ifchkdim

tokens\or

l \ifchkdimension

tokens\or

l \ifchknum

tokens\or

l \ifchknumber

tokens\or

l \ifcmpdim

dimension dimension

l \ifcmpnum

integer integer

l \ifcondition

\if...

e \ifcsname

tokens\endcsname

l \ifcstok

tokens\relax

e \ifdefined

token

t \ifdim

see \ifabsdim

l \ifdimexpression

tokens\relax

l \ifdimval

tokens\or

l \ifempty

(*token* | { *tokens* })

t \iffalse

l \ifflags

\cs

l \iffloat

see \ifabsfloat

e \iffontchar

integer integer

l \ifhaschar

token { *tokens* }

l \ifhastok

token { *tokens* }

l \ifhastoks

tokens\relax

l \ifhasxtoks

tokens\relax

t \ifhbox

(index | *box*)

t \ifhmode

l \ifinalignment

l \ifincsname
    tokens\endcsname

t \ifinner

l \ifinsert
    integer

l \ifintervaldim
    dimension dimension dimension

l \ifintervalfloat
    integer integer integer

l \ifintervalnum
    float float float

l \iflastnamedcs

l \ifmathparameter
    integer

l \ifmathstyle
    mathstyle

t \ifmmode

t \ifnum
    see \ifabsnum

l \ifnumexpression
    tokens\relax

l \ifnumval
    tokens\or

t \ifodd
    integer

l \ifparameter
    parameter\or

l \ifparameters

l \ifrelax
    token

l \iftok
    tokens\relax

t \iftrue

t \ifvbox
    see \ifhbox

t \ifvmode

t \ifvoid
    see \ifhbox

t \ifx
    token

l \ifzerodim
    dimension

l \ifzerofloat
    float

l \ifzeronum
    integer

t \or

l \orelse

l \orunless

41 ignoresomething

l \ignorearguments

l \ignorenestedupto
    token

l \ignorepars

l \ignorerest

t \ignorespaces

l \ignoreupto
    token

42 input

t \endinput

t \eofinput
    { *tokens* } ({ filename } | filename)

t \input
    ({ filename } | filename)

