Oral TEX: Erratum

TUGboat **12, no. 2, p. 272–276**

Victor Eijkhout

Alert reader Bernd Raichle pointed out that my macro for lexicographic ordering was not correct. Here is a repaired version. Replace the definition of `\ifallchars` at the bottom of page 274, column 1, by the following.

```
\def\ifallchars#1#2\are#3#4\before
   {\if#1$\say{true\xp}\else
     \if#3$\say{false\xp\xp\xp}\else
      \ifnum`#1>`#3 \say{false%
         \xp\xp\xp\xp\xp\xp\xp}\else
       \ifnum`#1<`#3 \say{true%
         \xp\xp\xp\xp\xp\xp\xp
         \xp\xp\xp\xp\xp\xp\xp\xp}\else
       \ifrest#2\before#4\fi\fi\fi\fi}
```

This macro contains the slightly ridiculous sequence of 15 `\expandafter` commands. However, Bernd Raichle also supplied his own solution to string testing and lexicographic ordering, which use a somewhat different principle (and are in addition shorter) than mine.

```
\def\ifsamestring#1#2{\csname
     if\allchars#1$\are#2$\same
     \endcsname}
\def\allchars#1#2\are#3#4\same{%
  \if#1$%
    \if#3$true\else false\fi
  \else
    \if#1#3\allchars#2\are#4\same
    \else false\fi
  \fi
}
\def\ifbefore#1#2{\csname
     if\allchars#1$\are#2$\before
     \endcsname}
\def\allchars#1#2\are#3#4\before{%
  \if#1$%
    true%
  \else \if#3$%
    false%
  \else \ifnum`#1>`#3
    false%
  \else \ifnum`#1<`#3
    true%
  \else
    \allchars#2\are#4\before
  \fi\fi\fi\fi
}
```

Some Basic Control Macros for TEX

Jonathan Fine

Abstract

This article is concerned with the mouth of TEX, particularly macros and the primitives `\if...`, `\else` and `\fi` used to control expansion. (Recall that the mouth expands the input stream until it comes to something unexpandable, which is then passed to the stomach.)

Although it can do little but absorb parameters and expand macros, the mouth is powerful. Alan Jeffrey (*Lists in TEX's Mouth*, *TUGboat* 11, no. 2 pp. 237–245, June 1990) shows that it can do the lambda calculus.

Our purpose is more limited. It is to define and describe macros `\break`, `\continue`, `\switch`, `\return`, `\exit`, `\chain`, and labels `\end` and `':'` that make it easier to write TEX macros. These macros will be collected into a file `control.sty`.

Several worked examples are given.

1 Introduction

There are now some substantial programs written in TEX. The source for LATEX runs to 8,500 lines. PICTEX has 3,500 lines. A style file might have 120 lines of code and 300 lines of comments. TEX is a terse and at times cryptic language. A great deal can be done in 26 short lines. This article is devoted to making life easier for that suffering creature, the writer of TEX macros.

Acknowledgements. The author thanks the referees for their careful comments, which have greatly improved the article.

1.1 Ignoring spaces

TEX has rules for ignoring spaces in the input stream that are well adapted to reading a text file spiced with control sequences. But these rules do not suit the macro writer, whose words are few, and control sequences many. Many programming languages today are 'free form'. White space is ignored, allowing the programmer to indent or otherwise arrange the code, so that the meaning is more easily read.

Accordingly, by changing the category of tab, carriage return, space, and `'~'`

```
\chardef\ignore 9
\catcode 9\ignore
\catcode13\ignore
\catcode32\ignore
\catcode`\~ 10\relax  % make ~ space
```

we ignore all white space except when we explictly ask for it.

1.2 Speaking clearly

All m@nn@r @f d@vic@s @re us@d t@ gener@te priv@te c@ntr@l s@qu@nc@s in m@cro fil@s. I'll say that again. All manner of devices are used to generate private control sequences in macro files. And they are a nuisance. Here, we will take the programmer's side, and use names built out of ordinary letters.

