```
 end

/* Cleanup. */
'k 'texauxfile
'qquit'

/* File the answer file. */
'k 'answerfilename
'file '
/* Move back to where we were when the macro was called. */
'k 'fileid.1
'set point .w off'
'locate .z'
'set point .z off'
'set msgmode on'   /* Matches the "off" above. */
'msg Answers appended to '||answerfilename||'.'
```

◇ Jim Hefferon
Mathematics
St. Michael's College
Colchester, VT 05439
BITnet: **hefferon@smcvax**

Oral TEX

Victor Eijkhout

TEX knows two sorts of activity: those actions that can be classified under 'execution', and those that fall under 'expansion'. The first class comprises everything that gives a typeset result, or that alters the internal state of TEX. Examples of this are control sequences such as `\vskip`, macro definitions, and all assignments.

Expansion activities are those that are performed by what is called the mouth of TEX. The most obvious example is macro expansion, but the command `\the` and evaluation of conditionals are also examples. The full list can be found on pages 212–215 of the TEXbook [1].

In this article I will give two examples of complicated macros that function completely by expansion. Some fancy macro argument delimiting occurs, and there are lots of applications of various conditionals. For a better understanding of these I will start off with a short section on the expansion of conditionals.

About conditionals

For many purposes one may picture TEX's conditionals as functioning like conditionals in any other programming language. Every once in a while, however, it becomes apparent that TEX is a macro processor, absorbing a stream of tokens, and that conditionals consist of nothing more than just that: tokens.

Consider the following example:

```
\def\bold#1{{\bf #1}}
\def\slant#1{{\sl #1}}
\ifsomething \bold \else
             \slant \fi {word}
```

If the 'something' condition is true, the whole `\if ... \else ... \fi {word}` sequence is *not* replaced by `\bold {word}`; instead TEX will start processing the 'true' part of the conditional. It expands the `\bold` macro, and gives it the first token in the stream as argument. Thus the argument taken will be `\else`. TEX will only make a mental note that when it first encounters – more precisely: expands – an `\else` it will skip everything up to and including the first `\fi`[1].

[1] The reader may enjoy figuring out why in spite of the apparent accident in this example the 'word' will still be bold, and why TEX will report that 'end occurred inside a group at level 1' at the end of the job.

For this sort of problem there are (at least) two solutions. One solution is:

```
\ifsomething \let\next=\bold
   \else \let\next=\slant \fi
   \next {word}
```

Note that this uses more than just the mouth of TEX, because `\let` statements are executed, not expanded.

Second solution:

```
\ifsomething \expandafter\bold
   \else \expandafter\slant \fi {word}
```

The `\expandafter` commands will let TEX find the delimiting tokens of the conditional before it starts expanding for instance `\bold`. When this command is finally expanded, the irrelevant parts of the conditional will have been removed.

The reader may not see the point of this second solution at first, and indeed, the first solution is more natural in a sense. However, there is a very important advantage to the second. Primitive conditionals of TEX are fully expanded, for instance inside an `\edef`. Example:

```
\edef\savelastskip
    {\ifhmode \hskip \else
              \vskip \fi \the\lastskip}
```

will expand to

```
macro: -> \hskip ...
```

or

```
macro: -> \vskip ...
```

depending on the mode. If you are implementing a test that should function in a context where there is only expansion `\let` cannot be used, so a number of `\expandafter` commands will probably be needed.

Application 1: string comparison

Suppose we would like to have a macro that tests for equality of two strings, and it should be useable as if it were a conditional:

```
\ifsamestring{some}{other}
   \message{yes!}\else
   \message{no!}\fi
```

A simple construction exists for this:

```
\def\ifsamestring
   #1#2{\def\testa{#1}\def\testb{#2}%
       \ifx\testa\testb}
```

however, this suffers from the objection mentioned above: it uses more than just the expansion performed in TEX's mouth. Thus, the following call

```
\message{\ifsamestring
   {some}{other}yes\else no\fi!}
```

gives a, probably, somewhat unexpected result:

```
! Undefined control sequence.
\ifsamestring #1#2->\def \testa
                                ...
```

Solutions using only expansion are possible, but they are more complicated. The reader may want to try solving this before looking at the solution below. Keep in mind that we want something that behaves like a conditional: the final result should be allowed to be followed by

```
 ... \else ... \fi
```

The solution given here is not the only possible one: variations may exist. However, there is probably only one basic principle, which is to compare the strings one character at a time.

Here is the first part:

```
\def\ifsamestring
    #1#2{\ifallchars#1$\are#2$\same}
```

The strings are terminated by a dollar character, which we suppose not to appear in the string.