l \quitloop

l \quitloopnow

l \retokenized
    [catcodetable] { *tokens* }

l \scantextokens
    { *tokens* }

e \scantokens
    { *tokens* }

l \tokenized
    { *tokens* }

43 insert

t \insert
    integer

44 interaction

t \batchmode

t \errorstopmode

t \nonstopmode

t \scrollmode

45 internaldimension

t \boxmaxdepth
    [=] *dimension*

: *dimension*

t \delimitershortfall
[=] *dimension*
: *dimension*

t \displayindent
[=] *dimension*
: *dimension*

t \displaywidth
[=] *dimension*
: *dimension*

t \emergencyextrastretch
[=] *dimension*
: *dimension*

t \emergencystretch
[=] *dimension*
: *dimension*

l \glyphxoffset
[=] *dimension*
: *dimension*

l \glyphyoffset
[=] *dimension*
: *dimension*

t \hangindent
[=] *dimension*
: *dimension*

t \hfuzz
[=] *dimension*
: *dimension*

t \hsize
[=] *dimension*
: *dimension*

l \ignoredepthcriterion
[=] *dimension*
: *dimension*

t \lineskiplimit
[=] *dimension*
: *dimension*

t \mathsurround
[=] *dimension*
: *dimension*

t \maxdepth
[=] *dimension*
: *dimension*

t \nulldelimiterspace
[=] *dimension*
: *dimension*

t \overfullrule
[=] *dimension*
: *dimension*

l \pageextragoal
[=] *dimension*
: *dimension*

t \parindent
[=] *dimension*
: *dimension*

t \predisplaysize
[=] *dimension*
: *dimension*

l \pxdimen
[=] *dimension*
: *dimension*

t \scriptspace
[=] *dimension*
: *dimension*

l \shortinlinemaththreshold
[=] *dimension*
: *dimension*

t \splitmaxdepth
[=] *dimension*
: *dimension*

l \tabsize
[=] *dimension*
: *dimension*

t \vfuzz
[=] *dimension*
: *dimension*

t \vsize
[=] *dimension*
: *dimension*

46 internalglue

t \abovedisplayshortskip
[=] *glue*
: *glue*

t \abovedisplayskip
[=] *glue*
: *glue*

l \additionalpageskip
[=] *glue*
: *glue*

t \baselineskip
[=] *glue*
: *glue*

t \belowdisplayshortskip
[=] *glue*
: *glue*

t \belowdisplayskip
  [=] *glue*
  : *glue*

l \emergencyleftskip
  [=] *glue*
  : *glue*

l \emergencyrightskip
  [=] *glue*
  : *glue*

l \initialpageskip
  [=] *glue*
  : *glue*

l \initialtopskip
  [=] *glue*
  : *glue*

t \leftskip
  [=] *glue*
  : *glue*

t \lineskip
  [=] *glue*
  : *glue*

l \mathsurroundskip
  [=] *glue*
  : *glue*

l \maththreshold
  [=] *glue*
  : *glue*

l \parfillleftskip
  [=] *glue*
  : *glue*

l \parfillrightskip
  [=] *glue*
  : *glue*

t \parfillskip
  [=] *glue*
  : *glue*

l \parinitleftskip
  [=] *glue*
  : *glue*

l \parinitrightskip
  [=] *glue*
  : *glue*

t \parskip
  [=] *glue*
  : *glue*

t \rightskip
  [=] *glue*
  : *glue*

t \spaceskip
  [=] *glue*
  : *glue*

t \splittopskip
  [=] *glue*
  : *glue*

t \tabskip
  [=] *glue*
  : *glue*

t \topskip
  [=] *glue*
  : *glue*

t \xspaceskip
  [=] *glue*
  : *glue*

47 internalinteger

t \adjdemerits
  [=] *integer*
  : *integer*

l \adjustspacing
  [=] *integer*
  : *integer*

l \adjustspacingshrink
  [=] *integer*
  : *integer*

l \adjustspacingstep
  [=] *integer*
  : *integer*

l \adjustspacingstretch
  [=] *integer*
  : *integer*

l \alignmentcellsource
  [=] *integer*
  : *integer*

l \alignmentwrapsource
  [=] *integer*
  : *integer*

l \automatichyphenpenalty
  [=] *integer*
  : *integer*

l \automigrationmode
  [=] *integer*
  : *integer*

l \autoparagraphmode
  [=] *integer*
  : *integer*

t \binoppenalty
[=] *integer*
: *integer*

t \brokenpenalty
[=] *integer*
: *integer*

l \catcodetable
[=] *integer*
: *integer*

t \clubpenalty
[=] *integer*
: *integer*

t \day
[=] *integer*
: *integer*

t \defaulthyphenchar
[=] *integer*
: *integer*

t \defaultskewchar
[=] *integer*
: *integer*

t \delimiterfactor
[=] *integer*
: *integer*

l \discretionaryoptions
[=] *integer*
: *integer*

t \displaywidowpenalty
[=] *integer*
: *integer*

t \doubleadjdemerits
[=] *integer*
: *integer*

t \doublehyphendemerits
[=] *integer*
: *integer*

t \endlinechar
[=] *integer*
: *integer*

t \errorcontextlines
[=] *integer*
: *integer*

t \escapechar
[=] *integer*
: *integer*

l \eufactor
[=] *integer*
: *integer*

l \exceptionpenalty
[=] *integer*
: *integer*

t \exhyphenchar
[=] *integer*
: *integer*

t \exhyphenpenalty
[=] *integer*
: *integer*

l \explicithyphenpenalty
[=] *integer*
: *integer*

t \fam
[=] *integer*
: *integer*

t \finalhyphendemerits
[=] *integer*
: *integer*

l \firstvalidlanguage
[=] *integer*
: *integer*

t \floatingpenalty
[=] *integer*
: *integer*

t \globaldefs
[=] *integer*
: *integer*

l \glyphdatafield
[=] *integer*
: *integer*

l \glyphoptions
[=] *integer*
: *integer*

l \glyphscale
[=] *integer*
: *integer*

l \glyphscriptfield
[=] *integer*
: *integer*

l \glyphscriptscale
[=] *integer*
: *integer*

l \glyphscriptscriptscale
[=] *integer*
: *integer*

l \glyphslant
[=] *integer*
: *integer*

```
l \glyphstatefield
      [=] integer
    : integer
l \glyphtextscale
      [=] integer
    : integer
l \glyphweight
      [=] integer
    : integer
l \glyphxscale
      [=] integer
    : integer
l \glyphyscale
      [=] integer
    : integer
t \hangafter
      [=] integer
    : integer
t \hbadness
      [=] integer
    : integer
t \holdinginserts
      [=] integer
    : integer
l \holdingmigrations
      [=] integer
    : integer
l \hyphenationmode
      [=] integer
    : integer
t \hyphenpenalty
      [=] integer
    : integer
t \interlinepenalty
      [=] integer
    : integer
t \language
      [=] integer
    : integer
e \lastlinefit
      [=] integer
    : integer
t \lefthyphenmin
      [=] integer
    : integer
l \linebreakcriterion
      [=] integer
    : integer
l \linebreakoptional
      [=] integer
    : integer
l \linebreakpasses
      [=] integer
    : integer
l \linedirection
      [=] integer
    : integer
t \linepenalty
      [=] integer
    : integer
l \localbrokenpenalty
      [=] integer
    : integer
l \localinterlinepenalty
      [=] integer
    : integer
l \localpretolerance
      [=] integer
    : integer
l \localtolerance
      [=] integer
    : integer
t \looseness
      [=] integer
    : integer
l \luacopyinputnodes
      [=] integer
    : integer
l \mathbeginclass
      [=] integer
    : integer
l \mathcheckfencesmode
      [=] integer
    : integer
l \mathdictgroup
      [=] integer
    : integer
l \mathdictproperties
      [=] integer
    : integer
l \mathdirection
      [=] integer
    : integer
l \mathdisplaymode
      [=] integer
    : integer
```

```
l \mathdisplaypenaltyfactor
      [=] integer
    : integer
l \mathdisplayskipmode
      [=] integer
    : integer
l \mathdoublescriptmode
      [=] integer
    : integer
l \mathendclass
      [=] integer
    : integer
l \matheqnogapstep
      [=] integer
    : integer
l \mathfontcontrol
      [=] integer
    : integer
l \mathgluemode
      [=] integer
    : integer
l \mathgroupingmode
      [=] integer
    : integer
l \mathinlinepenaltyfactor
      [=] integer
    : integer
l \mathleftclass
      [=] integer
    : integer
l \mathlimitsmode
      [=] integer
    : integer
l \mathnolimitsmode
      [=] integer
    : integer
l \mathpenaltiesmode
      [=] integer
    : integer
l \mathpretolerance
      [=] integer
    : integer
l \mathrightclass
      [=] integer
    : integer
l \mathrulesfam
      [=] integer
    : integer
l \mathrulesmode
      [=] integer
    : integer
l \mathscriptsmode
      [=] integer
    : integer
l \mathslackmode
      [=] integer
    : integer
l \mathspacingmode
      [=] integer
    : integer
l \mathsurroundmode
      [=] integer
    : integer
l \mathtolerance
      [=] integer
    : integer
t \maxdeadcycles
      [=] integer
    : integer
t \month
      [=] integer
    : integer
t \newlinechar
      [=] integer
    : integer
l \normalizelinemode
      [=] integer
    : integer
l \normalizeparmode
      [=] integer
    : integer
l \nospaces
      [=] integer
    : integer
l \orphanpenalty
      [=] integer
    : integer
l \outputbox
      [=] integer
    : integer
t \outputpenalty
      [=] integer
    : integer
l \overloadmode
      [=] integer
    : integer
```

```
l \parametermode
      [=] integer
    : integer
l \pardirection
      [=] integer
    : integer
t \pausing
      [=] integer
    : integer
t \postdisplaypenalty
      [=] integer
    : integer
l \postinlinepenalty
      [=] integer
    : integer
l \postshortinlinepenalty
      [=] integer
    : integer
l \prebinoppenalty
      [=] integer
    : integer
e \predisplaydirection
      [=] integer
    : integer
l \predisplaygapfactor
      [=] integer
    : integer
t \predisplaypenalty
      [=] integer
    : integer
l \preinlinepenalty
      [=] integer
    : integer
l \prerelpenalty
      [=] integer
    : integer
l \preshortinlinepenalty
      [=] integer
    : integer
t \pretolerance
      [=] integer
    : integer
l \protrudechars
      [=] integer
    : integer
t \relpenalty
      [=] integer
    : integer
t \righthyphenmin
      [=] integer
    : integer
e \savinghyphcodes
      [=] integer
    : integer
e \savingvdiscards
      [=] integer
    : integer
l \setfontid
      [=] integer
    : integer
t \setlanguage
      [=] integer
    : integer
l \shapingpenaltiesmode
      [=] integer
    : integer
l \shapingpenalty
      [=] integer
    : integer
l \shortinlineorphanpenalty
      [=] integer
    : integer
t \showboxbreadth
      [=] integer
    : integer
t \showboxdepth
      [=] integer
    : integer
t \shownodedetails
      [=] integer
    : integer
l \singlelinepenalty
      [=] integer
    : integer
l \spacefactormode
      [=] integer
    : integer
l \spacefactorshrinklimit
      [=] integer
    : integer
l \spacefactorstretchlimit
      [=] integer
    : integer
l \supmarkmode
      [=] integer
    : integer
```

