The moral: Always surround nested loops with curly braces.

Fontdimens and Physical Fonts

TeX associates a series of parameters with each font, and looks for these values in the `tfm` file. These font dimensions are accessible to a TeX user via the `\fontdimen` command. (Their significance is summarized in tables on pages 433 and 447 of *The TeXbook.*) Mr. Khodulev has uncovered some puzzling behavior when he tries to alter the `\fontdimens` for his own uses.

For the sake of concreteness, we will use `\fontdimen2`, which specifies the normal interword space for a font. Suppose you wanted to increase the interword space for a certain font in special places in the document. You might try (as he apparently did) something like the following.

```
\font\rm=cmr10
\font\specrm=cmr10
\fontdimen2\specrm=9.99pt
```

You might expect that when you typeset using `\rm`, you get the normal interword spacing (3.33 pt), while you would extra large spaces only when using `\specrm`. In fact, after the `\fontdimen` declaration above, any `\fontname` tied to the physical font `cmr10` has its `\fontdimen` changed. As if to add insult to injury, you cannot attempt to surround changes to `\fontdimen` within a group, since `\fontdimen` assignments are *always* global.

The following lines of code present one way of resolving the problem. The font definitions are encumbered with longer names than usual, but the actual of mechanics of changing fonts are relegated to macros with names that closely resemble normal font calls. These macros have been designed to be used so the user thinks they are font calls, and the rare appearance of `\aftergroup` helps make this syntax possible.

```
%% First, fonts.
\font\roman=cmr10
\font\specroman=cmr10
%% Next, the special registers
\newdimen\savedvalue
 \savedvalue=\fontdimen2\roman
\newdimen\specialvalue
 \specialvalue=9.99pt
%% Finally, definitions.
\def\rm{%
 \fontdimen2\roman=\savedvalue }
\def\specrm{%
 \aftergroup\restoredimen
 \fontdimen2\specroman=\specialvalue
 \specroman }
\def\restoredimen{%
 \fontdimen2\roman=\savedvalue }
```

Mr. Khodulev did not specify his need in any more detail, so these macros should be revised as necessary. With these macros and definitions in force, the source text

```
\rm Here is some text.
{\specrm Here is some spaced out text.}
Here is more text, hopefully
back to normal.

\rm Here is more text.
\specrm Here is some spaced out text.
\rm Text is back to normal.
```

produces

Here is some text. Here is some spaced out text. Here is more text, hopefully back to normal.

Here is more text. Here is some spaced out text. Text is back to normal.

Bibliography

[1] Hoenig, Alan, "Line-Oriented Layout with TeX," in *TeX: Applications, Uses, Methods*, ed. Malcolm Clark. London: Ellis Horwood (1990).

◇ Alan Hoenig
17 Bay Ave.
Huntington, NY 11743 USA
(516) 385-0736
`ajhjj@cunyvm.bitnet`

Tutorials

The `\if`, `\ifx` and `\ifcat` Comparisons

David Salomon

Large, small, long, short, high, low, wide, narrow, light, dark, bright, gloomy, and everything of the kind which philosophers term accidental, because they may or may not be present in things,—all these are such as to be known only by comparison.

— Leon Battista Alberti

A general note: Square brackets are used throughout this article to refer to the TeXbook. Thus [209] refers to page 209, and [Ex. 7.7] to exercise 7.7, in the book. Advanced readers are referred to the actual `WEB` code by the notation [§495].

The ability to make decisions is a mandatory feature of any programming language, and TeX, being a programming language (with emphasis on typesetting), is no exception. There are 17 control sequences [209, 210] that compare various quantities, and they are used to make decisions and to implement loops. Most are easy to use even for beginners, but the three commands `\ifx`, `\if` and `\ifcat` are different. They are harder to learn, are executed in different ways, are intended for different applications, and are confusing. Hence this tutorial.

All three have the same syntax, and must follow one of the forms below

⟨*command*⟩⟨*comparands*⟩⟨*then part*⟩`\else`
    ⟨*else part*⟩`\fi`

⟨*command*⟩⟨*comparands*⟩⟨*then part*⟩`\fi`

⟨*command*⟩⟨*comparands*⟩`\else`⟨*else part*⟩`\fi`

They start with one of the commands `\if`, `\ifx` or `\ifcat`, and test their comparands, in a way that will be described later, to see if they agree or match. If the test is successful, the *then* part is executed; otherwise, the *then* part is skipped and the *else* part is executed. The two parts may consist of any tokens (control sequences, text, and even other `if`s).

The two parts are optional. Any of them, or even both, may be omitted (in practice, of course, one never omits both). If the *else* part is omitted, the `\else`, of course, should be omitted as well. It may also happen that the process of evaluating the comparands creates extra tokens, which become included in the *then* part. The `\fi` is important. It serves to indicate the end of the *else* part or, in the absence of that part, the end of the *then* part. Even more important, in the case of nested `if`s, there should be `\fi`s to indicate the end of any of the inner `\if`s, as well as that of the outer one.