We also give our control sequences the names we *want* to give them. *In particular* `\break` *and* `\end` *do not have the usual meanings.*

1.3 Top-down and step-wise

The author hopes that he has used here the techniques of *top-down programming* and *step-wise refinement* to obtain these basic control macros.

This means we first identify and solve some of the essential features of the problem, and then go to details. We may have to go round several times until we are finished.

1.4 Auxiliary macros

We will need some general purpose helper macros.

```
\def\unbrace #1 { #1 }
\def\gobble #1 { }
\def\gobbletwo  #1 #2 { }
```

2 Macros for loops

Repetition and termination are the essential features of a loop.

2.1 Writing `\myloop`

Our first attempt

```
\def\myloop {
  \myloop
}
```

produces a loop that will, when executed, endlessly do nothing useful. It has the required virtue of repetition, but in excess.

First, we will make the loop do something useful. (Then worry about stopping.) Suppose, for example, that we wish `\myloop abc...` to have

```
\myoperation a
\myoperation b
\myoperation c
  ...
```

as its result.

We add a parameter and an action to the definition of `\myloop`.

```
\def\myloop #1 {
  \myoperation #1
  \myloop
}
```

Now worry about stopping. Suppose that `\myloop` is to stop when some token, for example `\end`, is passed as a parameter. Before executing `\myoperation #1` we must make a test `\ifx#1\end` and if it succeeds, the loop is to be terminated.

Termination will require a strange trick which we will call `\break`. (The programming language *C* uses `break` for a similar purpose.)

We now have

```
\def\myloop #1 {
  \ifx #1 \end \break \fi
  \myoperation #1
  \myloop
}
```

where `\break`, when called, terminates the loop.

2.2 `\breaking` from `\myloop`

We are now able to specify and code the `\break` command. Its definition is at first less than intuitive, although arrived at logically. To prevent repetition, `\break` must absorb the lines

```
\myoperation #1
\myloop
```

of `\myloop`. We can do this by letting `\myloop` delimit the argument to `\break`. The command

```
\def\break #1 \myloop { }
```

will absorb the unwanted tokens. Sadly, it also absorbs the `\fi`, which we must put back again. This is easy! Just put it back.

```
\def\break #1 \myloop { \fi }
```

To summarize, the code for `\myloop` works because

- The expansion of `\myloop` terminates with `\myloop`. (This is tail recursion.)
- The expansion of `\break` successfully breaks the loop.

2.3 Don't do this

Someone coding their first loop might write

```
\def\myloop #1 {
  \ifx #1\end\else
    \myoperation #1
    \myloop
  \fi
}
```

which looks correct but isn't. The reason is subtle. We will expand `\myloop AB`, assuming that `\myoperation` is `\gobble`. Here it is, step-by-step.

1. `\myloop AB`
2. `\ifx A\end \else`
   `\myoperation A\myloop \fi B`
3. `\myoperation A\myloop \fi B`
4. `\myloop \fi B`

and now we are in trouble. `\myloop` is about to eat the `\fi`. It should be getting the B. In fact `\myloop` will continue to generate and consume `\fi` tokens, and will never get to B.

5. `\ifx \fi \end \else`
   `\myoperation \fi \myloop \fi B`
6. `\myoperation \fi \myloop \fi B`
7. `\myloop \fi B`

For `\myloop` to be successful, calling itself must, literally, be the last thing it does. Only then can it read the next token, which is B *in our example.* Computer scientists call this 'tail-recursion'. This trick avoids another hazard, the filling up of memory during a long loop. **5.1** gives an example of how this can arise (see also *The TEXbook*, p. 219).

2.4 Coding Tail Recursion

It is traditional to use an assignment to a scratch control sequence

```
\def\myloop #1 {
  \ifx #1\end
    \let\next\relax
  \else
    \myoperation #1
    \let\next\mymacro
  \fi
  \next
}
```

to achieve tail recursion.

Assignments (and other unexpandable primitives) are not performed within `\edef`, `\xdef`, `\message`, `\errmessage`, `\write`, `\mark`, `\special` and also the `\csname`–`\endcsname` pair. This limits the usefulness of the traditional design.

