```
 ______   __                                __                                    ______
/      \ /  |                              /  |                                  /      \
$$$$$$  |$$ |____     ______     ______   _$$ |_      ______     ______         /$$$$$$  | _
$ |  $$/ $$      \   /      \   /      \ / $$   |    /      \   /      \        $$ \__$$/ /
$ |      $$$$$$$  |  $$$$$$  | /$$$$$$  |$$$$$$/    /$$$$$$  | /$$$$$$  |       $$      \ $$
$ |   __ $$ |  $$ |  /    $$ | $$ |  $$ |  $$ | __  $$    $$ | $$ |  $$/        $$$$$$$  | _
$ \__/  |$$ |  $$ | /$$$$$$$ | $$ |__$$ |  $$ |/  | $$$$$$$$/  $$ |             $$ \__$$ |/
$    $$/ $$ |  $$ | $$    $$ | $$    $$/   $$  $$/  $$         $$ |             $$    $$/ $$
$$$$$$/  $$/   $$/   $$$$$$$/  $$$$$$$/     $$$$/    $$$$$$$/  $$/               $$$$$$/
                               $$ |
                               $$ |
                               $$/
```

```
     __                                  ______               _______                      __
    /  |                                /      \             /       \                    /  |
_   _$$ |_      ______     ______      /$$$$$$  | __         $$$$$$$  | __    __  _______  _______   $$/  __
 \ / $$   |    /      \   /      \     $$ \__$$/ /  |        $$ |__$$ |/  |  /  |/       \ /       \ /  |/
  |$$$$$$/    /$$$$$$  | /$$$$$$  |    $$      \ $$/         $$    $$< $$ |  $$ |$$$$$$$  |$$$$$$$  |$$ |$$$$
$ |  $$ | __  $$    $$ | $$ |  $$/     $$$$$$$  | __         $$$$$$$  |$$ |  $$ |$$ |  $$ |$$ |  $$ |$$ |$$
$ |  $$ |/  | $$$$$$$$/  $$ |          $$ \__$$ |/  |        $$ |  $$ |$$ \__$$ |$$ |  $$ |$$ |  $$ |$$ |$$
$/   $$  $$/  $$         $$ |          $$    $$/ $$/         $$ |  $$ |$$    $$/ $$ |  $$ |$$ |  $$ |$$ |$$
/     $$$$/    $$$$$$$/  $$/            $$$$$$/              $$/   $$/  $$$$$$/  $$/   $$/ $$/   $$/ $$/ $$/
```

```
   __                               ________  ________  __    _
  /  |                             /        |/        |/  |  /
  $$/   _______    _______         $$$$$$$$/ $$$$$$$$/ $$ |  $$
\ /  |/        \  /       \           $$ |   $$ |__    $$  \/$$
 |$$ |$$$$$$$  |/$$$$$$$  |           $$ |   $$    |    $$  $$<
 |$$ |$$ |  $$ |$$ |  $$ |            $$ |   $$$$$/      $$$$
 |$$ |$$ |  $$ |$$ \__$$ |            $$ |   $$ |_____  $$ /$$
 |$$ |$$ |  $$ |$$    $$ |            $$ |   $$       |$$ |  $$
/ $$/ $$/   $$/  $$$$$$$ |            $$/    $$$$$$$$/ $$/   $$
                /  \__$$ |
                $$    $$/
```

— running the program — ** — relax — return — asterisk — end — [1] — texput — dvi — device — independent — story.tex — hrule — vskip — leading, see vskip — centerline — Thor — ” — c — Drofnats — vfill — eject — rule — paragraphs — blank line — empty line —

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 TEXfrom 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):

```
\hrule
\vskip 1in
\centerline{\bf A SHORT STORY}
\vskip 6pt
\centerline{\sl by A. U. Thor}
\vskip .5cm
Once upon a time, in a distant
  galaxy called \”O\”o\c c,
there lived a computer
named R.\\J. Drofnats.

Mr.\\Drofnats-or ‘‘R. J.,’’ as
he preferred to be called
was happiest when he was at work
```

— umlaut — cedilla — ties — tilde — vfill — eject — file names — ** — * — input — ampersand — format file — preloaded formats —

```
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; he 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 ou don’t have to retype anything except the words that changed. Every time ou 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 bout 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 ‘O¨ ¨o¸c’ as the name of that distant galaxy.

The remaining text is simply a review of the conventions that we dis cussed 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 xample; 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 precomuted 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 ight 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 utput 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 utput (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 efore, 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 computer named R. J. Drofnats.
|size input Overfull box
Overfull \hbox (5.32132pt too wide)
in paragraph at lines 12–16\tenrm he
pre-erred 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 'O¨ ¨o¸c', 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 contruction; 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 '`Drofnats. |`'?

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 he 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 pon 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, n 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 Ooc, 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` or 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` ollowed 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 elatively 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 ight produce overfull boxes. If you really want to take corrective action, the second lternative 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 ne 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

Breaking a paragraph hfuzz ash rror messages same of a real file. So type 'story' and wait for TEX to find one of the faux pas n that file.

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

```
! Undefined control sequence.
1.2 \vship
                1in
?
```

— ? — inserting text online — online interaction, see interaction — interacting with TeX — deleting tokens — help messages — scrollmode — nonstopmode — batchmode — errorstopmode —

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 dvice 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.
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 yping ‘`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.

* 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.

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 orrespond 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
interrupt argument centerline
\centerline #1->\line {\hss #1 STORY
\hss }
\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 un 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 f 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 est 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`.

Chapter 6. Running TEX

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)