`\ifx`

Of the three comparisons, `\ifx` is the simplest and most useful one. It compares its two comparands without looking too deep into their values or meaning. The test

```
\def\qwe{trip} \def\rty{trip}
\ifx\qwe\rty
 \message{yes}\else\message{no}
\fi
```

will compare the two macros and, since their definitions are the same, the *then* part will be executed, displaying 'yes' in the log file and on the terminal. Similarly, the test `\ifx\qwe\rty True\else False\fi` results in the tokens 'True'.

A comparison always results in the expansion of either the *then* or the *else* parts. As mentioned before, each part may contain any tokens. Thus we may have, e.g.:

```
\baselineskip=\ifx\a\b 24pt\else 36pt\fi
\pageno=\count\ifx\a\b 0 \else 1 \fi
\message{\ifx\a\b success\else failure\fi}
\ifx\a\b true\else false\fi
\def\M{\ifx\a\b yes\else no\fi}
```

Or even something more sophisticated, such as:

```
\newif\ifSome
\csname
 Some\ifx\a\b true\else false\fi
\endcsname
\ifSome ...
```

The `\csname`, `\endcsname` pair creates one of the control sequences `\Sometrue`, `\Somefalse`, after which the test `\ifSome` is meaningful. Refs. 2 and 3 discuss `\csname`.

The following example is interesting. It shows the meaning of the words "...the *then* or *else* parts are *expanded.*"

`\ifx\a\b \x`⟨*argument*⟩ `\else \y`⟨*argument*⟩ `\fi`

Depending on how `\a`, `\b` are defined, either `\x` or `\y` is expanded, and its argument used in the expansion. However, if the argument is left outside the `\ifx`, it is not used in expanding either macro:

`\ifx\a\b \x \else \y \fi` ⟨*argument*⟩

If `\x` is expanded, its argument will be the `\else`; if `\y` is expanded, its argument will be the `\fi`. This is easy to verify with `\tracingmacros=1`.

The macros compared may have parameters. Thus

```
\def\qwe#1{samething}
\def\rty#1{samething}
\ifx\qwe\rty
```

evaluates to 'yes'. This suggests one use for `\ifx` namely, comparison of strings. To compare two strings, place them in macros, and compare the macros. `\ifx` is, in fact, heavily used in ref. 1 for this purpose. However

```
\def\qwe#1{samething}
\def\rty{samething}
\ifx\qwe\rty
```

will result in 'no'. Macros must have the same number of parameters to be considered equal by `\ifx`.

Can `\ifx` be used to compare a macro and a character? After defining `\def\a{*}`, both tests `\ifx\a*`, `\ifx*\a` are, surprisingly, a failure. However, after defining `\def\aster{*}`, both comparisons `\ifx\a\aster`, `\ifx\aster\a` are successful.

A similar example is a test for a null macro parameter. A straight comparison `\ifx#1\empty...` does not work. We first have to define a macro `\inner` whose value is `#1`, and then compare `\ifx\inner\empty` (See definition of `\empty` on [351]).

```
\def\testnull#1\\{\def\inner{#1}
 \ifx\inner\empty
  \message{yes}\else\message{no}
 \fi}
```

The test '`\testnull \\`' displays 'yes' on the terminal, while '`\testnull *\\`' displays 'no'.

To understand these results, we obviously need to know the rules for evaluating `\ifx`. They are numbered

- if both quantities being compared are macros, they should have the same number of parameters, the same top level definition, and the same status with respect to `\long` and `\outer`;
- in any other case, the quantities compared should have the same category code and the same character code.

A macro does not normally have either a character- or a category code. However, for the purpose of rule 2, a macro is considered to have character code 256 and category code 16. So when a macro is compared to a character, they will not match.

Rule 1 implies that `\ifx` can be used to compare macros, but it does not expand them and does not look too deep into their meanings. They are considered equal if they look the same on the surface (see examples later). Rule 2 implies that `\ifx` can compare two characters, but not, e.g., a character and a string, or two strings. Thus

▪ `\ifx AA` is a match by rule 2.

▪ `\ifx A#1` can be used, inside a macro, to see whether the first parameter is the letter 'A'.

▪ `\ifx Aa` is a failure since the comparands have different character codes.

▪ `\ifx {abc}{abc}` fails since it compares a '{' to an 'a'.

▪ `\ifx A{B}` fails since it compares the 'A' to the '{'.

▪ The test `\ifx*\a` above fails because of rule 1.

With these rules in mind, the following discussion and examples are easy to understand.

To compare a macro `\a` to a string `{abc}`, we first define `\def\b{abc}`, and then compare `\ifx\a\b`. If the string is a parameter of a macro, we can say `\def\mac#1{\def\inner{#1} \ifx\a\inner...}`. However, to compare `#1` to a single character, we can simply say

```
\def\mac#1{\ifx*#1...}
```

after which the expansion `\mac*` will be successful. There are some complex examples using this construct on [375–377].

