low level TeX

conditionals

Contents

1 Preamble

1.1 Introduction

You seldom need the low level conditionals because there are quite some so called support macros available in ConTEXt. For instance, when you want to compare two values (or more accurate: sequences of tokens), you can do this:

```
\doifelse {foo} {bar} {
    the same
} {
    different
}
```

But if you look in the ConTEXt code, you will see that often we use primitives that start with `\if` in low level macros. There are good reasons for this. First of all, it looks familiar when you also code in other languages. Another reason is performance but that is only true in cases where the snippet of code is expanded very often, because TEX is already pretty fast. Using low level TEX can also be more verbose, which is not always nice in a document source. But, the most important reason (for me) is the layout of the code. I often let the look and feel of code determine the kind of coding. This also relates to the syntax highlighting that I am using, which is consistent for TEX, MetaPost, Lua, etc. and evolved over decades. If code looks bad, it probably is bad. Of course this doesn't mean all my code looks good; you're warned. In general we can say that I often use `\if...` when coding core macros, and `\doifelse...` macros in (document) styles and modules.

In the sections below I will discuss the low level conditions in TEX. For the often more convenient ConTEXt wrappers you can consult the source of the system and support modules, the wiki and/or manuals.

Some of the primitives shown here are only available in LuaTEX, and some only in LuaMetaTEX. We could do without them for decades but they were added to these engines because of convenience and, more important, because then made for nicer code. Of course there's also the fun aspect. This manual is not an invitation to use these very low level primitives in your document source. The ones that probably make most sense are `\ifnum`, `\ifdim` and `\ifcase`. The others are often wrapped into support macros that are more convenient.

In due time I might add more examples and explanations. Also, maybe some more tests will show up as part of the LuaMetaTEX project.

1.2 Number and dimensions

Numbers and dimensions are basic data types in TEX. When you enter one, a number is just that but a dimension gets a unit. Compare:

```
1234
1234pt
```

If you also use MetaPost, you need to be aware of the fact that in that language there are not really dimensions. The `post` part of the name implies that eventually a number becomes a PostScript unit which represents a base point (`bp`) in TEX. When in MetaPost you entry `1234pt` you actually multiply `1234` by the variable `pt`. In TEX on the other hand, a unit like `pt` is one of the keywords that gets parsed. Internally dimensions are also numbers and the unit (keyword) tells the scanner what multiplier to use. When that multiplier is one, we're talking of scaled points, with the unit `sp`.

```
\the\dimexpr 12.34pt \relax
\the\dimexpr 12.34sp \relax
\the\dimexpr 12.99sp \relax
\the\dimexpr 1234sp  \relax
\the\numexpr 1234    \relax
```

```
12.34pt
0.00018pt
0.00018pt
0.01883pt
1234
```

When we serialize a dimension it always shows the dimension in points, unless we serialize it as number.

```
\scratchdimen1234sp
\number\scratchdimen
\the\scratchdimen
```

```
1234
0.01883pt
```

When a number is scanned, the first thing that is taken care of is the sign. In many cases, when TEX scans for something specific it will ignore spaces. It will happily accept multiple signs:

```
\number +123
\number +++123
\number + + + 123
\number +-+-+123
\number --123
\number ---123
```

```
123
123
123
123
123
-123
```

Watch how the negation accumulates. The scanner can handle decimal, hexadecimal and octal numbers:

```
\number -123
\number -"123
\number -'123
```

```
-123
-291
-83
```

A dimension is scanned like a number but this time the scanner checks for upto three parts: an either or not signed number, a period and a fraction. Here no number means zero, so the next is valid:

```
\the\dimexpr  . pt \relax
\the\dimexpr 1. pt \relax
\the\dimexpr  .1pt \relax
```

```
\the\dimexpr 1.1pt \relax
```

0.0pt
1.0pt
0.1pt
1.1pt

Again we can use hexadecimal and octal numbers but when these are entered, there can be no fractional part.

```
\the\dimexpr  16 pt \relax
\the\dimexpr "10 pt \relax
\the\dimexpr '20 pt \relax
```

16.0pt
16.0pt
16.0pt

The reason for discussing numbers and dimensions here is that there are cases where when TEX expects a number it will also accept a dimension. It is good to know that for instance a macro defined with `\chardef` or `\mathchardef` also is treated as a number. Even normal characters can be numbers, when prefixed by a ` (backtick).

