

# The ifthen package\*

David Carlisle

2001/05/26

## Abstract

This file implements an `\ifthenelse` command for L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>. The algorithm used is compatible with that used in the L<sup>A</sup>T<sub>E</sub>X 2.09 `ifthen` style option. It has been recoded, making the resulting definitions somewhat more compact and efficient.

## 1 Introduction

`\ifthenelse` `\ifthenelse{<test>}{<then clause>}{<else clause>}`

Evaluates `<test>` as a boolean function, and then executes either `<then clause>` or `<else clause>`.

`<test>` is a boolean expression using the infix connectives, `\and`, `\or`, the unary `\not` and parentheses `\( \)`.

As an alternative notation `\AND`, `\OR` and `\NOT` can be used. This is safer since it can't be misinterpreted when appearing inside a T<sub>E</sub>X-conditional in which `\or` has a different meaning.

The atomic propositions are:

`<number> < <number>`  
`<number> = <number>`  
`<number> > <number>`  
`\isodd{ <number> }`  
`\isundefined{ <command name> }`  
`\equal{<string>}{<string>}`  
`\lengthtest{<dimen>< <dimen>}`  
`\lengthtest{<dimen>= <dimen>}`  
`\lengthtest{<dimen>> <dimen>}`  
`\boolean{<name>}`

The `<string>`s tested by `\equal` may be any sequence of commands that expand to a list of tokens. If these expansions are equal, then the proposition is true.

`\isodd` is true if the `<number>` is odd, and false otherwise (even if the argument is not a number).

`\isundefined{<cmd>}` is true if `<cmd>` is not defined.

`\boolean{xyz}` returns the truth value contained in the primitive T<sub>E</sub>X `\if`, `\ifxyz`. This is usually used with boolean flags created with `\newboolean` and `\provideboolean` described below. It can also be used with the names of `\newif` created tokens, and primitive T<sub>E</sub>X `\if` constructs, for example `\boolean{true}` (`\iftrue`), `\boolean{mmode}` (`\ifmmode`) etc.

The commands:

`\newboolean` `\newboolean{<name>}` and `\provideboolean{<name>}` are provided so the user  
`\provideboolean` can easily create new boolean flags. As for `\newcommand`, `\newboolean` generates an error if the command name is not new. `\provideboolean` silently does nothing in that case.

The boolean flags may be set with:

`\setboolean` `\setboolean{<name>}{<value>}`

---

\*This file has version number v1.1c, last revised 2001/05/26.

$\langle value \rangle$  may be either `true` or `false` (any CaSe).

Note that there is no precedence between `\and` and `\or`. The proposition is evaluated in a left right manner. `\not` only applies to the immediately following proposition. (This is consistent with Lamport’s `ifthen.sty`.) In this style, though the test is ‘lazily’ evaluated, so for instance if the first proposition in an `\or` is true, the second one is skipped. (On the second pass—the first pass in an `\edef` expands clauses in all propositions.)

Apart from the addition of the extra atomic propositions `\isodd`, `\boolean`, `\lengthtest` and `\isundefined`, the only known incompatibility is that in this package the expression `\not\not\langle P \rangle` is equivalent to  $\langle P \rangle$ . However in the original style it was equivalent to `\not\langle P \rangle`. This is intentional (bug fix:-).

`\whiledo` The command `\whiledo` is also defined (copied directly from the L<sup>A</sup>T<sub>E</sub>X 2.09 definition).

`\whiledo{<test>}{<while clause>}`

With  $\langle test \rangle$  as above, repeatedly executes  $\langle while\ clause \rangle$  while the test remains true.

## 2 The Implementation

1 `\*package`

`\TE@throw` In order to support the syntax of `ifthen.sty`, which allows access to the primitive T<sub>E</sub>X syntax for a numeric test, rather than a `{}` delimited argument form, it is most convenient to work ‘within’ an `\ifnum`. `\ift@throw` ‘throws’ you out of the current `\ifnum` so that you can (eg) start an `\ifdim` for the length tests.

2 `\def\TE@throw{\@ne=\@ne\noexpand\fi}`

`\boolean` A non-standard extension to `ifthen`, supporting boolean was previously available, this is a simpler implementation.

3 `\def\boolean#1#2{%`

4 `\TE@throw\expandafter\noexpand\csname if#1\endcsname#2}`

`\TE@length` Testing lengths. `#1` is the test. The extra argument gobbles spaces.

5 `\def\TE@length#1#2{\TE@throw\noexpand\ifdim#1#2}`

`\TE@odd` Testing odd/even. This is true if `#1` is an odd number, and false otherwise (even if `#1` is not a number at all).

It is hard to make this completely reliable. Here I have erred on the side of safety. This should not generate a T<sub>E</sub>X error if given any robust commands as its argument. However it returns true on any argument that *starts* with an odd number 11xx which is bad, and it can not deal with T<sub>E</sub>X’s count registers, although L<sup>A</sup>T<sub>E</sub>X counters work (via `\value`).

