A MACRO MENAGERIE

Brendan D. McKay

1. Math-style testing

One of the things which one should be able to do in TEX, but which apparently is impossible, is to test for the current math-style (display, text, script or scriptscript). For example, how does one write a macro which produces a bold-face "g" of the right size? (`\def\g{\hbox{\bf g}}` obviously doesn't.) Here's a trick which doesn't completely solve the problem, but which goes a long way towards that goal.

```
\xdef\subscr{↓}  \xdef\supscr{↑}
\chcode'1=13 \chcode'136=13
\def\MS{T}
\def↓#1{\subscr{\if\MS T{\def\MS{S}#1}\else
   {\def\MS{X}#1}}}
\def↑#1{\supscr{\if\MS T{\def\MS{S}#1}\else
   {\def\MS{X}#1}}}
```

The idea is to maintain a macro `\MS` which has the value "S" for scriptstyle; "X" for scriptscript-style, and "T" for all other styles, including non-math mode. This can then be tested using the `\if` macro. The definitions above will maintain `\MS` correctly if the style changes by use of the sub-script or superscript characters, but not otherwise. Style change macros like `\scriptstyle` can also be redefined to maintain `\MS`, but automatic changes caused by things like `\over` will go undetected. In such cases the user must define `\MS` himself, if it is going to be tested.

As a sample application, here is a definition of `\bf` ("select bold-face") which behaves the same as normal in non-math mode and selects a font of the right size in math-mode. In the latter case it acts only on the following character, control sequence or group. Let's suppose that A, B and C are bold-face fonts of the required sizes.

```
\def\bf{\ifmmode{\gdef\Fnt##1{\hbox
   {\if\Ms T{\:A##1}\else
   {\if\MS S{\:B##1}\else{\:C##1}}}}}\else
   {\gdef\Fnt{\:A}}\Fnt}
```

In our second application we'll define a macro for raising a portion of text. If you type "`\lift⟨text⟩\by⟨dimen⟩\\`", then ⟨text⟩ is put in a `\hbox` and raised by an amount ⟨dimen⟩. ⟨text⟩ will appear in the current style, except that display style and text style are not distinguished.

```
\def\lift#1\by#2\\{\raise#2\hbox{\ifmmode
{\if\MS T{$#1 $}\else
   {\if\MS S{$\scriptstyle#1$}\else
   {$\scriptscriptstyle#1$}}}\else{#1}}}
```

2. Groupless `\if`s

A good source of inscrutable bugs involves the way that TEX handles conditionals like `\if`, `\ifpos`, `\ifvmode` etc.. Let's suppose that we want to select font A if the macro `\format` has the value "1" and font B otherwise. The obvious method is

```
\if\format1{\:A}\else{\:B}
```

but this doesn't work. The reason is that the text produced is not "`\:A`" or "`\:B`", but "`{\:A}`" or "`{\:B}`". Since font definitions are revoked at the end of groups the total effect is ⟨nothing useful⟩. It is sometimes handy to have another version of `\if` which avoids this rather unsatisfactory state of affairs. While we're at it, we'll change the format to

```
\If⟨char1⟩⟨char2⟩⟨true text⟩\else
   ⟨false text⟩\endif
```

— a few fewer braces never hurt anybody. Three possible definitions for `\If` are as follows.

```
(1) \def\If#1#2#3\else#4\endif
    {\if#1#2{\gdef\Iftemp{#3}}\else
    {\gdef\Iftemp{#4}}\Iftemp}
(2) \def\else#1\endif{}  \def\endif{}
    \def\If#1#2{\if#1#2{\gdef\Iftemp{}}\else
    {\gdef\Iftemp##1\else{}}\Iftemp}
(3) \def\If#1#2{\if#1#2{\gdef\Iftemp##1\else
    ##2\endif{##1}}\else{\gdef\Iftemp##1\else
    ##2\endif{##2}}\Iftemp}
```

All three definitions work in most ordinary circumstances. The first definition has the unpleasant peculiarity that any #s which occur in ⟨true text⟩ or ⟨false text⟩ must be typed as ##, a problem which grows exponentially if `\If`s are nested. The second definition avoids this problem but has another deficiency: it won't nest properly (*why?*). The third definition avoids both problems. If ⟨true text⟩ or ⟨false text⟩ contains another `\If`, simply enclose it in {}s. This doesn't cause grouping, of course, but it will ensure that each `\else` or `\endif` gets paired with the right `\If`.

3. Recursion

Although the TEX manual apparently never says so, the macro facility in TEX is completely recursive. In other words, macros can directly or indirectly call themselves. Of course, we are not given this little gem of information because the knowledge would be almost useless. Nevertheless, there is a little gap between "almost useless" and "completely useless", and this section is devoted to exploring it. Three applications of recursion we will consider are (i) loop structures, (ii) counter arithmetic and (iii) macros accepting variable numbers of arguments.

