Text Editing with ed

Editor's Note: Cross-references in the text refer to chapters in the companion book, UNIX: *The Complete Reference, Second Edition,* by Rosen, Host, Klee, Farber, and Rosinski.

The UNIX System provides a number of tools useful in creating and modifying text, and in formatting the text for presentation. This appendix provides an introduction to the **ed** text editor. **ed** is the original, UNIX System, line-oriented text editor. Before screen editors were developed, everyone used it, but today very few people use it as their primary editor. However, **ed** is not only of historical interest, since its commands are the basis of many other applications. The **ed** command syntax forms a *little language* that underlies other applications. In fact, even users who never use **ed** often use its command syntax, and there are good reasons why you should be familiar with **ed**: Many of the **ed** commands can be used in other editors; some actions (like global searches and replacements) that may otherwise be difficult are easy in **ed**'s language; and the syntax of **ed** commands is used in other programs. The easiest way to learn this little language is to learn how **ed** commands work. One important reason to learn **ed** is to master its language so that you can use tools that use this language, such as the UNIX stream editor, **sed** (for *s*tream *ed*itor). **sed** is a tool for filtering text files; it can perform almost all of the editing functions of **ed**. The **sed** command is discussed in more detail in Chapter 21.

Of course, today many application programs, including word processors, let you edit and format files with an easy-to-use interface. However, such programs do not allow you the flexibility of the basic UNIX System tools that can be used in many different ways for a wide variety of tasks. You will find it worthwhile to learn how to use **ed**, not so much for the editor itself, but rather because its syntax underlies so many UNIX utilities.

ed

To a new user, **ed** may seem especially terse and a little mysterious. Most of the commands are single letters or characters, and a short string of commands can make major changes in a text file. **ed** provides very little feedback to you. When you issue a command, **ed** performs the action you asked for, but it doesn't tell you what has happened. **ed** will simply execute the command and then wait for the next command. If you make a mistake, **ed** has only a single error or help message: the question mark (?). A "?" is displayed whenever the program doesn't understand something typed at the keyboard. It's up to you to figure out what is wrong. If you can't figure out what's wrong, the command

```
h
```

(help) will give a brief explanation. If you want a more detailed explanation of the errors than **ed** usually gives, you can use the **H** (big help) command to get explanatory messages instead of the "?"; simply type in

```
H
```

Background of ed

The terse, shorthand style of **ed** was a result of the nature of computing when the UNIX System was invented. In the 1970s, the computing environment was considerably different from what it is now. Minicomputers had little power and were shared by several users. Terminals were usually typewriter-like machines that printed on a roll of paper (the abbreviation *tty* stands for *t*ele*ty*pewriter), and the terminals were connected to minicomputers by low-speed (300 to 1200 bps) connections.

Computer processing power was a scarce resource. (The average technical person over 30 probably has more computing power available on his or her desk than was available in a whole college during his or her undergraduate days.) A program that used a single-letter dialog between machine and user (for example, "?"), that allowed simple commands to work on whole files, and that provided a syntax that allowed stringing together groups of commands made very efficient use of limited computer resources. The cost of this efficiency was the time needed to learn and become comfortable with this terse style of interacting with the computer.

The benefit to the user of a program such as **ed** is that complex things can be done within the editor (such as searching through a file, or substituting text) with a simple set of commands and a simple set of rules (syntax) for using the commands.

The efficiency and simplicity of **ed** may provide a partial justification of the program, but why has **ed** continued to be used? Why, at a time of personal computers, color monitors, and high-speed local area networks, hasn't **ed** become extinct? The answer goes back to the UNIX philosophy of self-contained, interoperating tools. **ed**'s command syntax is a powerful little language for the manipulation of text. Of course many other applications use a syntax to manipulate text. What makes UNIX especially useful is that it uses the same (**ed**-based) syntax for all of these applications. Learn the command language for **ed**, and you'll know the language needed for searching in files (**grep**, **fgrep**, **egrep**), for making changes in files via noninteractive shell scripts (**sed**), for comparing files (**diff**, **diff3**), for processing large files (**bfs**), and for issuing commands in the **vi** screen editor.

Editing Modes

ed and **vi** are both editors with separate *input* and *command* modes. That is, in input mode, anything you type at the keyboard is interpreted as input intended to be placed in the file that is being edited; in command mode, anything entered at the keyboard is taken as a command to the editor to allow you to move around in the file, or to change parts of it.

