The \CASE and \FIND macros

Jonathan Fine

Abstract

This article is a continuation of the author's *Some Basic Control Macros for TEX* in *TUGboat* 13, no. 1. It introduces macros \CASE and \FIND which are useful for selecting an action to be performed on the basis of the value of a parameter. These macros cannot be used in the mouth of TEX. Also, some changes to the *Basic Control Macros* are reported.

Introduction

First an example is given of the use of the \CASE macro, and then the macro itself is given. The next section does the same, for the \FIND macro. On both occasions, step-by-step examples of the functioning of these macros are given. A discussion of pitfalls in the use of the macros follows, and some other items, and finally a report on the *Basic Control Macros* is given. To the best of my knowledge, there are no jokes in this article.

Much of the inspiration for \CASE and \FIND came from studying Mittelbach and Schöpf's article *A new font selection scheme for TEX macro packages — the basic macros* in *TUGboat* 10, no. 2, while the rest came from the author's own needs.

Independently, Kees van der Laan has developed a macro \loc which has something in common with \FIND. It can be found on page 229 of his article *FIFO and LIFO incognito*, which appears in the *EuroTEX 92* proceedings, published by the Czechoslovak TUG.

In about 1,000 lines of documented code the author had occasion to use \continue (and the ':' variant) 17 times, \return 5 times, and \break but once. The macro \CASE was used 5 times, and \FIND twice. By comparison, the 17 primitive \if... commands of TEX were used 35 times altogether.

Acknowledgements The author thanks the referee for many helpful comments, and particularly for requesting a more accessible exposition.

1 The \CASE macro

The \CASE macro is similar to the \SWITCH macro defined in *Basic Control.* However, it requires assignment and so cannot be used in the mouth of TEX.

By way of an example, suppose that one wishes to code a macro \fruit such that

\fruit\a *produces* apple
\fruit\b *produces* banana
\fruit\x *produces* \error\x

where \error is to handle unknown arguments to the \fruit command.

To code the macro \fruit, the association of the ⟨*key*⟩s \a, \b, etc., with the ⟨*action*⟩s apple, banana, \error\x must be stored in some form or another. Using the semicolon ';' as a delimiter, the code fragment

```
\a apple ;
\b banana ;
\x \error \x ;
```

will store the data for the \fruit macro. Each of the lines above we will call an ⟨*alternative*⟩.

The \fruit macro can be coded as

```
\def\fruit #1
{
  \CASE #1
      \a apple ;
      \b banana ;
      % default action
      % omit at your own peril
      #1 \error #1 ;
  \END
}
```

where the macro \CASE is to search for the token #1 amongst the ⟨*key*⟩s and then extract and execute the associated ⟨*action*⟩. (This and all other code in this article is assumed to be read in an environment where white space characters are ignored. I will explain later how to set this up.)

Here is the code for the \CASE macro which supports this style of programming.

```
\long\def\CASE #1
{
   \long
   \def\next ##1          % discard
             ;#1          % find ⟨key⟩
             ##2;         % ⟨action⟩
             ##3 \END     % discard
             { ##2 }      % copy ⟨action⟩
   \next ;  % do \next - note the ';'
}
```

Perhaps the easiest way of understanding the \CASE macro is to follow its functioning step by step. We shall do two examples, \fruit\b and \fruit\x. The example of \fruit\a — which is left to the reader — shows why the ';' is required after \next in the definition of \CASE.

The expansion of \fruit \b is

```
\CASE \b \a apple;\b banana;\b \error \b ;\END
```

and now \CASE expands to produce

```
\long \def \next #1;\b #2;#3\END {#2}
\next ;\a apple;\b banana;\b \error \b ;\END
```

and so \next will be defined as a \long macro with ;\b and ; and \END as delimiters. After \next has been defined the tokens

```
\next ;\a apple;\b banana;\b \error \b ;\END
```

remain. Now for the crucial step. By virtue of the definition of \next, *all tokens up to the* \END *will be absorbed while forming the parameter text for* \next. The tokens ;\a apple form #1. *The vital parameter* #2 *is* banana. Finally, #3 get \b \error \b;. Thus, the result of the expansion of \next will be

```
banana
```

just as desired. (*The TEXbook* discusses macros with delimited parameters on pages 203–4.)