The maximum number in TEX is 2147483647 so we can do this:

```
\scratchcounter2147483647
```

but not this

```
\scratchcounter2147483648
```

as it will trigger an error. A dimension can be positive and negative so there we can do at most:

```
\scratchdimen  1073741823sp
```

```
\scratchdimen1073741823sp
\number\scratchdimen
\the\scratchdimen
\scratchdimen16383.99998pt
\number\scratchdimen
\the\scratchdimen
```

1073741823

16383.99998pt
1073741823
16383.99998pt

We can also do this:

```
\scratchdimen16383.99999pt
\number\scratchdimen
\the\scratchdimen
```

1073741823
16383.99998pt

but the next one will fail:

```
\scratchdimen16383.9999999pt
```

Just keep in mind that TEX scans both parts as number so the error comes from checking if those numbers combine well.

```
\ifdim 16383.99999  pt = 16383.99998  pt the same \else different \fi
\ifdim 16383.999979 pt = 16383.999980 pt the same \else different \fi
\ifdim 16383.999987 pt = 16383.999991 pt the same \else different \fi
```

Watch the difference in dividing, the / rounds, while the : truncates.

the same
the same
the same

You need to be aware of border cases, although in practice they never really are a problem:

```
\ifdim \dimexpr16383.99997 pt/2\relax = \dimexpr 16383.99998 pt/2\relax
    the same \else different
\fi
\ifdim \dimexpr16383.99997 pt:2\relax = \dimexpr 16383.99998 pt:2\relax
    the same \else different
\fi
```

different
the same

```
\ifdim \dimexpr1.99997 pt/2\relax = \dimexpr 1.99998 pt/2\relax
```

```
    the same \else different
\fi
\ifdim \dimexpr1.99997 pt:2\relax = \dimexpr 1.99998 pt:2\relax
    the same \else different
\fi
```

different
the same

```
\ifdim \dimexpr1.999999 pt/2\relax = \dimexpr 1.9999995 pt/2\relax
    the same \else different
\fi
\ifdim \dimexpr1.999999 pt:2\relax = \dimexpr 1.9999995 pt:2\relax
    the same \else different
\fi
```

the same
the same

This last case demonstrates that at some point the digits get dropped (still assuming that the fraction is within the maximum permitted) so these numbers then are the same. Anyway, this is not different in other programming languages and just something you need to be aware of.

2 TEX primitives

2.1 \if

I seldom use this one. Internally TEX stores (and thinks) in terms of tokens. If you see for instance `\def` or `\dimen` or `\hbox` these all become tokens. But characters like `A` or @ also become tokens. In this test primitive all non-characters are considered to be the same. In the next examples this is demonstrated.

```
[\if AB yes\else nop\fi]
[\if AA yes\else nop\fi]
[\if CDyes\else nop\fi]
[\if CCyes\else nop\fi]
[\if\dimen\font yes\else nop\fi]
[\if\dimen\font yes\else nop\fi]
```

Watch how spaces after the two characters are kept: [nop] [yes] [nop] [yes] [yes] [yes]. This primitive looks at the next two tokens but when doing so it expands. Just look at the following:

```
\def\AA{AA}%
\def\AB{AB}%
[\if\AA yes\else nop\fi]
[\if\AB yes\else nop\fi]
```

We get: [yes] [nop].

2.2 \ifcat

In TEX characters (in the input) get interpreted according to their so called catcodes. The most common are letters (alphabetic) and and other (symbols) but for instance the backslash has the property that it starts a command, the dollar signs trigger math mode, while the curly braced deal with grouping. If for instance either or not the ampersand is special (for instance as column separator in tables) depends on the macro package.

```
[\ifcat AB yes\else nop\fi]
[\ifcat AA yes\else nop\fi]
[\ifcat CDyes\else nop\fi]
[\ifcat CCyes\else nop\fi]
[\ifcat C1yes\else nop\fi]
[\ifcat\dimen\font yes\else nop\fi]
[\ifcat\dimen\font yes\else nop\fi]
```

This time we also compare a letter with a number: [yes] [yes] [yes] [yes] [nop] [yes] [yes]. In that case the category codes differ (letter vs other) but in this test comparing the letters result in a match. This is a test that is used only once in ConTEXt and even that occasion is dubious and will go away.

You can use \noexpand to prevent expansion:

```
\def\A{A}%
\let\B B%
\def\C{D}%
\let\D D%
[\ifcat\noexpand\A Ayes\else nop\fi]
[\ifcat\noexpand\B Byes\else nop\fi]
[\ifcat\noexpand\C Cyes\else nop\fi]
[\ifcat\noexpand\C Dyes\else nop\fi]
```

```
[\ifcat\noexpand\D Dyes\else nop\fi]
```

We get: [nop] [yes] [nop] [nop] [yes], so who still thinks that TEX is easy to understand for a novice user?

