* * * * * * * * * * * * * *

MACRO MADNESS

Michael Spivak
2478 Woodridge Drive,
Decatur, GA 30033

This article is an extract from the documentation for the not-yet-completed AMS-TEX macro package. It discusses certain tricks and pitfalls that other macro writers might want to know about. Needless to say, none of this trickery would have been possible without the help of Don Knuth.

It should be mentioned that the AMS-TEX macro package initially \chcodes the symbol | (ASCII '174) to be a letter, and all internal AMS-TEX macros contain a | as one of their letters. At the very end of the macro file, | is re-\chcoded to be of type 12, so that the AMS-TEX user cannot redefine, or even use, these control sequences (the input \cs| will be read as \cs |). For convenience, we will omit the |s here, and we will use mnemonic names for control sequences—the actual names used by AMS-TEX are very short (at most three letters, including any |s), in order to preserve memory space.

Please report any bugs to the above address as soon as possible—before the macro package gets distributed widely!

I. Branching Mechanisms.

The only branching mechanism provided by TEX is

```
\if ⟨char1⟩⟨char2⟩{⟨true text⟩}
\else{⟨false text⟩}
```

and its relatives. Unfortunately, there are certain peculiarities of \if...\else that require special care.

(a) An \if...\else construction is processed in TEX's "digestive system", rather than in its "mouth". Suppose, for example, that we have two control sequences \csa#1 and \csb#1#2, taking one and two arguments, respectively, and a control sequence \flag that is sometimes defined to be T and sometimes defined to be F. We would like to define \cs to be \csa if \flag is T, and \csb if \flag is F [the argument(s) for \cs will simply be whatever comes next in the input text]. If we try to define

```
\def\cs{\if T\flag{\csa}\else{\csb}}
```

then a use of \cs will produce the error message

```
! Argument of \csa has an extra }
```

because TEX sees the } as soon as it looks for the argument after \csa or \csb. The solution to this problem is to define

```
\def\cs{\if T\flag{\gdef\result{\csa}}
        \else{\gdef\result{\csb}}
        \result}
```

A similar problem arises in the following situation. Suppose that we have two different macro files, mfile.1 and mfile.2, and the value of \flag is supposed to determine which file to use (such a scheme is useful for saving TEX memory space). A definition like

```
\def\cs{\if T\flag{\input mfile.1}
        \else{\input mfile.2}...}
```

gives a different error message:

```
! Input page ended on nesting level 1
```

but the basic problem (and the solution) is exactly the same.

If we make the definition

```
\def\If#1\then#2\else#3{\if#1
     {\gdef\result{#2}}
     \else{\gdef\result{#3}}\result}
```

then we can safely use constructions like

```
\If T\flag\then...\else{...}
```

The token \then is made part of the syntax of \If so that we can have constructions like \If \cs a\cs b\then..., where \cs#1 is a control sequence with one argument.

(b) Although ⟨char$_1$⟩ and ⟨char$_2$⟩ may be specified by control sequences like \flag, which TEX expands out, they cannot involve \if...\else again. Suppose, for example, that we have already defined

```
\def\ab#1{\if#1a{T}
          \else{\if#1b{T}\else{F}}}
```

so that \ab#1 is T if #1 is a or b, and F otherwise. We would now like to define \cs#1 to be ⟨true text⟩ if #1 is a or b, and ⟨false text⟩ otherwise. We cannot conveniently define

```
\def\cs#1{\if T\ab#1{⟨true text⟩}
          \else{⟨false text⟩}}
```

If we do this, then the input \cs x will become

```
\if T\if xa{T}\else{\if xb{T}
   \else{F}}{⟨true text⟩}\else{⟨false text⟩}
```

which causes TEX to try to compare T with \if, giving an error message.