Now for \fruit \x. The first level expansion will be

```
\CASE \x \a apple;\b banana;\x \error \x ;\END
```

where the #1 in the default option has been replaced by \x. As before, \CASE \x will define \next to be delimited by ;\x and ; and \END. This time, because \x is not an explicit key within \fruit, the default ⟨*action*⟩

```
\error \x
```

will be the result of the expansion of \next. As mentioned earlier, \error is to handle unknown arguments to \fruit.

This last example shows the importance of coding a default option within a \CASE. This option should be placed last amongst the ⟨*alternative*⟩s. If omitted an unknown key will cause the scratch macro \next to not properly find its delimiters. Usually, this will result in a TEX error.

2 The \FIND macro

There are situations — for example the problem of printing vowels in boldface — where several of the values of the parameter will give rise to what is basically the same action. The \FIND macros is better than \CASE in such situations.

Suppose that the desired syntax is that

```
\markvowels Audacious \end
```

is intended to produce

Audacious

where \end is used as delimiter.

The macro \markvowels can be coded as a loop, reading tokens one at a time. It is to be concluded when \end is read. Should the token read by \markvowels be a vowel, then this letter should be printed in boldface, otherwise the token should be printed in the default font. Vowel or not, \markvowels is a loop and so after processing a non-\end token \markvowels should be called again.

Thus, there are three sorts of actions

- print token in \bf and continue
- print token in default font and continue
- end the loop — i.e. do nothing

and as any of the ten letters aeiouAEIOU give rise to the first type of action, it is better to use \FIND, which is similar to \CASE except that a single alternative can have several keys.

The syntax for \CASE is

```
\CASE ⟨search token⟩
   % one or more times
   ⟨key⟩ ⟨option⟩ ;
   % don't forget the default
\END
```

while for \FIND the syntax is

```
\FIND ⟨search token⟩
   % one or more times
   % one or more ⟨key⟩s
   ⟨key⟩ ... ⟨key⟩ * ⟨option⟩ ;
   % don't forget the default
\END
```

where \FIND will look for the ⟨*search token*⟩ (amongst the ⟨*key*⟩s, we hope) and having found it will save the ⟨*option*⟩ (between * and ;) as it gobbles to the \END, and then execute the ⟨*option*⟩.

So much for the theory. We shall now code the macros \markvowels and \FIND, and then run through some examples step by step. Here is the enboldening macro coded.

```
\def\markvowels #1
{
  \FIND #1
    % the ⟨action⟩ for \end is empty
    \end  * ;

    % vowels
    aeiou AEIOU
    * {\bf #1} \markvowels ;

    % other tokens
    #1 * #1 \markvowels ;

  \END
}
```

where \FIND should search for the ⟨*key*⟩ and then the next * tag. What follows up to the next ; is the selected ⟨*action*⟩, which is to be reproduced. The remaining tokens up to \END are discarded.

```
\long\def\FIND #1
{
   \long
   \def\next ##1        % discard
              #1        % find ⟨key⟩
             ##2 *      % discard up to
                        %   next tag
             ##3 ;      % ⟨action⟩
             ##4 \END   % discard
            { ##3 }     % copy ⟨action⟩
   \next               % do \next
}
```

Now for the examples. We shall follow the expansion of `\markvowels AZ\end`, step by step.

First, `\markvowels A` will expand to yield

```
\FIND A\end *;aeiouAEIOU*{\bf A}\markvowels ;
  A*A\markvowels ;\END Z\end
```

(please note the `Z\end` awaiting processing after the `\END`) and now `\FIND` expands

```
\def \next #1A#2*#3;#4\END {#3}\next
  \end *;aeiouAEIOU*{\bf A}\markvowels ;
  A*A\markvowels ;\END Z\end
```

to define `\next` delimited by `A * ; \END`. The expansion

```
\next
  \end *;aeiouAEIOU*{\bf A}\markvowels ;
  A*A\markvowels ;\END Z\end
```

of `\next` will result in

```
{\bf A}\markvowels Z\end
```

and so the letter `A` will be set in `\bf`. The tokens `Z\end` have been carried along, from the beginning of this example. Next, `\markvowels` is expanded

```
\FIND Z\end *;aeiouAEIOU*{\bf Z}\markvowels ;
  Z*Z\markvowels ;\END \end
```

