COMPUTERS & TYPESETTING

JUBILEE • 2021 • edition

DONALD E. KNUTH

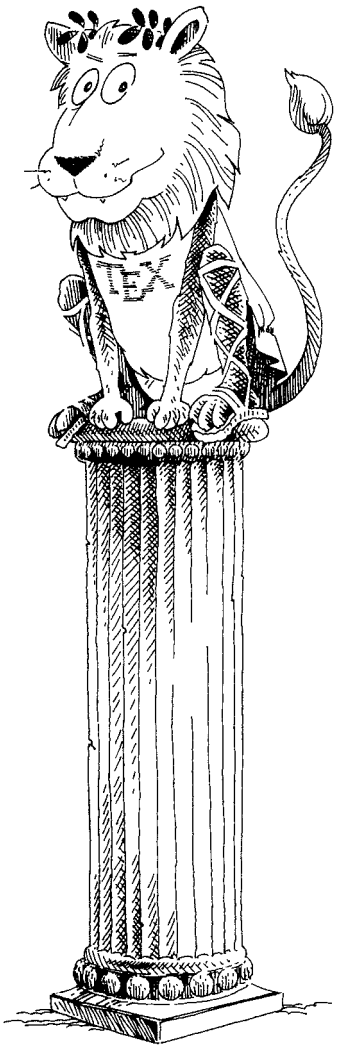
COMPUTERS & TYPESETTING / A

The TEXbook

COMPUTERS & TYPESETTING / A

The TEXbook

DONALD E. KNUTH *Stanford University*



Illustrations by
DUANE BIBBY

ADDISON–WESLEY

Upper Saddle River, NJ
Boston · Indianapolis
San Francisco · New York
Toronto · Montréal
London · Munich
Paris · Madrid
Capetown · Sydney · Tokyo
Singapore · Mexico City

This manual describes TEX Version 3.0. Some of the advanced features mentioned here are absent from earlier versions.

TEX is a trademark of the American Mathematical Society.

METAFONT is a trademark of Addison–Wesley Publishing Company.

Library of Congress cataloging in publication data

```
Knuth, Donald Ervin, 1938-
   The TeXbook.

   (Computers & Typesetting ; A)
   Includes index.
   1. TeX (Computer system).  2. Computerized
typesetting.  3. Mathematics printing.  I. Title.
II. Series:  Knuth, Donald Ervin, 1938-    .
Computers & typesetting ; A.
Z253.4.T47K58  1986         686.2'2544      85-30845
ISBN 0-201-13447-0
ISBN 0-201-13448-9 (soft)
```

Incorporates all corrections known in 2020.

Internet page `http://www-cs-faculty.stanford.edu/~knuth/abcde.html` contains current information about this book and related books.

ISBN-13 978-0-201-13447-6
ISBN-10 0-201-13447-0

Twenty-third Printing, February 2021

Electronic edition, May 2021

To Jill:
For your books and brochures

Preface

GENTLE READER: This is a handbook about TEX, a new typesetting system intended for the creation of beautiful books—and especially for books that contain a lot of mathematics. By preparing a manuscript in TEX format, you will be telling a computer exactly how the manuscript is to be transformed into pages whose typographic quality is comparable to that of the world's finest printers; yet you won't need to do much more work than would be involved if you were simply typing the manuscript on an ordinary typewriter. In fact, your total work will probably be significantly less, if you consider the time it ordinarily takes to revise a typewritten manuscript, since computer text files are so easy to change and to reprocess. (If such claims sound too good to be true, keep in mind that they were made by TEX's designer, on a day when TEX happened to be working, so the statements may be biased; but read on anyway.)

This manual is intended for people who have never used TEX before, as well as for experienced TEX hackers. In other words, it's supposed to be a panacea that satisfies everybody, at the risk of satisfying nobody. Everything you need to know about TEX is explained here somewhere, and so are a lot of things that most users don't care about. If you are preparing a simple manuscript, you won't need to learn much about TEX at all; on the other hand, some things that go into the printing of technical books are inherently difficult, and if you wish to achieve more complex effects you will want to penetrate some of TEX's darker corners. In order to make it possible for many types of users to read this manual effectively, a special sign is used to designate material that is for wizards only: When the symbol

appears at the beginning of a paragraph, it warns of a "dangerous bend" in the train of thought; don't read the paragraph unless you need to. Brave and experienced drivers at the controls of TEX will gradually enter more and more of these hazardous areas, but for most applications the details won't matter.

All that you really ought to know, before reading on, is how to get a file of text into your computer using a standard editing program. This manual explains what that file ought to look like so that TEX will understand it, but basic computer usage is not explained here. Some previous experience with technical typing will be quite helpful if you plan to do heavily mathematical work with TEX, although it is not absolutely necessary. TEX will do most of the necessary

formatting of equations automatically; but users with more experience will be able to obtain better results, since there are so many ways to deal with formulas.

Some of the paragraphs in this manual are so esoteric that they are rated

 ;

everything that was said about single dangerous-bend signs goes double for these. You should probably have at least a month's experience with TEX before you attempt to fathom such doubly dangerous depths of the system; in fact, most people will never need to know TEX in this much detail, even if they use it every day. After all, it's possible to drive a car without knowing how the engine works. Yet the whole story is here in case you're curious. (About TEX, not cars.)

The reason for such different levels of complexity is that people change as they grow accustomed to any powerful tool. When you first try to use TEX, you'll find that some parts of it are very easy, while other things will take some getting used to. A day or so later, after you have successfully typeset a few pages, you'll be a different person; the concepts that used to bother you will now seem natural, and you'll be able to picture the final result in your mind before it comes out of the machine. But you'll probably run into challenges of a different kind. After another week your perspective will change again, and you'll grow in yet another way; and so on. As years go by, you might become involved with many different kinds of typesetting; and you'll find that your usage of TEX will keep changing as your experience builds. That's the way it is with any powerful tool: There's always more to learn, and there are always better ways to do what you've done before. At every stage in the development you'll want a slightly different sort of manual. You may even want to write one yourself. By paying attention to the dangerous bend signs in this book you'll be better able to focus on the level that interests you at a particular time.

Computer system manuals usually make dull reading, but take heart: This one contains JOKES every once in a while, so you might actually enjoy reading it. (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully*.)

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. When certain concepts of TEX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the

earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

In order to help you internalize what you're reading, EXERCISES are sprinkled through this manual. It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure.

The TEX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details. Both languages have been called TEX; but henceforth the old language should be called TEX78, and its use should rapidly fade away. Let's keep the name TEX for the language described here, since it is so much better, and since it is not going to change any more.

I wish to thank the hundreds of people who have helped me to formulate this "definitive edition" of the TEX language, based on their experiences with preliminary versions of the system. My work at Stanford has been generously supported by the National Science Foundation, the Office of Naval Research, the IBM Corporation, and the System Development Foundation. I also wish to thank the American Mathematical Society for its encouragement, for establishing the TEX Users Group, and for publishing the *TUGboat* newsletter (see Appendix J).

Stanford, California — D. E. K.
June 1983

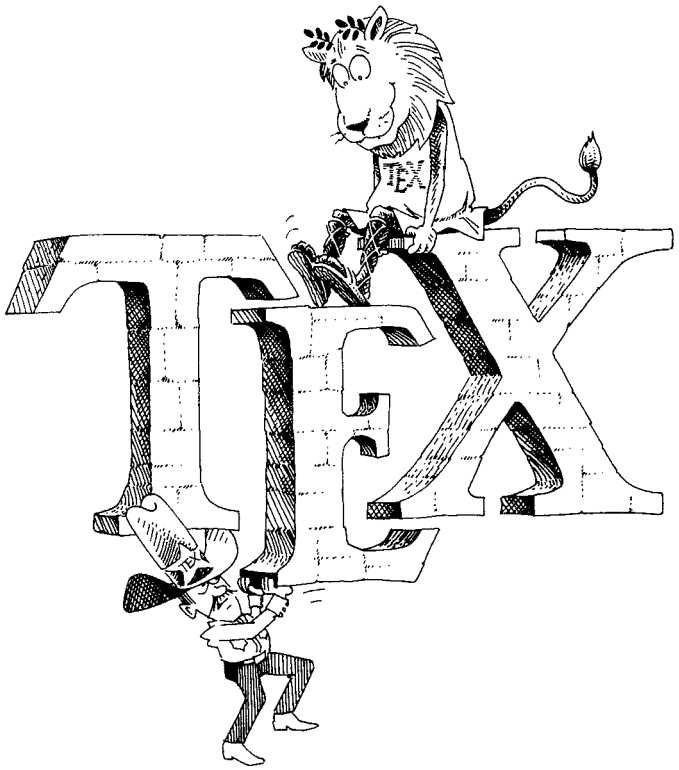
'Tis pleasant, sure, to see one's name in print;
A book's a book, although there's nothing in 't.
— BYRON, *English Bards and Scotch Reviewers* (1809)

A question arose as to whether we were covering the field
that it was intended we should fill with this manual.
— RICHARD R. DONNELLEY, *Proceedings, United Typothetæ of America* (1897)

Contents

1

The Name of the Game



English words like 'technology' stem from a Greek root beginning with the letters $\tau\epsilon\chi\ldots$; and this same Greek word means *art* as well as technology. Hence the name TEX, which is an uppercase form of $\tau\epsilon\chi$.

Insiders pronounce the χ of TEX as a Greek chi, not as an 'x', so that TEX rhymes with the word blecchhh. It's the 'ch' sound in Scottish words like *loch* or German words like *ach*; it's a Spanish 'j' and a Russian 'kh'. When you say it correctly to your computer, the terminal may become slightly moist.

The purpose of this pronunciation exercise is to remind you that TEX is primarily concerned with high-quality technical manuscripts: Its emphasis is on art and technology, as in the underlying Greek word. If you merely want to produce a passably good document—something acceptable and basically readable but not really beautiful—a simpler system will usually suffice. With TEX the goal is to produce the *finest* quality; this requires more attention to detail, but you will not find it much harder to go the extra distance, and you'll be able to take special pride in the finished product.