6 `\def\TE@odd#1#2{%`

7 `\TE@throw\noexpand\TE@odd#1\noexpand\@nil\noexpand\ifodd\count@#2}`

`\TE@odd` is not expanded on the first pass.

8 `\def\TE@odd#1#2\@nil{%`

9 `\@defaultunits`

10 `\count@\if-#1-0\else0\expandafter#1\fi#2\relax\@nnil}`

`\TE@repl` `\TE@repl` replaces the single token `#1` by `#2`. (Not within `{}` groups.) It is used to replace `\or` by `\TE@or` without the need to redefine `\or`. Earlier versions just `\let\or\TE@or` but this has a bad effect on the expansion of commands which use the primitive `\or` internally, eg `\alph`, and so caused surprising results if these commands were used inside `\equal`.

11 `\def\TE@repl#1#2{%`

12 `\long\def\@tempc##1#1#2{%`

13 `\def\@tempa{##2}\def\@tempb{\@tempc}%`

14 `\ifx\@tempa\@tempb`

```

15     \toks@\expandafter{\the\toks@##1}%
16     \expandafter\@gobble
17   \else
18     \toks@\expandafter{\the\toks@##1#2}%
19     \expandafter\@tempc
20     \fi
21   ##2}%
22   \expandafter\toks@\expandafter{\expandafter}%
23   \expandafter\@tempc\the\toks@#1\@tempc}

```

`\ifthenelse` The remaining macros in this file are derived from the ones in `ifthen.sty` but recoded and simplified. The main simplification is that the original style (and the `\boolean` extensions) expressed logical values always in terms of `\ifnum`. As `\fi` is ‘untyped’ this is not necessary, so for example the length tests can return values via `\ifdim`, the trailing `\fi` will not complain, even though it was ‘expecting’ an `\ifnum`. Also the system of passing information via macros expanding to T or F has been completely replaced by a simpler system using `\iftrue`, which furthermore allows lazy evaluation on the second pass.

```

24 \long\def\ifthenelse#1{%
25   \toks@{#1}%
26   \TE@repl\or\TE@or
27   \TE@repl\and\TE@and
28   \TE@repl\not\TE@neg

```

Support alternate names for the boolean operators (strictly speaking only `\OR` would be necessary).

```

29   \TE@repl\OR\TE@or
30   \TE@repl\AND\TE@and
31   \TE@repl\NOT\TE@neg

```

The original `ifthen.sty` processed everything inside a box assignment, to catch any extra spaces before they appeared in the output. Instead I have added extra arguments to the commands so they each remove any following space.

Set up the user level names `\not` etc.

```

32   \begingroup
33     \let\protect\@unexpandable@protect
34     \def\@setref##1##2##3{%
35       \ifx##1\relax\z@\else\expandafter##2##1\fi}%
36     \def\value##1{\the\csname c@##1\endcsname}%
37     \let\equal\TE@equal \let\(\TE@lparen \let\)\TE@rparen
38     \let\isodd\TE@odd \let\lengthtest\TE@length
39     \let\isundefined\TE@undef

```

For the first pass, in a group, make various tokens non-expandable.

It is unfortunate that in order to remain compatible with `ifthen` syntax, it is necessary to have a two pass system. The first pass inside an `\edef` ‘exposes’ the `\if... \fi` tokens, so the correct clauses may be skipped on the second pass. This means that the whole `\ifthenelse` command does not work by expansion, and so possibly has only limited usefulness for macro code writers. The main problem with the `ifthen` syntax is that (unique for L<sup>A</sup>T<sub>E</sub>X) it does not use a brace delimited argument form, and exposes the primitive T<sub>E</sub>X syntax for  $\langle number \rangle$ . Pretty much the only way of parsing `1 > 2 \or 2 < 1` is to actually evaluate the primitive `\ifnum`s. A syntax such as:

```
\or{\numtest{1<2}}{\lengthtest{1pt<1in}}
```

could easily be evaluated in a one pass way, operating directly via expansion, and leaving no extra tokens in the token stream.