Of course, the test for #1 being a or b could be made part of the definition of \cs, but the following scheme is far more advantageous:

```
\def\ab#1{\if#1a{\gdef\Ab{T}}
          \else{\if#1b{\gdef\Ab{T}}
               \else{\gdef\Ab{F}}}}
\def\cs#1{\ab#1
          \if T\Ab{⟨true text⟩}
          \else{⟨false text⟩}}
```

(c) In an \if...\else construction, ⟨char$_1$⟩ and ⟨char$_2$⟩ are supposed to be single characters (of type

0 to 12), or defined control sequences, possibly with arguments, that expand out to a character. So we can't use an `\if...\else` construction in a situation where we don't know for sure what the next input text will be. Suppose, for example, that `\cs#1` is supposed to be ⟨true text⟩ if #1 is a comma, but ⟨false text⟩ otherwise. If we define

```
\def\cs#1{\if#1,{⟨true text⟩}
              \else{⟨false text⟩}}
```

there is always the possibility that our input text will contain

```
\cs ...
```

where ... is a token that can't be used with `\if`, or even worse, a group {...}, which might produce total chaos. In order to deal with this we will use several tricks, which are also useful in other situations.

II. Basic Kludges

Consider the definitions

```
\def\false#1{\gdef\ans{F}}
\def\tricka{A}
\def\trickb{B}
\def\trickc#1{\if#1A
        {\gdef\result{\false}}
        \else{\gdef\result{\gdef\ans{T}}}
        \result}
\def\empty#1#2\tricka{\trickc}
```

The control sequence `\trickc` will be used only in situations where the `\if` is safe. In fact, `\trickc` will arise only from an occurrence of `\empty`, and the control sequences `\tricka`, `\trickb`, `\trickc` and `\empty` will be used only in the construction

```
\empty...\tricka\tricka\trickb
```

Here ... will be some input text, with perhaps a few special $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX control sequences thrown in, but ... will never involve `\tricka` (remember that `\tricka` is really `\tricka|`, so it can't appear in a user's file).

We have to consider two possibilities for ... in order to determine the result of this construction. Suppose first that ... is not empty. Then argument #1 for `\empty` will be the first token or group of ... and argument #2 will be whatever remains (if anything). Hence

```
\empty...\tricka\tricka\trickb →
        → \trickc\tricka\trickb →
        → \false\trickb → \gdef\ans{F}
```

But suppose that ... is empty, so that we have

```
\empty\tricka\tricka\trickb
```

Note that argument #1 for `\empty` must be *non-empty*, since it is not followed by a token in the definition of `\empty`. So in the present case argument #1 for `\empty` will be the first `\tricka`. Consequently, the *second* `\tricka` will play the role of the token `\tricka` in the definition of `\empty` (and argument #2 will be empty). Thus

```
\empty\tricka\tricka\trickb →
        → \trickc\trickb → \gdef\ans{T}
```

In other words,

`\empty...\tricka\tricka\trickb` defines `\ans` to be T if ... is empty, and F otherwise.

There would appear to be one exception to this rule: If ... is a blank space, or a sequence of blank spaces, then `\ans` will still be defined to be T, since spaces after the control sequence `\empty` are ignored. But in practice ... will always be an argument from some other macro, and in this case the exception does not arise. Suppose, for example, that we define

```
\def\try#1{\empty#1\tricka\tricka\trickb}
```

so that `\try{#1}` will test whether #1 is empty or not. If we give TEX the input

```
\try{ }
```

then the braces will be removed from { }, so this will be translated into

```
\empty␣\tricka\tricka\trickb
```

But in this situation the space indicated by ␣ is *not* ignored, so `\ans` will be defined to be F.

We might have arranged for the result of the combination `\empty...\tricka\tricka\trickb` simply to be T or F, rather than defining `\ans` to be T or F. But if we did this, a construction like

```
\if T\empty#1\tricka\tricka\trickb{...}
\else{...}
```

wouldn't work, because TEX would think that we were trying to compare T with the result of `\empty#1\tricka`.

The following variant of `\empty` is also useful:

```
\def\emptygp#1\endd
        {\empty#1\tricka\tricka\trickb}
```

Then

`\emptygp...\endd` defines `\ans` to be T if ... is empty or {}, and F otherwise.

It will be convenient to use the same flag `\ans` for the result of several of our macros. This won't produce problems if we ever have to perform two tests on two different arguments: we can always first use `\empty`, then `\let\firstans=\ans`, then use `\emptygp`, etc.