l \textdirection
[=] *integer*
: *integer*

t \time
[=] *integer*
: *integer*

t \tolerance
[=] *integer*
: *integer*

l \tracingadjusts
[=] *integer*
: *integer*

l \tracingalignments
[=] *integer*
: *integer*

e \tracingassigns
[=] *integer*
: *integer*

t \tracingcommands
[=] *integer*
: *integer*

l \tracingexpressions
[=] *integer*
: *integer*

l \tracingfonts
[=] *integer*
: *integer*

l \tracingfullboxes
[=] *integer*
: *integer*

e \tracinggroups
[=] *integer*
: *integer*

l \tracinghyphenation
[=] *integer*
: *integer*

e \tracingifs
[=] *integer*
: *integer*

l \tracinginserts
[=] *integer*
: *integer*

l \tracinglevels
[=] *integer*
: *integer*

l \tracinglists
[=] *integer*
: *integer*

t \tracinglostchars
[=] *integer*
: *integer*

t \tracingmacros
[=] *integer*
: *integer*

l \tracingmarks
[=] *integer*
: *integer*

l \tracingmath
[=] *integer*
: *integer*

e \tracingnesting
[=] *integer*
: *integer*

l \tracingnodes
[=] *integer*
: *integer*

t \tracingonline
[=] *integer*
: *integer*

t \tracingoutput
[=] *integer*
: *integer*

t \tracingpages
[=] *integer*
: *integer*

t \tracingparagraphs
[=] *integer*
: *integer*

l \tracingpasses
[=] *integer*
: *integer*

l \tracingpenalties
[=] *integer*
: *integer*

t \tracingrestores
[=] *integer*
: *integer*

t \tracingstats
[=] *integer*
: *integer*

t \uchyph
[=] *integer*
: *integer*

l \variablefam
[=] *integer*
: *integer*

t \vbadness
[=] *integer*
: *integer*

t \widowpenalty
[=] *integer*
: *integer*

t \year
[=] *integer*
: *integer*

48 internalmuglue

t \medmuskip
[=] *muglue*
: *muglue*

l \pettymuskip
[=] *muglue*
: *muglue*

t \thickmuskip
[=] *muglue*
: *muglue*

t \thinmuskip
[=] *muglue*
: *muglue*

l \tinymuskip
[=] *muglue*
: *muglue*

49 internaltoks

t \errhelp
[=] *toks*
: *toks*

l \everybeforepar
[=] *toks*
: *toks*

t \everycr
[=] *toks*
: *toks*

t \everydisplay
[=] *toks*
: *toks*

e \everyeof
[=] *toks*
: *toks*

t \everyhbox
[=] *toks*
: *toks*

t \everyjob
[=] *toks*
: *toks*

t \everymath
[=] *toks*
: *toks*

l \everymathatom
[=] *toks*
: *toks*

t \everypar
[=] *toks*
: *toks*

l \everytab
[=] *toks*
: *toks*

t \everyvbox
[=] *toks*
: *toks*

t \output
[=] *toks*
: *toks*

50 italiccorrection

t \/

l \explicititaliccorrection
TODO

51 kern

t \hkern
dimension

t \kern
dimension

t \vkern
dimension

52 leader

t \cleaders
(*box* | *rule* | glyph) *glue*

l \gleaders
see \cleaders

t \leaders
see \cleaders

l \uleaders
[callback *integer*] (*box* | *rule* | glyph) *glue*

t \xleaders
see \cleaders

53 legacy

t \shipout
{ *tokens* }

54 let

l \futuredef
\cs \cs

t \futurelet
\cs [=] \cs

l \glet
\cs

l \gletcsname
tokens\endcsname

l \glettonothing
\cs

t \let
\cs

l \letcharcode
\cs

l \letcsname
tokens\endcsname

l \letfrozen
\cs

l \letprotected
\cs

l \lettolastnamedcs
\cs

l \lettonothing
\cs

l \swapcsvalues
\cs \cs

l \unletfrozen
\cs

l \unletprotected
\cs

55 localbox

l \localleftbox
box

l \localmiddlebox
box

l \localrightbox
box

56 luafunctioncall

l \luabytecodecall
integer

l \luafunctioncall
integer

57 makebox

t \box
(index | *box*)

t \copy
see \box

l \dbox
[target *integer*] [to *dimension*] [adapt] [attr *integer integer*] [anchor *integer*] [axis *integer*] [shift *dimension*] [spread *dimension*] [source *integer*] [direction *integer*] [delay] [orientation *integer*] [xoffset *dimension*] [xmove *dimension*] [yoffset *dimension*] [ymove *dimension*] [reverse] [retain] [container] [class *integer*] { *tokens* }

l \dpack
see \dbox

l \dsplit
[attr] [to] [upto] { *tokens* }

t \hbox
see \dbox

l \hpack
see \dbox

l \insertbox
integer

l \insertcopy
integer

t \lastbox

l \localleftboxbox

l \localmiddleboxbox

l \localrightboxbox

l \tpack
see \dbox

l \tsplit
see \dsplit

t \vbox
see \dbox

l \vpack
see \dbox

t \vsplit
see \dsplit

t \vtop
see \dbox

58 mark

l \clearmarks
integer

l \flushmarks

t \mark
{ *tokens* }

e \marks
integer { *tokens* }

59 mathaccent

l \Umathaccent
[attr *integer integer*] [center]
[class *integer*] [exact] [source
integer] [stretch] [shrink]
[fraction *integer*] [fixed]
[keepbase] [nooverflow] [base]
(both [fixed] *character* [fixed]
character | bottom [fixed]
character | top [fixed]
character | overlay
character | *character*)

t \mathaccent
{ *tokens* }

60 mathcharnumber

l \Umathchar
integer

t \mathchar
integer

l \mathclass
integer

l \mathdictionary
integer mathchar

61 mathchoice

t \mathchoice
{ *tokens* } { *tokens* } { *tokens* } { *tokens* }

l \mathdiscretionary
[class *integer*] { *tokens* } { *tokens* }
{ *tokens* }

l \mathstack
{ *tokens* }

62 mathcomponent

l \mathatom
[attr *integer integer*] [all *integer*]
[leftclass *integer*] [limits]
[rightclass *integer*] [class *integer*]
[unpack] [unroll] [single] [source
integer] [text*font*] [math*font*]
[options *integer*] [nolimits]
[nooverflow] [void] [phantom]
[*integer*]

t \mathbin
{ *tokens* }

t \mathclose
{ *tokens* }

t \mathinner
{ *tokens* }

t \mathop
{ *tokens* }

t \mathopen
{ *tokens* }

t \mathord
{ *tokens* }

t \mathpunct
{ *tokens* }

t \mathrel
{ *tokens* }

t \overline
{ *tokens* }

t \underline
{ *tokens* }

63 mathfence

l \Uleft
[auto] [attr *integer integer*] [axis]
[bottom *dimension*] [depth *dimension*]
[factor *integer*] [height *dimension*]
[noaxis] [nocheck] [nolimits]
[nooverflow] [leftclass *integer*]
[limits] [exact] [void] [phantom]
[class *integer*] [rightclass *integer*]
[scale] [source *integer*] [top]
delimiter

l \Umiddle
see \Uleft

l \Uoperator
see \Uleft

l \Uright
see \Uleft

l \Uvextensible
see \Uleft

t \left
see \Uleft

t \middle
see \Uleft

t \right
see \Uleft

64 mathfraction

l \Uabove
dimension [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*]

l \Uabovewithdelims
delimiter delimiter *dimension* [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*]

l \Uatop
see \Uabove

l \Uatopwithdelims
see \Uabovewithdelims

l \Uover
[attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*]

l \Uoverwithdelims
delimiter delimiter [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*]

l \Uskewed
delimiter [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*]

l \Uskewedwithdelims
delimiter delimiter delimiter [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*]

l \Ustretched
see \Uskewed

l \Ustretchedwithdelims
see \Uskewedwithdelims

t \above
dimension

t \abovewithdelims
delimiter delimiter *dimension*

t \atop
dimension

t \atopwithdelims
delimiter delimiter *dimension*

t \over

t \overwithdelims
delimiter delimiter

65 mathmodifier

l \Umathadapttoleft

l \Umathadapttoright

l \Umathlimits

l \Umathnoaxis

l \Umathnolimits

l \Umathopenupdepth
dimension

l \Umathopenupheight
dimension

l \Umathphantom

l \Umathsource
  [nucleus] *integer*
l \Umathuseaxis
l \Umathvoid
t \displaylimits
t \limits
t \nolimits