There is a relationship between the two modes, input mode and command mode, of **ed**. While the program is in input mode, any characters typed are placed in the document. Typing a single dot (**.**) alone on a line moves you to command mode. In command mode, characters typed are interpreted as directions to **ed** to perform some action. In command mode, the commands **a**, **c**, and **i** move you back to input mode.

ed is a *line-oriented* editor; it displays and operates on either a single line or a number of lines of text. **ed** has been available from the earliest versions of the UNIX System and works on even the simplest and slowest terminals.

vi is a *screen-oriented* editor; it lets you see an entire screen full of text. This works well only on video (screen-based) terminals. *vi* is discussed in detail in Chapter 5 of the book.

Starting ed

The simplest way to begin using **ed** is to type the command with a filename:

```
$ ed file1
52
```

ed opens up the file, reads it into its buffer, and responds with the number of bytes in the file (in this case, 52). If you are creating a new file (one that doesn't exist in the current directory), **ed** will respond with a "?" to remind you that this is a new file:

```
$ ed file88
?file88
h
cannot open input file
```

The "?" message is displayed to remind you that **ed** cannot find the file you specified. If you ask for help with the **h** command, you get a terse message that says **ed** cannot open the file, since it doesn't exist yet. In this case, that's no problem. **ed** keeps the text that is being worked on in a *buffer.* This buffer can be thought of as a note pad. If you specify a file to be edited, **ed** will have that file read into the buffer. If no file is specified, **ed** starts with an empty buffer. You can type some text, change it, delete some, or move it around. When you are done, you can save it by writing the buffer to a file on disk.

Adding Text

ed now waits for a command. Most commands are single letters, sometimes preceded by the line numbers the command refers to. Except for the **H** (help) command, *all commands must be lowercase.* If you have an empty buffer, the first step is to type something into it. The first command to use is **a**, which stands for *append.* The **a** command must be on a line by itself. Append puts **ed** in input mode so that all subsequent characters typed are interpreted as input and placed in the editing buffer; for example,

```
$ ed mydog
?mydog
a
The quick brown
fox jumped
over the
lazy dog
Through half-
shut eyes,
the dog
watched the
fox jump, and
then wrote down
his name.
```

```
The dog drifted
back to sleep
and dreamed
of biting the
fox.
.
```

The **a** command places **ed** in input mode and appends your text *after* the current line. The **i** command stands for *insert* and places your typed text *before* the current line. Insert works exactly the same way as append. The command must be on a line by itself. Both **a** and **i** put you in input mode, and neither provides any feedback that the mode you are in has changed. The only difference between the two input modes is that **i** inserts before the current line and **a** appends after the current line.

Leaving Input Mode

Since **ed** is a two-mode editor, the most important commands for a beginner to remember are the ones that are needed to change modes. Both **a** and **i** move you into input mode.

The only way to stop appending or inserting text and return to command mode is to type a single period (**.**) alone on a line. This gets **ed** out of input mode and back into command mode.

Saving Your Work

While you are editing, the text that you have entered is held by **ed** in a temporary memory (the buffer). When you are done editing, you will want to save your work to more permanent storage. To do this, write the buffer to a disk file by using the **w** (*w*rite) command. The **w** should begin the line or be on a line by itself:

```
w
191
```

ed writes the file to disk and tells you how many characters were written (in this case, 191). **ed** writes the buffer to the filename you specified when you first started the editor.

If you wish to put the text in the buffer in a different file, give the **w** command the new filename as an argument. For example,

```
w newfile
191
```

This will create the file if it does not already exist. If you specify a file that already exists, **ed** will replace the contents of that file with what is in your buffer. That is, whatever was in that file is replaced by the text you have in **ed**. **ed** does not warn you that your old file contents will be lost if you do this, so be careful about selecting names for your files. Writing to a file does not affect the contents of the buffer that you are working on; it simply saves the current contents.

It's good practice to issue the **w** command every few minutes when editing. If the system crashes or the line or LAN to your PC or terminal goes down, everything in the temporary buffer will be lost. Writing your work periodically to more permanent storage reduces the risk of losing everything in temporary storage.

The Quit Command

When you have finished working on your text and wish to end your editing session and store what you have done, write the file with the **w** command. You can then quit the editor and return to the shell by using the **q** (*q*uit) command. **ed** exits and the UNIX System responds with your prompt ($):

```
w
191
q
$
```

You can also combine write and quit, so that

```
wq
191
$
```

will write the file and quit the editor.