Next the routine `\ifallchars` will be used recursively. At first it tests if either of the two strings has run out. If some incarnation of this routine finds that both strings are empty the initial strings must have been equal, if exactly one is empty the initial strings were of unequal length, thus unequal; if neither is empty another routine should check if their leading characters are the same, and if so, do a recursive call to `\ifallchars` to see if the rest is also the same.

```
\def\ifallchars#1#2\are#3#4\same
   {\if#1$\if#3$\say{true}%
          \else \say{false}\fi
    \else \if#1#3\ifrest#2\same#4\else
                 \say{false}\fi\fi}
```

The `\say` macro is something of a trick; we'll get to that. Let's first consider the last clause: the test `\if#1#3` checks if the leading characters of the strings are equal; if so, the remainder should be tested for string equality.

In the previous section I showed that a `\fi` is just a token standing in the input stream. Standing behind the call to `\ifrest` there are two such tokens, and somehow they are to be removed. Unfortunately the `\expandafter` method of the previous section cannot be used here, as there may be an indefinite number of tokens between the `\ifrest` and the `\fi` tokens.

One solution here is to let the final argument of `\ifrest` be delimited by the whole closing sequence:

```
\def\ifrest#1\same#2\else#3\fi\fi
   {\fi\fi \ifallchars#1\are#2\same}
```

A trick if ever there was one. When the `\if#1#3` test turns out false the `\ifrest` call is skipped, and the `\fi\fi` sequence delimits both conditionals that are open at that moment. When `\if#1#3` is true, everything up to and including `\fi\fi` is scooped up as part of the parameter text. The delimiting `\fi\fi` sequence is not expanded in this process, so the conditionals must still be closed; this is done by the `\fi\fi` sequence with which the replacement text starts. Thus this construction effectively lifts the relevant part of the macro outside the boundaries of the conditional.

Now for the `\say` macro. What we want to accomplish by it is this: the call `\say{true}` should put the primitive `\iftrue` outside all conditionals that are active at the moment, and similarly for `\say{false}`. The implementation of `\say` given here is not very general: it uses the fact that all three calls are nested two conditionals deep, and that the final boundary is the `\fi\fi` sequence.

```
\def\say#1#2\fi\fi
   {\fi\fi\csname if#1\endcsname}
```

All tokens in between the first argument of `\say` and the delimiting `\fi\fi` are lumped together in `#2` and they are never used. The reason that `\say` is necessary at all, is that TeX would misinterpret the occurrence of `\iftrue` and `\iffalse` in clauses that are skipped.

There is a, maybe somewhat surprising, second possible implementation of `\say`. Inside a `\csname ... \endcsname` sequence all expandable tokens are expanded, and in particular `\expandafter`. This means that one may alter the state of the input stream after `\endcsname` by putting `\expandafter` directly in front of it. I leave it to the reader to figure out what are the exact mechanisms of this second implementation:

```
\def\say#1{\csname if#1\expandafter
    \expandafter\expandafter\endcsname}
```

Note that the calls to `\say` always occur directly in front of an `\else` or a `\fi`.

Once the reader understands this trick, the following routine for alphabetical comparison of two strings will not present insurmountable problems.

```
\let\xp=\expandafter
\def\ifbefore
    #1#2{\ifallchars#1$\are#2$\before}
\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
       \ifrest#2\before#4\fi\fi\fi}
\def\ifrest#1\before#2\fi\fi\fi
   {\fi\fi\fi
    \ifallchars#1\are#2\before}
\def\say#1{\csname if#1\endcsname}
```

Lexicographic comparison is done here by numerical comparison of character codes. The `\say` macro now occurs on three different levels, so the number of `\expandafter` commands needed to remove various amounts of `\else` and `\fi` tokens is 1, 3, and 7. The first two clauses of the test take care of the case where strings are of different lengths.

A comment about the principle underlying these macros. Every step replaces one `\if...` command by another, until finally only `\iftrue` or `\iffalse` results. All the magic with `\expandafter` and delimiting with `\fi` is necessary, because we can't deliver these final conditionals as a result of other conditionals. However, we can let TeX deliver some tokens that give a true or false test. The `\ifsamestring` test can for instance be implemented as[2]

```
\def\saytrue{0=0 } \def\sayfalse{0=1 }
\def\ifsamestring#1#2%
   {\ifnum \allchars#1$\are#2$\same}
\def\allchars#1#2\are#3#4\same
 {\if#1$\if#3$\saytrue\else\sayfalse\fi
  \else \if#1#3\allchars#2\are#4\same
        \else\sayfalse
        \fi
  \fi
 }
```

Now there is an outer `\ifnum` test, and TeX will expand tokens after that test until two numbers and a relation remain.