We also want to be able to check if ... is a single token or group, rather than a string of several tokens or groups. One idea is to consider `\single...\endd` where `\single#1#2\endd` checks whether `#2` is empty:

```
\def\single#1#2\endd
        {\empty#2\tricka\tricka\trickb}
```

This won't quite work, since ... might be something like ⟨token⟩`{}`; in this case `#2` appearing in `\empty#2\tricka\tricka\trickb` will still be empty, since TeX removes an outer set of braces from any argument. So to be on the safe side, we add some extraneous character after ... and let `\single#1#2#3\endd` check if `#3` is empty:

```
\def\single#1#2#3\endd
        {\empty#3\tricka\tricka\trickb}
```

Then

`\single...*\endd` defines `\ans` to be `T` if ... is a single token or group, and `F` otherwise.

Before using `\single...*\endd` it is essential to check that ... isn't empty. Otherwise there will be problems, because of the very considerations that made `\empty` work. (An `\empty` check could be incorporated into the definition of `\single`, but whenever AMS-TeX uses `\single` a separate check has to be made anyway.)

As in the case of `\empty`, a space may legitimately occur as argument `#1`. For example, if we define

```
\def\try#1{\single#1*\endd}
```

then `\try{ X}` defines `\ans` to be `F`. (But `\try{␣␣}` defines `\ans` to be `T`—the second space never even gets read by TeX.)

It is now fairly easy to check whether an argument `#1` (which might *a priori* be an arbitrary token or even a group) is a comma. The basic idea is to define

```
\def\check#1,#2\endd
        {\empty#1\tricka\tricka\trickb}
```

and then define

```
\def\comma#1{\check#1,\endd}
```

so that `\comma{#1}` will define `\ans` to be `T` if `#1` is a comma, and `F` otherwise. This won't quite work for the following reasons:

(i) If `#1` is `{}` or `{{}}`, then `\comma{#1}` is `\comma{}` or `\comma{{}}`. This means that the `#1` appearing in `\check#1,\endd` is empty or `{}`, and thus the `#1` in `\empty#1\tricka\tricka\trickb` is empty.

(ii) If `#1` is a group `{,...}` that happens to begin with a comma, then `\comma{#1}` will define `\ans` to be `T`, whereas we want it to be `F` (this, admittedly, is a matter of taste).

So we will use `\emptygp` and `\single` to check on these possibilities:

```
\def\comma#1{\emptygp#1\endd
        \if T\ans{\gdef\ans{F}}
        \else{\single#1*\endd
            \if F\ans{}
            \else{\check#1,\endd}}}
```

Then

`\comma{#1}` defines `\ans` to be `T` if `#1` is `,` or `{,}`, and `F` otherwise.

(The inability to distinguish between `,` and `{,}` is a minor problem that seems insurmountable.)

AMS-TeX needs many such checks, so they are all made in terms of one generalized check. For example, `\comma` is actually defined by

```
\def\comma#1{\compare*,{#1}}
```

where `\compare` is defined as

```
\def\compare*#1#2{\def\check##1#1##2\endd
        {\empty##1\tricka\tricka\trickb}
    \emptygp#2\endd
    \if T\ans{\gdef\ans{F}}
    \else{\single#2*\endd
        \if F\ans{}
        \else{\check#2#1\endd}}}
```

The `*` was made part of the syntax for `\compare` to allow `\def\space#1{\compare*␣{#1}}`.

III. Saving Braces

We have just seen that there can sometimes be problems when braces are removed from the argument of a control sequence. Actually, the problem can be much more critical. For example, the AMS-TeX control sequence `\dots#1` first examines `#1` to determine what sort of dots and spacing are needed, and then produces these dots, followed by `#1` (and the remaining input). The removal of braces would be a minor annoyance if `#1` were something like `{+}`, where the braces are meant to make the `+` into a `\mathord` (something that AMS-TeX users aren't supposed to know about anyway). But it could be a major catastrophe if `#1` were something like `{a\frac b}`. To handle such problems we define

```
\def\braced#1{\empty
        #1\tricka\tricka\trickb
    \if T\ans{\gdef\Braced{{#1}}}
    \else{\single#1*\endd
        \if F\ans{\gdef\Braced{{#1}}}
        \else{\gdef\Braced{#1}}}}
```

In other words, `\braced` puts back a pair of braces if `#1` is `{}` or a group with more than one token or group in it. Thus, `\braced{#1}` defines `\Braced` to be `#1` except when `#1` is `{⟨token⟩}` or `{{...}}`, in which case the outer set of braces is removed. So, aside from the unavoidable `{⟨token⟩}` case, `\Braced` has enough braces to give the same result as `#1`.