66 mathparameter

l \Umathaccentbasedepth
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentbaseheight
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentbottomovershoot
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentbottomshiftdown
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentextendmargin
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentsuperscriptdrop
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentsuperscriptpercent
  mathstyle [=] *integer*
  > *mathstyle* : *integer*
l \Umathaccenttopovershoot
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccenttopshiftup
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathaccentvariant
  [=] *mathstyle*
  : *mathstyle*
l \Umathaxis
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathbottomaccentvariant
  [=] *mathstyle*
  : *mathstyle*
l \Umathconnectoroverlapmin
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathdegreevariant
  [=] *mathstyle*
  : *mathstyle*
l \Umathdelimiterextendmargin
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathdelimiterovervariant
  [=] *mathstyle*
  : *mathstyle*
l \Umathdelimiterpercent
  mathstyle [=] *integer*
  > *mathstyle* : *integer*
l \Umathdelimitershortfall
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathdelimiterundervariant
  [=] *mathstyle*
  : *mathstyle*
l \Umathdenominatorvariant
  [=] *mathstyle*
  : *mathstyle*
l \Umathexheight
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasubpreshift
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasubprespace
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasubshift
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasubspace
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasuppreshift
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasupprespace
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasupshift
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*
l \Umathextrasupspace
  mathstyle [=] *dimension*
  > *mathstyle* : *dimension*

l \Umathflattenedaccentbasedepth
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccentbaseheight
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccentbottomshiftdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccenttopshiftup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractiondelsize
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractiondenomdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractiondenomvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractionnumup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractionnumvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractionrule
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractionvariant
[=] *mathstyle*
: *mathstyle*

l \Umathhextensiblevariant
[=] *mathstyle*
: *mathstyle*

l \Umathlimitabovebgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathlimitabovekern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathlimitabovevgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathlimitbelowbgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathlimitbelowkern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathlimitbelowvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathnolimitsubfactor
mathstyle [=] *integer*
> *mathstyle* : *integer*

l \Umathnolimitsupfactor
mathstyle [=] *integer*
> *mathstyle* : *integer*

l \Umathnumeratorvariant
[=] *mathstyle*
: *mathstyle*

l \Umathoperatorsize
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathoverbarkern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathoverbarrule
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathoverbarvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathoverdelimiterbgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathoverdelimitervariant
[=] *mathstyle*
: *mathstyle*

l \Umathoverdelimitervgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathoverlayaccentvariant
[=] *mathstyle*
: *mathstyle*

l \Umathoverlinevariant
[=] *mathstyle*
: *mathstyle*

l \Umathpresubshiftdistance
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathpresupshiftdistance
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathprimeraise
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathprimeraisecomposed
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathprimeshiftdrop
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathprimeshiftup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathprimespaceafter
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathprimevariant
[=] *mathstyle*
: *mathstyle*

l \Umathprimewidth
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathquad
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicaldegreeafter
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicaldegreebefore
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicaldegreeraise
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalextensibleafter
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalextensiblebefore
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalkern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalrule
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalvariant
[=] *mathstyle*
: *mathstyle*

l \Umathradicalvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathruledepth
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathruleheight
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathskeweddelimitertolerance
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathskewedfractionhgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathskewedfractionvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathspaceafterscript
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathspacebeforescript
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathstackdenomdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathstacknumup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathstackvariant
[=] *mathstyle*
: *mathstyle*

l \Umathstackvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubscriptvariant
[=] *mathstyle*
: *mathstyle*

l \Umathsubshiftdistance
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubshiftdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubshiftdrop
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubsupshiftdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubsupvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubtopmax
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupbottommin
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsuperscriptvariant
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupshiftdistance
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupshiftdrop
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupshiftup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupsubbottommax
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathtopaccentvariant
[=] *mathstyle*
: *mathstyle*

l \Umathunderbarkern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathunderbarrule
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathunderbarvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathunderdelimiterbgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathunderdelimitervariant
[=] *mathstyle*
: *mathstyle*

l \Umathunderdelimitervgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathunderlinevariant
[=] *mathstyle*
: *mathstyle*

l \Umathvextensiblevariant
[=] *mathstyle*
: *mathstyle*

l \Umathxscale
mathstyle [=] *integer*
> *mathstyle* : *integer*

l \Umathyscale
mathstyle [=] *integer*
> *mathstyle* : *integer*

l \copymathatomrule
integer integer

l \copymathparent
integer integer

l \copymathspacing
integer integer

l \letmathatomrule
integer integer integer integer integer

l \letmathparent
integer integer

l \letmathspacing
see \letmathatomrule

l \resetmathspacing

l \setdefaultmathcodes

l \setmathatomrule
integer integer mathstyle integer integer

l \setmathdisplaypostpenalty
integer [=] *integer*

l \setmathdisplayprepenalty
integer [=] *integer*

l \setmathignore
mathparameter *integer*

l \setmathoptions
integer [=] *integer*

l \setmathpostpenalty
integer [=] *integer*

l \setmathprepenalty
integer [=] *integer*

l \setmathspacing
integer integer mathstyle glue

67 mathradical

l \Udelimited
[attr *integer integer*] [bottom]

[exact] [top] [style *mathstyle*]
[source *integer*] [stretch] [shrink]
[width *dimension*] [height *dimension*]
[depth *dimension*] [left] [middle]
[right] [nooverflow] delimiter
delimiter [delimiter] [delimiter]
(mathatom | { *tokens* })

l \Udelimiterover
[attr *integer integer*] [bottom]
[exact] [top] [style *mathstyle*]
[source *integer*] [stretch] [shrink]
[width *dimension*] [height *dimension*]
[depth *dimension*] [left] [middle]
[right] [nooverflow] delimiter
[delimiter] [delimiter]
(mathatom | { *tokens* })

l \Udelimiterunder
see \Udelimiterover

l \Uhextensible
see \Udelimiterover

l \Uoverdelimiter
see \Udelimiterover

l \Uradical
see \Udelimiterover

l \Uroot
[attr *integer integer*] [bottom]
[exact] [top] [style *mathstyle*]
[source *integer*] [stretch] [shrink]
[width *dimension*] [height *dimension*]
[depth *dimension*] [left] [middle]
[right] [nooverflow] delimiter
[delimiter] [delimiter]
(mathatom | { *tokens* })
(mathatom | { *tokens* })

l \Urooted
[attr *integer integer*] [bottom]
[exact] [top] [style *mathstyle*]
[source *integer*] [stretch] [shrink]
[width *dimension*] [height *dimension*]
[depth *dimension*] [left] [middle]
[right] [nooverflow] delimiter
delimiter [delimiter] [delimiter]
(mathatom | { *tokens* })
(mathatom | { *tokens* })

l \Uunderdelimiter
see \Udelimiterover

t \radical
see \Uroot

68 mathscript

l \noatomruling
t \nonscript
l \nosubprescript
l \nosubscript
l \nosuperprescript
l \nosuperscript
l \primescript
(mathatom | { *tokens* })

l \shiftedsubprescript
see \primescript

l \shiftedsubscript
see \primescript

l \shiftedsuperprescript
see \primescript

l \shiftedsuperscript
see \primescript

l \subprescript
see \primescript

l \subscript
see \primescript

l \superprescript
see \primescript

l \superscript
see \primescript

69 mathshiftcs

l \Ustartdisplaymath
l \Ustartmath
l \Ustartmathmode
l \Ustopdisplaymath
l \Ustopmath
l \Ustopmathmode