(i) Loops are quite easy to create in TEX as long as one respects TEX's finite capacity. In order to make loops which can be repeated a large number

of times, the recursive call must be the very last thing in the expansion, and in particular it must not be in a group (TEX won't nest groups to an indefinite depth). The last requirement means that the recursive call can't be part of the result text of a conditional (see Section 2). Here's some examples:

```
\def\savecount#1#2{\ifpos#1{\xdef
   #2{\count#1}}\else
   {\setcount#1-\count#1\xdef#2{-\count#1}
   \setcount#1-\count#1}}
\gdef\Wtemp#1#2{#2\Wloop#1{#2}}
\def\Wloop#1#2{\ifpos#1{}\else
   {\gdef\Wtemp##1##2{}}\Wtemp#1{#2}}
\def\while#1#2\endwhile{\Wloop#1{#2}\gdef
   \Wtemp##1##2{##2\Wloop##1{##2}}}
\def\repeat#1\times#2\endrepeat
   {\savecount9\Rtemp\setcount9#1
   \while9{#2\advcount9by-1}\endwhile
   {\setcount9\Rtemp}}
```

\savecount⟨digit⟩⟨control sequence⟩ saves the value of a counter in a control sequence. \while⟨digit⟩⟨text⟩\endwhile will produce ⟨text⟩ repeatedly until \count⟨digit⟩ becomes non-positive. (Presumably ⟨text⟩ will set the counter non-positive eventually.) \whiles can be nested if they use different counters.

\repeat⟨value⟩\times⟨text⟩\endrepeat will produce ⟨text⟩ precisely ⟨value⟩ times, where ⟨value⟩ can be either a number or a counter. The use of \Rtemp in \repeat enables \repeats to be nested to one level, but no further. For example, \repeat5\times{x-\repeat3\times aA\endrepeat}\endrepeat produces

x-aAaAaAx-aAaAaAx-aAaAaAx-aAaAaAx-aAaAaA.

(ii) The fact that counter operations like multiplication and division are not provided by TEX is one indication of their likely usefulness. Of course, that won't stop us from doing these operations anyhow.

```
\def\neg#1{\setcount#1-\count#1}
\def\Hlf#1#2 {\advcount9by-#2\advcount9by-#2
   \ifpos9{\advcount#1by#2}\else
   {\advcount9by#2\advcount9by#2}}
\def\halve#1{\savecount9\Htemp
   \setcount9\count#1\advcount9\setcount#1 0
   \Hlf#11073741824 \Hlf#1536870912
   \Hlf#1268435456 \Hlf#1134217728
   \Hlf#167108864 \Hlf#133554432 \Hlf#116777216
   \Hlf#18388608 \Hlf#14194304 \Hlf#12097152
   \Hlf#11048576 \Hlf#1524288 \Hlf#1262144
   \Hlf#1131072 \Hlf#165536 \Hlf#132768
   \Hlf#116384 \Hlf#18192 \Hlf#14096
   \Hlf#12048 \Hlf#11024 \Hlf#1512 \Hlf#1256
   \Hlf#1128 \Hlf#164 \Hlf#132 \Hlf#116 \Hlf#18
   \Hlf#14 \Hlf#12 \Hlf#11 {\setcount9\Htemp}}
\def\multiply#1\into#2{\setcount8#1\setcount9
   \count#2\setcount#2 0
   \while8\ifeven8{}\else
   {\advcount#2by\count9}
   \advcount9by\count9\halve8\endwhile}
\def\divide#1\into#2{\setcount9\count#2
   \setcount#2-1\advcount9
   \while9{\advcount#2by1
   \advcount9by-#1}\endwhile}
\def\Divide#1\into#2{\ifpos#2{\divide#1\into
   #2}\else{\neg#2\divide#1\into#2\neg#2}}
\def\sqroot#1{\setcount9\count#1\advcount9
   \setcount#1-1\setcount81
   \while9{\advcount9by-\count8
   \advcount#1by1\advcount8by2}\endwhile}
```

\halve⟨digit⟩ will divide any counter other than counter 9 by two, provided its original value is in the range 0 to 4294967294. Some of the earliest calls to \Hlf will need to be removed for machines with small word-sizes. \sqroot⟨digit⟩ will take the square root of any non-negative counter other than counter 8 or 9. In the other cases, the format is \operation⟨value⟩\into⟨digit⟩, where ⟨value⟩ is a number or a counter and ⟨digit⟩ is a counter number for the other argument and the answer. ⟨value⟩ must be non-negative in each case. \count⟨digit⟩ may be negative for \Divide or \multiply but not for \divide. The restrictions on which counters can't be used and which counters are destroyed are most easily seen by examining the definitions. Both \halve and \multiply are quite fast, but \divide, \Divide and \sqroot take time proportional to the answer.