To understand the meaning of 'top level definition', consider the following. Defining `\def\a{*} \def\b{*}`, the test `\ifx\a\b` is a success. However, the test

```
\def\a{\b} \def\c{\d}
\def\b{tests} \def\d{tests}
\ifx\a\c
```

is a failure, since `\ifx` compares only the top level definitions, and does not bother to expand `\b` and `\d` to find out that they are equal. Both tests

```
\def\qwe{\par} \def\rty{\par} \ifx\qwe\rty
```

and

```
\let\qwe=\par \let\rty=\par \ifx\qwe\rty
```

are successful, but

```
\let\xxx=\par \def\qwe{\par}
\def\rty{\xxx} \ifx\qwe\rty
```

fails, since `\ifx` does not expand its comparands to find their deep meaning.

This is an important feature of `\ifx` that has several consequences. One consequence is that `\ifx` is not bothered by undefined control sequences. It simply considers them all to be equal, so an `\ifx` comparison of two undefined control sequences always results in a match. Another consequence of the same feature is that two defined control sequences—such as e.g., `\if` and `\ifcat`—can be compared by `\ifx` without worrying about side effects resulting from their expansions. The comparison `\ifx\if\ifcat` is a failure, whereas `\ifx\if\if` is a success. It is also possible to compare the control sequence `\ifx` to itself, by means of `\ifx`. Thus `\ifx\ifx\ifx yes\else no\fi` results in 'yes'. Note that the `\fi` matches the first `\ifx`, and the other `if`s shouldn't have any matching `\fi`s since they are being compared, not executed. Consequently, the test `\ifx\fi\fi yes\else no\fi` also produces 'yes', as does `\ifx\message\message \message{yes} \else \message{no}\fi`

An interesting effect occurs when we try (perhaps as a serendipitous error)

```
\ifx\message \message{yes}\else
        \message{no}\fi
```

The `\ifx` compares the two tokens following, which are `\message` and `\message`. They are equal, so the word 'yes' is typeset. It is not displayed in the log file or on the terminal because the control sequence `\message`, which would normally have displayed it, has been used in the comparison. Continuing along the same lines, the test `\ifx \message{yes} \else \message{no}\fi` compares the control sequence `\message` to the '{'. They are not equal, because of rule 2 and, as a result, the *else* part is expanded, displaying 'no' in the log file, etc. Note that the string 'yes}' becomes part of the *then* part, and is skipped.

The reader should be able, at this point, to easily figure out the results of the following tests:

```
   \def\qwe#1{trip} \def\rty#1{trip}
1. \ifx\qwe\message{yes}\else\message{no}\fi
2. \ifx\qw\message{yes}\else\message{no}\fi
3. \ifx\q\message{yes}\else\message{no}\fi
4. \ifx\\message{yes}\else\message{no}\fi
5. \ifx\message{yes}\else\message{no}\fi
6. \ifx message{yes}\else\message{no}\fi
7. \ifx mmessage{yes}\else\message{no}\fi;
```

Tests 1–3 are similar, the control sequences `\qwe`, `\qw`, `\q` are compared to `\message`, which results, of course, in a 'no'. The undefined `\qw` and `\q` do not produce any errors. In 4, the control sequence `\\` (which is normally undefined) is compared to the first 'm' of 'message' and, in 5, `\message` is compared to the '{'. Test 6 compares the first two letters 'me' of 'message' to each other. The 6 tests result in a 'no' being displayed in the log file.

Test 7 compares the first two letters 'mm' of 'mmessage' to each other. They are equal, so everything up to the `\else` is expanded. This results in the tokens 'essageyes'.

Exercise 1. What are the results of:

```
    \def\\{message}
(a) \ifx\\message{yes}\else\message{no}\fi
(b) \ifx\\\\ yes\else no\fi
```

and why?

More about undefined control sequences. The test `\ifx\a\b`, where `\a`, `\b` are undefined, results in a match. This means that all undefined control sequences have the same meaning. On the other hand, the test `\ifx\a\relax`, where `\a` is undefined, is a failure. This means that an undefined control sequence is not equal to `\relax` (at least not its upper level meaning).

However, when the name of an undefined control sequence is synthesized by a `\csname`–`\endcsname` pair, that control sequence is made equal to `\relax`. Thus if `\a` is undefined, the construct `\csname a\endscsname` (which creates the name `\a`) is set equal to `\relax`, and the test `\expandafter\ifx\csname a\endcsname\relax` is a success. This is the basis of [Ex. 7.7]. It describes a macro `\ifundefined` that determines if any given string is the name of a defined macro.

```
\def\ifundefined#1{%
 \expandafter\ifx\csname#1\endcsname\relax}
```

The test

```
\ifundefined a
 \message{yes}\else\message{no}\fi
```

displays 'yes' in the log file if `\a` is undefined.

Exercise 2. Define a macro `\ifdefined#1` that will be the opposite of `\ifundefined` and be used in the same way.

