macro **1**

`macro.` A *macro* is a definition that gives a name to a pattern of TEX input text.[1] The name can be either a control sequence or an active character. The pattern is called the "replacement text". The primary command for defining macros is the `\def` control sequence.

As a simple example, suppose that you have a document in which the sequence '$\cos\theta + i \sin\theta$' occurs many times. Instead of writing it out each time, you can define a macro for it:

```
\def\arctheta{\cos \theta + i \sin \theta}
```

Now whenever you need this sequence, you can just "call" the macro by writing '`\arctheta`' and you'll get it. For example, '`$e^{\arctheta}$`' will give you '$e^{\cos\theta+i\sin\theta}$'.

But the real power of macros lies in the fact that a macro can have parameters. When you call a macro that has parameters, you provide arguments that are substituted for those parameters. For example, suppose you write:

```
\def\arc#1{\cos #1 + i \sin #1}
```

The notation `#1` indicates the first parameter of the macro, which in this case has only one parameter. You now can produce a similar form, such as '$\cos 2t + i \sin 2t$', with the macro call '`\arc {2t}`'.

More generally, a macro can have up to nine parameters, which you indicate as '`#1`', '`#2`', etc. in the macro definition. TEX provides two kinds of parameters: delimited parameters and undelimited parameters. Briefly, a delimited parameter has an argument that's delimited, or ended, by a specified sequence of tokens (the delimiter), while an undelimited parameter has an argument that doesn't need a delimiter to end it. First we'll explain how macros work when they have only undelimited parameters, and then we'll explain how they work when they have delimited parameters.

If a macro has only undelimited parameters, those parameters must appear one after another in the macro definition *with nothing between them or between the last parameter and the left brace in front of the replacement text.* A call on such a macro consists of the macro name followed by the arguments of the call, one for each parameter. Each argument is either:

- a single token other than a left or right brace, or
- a sequence of tokens enclosed between a left brace and a matching right brace.[2]

When TEX encounters a macro, it expands the macro in its gullet (see "anatomy of TEX", p. '`\anatomy`') by substituting each argument for the corresponding parameter in the replacement text. The resulting text may contain other macro calls. When TEX encounters such an embedded

[1] More precisely, the definition gives a name to a sequence of tokens.

[2] The argument can have nested pairs of braces within it, and each of these pairs can indicate either a group or a further macro argument.

macro call, it expands that call immediately without looking at what follows the call.[3] When TEX's gullet gets to a primitive command that cannot be further expanded, TEX passes that command to TEX's stomach. The order of expansion is sometimes critical, so in order to help you understand it we'll give you an example of TEX at work.

Suppose you provide TEX with the following input:

```
\def\a#1#2{\b#2#1\kern 2pt #1}
\def\b{bb}
\def\c{\char49 cc}
\def\d{dd}
\a\c{e\d} % Call on \a.
```

Then the argument corresponding to `#1` is `\c`, and the argument corresponding to `#2` is `e\d`. TEX expands the macro call in the following steps:

```
\b e\d\c\kern 2pt \c
bbe\d\c\kern 2pt \c
\d\c\kern 2pt \c   (‘b’, ‘b’, ‘e’ sent to stomach)
dd\c\kern 2pt \c
\c\kern 2pt \c   (‘d’, ‘d’ sent to stomach)
\char49 cc\kern 2pt \c
\c   (‘\char’, ‘4’, ‘9’, ‘c’, ‘c’, ‘\kern’, ‘2’, ‘p’, ‘t’ sent to stomach)
\char49 cc
(‘\char49’, ‘c’, ‘c’ sent to stomach)
```

Note that the letters ‘`b`’, ‘`c`’, ‘`d`’, and ‘`e`’ and the control sequences ‘`\kern`’ and ‘`\char`’ are all primitive commands that cannot be expanded further.

A macro can also have “delimited parameters”, which can be mixed with the undelimited ones in any combination. The idea of a delimited parameter is that TEX finds the corresponding argument by looking for a certain sequence of tokens that marks the end of the argument—the delimiter. That is, when TEX is looking for such an argument, it takes the argument to be all the tokens from TEX's current position up to but not including the delimiter.

You indicate a delimited parameter by writing ‘`#`n’ (n must be between 0 and 9) followed by one or more tokens that act as the delimiter. The delimiter extends up to the next ‘`#`’ or ‘`{`’—which makes sense since ‘`#`’ starts another parameter and ‘`{`’ starts the replacement text.

The delimiter can't be ‘`#`’ or ‘`{`’, so you can tell a delimited parameter from an undelimited one by looking at what comes after it.

If the character after the parameter is ‘`#`’ or ‘`{`’, you've got an undelimited parameter; otherwise you've got a delimited one. Note the difference in arguments for the two kinds of parameters—an undelimited parameter

[3] In computer science terminology, the expansion is “depth first” rather than “breadth first”. Note that you can modify the order of expansion with commands such as `\expandafter`.

is matched either by a single token or by a sequence of tokens enclosed in braces, while a delimited parameter is matched by any number of tokens, even zero.

An example of a macro that uses two delimited parameters is:

```
\def\diet#1 #2.{On #1 we eat #2!}
```

Here the first parameter is delimited by a single space and the second parameter is delimited by a period. If you write:

```
\diet Tuesday turnips.
```

you'll get the text "On Tuesday we eat turnips!". But if the delimiting tokens are enclosed in a group, TEX doesn't consider them as delimiting. So if you write:

```
\diet {Sunday mornings} pancakes.
```

you'll get the text 'On Sunday mornings we eat pancakes!' even though there's a space between '`Sunday`' and '`morning`'. When you use a space as a delimiter, an end-of-line character ordinarily also delimits the argument since TEX converts the end-of-line to a space before the macro mechanism ever sees it.

Once in a while you might need to define a macro that has '`#`' as a meaningful character within it. You're most likely to need to do this when you're defining a macro that in turn defines a second macro. What then do you do about the parameters of the second macro to avoid getting TEX confused? The answer is that you write two '`#`'s for every one that you want when the first macro is expanded. For example, suppose you write the macro definition:

```
\def\first#1{\def\second##1{#1/##1}}
```

Then the call '`\first{One}`' defines '`\second`' as:

```
\def\second#1{One/#1}
```

and the subsequent call '`\second{Two}`' produces the text 'One/Two'.

A number of commands provide additional ways of defining macros (see pp. '`mac1`'–'`mac2`'). For the complete rules pertaining to macros, see Chapter 20 of *The TEXbook*.