

The `xifthen` package

Josselin Noirel*

<http://www.jnoirel.fr/>

17th April 2009

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
ifthen's interface	1
Declaring and setting booleans	1
Executing conditional code	2
New tests	3
Examples	4

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).

ifthen's interface

Declaring and setting booleans

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

```
\newboolean{boolean}
```

*This document corresponds to version v1.3 (2009/04/17) of `xifthen.sty`.

where $\langle boolean \rangle$ is a name made up of alphanumeric characters. For instance,

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

Then your boolean is ready to be set with

```
\setboolean{\langle boolean \rangle}{\langle truth value \rangle}
```

where $\langle truth value \rangle$ can be `true` or `false`.

Executing conditional code

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

```
\ifthenelse{\langle test expression \rangle}{\langle true code \rangle}{\langle false code \rangle}
```

Evaluates the $\langle test expression \rangle$ and executes $\langle true code \rangle$ if the test turns out to be true and $\langle false code \rangle$ 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{\langle boolean \rangle}
```

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

```
\langle value1 \rangle = \langle value2 \rangle
\langle value1 \rangle < \langle value2 \rangle
\langle value1 \rangle > \langle value2 \rangle
\isodd{\langle number \rangle}
```

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

```
\lengthtest{\langle dimen1 \rangle = \langle dimen2 \rangle}
\lengthtest{\langle dimen1 \rangle < \langle dimen2 \rangle}
\lengthtest{\langle dimen1 \rangle > \langle dimen2 \rangle}
```

Tests on commands You can test if a command is undefined².

```
\isundefined \langle command \rangle
```

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

```
\equal{\langle string1 \rangle}{\langle string2 \rangle}
```

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

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.

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

Building more elaborated expressions You can build more sophisticated expressions using the `\AND` (conjunction), `\OR` (disjunction), and `\NOT` (negation) operators⁴.

```
⟨expression1⟩ \AND ⟨expression2⟩
⟨expression1⟩ \OR  ⟨expression2⟩
\not ⟨expression⟩
```

The evaluation is lazy, meaning that if you write

```
⟨expression1⟩ \AND ⟨expression2⟩
```

then `⟨expression2⟩` won't be evaluated if `⟨expression1⟩` is true⁵.

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 expression1⟩}{⟨comparison⟩}{⟨counter expression2⟩}
```

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 expression1⟩}{⟨comparison⟩}{⟨dimen expression2⟩}
```

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 `>=`.

Tests on commands We define a companion of `\isundefined` that uses a command name rather than a command⁶.

```
\isnamedefined{⟨command name⟩}
```

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 evil is in the details: `ifthen` works by reading its argument twice. The tests are evaluated on the second pass, but an expansion is performed on the first one, regardless of the truth value. So that, quite surprisingly, the following test fails:

```
\NOT \isundefined{\foo} \AND \equal{\foo}{⟨string⟩}
```

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`.

Returns *true* if the command `\langle command name \rangle` is defined⁷.

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

```
\isequivalentto{\langle command_1 \rangle}{\langle command_2 \rangle}
```

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 `\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{\langle content \rangle}
```

Returns *true* if `\langle content \rangle` is empty. It is essentially equivalent to `\equal{\langle content \rangle}{}` except that the argument of `\isempty` isn't expanded and therefore isn't affected by fragile commands⁸.

It is possible to test whether a substring appears within another string⁹.

```
\isin{\langle substring \rangle}{\langle string \rangle}
```

Sometimes, you need to check whether a string ends with a particular substring. This can be achieved using¹⁰:

```
\endswith{\langle string \rangle}{\langle substring \rangle}
```

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

```
\newtest{\langle command \rangle}[\langle n \rangle]{\langle test expression \rangle}
```

Surprisingly, a simple `\newcommand` would not work. The `\newtest` macro defines a command `\langle command \rangle` taking `n` arguments (no optional argument is allowed¹¹ consisting of the test as specified by `\langle test expression \rangle` 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:

```
\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 $2 \times (11 + 11) = 44$.

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 \).

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

```
4 - 1 < 4: true  4 < 4: false  4 + 1 < 4: false
4 - 1 ≤ 4: true  4 ≤ 4: true  4 + 1 ≤ 4: false
4 - 1 = 4: false  4 = 4: true  4 + 1 = 4: false
4 - 1 ≥ 4: false  4 ≥ 4: true  4 + 1 ≥ 4: true
4 - 1 > 4: false  4 > 4: false  4 + 1 > 4: true

4pt - 1pt < 4pt: true  4pt < 4pt: false  4pt + 1pt < 4pt: false
4pt - 1pt ≤ 4pt: true  4pt ≤ 4pt: true  4pt + 1pt ≤ 4pt: false
4pt - 1pt = 4pt: false  4pt = 4pt: true  4pt + 1pt = 4pt: false
4pt - 1pt ≥ 4pt: false  4pt ≥ 4pt: true  4pt + 1pt ≥ 4pt: true
4pt - 1pt > 4pt: false  4pt > 4pt: false  4pt + 1pt > 4pt: 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{\cntttest{10 * 10 + 1}>{100}}{true}{false} true
\ifthenelse{\cntttest{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
```