70 mathstyle

l \allcrampedstyles
l \alldisplaystyles
l \allmainstyles
l \allmathstyles
l \allscriptscriptstyles
l \allscriptstyles
l \allsplitstyles
l \alltextstyles
l \alluncrampedstyles
l \allunsplitstyles

l \crampeddisplaystyle
l \crampedscriptscriptstyle
l \crampedscriptstyle
l \crampedtextstyle
t \displaystyle
l \givenmathstyle
mathstyle
l \scaledmathstyle
integer
> *mathstyle* : *integer*
t \scriptscriptstyle
t \scriptstyle
t \textstyle

71 message

t \errmessage
{ *tokens* }
t \message
{ *tokens* }

72 mkern

t \mkern
dimension

73 mskip

l \mathatomskip
muglue
t \mskip
muglue

74 noexpand

t \noexpand
token

75 pageproperty

t \deadcycles
[=] *integer*
: *integer*
l \insertdepth
integer [=] *dimension*
> *integer* : *dimension*
l \insertdistance
integer [=] *dimension*
> *integer* : *dimension*
l \insertheight
integer [=] *dimension*
> *integer* : *dimension*
l \insertheights
[=] *dimension*
: *dimension*
l \insertlimit
integer [=] *dimension*
> *integer* : *dimension*
l \insertmaxdepth
integer [=] *dimension*
> *integer* : *dimension*
l \insertmultiplier
integer [=] *integer*
> *integer* : *integer*
t \insertpenalties
[=] *integer*
: *integer*
l \insertpenalty
integer [=] *integer*
> *integer* : *integer*
l \insertstorage
integer [=] *integer*
> *integer* : *integer*
l \insertstoring
[=] *integer*
: *integer*
l \insertwidth
integer [=] *dimension*
> *integer* : *dimension*
l \pagedepth
[=] *dimension*
: *dimension*
l \pageexcess
[=] *dimension*
: *dimension*
t \pagefilllstretch
[=] *dimension*
: *dimension*
t \pagefillstretch
[=] *dimension*
: *dimension*
t \pagefilstretch
[=] *dimension*
: *dimension*
l \pagefistretch
[=] *dimension*
: *dimension*

t \pagegoal
 [=] *dimension*
 : *dimension*
l \pagelastdepth
 [=] *dimension*
 : *dimension*
l \pagelastfilllstretch
 [=] *dimension*
 : *dimension*
l \pagelastfillstretch
 [=] *dimension*
 : *dimension*
l \pagelastfilstretch
 [=] *dimension*
 : *dimension*
l \pagelastheight
 [=] *dimension*
 : *dimension*
l \pagelastshrink
 [=] *dimension*
 : *dimension*
l \pagelaststretch
 [=] *dimension*
 : *dimension*
t \pageshrink
 [=] *dimension*
 : *dimension*
t \pagestretch
 [=] *dimension*
 : *dimension*
t \pagetotal
 [=] *dimension*
 : *dimension*
l \pagevsize
 [=] *dimension*
 : *dimension*

76 parameter

l \alignmark
l \parametermark

77 penalty

l \hpenalty
 integer
t \penalty
 integer
l \vpenalty
 integer

78 prefix

l \aliased
l \constant
l \constrained
l \deferred
l \enforced
l \frozen
t \global
l \immediate
l \immutable
l \inherited
l \instance
t \long
l \mutable
l \noaligned
t \outer
l \overloaded
l \permanent
e \protected
l \retained
l \semiprotected
l \tolerant
l \untraced

79 register

l \attribute
 (index | *box*) [=] *integer*
 > (index | *box*) : *integer*
t \count
 see \attribute
t \dimen
 (index | *box*) [=] *dimension*
 > (index | *box*) : *dimension*
l \float
 (index | *box*) [=] *float*
 > (index | *box*) : *float*
t \muskip
 (index | *box*) [=] *muglue*
 > (index | *box*) : *muglue*
t \skip
 (index | *box*) [=] *glue*
 > (index | *box*) : *glue*

t \toks
    (index | *box*) [=] { *tokens* }
  > (index | *box*) : { *tokens* }

80 relax

l \norelax
t \relax

81 removeitem

t \unboundary
t \unkern
t \unpenalty
t \unskip

82 setbox

t \setbox
    (index | *box*) [=]

83 setfont

t \nullfont

84 shorthanddef

l \Umathchardef
    \cs *integer*
l \Umathdictdef
    \cs *integer integer*
l \attributedef
    \cs *integer*
t \chardef
    \cs *integer*
t \countdef
    \cs *integer*
t \dimendef
    \cs *integer*
l \dimensiondef
    \cs *integer*
l \floatdef
    \cs *integer*
l \fontspecdef
    \cs (*font* | *integer*)
l \gluespecdef
    \cs *integer*
l \integerdef
    \cs *integer*
l \luadef
    \cs *integer*
t \mathchardef
    \cs *integer*
l \mugluespecdef
    \cs *integer*
t \muskipdef
    \cs *integer*
l \parameterdef
    \cs *integer*
l \positdef
    \cs *integer*
t \skipdef
    \cs *integer*
t \toksdef
    \cs *integer*

85 someitem

t \badness
    [=] *integer*
  : *integer*
e \currentgrouplevel
    [=] *integer*
  : *integer*
e \currentgrouptype
    [=] *integer*
  : *integer*
e \currentifbranch
    [=] *integer*
  : *integer*
e \currentiflevel
    [=] *integer*
  : *integer*
e \currentiftype
    [=] *integer*
  : *integer*
l \currentloopiterator
    [=] *integer*
  : *integer*
l \currentloopnesting
    [=] *integer*
  : *integer*
e \currentstacksize
    [=] *integer*
  : *integer*