However, the macros of `control.sty` can safely be used in these situations, and also when TEX is looking for a number, dimension, glue or filename.

2.5 Writing `\yourloop`

Now suppose you wish to use the above to code another loop, `\yourloop`. A problem appears. The definition

```
\def\break #1 \myloop{ \fi }
```

has `\myloop` coded into it, and so is not suitable for coding `\yourloop`. We do not wish each loop to need a different `\break` command. This would be wasteful. We notice that the key to `\break` is that it gobbles to a certain point, and then puts down a balancing `\fi`. Here is a first guess to a universal `\break`.

```
\def\break #1 : { \fi }
```

(*C* uses the colon ':' as a label to allow use of the much-abused `goto` command.)

Given this `\break`, the definition

```
\def\yourloop #1 {
  \ifx #1 \end \break \fi
  \youroperation #1
  :\yourloop        % notice the colon!
}
```

is natural. However, as we have introduced a ':' into each iteration of the loop, we should ensure that its expansion is empty. (`plain` has `\def\empty{}`.)

```
\catcode`\: \active
\let : \empty
```

The rules around the code indicate that it is to be part of the macro file `control.sty` and not an example.

There is another failing—`\break` will gobble

```
\youroperation #1
:
```

and leave

```
\yourloop        % notice the colon!
```

which is the `continue` command in *C*! To be successful, `\break` must consume also the token that follows the ':' delimiter.

(The author expects `\continue` will be used less often than `\break`. It causes the next iteration of the loop to begin. For example, to process only some of the input tokens, code similar to

```
\def\ignoresome #1 {
  \ifignore #1 \continue \fi
  % now process those tokens that
  % have not been ignored
  :\ignoresome
}
```

should be used, where `\ifignore` determines the fate of the token.)

The definitions

```
\def\break #1 : #2 { \fi }
\def\continue #1 : { \fi }
```

will be refined no more in this article.

3 Use \switch or \else!

Here we construct in TEX an analogue to the switch construction provided by *C*. It is useful when one of a list of cases is selected, depending on the value of some quantity. (Note that \switch does not share with *C* the *fall through* property. It is more like the CASE construction in Pascal.)

3.1 The alphabetic \fruit macro

Suppose we wish to write a macro \fruit such that \fruit a will result in \apple, \fruit b in \banana etc. One method is to produce a cascade of \if...\else...\fi statements. However, we could write

```
\def\fruit #1 {
  \switch \if #1 \is
      a \apple
      b \banana
      c \cherry
      d \date
  \end
}
```

if only we had a suitable \switch command. We will produce such a command. (The reader may benefit from trying to write such a command before reading on.)

First, some terms.

'\if #1' is the *test*
'a \apple' is the first *alternative*
'a' is the *key* to the first alternative
'\apple' is the *option* for the first alternative

It is clear that \switch must go through the alternatives one after another, reproducing the *test*

```
\def\switch #1 \is  %  the test
       #2 #3        %  key & option
{
  ...
  \switch #1 \is    % reproduce
}
```

and doing nothing unless the *key* fits the test

```
\def\switch #1 \is  %  the test
       #2 #3        %  key & option
{
  #1 #2 ... \fi  %  test key
  \switch #1 \is
}
```

in which case we should

- gobble to the end (marked by \end) of the expansion of \fruit
- insert the current *option* #3

and so we have

```
\def\switch #1 \is
       #2 #3
{
  % if  ( test key )  succeeds
  #1 #2 \exit #3 \fi  % do option
  \switch #1 \is
}
```

where \exit is a helper macro for \switch.

3.2 An \exit for \switch

As with \continue, \exit must gobble to some point and restore the \fi balance

```
\def\exit #1 \end { \fi }
```

but it should also pick up and reinsert the current option

```
\def\exit #1 #2 \end { \fi #1 }
```

which will work in the context of \fruit. It will fail if the option has several tokens.