2.3 \ifnum

This condition compares its argument with another one, separated by an <, = or > character.

```
\ifnum\scratchcounter<0
    less than
\else\ifnum\scratchcounter>0
    more than
\else
    equal to
\fi zero
```

This is one of these situations where a dimension can be used instead. In that case the dimension is in scaled points.

```
\ifnum\scratchdimen<0
    less than
\else\ifnum\scratchdimen>0
    more than
\else
    equal to
\fi zero
```

Of course this equal treatment of a dimension and number is only true when the dimension is a register or box property.

2.4 \ifdim

This condition compares one dimension with another one, separated by an <, = or > sign.

```
\ifdim\scratchdimen<0pt
    less than
\else\ifdim\scratchdimen>0pt
    more than
\else
```

```
    equal to
\fi zero
```

While when comparing numbers a dimension is a valid quantity but here you cannot mix them: something with a unit is expected.

2.5 \ifodd

This one can come in handy, although in ConTEXt it is only used in checking for an odd of even page number.

```
\scratchdimen  3sp
\scratchcounter4

\ifodd\scratchdimen   very \else not so \fi odd
\ifodd\scratchcounter very \else not so \fi odd
```

As with the previously discussed `\ifnum` you can use a dimension variable too, which is then interpreted as representing scaled points. Here we get:

very odd
not so odd

2.6 \ifvmode

This is a rather trivial check. It takes no arguments and just is true when we're in vertical mode. Here is an example:

```
\hbox{\ifvmode\else\par\fi\ifvmode v\else h\fi mode}
```

We're always in horizontal mode and issuing a `\par` inside a horizontal box doesn't change that, so we get: hmode.

2.7 \ifhmode

As with `\ifvmode` this one has no argument and just tells if we're in vertical mode.

```
\vbox {
    \noindent \ifhmode h\else v\fi mode
    \par
    \ifhmode h\else \noindent v\fi mode
}
```

You can use it for instance to trigger injection of code, or prevent that some content (or command) is done more than once:

```
hmode

vmode
```

2.8 \ifmmode

Math is something very TEX so naturally you can check if you're in math mode. here is an example of using this test:

```
\def\enforcemath#1{\ifmmode#1\else$ #1 $\fi}
```

Of course in reality macros that do such things are more advanced than this one.

2.9 \ifinner

```
\def\ShowMode
  {\ifhmode      \ifinner inner \fi hmode
   \else\ifvmode \ifinner inner \fi vmode
   \else\ifmmode \ifinner inner \fi mmode
   \else         \ifinner inner \fi unset
   \fi\fi\fi}

\ShowMode \ShowMode

\vbox{\ShowMode}

\hbox{\ShowMode}

$\ShowMode$

$$\ShowMode$$
```

The first line has two tests, where the first one changes the mode to horizontal simply because a text has been typeset. Watch how display math is not inner.

vmode hmode
inner vmode
inner hmode
innermmode
innermmode

By the way, moving the `\ifinner` test outside the branches (to the top of the macro) won't work because once the word `inner` is typeset we're no longer in vertical mode, if we were at all.

2.10 \ifvoid

A box is one of the basic concepts in TEX. In order to understand this primitive we present four cases:

```
\setbox0\hbox{}         \ifvoid0 void \else content \fi
\setbox0\hbox{123}      \ifvoid0 void \else content \fi
\setbox0\hbox{} \box0   \ifvoid0 void \else content \fi
\setbox0\hbox to 10pt{} \ifvoid0 void \else content \fi
```

In the first case, we have a box which is empty but it's not void. It helps to know that internally an hbox is actually an object with a pointer to a linked list of nodes. So, the first two can be seen as:

```
hlist -> [nothing]
hlist -> 1 -> 2 -> 3 -> [nothing]
```

but in any case there is a hlist. The third case puts something in a hlist but then flushes it. Now we have not even the hlist any more; the box register has become void. The last case is a variant on the first. It is an empty box with a given width. The outcome of the four lines (with a box flushed in between) is:

content
content

void
content

So, when you want to test if a box is really empty, you need to test also its dimensions, which can be up to three tests, depending on your needs.

```
\setbox0\emptybox                  \ifvoid0 void\else content\fi
\setbox0\emptybox        \wd0=10pt \ifvoid0 void\else content\fi
\setbox0\hbox to 10pt {}           \ifvoid0 void\else content\fi
\setbox0\hbox         {} \wd0=10pt \ifvoid0 void\else content\fi
```

Setting a dimension of a void voix (empty) box doesn't make it less void:

void

void
content
content

2.11 \ifhbox

This test takes a box number and gives true when it is an hbox.