IV. Recursions

There are several ways of handling recursions, all of which are used at some point in AMS-TEX.

(a) Suppose that we want to define `\qms #1` so that

```
\qms 1     is ?
\qms 2     is ??
\qms 3     is ???
  ......
\qms {10}  is ??????????
    etc.
```

We can define

```
\def\qms#1{\setcount1 #1
      \def\string
          {\ifpos1{\advcount1 by -1
                \gdef\newstring{?\string}}
           \else{\gdef\newstring{}}
         \newstring}  %end of \def\string
      \string}
```

This only appears to violate the rule not to define a control sequence in terms of itself: An occurrence of `\string` may produce `\gdef\newstring{?\string}`, but TEX will simply record this definition, and not try to expand out the `\string` that occurs in it until `\newstring` is expanded, at which time an `\if` test is made, which produces a new `\gdef`.

`\newstring` should be defined as `?\string` rather than as `\string?` to keep TEX's internal "input stack" from growing unboundedly.

(b) Suppose that we have some input of the form

$\langle$`string`$_1\rangle$`,`$\langle$`string`$_2\rangle$`,...,`$\langle$`string`$_n\rangle$

with strings separated by some character, like a comma, and we want the control sequence `\operate` to perform some operation on each string. For example, we might want to replace each $\langle$`string`$_i\rangle$ by A$\langle$`string`$_i\rangle$Z, so that

`\operate{`$\langle$`string`$_1\rangle$`,`$\langle$`string`$_2\rangle$`,...,`$\langle$`string`$_n\rangle$`}`

will produce

A$\langle$`string`$_1\rangle$ZA$\langle$`string`$_2\rangle$Z...A$\langle$`string`$_n\rangle$Z

(We might also want to consider the case where there are no separators, so that an A and a Z will be inserted before and after each token or group.) We will use the token `\marker` as a "marker" to tell us when our recursion is over, so we define

```
\def\ismarker#1{\compare*\marker{#1}}
```

Now the basic idea is to define

```
\def\op#1,#2{\ismarker{#2}
            \if T\ans{A#1Z\gdef\nextop{}}
            \else{A#1Z\gdef\nextop{\op#2}}
            \nextop}
\def\operate#1{\op#1,\marker}
```

(omitting the commas in these definitions for the case of no separators).

Unfortunately this won't work, because there are problems concerned with the removal of braces. Each time `\op#1,#2` is used, argument `#2` is the first token or *group* following the comma, and if it is a group the braces will be removed. The removal of braces again causes problems if `#1` is something like `{a\frac b}`, and also if `#2` is something like `{(a,b)}`, where the braces are meant to "hide" the comma. We could use `\braced` here, but it isn't quite foolproof, since `#2` might be a "hidden" comma `{,}`, which `\braced` can't distinguish from an ordinary comma. Moreover, `\braced` can't help us with argument `#1`. Although this argument is usually a sequence, terminated by a comma, it just might happen to be a single group followed by a comma, and there is no way of distinguishing between these possibilities once argument `#1` has been read.

In the cases where AMS-TEX uses a recursive scheme of this sort, the particular circumstances, or simple tricks, usually circumvent these problems. The following definition illustrates a general scheme that will always work:

```
\def\kill#1{}
\def\op#1,#2\endd{\ismarker{#2}
       \if T\ans{A\kill#1Z\gdef\nextop{}}
       \else{A\kill#1Z\gdef\nextop
                         {\op*#2\endd}}
       \nextop}
\def\operate#1{\op*#1,\marker\endd}
```

Notice that each time `\op#1,#2\endd` is used, argument `#1` now begins with `*` (which is removed by `\kill`), so it can't possibly be a group. And argument `#2` is always the remaining input, terminated by `\marker`, so it can't be a group either.