and as before `\FIND` results in

```
\long \def \next #1Z#2*#3;#4\END {#3}\next
  \end *;aeiouAEIOU*{\bf Z}\markvowels ;
  Z*Z\markvowels ;\END \end
```

the definition of `\next` (delimiters `Z * ; \END`), whose expansion

```
\next
  \end *;aeiouAEIOU*{\bf Z}\markvowels ;
  Z*Z\markvowels ;\END \end
```

produces

```
Z\markvowels \end
```

and so `Z` is set in the default font. Now for `\markvowels \end`, which expands to

```
\FIND \end \end *;
  aeiouAEIOU*{\bf \end }\markvowels ;
  \end *\end \markvowels ;\END
```

and again `\FIND` defines `\next`

```
\def \next #1\end #2*#3;#4\END {#3}\next
  \end *;aeiouAEIOU*{\bf \end }\markvowels ;
  \end *\end \markvowels ;\END
```

(with delimiters `\end * ; \END`) and the expansion

```
\next
  \end *;aeiouAEIOU*{\bf \end }\markvowels ;
  \end *\end \markvowels ;\END
```

of `\next` is **empty**. (Why is this? The macro `\next` will first search for `\end`. The tokens before this `\end` form `#1`. They happen to be empty, but in any case they are discarded. Similarly, `#2` is empty, and is discarded. However, `#3` is the ⟨*action*⟩, and in this case it is empty. The remaining tokens, between `\end *;` and `\END`, form `#4`, and are discarded.)

(In terms of `\FIND`, the `\loc` macro of van der Laan can be written as

```
\def\loc #1#2
{
  \FIND #1
    #2 * \let\iffound\iffalse ;
    #1 * \let\iffound\iftrue  ;
  \END
}
```

but there is no easy expression for `\FIND` in terms of `\loc`.)

3 Warnings

There are several ways in which these macros can trip up the unwary.

No default A default action must be supplied, and it should be the last option, unless you are certain that it will never be required. The code fragment

```
\lowercase{ \CASE #1 }
    h \help ;
    #1 ;
\END
```

lacks a default, for when `#1` is `A` the fragment

```
\CASE a
    h \help ;
    A ;
\END
```

remains once `\lowercase` has executed. To avoid this, either apply `\lowercase` to the *whole* `\CASE` statement, or write

```
\lowercase{ \amacro #1 }
```

where `\amacro` contains the `\CASE` statement.

Meaning ignored The `\CASE` and `\FIND` macros depend on the token passed as parameter, but *not* on its `\meaning`. This token can be a control sequence or a character token. Thus, the operation

of \markvowels is independent of the meaning of \end. This is often what is wanted, but is different from usual \ifx comparison.

Braces stripped Selecting an option such as

```
\group * { \bf stuff } ;
```

within \FIND will result in

```
\bf stuff
```

being processed without the enclosing braces — an error which nearly occurs in \markvowels. This is a consequence of TEX's rules for reading parameters. The same failure can happen with the \CASE macro.

Braces not supplied Consider the macro

```
\def\puzzle #1
{
  \FIND #1
      abc * [#1] ;
      def * (#1) ;
      #1  * '#1' ;
  \END
}
```

applied to x. The result of \puzzle x will not be the default 'x'. It will be (x)!

The invocation of \FIND x will produce

```
\long\def\next #1 x #2 * #3 ; # 4 \END
  {#3}
```

and as x will replace #1 in \puzzle, the parameters to \next will be (delimiters italicized)

```
#1 <-  abc * [  x
#2 <-  ] ; def  *
#3 <-  (x)  ;
#4 <-  x * 'x'  \END
```

and so in this situation the ⟨*action*⟩ for def will have been selected.

The problem is that #1 is prematurely visible. The solution is to hide it. This is done by writing

```
abc * { [#1] } ;
def * { (#1) } ;
```

which has enclosed the troublesome ⟨*action*⟩s in braces. As mentioned earlier, these braces will be stripped before the action is executed.

Surplus semicolons Code such as

```
\FIND #1
    0123456789
    * \action\one ;
      \action\two ;
    #1 * \default #1 ;
\END
```

is deceptive. When the parameter is 1 only \action\one will be performed. (There is an erroneous semicolon that the eye easily misses.) In this context the layout

```
\FIND #1
    0123456789
    *
      \action\one
      \action\two
    ;
    #1 * \default #1 ;
\END
```

reads better.