2.12 \ifvbox

This test takes a box number and gives true when it is an vbox. Both a `\vbox` and `\vtop` are vboxes, the difference is in the height and depth and the baseline. In a `\vbox` the last line determines the baseline

vbox or vtop

vtop or vbox

And in a `\vtop` the first line takes control:

vbox or vtop

vtop or vbox

but, once wrapped, both internally are just vlists.

2.13 \ifx

This test is actually used a lot in ConTEXt: it compares two token(list)s:

```
                  \ifx a b  Y\else N\fi
                  \ifx ab   Y\else N\fi
\def\A {a}\def\B{b}\ifx \A\B Y\else N\fi
\def\A{aa}\def\B{a}\ifx \A\B Y\else N\fi
\def\A {a}\def\B{a}\ifx \A\B Y\else N\fi
```

Here the result is: “NNNNY”. It does not expand the content, if you want that you need to use an `\edef` to create two (temporary) macros that get compared, like in:

```
\edef\TempA{...}\edef\TempB{...}\ifx\TempA\TempB ...\else ...\fi
```

2.14 `\ifeof`

This test checks if a the pointer in a given input channel has reached its end. It is also true when the file is not present. The argument is a number which relates to the `\openin` primitive that is used to open files for reading.

2.15 `\iftrue`

It does what it says: always true.

2.16 `\iffalse`

It does what it says: always false.

2.17 `\ifcase`

The general layout of an `\ifcase` tests is as follows:

```
\ifcase<number>
    when zero
\or
    when one
\or
    when two
\or
    ...
\else
    when something else
\fi
```

As in other places a number is a sequence of signs followed by one of more digits

3 ε-TEX primitives

3.1 `\ifdefined`

This primitive was introduced for checking the existence of a macro (or primitive) and with good reason. Say that you want to know if `\MyMacro` is defined? One way to do that is:

```
\ifx\MyMacro\undefined
    {\bf undefined indeed}
\fi
```

This results in: **undefined indeed**, but is this macro really undefined? When TEX scans your source and sees a the escape character (the forward slash) it will grab the next characters and construct a control sequence from it. Then it finds out that there is nothing with that name and it will create a hash entry for a macro with that name but with no meaning. Because `\undefined` is also not defined, these two macros have the same meaning and therefore the `\ifx` is true. Imagine that you do this many times, with different macro names, then your hash can fill up. Also, when a user defined `\undefined` you're suddenly get a different outcome.

In order to catch the last problem there is the option to test directly:

```
\ifdefined\MyOtherMacro \else
    {\bf also undefined}
\fi
```

This (or course) results in: **also undefined**, but the macro is still sort of defined (with no meaning). The next section shows how to get around this.

3.2 \ifcsname

A macro is often defined using a ready made name, as in:

```
\def\OhYes{yes}
```

The name is made from characters with catcode letter which means that you cannot use for instance digits or underscores unless you also give these characters that catcode, which is not that handy in a document. You can however use `\csname` to define a control sequence with any character in the name, like:

```
\expandafter\def\csname Oh Yes : 1\endcsname{yes}
```

Later on you can get this one with `\csname`:

```
\csname Oh Yes : 1\endcsname
```

However, if you say:

```
\csname Oh Yes : 2\endcsname
```

you won't get some result, nor a message about an undefined control sequence, but the name triggers a define anyway, this time not with no meaning (undefined) but as equivalent to `\relax`, which is why

```
\expandafter\ifx\csname Oh Yes : 2\endcsname\relax
    {\bf relaxed indeed}
\fi
```

is the way to test its existence. As with the test in the previous section, this can deplete the hash when you do lots of such tests. The way out of this is:

```
\ifcsname Oh Yes : 2\endcsname \else
    {\bf unknown indeed}
\fi
```

This time there is no hash entry created and therefore there is not even an undefined control sequence.

In LuaTeX there is an option to return false in case of a messy expansion during this test, and in LuaMetaTeX that is default. This means that tests can be made quite robust as it is pretty safe to assume that names that make sense are constructed from regular characters and not boxes, font switches, etc.