If you attempt to leave the editor without writing the file, **ed** will warn you with a single "?" character. If you enter the **q** command again, **ed** will quit and discard the buffer. All the work done in this editing session will be lost:

```
q
?
q
$
```

Displaying Text

Let's get your file, *mydog,* back into **ed** and make some changes in it:

```
$ ed mydog
191
```

Remember, **ed** is a line-oriented editor. By default, commands that you issue operate on one specific line, called the *current line.* When **ed** loads the file into its buffer, it sets the value of the current line to the last line in the file. If you were to issue the append command again, as shown here,

```
a
What a foolish,
sleepy dog.
.
```

ed would go into input mode, and all that you typed would be appended to the buffer after the last line. If you wish to print out one of the lines that you typed in, issue the

```
p
```

command, which will print out the current line. While you are in command mode, the command . (dot) stands for the current line. So

```
.p
```

also prints the current line, and

```
.=
```

prints out the *number* of the current line. The following three commands,

```
.
p
.p
```

will all print out the current line. Line *addresses* (line numbers) can be included with the **p** command. For instance,

```
1p
```

will print out the first line of the file and this command,

```
1,6p
```

will print the range of lines between lines 1 and 6. The symbol $ stands for the last line, so

```
$p
```

will print out the last line in the buffer, and

```
1,$p
```

will print out all the lines in the buffer. The abbreviation

```
,p
```

is a shorthand way to accomplish the same thing.

Line addressing can also be done relative to the current line. For example,

```
+1p
```

will print out the next line, as will a carriage return all by itself. The command

```
n
```

is similar to **p** except it prints out the line numbers as well as the lines. For example,

```
$ ed mydog
219
1,5p
The quick brown fox jumped
over the lazy dog.
Through half-shut eyes,
the dog watched the fox jump,
and then wrote down his name.
1,5n
1    The quick brown fox jumped
2    over the lazy dog.
3    Through half-shut eyes,
4    the dog watched the fox jump,
```

Command	Action
p	Print current line
*N***p**	Print line *N*
.p	Print current line
A,*B***p**	Print from line number *A* to line number *B*
n	Print current line showing line number
.n	Print current line showing line number
*N***n**	Print line number *N* showing line number
A,*B***n**	Print from line number *A* to line number *B* showing the line numbers
l	Print current line, including non-printing characters
*N***l**	Print line *N*, including non-printing characters
.l	Print current line, including non-printing characters
A,*B***1**	Print from line number *A* to line number *B* including non-printing characters
.=	Print line number of current line

TABLE 1 ed Commands Used to Display Text

```
5    and then wrote down his name.
```

Using **n** instead of **p** is an easy way to keep track of where the current line (dot) is in a file and which lines you are working on. Table 1 lists the **ed** commands used to display text.

Displaying Nonprinting Characters

Occasionally when working with **ed**, you may accidentally type some nonprinting characters in your text. For example, you may type CTRL-L in place of **L** (SHIFT-L). Such control characters are not normally visible in the file but can cause your document to do strange things when you try to print or format it. For many printers, CTRL-L Stands for *form feed.* Every time you try to print out this file, the printer will skip a page when it reaches CTRL-L. To help with this kind of problem, **ed** provides the **l** command, which gives a list of all the characters on the line. For example,

```
$ ed mydog
219
2,3n
2    over the lazy dog.
3    Through half-shut eyes,
2,3l
over the lazy dog. \014
Through half-shut eyes,
```

In line 2 of the file, you see a control character—in this case, \014—which is the way **ed** represents CTRL-L in ASCII octal.

Deleting Text

The **a** command appends text after the current line and puts **ed** in input mode; the **i** command inserts text before the current line and puts **ed** in input mode. To delete the current line, you use the **d** (*d*elete) command. Simply type **d** alone on a line when in command mode:

```
d
```

ed will not give you any message to confirm that you have deleted the text. To see the result, use the **p**, **n**, or **l** command.

Delete, like many of the commands in **ed**, will take line addresses. You can delete several consecutive lines very easily. For example,

```
4d
```

will delete line 4. You can also delete a range of lines with the **d** command. For example,

```
$ ed mydog
219
1,6p
The quick brown fox jumped
over the lazy dog.
Through half-shut eyes,
the dog watched the fox jump,
and then wrote down his name.
The dog drifted back to sleep
4,6d
.=
4
```

will delete everything from line 4 through line 6 of the buffer. When deleting text, **ed** sets the value of the current line to the next line after the deleted material. In this case, you deleted lines 4 through 6, and the current line number is 4. If you delete everything to the end of the file, the current line is set to 4.

Avoiding Input Errors

Since **ed** is a two-mode editor, entering commands while still in input mode is a 0000000common mistake. For example, you may enter some text, and then type

```
1,$p
```

to see it. If you forgot to type **.** (dot) alone on a line to get into command mode, the characters "1,$p" will simply be added to your text. You'll know when this happens because **ed** will remain in input mode when you expect it to display text from the buffer on the screen. To correct this, leave input mode, find the offending line, and delete it this way:

```
.
$n
22    1,$p
22d
```

Undoing a Change

If you make an editing mistake and notice it quickly, the last command (and only the last command) can be undone using the **u** (*u*ndo) command. Any changes you make are temporarily held by **ed**. Undo works for all modifications, but it is especially important for text deletions. The undo command must be issued immediately, for it operates only on the last command that modified the text. If you delete something and then add something else, the deletion is lost forever if you failed to use undo before adding.