Application 2: implementing the Lisp backquote macro

Coming (partly) from a Lisp programming background, I can't help being reminded of the Lisp backquote macro when using TeX's `\edef`. The backquote macro [2] is something like a reverse `\edef`: it expands nothing, unless you explicity order it to. And, relishing a good TeX hack, I was wondering if I could write something like that in TeX. The answer turned out to be: yes. But it wasn't particularly easy.

Another incentive than sheer curiosity was the fact that, in a certain application, I was writing things like

```
\edef\act{\noexpand\a{\noexpand\b
```

[2] This implementation was suggested to me by Marc van Leeuwen.

```
    {\noexpand\c\noexpand\d{\e}}}}
\act
```

and I was getting tired of typing all the `\noexpand` commands. I wanted to tell TEX: 'expand this', instead of having to point out everything that shouldn't be expanded.

My solution to this problem takes the form of a 'backquoting definition' `\bdef`, which, by the way, defines only macros without parameters. The basic principle is to traverse the replacement text, to reproduce everything in it, but to expand everything that has `\expand` in front of it.

The macro `\bdef` is just an `\edef` in disguise: it appends a terminator and installs a routine that will eat its way through the argument list.

```
\def\bdef#1#2{\edef#1{\TakeItem#2\Stop}}
```

As before, I save myself a lot of typing by putting

```
\let\xp=\expandafter
```

For this macro I wanted to allow conditionals to be part of the argument. This meant being careful: sequences such as

```
\ifsomething #1 \else #2 \fi
```

are completely misunderstood by TEX if either of the arguments is some `\if...`, `\else`, or `\fi`. Therefore I allowed macro arguments to appear only in the tests themselves, and outside conditionals.

The first test is easy: if we have found the terminator we can stop.

```
\def\TakeItem#1%
   {\ifx\Stop#1\xp\StopTesting \else
        \xp\GroupTest\fi #1\stop\Stop}
\def\StopTesting#1\stop\Stop{}
\def\Stop{1}\def\stop{0}
```

If not, we have now one argument. This can be a single token, or it can be a group that was enclosed in `{...}`. We have to test for this distinction.

Note how the argument occurs only in the test, and is then reproduced outside the conditional for the benefit of the `\GroupTest` macro. The other macro, `\StopTesting` doesn't need the argument, so it has to remove it. This slight overhead (also in most of the following macros) ensures that we will not have conditional tokens inside a conditional.

Now the macro `\GroupTest` receives as argument a string of tokens, delimited by `\stop\Stop`. If the argument of `\TakeItem` was a single token, `\GroupTest` will find `\stop` as its second argument, and it will invoke a routine that handles single tokens; otherwise the argument of `\TakeItem` must have been a group, and it will invoke a routine that handles groups.

```
\def\GroupTest#1#2#3\Stop
   {\ifx#2\stop \xp\TakeToken
    \else \xp\TakeGroup\fi #1#2#3\Stop}
```

Single tokens can be `\stop` in which case the end of a group has been reached, and intake of tokens on this level can stop; otherwise it is a token that must be expanded or must be reproduced without expansion.

```
\def\TakeToken#1\stop\Stop
   {\ifx#1\stop
    \xp\RemoveToken \else
    \xp\MaybeExpand \fi #1}
\def\RemoveToken#1{}%get rid of a \stop
```

Groups are handled by putting a left brace (recall that braces around macro arguments are removed), tackling in succession all tokens of the group, putting a right brace, and continuing with the items after the group.

```
\def\TakeGroup#1\Stop
    {\leftbrace\TakeItem#1\rightbrace
     \TakeItem}
```

In between braces there is now a sequence delimited by `\stop`.

The following macros yield a left and right brace respectively;

```
\def\leftbrace{\iftrue{\else}\fi}
\def\rightbrace{\iffalse{\else}\fi}
```

which is based on the fact that the nesting structures of groups and conditionals are independent.

Now for the single tokens. If the token is `\expand` we have to expand the token following it, otherwise we reproduce the token without expansion.

```
\def\MaybeExpand#1{\ifx#1\expand
    \else \xp\id \fi #1}
\def\id#1{\noexpand#1\TakeItem}
```

If parameter 1 is `\expand` we let it stand; this has the effect of applying the macro `\expand` (see below) to what follows. Otherwise we apply `\id`, which has the effect of simply reproducing its argument; however, as we are still in the context of an `\edef`, this argument has to be prefixed with `\noexpand`.

