low level TeX conditionals
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 Lua\TeX{}, and some only in LuaMeta\TeX{}. 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 \texttt{\ifnum}, \texttt{\ifdim} and \texttt{\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 LuaMeta\TeX{} 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:

\begin{verbatim}
1234
1234pt
\end{verbatim}

If you also use MetaPost, you need to be aware of the fact that in that language there are not really dimensions. The \texttt{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 \texttt{1234pt} you actually multiply 1234 by the variable \texttt{pt}. In \TeX{} on the other hand, a unit like \texttt{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 \texttt{sp}.

\begin{verbatim}
\the\dimexpr 12.34pt \relax
\the\dimexpr 12.34sp \relax
\the\dimexpr 12.99sp \relax
\the\dimexpr 1234sp \relax
\the\numexpr 1234 \relax
\end{verbatim}

\begin{verbatim}
12.34pt
0.00018pt
0.00018pt
0.01883pt
1234
\end{verbatim}

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

\begin{itemize}
  \item \texttt{\number +123}
  \item \texttt{\number +++123}
  \item \texttt{\number + + + 123}
  \item \texttt{\number +++++123}
  \item \texttt{\number --123}
  \item \texttt{\number ---123}
\end{itemize}

-123
-291
-83

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

\begin{itemize}
  \item \texttt{\number -123}
  \item \texttt{\number -"123}
  \item \texttt{\number -'123}
\end{itemize}

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:

\begin{itemize}
  \item \texttt{\thedimen . pt \relax}
  \item \texttt{\thedimen 1. pt \relax}
  \item \texttt{\thedimen .1pt \relax}
\end{itemize}
Again we can use hexadecimal and octal numbers but when these are entered, there can be no fractional part.

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

The maximum number in \texttt{\TeX} is 2147483647 so we can do this:

\begin{verbatim}
\scratchcounter2147483647
\end{verbatim}

but not this

\begin{verbatim}
\scratchcounter2147483648
\end{verbatim}

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

\begin{verbatim}
\scratchdimen 1073741823sp
\number\scratchdimen
\the\scratchdimen
\scratchdimen16383.99998pt
\number\scratchdimen
\the\scratchdimen
1073741823
\end{verbatim}
We can also do this:

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

but the next one will fail:

\scratchdimen16383.999999pt

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

\ifdim16383.99999 pt = 16383.99998 pt the same \else different \fi
\ifdim16383.999979 pt = 16383.999980 pt the same \else different \fi
\ifdim16383.999987 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 = \dimexpr16383.99998 pt/2\relax the same \else different \fi
\ifdim\dimexpr16383.99997 pt:2\relax = \dimexpr16383.99998 pt:2\relax the same \else different \fi
\ifdim\dimexpr1.99997 pt/2\relax = \dimexpr1.99998 pt/2\relax

Preamble
the same \else different \fi
\ifdim \dimexpr 1.99997 pt:2\relax = \dimexpr 1.99998 pt:2\relax
the same \else different \fi
different
the same
\ifdim \dimexpr 1.999999 pt/2\relax = \dimexpr 1.9999995 pt/2\relax
the same \else different \fi
\ifdim \dimexpr 1.999999 pt:2\relax = \dimexpr 1.9999995 pt:2\relax
the same \else different \fi

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.

\[\if AB \text{yes} \else \text{nop} \fi\]
\[\if AA \text{yes} \else \text{nop} \fi\]
\[\if CD \text{yes} \else \text{nop} \fi\]
\[\if CC \text{yes} \else \text{nop} \fi\]
\[\if \dimen \text{font} \text{yes} \else \text{nop} \fi\]
\[\if \dimen \text{font} \text{yes} \else \text{nop} \fi\]
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 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 Con\TeXt 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 yes\else nop\fi]
[\ifcat\noexpand B yes\else nop\fi]
[\ifcat\noexpand C yes\else nop\fi]
[\ifcat\noexpand D yes\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 \texttt{\textbackslash ifnum}

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

\begin{verbatim}
\texttt{\textbackslash ifnum}\texttt{\textbackslash scratchcounter<0}
  \texttt{\textbackslash less than}
\texttt{\textbackslash else}\texttt{\textbackslash ifnum}\texttt{\textbackslash scratchcounter>0}
  \texttt{\textbackslash more than}
\texttt{\textbackslash else}
  \texttt{\textbackslash equal to}
\texttt{\textbackslash fi}\texttt{\textbackslash zero}
\end{verbatim}

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

\begin{verbatim}
\texttt{\textbackslash ifdim}\texttt{\textbackslash scratchdimen<0pt}
  \texttt{\textbackslash less than}
\texttt{\textbackslash else}\texttt{\textbackslash ifdim}\texttt{\textbackslash scratchdimen>0pt}
  \texttt{\textbackslash more than}
\texttt{\textbackslash else}
  \texttt{\textbackslash equal to}
\texttt{\textbackslash fi}\texttt{\textbackslash zero}
\end{verbatim}

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

3 \texttt{\textbackslash ifdim}

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

\begin{verbatim}
\texttt{\textbackslash ifdim}\texttt{\textbackslash scratchdimen<0pt}
  \texttt{\textbackslash less than}
\texttt{\textbackslash else}\texttt{\textbackslash ifdim}\texttt{\textbackslash scratchdimen>0pt}
  \texttt{\textbackslash more than}
\end{verbatim}
\ else
   equal to
\ fi

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

3.1 \ ifodd

This one can come in handy, although in Con\TeXt 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

3.2 \ 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.