For example, the definitions above expand

```
\switch \if a \is a {Jonathan} \end
```

to 'J'.

There are several solutions to this problem. Here is the one that executes the most rapidly.

```
\catcode`\@ 11^  %  make  @  a letter
                 %  Section 1.2  lies
\let \@fi \fi
\def\switch #1 \is #2 #3 {
  #1 #2 \@exit #3 \@fi
  \switch #1 \is
}
\def\@exit #1 \@fi #2 \end { \fi #1 }
\catcode`\@ 12^   %  put  @  back again
```

where \@exit and \@fi are helper macros, private to \switch.

3.3 Default actions for \switch

The expansion of '\fruit z' will fail horribly. As 'z' is not a key, \switch will read and discard up to the 'd \date' alternative, and then read \end and another parameter from the input stream. Now we are in trouble. \switch is still expanding, and there is no \end in sight.

Unless a matching key is sure to be found, a \switch should have a line handling the default. If \nofruit is to handle the default for \fruit, the line

```
#1 \nofruit
```

should be inserted as the last alternative.

(Another method would be to have \switch test for the \end token before reading '#1' and '#2'. Using macros to do this would result in a much slower \switch. But see section **11.**)

4 Applying \switch to \markvowels

By way of an example, we apply \switch to a problem posed and solved in Norbert Schwarz's *Introduction to TEX*, Ch7 §7.

4.1 The problem

We wish to write a macro \markvowels that prints the vowels of a given word in a different typeface. For example

```
\markvowels audacious.\endlist
```

is to give

audacious.

(There is a subtle reason why we use \endlist rather than \end. There is a surprise in the expansion of a \switch that has \end as a key.)

4.2 The solution

Here are some pointers for the solution.

- We need a \switch whose keys are a, e, i, o, u, \endlist and the default handler #1.
- Every letter, vowel or not, is to be printed.
- If a letter is a vowel, we apply \enbold to it.
- The expansion of \markvowels is to finish with \markvowels.
- When an option is selected, all up to the \end of the \switch is gobbled. For the key \endlist the two tokens #1 \markvowels must be absorbed.
- We are not obliged to use ':' when constructing a loop.

And here it is

```
\def\markvowels #1 {
  \switch \ifx #1 \is
      a \enbold
      e \enbold
      i \enbold
      o \enbold
      u \enbold
      \endlist \gobbletwo
     #1 \empty
  \end
  #1 \markvowels
}
```

with helpers

```
\def\enbold #1 {{ \bf #1 }}
\def\endlist { \gobble \endlist }
```

5 Finite State Automata (FSA)

The stomach of TEX, as the reader must well be aware, can be in one of number of states—horizontal mode, vertical mode, etc. The result of a command, such as \hbox{A}, will often depend on the current state. There are also rules that govern the transition from one state to another. Similarly, the text of a document passes from state to state—ordinary text, quotation, theorem, list item, and so forth. LATEX does this by changing the environment.

One way of coding such a device is to let the state be represented by a macro or parameter, whose value is then tested or altered by a single macro that contains code for *all* of the automaton's states. Although such a design is not without merit, here we will code Finite State Automata by using one macro for each state.

5.1 Skipping multiple blank lines

We proceed by means of an example. Suppose that we are \reading a file, and that we wish to ignore all but the first of adjacent blank lines. We have two states.

- \lastlineblank
- \lastlinenotblank

Here is a first attempt to code the states.

```
\def\lastlineblank {
  \read\thefile to \currentline
  \ifx\currentline\blankline
    % do nothing, call same state
    \lastlineblank
  \else
    \process\currentline
    \lastlinenotblank
  \fi
}

\def\lastlinenotblank {
  \read\thefile to \currentline
  \ifx\currentline\blankline
    \processblankline
    \lastlineblank
  \else
    \process\currentline
    \lastlinenotblank
  \fi
}
```

Although the above works for small files, it has a fault. Each time a line is read, the number of unbalanced \fis increases by one. The missing \fis (and other code) are pushed into the input stream, and will produce

```
! TeX capacity exceeded,
    sorry [input stack size=200].
```

before too long.