(iii) The method by which recursion can allow a macro to apparently accept any number of arguments is best illustrated by an example. The macro \options below will accept any number of single character arguments, each of which will presumably cause some useful action. If an "x" occurs it must be followed by two arguments (which somehow belong to the x). Also, a "d" implies a "j" as well. The end of the argument list is indicated by a period. A possible call would be "\options rx{30pt}{75pt}d.".

```
\def\options#1{def\Next{\options}
   \If a#1⟨something⟩\else\endif
   \If j#1⟨something⟩\else\endif
   \If x#1
    \def\Next##1##2{⟨something⟩\options}
    \else\endif
   \If d#1⟨something⟩
    \options j. \def\Next{\options}\else\endif
   \If .#1\def\Next{}\else\endif
   \Next}
```

A macro of this sort is invaluable in writing a general purpose macro package, especially one to be used by many people. A large number of different style options can be provided, and each user can easily select any combination.

Research Problems:

(1) Speed up \divide and \sqroot.

(2) Write a macro which tests two character strings for a character in common. Then dream up an application.

4. Pictures

In this section we describe a few macros which can facilitate the drawing of complicated diagrams. The two macros at the heart of the method are these:

```
\def\picture#1#2#3#4\endpicture{{\varunit#1
   \vbox to #2{\vss\hbox to #3{\!#4\hss}}}}
\def\put#1(#2,#3){\raise#3vu\hbox to 0pt
   {\hskip#2vu#1\hss}\!}
```

The result of \picture⟨dimen$_1$⟩⟨dimen$_2$⟩⟨dimen$_3$⟩⟨hlist⟩\endpicture is a vertical box of height ⟨dimen$_2$⟩ containing a horizontal box of width ⟨dimen$_3$⟩ which contains ⟨hlist⟩. ⟨dimen$_1$⟩ becomes 1vu. Each position in the picture has coordinates of the form (x, y), where x is the distance in vu from the left boundary and y is the distance in *vu* from the bottom of the picture. Thus (0,0) is the reference point of the picture. To put ⟨something⟩ at position (36,475) simply type \put⟨something⟩(36,475). The second coordinate cannot be negative. Both coordinates can be specified as the values of counters.

By putting \puts inside \puts, a temporary change of origin can be affected, allowing sections of the picture to be moved around in one piece. For even greater flexibility, picture a \picture within a \picture. (The inside \picture should be given width zero.) The overall scale of the picture can be adjusted by changing ⟨dimen$_1$⟩.

Just for fun, we'll give macros for inserting horizontal or vertical rules into a picture and for drawing dotted lines.

```
\def\line(#1,#2)(#3,#4){\put\setcount8#4
   \advcount8by-#2
   \ifpos8{\hskip-0.2pt
   \vrule depth0pt width0.4pt height \count8vu}
   \else{\setcount8#3\advcount8by-#1
   \vrule depth0.2pt width \count8vu
   height 0.2pt}(#1,#2)}
\def\speck{\hskip-0.3pt
   \vrule height0.3pt depth0.3pt width0.6pt}
\def\dotline#1(#2,#3)(#4,#5){\put
   \speck(#2,#3)\setcount7#4
   \advcount7by-#2\setcount8#5\advcount8by-#3
   \Divide#1\into7\Divide#1\into8
   \setcount5#2\setcount6#3
   \repeat#1\times\advcount5by\count7
   \advcount6by\count8
   \put\speck(\count5,\count6)\endrepeat\!}
```

\line((coords$_1$))((coords$_2$)) will draw a solid line between the points given. These must be specified in the order left–right for a horizontal rule and bottom–top for a vertical rule.

\dotline⟨value⟩((coords$_1$))((coords$_2$)) will draw a dotted line consisting of ⟨value⟩+1 \specks between the points specified, which can be given in either order. The last \speck can be misplaced by up to ⟨value⟩vu due to rounding error, so 1vu should be small if ⟨value⟩ is large. \dotline can be used to make solid diagonal lines by placing many small dots very close together, but you won't get far before TEX runs out of space. Both \line and \dotline will only accept integer coordinates, but this is no restriction if 1vu is small.

\picture can also be used as a very versatile and simple to use system for creating complicated symbols, like ⊞.

We conclude with a couple of more complicated \pictures. Here is the source for the second:

```
\def\overt{\lower2.5pt\hbox
   {\hskip-2.3pt\:u\char'5}}
\def\cvert{\lower2.5pt\hbox
   {\hskip-2.3pt\:u\char'17}}
\picture{0.083pt}{size}{size}
   \setcount4 5000\setcount5 4980
   \setcount6 4620\setcount7 0
   \while4{\setcount3 3800\setcount2 3780
   \setcount1 3420
   \while3\ifeven7
   {\put\overt(\count3,\count4)}\else
   {\put\cvert(\count3,\count4)}\ifpos1{\line
   (\count1,\count4)(\count2,\count4)}\else{}
   \ifpos6{\line
   (\count3,\count6)(\count3,\count5)}\else{}
   \advcount7\advcount1by-400\advcount2by-400
   \advcount3by-400\endwhile
   \advcount7\advcount4by-400\advcount5by-400
   \advcount6by-400}\endwhile
\endpicture
```