What if `\a` has been defined as `\relax`? Predictably, the test '`\let\a=\relax \ifundefined a`' is successful. It is (somewhat) more surprising that the test '`\def\a{\relax} \ifundefined a`' is a failure. This difference is a direct consequence of the difference between `\let` and `\def`. Following is a short discussion of that difference, which is important in advanced applications, where macros are defined and compared.

The general form of `\let` is

`\let`⟨*control sequence*⟩`=`⟨*token*⟩

It defines the control sequence as being identical to the token. This is similar, but not identical to, `\def`⟨*control sequence*⟩`{`⟨*token*⟩`}`, and the following illustrates that difference. After `\def\a{X} \let\g=\a \def\h{\a}`, the sequence `\g\h` produces 'XX'. If we now redefine `\a`, the meaning of `\h` will change (since it was defined by `\def`) but `\g` will not change. Thus `\def\a{*} \g\h` produces 'X*'.

As a result, we can say that `\let\a=\b` assigns `\a` that value of `\b` which is current at the time the `\let` is executed, and this assignment is permanent. In contrast, `\def\a{\b}` assigns `\a` the name `\b`.

When \a is expanded, its expansion causes an expansion of \b, so the result is the value of \b. Each expansion of \a may, therefore, be different since \b may be redefined.

A more formal way of saying the same thing is: A \let makes a copy of the definition of \b, and that copy becomes the definition of \a; in contrast \def sets a pointer to point to the definition of \b, and that pointer becomes the definition of \a.

Back to \ifx. The comparands of an \ifx are not limited to just macros, primitives, or characters. They can also be:

- font names. \font\abc=cmr10 \font\xyz=cmr10 \relax \ifx\xyz\abc produces 'yes'.
- Active characters (see [Ex. 7.3]). The result of \let\a=~ \ifx\a~ is a match.

Exercise 3. Why does \def\a{~} \ifx\a~ fail?

- Names of the same TeX register. A test such as \countdef\me=3 \countdef\you=3 \ifx\you\me is a success.
- Macros defined at run time, such as in:

```
\def\tone{\count0=9 A }%
\message{Enter a definition}%
\read16 to\note
\ifx\tone\note
 \message{yes}\else\message{no}
\fi
```

If the user enters '\count0=9 A' from the keyboard, in response to the message, there will be a match. Entering anything else, such as '\count0=9 a', will result in a failure. In either case the value of \count0 will not be changed (by the way, what is it?), nor will the letters 'A' or 'a' be typeset. Notice that a message entered from the keyboard must terminate with a carriage return which, in turn, is converted by TeX into a space. This is why the definition of \tone must end with a space (to avoid that, change the value of \endlinechar as explained on [48]).

\if

The second comparison, \if, is executed in a completely different way. TeX expands the token following the \if (if it is expandable), then expands its expansion (if possible), and so on until only unexpandable tokens (characters or unexpandable control sequences) are left. If less than two unexpandable tokens are left, the process is repeated with the next input token. The process ends when there are two or more unexpandable tokens to be compared, or when an \else or a \fi are encountered. The final result is a string of unexpandable tokens, the first two of which are compared by *character code* but not by category code. The rest of the tokens, if any, are added to the *then* part.

If a comparand is an unexpandable control sequence, rather than a character, it is assigned a character code 256 and a catcode of 16. Thus the tests \if\hbox\vbox, \if\hskip\vskip, \if\hbox\kern, succeed. (See [209] for exceptions regarding the use of \let.) This also implies that comparing a primitive to a character always fails.

There is also the case where evaluating the comparands results in just one unexpandable token. Such a comparison should not be used since its result is undefined. Unfortunately, no error message is given by TeX. The advanced reader is referred to [§495] for the details of such a case.

The first example is simple \def\a{*}. Both tests \if\a*, \if*\a are successful (compare with the similar \ifx test above).

After \def\a{\b}, \def\c{\d}, \def\b{*}, \def\d{*}, the test \if\a\c is a 'yes'. However, \def\a{\b}, \def\c{\d}, \def\b{testing}, \def\d{testing}, \if\a\c will fail, since the two tokens compared are the first two characters resulting from the expansion of \a, which are 'te'. As mentioned above, the rest of \a (the string 'sting') and the whole of \c (the string 'testing') do not participate in the comparison, are added to the *then* part, and are therefore skipped. More insight into the working of \if is provided by the test

```
\def\a{\b} \def\c{\d}
\def\b{ttsting} \def\d{ttsting}
\if\a\c \message{yes}\else \message{no}\fi
```

It compares the first two t's of \a. They are equal, so TeX expands everything up to the \else. It displays 'yes', and also typesets the rest of \a ('sting') and the whole of \c ('ttsting'). Note that, again, \c is not used in the test.

Similar results are obtained in the experiment

```
\def\tone{*}
\message{Enter a}\read16 to\note%
\if\tone\note
```

Assuming that the user enters '*\count90=89', the result will be a match, and \count90 will also be set to 89. However, if the user enters '?\count90=89', the comparison will fail, and \count90 will not be affected. Similarly, if the user enters '*abc', the comparison will be successful, and the string 'abc' will be typeset. Entering, '?abc' however, will result in 'no', and the string 'abc' will be skipped.