On the other hand, it's important to notice another thing about TEX's name: The 'E' is out of kilter. This displaced 'E' is a reminder that TEX is about typesetting, and it distinguishes TEX from other system names. In fact, TEX (pronounced *tecks*) is the admirable *Text EXecutive* processor developed by Honeywell Information Systems. Since these two system names are pronounced quite differently, they should also be spelled differently. The correct way to refer to TEX in a computer file, or when using some other medium that doesn't allow lowering of the 'E', is to type '`TeX`'. Then there will be no confusion with similar names, and people will be primed to pronounce everything properly.

▸ **EXERCISE 1.1**

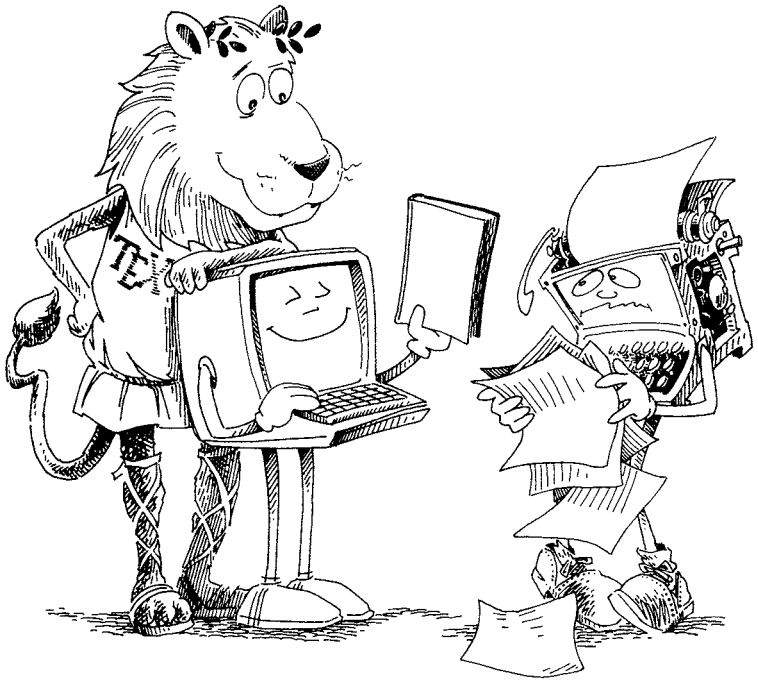
After you have mastered the material in this book, what will you be: a TEXpert, or a TEXnician?

They do certainly give
very strange and new-fangled names to diseases.
— PLATO, *The Republic*, Book 3 (c. 375 B.C.)

Technique! The very word is like the shriek
Of outraged Art. It is the idiot name
Given to effort by those who are too weak,
Too weary, or too dull to play the game.
— LEONARD BACON, *Sophia Trenton* (1920)

2

Book Printing versus Ordinary Typing



When you first started using a computer terminal, you probably had to adjust to the difference between the digit '1' and the lowercase letter 'l'. When you take the next step to the level of typography that is common in book publishing, a few more adjustments of the same kind need to be made; your eyes and your fingers need to learn to make a few more distinctions.

In the first place, there are two kinds of quotation marks in books, but only one kind on the typewriter. Even your computer terminal, which has more characters than an ordinary typewriter, probably has only a non-oriented double-quote mark ("), because the standard ASCII code for computers was not invented with book publishing in mind. However, your terminal probably does have two flavors of single-quote marks, namely ` and '; the second of these is useful also as an apostrophe. American keyboards usually contain a left-quote character that shows up as something like `, and an apostrophe or right-quote that looks like ' or ´.

To produce double-quote marks with TEX, you simply type two single-quote marks of the appropriate kind. For example, to get the phrase

"I understand."

(including the quotation marks) you should type

```
``I understand.''
```

to your computer.

A typewriter-like style of type will be used throughout this manual to indicate TEX constructions that you might type on your terminal, so that the symbols actually typed are readily distinguishable from the output TEX would produce and from the comments in the manual itself. Here are the symbols to be used in the examples:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789"#$%&@*+-=,.:;?!
()<>[]{}``''\|/_^~
```

If your computer terminal doesn't happen to have all of these, don't despair; TEX can make do with the ones you have. An additional symbol

```
␣
```

is used to stand for a *blank space*, in case it is important to emphasize that a blank space is being typed; thus, what you *really* type in the example above is

```
``I␣understand.''
```

Without such a symbol you would have difficulty seeing the invisible parts of certain constructions. But we won't be using '␣' very often, because spaces are usually visible enough.

Book printing differs significantly from ordinary typing with respect to dashes, hyphens, and minus signs. In good math books, these symbols are all different; in fact there usually are at least four different symbols:

a hyphen (-);
an en-dash (–);
an em-dash (—);
a minus sign (−).

Hyphens are used for compound words like 'daughter-in-law' and 'X-rated'. En-dashes are used for number ranges like 'pages 13–34', and also in contexts like 'exercise 1.2.6–52'. Em-dashes are used for punctuation in sentences—they are what we often call simply dashes. And minus signs are used in formulas. A conscientious user of TEX will be careful to distinguish these four usages, and here is how to do it:

for a hyphen, type a hyphen (`-`);
for an en-dash, type two hyphens (`--`);
for an em-dash, type three hyphens (`---`);
for a minus sign, type a hyphen in mathematics mode (`$-$`).

(Mathematics mode occurs between dollar signs; it is discussed later, so you needn't worry about it now.)

▸**EXERCISE 2.1**
Explain how to type the following sentence to TEX: Alice said, "I always use an en-dash instead of a hyphen when specifying page numbers like '480–491' in a bibliography."

▸**EXERCISE 2.2**
What do you think happens when you type four hyphens in a row?

If you look closely at most well-printed books, you will find that certain combinations of letters are treated as a unit. For example, this is true of the 'f' and the 'i' of 'find'. Such combinations are called *ligatures*, and professional typesetters have traditionally been trained to watch for letter combinations such as `ff`, `fi`, `fl`, `ffi`, and `ffl`. (The reason is that words like 'find' don't look very good in most styles of type unless a ligature is substituted for the letters that clash. It's somewhat surprising how often the traditional ligatures appear in English; other combinations are important in other languages.)

▸**EXERCISE 2.3**
Think of an English word that contains two ligatures.

The good news is that you do *not* have to concern yourself with ligatures: TEX is perfectly capable of handling such things by itself, using the same mechanism that converts '`--`' into '–'. In fact, TEX will also look for combinations of adjacent letters (like '`A`' next to '`V`') that ought to be moved closer together for better appearance; this is called *kerning*.

To summarize this chapter: When using TEX for straight copy, you type the copy as on an ordinary typewriter, except that you need to be careful about quotation marks, the number 1, and various kinds of hyphens/dashes. TEX will automatically take care of other niceties like ligatures and kerning.

(Are you sure you should be reading this paragraph? The "dangerous bend" sign here is meant to warn you about material that ought to be skipped on first reading. And maybe also on second reading. The reader-beware paragraphs sometimes refer to concepts that aren't explained until later chapters.)

If your keyboard does not contain a left-quote symbol, you can type `\lq`, followed by a space if the next character is a letter, or followed by a `\` if the next character is a space. Similarly, `\rq` yields a right-quote character. Is that clear?

```
\lq\lq␣I␣understand.\rq\rq\␣
```

In case you need to type quotes within quotes, for example a single quote followed by a double quote, you can't simply type `'''` because TEX will interpret this as "' (namely, double quote followed by single quote). If you have already read Chapter 5, you might expect that the solution will be to use grouping—namely, to type something like `{'}''`. But it turns out that this doesn't produce the desired result, because there is usually less space following a single right quote than there is following a double right quote: What you get is '", which is indeed a single quote followed by a double quote (if you look at it closely enough), but it looks almost like three equally spaced single quotes. On the other hand, you certainly won't want to type `'␣''`, because that space is much too large—it's just as large as the space between words—and TEX might even start a new line at such a space when making up a paragraph! The solution is to type `'\thinspace''`, which produces '" as desired.

▸**EXERCISE 2.4**
OK, now you know how to produce "' and '"; how do you get "‘ and ‘"?

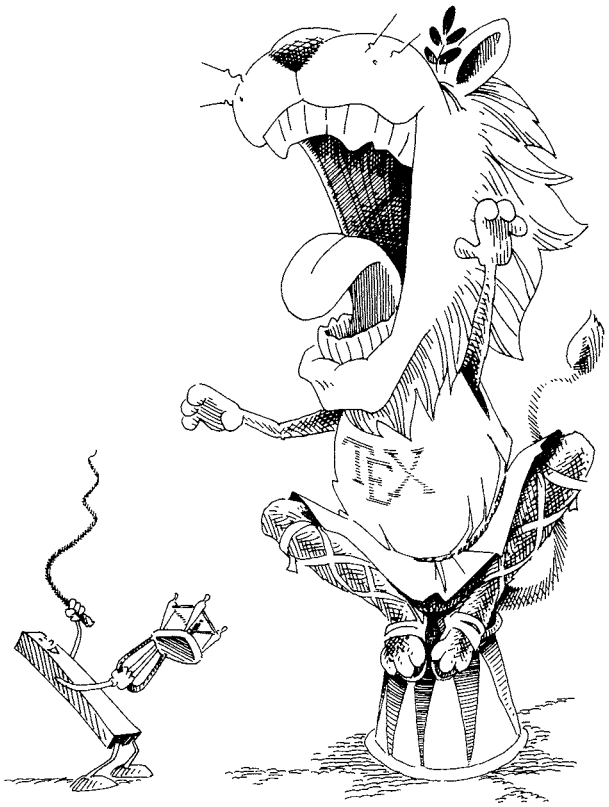
▸**EXERCISE 2.5**
Why do you think the author introduced the control sequence `\thinspace` to solve the adjacent-quotes problem, instead of recommending the trickier construction `'$\,$''` (which also works)?

In modern Wit all printed Trash, is
Set off with num'rous Breaks——*and* Dashes—
— JONATHAN SWIFT, *On Poetry: A Rapsody* (1733)

Some compositors still object to work
in offices where type-composing machines are introduced.
— WILLIAM STANLEY JEVONS, *Political Economy* (1878)

3

Controlling TEX



Your keyboard has very few keys compared to the large number of symbols that you may want to specify. In order to make a limited keyboard sufficiently versatile, one of the characters that you can type is reserved for special use, and it is called the *escape character*. Whenever you want to type something that controls the format of your manuscript, or something that doesn't use the keyboard in the ordinary way, you should type the escape character followed by an indication of what you want to do.

Note: Some computer terminals have a key marked '`ESC`', but that is *not* your escape character! It is a key that sends a special message to the operating system, so don't confuse it with what this manual calls "escape."

TEX allows any character to be used for escapes, but the "backslash" character '\' is usually adopted for this purpose, since backslashes are reasonably convenient to type and they are rarely needed in ordinary text. Things work out best when different TEX users do things consistently, so we shall escape via backslashes in all the examples of this manual.

Immediately after typing '\' (i.e., immediately after an escape character) you type a coded command telling TEX what you have in mind. Such commands are called *control sequences*. For example, you might type

