The xifthen package

Josselin Noirel*
https://github.com/JosselinNoirel/xifthen

5th November 2015

Abstract

This package implements new commands that can be used within the first argument of ifthen's `\ifthenelse` to test whether a string is void or not, if a command is defined or equivalent to another. It includes also the possibility to make use of the complex expressions introduced by the package calc, together with the ability of defining new commands to handle complex tests. This package requires the ε-TeX features.

Contents

What's new

1.1 Now `\cnttest` and `\dimtest` accept `<=` and `>=`.

- I renamed `\terminateswith` in `\endswith`.

1.2 Corrected a bug related to a bad interaction between new tests and ifthen's replacement macro (credits go to MPG & P. Albarède).

1.3 Removed a spurious space (thanks to Ulrike Fisher).

1.4.0 Removed the reliance on the `\etex` package, following an exmail exchange with David Carlisle and Maïeul Rouquette. (I also updated the documentation and pushed the files on GitHub.)

*This document corresponds to version v1.4.0 (2015/11/05) of xifthen.sty.

ifthen's interface

Declaring and setting booleans

You can declare boolean (presumably in the preamble of your document) with

`\newboolean{`⟨boolean⟩`}`

where ⟨boolean⟩ is a name made up of alphanumeric characters. For instance,

```
\newboolean{appendix}
\first{appendix}
```

Then your boolean is ready to be set with

`\setboolean{`⟨boolean⟩`}{`⟨truth value⟩`}`

where ⟨truth value⟩ can be `true` or `false`.

Executing conditional code

The general syntax is inherited of that of the package ifthen:

`\ifthenelse{`⟨test expression⟩`}{`⟨true code⟩`}{`⟨false code⟩`}`

Evaluates the ⟨test expression⟩ and executes ⟨true code⟩ if the test turns out to be true and ⟨false code⟩ otherwise. ifthen provides the tests explained in the next paragraphs.

Value of a boolean You can use the value of a boolean you declared, or the value of a primitive boolean of TEX[1]

`\boolean{`⟨boolean⟩`}`

Tests on integers To test whether an integer is equal to, strictly less than, or strictly greater than, you write the expression straightforwardly.

⟨value$_1$⟩ `=` ⟨value$_2$⟩

⟨value$_1$⟩ `<` ⟨value$_2$⟩

⟨value$_1$⟩ `>` ⟨value$_2$⟩

`\isodd{`⟨number⟩`}`

Tests on lengths There exist similar tests for the lengths, but you need in this case to surround the whole expression with `\lengthtest`.

`\lengthtest{`⟨dimen$_1$⟩ `=` ⟨dimen$_2$⟩`}`

`\lengthtest{`⟨dimen$_1$⟩ `<` ⟨dimen$_2$⟩`}`

`\lengthtest{`⟨dimen$_1$⟩ `>` ⟨dimen$_2$⟩`}`

Tests on commands You can test if a command is undefined[2].

`\isundefined{`⟨command⟩`}`

[1] The primitive booleans include: `mmode` (Are we in math mode?), `hmode` (Are we in horizontal mode?), `vmode` (Are we in vertical mode?), etc.

[2] This test differs from `\@ifundefined` in that it takes a real command—and not a command name—as argument, and also in that command which is let equal to `\relax` is not considered undefined.

Tests on character strings You want to know whether two character strings are equal? Use:

`\equal{`⟨string$_1$⟩`}{`⟨string$_2$⟩`}`

Remark that the two arguments are fully expanded. In other words, it is the result of the expansion of the macros that is compared. This behaviour also entails a moving argument and you should protect fragile command to avoid bizarre errors[3].

Building more elaborated expressions You can build more sophisticated expressing using the `\AND` (conjunction), `\OR` (disjunction), and `\NOT` (negation) operators[4].

⟨expression$_1$⟩ `\AND` ⟨expression$_2$⟩

⟨expression$_1$⟩ `\OR` ⟨expression$_2$⟩

`\NOT` ⟨expression⟩

The evaluation is lazy, meaning that if you write

⟨expression$_1$⟩ `\AND` ⟨expression$_2$⟩

then ⟨expression$_2$⟩ won't be evaluated if ⟨expression$_1$⟩ is true[5].