5.2 Using \end to \return a state

This problem arises because the next state is called before the current state is finished. As in \switch, at the end of each state macro we will place an \end marker, and use \return to move the next state to the head of the input stream. (*C* uses return to terminate a function with a specified value.)

Here we go. \lastlineblank should be

```
\def\lastlineblank {
  \read\thefile to \currentline
  \ifx\currentline\blankline
    \return\lastlineblank
  \else
    \process\currentline
    \return\lastlinenotblank
  \fi
  \end
}
```

where \return

```
\def\return #1 #2 \end { \fi #1 }
```

will gobble to the \end of the current state, balance the \fi, and place the next state at the front of the input stream.

To end the current state and do nothing more, the command \exit

```
\def\exit #1 \end { \fi }
```

should be called.

The command \end is merely a delimiter. We define

```
\let \end \empty
```

so that no harm occurs should it be executed.

5.3 Dealing with end-of-file

The code above continues to \read, even when the file has come to an end. An elegant solution is to write

```
\def\lastlineblank {
  \readfile\thefile\currentline\exit
  \ifx\currentline\blankline
    \return\lastlineblank
  \else
    \process\currentline
    \return\lastlinenotblank
  \fi
  \end
}
```

where \readfile takes three parameters.

- #1 an input stream number.
- #2 *the macro* the stream is to be read to.
- #3 the action to be taken on end of file.

Here is \readfile (see also **7** and **9.5**).

```
\def\readfile #1 #2 #3 {
  \ifeof #1
    % #3 may be several tokens
    % to be safe, we brace it
    \return { #3 }
  \else
    \read #1 to #2
  \fi
  \end
}
```

6 Choosing between ':' and \end

The delimiters ':' and \end perform similar but different functions. The programmer is advised on their use, and introduced to the last control *macro*, \chain.

6.1 The differences

':' and \end have the same \empty meaning. The difference is that ':' delimits \break and \continue, while \end delimits \return and \exit. Each of these macros will jump to ':' or \end, and put down a balancing \fi.

Although \break and \exit are analogues, \return has a flexibility that \continue lacks. We can (and must) decide what to \return but \continue provides no such choice.

To complete the use of ':' we introduce \chain. (See **9.5** for an example of its use.)

```
\def\chain #1 #2 : #3 { \fi #1 }
% Here ends control.sty .
% If you wish, restore white space.
```

6.2 Making the choice

Suppose that in the normal course of events, `\mymacro` will be followed by `\usualmacro`, where `\usualmacro` may or may not be `\mymacro`. Then the form

```
\def\mymacro ... {
    % code goes here
    % use \break, \continue
    % and \chain
  :\usualmacro
}
```

is preferred.

If there is no single most likely outcome, then

```
\def\mymacro ... {
    % code goes here
    % use \exit and \return
  \end
}
```

is probably best.

7 The `\fi` count problem

There is an error in `\readfile` that the author and the referees did not notice. However, when this macro was used, TEX found it.

We get the '`! Extra \fi.`' error. To understand why, suppose `\thefile` is at an end. `\lastlineblank` calls `\readfile` which `\returns` `\exit`. At this point the `\ifeof` in `\readfile` has been exactly matched by the `\fi` put down by `\return`. Now `\exit` gobbles to the `\end` and puts down *another* `\fi`. This is the error.

The problem is with the `\fi` count. `\exit` and the like put down `\fi`s to balance the ones they gobble. They have to do this, because TEX keeps in its main memory a record of each unbalanced `\if`. When the job is finished, they are reported. If TEX could be told not to do this, the balancing `\fi`s could be omitted and the problem would go away.

(The apology

```
! TeX capacity exceeded,
    sorry [main memory size=65533].
```

is produced when the macro

```
\def\doit { \iftrue \doit }
```

is executed.)

Given TEX as it is, it seems best to produce `\fi`-less verions of `\exit` and the like for precisely this situation. Replacing the `\exit` in `\lastlineblank` by

```
\def\gotoend #1 \end { }
```

will make the problem go away.