There is one way to partly recover from serious error. Suppose you had the following:

```
$ ed mydog
219
1,4n
1    The quick brown fox jumped
2    over the lazy dog.
3    Through half-shut eyes,
4    and dreamed of biting
1,$d
a
an easy
.
1,$n
1       an easy
```

You've deleted all your original text and added two words! Since **u** (undo) works only on the last command, it can't restore your original text. (Undo would reverse the last command, which was to append the words "an easy.") What can you do? The only solution here is to quit the editor without writing the changes to the file:

```
q
?
h
warning: expecting w
q
$
```

A **q** (quit) command without a **w** command gives a warning ("?"), because you are quitting without saving any of the changes made in the file. If you confirm by asking to quit a second time, **ed** assumes you know what you are doing and quits without altering the original file. This is only a partly acceptable solution. Since you have not saved any of the changes made in the file, you have only the original text stored, which includes the text you accidentally deleted. However, because you have not saved any of the changes you made in the file, you've lost all the work you did since the last **w** (write) command.

Making a Backup Copy

It is often a good idea to make a *backup* copy of your work before you make significant alterations or deletions. If you've made a mistake or have changed your mind about deleting material, you can still recover it from the backup file. You have several ways to do this on UNIX.

Before you begin editing, you can simply copy the file using the **cp** (*cop*y) utility:

```
$ cp mydog mydog.bak
```

If you make a backup copy of the file when you begin to work on it and write the file when you are done with an editing session, you will have the file both in its original form now *(mydog.bak)* and in its new, changed form *(mydog).* If you make a mistake or change your mind about something, you can recover without losing too much of your work.

You can also create a backup file while inside **ed**. This is useful if you want to save your work before making substantial changes to the rest of the file, or if you want to create different versions of the same work. For example, in the middle of an editing session you may want to save your work to another file before making more changes, as shown here:

```
$ cp mydog mydog.bak
$ ed mydog
219
<<Many editing changes>>
1,$w mydog.bak1
372
<<Many more editing changes>>
w
418
q
$
```

In this example, you have made a copy of your original file *(mydog.bak)* before beginning to make any editing changes. Part of the way through the editing session, you made another backup *(mydog.bak1).* At the end, you write the file to the original file, *mydog.* You now have three versions of the file in various stages of development.

Manipulating Text

In addition to providing an easy way to enter text, **ed** provides several ways to manipulate it.

Moving Text Around

After you have entered some text in a document, you may find that you do not like the way the material is organized. Maybe part of the text in one section really belongs in the introduction, and some text at the beginning of the document should be moved to the summary. **ed** provides an easy way to move blocks of text to other places in the file with the **m** (*m*ove) command. **m** is used with line addresses. For example,

```
<start line number>,<end line number> m <after this line number>
```

means move the block of text from the starting line number through the ending line number, and put the whole block *after* the designated third line number. Therefore,

```
3,14 m 56
```

takes lines 3 through 14 and places them after line 56.

To move text before the first line of the buffer, type

```
3,14 m 0
```

which takes lines 3 through 14 and places them after line 0 (before line 1). The current line will be at the last line of the material moved.

Transferring Lines of Text

If you want to make copies of part of the file (for example, to repeat something in the summary), **ed** provides the **t** (*t*ransfer) command. The **t** command works exactly like the **m** command, except that a *copy* of the addressed lines is placed after the last named address. The current line (dot) is left at the last line of the copy. The syntax of the **t** command is

```
<start line number>,<end line number> t <after this line number>
```

To copy text, type

```
3,14 t 0
```

which makes a copy of lines 3 through 14 and places them after line 0 (before line 1).

Modifying Text

At this point, you know how to create, delete, and move text around. If you were to find an error in the text, you could delete the line that has the error and retype it. To avoid having to retype a whole line to correct a single letter, you need some additional commands.

Change

The **c** (*c*hange) command allows you to replace a line or group of lines with new text:

```
4,7c
Some new stuff that should be put in place of lines
4 through 7
.
```

The lines typed between the original **c** command and the final **.** (dot) replace the addressed lines.

Using change is a little more efficient than deleting and creating, but you still have to type a whole line to correct an error. What you really need is a way to correct small errors without massive retyping.