e \dimexpr
tokens\relax [=] *dimension*
> *tokens*\relax : *dimension*

l \dimexpression
tokens\relax [=] *dimension*
> *tokens*\relax : *dimension*

l \floatexpr
tokens\relax [=] *float*
> *tokens*\relax : *float*

l \fontcharba
integer [=] *dimension*
> *integer* : *dimension*

e \fontchardp
integer [=] *dimension*
> *integer* : *dimension*

e \fontcharht
integer [=] *dimension*
> *integer* : *dimension*

e \fontcharic
integer [=] *dimension*
> *integer* : *dimension*

l \fontcharta
integer [=] *dimension*
> *integer* : *dimension*

e \fontcharwd
integer [=] *dimension*
> *integer* : *dimension*

l \fontid
(*font* | *integer*) [=] *integer*
> (*font* | *integer*) : *integer*

l \fontmathcontrol
see \fontid

l \fontspecid
see \fontid

l \fontspecifiedsize
see \fontid

l \fontspecscale
see \fontid

l \fontspecslant
see \fontid

l \fontspecweight
see \fontid

l \fontspecxscale
see \fontid

l \fontspecyscale
see \fontid

l \fonttextcontrol
see \fontid

e \glueexpr
tokens\relax [=] *glue*
> *tokens*\relax : *glue*

e \glueshrink
glue [=] *dimension*
> *glue* : *dimension*

e \glueshrinkorder
glue [=] *dimension*
> *glue* : *dimension*

e \gluestretch
glue [=] *integer*
> *glue* : *integer*

e \gluestretchorder
glue [=] *integer*
> *glue* : *integer*

e \gluetomu
glue [=] *glue*
> *glue* : *glue*

l \glyphxscaled
[=] *integer*
: *integer*

l \glyphyscaled
[=] *integer*
: *integer*

l \indexofcharacter
integer [=] *integer*
> *integer* : *integer*

l \indexofregister
integer [=] *integer*
> *integer* : *integer*

t \inputlineno
[=] *integer*
: *integer*

l \insertprogress
integer [=] *dimension*
> *integer* : *dimension*

l \lastarguments
[=] *integer*
: *integer*

l \lastatomclass
[=] *integer*
: *integer*

l \lastboundary
[=] *integer*
: *integer*

l \lastchkdimension
[=] *dimension*
: *dimension*

l \lastchknumber
  [=] *integer*
  : *integer*

t \lastkern
  [=] *dimension*
  : *dimension*

l \lastleftclass
  [=] *integer*
  : *integer*

l \lastloopiterator
  [=] *integer*
  : *integer*

l \lastnodesubtype
  [=] *integer*
  : *integer*

e \lastnodetype
  [=] *integer*
  : *integer*

l \lastpageextra
  [=] *dimension*
  : *dimension*

l \lastparcontext
  [=] *integer*
  : *integer*

t \lastpenalty
  [=] *integer*
  : *integer*

l \lastrightclass
  [=] *integer*
  : *integer*

t \lastskip
  [=] *glue*
  : *glue*

l \leftmarginkern
  [=] *dimension*
  : *dimension*

l \luatexrevision
  [=] { *tokens* }
  : { *tokens* }

l \luatexversion
  [=] { *tokens* }
  : { *tokens* }

l \mathatomglue
  [=] *glue*
  : *glue*

l \mathcharclass
  integer [=] *integer*
  > *integer* : *integer*

l \mathcharfam
  integer [=] *integer*
  > *integer* : *integer*

l \mathcharslot
  integer [=] *integer*
  > *integer* : *integer*

l \mathmainstyle
  [=] *integer*
  : *integer*

l \mathscale
  [=] *integer*
  : *integer*

l \mathstackstyle
  [=] *integer*
  : *integer*

l \mathstyle
  [=] *integer*
  : *integer*

l \mathstylefontid
  [=] *integer*
  : *integer*

e \muexpr
  tokens\relax [=] *muglue*
  > *tokens*\relax : *muglue*

e \mutoglue
  muglue [=] *glue*
  > *muglue* : *glue*

l \nestedloopiterator
  [=] *integer*
  : *integer*

l \numericscale
  (*integer* | *float*) [=] *integer*
  > (*integer* | *float*) : *integer*

l \numericscaled
  see \numericscale

e \numexpr
  tokens\relax [=] *integer*
  > *tokens*\relax : *integer*

l \numexpression
  tokens\relax [=] *integer*
  > *tokens*\relax : *integer*

l \overshoot
  [=] *dimension*
  : *dimension*

l \parametercount
  [=] *integer*
  : *integer*

l \parameterindex
  [=] *integer*

: *integer*

e \parshapedimen

integer [=] *dimension*

> *integer* : *dimension*

e \parshapeindent

integer [=] *dimension*

> *integer* : *dimension*

e \parshapelength

[=] *dimension*

: *dimension*

l \previousloopiterator

[=] *integer*

: *integer*

l \rightmarginkern

[=] *dimension*

: *dimension*

l \scaledemwidth

(*font* | *integer*) [=] *dimension*

> (*font* | *integer*) : *dimension*

l \scaledexheight

see \scaledemwidth

l \scaledextraspace

see \scaledemwidth

l \scaledfontcharba

integer [=] *dimension*

> *integer* : *dimension*

l \scaledfontchardp

integer [=] *dimension*

> *integer* : *dimension*

l \scaledfontcharht

integer [=] *dimension*

> *integer* : *dimension*

l \scaledfontcharic

integer [=] *dimension*

> *integer* : *dimension*

l \scaledfontcharta

integer [=] *dimension*

> *integer* : *dimension*

l \scaledfontcharwd

integer [=] *dimension*

> *integer* : *dimension*

l \scaledinterwordshrink

see \scaledemwidth

l \scaledinterwordspace

see \scaledemwidth

l \scaledinterwordstretch

see \scaledemwidth

l \scaledmathaxis

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \scaledmathemwidth

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \scaledmathexheight

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \scaledslantperpoint

see \scaledemwidth

86 specification

e \clubpenalties

[options] *integer* n * (*integer*)

: *integer*

e \displaywidowpenalties

see \clubpenalties

e \interlinepenalties

see \clubpenalties

l \mathbackwardpenalties

see \clubpenalties

l \mathforwardpenalties

see \clubpenalties

l \orphanpenalties

see \clubpenalties

l \parpasses

[options] n * ([adjdemerits *integer*] [adjustspacing *integer*] [adjustspacingstep *integer*] [adjustspacingshrink *integer*] [adjustspacingstretch *integer*] [badness *integer*] [classes *integer*] [callback *integer*] [doubleadjdemerits *integer*] [doublehyphendemerits *integer*] [emergencystretch *dimension*] [extrahyphenpenalty *integer*] [finalhyphendemerits *integer*] [identifier *integer*] [ifadjustspacing *integer*] [looseness *integer*] [linebreakcriterium *integer*] [linebreakoptional *integer*] [linepenalty *integer*] [next] [orphanpenalty *integer*] [quit] [skip] [threshold *dimension*] [tolerance *integer*])

: *integer*

t \parshape

[options] *integer* n * (*dimension*

dimension)
: *integer*

e \widowpenalties
see \clubpenalties

87 the

e \detokenize
{ *tokens* }

l \expandeddetokenize
{ *tokens* }

l \protecteddetokenize
{ *tokens* }

l \protectedexpandeddetokenize
{ *tokens* }

t \the
dimension

l \thewithoutunit
quantity

e \unexpanded
{ *tokens* }

88 unhbox

t \unhbox
integer

t \unhcopy
integer

l \unhpack
integer

89 unvbox

l \insertunbox
integer

l \insertuncopy
integer

e \pagediscards

e \splitdiscards

t \unvbox
integer

t \unvcopy
integer

l \unvpack
integer

90 vadjust

t \vadjust
[pre] [post] [baseline] [before] [index *integer*] [after] [attr *integer integer*] [depth (after | before | check | last)] { *tokens* }

91 valign

t \valign
[attr *integer integer*] [callback *integer*] [discard] [noskips] [reverse] [to *dimension*] [spread *dimension*] { *tokens* }

92 vcenter

t \vcenter
[target *integer*] [to *dimension*] [adapt] [attr *integer integer*] [anchor *integer*] [axis *integer*] [shift *dimension*] [spread *dimension*] [source *integer*] [direction *integer*] [delay] [orientation *integer*] [xoffset *dimension*] [xmove *dimension*] [yoffset *dimension*] [ymove *dimension*] [reverse] [retain] [container] [class *integer*] { *tokens* }

93 vmove

t \lower
dimension box

t \raise
dimension box

94 vrule

l \novrule
[attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [left *dimension*] [right *dimension*] [top *dimension*] [bottom *dimension*] [xoffset *dimension*] [yoffset *dimension*] [*font integer*]