The test `\if\s`, where `\s` is undefined, results in the message `! Undefined control sequence`, since `\if` always tries to expand its comparands.

Defining `\def\w{xyz}`, the test

```
\if x\w yes\else no\fi
```

is a success, since the first token of `\w` is an 'x'. However, The other two tokens are added to the *then* part, and the result of the test is the string 'yzyes'. Sometimes it is desirable to discard that part of `\w` that does not participate in the comparison. This is a special case of the general problem of how to extract the first token of a macro `\w` and discard the rest.

One way of doing it is:

```
\def\tmp#1#2\\{#1}
\ifx\w\empty
 \def\W{}
\else
 \def\W{\expandafter\tmp\w\\}
\fi
```

When `\W` is expanded, the first step is to expand `\w`, and the second, to expand `\tmp`. The first argument of `\tmp` is thus the first token of `\w`, and the second argument, the rest of `\w`. The result of expanding `\tmp` is thus the single token 'x', and that token becomes the definition of `\W`. The test `\if x\W yes\else no\fi` now results in the string 'yes'. This method works even if `\w` is `\empty`.

Exercise 4. Perform the test:

```
\if\the\count90 \the\count90
 \message{yes}\else\message{no}\fi
```

for `\count90` set to 1, 11 and 12.

The next example is the two tests `\let\a=~ \if\a~`, `\def\b{~} \if\b~`. In the first test, the `\let` makes `\a` equivalent to the active character '~'. In the second one, the `\def` makes `\b` a macro whose definition is the same active character '~'. The `\if` expands its comparands, so it ends up comparing '~' to '~'. Both tests thus result in a match.

Having mentioned active characters, let's use them to further illustrate the behaviour of `\if`. The following:

```
\def\a{*~}
\hbox{Mr. Drofnats}
\hbox{Mr.\if*\a\fi Drofnats}
\hbox{Mr.\if+\a\fi Drofnats}
```

results in:
Mr. Drofnats
Mr. Drofnats
Mr.Drofnats

which is easy to explain. The test `\if*\a\fi` expands `\a` and only uses its first character (the '*'). The second character (the tilde) remains and affects the space between 'Mr.' and 'Drofnats' (it has the effect of `\frenchspacing`). In contrast, the test `\if+\a\fi` expands `\a` and, since there is no match, *skips* the second character. As a result, there is no space between 'Mr.' and 'Drofnats'.

The `\noexpand` command can be used to suppress expansion during an `\if`. Assuming the definitions `\def\q{A}`, `\def\p{9}`, the test `\if\p\q` fails since it compares the characters 'A', '9'; however, the test `\if\noexpand\p\noexpand\q` is a success (even if the macros involved are undefined).

`\ifcat`

The third comparison, `\ifcat`, is less useful. It works like `\if`, expanding its comparands, and resulting in a string of characters, of which the first two are compared by category codes [37], but not by character codes. For example, the catcode of '&' is 4 (alignment tab) and the catcode of '8' is 12 (other). If we change the catcode of '8' to 4 and compare `\catcode`\8=4 \ifcat 8&`, we get a 'yes'. It is hard, however, to find simple, practical examples for `\ifcat` (the examples on the notorious [377] are hardly simple or practical).

Similar to an `\if`, there is also the case where expanding the comparands results in a non-expandable control sequence, rather than a character. In such a case, TeX assigns it a character code 256 and a catcode of 16. Thus all the following comparisons `\ifcat\hbox\vbox`, `\ifcat\hskip\vskip`, `\ifcat\hbox\kern`, succeed. (Again, see [209] for exceptions concerning the use of `\let`.)

The category code of a character can be typeset by the command `\the\catcode`\A`. It can be displayed in the log file by `\showthe\catcode`\A`. This does not work for control sequences since they have no catcode. When comparing control sequences with an `\ifcat`, they are first expanded, and the first two tokens are compared. For example, after defining `\def\a{&} \def\b{+=} \def\c{true}` the comparison `\ifcat\a\b` fails, since the catcodes of '&' and '+' are different. However, the comparison `\ifcat\b\c` is a 'yes' since the comparands are '+' and '='. The string 'true' is typeset.

It is possible to compare macros without expanding them. Assuming the definitions of `\a`, `\b` above, the test `\ifcat\noexpand\a\noexpand\b` results in a match since it does not expand the macros, and they are treated as undefined (category code 16).

Exercise 5. With `\c` defined as above, what is the result of `\ifcat\c`?

Perhaps the simplest practical example of `\ifcat` is a test for a letter. Assuming that the parameter of macro `\suppose` is supposed to be a letter or a string starting with a letter. The macro can be defined as: `\def\suppose#1{\ifcat A#1...\fi...}`.

Exercise 6. If the parameter of `\suppose` is a string, only the first character will be used by the `\ifcat`, and the rest will be added to the *then* part, perhaps interfering with the rest of the macro. Generalize the definition of `\suppose` to suppress the rest of the parameter during the `\ifcat`.