3.3 \iffontchar

This test was also part of the ε-TeX extensions and it can be used to see if a font has a character.

```
\iffontchar\font`A
    {\em This font has an A!}
\fi
```

And, as expected, the outcome is: “*This font has an A!*”. The test takes two arguments, the first being a font identifier and the second a character number, so the next checks are all valid:

```
\iffontchar\font     `A yes\else nop\fi\par
\iffontchar\nullfont `A yes\else nop\fi\par
\iffontchar\textfont0`A yes\else nop\fi\par
```

In the perspective of LuaMetaTeX I considered also supporting `\fontid` but it got a bit messy due to the fact that this primitive expands in a different way so this extension was rejected.

3.4 \unless

You can negate the results of a test by using the `\unless` prefix, so for instance you can replace:

```
\ifdim\scratchdimen=10pt
    \dosomething
\else\ifdim\scratchdimen<10pt
    \dosomething
\fi\fi
```

by:

```
\unless\ifdim\scratchdimen>10pt
    \dosomething
\fi
```

4 LuaTEX primitives

4.1 \ifincsname

As it had no real practical usage uit might get dropped in LuaMetaTEX, so it will not be discussed here.

4.2 \ifprimitive

As it had no real practical usage due to limitations, this one is not available in LuaMeta-TEX so it will not be discussed here.

4.3 \ifabsnum

This test is inherited from pdfTEX and behaves like `\ifnum` but first turns a negative number into a positive one.

4.4 \ifabsdim

This test is inherited from pdfTEX and behaves like `\ifdim` but first turns a negative dimension into a positive one.

4.5 \ifcondition

This is not really a test but in order to unstand that you need to know how TEX internally deals with tests.

```
\ifdimen\scratchdimen>10pt
    \ifdim\scratchdimen<20pt
        result a
    \else
        result b
    \fi
\else
    result c
\fi
```

When we end up in the branch of “result a” we need to skip two `\else` branches after we're done. The `\if..` commands increment a level while the `\fi` decrements a level. The `\else` needs to be skipped here. In other cases the true branch needs to be skipped till we end up a the right `\else`. When doing this skipping, TEX is not interested in what it encounters beyond these tokens and this skipping (therefore) goes real fast but it does see nested conditions and doesn't interpret grouping related tokens.

A side effect of this is that the next is not working as expected:

```
\def\ifmorethan{\ifdim\scratchdimen>}
\def\iflessthan{\ifdim\scratchdimen<}

\ifmorethan10pt
    \iflessthan20pt
        result a
    \else
        result b
    \fi
\else
    result c
\fi
```

The `\iflessthan` macro is not seen as an `\if...` so the nesting gets messed up. The solution is to fool the scanner in thinking that it is. Say we have:

```
\scratchdimen=25pt
```

```
\def\ifmorethan{\ifdim\scratchdimen>}
```

```
\def\iflessthan{\ifdim\scratchdimen<}
```

and:

```
\ifcondition\ifmorethan10pt
    \ifcondition\iflessthan20pt
        result a
    \else
        result b
    \fi
\else
    result c
\fi
```

When we expand this snippet we get: “result b” and no error concerning a failure in locating the right `\fi's`. So, when scanning the `\ifcondition` is seen as a valid `\if...` but when the condition is really expanded it gets ignored and the `\ifmorethan` has better come up with a match or not.

In this perspective it is also worth mentioning that nesting problems can be avoided this way:

```
\def\WhenTrue {something \iftrue  ...}
\def\WhenFalse{something \iffalse ...}

\ifnum\scratchcounter>123
    \let\next\WhenTrue
\else
    \let\next\WhenFalse
\fi
\next
```

This trick is mentioned in The TEXbook and can also be found in the plain TEX format. A variant is this:

```
\ifnum\scratchcounter>123
    \expandafter\WhenTrue
\else
    \expandafter\WhenFalse
\fi
```

but using `\expandafter` can be quite intimidating especially when there are multiple in a row. It can also be confusing. Take this: an `\ifcondition` expects the code that follows to produce a test. So:

```
\def\ifwhatever#1%
  {\ifdim#1>10pt
      \expandafter\iftrue
   \else
      \expandafter\iffalse
   \fi}

\ifcondition\ifwhatever{10pt}
    result a
\else
    result b
\fi
```

This will not work! The reason is in the already mentioned fact that when we end up in the greater than `10pt` case, the scanner will happily push the `\iftrue` after the `\fi`, which is okay, but when skipping over the `\else` it sees a nested condition without matching `\fi`, which makes ity fail. I will spare you a solution with lots of nasty tricks, so here is the clean solution using `\ifcondition`:

```
\def\truecondition {\iftrue}
\def\falsecondition{\iffalse}

\def\ifwhatever#1%
  {\ifdim#1>10pt
      \expandafter\truecondition
   \else
      \expandafter\falsecondition
   \fi}

\ifcondition\ifwhatever{10pt}
    result a
\else
    result b
\fi
```

It will be no surprise that the two macros at the top are predefined in ConTEXt. It might be more of a surprise that at the time of this writing the usage in ConTEXt of this `\ifcondition` primitive is rather minimal. But that might change.