```
\input MS
```

which (as we will see later) causes TEX to begin reading a file called '`MS.tex`'; the string of characters '`\input`' is a control sequence. Here's another example:

```
George P\'olya and Gabor Szeg\"o.
```

TEX converts this to 'George Pólya and Gabor Szegö.' There are two control sequences, `\'` and `\"`, here; these control sequences have been used to place accents over some of the letters.

Control sequences come in two flavors. The first kind, like `\input`, is called a *control word*; it consists of an escape character followed by one or more *letters*, followed by a space or by something besides a letter. (TEX has to know where the control sequence ends, so you must put a space after a control word if the next character is a letter. For example, if you type '`\inputMS`', TEX will naturally interpret this as a control word with seven letters.) In case you're wondering what a "letter" is, the answer is that TEX normally regards the 52 symbols `A...Z` and `a...z` as letters. The digits `0...9` are *not* considered to be letters, so they don't appear in control sequences of the first kind.

A control sequence of the other kind, like `\'`, is called a *control symbol*; it consists of the escape character followed by a single *nonletter*. In this case you don't need a space to separate the control sequence from a letter that follows, since control sequences of the second kind always have exactly one symbol after the escape character.

▸ **EXERCISE 3.1**

What are the control sequences in '`\I'm \exercise3.1\\!`'?

▸**EXERCISE 3.2**
We've seen that the input `P\'olya` yields 'Pólya'. Can you guess how the French words 'mathématique' and 'centimètre' should be specified?

When a space comes after a control word (an all-letter control sequence), it is ignored by TEX; i.e., it is not considered to be a "real" space belonging to the manuscript that is being typeset. But when a space comes after a control symbol, it's truly a space.

Now the question arises, what do you do if you actually *want* a space to appear after a control word? We will see later that TEX treats two or more consecutive spaces as a single space, so the answer is *not* going to be "type two spaces." The correct answer is to type "control space," namely

```
\␣
```

(the escape character followed by a blank space); TEX will treat this as a space that is not to be ignored. Notice that \␣ is a control sequence of the second kind, namely a control symbol, since there is a single nonletter (␣) following the escape character. Two consecutive spaces are considered to be equivalent to a single space, so further spaces immediately following \␣ will be ignored. But if you want to enter, say, three consecutive spaces into a manuscript you can type '\␣\␣\␣'. Incidentally, typists are often taught to put two spaces at the ends of sentences; but we will see later that TEX has its own way to produce extra space in such cases. Thus you needn't be consistent in the number of spaces you type.

Nonprinting control characters like ⟨return⟩ might follow an escape character, and these lead to distinct control sequences according to the rules. TEX is initially set up to treat \⟨return⟩ and \⟨tab⟩ the same as \␣ (control space); these special control sequences should probably not be redefined, because you can't see the difference between them when you look at them in a file.

It is usually unnecessary for you to use "control space," since control sequences aren't often needed at the ends of words. But here's an example that might shed some light on the matter: This manual itself has been typeset by TEX, and one of the things that occurs fairly often is the tricky logo 'TEX', which requires backspacing and lowering the E. There's a special control word

```
\TeX
```

that produces the half-dozen or so instructions necessary to typeset 'TEX'. When a phrase like 'TEX ignores spaces after control words.' is desired, the manuscript renders it as follows:

```
\TeX\ ignores spaces after control words.
```

Notice the extra \ following `\TeX`; this produces the control space that is necessary because TEX ignores spaces after control words. Without this extra \, the result would have been

TEXignores spaces after control words.

On the other hand, you can't simply put `\` after `\TeX` in all contexts. For example, consider the phrase

```
the logo `\TeX'.
```

In this case an extra backslash doesn't work at all; in fact, you get a curious result if you type

```
the logo `\TeX\'.
```

Can you guess what happens? Answer: The `\'` is a control sequence denoting an acute accent, as in our `P\'olya` example above; the effect is therefore to put an accent over the next nonblank character, which happens to be a period. In other words, you get an accented period, and the result is

the logo 'TEX.'

Computers are good at following instructions, but not at reading your mind.

TEX understands about 900 control sequences as part of its built-in vocabulary, and all of them are explained in this manual somewhere. But you needn't worry about learning so many different things, because you won't really be needing very many of them unless you are faced with unusually complicated copy. Furthermore, the ones you do need to learn actually fall into relatively few categories, so they can be assimilated without great difficulty. For example, many of the control sequences are simply the names of special characters used in math formulas; you type '`\pi`' to get 'π', '`\Pi`' to get 'Π', '`\aleph`' to get '$\aleph$', '`\infty`' to get '∞', '`\le`' to get '$\le$', '`\ge`' to get '$\ge$', '`\ne`' to get '$\ne$', '`\oplus`' to get '$\oplus$', '`\otimes`' to get '$\otimes$'. Appendix F contains several tables of such symbols.

There's no built-in relationship between uppercase and lowercase letters in control sequence names. For example, '`\pi`' and '`\Pi`' and '`\PI`' and '`\pI`' are four different control words.

The 900 or so control sequences that were just mentioned actually aren't the whole story, because it's easy to define more. For example, if you want to substitute your own favorite names for math symbols, so that you can remember them better, you're free to go right ahead and do it; Chapter 20 explains how.

About 300 of TEX's control sequences are called *primitive*; these are the low-level atomic operations that are not decomposable into simpler functions. All other control sequences are defined, ultimately, in terms of the primitive ones. For example, `\input` is a primitive operation, but `\'` and `\"` are not; the latter are defined in terms of an `\accent` primitive.

People hardly ever use TEX's primitive control sequences in their manuscripts, because the primitives are ... well ... so *primitive*. You have to type a lot of instructions when you are trying to make TEX do low-level things; this takes time and invites mistakes. It is generally better to make use of higher-level control sequences that state what functions are desired, instead of typing out the way to achieve each function each time. The higher-level control sequences

need to be defined only once in terms of primitives. For example, `\TeX` is a control sequence that means "typeset the TEX logo"; `\'` is a control sequence that means "put an acute accent over the next character"; and both of these control sequences might require different combinations of primitives when the style of type changes. If TEX's logo were to change, the author would simply have to change one definition, and the changes would appear automatically wherever they were needed. By contrast, an enormous amount of work would be necessary to change the logo if it were specified as a sequence of primitives each time.

At a still higher level, there are control sequences that govern the overall format of a document. For example, in the present book the author typed '`\exercise`' just before stating each exercise; this `\exercise` command was programmed to make TEX do all of the following things:

- compute the exercise number (e.g., '3.2' for the second exercise in Chapter 3);
- typeset '▸**EXERCISE 3.2**' with the appropriate typefaces, on a line by itself, and with the triangle sticking out in the left margin;
- leave a little extra space just before that line, or begin a new page at that line if appropriate;
- prohibit beginning a new page just after that line;
- suppress indentation on the following line.

It is obviously advantageous to avoid typing all of these individual instructions each time. And since the manual is entirely described in terms of high-level control sequences, it could be printed in a radically different format simply by changing a dozen or so definitions.

How can a person distinguish a TEX primitive from a control sequence that has been defined at a higher level? There are two ways: (1) The index to this manual lists all of the control sequences that are discussed, and each primitive is marked with an asterisk. (2) You can display the meaning of a control sequence while running TEX. If you type '`\show\cs`' where `\cs` is any control sequence, TEX will respond with its current meaning. For example, '`\show\input`' results in '`> \input=\input.`', because `\input` is primitive. On the other hand, '`\show\thinspace`' yields

```
> \thinspace=macro:
->\kern .16667em .
```

This means that `\thinspace` has been defined as an abbreviation for '`\kern .16667em `'. By typing '`\show\kern`' you can verify that `\kern` is primitive. The results of `\show` appear on your terminal and in the log file that you get after running TEX.

▸**EXERCISE 3.3**
Which of the control sequences `\␣` and `\⟨return⟩` is primitive?

In the following chapters we shall frequently discuss "plain TEX" format, which is a set of about 600 basic control sequences that are defined in Appendix B. These control sequences, together with the 300 or so primitives,

are usually present when TEX begins to process a manuscript; that is why TEX claims to know roughly 900 control sequences when it starts. We shall see how plain TEX can be used to create documents in a flexible format that meets many people's needs, using some typefaces that come with the TEX system. However, you should keep in mind that plain TEX is only one of countless formats that can be designed on top of TEX's primitives; if you want some other format, it will usually be possible to adapt TEX so that it will handle whatever you have in mind. The best way to learn is probably to start with plain TEX and to change its definitions, little by little, as you gain more experience.

Appendix E contains examples of formats that can be added to Appendix B for special applications; for example, there is a set of definitions suitable for business correspondence. A complete specification of the format used to typeset this manual also appears in Appendix E. Thus, if your goal is to learn how to design TEX formats, you will probably want to study Appendix E while mastering Appendix B. After you have become skilled in the lore of control-sequence definition, you will probably have developed some formats that other people will want to use; you should then write a supplement to this manual, explaining your style rules.

The main point of these remarks, as far as novice TEX users are concerned, is that it is indeed possible to define nonstandard TEX control sequences. When this manual says that something is part of "plain TEX," it means that TEX doesn't insist on doing things exactly that way; a person could change the rules by changing one or more of the definitions in Appendix B. But you can safely rely on the control sequences of plain TEX until you become an experienced TEXnical typist.

▸ **EXERCISE 3.4**

How many different control sequences of length 2 (including the escape character) are possible? How many of length 3?

Syllables govern the world.
— JOHN SELDEN, *Table Talk* (1689)

I claim not to have controlled events,
but confess plainly that events have controlled me.
— ABRAHAM LINCOLN (1864)

4

Fonts of Type

Occasionally you will want to change from one typeface to another, for example if you wish to be **bold** or to *emphasize* something. TEX deals with sets of up to 256 characters called "fonts" of type, and control sequences are used to select a particular font. For example, you could specify the last few words of the first sentence above in the following way, using the plain TEX format of Appendix B:

```
to be \bf bold \rm or to \sl emphasize \rm something.
```

Plain TEX provides the following control sequences for changing fonts:

`\rm` switches to the normal "roman" typeface:	Roman
`\sl` switches to a slanted roman typeface:	*Slanted*
`\it` switches to italic style:	*Italic*
`\tt` switches to a typewriter-like face:	`Typewriter`
`\bf` switches to an extended boldface style:	**Bold**

At the beginning of a run you get roman type (`\rm`) unless you specify otherwise.

Notice that two of these faces have an "oblique" slope for emphasis: *Slanted type is essentially the same as roman, but the letters are slightly skewed, while the letters in italic type are drawn in a different style.* (You can perhaps best appreciate the difference between the roman and italic styles by contemplating letters that are in an unslanted italic face.) Typographic conventions are presently in a state of transition, because new technology has made it possible to do things that used to be prohibitively expensive; people are wrestling with the question of how much to use their new-found typographic freedom. Slanted roman type was introduced in the 1930s, but it first became widely used as an alternative to the conventional italic during the late 1970s. It can be beneficial in mathematical texts, since slanted letters are distinguishable from the italic letters in math formulas. The double use of italic type for two different purposes—for example, when statements of theorems are italicized as well as the names of variables in those theorems—has led to some confusion, which can now be avoided with slanted type. People are not generally agreed about the relative merits of slanted versus italic, but slanted type is rapidly becoming a favorite for the titles of books and journals in bibliographies.

Special fonts are effective for emphasis, but not for sustained reading; your eyes would tire if long portions of this manual were entirely set in a bold or slanted or italic face. Therefore roman type accounts for the bulk of most typeset material. But it's a nuisance to say '`\rm`' every time you want to go back to the roman style, so TEX provides an easier way to do it, using "curly brace" symbols: You can switch fonts inside the special symbols `{` and `}`, without affecting the fonts outside. For example, the displayed phrase at the beginning of this chapter is usually rendered

```
to be {\bf bold} or to {\sl emphasize} something.
```

This is a special case of the general idea of "grouping" that we shall discuss in the next chapter. It's best to forget about the first way of changing fonts, and

to use grouping instead; then your TEX manuscripts will look more natural, and you'll probably never* have to type '`\rm`'.

▸**EXERCISE 4.1**
Explain how to type the bibliographic reference 'Ulrich Dieter, *Journal für die reine und angewandte Mathematik* **201** (1959), 37–70.' [Use grouping.]

We have glossed over an important aspect of quality in the preceding discussion. Look, for example, at the *italicized* and *slanted* words in this sentence. Since italic and slanted styles slope to the right, the d's stick into the spaces that separate these words from the roman type that follows; as a result, the spaces appear to be too skimpy, although they are correct at the base of the letters. To equalize the effective white space, TEX allows you to put the special control sequence '`\/`' just before switching back to unslanted letters. When you type

```
{\it italicized\/} and {\sl slanted\/} words
```

you get *italicized* and *slanted* words that look better. The '`\/`' tells TEX to add an "*italic correction*" to the previous letter, depending on that letter; this correction is about four times as much for an '*f*' as for a '*c*', in a typical italic font.

Sometimes the italic correction is not desirable, because other factors take up the visual slack. The standard rule of thumb is to use `\/` just before switching from slanted or italic to roman or bold, unless the next character is a period or comma. For example, type

```
{\it italics\/} for {\it emphasis}.
```

Old manuals of style say that the punctuation after a word should be in the *same* font as that *word;* but an italic semicolon often looks wrong, so this convention is changing. When an italicized word occurs just before a semicolon, the author recommends typing '`{\it word\/};`'.

▸**EXERCISE 4.2**
Explain how to typeset a roman *word in the midst of an italicized sentence.*


Every letter of every font has an italic correction, which you can bring to life by typing `\/`. The correction is usually zero in unslanted styles, but there are exceptions: To typeset a bold '**f**' in quotes, you should say `a bold '{\bf f\/}'`, lest you get a bold '**f**'.


▸**EXERCISE 4.3**
Define a control sequence `\ic` such that '`\ic c`' puts the italic correction of character *c* into TEX's register `\dimen0`.


The primitive control sequence `\nullfont` stands for a font that has no characters. This font is always present, in case you haven't specified any others.

---

* Well . . . , hardly ever.

Fonts vary in size as well as in shape. For example, the font you are now reading is called a "10-point" font, because certain features of its design are 10 points apart, when measured in printers' units. (We will study the point system later; for now, it should suffice to point out that the parentheses around this sentence are exactly 10 points tall—and the em-dash is just 10 points wide.) The "dangerous bend" sections of this manual are set in 9-point type, the footnotes in 8-point, subscripts in 7-point or 6-point, sub-subscripts in 5-point.

Each font used in a TEX manuscript is associated with a control sequence; for example, the 10-point font in this paragraph is called `\tenrm`, and the corresponding 9-point font is called `\ninerm`. The slanted fonts that match `\tenrm` and `\ninerm` are called `\tensl` and `\ninesl`. These control sequences are not built into TEX, nor are they the actual names of the fonts; TEX users are just supposed to make up convenient names, whenever new fonts are introduced into a manuscript. Such control sequences are used to change typefaces.

When fonts of different sizes are used simultaneously, TEX will line the letters up according to their "baselines." For example, if you type

```
\tenrm smaller \ninerm and smaller
\eightrm and smaller \sevenrm and smaller
\sixrm and smaller \fiverm and smaller \tenrm
```

the result is smaller and smaller and smaller and smaller and smaller. Of course this is something that authors and readers aren't accustomed to, because printers couldn't do such things with traditional lead types. Perhaps poets who wish to speak in a still small voice will cause future books to make use of frequent font variations, but nowadays it's only an occasional font freak (like the author of this manual) who likes such experiments. One should not get too carried away by the prospect of font switching unless there is good reason.

An alert reader might well be confused at this point because we started out this chapter by saying that '`\rm`' is the command that switches to roman type, but later on we said that '`\tenrm`' is the way to do it. The truth is that both ways work. But it has become customary to set things up so that `\rm` means "switch to roman type in the current size" while `\tenrm` means "switch to roman type in the 10-point size." In plain TEX format, nothing but 10-point fonts are provided, so `\rm` will always get you `\tenrm`; but in more complicated formats the meaning of `\rm` will change in different parts of the manuscript. For example, in the format used by the author to typeset this manual, there's a control sequence '`\tenpoint`' that causes `\rm` to mean `\tenrm`, `\sl` to mean `\tensl`, and so on, while '`\ninepoint`' changes the definitions so that `\rm` means `\ninerm`, etc. There's another control sequence used to introduce the quotations at the end of each chapter; when the quotations are typed, `\rm` and `\sl` temporarily stand for 8-point unslanted sans-serif type and *8-point slanted sans-serif type*, respectively. This device of constantly redefining the abbreviations `\rm` and `\sl`, behind the scenes, frees the typist from the need to remember what size or style of type is currently being used.

▸**EXERCISE 4.4**
Why do you think the author chose the names '`\tenpoint`' and '`\tenrm`', etc., instead of '`\10point`' and '`\10rm`'?

▸**EXERCISE 4.5**
Suppose that you have typed a manuscript using slanted type for emphasis, but your editor suddenly tells you to change all the slanted to italic. What's an easy way to do this?

Each font has an external name that identifies it with respect to all other fonts in a particular library. For example, the font in this sentence is called '`cmr9`', which is an abbreviation for "Computer Modern Roman 9 point." In order to prepare TEX for using this font, the command

```
\font\ninerm=cmr9
```

appears in Appendix E. In general you say '`\font\cs=`⟨external font name⟩' to load the information about a particular font into TEX's memory; afterwards the control sequence `\cs` will select that font for typesetting. Plain TEX makes only sixteen fonts available initially (see Appendix B and Appendix F), but you can use `\font` to access anything that exists in your system's font library.

It is often possible to use a font at several different sizes, by magnifying or shrinking the character images. Each font has a so-called design size, which reflects the size it normally has by default; for example, the design size of `cmr9` is 9 points. But on many systems there is also a range of sizes at which you can use a particular font, by scaling its dimensions up or down. To load a scaled font into TEX's memory, you simply say '`\font\cs=`⟨external font name⟩ `at` ⟨desired size⟩'. For example, the command

```
\font\magnifiedfiverm=cmr5 at 10pt
```

brings in 5-point Computer Modern Roman at twice its normal size. (Caution: Before using this '`at`' feature, you should check to make sure that your typesetter supports the font at the size in question; TEX will accept any ⟨desired size⟩ that is positive and less than 2048 points, but the final output will not be right unless the scaled font really is available on your printing device.)

What's the difference between `cmr5 at 10pt` and the normal 10-point font, `cmr10`? Plenty; a well-designed font will be drawn differently at different point sizes, and the letters will often have different relative heights and widths, in order to enhance readability.

Ten-point type is different from magnified five-point type.

It is usually best to scale fonts only slightly with respect to their design size, unless the final product is going to be photographically reduced after TEX has finished with it, or unless you are trying for an unusual effect.

Another way to magnify a font is to specify a scale factor that is relative to the design size. For example, the command

```
\font\magnifiedfiverm=cmr5 scaled 2000
```

is another way to bring in the font `cmr5` at double size. The scale factor is specified as an integer that represents a magnification ratio times 1000. Thus, a scale factor of 1200 specifies magnification by 1.2, etc.

▸ **EXERCISE 4.6**
State two ways to load font `cmr10` into TEX's memory at half its normal size.

At many computer centers it has proved convenient to supply fonts at magnifications that grow in geometric ratios—something like equal-tempered tuning on a piano. The idea is to have all fonts available at their true size as well as at magnifications 1.2 and 1.44 (which is 1.2×1.2); perhaps also at magnification 1.728 ($= 1.2 \times 1.2 \times 1.2$) and even higher. Then you can magnify an entire document by 1.2 or 1.44 and still stay within the set of available fonts. Plain TEX provides the abbreviations `\magstep0` for a scale factor of 1000, `\magstep1` for a scaled factor of 1200, `\magstep2` for 1440, and so on up to `\magstep5`. You say, for example,