4 Setting up the catcodes

The macros \CASE and \FIND will have confusing results if the characters ; or * are passed as parameters. This may happen if the document author writes \fruit; or \markvowels Abc; def \end. To prevent this confusion while preserving the syntax we shall alter some catcodes. We shall also ignore white space. By setting

```
\catcode`\;=4      \catcode`\*=4
\catcode`\ =9      \catcode`\^^I=9
\catcode`\^^M=9    \catcode`\~=10
```

at the beginning of the file containing \CASE and \FIND, and macros calling \CASE and \FIND, and placing

```
\catcode`\;=12     \catcode`\*=12
\catcode`\ =10     \catcode`\^^I=5
\catcode`\^^M=10   \catcode`\~=13
```

to restore values at the end of the file, we can be sure that any ; or * characters generated by a document author will not match the private delimiting tokens ; and * used within \CASE, \FIND, and their calling macros.

The character ~ has been given a \catcode of 10 which is ⟨*space*⟩. According to *The TEXbook*, p47, when a character with \catcode ⟨*space*⟩ is read from a file, it is "converted to a token of category 10 whose character code is 32" and so ~ can be used to place an *ordinary* space token into a macro. Incidentally, it is a consequence of this rule, and the rules for \uppercase, \lowercase, and \string (see pages 40–41) that it is impossible to place a character token with category 10 and character code zero into the stomach of TEX.

(The characters ; and * have been given \catcode 4, which is ⟨*alignment tab*⟩, to help detect errors. If the TEX error message

```
! Misplaced alignment tab character ;.
```

or similar with * occurs, then there is an error in the coding or execution of a \CASE or \FIND macro.)

5 Variable delimiter macros

The macros \CASE and \FIND are particular examples of what I call *variable delimiter macros*. They are useful for control and selection. Their essence is to define and execute a scratch macro — \next — which has as delimiter a token that was originally passed as a parameter.

Even though TEX is fixed and unchanging, change can be discussed. Currently a macro parameter character # cannot serve as a delimiter for a macro. The code

```
\def\a ## {}
```

will produce the TEX error

```
! Parameters must be numbered
    consecutively.
```

and this provides a place for an extension to be built.

Suppose that ## were allowed in the parameter text of a macro, to stand for a variable delimiter. Then \FIND could be coded as

```
\long\def\FIND #1 ## #2 * #3 ; #4 \END
  { #3 }
```

where the expansion of \FIND consists of first replacing ## by the next token in the input stream (assumed not { or } or #) and then expanding the resulting macro.

The code in this style

```
\long\def\CASE #1 ; ## #2 ; #3 \END
  { #2 }
```

for \CASE is not quite right, for it misses the vital semicolon after \next in the original definition.

6 Benefits of the \noname package

The catcode changes listed above — or rather the effect of these changes — is obtained automatically should the macro file be processed by the author's \noname package, which is described in *TUGboat* 13, no. 4. Should the macro writer wish to place an *ordinary* ; or * within a \CASE or \FIND macro, this can easily be done using \noname. (Without \noname this will require explicit and unpleasant dirty tricks.)

The \noname package will also translate the label ':' used by the *Basic Control* macros into an otherwise inaccessible control sequence, as it \loads a macro source file.

The step-by-step expansion of examples of the use of the \CASE and \FIND macros was generated by the single-step debugger \ssd which is also part of the \noname package. (The output has been lightly edited to improve the appearance, and particularly to get decent line breaks.)

7 Basic Control — a report

Experience has brought the following changes to the basic control macros.

In the original article, both ':' and \END were used as delimiting labels. It turns out to be more convenient to have but one label. Thus one has

```
\long\def\break #1 : #2 {}
\long\def\continue #1 : {}
\long\def\chain #1 #2 : #3 { #1 }
\long\def\return #1 #2 : { #1 }
```

and the \fi'ed variants, but \exit has gone and \return gobbles to ':' rather than to \END. Another change — the macros are now \long.

Finally, the *soft double-fi*

```
\def\::fi { \fi \fi }
```

is introduced for the situations where one would like to have \::continue, etc., available. (Just write \::\continue instead.)

The macro \switch has so far turned out to be not so useful. Much of its functionality has been subsumed by \CASE and \FIND.

◇ Jonathan Fine
203 Coldhams Lane
Cambridge
CB1 3HY
England
J.Fine@pmms.cam.ac.uk