Expansion of a token is a tricky activity. Merely reproducing a token will cause it to be fully expanded, as we are still inside an `\edef`. However, once we abandon control, we cannot get it back, so we will have to do all expansion ourselves.

At first I had here

```
\def\expand#1{\expandafter\TakeItem#1}
```

which is in the spirit of the Lisp backquote macro. However, it will not expand completely the way it is done inside an `\edef`. The solution I found to this problem necessitated me to put a delimiter after the string to be expanded, instead of having it only prefixed. Tokens in between

```
\expand ... \endexpand
```

delimiters will be fully expanded. I don't believe solutions are possible without this closing delimiter.

As `\expandafter` is the only mechanism by which the user can explicitly force expansion, I arrived at the following idea. If a token is to be expanded, store a copy for comparison, hit the original over the head with an `\expandafter`, see if it still moves (that is, if it is not equal to the comparison copy), and if so, repeat this algorithm. Crude but effective[3].

First the simple part: if we have found the closing `\endexpand` delimiter, we remove it and go on absorbing tokens after it.

```
\def\expand#1{\ifx#1\endexpand
    \xp\TakeFirst\xp\TakeItem
    \else \xp\fullexpand \fi #1}
\def\endexpand{2}%just to have it defined
```

Otherwise we compare the token to its expansion:

```
\def\fullexpand#1%
   {\xp\maybeexpandfurther\xp#1#1}
```

Comparison can be done by `\ifx`, which is able to handle both characters and control sequences.

```
\def\maybeexpandfurther#1#2%
   {\ifx#1#2%
    \xp\TakeFirstAndExpandOn \else
    \xp\TakeFirst\xp\expand \fi #1#2}
\def\TakeFirstAndExpandOn#1#2{#1\expand}
```

If the two parameters are the same we stop; otherwise we again `\expand`. Note that the call to `\TakeFirst` has the effect of removing the `#1` after the conditional; the `#2` is the expanded token, and it should be expanded further.

Now that we have all pieces together we can perform a small test: I put some conditionals in the test to make sure it would be hard.

```
\def\a#1#2{\ifnum\count0>0
    \twice{#1}\else \thrice{#2}\fi}
\def\twice#1{#1#1} \def\thrice#1{#1#1#1}
\count0=0
\bdef\tmp{\a{bc}\fi\iftrue\b
 {\expand\a{fg}{hj}\endexpand}z\else}
\show\tmp
```

which gives

```
> \tmp=macro:
-> \a {bc}\fi \iftrue \b {hjhjhj}z\else .
```

and

```
\count0=1
\bdef\tmp{\a{bc}\fi\iffalse\b
 {\expand\a{fg}{hj}\endexpand}z\else}
\show\tmp
```

which gives

```
> \tmp=macro:
-> \a {bc}\fi \iffalse \b {fgfg}z\else .
```

The above implementation has a slight shortcoming, as it cannot distinguish between a single token and that same token with braces around it[4]. Both

```
\bdef\tmp{\a{b}}
```

and

```
\bdef\tmp{\a b}
```

give

```
>\tmp=macro:
-> \a b
```

Of course, shortcoming or not, this whole section is of rather academic value: `\bdef` is so much slower than `\edef` in execution that I've reconciled myself with writing lots of `\noexpand` tokens. But I do hope that these farfetched examples give inspiration to macro writers. Because TEX's mouth does lend itself to useful purposes [3, 4]. And to loads of fun.

References

[1] Donald Knuth, *The TEXbook*, Addison-Wesley Publishing Company, 1984.

[2] Guy L. Steele jr., Common Lisp, the language, Digital Press 1990.

[3] Alan Jeffrey, Lists in TEX's mouth, *TUGboat*, 11(1990), no. 2, 237–245.

[4] Sonja Maus, An expansion power lemma, *TUGboat*, 12(1991), no. 2, 277.

◇ Victor Eijkhout
Center for Supercomputing Research and Development
University of Illinois
305 Talbot Laboratory
104 South Wright Street
Urbana, Illinois 61801-2932, USA
eijkhout@csrd.uiuc.edu

[3] Well ... Cases like `\ifnum ... \ifnum ... \fi\fi`, where expansion of one token yields the same token, go wrong.

[4] Also, it cannot cope with macros that expand to a space token or to nothing. The second objection can probably be repaired; the first is inherent to TEX's parameter mechanism.