Substitute

The **s** (*s*ubstitute) command allows you to change individual letters and words within a line or range of lines. Note the word "exiting" in the file called *session*:

```
ed session
147
,n
1       This is some text
2       being typed into the
3       buffer to be used as
4       an example of an exiting
5       session.
6       Some new stuff typed in during my last
7       work session.
```

This is an example of an *editing* session, not an *exiting* session. To correct the error, position **.** (dot) at line 4, issue the **s** command, and print it out by issuing the **n** command, in a combined command that takes this form:

```
<start line number>,<end line number>s/change this/to this/n
```

You can do this, for example:

```
4s/exi/edi/n
4       an example of an editing
```

You can delete a single word or a group of letters by typing

```
s/an//n
4       example of editing
```

In other words, for the letter combination *an,* substitute nothing. In this case, two adjacent slashes mean nothing; separating the slashes with a space would have replaced the word "an" with an extra space.

Substitute changes only the first occurrence of the pattern found on the line. If you had used

```
4s/ex/ed/n
4       an edample of exiting
```

a new error would have been created by changing the first *ex* in the line instead of the second one. It's a good idea to type short lines in **ed**. Since the substitute command only applies to the first occurrence of a word on a line, long lines with lots of material become tricky to change. Substitute works with a range of line addresses as well. For example,

```
1,$s/selling/spelling/
```

will go through the entire file, from line 1 to line $, and change the *first* occurrence of "selling" on every line to the word "spelling."

Global Substitution

To change all occurrences of a word on one line, place the **g** (*g*lobal) command after the last / in the substitute command line. For example,

```
1,$s/selling/spelling/g
```

will go through the entire file from line 1 to line $ and change *every* occurrence of "selling" on every line to the word "spelling." Table 2 reviews the basic **ed** commands used to this point.

Advanced Editing with ed

At this point, you have all the capabilities you need to provide a workable editor. You can add, delete, move, and change parts of text. However, **ed** also provides several other features that are very sophisticated, compared to other early editors. These other capabilities are the basis for the use of **ed**'s syntax in other editing programs.

Command	Action
<line number> **a**	Place ed in input mode and append text after the specified line number. If no line number is specified, the current line is used as the default.
<line number>**i**	Place ed in input mode and insert text before the specified line number. If no line number is specified, the current line is used as the default.
<start line numb>, <end line numb>**p**	Print on the terminal the lines that go from starting line to ending line. If a single line number is given, print that line. If no line number is given, print the current line.
<start line numb>, <end line numb>**n**	Print the range of lines with their line numbers.
<start line numb>, <end line numb>**l**	Print the range of lines in a list form that displays any non-printing (control) characters.
.	Print out the line number of the current line.
.=	Print out the current line. This is synonymous with .p.
<start line numb>, <end line numb>**m** <after line>	Take all the text that occurs between starting line and ending line and move the whole block to after the last line address.
<line number>**r** <filename>	Read in the contents of <filename> and place it in the buffer after the line number given.
<start line numb>, <end line numb> **w** <filename>	Write all the lines from start to end into a file called <filename>. If no file is specified, the name of the current file in the buffer is assumed.
<start line numb>, <end line numb> **d**	Delete all the lines from the starting address to the ending address.
u	Undo the last change made in the buffer; restore any deletions, remove any additions, put back changes.
s/this stuff/that stuff/	Find the first place in the current line where **this stuff** appears and substitute **that stuff** for it.

TABLE 2 Basic ed Command Formats

Searching and Context Editing

Having to specify the line address for a command is tedious. To find an error, you need to scan through the file, find the line number, and make a substitution. Making a change in the file by adding or deleting lines changes all the remaining line numbers and makes subsequent editing more difficult. **ed** has commands that allow you to search for specific combinations of letters.

The command

```
/Stuff/
```

will search through the buffer, beginning at the current line (dot) until it finds the *first* occurrence of "Stuff." The value of the current line is reset. The search starts at the current line, proceeds forward to the end of the file, and then *wraps around* to search from the beginning of the file to the current line. If the search expression is not found, the current line (dot) is unchanged.

You can also do a search backward through the file. The command

```
?Stuff?
```

specifies a search backward through the buffer from the current line (dot) up to the beginning of the file. The search wraps around to the end and continues back to the current line. If the search expression is not found, the current line (dot) is unchanged.

Context searching can be used with any command in the same way a command address is used. The context search commands are presented in Table 3.

Often a search will not turn up the instance of "Stuff" you want. The command

```
/Stuff/
```

may turn up the wrong "Stuff," and you may want to search again to find the right "Stuff." **ed** provides shorthand for this. The command

```
//
```

means "the most recently used context search expression." This shorthand can also be used in the substitute command in context editing. For example, if you've just used the command

```
/Stuff1/
```

then

```
//s//Stuff2/
```

means "find the next instance of Stuff1, and substitute Stuff2 in place of it" (Stuff1 is the most recently used context search expression). In the same way,

```
??
```

means scan backward for the last search expression.