```
\font\bigtenrm=cmr10 scaled\magstep2
```

to load font `cmr10` at 1.2×1.2 times its normal size.

"This is `cmr10` at normal size (`\magstep0`)."

"This is `cmr10` scaled once by 1.2 (`\magstep1`)."

"This is `cmr10` scaled twice by 1.2 (`\magstep2`)."

(Notice that a little magnification goes a long way.) There's also `\magstephalf`, which magnifies by $\sqrt{1.2}$, i.e., halfway between steps 0 and 1.

Chapter 10 explains how to apply magnification to an entire document, over and above any magnification that has been specified when fonts are loaded. For example, if you have loaded a font that is scaled by `\magstep1` and if you also specify `\magnification=\magstep2`, the actual font used for printing will be scaled by `\magstep3`. Similarly, if you load a font scaled by `\magstephalf` and if you also say `\magnification=\magstephalf`, the printed results will be scaled by `\magstep1`.

Type faces—like people's faces—have distinctive features indicating aspects of character.

— MARSHALL LEE, *Bookmaking* (1965)

This was the Noblest Roman of them all.

— WILLIAM SHAKESPEARE, *The Tragedie of Julius Cæsar* (1599)

5
Grouping

Every once in a while it is necessary to treat part of a manuscript as a unit, so you need to indicate somehow where that part begins and where it ends. For this purpose TEX gives special interpretation to two "grouping characters," which (like the escape character) are treated differently from the normal symbols that you type. We assume in this manual that `{` and `}` are the grouping characters, since they are the ones used in plain TEX.

We saw examples of grouping in the previous chapter, where it was mentioned that font changes inside a group do not affect the fonts in force outside. The same principle applies to almost anything else that is defined inside a group, as we will see later; for example, if you define a control sequence within some group, that definition will disappear when the group ends. In this way you can conveniently instruct TEX to do something unusual, by changing its normal conventions temporarily inside of a group; since the changes are invisible from outside the group, there is no need to worry about messing up the rest of a manuscript by forgetting to restore the normal conventions when the unusual construction has been finished. Computer scientists have a name for this aspect of grouping, because it's an important aspect of programming languages in general; they call it "block structure," and definitions that are in force only within a group are said to be "local" to that group.

You might want to use grouping even when you don't care about block structure, just to have better control over spacing. For example, let's consider once more the control sequence `\TeX` that produces the logo 'TEX' in this manual: We observed in Chapter 3 that a blank space after this control sequence will be gobbled up unless one types '`\TeX\ `', yet it is a mistake to say '`\TeX\`' when the following character is not a blank space. Well, in *all* cases it would be correct to specify the simple group

```
{\TeX}
```

whether or not the following character is a space, because the `}` stops TEX from absorbing an optional space into `\TeX`. This might come in handy when you're using a text editor (e.g., when replacing all occurrences of a particular word by a control sequence). Another thing you could do is type

```
\TeX{}
```

using an *empty* group for the same purpose: The '`{}`' here is a group of no characters, so it produces no output, but it does have the effect of stopping TEX from skipping blanks.

▸ **EXERCISE 5.1**
Sometimes you run into a rare word like 'shelfful' that looks better as 'shelffful' without the 'ff' ligature. How can you fool TEX into thinking that there aren't two consecutive f's in such a word?

▸ **EXERCISE 5.2**
Explain how to get three blank spaces in a row without using '`\␣`'.

But TEX also uses grouping for another, quite different, purpose, namely to determine how much of your text is to be governed by certain control sequences. For example, if you want to center something on a line you can type

```
\centerline{This information should be centered.}
```

using the control sequence `\centerline` defined in plain TEX format.

Grouping is used in quite a few of TEX's more intricate instructions; and it's possible to have groups within groups within groups, as you can see by glancing at Appendix B. Complex grouping is generally unnecessary, however, in ordinary manuscripts, so you needn't worry about it. Just don't forget to finish each group that you've started, because a lost '`}`' might cause trouble.

Here's an example of two groups, one nested inside the other:

```
\centerline{This information should be {\it centered}.}
```

As you might expect, TEX will produce a centered line that also contains italics:

This information should be *centered*.

But let's look at the example more closely: '`\centerline`' appears outside the curly braces, while '`\it`' appears inside. Why are the two cases different? And how can a beginner learn to remember which is which? Answer: `\centerline` is a control sequence that applies only to the very next thing that follows, so you want to put braces around the text that is to be centered (unless that text consists of a single symbol or control sequence). For example, to center the TEX logo on a line, it would suffice to type '`\centerline\TeX`', but to center the phrase 'TEX has groups' you need braces: '`\centerline{\TeX\ has groups}`'. On the other hand, `\it` is a control sequence that simply means "change the current font"; it acts without looking ahead, so it affects *everything* that follows, at least potentially. The braces surround `\it` in order to confine the font change to a local region.

In other words, the two sets of braces in this example actually have different functions: One serves to treat several words of the text as if they were a single object, while the other provides local block structure.

▸**EXERCISE 5.3**

What do you think happens if you type the following:

```
\centerline{This information should be {centered}.}
\centerline So should this.
```

▸**EXERCISE 5.4**

And how about this one?

```
\centerline{This information should be \it centered.}
```

▸**EXERCISE 5.5**

Define a control sequence `\ital` so that a user could type '`\ital{text}`' instead of '`{\it text\/}`'. Discuss the pros and cons of `\ital` versus `\it`.


Subsequent chapters describe many primitive operations of TEX for which the locality of grouping is important. For example, if one of TEX's internal parameters is changed within a group, the previous contents of that parameter will be restored when the group ends. Sometimes, however, it's desirable to make a definition that transcends its current group. This effect can be obtained by prefixing '`\global`' to the definition. For example, TEX keeps the current page number in a register called `\count0`, and the routine that outputs a page wants to increase the page number. Output routines are always protected by enclosing them in groups, so that they do not inadvertently mess up the rest of TEX; but the change to `\count0` would disappear if it were kept local to the output group. The command

```
\global\advance\count0 by 1
```

solves the problem; it increases `\count0` and makes this value stick around at the end of the output routine. In general, `\global` makes the immediately following definition pertain to all existing groups, not just to the innermost one.


▸ **EXERCISE 5.6**
If you think you understand local and global definitions, here's a little test to make sure: Suppose `\c` stands for '`\count1=`', `\g` stands for '`\global\count1=`', and `\s` stands for '`\showthe\count1`'. What values will be shown?

```
{\c1\s\g2{\s\c3\s\g4\s\c5\s}\s\c6\s}\s
```


Another way to obtain block structure with TEX is to use the primitives `\begingroup` and `\endgroup`. These control sequences make it easy to begin a group within one control sequence and end it within another. The text that TEX actually executes, after control sequences have been expanded, must have properly nested groups, i.e., groups that don't overlap. For example,

```
{ \begingroup } \endgroup
```

is not legitimate.


▸ **EXERCISE 5.7**
Define control sequences `\beginthe`⟨block name⟩ and `\endthe`⟨block name⟩ that provide a "named" block structure. In other words,

```
\beginthe{beguine}\beginthe{waltz}\endthe{waltz}\endthe{beguine}
```

should be permissible, but not

```
\beginthe{beguine}\beginthe{waltz}\endthe{beguine}\endthe{waltz}.
```

I have had recourse to varieties of type,
and to braces.

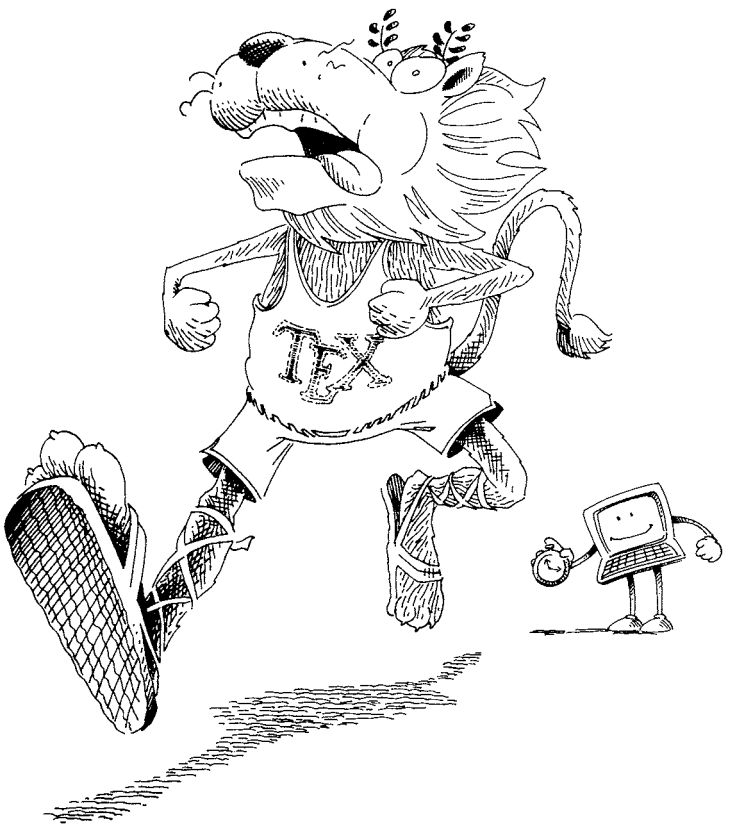
— JAMES MUIRHEAD, *The Institutes of Gaius* (1880)

An encounter group is a gathering, for a few hours or a few days,
of twelve or eighteen personable, responsible, certifiably normal
and temporarily smelly people.

— JANE HOWARD, *Please Touch* (1970)

6

Running TEX



The best way to learn how to use TEX is to use it. Thus, it's high time for you to sit down at a computer terminal and interact with the TEX system, trying things out to see what happens. Here are some small but complete examples suggested for your first encounter.

Caution: This chapter is rather a long one. Why don't you stop reading now, and come back fresh tomorrow?

OK, let's suppose that you're rested and excited about having a trial run of TEX. Step-by-step instructions for using it appear in this chapter. First do this: Go to the lab where the graphic output device is, since you will be wanting to see the output that you get—it won't really be satisfactory to run TEX from a remote location, where you can't hold the generated documents in your own hands. Then log in; and start TEX. (You may have to ask somebody how to do this on your local computer. Usually the operating system prompts you for a command and you type '`tex`' or '`run tex`' or something like that.)

When you're successful, TEX will welcome you with a message such as