There is not precedence rules: the argument is read from left to right and `\NOT` applies to the very next test. When the precedence must be changed you can use the parentheses:

`\(`⟨expression⟩`\)`

New tests

After this brief review of ifthen's principles, we introduce the new tests provided by xifthen.

Tests on integers One of the drawback of TEX's tests and of `\ifthen`'s as well, is the impossibility to use calc's syntax in it. The `\numexpr` primitive of ε-TEX somehow allows to overcome this difficulty but it is not well documented and normal users are certainly more familiar with the capabilities offered by calc. The xifthen package allows to use calc-valid expressions via the new test `\cnttest`. The syntax is as follows:

`\cnttest{`⟨counter expression$_1$⟩`}{`⟨comparison⟩`}{`⟨counter expression$_2$⟩`}`

It evaluates the two counter expressions, compares them, and returns the value of the test. The comparison can be one of the following sequences `<`, `>`, `=`, `<=`, or `>=`.

Tests on lengths The similar test has been designed for the lengths and dimensions:

`\dimtest{`⟨dimen expression$_1$⟩`}{`⟨comparison⟩`}{`⟨dimen expression$_2$⟩`}`

It evaluates the two dimension expressions, compares them, and returns the value of the test. The comparison can be one of the following sequences `<`, `>`, `=`, `<=`, or `>=`.

[3] Practically, the fact that the content is expanded, means that if the macro `\bar` is defined as `\baz{o}`, and the command `\baz` is defined as `f#1#1`, then `\equal{\bar}{foo}` turns out to be true, because `\bar` eventually expands into `foo`. This is usually the desired behaviour.

[4] Lowercase versions of these commands also exist but we advise the user to stick to the uppercase ones because `\or` is part of TEX's syntax.

[5] The devil is in the details, however: ifthen works by reading its argument twice. The tests are evaluated on the second pass, but the expansion is performed on the first one, regardless of the truth value.

Tests on commands We define a companion of `\isundefined` that uses a command name rather than a command[6].

`\[1,syntax]isnamedefinedcommand name`

Returns *true* if the command `\`⟨command name⟩ is defined[7].

Sometimes, you need to compare two macros `\foo` and `\bar` and test whether they are actually the same macro.

`\isequivalentto{`⟨command$_1$⟩`}{`⟨command$_2$⟩`}`

Corresponds to the `\ifx` test: it returns *true* when the two commands are exactly equivalent (same definition, same number of arguments, same prefixes, etc., otherwise *false* is returned).

Tests on character strings Very often, we see people using `\[2]equal#1` in their command definitions (for instance, to test whether an optional argument had been passed to their macro). A more efficient test can be used:

`\isempty{`⟨content⟩`}`

Returns *true* if ⟨content⟩ is empty. It is essentially equivalent to `\equal{`⟨content⟩`}{}` except that the argument of `\isempty` isn't expanded and therefore isn't affected by fragile commands[8].

It is possible to test whether a substring appears within another string[9].

`\isin{`⟨substring⟩`}{`⟨string⟩`}`

Sometimes, you need to check whether a string ends with a particular substring. This can be achieved using[10]:

`\endswith{`⟨string⟩`}{`⟨substring⟩`}`

Building more elaborated expressions It is then possible to create new tests with:

`\newtest{`⟨command⟩`}[`n`]{`⟨test expression⟩`}`

Surprisingly, a simple `\newcommand` would not work. The `\newtest` macro defines a command ⟨command⟩ taking n arguments (no optional argument is allowed[11] consisting of the test as specified by ⟨test expression⟩ that can be used in the argument of `\ifthenelse`.

Examples

Let's illustrate the most important features of xifthen with the following problem: if we want to test whether a rectangle having dimensions l and L meets the two following conditions: $S = l \times L > 100$ and $P = 2(l + L) < 60$[12]:

[6] If you are stuck with the distinction between 'command' and 'command name', let me explain it further with an example: the command name of the command `\foo` is `foo`. This is sometimes more convenient to use the command name than the name. Still, this functionality is probably intended more for experienced programmers who want to use the niceties of ifthen and xifthen.