[fam *integer*] [char *integer*]

l \srule

see \novrule

l \virtualvrule

[attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [left *dimension*] [right *dimension*] [top *dimension*] [bottom *dimension*] [xoffset *dimension*] [yoffset *dimension*]

t \vrule

see \novrule

95 vskip

t \vfil

t \vfill

t \vfilneg

t \vskip

dimension [plus (*dimension* | fi[n*l])] [minus (*dimension* | fi[n*l])]

t \vss

96 xray

t \show

token

t \showbox

(index | *box*)

e \showgroups

e \showifs

t \showlists

e \showstack

t \showthe

quantity

e \showtokens

{ *tokens* }

Rationale

Some words about the why and how it came. One of the early adopters of ConTEXt was Taco Hoekwater and we spent numerous trips to TEX meetings all over the globe. He was also the only one I knew who had read the TEX sources. Because ConTEXt has always been on the edge of what is possible and at that time we both used it for rather advanced rendering, we also ran into the limitations. I'm not talking of TEX features here. Naturally old school TEX is not really geared for dealing with images of all kind, colors in all kind of color spaces, highly interactive documents, input methods like xml, etc. The nice thing is that it offers some escapes, like specials and writes and later execution of programs that opened up lots of possibilities, so in practice there were no real limitations to what one could do. But coming up with a consistent and extensible (multi lingual) user interface was non trivial, because it had an impact in memory usage and performance. A lot could be done given some programming, as ConTEXt MkII proves, but it was not always pretty under the hood. The move to LuaTEX and MkIV transferred some action to Lua, and because LuaTEX effectively was a ConTEXt related project, we could easily keep them in sync.

Our traveling together, meeting several times per year, and eventually email and intense LuaTEX developments (lots of Skype sessions) for a couple of years, gave us enough opportunity to discuss all kind of nice features not present in the engine. The previous century we discussed lots of them, rejected some, stayed with others, and I admit that forgot about most of the arguments already. Some that we did was already explored in `eetex`, some of those ended up in LuaTEX, and eventually what we have in LuaMetaTEX can been seen as the result of years of programming in TEX, improving macros, getting more performance and efficiency out of existing ConTEXt code and inspiration that we got out of the ConTEXt community, a demanding lot, always willing to experiment with us.

Once I decided to work on LuaMetaTEX and bind its source to the ConTEXt distribution so that we can be sure that it won't get messed up and might interfere with the ConTEXt expectations, some more primitives saw their way into it. It is very easy to come up with all kind of bells and whistles but it is equally easy to hurt performance of an engine and what might go unnoticed in simple tests can really affect a macro package that depends on stability. So, what I did was mostly looking at the ConTEXt code and wondering how to make some of the low level macros look more natural, also because I know that there are users who look into these sources. We spend a lot of time making them look consistent and nice and the nicer the better. Getting a better performance was seldom an argument because much is already as fast as can be so there is not that much to gain, but less clutter in tracing was an argument for some new primitives. Also, the fact that we soon might need to fall back on our phones to use TEX a smaller memory footprint and less byte shuffling also was a consideration. The LuaMetaTEX memory footprint is somewhat smaller than the LuaTEX footprint. By binding LuaMetaTEX to ConTEXt we can also guarantee that the combinations works as expected.

I'm aware of the fact that ConTEXt is in a somewhat unique position. First of all it has always been kind of cutting edge so its users are willing to experiment. There are users who immediately update and run tests, so bugs can and will be fixed fast. Already for a long time the community has an convenient infrastructure for updating and the build farm for generating binaries (also for other engines) is running smoothly.

Then there is the ConTEXt user interface that is quite consistent and permits extensions with staying backward compatible. Sometimes users run into old manuals or examples and then complain that ConTEXt is not compatible but that then involves obsolete technology: we no longer need font and input encodings and font definitions are different for OpenType fonts. We always had an abstract backend model, but nowadays pdf is kind of dominant and drives a lot of expectations. So, some of the MkII commands are gone and MkIV has some more. Also, as MetaPost evolved that department in ConTEXt also evolved. Think of it like cars: soon all are electric so one cannot expect a hole to poor in some fluid but gets a (often incompatible) plug instead. And buttons became touch panels. There is no need to use much force to steer or brake. Navigation is different, as are many controls. And do we need to steer ourselves a decade from now?

So, just look at TEX and ConTEXt in the same way. A system from the nineties in the previous century differs from one three decades later. Demands differ, input differs, resources change, editing and processing moves on, and so on. Manuals, although still being written are seldom read from cover to cover because online searching replaced them. And who buys books about programming? So LuaMetaTEX, while still being TEX also moves on, as do the way we do our low level coding. This makes sense because the original TEX ecosystem was not made with a huge and complex macro package in mind, that just happened. An author was supposed to make a style for each document. An often used argument for using another macro package over ConTEXt was that the later evolved and other macro packages would work the same forever and not change from the perspective of the user. In retrospect those arguments were somewhat strange because the world, computers, users etc. do change. Standards come and go, as do software politics and preferences. In many aspects the TEX community is not different from other large software projects, operating system wars, library devotees, programming language addicts, paradigm shifts. But, don't worry, if you don't like LuaMetaTEX and its new primitives, just forget about them. The other engines will be there forever and are a safe bet, although LuaTEX already stirred up the pot I guess. But keep in mind that new features in the latest greatest ConTEXt version will more and more rely on LuaMetaTEX being used; after all that is where it's made for. And this manual might help understand its users why, where and how the low level code differs between MkII, MkIV and LMTX.

Can we expect more new primitives than the ones introduced here? Given the amount of time I spent on experimenting and considering what made sense and what not, the answer probably is “no”, or at least “not that much”. As in the past no user ever requested the kind of primitives that were added, I don't expect users to come up with requests in the future either. Of course, those more closely related to ConTEXt development look at it from the other end. Because it's there where the low level action really is, demands might still evolve.

Basically there are wo areas where the engine can evolve: the programming part and the rendering. In this manual we focus on the programming and writing the manual sort of influences how details get filled in. Rendering in more complex because there heuristics and usage plays a more dominant role. Good examples are the math, par and page builder. They were extended and features were added over time but improved rendering came later. Not all extensions are critical, some are there (and got added) in order to write more readable code but there is only so much one can do in that area. Occasionally a feature pops up that is a side effect of a challenge. No matter what gets added it might not affect complexity too much and definitely not impact performance significantly!

