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 font 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 descriptions out there already. Primitives that relate to typesetting, like those controlling math, fonts, boxes, attributes, directions, catcodes, Lua (functions) etc are not discussed 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. These are not discussed here.

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 datatypes and resources, deal with fonts and languages, manipulate boxes and glyphs, etc. are not discussed here. In this document we concentrate on the programming aspects.

1 \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)

2 \meaningless

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

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

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

3 \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)

4 \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)}

5 \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:

! ! ! ?

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 \aftergroup

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

Multiple `\aftergroup`s are combined:

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

before ! after

8 \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

9 \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.

10 \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)**.

11 \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.

12 \etoksapp

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

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

13 \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.

14 \etokspre

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

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

15 \gtoksapp

This is the global variant of `\toksapp`.

16 \xtoksapp

This is the global variant of `\etoksapp`.

17 \gtokspre

This is the global variant of `\tokspre`.

18 \xtokspre

This is the global variant of `\etokspre`.

19 \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
```

20 \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.

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 \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.

23 \futureexpand

This primitive can be used as an alternative to a `\futurelet` approach, which is where the name comes from.[1]

```
\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:

(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>]
```

---

[1] In the engine primitives that have similar behavior are grouped in commands that are then dealt with together, code wise.

```
[\temp {two}]
[\expandafter\temp\space <one>]
[\expandafter\temp\space {two}]
```

This gives us:

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

24 `\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]

25 `\futureexpandisap`

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

26 `\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.

27 `\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).

28 `\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.

29 `\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:

```
\edef\tempA{\expandtoken  9 `X} \meaning\tempA
\edef\tempB{\expandtoken 10 `X} \meaning\tempB
\edef\tempC{\expandtoken 11 `X} \meaning\tempC
\edef\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: Y] [AC: Y] [AD: Y] [BC: Y] [BD: Y] [CD: Y], and in case you wonder, the characters with catcode 10 are spaces, while those with code 9 are ignored.

30 `\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.

31 \expand

Normally a protected macro will not be expanded inside for instance an `\edef` but there is a way out:[2]

```
\edef\temp       {\doifelse{a}{b}{c}{d}} \meaning\temp \crlf
\edef\temp{\expand\doifelse{a}{b}{c}{d}} \meaning\temp \par
```

In the second case, the `\doifelse` command *is* expanded, but keep in mind that this only makes sense when the body of such a macro is expandable. This is the case in ConTEXt LMTX, but not in MkIV.

```
macro:\doifelse {a}{b}{c}{d}
macro:d
```

32 \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.

33 \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).

34 \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}
```

---

[2] This primitive is dedicated to Hans vd Meer who was playing with the unprotected version of `\doifelse` and wondered about the reason for it not being expandable in the first place.

```
\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.

35 \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

36 \scantokens

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

37 \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.

38 \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.

39 \tointeger

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

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

40 \tohexadecimal

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

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

41 \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
```

42 \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
```

43 \tosparsedimension

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

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

44 \tosparsescaled

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

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

45 \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 and the period signals this:

```
1323
1323000
1323
13230
```

When there is a period (indicating the fraction) the result is an integer (count) that has the multiplier 1000 applied.

46 \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.

47 \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`.

48 \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 ?
```

49 \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
```

50 \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).

51 \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}  \crlf
          \def\B#1{\string#1} \normalexpanded{\noexpand\B{\A}} \crlf
\protected\def\B#1{\string#1}                          \B{\A}  \par
```

```
\A
!
\A
```

52 \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
```

53 \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.

54 \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]

55 \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.

56 \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).

57 \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

58 \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.

59 \ifvmode

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

60 \ifhmode

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

61 \ifmmode

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

62 \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

63 \ifvoid

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

64 \ifhbox

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

65 `\ifvbox`

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

66 `\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).

67 `\ifeof`

This traditional conditional checks if current pointer into the the file bound to the given index is past the end of file. The read and write channels are not really used in ConTEXt: in MkII we only had one file for all multi-pass data, and in MkIV all file related stuff is dealt with in LuaTEX.

68 `\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).

69 `\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).

70 `\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`.

71 `\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.

72 \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.

73 \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.

74 \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.

75 \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
```

76 \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
```

77 \ifzerodim

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

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

78 \ifzeronum

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

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

79 \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
```

The result is as expected:

```
error   okay   okay
```

80 \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
```

We get:

error   error   okay

81 \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]

82 \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]

83 \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]

84 \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]

85 \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.

86 \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.

87 \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:\overloaded \frozen \let \ifMyTest \iftrue
protected macro:\overloaded \frozen \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?

88 \iffrozen

This conditional checks if a control sequence is frozen:

```
is \iffrozen\MyMacro \else not \fi frozen
```

89 \ifprotected

This conditional checks if a control sequence is protected:

```
is \ifprotected\MyMacro \else not \fi protected
```

90 \ifusercmd

This conditional checks if a control sequence is not one of the primitives:

```
is \ifusercmd\MyMacro \else not \fi a primitive
```

It is not always possible to determine this but it should work okay for regular macros, register allocations and character definitions.

91 \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.

92 \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{}
```

93 \ifboolean

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

94 \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
```

95 \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
```

96 \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
```

97 \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

98 \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)

99 \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
```

100 `\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.

101 `\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.

102 `\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.

103 `\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`.

104 `\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`.

105 `\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).

106 \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.

107 \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[3]

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

---

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

```
\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.

108 `\orunless`

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

109 `\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\MySpecialToken[
\def\DoWhatever{\ifx\NextToken\MySpecialToken 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)

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 \futurecsname

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

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

112 \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
```

113 \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.

114 \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).

115 \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).

116 \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.

117 \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
```

118 \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).[4]

119 \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.

120 \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.[5]

121 \letfrozen

You can explicitly freeze an unfrozen macro:

---

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

[5] The `\outer` and `\long` prefixes are no-ops in LuaMetaTEX and LuaTEX can be configured to ignore them.

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

A redefinition will now give:

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

122 \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.

123 \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
```

124 \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
```

125 \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.

126 \endlocalcontrol

See previous section.

127 \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.

128 \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.

129 \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.

130 \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.

131 \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.

132 \edefcsname

This is the companion of `\edef`:

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

133 \gdefcsname

As with standard TEX we also define global ones:

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

134 \xdefcsname

This is the companion of `\xdef`:

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

135 \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).

136 \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
```

137 \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.

138 \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
```

139 `\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}
```

140 `\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
```

141 `\glettonothing`

This is the global companion of `\lettonothing`.

142 `\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  constant macro:
```

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

143 `\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.

144 \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.

145 \dimensiondef

A variant of `\integerdef` is:

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

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

146 \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.

147 \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.

148 \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.

149 \multiplyby

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

150 \divideby

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

151 \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.

152 \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 }
```

The next section shows a companion primitive.

153 \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
```

```
\constant \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.

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).

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 spend 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.

Hans Hagen
Hasselt NL