Command	Action
/Stuff/**n**	Find the next line with "Stuff" in it, and print the line with its line number.
/Stuff/**d**	Find the next line with "Stuff" in it, and delete it.
/Stuff1/,/Stuff2/ **m $**	Take everything from the next occurrence of "Stuff1" up to the next occurrence of "Stuff2" and move it all to the end of the buffer. Both the search for "Stuff1" and the search for "Stuff2" begin at the same point, the current line (dot).
/Stoff/**s**/tof/tuf/	Find the misspelled word "Stoff" and substitute "tuf" in place of "tof."

Table 3 ed Context Search Command Formats

Global Searches

The **g** (global) command also applies to the search expressions discussed earlier. When used as a global search command, the **g** comes before the first /. The **g** command selects all lines that match a pattern and then executes an action on each in turn. For example,

```
g/the/p
```

prints out all the lines that have the word "the" in them, and

```
g/the/s/the/that/
```

selects all lines that contain the word "the" and changes the first occurrence of "the" in each line to "that."

The v Command

The **v** command is the inverse of **g** and is also global. **v** selects all lines that *do not* have the pattern in them, and it performs the action on those lines. Thus,

```
v/the/p
```

prints out all lines that do not contain the word "the", and

```
v/the/s/selling/spelling/
```

looks for all lines that do not contain the word "the" and changes the first occurrence of "selling" to "spelling" only in those lines. Global commands also take line number addresses; for example,

```
1,250g/the/p
```

prints all lines between 1 and 250 that contain the word "the."

Regular Expressions

Searching for text strings in a file during an editing session is done frequently. The **ed** command provides an exceptional search capability. In addition to being able to specify exact text strings to be searched for, **ed**'s search capability includes a general language (syntax) that allows you to search for many different patterns. This syntax is called the *regular expression.*

Regular expressions allow you to search for similar or related patterns, not just exact matches to strings of characters.

Metacharacters

Regular expression syntax uses a set of characters with special meaning to guide searches. These *metacharacters* have special meaning when used in a search expression.

Beginning and End of Line The caret (^) refers to the beginning of the line in a search, and the dollar sign ($) refers to the end of the line.

```
/^The/
```

```
/The$/
```

These commands will respectively match a "The" only at the beginning of the line, and a "The" only at the end of the line.

Wildcards When using regular expressions, you should remember that they are often a difficult aspect of **ed** to learn, because the meaning of a symbol can depend on where it is used in an expression. For example, in input mode a dot (**.**) is just an ordinary character in the text, unless it is on a line by itself, in which case it means "put me back in command mode." In command mode, **.** by itself means "print out the current line." In a regular expression, **.** means "any character." So the command

```
/a...b/
```

means "find an *a* and a *b* that are separated by any three characters." The command

```
/./
```

means "find any character" (except a newline character) and matches the first character on the line regardless of what it is.

When **.** occurs on the right-hand side of a substitute expression, it means "a period." These can be combined in a single command:

```
.s/././
```

This command shows all three meanings of **.** in an expression. The first **.** means "on the current line, substitute (for any character) a period (.)." For example,

```
p
How are you?
.s/././
.ow are you?
```

The Asterisk (*) The * metacharacter means "as many instances as happen to occur, including none." So the command

```
s/xx*/y/
```

instructs **ed** to substitute for two or more occurrences of *x*, a single *y*. The command

```
s/x.*y/Y
```

means "substitute the character *Y* for any string that begins with an *x* and ends with a *y* separated by any number of any characters." The strings *xqwertyy*, *xasdy*, and *xy* would all be replaced by *Y*.

The Ampersand (&) The & is an abbreviation that saves a great deal of typing. If you wanted to change

```
This project has been a success.
```

into

```
(This project has been a success.)
```

you could use the command

```
s/This project has been a success./(This project has been a success.)/
```

to make the change.

This is a bit of unnecessary typing, and unless you are a skilled typist, you take a chance of introducing a typographical error in retyping the line. Instead, you can type

```
s/This project has been a success./(&)/
```

where the "&" stands for the last matched pattern, which in this case is "This project has been a success."