As a further teaser I'll show another simple one,

```
\def\HowOdd#1{\unless\ifnum\numexpr ((#1):2)*2\relax=\numexpr#1\relax}
```

```
\ifcondition\HowOdd{1}very \else not so \fi odd
\ifcondition\HowOdd{2}very \else not so \fi odd
\ifcondition\HowOdd{3}very \else not so \fi odd
```

This renders:

very odd
not so odd
very odd

The code demonstrates several tricks. First of all we use `\numexpr` which permits more complex arguments, like:

```
\ifcondition\HowOdd{4+1}very \else not so \fi odd
\ifcondition\HowOdd{2\scratchcounter+9}very \else not so \fi odd
```

Another trick is that we use an integer division (the `:`) which is an operator supported by LuaMetaTEX.

5 LuaMetaTEX primitives

5.1 `\ifcmpnum`

This one is part of s set of three tests that all are a variant of a `\ifcase` test. A simple example of the first test is this:

```
\ifcmpnum 123 345 less \or equal \else more \fi
```

The test scans for two numbers, which of course can be registers or expressions, and sets the case value to 0, 1 or 2, which means that you then use the normal `\or` and `\else` primitives for follow up on the test.

5.2 `\ifchknum`

This test scans a number and when it's okay sets the case value to 1, and otherwise to 2. So you can do the next:

```
\ifchknum 123\or good \else bad \fi
\ifchknum bad\or good \else bad \fi
```

An error message is suppressed and the first `\or` can be seen as a sort of recovery token, although in fact we just use the fast scanner mode that comes with the `\ifcase`: because the result is 1 or 2, we never see invalid tokens.

5.3 \ifnumval

A sort of combination of the previous two is `\ifnumval` which checks a number but also if it's less, equal or more than zero:

```
\ifnumval 123\or less \or equal \or more \else error \fi
\ifnumval bad\or less \or equal \or more \else error \fi
```

You can decide to ignore the bad number or do something that makes more sense. Often the to be checked value will be the content of a macro or an argument like `#1`.

5.4 \ifcmpdim

This test is like `\ifcmpnum` but for dimensions.

5.5 \ifchkdim

This test is like `\ifchknum` but for dimensions. The last checked value is available as `\lastchknum`.

5.6 \ifdimval

This test is like `\ifnumval` but for dimensions. The last checked value is available as `\lastchkdim`

5.7 \iftok

Although this test is still experimental it can be used. What happens is that two to be compared ‘things’ get scanned for. For each we first gobble spaces and `\relax` tokens. Then we can have several cases:

1. When we see a left brace, a list of tokens is scanned upto the matching right brace.
2. When a reference to a token register is seen, that register is taken as value.
3. When a reference to an internal token register is seen, that register is taken as value.
4. When a macro is seen, its definition becomes the to be compared value.
5. When a number is seen, the value of the corresponding register is taken

An example of the first case is:

```
\iftok {abc} {def}%
  ...
```

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

The second case goes like this:

```
\iftok\scratchtoksone\scratchtokstwo
  ...
\else
  ...
\fi
```

Case one and four mixed:

```
\iftok{123}\TempX
  ...
\else
  ...
\fi
```

The last case is more a catch: it will issue an error when no number is given. Eventually that might become a bit more clever (depending on our needs.)

5.8 \ifcstok

There is a subtle difference between this one and `iftok`: spaces and `\relax` tokens are skipped but nothing gets expanded. So, when we arrive at the to be compared ‘things’ we look at what is there, as-is.

5.9 \iffrozen

This is an experimental test. Commands can be defined with the `\frozen` prefix and this test can be used to check if that has been the case.

5.10 \ifprotected

Commands can be defined with the `\protected` prefix (or in ConTEXt, for historic reasons, with `\unexpanded`) and this test can be used to check if that has been the case.

5.11 \ifusercmd

This is an experimental test. It can be used to see if the command is defined at the user level or is a build in one. This one might evolve.

5.12 \ifarguments

This conditional can be used to check how many arguments were matched. It only makes sense when used with macros defined with the `\tolerant` prefix and/or when the sentinel `\ignorearguments` after the arguments is used. More details can be found in the lowlevel macros manual.