Or rather, this will move the problem. The macro writer now has to determine whether a balancing `\fi` is needed.

8 The Official version of `control.sty`

Here we list the version of `control.sty` that is to be used by macro writers. Compatibility with other macros demands that some changes be made.

- To allow active ':' to be used by other macro packages, ':' is made a letter, and throughout ':' is replaced by '`\::`'.
- Because the names `\break` and `\end` are already taken, uppercase names are used throughout.
- The `\fi`-less version of `\break` is to be `\BREAK`, while `\:BREAK` puts down a balancing `\fi`.

There is another problem — `\exit` and `\return` clash with `\switch`. All three macros use `\end` as a delimiter. This is not a desirable feature, and so `\SEND` (Switch-END) will be used to delimit `\SWITCH`.

Finally, by letting `\IS` equal `\fi` allows lines such as

```
\SWITCH \ifx #1 \IS
```

to be correctly skipped in conditional text (see *The TEXbook*, p. 211).

This article will now use these definitions.

```
\immediate\write16{ control.sty v1.0
 --- Jonathan Fine, 24 March 1991. }
\immediate\write16{ Public Domain, see
TUGboat (to appear) for documentation}

\catcode`\: 11    \catcode`\@ 11
\let\::\empty     \let\END\::

\def\BREAK#1\::#2{}
\def\CONTINUE#1\::{}
\def\CHAIN#1#2\::#3{#1}
\def\RETURN#1#2\END{#1}
\def\EXIT#1\END{}

\def\:BREAK#1\::#2{\fi}
\def\:CONTINUE#1\::{\fi}
\def\:CHAIN#1#2\::#3{\fi#1}
\def\:RETURN#1#2\END{\fi#1}
\def\:EXIT#1\END{\fi}

\let\@fi\fi       \let\IS\fi
\def\SWITCH#1\IS#2#3%
  {#1#2\@EXIT#3\@fi\SWITCH#1\IS}
\def\@EXIT#1\@fi#2\SEND{\fi#1}

```

26. `\catcode`\: 12 \catcode`\@ 12`

The author would like to receive examples of the use of these macros, and reports of problems and bugs.

As a general rule, before using a control macro that gobbles from *a control macro* **A**, to *a label,* `\END` *or* `\::` *as appropriate,* **B** start at **A** and read on until one reaches one of

- a `\fi`
- an `\if...`
- **B**

(but skip code enclosed by braces). In the first case use the `\fi`-ed version, otherwise the `\fi`-less.

9 Odds and Ends

Here are various bits and pieces that don't belong anywhere else. Some are quite important.

9.1 Name and Context

Other programming languages avoid conflict of names by giving each identifier a scope which is usually less than global. This is done by mapping each scoped identifier to a unique symbol, such as a number. I have work in progress that will add this capability to TEX. It will be a TEX macro package.

9.2 Nested conditionals

Because `\:BREAK` et al. replace only one gobbled `\fi`,

```
\if...
  \if...
    \:BREAK
  \fi
\fi
\::\nextmacro
```

will unbalance the `\fi`-count.

Rather than introduce `\::BREAK` it is better for the moment to say that such code is bad style, and discourage it. (The author would like to see any problem whose best solution requires breaking from a nested `\fi`.)

9.3 `\RETURN` or `\EXIT`

If the completed execution of `\mymacro` requires no parameters, *and* buildup of the input stack is not a problem, then instead of

```
\RETURN \mymacro
```

one can use

```
\mymacro \EXIT
```

which is slightly quicker. (`\EXIT` and any tokens between it and the matching `\END` will be sitting in the input stack waiting to be skipped until `\mymacro` has done its work.)

9.4 Dedicated `\SWITCH`

If large use is made of, for example,

```
\SWITCH \ifx #1 \IS
```

then it is better to use a specially adapted switch.

```
\def\SWITCHx #1 #2 #3 {
  \ifx #1 #2 \@EXIT #3 \@fi
  \SWITCHx #1
}
```

9.5 A better `\readfile`

In the expansion of `\readfile`, `#3` is read, copied into place, and then either thrown away or read and copied again.