The examples

```
\def\a{~} \ifcat\a~
\let\b=~ \ifcat\b~
```

are identical to the ones shown earlier, in connection with `\if`. They behave the same as in that case, resulting both in a 'yes'.

Active characters may also be compared with an `\ifcat`, since they all have the same catcode (13). After defining ``\catcode`\?=13 \def?{:}``, ``\catcode`\!=13 \def!{;}``, the test `\ifcat?!` is a success, seemingly confirming the above statement. A deeper look, however, shows that the test expands the two active characters, and compares the catcodes of their values! The values just happen to have the same catcode. To actually compare the catcodes of the active characters, a `\noexpand` should be used to prevent their expansions. Thus the test `\ifcat\noexpand?\noexpand!` compares the catcodes of the active characters without expanding them (and is also a success).

Nested ifs

In principle, it is possible to nest `ifs` one inside another. An `if` may be a comparand of another `if`, or it may be nested in either the *then* or the *else* part of another `if`. However, because our three `ifs` work in different ways, not every combination of nested `ifs` is valid. In general, a nested `if` is written as

`\if..\if`⟨*inner*⟩`\fi..\else..\if`⟨*inner*⟩`\fi..\fi`

where any of the inner `ifs` may have an *else* part, and may itself be nested by other `ifs`. However, as the examples below show, such an `if` should be carefully analyzed before it is used, since it tends to produce unexpected results.

Since `\if` evaluates its comparands, they can be other `ifs`. Defining `\def\a{}`, `\def\b{**}`, the test

```
\if\ifx\a\b1\else\if\a\b23\fi\fi\else4\fi
```

(see [Ex. 20.13g]) is an `\if` with an `\ifx` as a comparand. The `\ifx`, in turn, has another `\if` nested in its *else* part.

The process starts when the outer `\if` evaluates its comparands in order to come up with two tokens for comparison. It activates the `\ifx` which, in turn, compares `\a` and `\b`. They are not equal, so the '1' is skipped, and TeX starts executing the *else* part of the `\ifx`. This part contains the inner `\if`, which evaluates `\a` and `\b`, compares the two asterisks, and results in the '23'. The outer `\if` is now equivalent to `\if23\else4\fi`, which typesets the '4'.

The test

```
\if\ifx\a\b1\else\if\a\b22+\fi\fi\else3\fi
```

is similar, it has the '+' left over after the comparison, so it gets typeset.

Exercise 7. What gets typeset by the following?
`\if\ifx\b\b1\else\if\a\b2\fi\fi+\else3\fi`

The `\ifcat` comparison is similar to `\if` in that it first evaluates its comparands. As a result, other comparisons may be used as comparands, and may also be nested inside an `\ifcat`. The following tests can be analyzed similarly to the ones above:

```
\ifcat\ifx\a\b1\else\if\a\b234\fi\fi\else5\fi
\ifcat\ifx\a\b1\else\if\a\b22+\fi\fi\else3\fi
\ifcat\ifx\b\b1\else\if\a\b2\fi\fi\else3\fi
```

Since `\ifx` does not evaluate its comparands, they cannot be other `\ifs`. Trying, e.g., `\ifx\if\a\b...`, the `\ifx` would simply compare the `\if` to the `\a`. We cannot even use braces to separate the inner and outer `ifs` `\ifx{\if\a\b...}...` since the `\ifx` will compare the '`{`' with the `\if`, and TeX will eventually complain of an '`Extra }`'.

We can, however, nest an `if` (of any type) in the *then* or *else* parts of an `\ifx`. The test

```
\ifx\a\b\else\if\a\b ok\fi\fi
```

(with `\a`, `\b` defined as above) typesets 'ok'. The `\ifx` compares `\a` and `\b` and finds them different. It skips to the *else* part and expands it. The inner `\if` is thus executed in the usual way; it finds two identical tokens (the two asterisks of `\b`), and typesets 'ok'.

Examples

1. A practical example is macro `\flexins` below. It lets the user decide, *at run time*, whether any floating insertion should be a `\midinsert` or a `\topinsert`. The user is prompted to enter either 'mid' or 'top' from the keyboard. In response, the macro uses a nested `\ifx` to create either a `\midinsert` or a `\topinsert`.

```
\def\flexins{%
  \def\b{mid } \def\d{top }
  \message{mid or top? }\read-1 to\a
  \csname
    \ifx\a\b mid%
    \else
      \ifx\a\d top\fi
    \fi insert%
  \endcsname
}

\flexins
⟨Insertion material⟩
\endinsert
```

What if the user enters none of these inputs. Clearly `\flexins` should be extended so it can recover from a bad input. It is a good idea to expand `\flexins` recursively, in such a case, to give the user another chance to enter a valid input. The first try is:

```
\def\flexins{%
  \def\b{mid } \def\d{top }
  \message{mid or top? }\read-1 to\a
  \csname
    \ifx\a\b mid\else
      \ifx\a\d top\else \flexins\fi
    \fi insert\endcsname
}
```

It does not work! When TeX expands `\flexins` recursively, it is still inside the `\csname`. During the recursive expansion it finds `\def\b`, but `\def` is not expandable, and thus not supposed to be inside a `\csname` [40]. The result is an error message (which one?).