5.13 \orelse

This it not really a test primitive but it does act that way. Say that we have this:

```
\ifdim\scratchdimen>10pt
    case 1
\else\ifdim\scratchdimen<20pt
    case 2
\else\ifcount\scratchcounter>10
    case 3
\else\ifcount\scratchcounter<20
    case 4
\fi\fi\fi\fi
```

A bit nicer looks this:

```
\ifdim\scratchdimen>10pt
    case 1
\orelse\ifdim\scratchdimen<20pt
    case 2
\orelse\ifcount\scratchcounter>10
    case 3
\orelse\ifcount\scratchcounter<20
    case 4
\fi
```

We stay at the same level. Sometimes a more flat test tree had advantages but if you think that it gives better performance then you will be disappointed. The fact that we stay at the same level is compensated by a bit more parsing, so unless you have millions such cases (or expansions) it might make a bit of a difference. As mentioned, I'm a bit sensitive for how code looks so that was the main motivation for introducing it. There is a companion `\orunless` continuation primitive.

A rather neat trick is the definition of `\quitcondition`:

```
\def\quitcondition{\orelse\iffalse}
```

This permits:

```
\ifdim\scratchdimen>10pt
    case 1a
    \quitcondition
    case 4b
\fi
```

where, of course, the quitting normally is the result of some intermediate extra test. But let me play safe here: beware of side effects.

6 For the brave

6.1 Full expansion

If you don't understand the following code, don't worry. There is seldom much reason to go this complex but obscure TEX code attracts some users so ...

When you have a macro that has for instance assignments, and when you expand that macro inside an `\edef`, these assignments are not actually expanded but tokenized. In LuaMetaTEX there is a way to apply these assignments without side effects and that feature can be used to write a fully expandable user test. For instance:

```
\def\truecondition {\iftrue}
\def\falsecondition{\iffalse}

\def\fontwithidhaschar#1#2%
  {\beginlocalcontrol
   \scratchcounter\numexpr\fontid\font\relax
   \setfontid\numexpr#1\relax
   \endlocalcontrol
   \iffontchar\font\numexpr#2\relax
      \beginlocalcontrol
      \setfontid\scratchcounter
      \endlocalcontrol
      \expandafter\truecondition
   \else
      \expandafter\falsecondition
   \fi}
```

The `\iffontchar` test doesn't handle numeric font id, simply because at the time it was added to ε-TEX, there was no access to these id's. Now we can do:

```
\edef\foo{\fontwithidhaschar{1} {75}yes\else nop\fi} \meaning\foo
\edef\foo{\fontwithidhaschar{1}{999}yes\else nop\fi} \meaning\foo

[\ifcondition\fontwithidhaschar{1} {75}yes\else nop\fi]
[\ifcondition\fontwithidhaschar{1}{999}yes\else nop\fi]
```

These result in:

macro:yes
macro:nop

[yes]
[nop]

If you remove the `\immediateassignment` in the definition above then the typeset results are still the same but the meanings of `\foo` look different: they contain the assignments and the test for the character is actually done when constructing the content of the `\edef`, but for the current font. So, basically that test is now useless.

6.2 User defined if's

There is a `\newif` macro that defines three other macros:

```
\newif\ifOnMyOwnTerms
```

After this, not only `\ifOnMyOwnTerms` is defined, but also:

```
\OnMyOwnTermstrue
\OnMyOwnTermsfalse
```

These two actually are macros that redefine `\ifOnMyOwnTerms` to be either equivalent to `\iftrue` and `\iffalse`. The (often derived from plain TEX) definition of `\newif` is a bit if a challenge as it has to deal with removing the `if` in order to create the two extra macros and also make sure that it doesn't get mixed up in a catcode jungle.

In ConTEXt we have a variant:

```
\newconditional\MyConditional
```

that can be used with:

```
\settrue\MyConditional
\setfalse\MyConditional
```

and tested like:

```
\ifconditional\MyConditional
    ...
\else
    ...
\fi
```

This one is cheaper on the hash and doesn't need the two extra macros per test. The price is the use of `\ifconditional`, which is *not* to confused with `\ifcondition` (it has bitten me already a few times).