Hans Hagen
Hasselt NL

To be checked primitives

additionalpageskip
adjustspacing
adjustspacingshrink
adjustspacingstep
adjustspacingstretch
aligncontent
alignmentcellsource
alignmentwrapsource
allcrampedstyles
alldisplaystyles
allmainstyles
allscriptscriptstyles
allscriptstyles
allsplitstyles
alltextstyles
alluncrampedstyles
allunsplitstyles
amcode
automaticdiscretionary
automatichyphenpenalty
automigrationmode
autoparagraphmode
boundary
boxadapt
boxanchor
boxanchors
boxattribute
boxdirection
boxfreeze
boxgeometry
boxlimitate
boxorientation
boxrepack
boxshift
boxshrink
boxsource
boxstretch
boxtarget
boxtotal
boxvadjust
boxxmove
boxxoffset
boxymove
boxyoffset
cfcode
clearmarks
crampeddisplaystyle
crampedscriptscriptstyle
crampedscriptstyle
crampedtextstyle
currentmarks
dbox
discretionaryoptions
dpack
dsplit
efcode
emergencyleftskip
emergencyrightskip
everybeforepar
everytab
exceptionpenalty
explicithyphenpenalty
firstvalidlanguage
flushmarks
fontcharba
fontcharta
fontid
fontspecdef
fontspecid
fontspecifiedname
fontspecifiedsize
fontspecscale
fontspecslant
fontspecweight
fontspecxscale
fontspecyscale
fonttextcontrol
gleaders
glyphdatafield
glyphoptions
glyphscale
glyphscriptfield
glyphscriptscale
glyphscriptscriptscale
glyphslant
glyphstatefield
glyphtextscale
glyphweight
glyphxoffset
glyphxscale
glyphxscaled
glyphyoffset
glyphyscale
glyphyscaled

hccode
hjcode
hmcode
holdingmigrations
hpack
hpenalty
hyphenationmin
hyphenationmode
ifinsert
ifparameters
ignoredepthcriterion
indexofcharacter
indexofregister
inherited
initialpageskip
initialtopskip
insertbox
insertcopy
insertdepth
insertdistance
insertheight
insertheights
insertlimit
insertmaxdepth
insertmode
insertmultiplier
insertpenalty
insertprogress
insertstorage
insertstoring
insertunbox
insertuncopy
insertwidth
lastatomclass
lastboundary
lastchkdimension
lastchknumber
lastleftclass
lastnodesubtype
lastpageextra
lastparcontext
lastrightclass
leftmarginkern
linebreakcriterion
linebreakoptional
linebreakpasses
linedirection
localbrokenpenalty
localinterlinepenalty
localleftbox
localleftboxbox
localmiddlebox
localmiddleboxbox
localpretolerance
localrightbox
localrightboxbox
localtolerance
lpcode
luabytecode
luabytecodecall
luacopyinputnodes
luadef
luafunction
luafunctioncall
mutable
nestedloopiterator
noaligned
noatomruling
noboundary
nohrule
normalizelinemode
normalizeparmode
nosubprescript
nosubscript
nosuperprescript
nosuperscript
novrule
optionalboundary
orphanpenalties
orphanpenalty
outputbox
overloaded
overloadmode
overshoot
pageboundary
pagedepth
pageexcess
pageextragoal
pagefistretch
pagelastdepth
pagelastfilllstretch
pagelastfillstretch
pagelastfilstretch
pagelastheight
pagelastshrink
pagelaststretch
pagevsize
parametercount

```
parattribute
pardirection
parfillleftskip
parfillrightskip
parinitleftskip
parinitrightskip
parpasses
pettymuskip
postexhyphenchar
posthyphenchar
postinlinepenalty
postshortinlinepenalty
prebinoppenalty
predisplaygapfactor
preexhyphenchar
prehyphenchar
preinlinepenalty
prerelpenalty
preshortinlinepenalty
primescript
protrudechars
protrusionboundary
quitvmode
rightmarginkern
rpcode
scaledemwidth
scaledexheight
scaledextraspace
scaledfontcharba
scaledfontchardp
scaledfontcharht
scaledfontcharic
scaledfontcharta
scaledfontcharwd
scaledfontdimen
scaledinterwordshrink
scaledinterwordspace
scaledinterwordstretch
scaledslantperpoint
setfontid
shapingpenaltiesmode
shapingpenalty
shiftedsubprescript
shiftedsubscript
shiftedsuperprescript
shiftedsuperscript
shortinlineorphanpenalty
singlelinepenalty
snapshotpar
spacefactormode
spacefactorshrinklimit
spacefactorstretchlimit
srule
subprescript
subscript
superprescript
superscript
supmarkmode
tabsize
textdirection
tinymuskip
tocharacter
tpack
tracingadjusts
tracingalignments
tracingfonts
tracingfullboxes
tracinghyphenation
tracinginserts
tracinglevels
tracinglists
tracingmarks
tracingnodes
tracingpasses
tracingpenalties
tsplit
uleaders
undent
unhpack
unvpack
variablefam
virtualhrule
virtualvrule
vpack
vpenalty
wordboundary
wrapuppar
```

Indexed primitives

expandedafter
expandeddetokenize
expandedendless
expandedloop
expandedrepeat
expandparameter
expandtoken
expandtoks
explicitdiscretionary
explicititaliccorrection
explicitspace
fi
float
floatdef
floatexpr
font
formatname
frozen
futurecsname
futuredef
futureexpand
futureexpandis
futureexpandisap
futurelet
gdef
gdefcsname
glet
gletcsname
glettonothing
global
globaldefs
glueexpr
glueshrink
glueshrinkorder
gluespecdef
gluestretch
gluestretchorder
glyph
gtoksapp
gtokspre
if
ifabsdim
ifabsfloat
ifabsnum
ifarguments
ifboolean
ifcase
ifcat
ifchkdim
ifchkdimension
ifchknum
ifchknumber
ifcmpdim
ifcmpnum
ifcondition
ifcsname
ifcstok
ifdefined
ifdim
ifdimexpression
ifdimval
ifempty
iffalse
ifflags
iffloat
iffontchar
ifhaschar
ifhastok
ifhastoks
ifhasxtoks
ifhbox
ifhmode
ifinalignment
ifincsname
ifinner
ifintervaldim
ifintervalfloat
ifintervalnum
iflastnamedcs
ifmathparameter
ifmathstyle
ifmmode
ifnum
ifnumexpression
ifnumval
ifodd
ifparameter
ifrelax
iftok
iftrue
ifvbox
ifvmode
ifvoid
ifx
ifzerodim
ifzerofloat
ifzeronum
ignorearguments

ignorenestedupto
ignorepars
ignorerest
ignorespaces
ignoreupto
immediate
immediateassigned
immediateassignment
immutable
initcatcodetable
input
inputlineno
instance
integerdef
interactionmode
jobname
lastarguments
lastloopiterator
lastnamedcs
lccode
let
letcharcode
letcsname
letfrozen
letprotected
lettolastnamedcs
lettonothing
localcontrol
localcontrolled
localcontrolledendless
localcontrolledloop
localcontrolledrepeat
long
lowercase
luaescapestring
luatexbanner
luatexrevision
luatexversion
mathgroupingmode
meaning
meaningasis
meaningful
meaningfull
meaningles
meaningless
mugluespecdef
multiply
multiplyby
nonstopmode
norelax
normalunexpanded
nospaces
number
numericscale
numericscaled
numexpr
numexpression
or
orelse
orunless
outer
par
parameterdef
parameterindex
parametermark
parametermode
permanent
positdef
previousloopiterator
protected
protecteddetokenize
protectedexpandeddetokenize
pxdimen
quitloop
quitloopnow
rdivide
rdivideby
relax
retained
retokenized
romannumeral
savecatcodetable
scantextokens
scantokens
scrollmode
semiexpand
semiexpanded
semiprotected
semprotected
skip
special
string
swapcsvalues
the
thewithoutunit
todimension
tohexadecimal
tointeger

tokenized
toksapp
tokspre
tolerant
toscaled
tosparsedimension
tosparsescaled
tracingexpressions
unexpanded
unexpandedendless
unexpandedloop
unexpandedrepeat
unless
unletfrozen
unletprotected
untraced
uppercase
write
xdef
xdefcsname
xtoks
xtoksapp
xtokspre