We now realize that we have to delay the recursive expansion of `\flexins` until we get out of the `\csname`–`\endcsname` pair. The final version is:

```
\def\flexins{%
  \def\b{mid } \def\d{top }
  \def\badinsert{\flexins}
  \message{mid or top? }\read-1 to\a
  \csname
    \ifx\a\b mid\else
      \ifx\a\d top\else bad\fi
    \fi insert\endcsname
}
```

In the case of bad input, the `\csname`–`\endcsname` pair creates the control sequence name `\badinsert`. We predefine it to simply expand `\flexins`, which then asks the user for another input.

2. (Proposed by R. Whitney.) This is a generalization of the previous example. Macro `\yesno` below prompts the user to respond with a 'Y' or a 'N', but also accepts the responses 'y', 'n'. It does the following:

- Prompts the user with a question where the response can be 'Y', 'N', 'y', or 'n'.
- Reads the response into `\ans`.
- Uses `\ifx` to compare `\ans` to macros containing one of the valid responses.
- If a match is found, uses `\csname` to create the name of, and expand, one of the macros `\yesresult`, `\noresult`. These macros should be predefined to do anything desirable.
- If no match is found, expands `\badresult`, which, in turn, should expand `\yesno` recursively.

```
\def\y{y } \def\n{n } \def\Y{Y }
\def\N{N } \def\badresult{\yesno}
\def\yesresult{⟨whatever⟩}
\def\noresult{⟨whatever⟩}
\def\yesno{%
  \message{Respond with a Y or N! }
  \read-1 to\ans
  \csname
    \ifx\y\ans yes\else
      \ifx\Y\ans yes\else
         \ifx\n\ans no\else
            \ifx\N\ans no\else bad%
            \fi
         \fi
      \fi
    \fi result\endcsname
}
\yesno
```

A different version of `\yesno` uses `\if` instead of `\ifx`. We start with:

```
\def\badresult{\yesno}
\def\yesresult{⟨whatever⟩}
\def\noresult{⟨whatever⟩}
\def\yesno{%
  \message{Respond with a Y or N! }
  \read-1 to\ans
  \csname
    \if y\ans yes\else
      \if Y\ans yes\else
        \if n\ans no\else
          \if N\ans no\else bad%
        \fi
      \fi
    \fi
  \fi result\endcsname
}
```

It compares `\ans` to the token 'y' instead of the macro `\y`, but it does not work! Macro `\ans` contains a 'y' (or 'Y' or whatever), followed by a space. The space gets added to the *then* part, which then becomes '␣yes', creating the control sequence `\␣yesresult`. To get this to work, the first token of `\ans` has to be extracted, and all the other ones discarded. This can be done, as shown elsewhere, by

```
\def\tmp#1#2\\{#1}
\def\sna{\expandafter\tmp\ans\\}
```

Macro `\sna` now contains just one character, and the next version is:

```
\def\badresult{\yesno}
\def\yesresult{⟨whatever⟩}
\def\noresult{⟨whatever⟩}
\def\yesno{%
  \message{Respond with a Y or N! }
  \read-1 to\ans
  \def\tmp##1##2\\{##1}
  \def\sna{\expandafter\tmp\ans\\}%
  \csname
    \if y\sna yes\else
      \if Y\sna yes\else
        \if n\sna no\else
          \if N\sna no\else bad%
          \fi
        \fi
      \fi
    \fi result\endcsname
}
```

Note that it works for any response that's a string starting with one of the four valid characters.

Exercise 8. Extend this example. Define a macro `\triresponse` that accepts the responses 'left', 'right', 'center', or any strings that start with 'l', 'r', or 'c'. The macro then expands one of the (predefined) macros `\doleft`, `\doright`, `\docenter` or `\dobad`.

3. A practical example of the use of `\ifcat` arises when style files are used. If such a file has internal macros, they can be made private by declaring `\catcode`\@=11`, and giving the macros names that include the '@'. At the end of the file, a matching `\catcode`\@=12` should be placed. The problem occurs when such a style file, say `b.sty`, is `\input` by another file, `a.sty`, that also contains the pair `\catcode`\@=11`, `\catcode`\@=12`. A simple test should reveal the problem to readers who still don't see it. The solution is to place the test

```
\ifcat @A\chardef\catcount=12
\else
  \chardef\catcount=\catcode`\@
\fi
\catcode`\@=11
```

at the beginning of `b.sty`, and reset at the end to `\catcode`\@=\catcount`.

Exercise 9. Use `\ifcat` to solve the following problem: Given `\def\foo#1{...}`, devise a test to see if, in the expansion `\foo...`, the argument is delimited by a space. Normally, such a space is automatically absorbed by TeX and cannot be recognized.