Still, on with the code... make `\@tempa` and `\@tempb` tokens non-expandable on the first pass.

```

40   \begingroup
41     \let\@tempa\relax\let\@tempb\relax
42     \xdef\@gtempa{\expandafter\TE@eval\the\toks@\TE@endeval}%

```

```

43         \endgroup

```

Now outside the group, execute `\@gtempa` which causes all the `\ifs` etc., to be evaluated, the final truth value is contained in the `\newif` token `\ifTE@val`. Finally this is tested and either the first or second following argument is chosen accordingly.

```

44         \@gtempa
45         \expandafter\endgroup\ifTE@val
46         \expandafter\@firstoftwo
47         \else
48         \expandafter\@secondoftwo
49         \fi}

```

`\TE@eval` Initialise a term. (Expanded on the first pass).

```

50 \def\TE@eval{\noexpand\TE@negatefalse\noexpand\iftrue\noexpand\ifnum}

```

`\ifTE@val` Two `\newifs` the first holds the current truth value of the expression. The second is a temporary flag which is true if we need to negate the current proposition.

```

51 \newif\ifTE@val
52 \newif\ifTE@negate

```

`\TE@endeval` Finalise a term. (Expanded on the first pass).

```

53 \def\TE@endeval{\relax
54     \noexpand\TE@setvaltrue\noexpand
55     \else
56     \noexpand\TE@setvalfalse\noexpand
57     \fi
58     \noexpand\TE@negatefalse\noexpand
59     \fi}

```

`\TE@setvaltrue` Set the `\ifTE@val` to true or false depending on the value of the current proposition, and the negate flag. (Not expanded on the first pass.)

```

60 \def\TE@setvaltrue{%
61     \ifTE@negate\TE@valfalse\else\TE@valtrue\fi}
62 \def\TE@setvalfalse{\let\ifTE@val\ifTE@negate}

```

`\TE@or` The internal version of `\or`. Ends the current term. If true skip the remaining terms.

```

63 \def\TE@or{\TE@endeval\noexpand\ifTE@val\noexpand\else\noexpand\ifnum}

```

`\TE@and` The internal version of `\and`. If false skip the remaining terms.

```

64 \def\TE@and{\TE@endeval\noexpand\ifTE@val\noexpand\ifnum}

```

`\TE@neg` `\not`. Throw the current context, set a negate flag, then restart the `\ifnum`.

`\TE@negswitch` `\TE@negswitch` is not expanded on the first pass.

```

65 \def\TE@neg{\TE@throw\noexpand\TE@negswitch\noexpand\ifnum}
66 \def\TE@negswitch{\ifTE@negate\TE@negatefalse\else\TE@negatetrue\fi}

```

`\TE@lparen` `\(`. Throw the current context, then restart a term inside a group.

```

67 \def\TE@lparen#1{\TE@throw\begin\group\TE@eval#1}

```

`\TE@rparen` `\)` end the current term, and the local group started by `\(`, but pass on the boolean value in `\if@val`. The `\noexpand` stops the `\expandafter` from expanding on the first pass.

```

68 \def\TE@rparen#1{%
69     \TE@endeval
70     \noexpand\expandafter\endgroup\noexpand\ifTE@val#1}

```

`\TE@equal` `\equal` greatly simplified from the original. `\def` may be used rather than `\edef` as the whole thing is expanded anyway in the first pass. The boolean can be directly encoded with the `\ifx`, there is no need to start an equivalent `\ifnum`.

```

71 \long\def\TE@equal#1#2#3{\TE@throw
72   \def\@tempa{#1}\def\@tempb{#2}%
73   \noexpand\ifx\@tempa\@tempb#3}

```

`\setboolean` `\setboolean` takes true or false, as #2, and sets #1 accordingly.

```

74 \def\setboolean#1#2{%
75   \lowercase{\def\@tempa{#2}}%
76   \@ifundefined{@tempa}{\@tempa}%
77   {\PackageError{ifthen}%
78     {You can only set a boolean to ‘true’ or ‘false’}\@ehc}%
79   {\@ifundefined{#1\@tempa}%
80     {\PackageError{ifthen}{Boolean #1 undefined}\@ehc}%
81     {\csname#1\@tempa\endcsname}}}

```

`\newboolean` Define a new ‘boolean’.

```

82 \def\newboolean#1{%
83   \expandafter\@ifdefinable\csname if#1\endcsname{%
84     \expandafter\newif\csname if#1\endcsname}}

```

`\provideboolean` Define a new ‘boolean’ if it is not already defined.

```

85 \def\provideboolean#1{%
86   \@ifundefined{if#1}{%
87     \expandafter\newif\csname if#1\endcsname}\relax}

```

`\whiledo` `\whiledo` copied directly from the original.  
`\whiledo{<test>}{<body>}`  
repeatedly evaluates `<body>` until `<test>` is true.

```

88 \long\def\whiledo#1#2{%
89   \ifthenelse{#1}%
90   {\@whiledotrue
91     \@whiles\if@whiledo\fi
92     {#2}%
93     \ifthenelse{#1}\@whiledotrue\@whiledofalse}}%
94   {}%
95   }

```

`\TE@undef` test if csname is defined. `\ifx` test.

```

96 \def\TE@undef#1#2{%
97   \TE@throw\noexpand\ifx\noexpand\@undefined\noexpand#1#2}

```

`\if@whiledo` Internal switch for `\whiledo`.

```

98 \newif\if@whiledo
99 \</package>

```