Character Classes in Searches — [ ]

By using regular expressions, you can specify *classes* of things to search for, not just exact strings. The symbols [and] are used to define the elements in the class. For example,

```
[xz]
```

means the class of lowercase letters that are either *x* or *z*; therefore

```
/[xz]/
```

will find either the next *x* or the next *z.* The expression

```
[fF]
```

stands for either an uppercase or lowercase *f.* As a result, the search command

```
/[fF]red/
```

will find both "fred" and "Fred." The expression

```
[0123456789]
```

will find any digit, as will the shorthand expression

```
[0-9]
```

which means all the characters in the range 0 to 9. In **ed**, the expression [0-9] refers to any digit in the file. The search command

```
/[0-9]/
```

searches for any digit in the file. The class of all uppercase letters can be defined as

```
[A-Z]
```

[A-Z] means all the characters in the range of *A* to *Z.* To search for a character that is not in the defined class, you use the caret (^) symbol inside the brackets. The expression

```
[^]
```

Character	Meaning
.	Any character other than a newline
*	Zero or more occurrences of the preceding character
.*	Zero or more occurrences of any character
^	Beginning of the line
$	End of the line
[–]	Match the character class defined in brackets
[^–]	Match anything not in the character class defined in the brackets
\	Escape character; treat next character, *X*, as a literal *X*

Table 4 Special Characters and Their Meanings in ed

means any character *not* in the range included in the brackets. The following expression,

```
[^0-9]
```

means any character that is not between 0 and 9, in other words, any character that is not a digit.

Turning Off Special Meanings

The characters $ [ ] * . are part of the regular expression syntax; they all have special meaning in a search. Table 4 shows these characters and their meanings.

How do you find one of these literal characters in a file? What if you need to find $ in a memo? The backslash (\) character is used to turn off the special meaning of metacharacters. Preceding a metacharacter with a \ means the literal character. If you were to type

```
/./
```

ed would search for *any* character, which is probably not what you had in mind. The following command, however,

```
/\./
```

searches for a literal period (.), not "any character." The following command,

```
/\*/
```

searches for a literal asterisk, or star, not "zero or more occurrences of the preceding character." And of course,

```
/\\/
```

searches for a literal backslash (\). Therefore, to find a $, use this search expression:

```
/\$/
```

Other Programs That Use the ed Language

The commands and syntax used by **ed** to search, replace, define global searches, and specify line addresses are used by many other UNIX programs. These programs are discussed in detail elsewhere in this book, but it is relevant to point them out here and note how they use the **ed** syntax for other tasks.

ed Scripts

Our examples have been using **ed** as an interactive editor to modify and display the text you are working on. However, you need not think of **ed** as only an interactive program. To make changes in a file, you really don't have to watch them happen. In editing, you read a file into the buffer, issue a sequence of editing commands, and then write the file. If you rely heavily on the **p**, **n**, and **l** commands to display your work, you do so mainly for your own reassurance. The following expression,

```
g/friend/s//my good friend/gp
w
q
```

finds all the instances of the word "friend" in a file, changes that word to "my good friend," and prints out every changed line. The next expression,

```
g/friend/s//my good friend/g
w
q
```

does the same thing but does not print out the changed lines.

It is possible with **ed** to put all of your editing commands in a *script* file and have **ed** execute these commands on the file to be edited. For example,

```
$ ed filename < script
```

takes the **ed** commands in *script* and performs them in sequence on *filename.* There are many times when this capability to do noninteractive editing is very useful. If you need to make repetitive changes in a file, as with a daily or weekly report, **ed** scripts provide an automatic way to make the changes, if you plan out the complete sequence of editing commands you want executed.

Here is an example of how to use a script. The program **cal** prints out a calendar on your screen. **cal 6 2006** will print out the calendar for June 2006; **cal 2006** will print out the calendar for the whole year. The commands

```
cal 6 2006 > tmp
ed tmp < script
```

will put the calendar for June 2006 in a file, *tmp,* and edit it according to any **ed** commands found in *script.* A script such as

```
g/January/s//Janeiro/
g/February/s//Fevereiro/
```

```
g/March/s//Marco/
g/April/s//Abril/
g/May/s//Maio/
g/June/s//Junho/
g/July/s//Julho/
g/August/s//Agosto/
g/September/s//Setembro/
g/October/s//Outubro/
g/November/s//Novembro/
g/December/s//Dezembro/
w
q
```

will relabel the name of the month in Portuguese. For example,

```
Junho 2006
 S   M  Tu   W  Th   F   S
                1   2   3
 4   5   6   7   8   9  10
11  12  13  14  15  16  17
18  19  20  21  22  23  24
25  26  27  28  29  30
```

diff

Another use of editing scripts is in conjunction with the program **diff**. **diff** is a UNIX program that compares two files and prints out the differences between them. By comparing your file before you edited it *(session.bak)* with its current form *(session)*, you should see the changes that have been made:

```
$ cat session.bak
This is some text
being typed into the
buffer to be used as
an example of an editing
session.

$ cat session
This is some text
being typed into the
buffer to be used as
an example of an editing
session.
Some new stuff typed in during my last
work session.
$ diff session.bak session
5a6,7
> Some new stuff typed in during my last
> work session.
$
```