```
This is TeX, Version 3.141 (preloaded format=plain 89.7.15)
**
```

The '`**`' is TEX's way of asking you for an input file name.

Now type '`\relax`' (including the backslash), and ⟨return⟩ (or whatever is used to mean "end-of-line" on your terminal). TEX is all geared up for action, ready to read a long manuscript; but you're saying that it's all right to take things easy, since this is going to be a real simple run. In fact, `\relax` is a control sequence that means "do nothing."

The machine will type another asterisk at you. This time type something like '`Hello?`' and wait for another asterisk. Finally type '`\end`', and stand back to see what happens.

TEX should respond with '`[1]`' (meaning that it has finished page 1 of your output); then the program will halt, probably with some indication that it has created a file called '`texput.dvi`'. (TEX uses the name `texput` for its output when you haven't specified any better name in your first line of input; and `dvi` stands for "device independent," since `texput.dvi` is capable of being printed on almost any kind of typographic output device.)

Now you're going to need some help again from your friendly local computer hackers. They will tell you how to produce hardcopy from `texput.dvi`. And when you see the hardcopy—Oh, glorious day!—you will see a magnificent 'Hello?' and the page number '1' at the bottom. Congratulations on your first masterpiece of fine printing.

The point is, you understand now how to get something through the whole cycle. It only remains to do the same thing with a somewhat longer document. So our next experiment will be to work from a file instead of typing the input online.

Use your favorite text editor to create a file called `story.tex` that contains the following 18 lines of text (no more, no less):

```
1 \hrule
2 \vskip 1in
3 \centerline{\bf A SHORT STORY}
4 \vskip 6pt
5 \centerline{\sl by A. U. Thor}
6 \vskip .5cm
7 Once upon a time, in a distant
8   galaxy called \"O\"o\c c,
9 there lived a computer
10 named R.~J. Drofnats.
11
12 Mr.~Drofnats---or ``R. J.,'' as
13 he preferred to be called---
14 was happiest when he was at work
15 typesetting beautiful documents.
16 \vskip 1in
17 \hrule
18 \vfill\eject
```

(Don't type the numbers at the left of these lines, of course; they are present only for reference.) This example is a bit long, and more than a bit silly; but it's no trick for a good typist like you and it will give you some worthwhile experience, so do it. For your own good. And think about what you're typing, as you go; the example introduces a few important features of TEX that you can learn as you're making the file.

Here is a brief explanation of what you have just typed: Lines 1 and 17 put a horizontal rule (a thin line) across the page. Lines 2 and 16 skip past one inch of space; '`\vskip`' means "vertical skip," and this extra space will separate the horizontal rules from the rest of the copy. Lines 3 and 5 produce the title and the author name, centered, in boldface and in slanted type. Lines 4 and 6 put extra white space between those lines and their successors. (We shall discuss units of measure like '`6pt`' and '`.5cm`' in Chapter 10.)

The main bulk of the story appears on lines 7–15, and it consists of two paragraphs. The fact that line 11 is blank informs TEX that line 10 is the end of the first paragraph; and the '`\vskip`' on line 16 implies that the second paragraph ends on line 15, because vertical skips don't appear in paragraphs. Incidentally, this example seems to be quite full of TEX commands; but it is atypical in that respect, because it is so short and because it is supposed to be teaching things. Messy constructions like `\vskip` and `\centerline` can be expected at the very beginning of a manuscript, unless you're using a canned format, but they don't last long; most of the time you will find yourself typing straight text, with relatively few control sequences.

And now comes the good news, if you haven't used computer typesetting before: You don't have to worry about where to break lines in a paragraph (i.e., where to stop at the right margin and to begin a new line), because TEX will do that for you. Your manuscript file can contain long lines or short lines, or both; it doesn't matter. This is especially helpful when you make changes, since you don't have to retype anything except the words that changed. *Every time you begin a new line in your manuscript file it is essentially the same as typing a space.* When TEX has read an entire paragraph—in this case lines 7 to 11—it will try to break up the text so that each line of output, except the last, contains about the same amount of copy; and it will hyphenate words if necessary to keep the spacing consistent, but only as a last resort.

Line 8 contains the strange concoction

```
\"O\"o\c c
```

and you already know that `\"` stands for an umlaut accent. The `\c` stands for a "cedilla," so you will get 'Öoç' as the name of that distant galaxy.

The remaining text is simply a review of the conventions that we discussed long ago for dashes and quotation marks, except that the '`~`' signs in lines 10 and 12 are a new wrinkle. These are called *ties*, because they tie words together; i.e., TEX is supposed to treat '`~`' as a normal space but not to break between lines there. A good typist will use ties within names, as shown in our example; further discussion of ties appears in Chapter 14.

Finally, line 18 tells TEX to '`\vfill`', i.e., to fill the rest of the page with white space; and to '`\eject`' the page, i.e., to send it to the output file.

Now you're ready for Experiment 2: Get TEX going again. This time when the machine says '`**`' you should answer '`story`', since that is the name of the file where your input resides. (The file could also be called by its full name '`story.tex`', but TEX automatically supplies the suffix '`.tex`' if no suffix has been specified.)

You might wonder why the first prompt was '`**`', while the subsequent ones are '`*`'; the reason is simply that the first thing you type to TEX is slightly different from the rest: If the first character of your response to '`**`' is not a backslash, TEX automatically inserts '`\input`'. Thus you can usually run TEX by merely naming your input file. (Previous TEX systems required you to start by typing '`\input story`' instead of '`story`', and you can still do that; but most TEX users prefer to put all of their commands into a file instead of typing them online, so TEX now spares them the nuisance of starting out with `\input` each time.) Recall that in Experiment 1 you typed '`\relax`'; that started with a backslash, so `\input` was not implied.

There's actually another difference between '`**`' and '`*`': If the first character after `**` is an ampersand ('`&`'), TEX will replace its memory with a precomputed format file before proceeding. Thus, for example, you can type '`&plain \input story`' or even '`&plain story`' in response to '`**`', if you are running some version of TEX that might not have the plain format preloaded.

Incidentally, many systems allow you to invoke TEX by typing a one-liner like '`tex story`' instead of waiting for the '`**`'; similarly, '`tex \relax`' works for Experiment 1, and '`tex &plain story`' loads the plain format before inputting the `story` file. You might want to try this, to see if it works on your computer, or you might ask somebody if there's a similar shortcut.

As TEX begins to read your story file, it types '`(story.tex`', possibly with a version number for more precise identification, depending on your local operating system. Then it types '`[1]`', meaning that page 1 is done; and '`)`', meaning that the file has been entirely input.

TEX will now prompt you with '`*`', because the file did not contain '`\end`'. Enter `\end` into the computer now, and you should get a file `story.dvi` containing a typeset version of Thor's story. As in Experiment 1, you can proceed to convert `story.dvi` into hardcopy; go ahead and do that now. The typeset output won't be shown here, but you can see the results by doing the experiment personally. Please do so before reading on.

▸**EXERCISE 6.1**

Statistics show that only 7.43 of 10 people who read this manual actually type the `story.tex` file as recommended, but that those people learn TEX best. So why don't you join them?

▸**EXERCISE 6.2**

Look closely at the output of Experiment 2, and compare it to `story.tex`: If you followed the instructions carefully, you will notice a typographical error. What is it, and why did it sneak in?

With Experiment 2 under your belt, you know how to make a document from a file. The remaining experiments in this chapter are intended to help you cope with the inevitable anomalies that you will run into later; we will intentionally do things that will cause TEX to "squeak."

But before going on, it's best to fix the error revealed by the previous output (see exercise 6.2): Line 13 of the `story.tex` file should be changed to

```
he preferred to be called---% error has been fixed!
```

The '`%`' sign here is a feature of plain TEX that we haven't discussed before: It effectively terminates a line of your input file, without introducing the blank space that TEX ordinarily inserts when moving to the next line of input. Furthermore, TEX ignores everything that you type following a `%`, up to the end of that line in the file; you can therefore put comments into your manuscript, knowing that the comments are for your eyes only.

Experiment 3 will be to make TEX work harder, by asking it to set the story in narrower and narrower columns. Here's how: After starting the program, type

```
\hsize=4in \input story
```

in response to the '`**`'. This means, "Set the story in a 4-inch column." More precisely, `\hsize` is a primitive of TEX that specifies the horizontal size, i.e., the width of each normal line in the output when a paragraph is being typeset; and `\input` is a primitive that causes TEX to read the specified file. Thus, you are instructing the machine to change the normal setting of `\hsize` that was defined by plain TEX, and then to process `story.tex` under this modification.

TEX should respond by typing something like '`(story.tex [1])`' as before, followed by '`*`'. Now you should type

```
\hsize=3in \input story
```

and, after TEX says '`(story.tex [2])`' asking for more, type three more lines

```
\hsize=2.5in \input story
\hsize=2in \input story
\end
```

to complete this four-page experiment.

Don't be alarmed when TEX screams '`Overfull \hbox`' several times as it works at the 2-inch size; that's what was supposed to go wrong during Experiment 3. There simply is no good way to break the given paragraphs into lines that are exactly two inches wide, without making the spaces between words come out too large or too small. Plain TEX has been set up to ensure rather strict tolerances on all of the lines it produces:

> You don't get spaces between words narrower than this, and
> you don't get spaces between words wider than this.

If there's no way to meet these restrictions, you get an overfull box. And with the overfull box you also get (1) a warning message, printed on your terminal, and (2) a big black bar inserted at the right of the offending box, in your output. (Look at page 4 of the output from Experiment 3; the overfull boxes should stick out like sore thumbs. On the other hand, pages 1–3 should be perfect.)

Of course you don't want overfull boxes in your output, so TEX provides several ways to remove them; that will be the subject of our Experiment 4. But first let's look more closely at the results of Experiment 3, since TEX reported some potentially valuable information when it was forced to make those boxes too full; you should learn how to read this data:

```
Overfull \hbox (0.98807pt too wide) in paragraph at lines 7--11
\tenrm tant galaxy called []O^^?o^^Xc, there lived|
Overfull \hbox (0.4325pt too wide) in paragraph at lines 7--11
\tenrm a com-puter named R. J. Drof-nats. |
Overfull \hbox (5.32132pt too wide) in paragraph at lines 12--16
\tenrm he pre-ferred to be called---was hap-|
```

Each overfull box is correlated with its location in your input file (e.g., the first two were generated when processing the paragraph on lines 7–11 of `story.tex`), and you also learn by how much the copy sticks out (e.g., 0.98807 points).

Notice that TEX also shows the contents of the overfull boxes in abbreviated form. For example, the last one has the words 'he preferred to be called—was hap-', set in font `\tenrm` (10-point roman type); the first one has a somewhat curious rendering of 'Ööç', because the accents appear in strange places within that font. In general, when you see '`[]`' in one of these messages, it stands either for the paragraph indentation or for some sort of complex construction; in this particular case it stands for an umlaut that has been raised up to cover an 'O'.

▸ **EXERCISE 6.3**
Can you explain the '|' that appears after '`lived`' in that message?

▸ **EXERCISE 6.4**
Why is there a space before the '|' in '`Drof-nats. |`'?

You don't have to take out pencil and paper in order to write down the overfull box messages that you get before they disappear from view, since TEX always writes a "transcript" or "log file" that records what happened during each session. For example, you should now have a file called `story.log` containing the transcript of Experiment 3, as well as a file called `texput.log` containing the transcript of Experiment 1. (The transcript of Experiment 2 was probably overwritten when you did number 3.) Take a look at `story.log` now; you will see that the overfull box messages are accompanied not only by the abbreviated box contents, but also by some strange-looking data about hboxes and glue and kerns and such things. This data gives a precise description of what's in that overfull box; TEX wizards will find such listings important, if they are called upon to diagnose some mysterious error, and you too may want to understand TEX's internal code some day.

The abbreviated forms of overfull boxes show the hyphenations that TEX tried before it resorted to overfilling. The hyphenation algorithm, which is described in Appendix H, is excellent but not perfect; for example, you can see from the messages in `story.log` that TEX finds the hyphen in 'pre-ferred', and it can even hyphenate 'Drof-nats'. Yet it discovers no hyphen in 'galaxy', and every once in a while an overfull box problem can be cured simply by giving TEX a hint about how to hyphenate some word more completely. (We will see later that there are two ways to do this, either by inserting discretionary hyphens each time as in '`gal\-axy`', or by saying '`\hyphenation{gal-axy}`' once at the beginning of your manuscript.)

In the present example, hyphenation is not a problem, since TEX found and tried all the hyphens that could possibly have helped. The only way to get rid of the overfull boxes is to change the tolerance, i.e., to allow wider spaces between words. Indeed, the tolerance that plain TEX uses for wide lines is completely inappropriate for 2-inch columns; such narrow columns simply can't be achieved without loosening the constraints, unless you rewrite the copy to fit.

TEX assigns a numerical value called "badness" to each line that it sets, in order to assess the quality of the spacing. The exact rules for badness are

different for different fonts, and they will be discussed in Chapter 14; but here is the way badness works for the roman font of plain TEX:

The badness of this line is 100.	(very tight)
The badness of this line is 12.	(somewhat tight)
The badness of this line is 0.	(perfect)
The badness of this line is 12.	(somewhat loose)
The badness of this line is 200.	(loose)
The badness of this line is 1000.	(bad)
The badness of this line is 5000.	(awful)

Plain TEX normally stipulates that no line's badness should exceed 200; but in our case, the task would be impossible since

'tant galaxy called Ööç, there'	has badness 1521;
'he preferred to be called—was'	has badness 568.

So we turn now to Experiment 4, in which spacing variations that are more appropriate to narrow columns will be used.

Run TEX again, and begin this time by saying

```
\hsize=2in \tolerance=1600 \input story
```

so that lines with badness up to 1600 will be tolerated. Hurray! There are no overfull boxes this time. (But you do get a message about an *underfull* box, since TEX reports all boxes whose badness exceeds a certain threshold called `\hbadness`; plain TEX sets `\hbadness=1000`.) Now make TEX work still harder by trying

```
\hsize=1.5in \input story
```

(thus leaving the tolerance at 1600 but making the column width still skimpier). Alas, overfull boxes return; so try typing

```
\tolerance=10000 \input story
```

in order to see what happens. TEX treats 10000 as if it were "infinite" tolerance, allowing arbitrarily wide space; thus, a tolerance of 10000 will *never* produce an overfull box, unless something strange occurs like an unhyphenatable word that is wider than the column itself.

The underfull box that TEX produces in the 1.5-inch case is really bad; with such narrow limits, an occasional wide space is unavoidable. But try

```
\raggedright \input story
```

for a change. (This tells TEX not to worry about keeping the right margin straight, and to keep the spacing uniform within each line.) Finally, type

```
\hsize=.75in \input story
```

followed by '`\end`', to complete Experiment 4. This makes the columns almost impossibly narrow.

The output from this experiment will give you some feeling for the problem of breaking a paragraph into approximately equal lines. When the lines are relatively wide, TEX will almost always find a good solution. But otherwise you will have to figure out some compromise, and several options are possible. Suppose you want to ensure that no lines have badness exceeding 500. Then you could set `\tolerance` to some high number, and `\hbadness=500`; TEX would not produce overfull boxes, but it would warn you about the underfull ones. Or you could set `\tolerance=500`; then TEX might produce overfull boxes. If you really want to take corrective action, the second alternative is better, because you can look at an overfull box to see how much sticks out; it becomes graphically clear what remedies are possible. On the other hand, if you don't have time to fix bad spacing—if you just want to know how bad it is—then the first alternative is better, although it may require more computer time.

▸**EXERCISE 6.5**

When `\raggedright` has been specified, badness reflects the amount of space at the right margin, instead of the spacing between words. Devise an experiment by which you can easily determine what badness TEX assigns to each line, when the `story` is set ragged-right in 1.5-inch columns.

A parameter called `\hfuzz` allows you to ignore boxes that are only slightly overfull. For example, if you say `\hfuzz=1pt`, a box must stick out more than one point before it is considered erroneous. Plain TEX sets `\hfuzz=0.1pt`.

▸**EXERCISE 6.6**

Inspection of the output from Experiment 4, especially page 3, shows that with narrow columns it would be better to allow white space to appear before and after a dash, whenever other spaces in the same line are being stretched. Define a `\dash` macro that does this.

You were warned that this is a long chapter. But take heart: There's only one more experiment to do, and then you will know enough about TEX to run it fearlessly by yourself forever after. The only thing you are still missing is some information about how to cope with error messages—i.e., not just with warnings about things like overfull boxes, but with cases where TEX actually stops and asks you what to do next.

Error messages can be terrifying when you aren't prepared for them; but they can be fun when you have the right attitude. Just remember that you really haven't hurt the computer's feelings, and that nobody will hold the errors against you. Then you'll find that running TEX might actually be a creative experience instead of something to dread.

The first step in Experiment 5 is to plant two intentional mistakes in the `story.tex` file. Change line 3 to

```
\centerline{\bf A SHORT \ERROR STORY}
```

and change '`\vskip`' to '`\vship`' on line 2.

Now run TEX again; but instead of '`story`' type '`sorry`'. The computer should respond by saying that it can't find file `sorry.tex`, and it will ask you to try again. Just hit ⟨return⟩ this time; you'll see that you had better give the

name of a real file. So type '`story`' and wait for TEX to find one of the *faux pas* in that file.

Ah yes, the machine will soon stop,* after typing something like this:

```
! Undefined control sequence.
l.2 \vship
           1in
?
```

TEX begins its error messages with '`!`', and it shows what it was reading at the time of the error by displaying two lines of context. The top line of the pair (in this case '`\vship`') shows what TEX has looked at so far, and where it came from ('`l.2`', i.e., line number 2); the bottom line (in this case '`1in`') shows what TEX has yet to read.

The '`?`' that appears after the context display means that TEX wants advice about what to do next. If you've never seen an error message before, or if you've forgotten what sort of response is expected, you can type '`?`' now (go ahead and try it!); TEX will respond as follows:

```
Type <return> to proceed, S to scroll future error messages,
R to run without stopping, Q to run quietly,
I to insert something, E to edit your file,
1 or ... or 9 to ignore the next 1 to 9 tokens of input,
H for help, X to quit.
```

This is your menu of options. You may choose to continue in various ways:

1. Simply type ⟨return⟩. TEX will resume its processing, after attempting to recover from the error as best it can.
2. Type '`S`'. TEX will proceed without pausing for instructions if further errors arise. Subsequent error messages will flash by on your terminal, possibly faster than you can read them, and they will appear in your log file where you can scrutinize them at your leisure. Thus, '`S`' is sort of like typing ⟨return⟩ to every message.
3. Type '`R`'. This is like '`S`' but even stronger, since it tells TEX not to stop for any reason, not even if a file name can't be found.
4. Type '`Q`'. This is like '`R`' but even more so, since it tells TEX not only to proceed without stopping but also to suppress all further output to your terminal. It is a fast, but somewhat reckless, way to proceed (intended for running TEX with no operator in attendance).
5. Type '`I`', followed by some text that you want to insert. TEX will read this line of text before encountering what it would ordinarily see next. Lines inserted in this way are not assumed to end with a blank space.

---

* Some installations of TEX do not allow interaction. In such cases all you can do is look at the error messages in your log file, where they will appear together with the "help" information.

6. Type a small number (less than 100). TEX will delete this many characters and control sequences from whatever it is about to read next, and it will pause again to give you another chance to look things over.
7. Type '`H`'. This is what you should do now and whenever you are faced with an error message that you haven't seen for a while. TEX has two messages built in for each perceived error: a formal one and an informal one. The formal message is printed first (e.g., '`! Undefined control sequence.`'); the informal one is printed if you request more help by typing '`H`', and it also appears in your log file if you are scrolling error messages. The informal message tries to complement the formal one by explaining what TEX thinks the trouble is, and often by suggesting a strategy for recouping your losses.
8. Type '`X`'. This stands for "exit." It causes TEX to stop working on your job, after putting the finishing touches on your `log` file and on any pages that have already been output to your `dvi` file. The current (incomplete) page will not be output.
9. Type '`E`'. This is like '`X`', but it also prepares the computer to edit the file that TEX is currently reading, at the current position, so that you can conveniently make a change before trying again.

After you type '`H`' (or '`h`', which also works), you'll get a message that tries to explain that the control sequence just read by TEX (i.e., `\vship`) has never been assigned a meaning, and that you should either insert the correct control sequence or you should go on as if the offending one had not appeared.

In this case, therefore, your best bet is to type