[7] Uses `\ifcsname...\endcsname` internally and not `\@ifundefined`.

[8] Internally, it uses `\unexpanded` and ifmtarg.

[9] Uses `\in@` and `\ifin@` internally.

[10] For compatibility reasons, there exist a command unfortunately called `\terminateswith` that performs the same test but it is deprecated.

[11] No optional argument is allowed because the macro needs to be expanded in the first pass and that optional arguments avoid that.

[12] Note that, because within the arguments of `\cnttest` the calc is used, you must use real parentheses `(` and `)` and not `\(` and `\)`.

```
\newtest{\condition}[2]{%
  \cnttest{(#1)*(#2)}>{100}%
  \AND
  \cnttest{((#1)+(#2))*2}<{60}%
}
```

Then `\ifthenelse{\condition{14}{7}}{TRUE}{FALSE}` returns FALSE because $14 \times 7 = 98$ and $2 \times (14 + 7) = 42$, while `\ifthenelse{\condition{11}{11}}{TRUE}{FALSE}` returns TRUE because $11 \times 11 = 121$ and $2 \times (11 + 11) = 44$.

Now a list of typical uses of xifthen's capabilities:

$4 - 1 < 4$: **true** $4 < 4$: false $4 + 1 < 4$: false
$4 - 1 \leq 4$: **true** $4 \leq 4$: **true** $4 + 1 \leq 4$: false
$4 - 1 = 4$: false $4 = 4$: **true** $4 + 1 = 4$: false
$4 - 1 \geq 4$: false $4 \geq 4$: **true** $4 + 1 \geq 4$: **true**
$4 - 1 > 4$: false $4 > 4$: false $4 + 1 > 4$: **true**

$4\,\text{pt} - 1\,\text{pt} < 4\,\text{pt}$: **true** $4\,\text{pt} < 4\,\text{pt}$: false $4\,\text{pt} + 1\,\text{pt} < 4\,\text{pt}$: false
$4\,\text{pt} - 1\,\text{pt} \leq 4\,\text{pt}$: **true** $4\,\text{pt} \leq 4\,\text{pt}$: **true** $4\,\text{pt} + 1\,\text{pt} \leq 4\,\text{pt}$: false
$4\,\text{pt} - 1\,\text{pt} = 4\,\text{pt}$: false $4\,\text{pt} = 4\,\text{pt}$: **true** $4\,\text{pt} + 1\,\text{pt} = 4\,\text{pt}$: false
$4\,\text{pt} - 1\,\text{pt} \geq 4\,\text{pt}$: false $4\,\text{pt} \geq 4\,\text{pt}$: **true** $4\,\text{pt} + 1\,\text{pt} \geq 4\,\text{pt}$: **true**
$4\,\text{pt} - 1\,\text{pt} > 4\,\text{pt}$: false $4\,\text{pt} > 4\,\text{pt}$: false $4\,\text{pt} + 1\,\text{pt} > 4\,\text{pt}$: **true**

`\ifthenelse{\isempty{}}{true}{false}` **true**
`\ifthenelse{\isempty{ }}{true}{false}` **true**
`\ifthenelse{\isempty{ foo }}{true}{false}` false

`\ifthenelse{\endswith{foo.}{.}}{true}{false}` **true**
`\ifthenelse{\endswith{foo!}{.}}{true}{false}` false

`\ifthenelse{\isin{foo}{foobar}}{true}{false}` **true**
`\ifthenelse{\isin{Foo}{foobar}}{true}{false}` false

`\ifthenelse{\cnttest{10 * 10 + 1}>{100}}{true}{false}` **true**
`\ifthenelse{\cnttest{10 * 10 + 1}>{100 * 100}}{true}{false}` false

`\ifthenelse{\isequivalentto{\usepackage}{\RequirePackage}}{true}{false}` **true**
`\ifthenelse{\isequivalentto{\usepackage}{\textit}}{true}{false}` false

`\ifthenelse{\isnamedefined{@foo}}{true}{false}` false
`\ifthenelse{\isnamedefined{@for}}{true}{false}` **true**