4. A compound macro argument.

Macro `\compndArg` accepts a compound argument and breaks it down into its components. The argument should be of the form `xxx,xxx,...,xxx;` (the ',' separates the individual components and the ';' delimits the entire argument). The macro accepts the argument (without the ';', of course), it appends ',;,' to the argument, and makes the whole thing the argument of `\pickup`, which is then expanded.

```
\def\compndArg#1;{\pickup#1,;,}
\def\pickup#1,{% Note that #1 may be \null
\if;#1\let\next=\relax
\else\let\next=\pickup
  \message{`#1'}% use #1 in any way
\fi\next}
```

Macro `\pickup` expects its arguments to be delimited by a comma, so it ends up getting the first component of the original argument. It uses it in any desired way and then expands itself recursively. The process ends when the current argument becomes the semicolon. Note the following:

- This is also an example of a macro with a variable number of parameters. The compound argument may have any number of components (even zero, see below).
- The method works even for an empty argument. The expansion '`\compndArg ;`' will cause `\pickup` to be expanded with a null argument.
- The macros do not create spurious spaces. In many macro lines, the end-of-line character gets converted to a space, which is eventually typeset if the macro is invoked in horizontal mode. Such lines should be identified, with a test such as `C\compndArg g;D`, and should be terminated by a

'%'. Try the above test with and without the '%' in the first line of \pickup.

Conclusion

The main source of the confusion surrounding the various \if comparisons is the inability to find out exactly what TEX is comparing. In future extensions of TEX it would be useful to have a control sequence \tracingcomparands such that setting \tracingcomparands=1 would show, on the terminal, the actual quantities compared.

Answers to exercises

1. (a) displays 'no' on the terminal since it compares the macro \\ to the letter 'm'; (b) typesets 'yes' since it compares two identical macros.

2. Macro \ifundefined supplies the \ifx, and the matching \else and \fi are provided outside. We want \ifdefined also to supply an if that can be completed outside. We start with the test for an undefined macro

```
\expandafter\ifx\csname#1\endcsname\relax
```

and create either an \iffalse (if the macro is undefined) or an \iftrue (in case it is defined), to be matched outside. The first version is:

```
\def\ifdefined#1 {%
 \expandafter\ifx\csname#1\endcsname\relax
 \let\next=\iffalse\else\let\next=\iftrue\fi
 \next}
```

But it fails! The reason is explained at the bottom of [211]. The next, working, version is:

```
\def\maca{\let\next=\iffalse}
\def\macb{\let\next=\iftrue}
\def\ifdefined#1 {%
 \expandafter\ifx\csname#1\endcsname\relax
 \maca\else\macb\fi \next}
```

After which, we can say:

```
\ifdefined a \message{yes}\else
         \message{no}\fi
```

3. Because \let\a=~ defines \a as an active character, where as \def\a{~} defines \a as a macro (whose value is the active character '~'). The \ifx does not look too deep into the meaning of its comparands, so it decides that a macro is not equal to an active character. In contrast, the \if comparison, discussed later, which looks deeper into the meaning of its comparands, returns a 'yes' for both tests.

4. Just do it. It's worth it. Then do the similar test

```
\if\the\count90\the\count90
 \message{yes}\else\message{no}\fi
```

5. A success, since it compares the catcodes of the two letters 't', 'r'. It also typesets 'ue'.

6. Macro \tmp expands to the first character of the parameter.

```
\def\suppose#1{\def\tmp##1##2\\{##1}%
  \ifcat A\tmp#1\\...\else...\fi...}
```

7. The \ifx compares \b and \b, and they, of course, match. The '1' is thus the first token left for the outer \if to compare. The rest of the \ifx (\else\if\a\b2\fi\fi) is skipped. Next comes the '+', followed by the *else* part of the outer \if, with the '3'. The outer \if can now be written as \if1+\else3\fi which, of course, typesets the '3'.

8. Answer not provided.

9. We place an expansion of \isnextspace at the end of \foo. This sets \next to the token following the parameter of \foo. The \ifcat can then be used to compare the category of \next to that of a space. The following test

```
\def\spacecheck{%
 \ifcat\next\space
 \message{yes}\else\message{no}\fi}
\def\isnextspace{\futurelet\next\spacecheck}
\def\foo#1{#1\isnextspace}
\foo{A} \foo{B}.\foo{C} D
```

produces 'yes no yes' on the terminal.

References

1. Greene, A. M., *BASIX—An Interpreter Written in TEX*, *TUGboat* 11 (1990), no. 3, pp. 385–392.
2. Bechtolsheim, S., \csname *and* \string, *TUGboat* 10 (1989), no. 3, pp. 203–206.
3. Hendrickson, A., *Getting TEXnical*, *TUGboat* 11 (1990), no. 3, pp. 359–370.

◇ David Salomon
California State University,
Northridge
Computer Science Department
Northridge, CA 91330
dxs@ms.secs.csun.edu