(c) A recursive procedure can be used to count the number of commas in a string:

```
\def\cm#1,#2\endd{\ismarker{#2}
          \if T\ans{\gdef\nextcm{}}
          \else{\advcount1
                \gdef\nextcm{\cm#2\endd}}
          \nextcm}
\def\countcommas#1{\setcount1 0
                  \cm#1,\marker\endd}
```

Then

```
\countcommas{#1}
makes the value of \count1 be the
number of commas in #1.
```

The `\endd` trick is used to handle "hidden" commas, but the `*` trick isn't needed, since we don't care what `\cm` does to `#1`.

(d) If we do `\countcommas{#1}`, then `\ifpos1` will tell us whether `#1` contains at least one comma. But it is preferable to use the following scheme, which doesn't involve any counters, and which stops as soon as the first comma is found:

```
\def\cm#1,#2{\ismarker{#2}
          \if T\ans{\gdef\nextcm{}}
          \else{\gdef\Hascomma{T}
                \gdef\nextcm##1\marker{}}
          \nextcm}
\def\hascomma#1{\gdef\Hascomma{F}
                \cm#1,\marker}
```

(e) Suppose we want to perform the operation in part (b) on some input of the form

$$\langle\texttt{string}_1\rangle\backslash\backslash\langle\texttt{string}_2\rangle\backslash\backslash\ldots\backslash\backslash\langle\texttt{string}_n\rangle$$

where the separator is the control sequence `\\` (which is never used in isolation, and is initially defined by `\def\\{}`). We could use exactly the same scheme, replacing `\def\op#1,#2\endd` by `\def\op#1\\#2\endd`. But we can also take advantage of the fact that the separator is a control sequence to obtain a definition that is both more elegant and more efficient:

```
\def\op#1\\{A\kill#1Z\\}
\def\operate#1{\def\\{\op*}
               \op*#1\def\op{\kill}\\
```

The `\def\op{}` needs to be replaced by `\gdef\op{}` if `\op` puts things inside braces; in this case, the original definition of `\op` should be made part of the definition of `\operate`.

There might appear to be possible confusion if some $\langle\texttt{string}_i\rangle$ contains `\\` within a group `{...\\...}`. In $\mathcal{AMS}$-TEX this occurs only in constructions like

```
{\align...\\...\endalign}
```

where `\\` is temporarily re-defined anyway.

V. Searching For Strings

TEX's method of determining where an argument in a definition ends has the following peculiar feature. Suppose we define

```
\def\cs#1ab#2{...}
```

Then the first argument is the smallest (possibly empty) token or group that is followed by a, *not* the smallest group that is followed by ab. So the input

```
\cs xayabc
```

gives the error message

```
! Use of \cs does not match its definition.
```

So if we want to know whether ab occurs in some string we can't simply replace the comma by ab in the method of part IV(d), because an a might occur alone. Instead we have to do something like the following:

```
\def\isb#1{\compare*b{#1}}
\def\finda#1a#2#3\endd{\ismarker{#2}
     \if T\ans{\gdef\nextfinda{}}
     \else{\isb{#2}
           \if T\ans{\gdef\Hasab{T}
                     \gdef\nextfinda{}}
           \else{\gdef\nextfinda
                       {\finda#2#3\endd}}}
     \nextfinda}
\def\hasab#1{\gdef\Hasab{F}
             \finda#1a\marker\endd}
```

* * * * * * * * * * *

Problems

* * * * * * * * * * *

The first formatting problems posed in this column come from the videotaped TEXarcana Class taught by Don Knuth last March. Solutions will be presented in the next issue. Readers with working TEX systems are encouraged to attempt solutions to these problems, in order to better appreciate the problems and their solutions.

Lynne A. Price

Problem no. 1:

Type:

```
\vskip 12pt
\noindent\hide{--}Allan Temko

\vskip 2pt
\noindent Architecture Critic
```

To get:

–Allan Temko
Architecture Critic