3.3 \ 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

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

\begin{verbatim}
\ifdim
\end{verbatim}

\begin{verbatim}
\hmode
\vmode
\end{verbatim}

3.4 \texttt{\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:

\begin{verbatim}
\def\enforcemath#1{\ifmmode
\else$\fi$
\end{verbatim}

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

3.5 \texttt{\ifinner}

\begin{verbatim}
\def\ShowMode\ifinner inner \fi
\else\ifvmode \ifinner inner \fi
\else\ifmmode \ifinner inner \fi
\else \ifinner inner \fi
\fi\fi\fi
\end{verbatim}

\begin{verbatim}
\ShowMode \ShowMode
\end{verbatim}

\begin{verbatim}
\vbox{\ShowMode}
\hbox{\ShowMode}
\end{verbatim}

$\ShowMode$

$$\ShowMode$$

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.

\begin{verbatim}
\vmode \hmode
inner \vmode
inner \hmode
\end{verbatim}

\ifdim
innermmode

mmode

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.

3.6 \ifvoid

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

\begin{verbatim}
\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
\end{verbatim}

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:

\begin{verbatim}
hlist -> [nothing]
hlist -> 1 -> 2 -> 3 -> [nothing]
\end{verbatim}

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.

\begin{verbatim}
\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
\end{verbatim}

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

\begin{verbatim}
\setbox0\hbox{ {}
\wd0=10pt \ifvoid0 void\else content\fi
\end{verbatim}

### 3.7 `\ifhbox`

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

### 3.8 `\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

\hline
vtop or vbox

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

vbox or vtop

\hline
vtop or vbox

but, once wrapped, both internally are just vlists.

### 3.9 `\ifx`

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

\begin{verbatim}
\ifx a b \else N\fi
\ifx ab \else N\fi
\def\A {a}\def\B{b}\ifx \A\B \else N\fi
\def\A{aa}\def\B{a}\ifx \A\B \else N\fi
\def\A {a}\def\B{a}\ifx \A\B \else N\fi
\end{verbatim}

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

3.10 \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.

3.11 \iftrue

It does what it says: always true.

3.12 \iffalse

It does what it says: always false.

3.13 \ifcase

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

\ifcase<number>
  \or
    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

4 \e-\TeX primitives

4.1 \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:
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.

### 4.2 \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

\texttt{\ExpandAfter\ifx\csname Oh Yes : 2\endcsname\relax}
\texttt{\{\bf relaxed indeed\}}
\texttt{\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:

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

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

In Lua\TeX\ there is an option to return false in case of a messy expansion during this test, and in LuaMeta\TeX\ 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.

4.3 \iffontchar

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

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

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

In the perspective of LuaMeta\TeX\ 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.
4.4 \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

5 LuaTEX primitives

5.1 \ifincsname

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

5.2 \ifprimitive

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

5.3 \ifabsnum

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

5.4 \ifabsdim

This test is inherited from pdfTEX and behaves like \ifdim but first turns a negative dimension into a positive one.
5.5 \texttt{\ifcondition}

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

\ifdimen\scratchdimen>10pt
  \ifdim\scratchdimen<20pt
    \texttt{result a}
  \else
    \texttt{result b}
  \fi
\else
  \texttt{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
    \texttt{result a}
  \else
    \texttt{result b}
  \fi
\else
  \texttt{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>}

LuaTeX primitives
\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:

Lua\TeX primitives
\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 is fall. 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\relax=\numexpr#1\relax}
LuaMetaTEX primitives

```
\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.

6 LuaMetaTEX primitives

6.1 \ifcmpnum

This one is part of a 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.

6.2 \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.
6.3 \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.

6.4 \ifcmpdim

This test is like \ifcmpnum but for dimensions.

6.5 \ifchkdim

This test is like \ifchknum but for dimensions.

6.6 \ifdimval

This test is like \ifnumval but for dimensions.

6.7 \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\
...
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.)

6.8 \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.

6.9 \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.

6.10 \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.

6.11 \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.
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 and the only test that cannot be used this way is \ifcondition but that is no real problem. 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.

A rather neat trick is the definition of \quitcondition:

\def \quitcondition {\orelse \iffalse}

This permits:

\ifdim \scratchdimen >10pt
  case 1a
  \quitcondition
  case 4b
where, of course, the quitting normally is the result of some intermediate extra test. But let me play safe here: beware of side effects.

7 For the brave

7.1 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 \LaTeX{} there is a way to immediately apply these assignments and that feature can be used to write a fully expandable user test. For instance:

\begin{verbatim}
def\truecondition {\iftrue}
def\falsecondition{\iffalse}
\def\fontwithidhaschar#1#2\{% 
  \immediateassignment\scratchcounter\numexpr\fontid\font\relax 
  \immediateassignment\setfontid\numexpr#1\relax 
  \iffontchar\font\numexpr#2\relax 
  \immediateassignment\setfontid\scratchcounter
  \expandafter\truecondition
  \else
  \expandafter\falsecondition
  \fi
\}
\end{verbatim}

The \iffontchar test doesn’t handle numeric font id, simply because at the time it was added to \ɛ-\TeX{}, there was no access to these id’s. Now we can do:

\begin{verbatim}
def\foo{\fontwithidhaschar{1} {75}yes\else nop\fi} \meaning\foo 
def\foo{\fontwithidhaschar{1}{999}yes\else nop\fi} \meaning\foo
\end{verbatim}

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.

7.2 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

For the brave
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).

Colofon

Author    Hans Hagen
\inputenx    2019.09.25 23:57
LuaMeta\inputex    2.0
Support    wwwpragma-ade.com
            contextgarden.net