```
I\vskip
```

(and ⟨return⟩), with no space after the '`I`'; this effectively replaces `\vship` by `\vskip`. (Do it.)

If you had simply typed ⟨return⟩ instead of inserting anything, TEX would have gone ahead and read '`1in`', which it would have regarded as part of a paragraph to be typeset. Alternatively, you could have typed '`3`'; that would have deleted '`1in`' from TEX's input. Or you could have typed '`X`' or '`E`' in order to correct the spelling error in your file. But it's usually best to try to detect as many errors as you can, each time you run TEX, since that increases your productivity while decreasing your computer bills. Chapter 27 explains more about the art of steering TEX through troubled text.

▸ **EXERCISE 6.7**
What would have happened if you had typed '`5`' after the `\vship` error?

You can control the level of interaction by giving commands in your file as well as online: The TEX primitives `\scrollmode`, `\nonstopmode`, and `\batchmode` correspond respectively to typing '`S`', '`R`', or '`Q`' in response to an error message, and `\errorstopmode` puts you back into the normal level of interaction. (Such changes are global, whether or not they appear inside a group.) Furthermore, many installations

have implemented a way to interrupt TEX while it is running; such an interruption causes the program to revert to `\errorstopmode`, after which it pauses and waits for further instructions.

What happens next in Experiment 5? TEX will hiccup on the other bug that we planted in the file. This time, however, the error message is more elaborate, since the context appears on six lines instead of two:

```
! Undefined control sequence.
<argument> \bf A SHORT \ERROR
                                  STORY
\centerline #1->\line {\hss #1
                                  \hss }
l.3 \centerline{\bf A SHORT \ERROR STORY}

?
```

You get multiline error messages like this when the error is detected while TEX is processing some higher-level commands—in this case, while it is trying to carry out `\centerline`, which is not a primitive operation (it is defined in plain TEX). At first, such error messages will appear to be complete nonsense to you, because much of what you see is low-level TEX code that you never wrote. But you can overcome this hangup by getting a feeling for the way TEX operates.

First notice that the context information always appears in pairs of lines. As before, the top line shows what TEX has just read (‘`\bf A SHORT \ERROR`’), then comes what it is about to read (‘`STORY`’). The next pair of lines shows the context of the first two; it indicates what TEX was doing just before it began to read the others. In this case, we see that TEX has just read ‘`#1`’, which is a special code that tells the machine to “read the first argument that is governed by the current control sequence”; i.e., “now read the stuff that `\centerline` is supposed to center on a line.” The definition in Appendix B says that `\centerline`, when applied to some text, is supposed to be carried out by sticking that text in place of the ‘`#1`’ in ‘`\line{\hss#1\hss}`’. So TEX is in the midst of this expansion of `\centerline`, as well as being in the midst of the text that is to be centered.

The bottom line shows how far TEX has gotten until now in the `story` file. (Actually the bottom line is blank in this example; what appears to be the bottom line is really the first of two lines of context, and it indicates that TEX has read everything including the ‘`}`’ in line 3 of the file.) Thus, the context in this error message gives us a glimpse of how TEX went about its business. First, it saw `\centerline` at the beginning of line 3. Then it looked at the definition of `\centerline` and noticed that `\centerline` takes an “argument,” i.e., that `\centerline` applies to the next character or control sequence or group that follows. So TEX read on, and filed ‘`\bf A SHORT \ERROR STORY`’ away as the argument to `\centerline`. Then it began to read the expansion, as defined in Appendix B. When it reached the `#1`, it began to read the argument it had saved. And when it reached `\ERROR`, it complained about an undefined control sequence.

▸**EXERCISE 6.8**

Why didn't TEX complain about `\ERROR` being undefined when `\ERROR` was first encountered, i.e., before reading '`STORY}`' on line 3?

When you get a multiline error message like this, the best clues about the source of the trouble are usually on the bottom line (since that is what you typed) and on the top line (since that is what triggered the error message). Somewhere in there you can usually spot the problem.

Where should you go from here? If you type '`H`' now, you'll just get the same help message about undefined control sequences that you saw before. If you respond by typing ⟨return⟩, TEX will go on and finish the run, producing output virtually identical to that in Experiment 2. In other words, the conventional responses won't teach you anything new. So type '`E`' now; this terminates the run and prepares the way for you to fix the erroneous file. (On some systems, TEX will actually start up the standard text editor, and you'll be positioned at the right place to delete '`\ERROR`'. On other systems, TEX will simply tell you to edit line 3 of file `story.tex`.)

When you edit `story.tex` again, you'll notice that line 2 still contains `\vship`; the fact that you told TEX to insert `\vskip` doesn't mean that your file has changed in any way. In general, you should correct all errors in the input file that were spotted by TEX during a run; the log file provides a handy way to remember what those errors were.

Well, this has indeed been a long chapter, so let's summarize what has been accomplished. By doing the five experiments you have learned at first hand (1) how to get a job printed via TEX; (2) how to make a file that contains a complete TEX manuscript; (3) how to change the plain TEX format to achieve columns with different widths; and (4) how to avoid panic when TEX issues stern warnings.

So you could now stop reading this book and go on to print a bunch of documents. It is better, however, to continue bearing with the author (after perhaps taking another rest), since you're just at the threshold of being able to do a lot more. And you ought to read Chapter 7 at least, because it warns you about certain symbols that you must not type unless you want TEX to do something special. While reading the remaining chapters it will, of course, be best for you to continue making trial runs, using experiments of your own design.

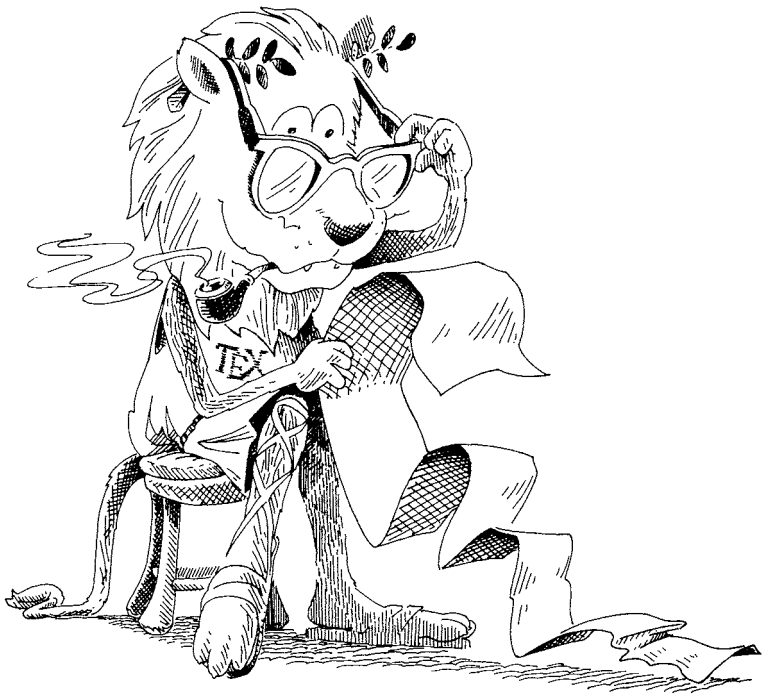
If you use TEX format packages designed by others, your error messages may involve many inscrutable two-line levels of macro context. By setting `\errorcontextlines=0` at the beginning of your file, you can reduce the amount of information that is reported; TEX will show only the top and bottom pairs of context lines together with up to `\errorcontextlines` additional two-line items. (If anything has thereby been omitted, you'll also see '`...`'.) Chances are good that you can spot the source of an error even when most of a large context has been suppressed; if not, you can say '`I\errorcontextlines=100 \oops`' and try again. (That will usually give you an undefined control sequence error and plenty of context.) Plain TEX sets `\errorcontextlines=5`.

What we have to learn to do we learn by doing.
— ARISTOTLE, *Ethica Nicomachea* II (c. 325 B.C.)

He may run who reads.
— HABAKKUK 2:2 (c. 600 B.C.)
He that runs may read.
— WILLIAM COWPER, *Tirocinium* (1785)

7

How TEX Reads What You Type



We observed in the previous chapter that an input manuscript is expressed in terms of "lines," but that these lines of input are essentially independent of the lines of output that will appear on the finished pages. Thus you can stop typing a line of input at any place that's convenient for you, as you prepare or edit a file. A few other related rules have also been mentioned:

- A ⟨return⟩ is like a space.
- Two spaces in a row count as one space.
- A blank line denotes the end of a paragraph.

Strictly speaking, these rules are contradictory: A blank line is obtained by typing ⟨return⟩ twice in a row, and this is different from typing two spaces in a row. Some day you might want to know the *real* rules. In this chapter and the next, we shall study the very first stage in the transition from input to output.

In the first place, it's wise to have a precise idea of what your keyboard sends to the machine. There are 256 characters that TEX might encounter at each step, in a file or in a line of text typed directly on your terminal. These 256 characters are classified into 16 categories numbered 0 to 15:

Category	*Meaning*	
0	Escape character	(\ in this manual)
1	Beginning of group	({ in this manual)
2	End of group	(} in this manual)
3	Math shift	($ in this manual)
4	Alignment tab	(& in this manual)
5	End of line	(⟨return⟩ in this manual)
6	Parameter	(# in this manual)
7	Superscript	(^ in this manual)
8	Subscript	(_ in this manual)
9	Ignored character	(⟨null⟩ in this manual)
10	Space	(␣ in this manual)
11	Letter	(A, . . . , Z and a, . . . , z)
12	Other character	(none of the above or below)
13	Active character	(~ in this manual)
14	Comment character	(% in this manual)
15	Invalid character	(⟨delete⟩ in this manual)

It's not necessary for you to learn these code numbers; the point is only that TEX responds to 16 different types of characters. At first this manual led you to believe that there were just two types—the escape character and the others—and then you were told about two more types, the grouping symbols { and }. In Chapter 6 you learned two more: ~ and %. Now you know that there are really 16. This is the whole truth of the matter; no more types remain to be revealed. The category code for any character can be changed at any time, but it is usually wise to stick to a particular scheme.