Looking at the output of *session.bak* and *session*, you notice that two sentences were added at the end. The **diff** command tells you, using **ed** syntax, that material was appended after line 5, and it shows you the text added. **diff** uses < to refer to lines in the first file and > to refer to lines in the second file. **diff** also has an option that allows it to generate a script of **ed**

commands that would convert file1 into file2. In the following example, **–e** is used to create the **ed** commands that change the file *session.bak* into the file *session.* In this case, you would have to add two lines after line 5, as shown in this example:

```
$ diff -e session.bak session
5a
Some new stuff typed in during my last
work session.
w
q
```

The **–e** option is useful in maintaining multiple versions of a document or in sending revisions of a document to others. Rather than storing every version of a document, just save the first draft of the file and save a set of editing scripts that converts it into any succeeding version. You can use the **ed** command and the **ed** script to create different versions of documents. For example,

```
$ ed document.old < rev3
```

will take the *document.old* file and edit it using the commands in *rev3* to update the original file.

You often see this method used by UNIX users to update information. On the Usenet, for example, people often distribute updates to source programs or to documentation by sending an **ed** script—the output of **diff –e**—instead of the complete new version of the material. Remember, however, that the **ed** script changes the original file. If you need a copy of the original, be sure to copy it to a safe place; for example: **cp** *document.old document.bak.*

grep

Searching files is a task that you will want to do often. The UNIX System provides a search utility that can search any ordinary text file. This utility is called **grep**. The name is a wordplay on the way searches are specified in **ed**: g/re/p for *g*lobal/*r*egular *e*xpression/*p*rint. The command's syntax is

```
grep pattern [filename]
```

The **grep** command searches input for a pattern and sends to standard output any lines that match the pattern. For example, to find all instances of "dog" in your *mydog* file, use the following command:

```
$ grep dog mydog
over the lazy dog.
the dog watched the fox jump,
The dog drifted back to sleep
sleepy dog.
```

In this example, **grep** looks for the pattern "dog" in the file *mydog,* and prints on the screen all lines that contain "dog."

The pattern that you provide for a search can be a regular expression, as used in **ed**. For example,

```
$ grep "[0-9]" mydog
```

will print out any lines in the file *mydog* that contain a digit. The following example, however,

```
$ grep "[^0-9]" mydog
```

will print out all lines that do not contain a digit. In both these examples, notice that you must use quotation marks with the regular expression to prevent the shell from interpreting the special characters before they are sent to **grep**.

sed

sed is a stream editor that uses much the same syntax as **ed**, but with extra programming capability to allow branching in a script. A stream editor is another noninteractive editor that allows changes to be made in large files. **sed** copies a line of input into its buffer, applies in sequence all editing commands that address that line, and at the end of the script copies the buffer to the standard output. **sed** does this repeatedly, until all the lines in the file have been processed by all the relevant lines in the script. The basic advantage of **sed** over **ed** is that **sed** can handle much bigger files than **ed**. Since **sed** reads and processes a line at a time, files that exceed **ed**'s buffer size can be handled. For example,

```
sed 'g/friend/s//my good friend/g' session
```

will change all occurrences of "friend" to "my good friend" in the file *session.* If you put the commands in the file *script,* then

```
sed -f script sessions > session.out
```

reads its commands from the file *script* and applies them to the file *sessions,* and puts the output in the file *session.out.*

sed is discussed in much greater detail in Chapter 21.

Summary

It's useful to know how to use **ed**. **ed** provides a way to enter and delete text, and **ed**'s global features, context editing, and regular expression searching make it powerful. A few keystrokes can accomplish a great deal.

Although **ed** is powerful in manipulating text, it is weak in displaying it. **ed** shows you the lines you are working on only when you ask for them. **ed** works well for editing a file that you have printed out and marked up, but it's difficult to do real-time editing of a document when you can't see much of it in front of you.

ed provides you with little feedback about the effects of commands you have entered. When you enter significant commands such as **1,$d** (delete all lines from the first to the last in the file), you won't see the effects of the command on the screen. In addition, **ed**'s error and warning messages are terse.

The concept of a regular expression is important in many other UNIX programs, and in shell programming as well. **ed** can fix a *.profile* or make changes in important programs or documents. On slow data connections or on very heavily loaded systems at busy times, **ed**'s line-editing capability may be the only reasonable way to get work done acceptably.