7 Relaxing

When TEX scans for a number or dimension it has to check tokens one by one. On the case of a number, the scanning stops when there is no digit, in the case of a dimension the unit determine the end of scanning. In the case of a number, when a token is not a digit that token gets pushed back. When digits are scanned a trailing space or `\relax` is pushed back. Instead of a number of dimension made from digits, periods and units, the scanner also accepts registers, both the direct accessors like `\count` and `\dimen` and those represented by one token. Take these definitions:

```
\newdimen\MyDimenA \MyDimenA=1pt  \dimen0=\MyDimenA
\newdimen\MyDimenB \MyDimenB=2pt  \dimen2=\MyDimenB
```

I will use these to illustrate the side effects of scanning. Watch the spaces in the result.

First I show what effect we want to avoid. When second argument contains a number (digits) the zero will become part of it so we actually check `\dimen00` here.

```
\def\whatever#1#2%
  {\ifdim#1=#20\else1\fi}
```

```
\whatever{1pt}{2pt}            [macro:1]
\whatever{1pt}{1pt}            [macro:0]
\whatever{\dimen 0}{\dimen 2}  [macro:1]
\whatever{\dimen 0}{\dimen 0}  [macro:]
\whatever\MyDimenA\MyDimenB    [macro:1]
\whatever\MyDimenA\MyDimenB    [macro:1]
```

The solution is to add a space but watch how that one can end up in the result:

```
\def\whatever#1#2%
  {\ifdim#1=#2 0\else1\fi}
```

```
\whatever{1pt}{2pt}            [macro:1]
\whatever{1pt}{1pt}            [macro:0]
\whatever{\dimen 0}{\dimen 2}  [macro:1]
\whatever{\dimen 0}{\dimen 0}  [macro:0]
\whatever\MyDimenA\MyDimenB    [macro:1]
\whatever\MyDimenA\MyDimenB    [macro:1]
```

A variant is using `\relax` and this time we get this token retained in the output.

```
\def\whatever#1#2%
  {\ifdim#1=#2\relax0\else1\fi}
```

```
\whatever{1pt}{2pt}            [macro:1]
\whatever{1pt}{1pt}            [macro:\relax 0]
\whatever{\dimen 0}{\dimen 2}  [macro:1]
\whatever{\dimen 0}{\dimen 0}  [macro:\relax 0]
\whatever\MyDimenA\MyDimenB    [macro:1]
\whatever\MyDimenA\MyDimenB    [macro:1]
```

A solution that doesn't have side effects of forcing the end of a number (using a space or `\relax` is one where we use expressions. The added overhead of scanning expressions is taken for granted because the effect is what we like:

```
\def\whatever#1#2%
  {\ifdim\dimexpr#1\relax=\dimexpr#2\relax0\else1\fi}
```

```
\whatever{1pt}{2pt}            [macro:1]
\whatever{1pt}{1pt}            [macro:0]
\whatever{\dimen 0}{\dimen 2}  [macro:1]
\whatever{\dimen 0}{\dimen 0}  [macro:0]
\whatever\MyDimenA\MyDimenB    [macro:1]
\whatever\MyDimenA\MyDimenB    [macro:1]
```

Just for completeness we show a more obscure trick: this one hides assignments to temporary variables. Although performance is okay, it is the least efficient one so far.

```
\def\whatever#1#2%
  {\beginlocalcontrol
```

```
   \MyDimenA#1\relax
   \MyDimenB#2\relax
   \endlocalcontrol
   \ifdim\MyDimenA=\MyDimenB0\else1\fi}
```

```
\whatever{1pt}{2pt}            [macro:1]
\whatever{1pt}{1pt}            [macro:0]
\whatever{\dimen 0}{\dimen 2}  [macro:1]
\whatever{\dimen 0}{\dimen 0}  [macro:0]
\whatever\MyDimenA\MyDimenB    [macro:1]
\whatever\MyDimenA\MyDimenB    [macro:1]
```

It is kind of a game to come up with alternatives but for sure those involve dirty tricks and more tokens (and runtime). The next can be considered a dirty trick too: we use a special variant of `\relax`. When a number is scanned it acts as relax, but otherwise it just is ignored and disappears.

```
\def\whatever#1#2%
  {\ifdim#1=#2\norelax0\else1\fi}
```

```
\whatever{1pt}{2pt}            [macro:1]
\whatever{1pt}{1pt}            [macro:0]
\whatever{\dimen 0}{\dimen 2}  [macro:1]
\whatever{\dimen 0}{\dimen 0}  [macro:0]
\whatever\MyDimenA\MyDimenB    [macro:1]
\whatever\MyDimenA\MyDimenB    [macro:1]
```

7 Colofon

Author	Hans Hagen
ConTEXt	2023.04.27 17:04
LuaMetaTEX	2.1008
Support	www.pragma-ade.com
	contextgarden.net