In the normal course of events, `\readfile` needs only `#1` and `#2`. The end of file action `#3` will be discarded. Thus,

```
\def\readfile #1 #2 {

   \read #1 to #2
\::\gobble   % gobble '#3'
}
```

is a step towards the more efficient (and smaller)

```
\def\readfile #1 #2 {
  \ifeof #1
    \:CHAIN \unbrace
  \fi
  \read #1 to #2
\::\gobble   % gobble '#3'
}
```

Note that `\::` is very helpful, even though `\readfile` is not a loop.

10 Performance

It seems that once the idiom is mastered, these basic control macros will make it easier to write TEX macros.

The result will be code that is concise and relatively easy to understand. Code that is compact will load more rapidly from mass storage and use fewer words of memory.

It also seems likely that the idiom here will encourage utility commands, such as `\readfile`. This will reduce the size of both the code and the hash table.

Where speed of execution is paramount, custom devices are required. A carefully crafted cascade of `\if ... \else ... \fi` statements will run somewhat quicker than the `\switch` alternative. In other areas of programming, the prevailing wisdom is that good algorithms make for rapid execution.

Once the program is tested and running properly, significantly quicker performance can be obtained by rewriting a small amount of the code in lower level commands.

11 Enhancements to TEX

The Grand Wizard has said "no further changes except to correct extremely serious bugs" (*TUGboat* 11, no. 4, p. 489, June 1990) but this does not stop the wanting. That unbalanced `\if`s accumulate in memory without limit has already been mentioned.

Here are two devices that would improve the basic control macros of this article.

`\nil`—a primitive that does nothing. Although this is available as a macro, `\def\nil{}`, as an interpreted command it is over three times slower than the primitive `\relax`, which does slightly more!

`\abort`—a command which when passed as a parameter to a macro immediately halts its expansion. (If the macro is not `\long` then the *token* `\par` has the desired effect, but the error condition so generated is an unwanted side-effect.)

For the rest of the section, suppose that `\abort` has *both* of these properties. There are nice results. Provided we

```
\let \SEND \abort
```

the `\fruit` with no action as default becomes

```
\def\fruit #1 {
  \SWITCH \if #1 \IS
      a \apple
      ...
      d \date
  \SEND
}
```

which is more intuitive.

Provided `\END` is also set to `\abort`, it can be used to delimit `\markvowels`, which becomes fewer tokens executing faster.

```
\def\markvowels #1 {
  \SWITCH \ifx #1 \IS
      a \enbold
      ...
      u \enbold
      \SEND
      #1 \markvowels
  }
```

It is also simpler. The somewhat obscure line

```
\endlist \gobbletwo
```

is no longer needed.

Finally, we (almost) have an elegant means of handling default values.

```
\let \DEFAULT \abort
```

allows

```
\def\fruit #1 {
  \SWITCH \if #1 \IS
      a \apple
      ...
      d \date
      \DEFAULT \nofruit
  \SEND
}
```

to code a default value of `\nofruit`.

(There are two problems here. If `\nofruit` expects a parameter, it will get `\SEND`, which will then `\abort` it! This is wrong. In this situation

```
\def\USE #1 \SEND { #1 }
```

will allow

```
\DEFAULT \USE \nofruit
```

to correctly code such a default. The second problem is more serious. Should a key satisfy the test, such a `\DEFAULT` will `\abort` the `\@EXIT` macro called by `\SWITCH`. This is wrong.)

This area needs further investigation.

12 A Groaning Pun

Asked to write a macro `\goodthing` that does something useful, the design

```
\def\goodthing ... {
  ...
  \END
}
```

was used by 7 out of 10 programmers.

This only goes to show that most `\goodthings` come to an `\END`.

◇ Jonathan Fine
203 Coldhams Lane
Cambridge CB1 3HY
England
Tel +44 223 215389