Introduction to the ed Line Editor

This tutorial introduces the interactive editor **ed**. It is intended both for readers who want a tutorial introduction to **ed**, and those who want to use specific sections as a reference.

Related tutorials include those for **sed**, the stream editor, and for **me**, the MicroEMACS screen editor. This tutorial assumes that you already understand the basics of using the COHERENT system, such as what a file is, what it means to edit text, and how to issue commands to the operating system. If you not yet know your way around the COHERENT system, we suggest that you first study the *Using the COHERENT System*, which appears in the front of this manual. It covers the basics of using COHERENT and introduces many useful programs.

Why You Need an Editor

A significant feature of computers is the capacity to store, retrieve, and operate upon information. A computer can store many different kinds of information: programs, computer commands and instructions, data for programs, financial information, electronic mail, or natural-language text (e.g., French, English) destined for a manuscript or book.

ed is a program with which you can enter and edit text on your computer. You can use **ed** to create or change computer programs, natural-language manuscripts, files of commands, or any other file that consists of text that you can read.

ed is designed to be easy to use, and requires little training to get started. The fundamental commands are simple, but have enough flexibility to perform complex tasks.

Learning To Use the Editor

Practice on your part will help you learn quickly. The following sections contain examples that illustrate each topic discussed. We strongly recommend that you type each example presented as you encounter it in the text. Even if you understand the concept presented, performing the example reinforces the lesson, and you will learn more quickly how to use **ed**.

In addition to reading the text and doing the examples as you encounter them in the text, try your own variations on the commands, and branch out on your own. Try things that you suspect might work, but are not shown as examples.

General Topics

This section presents the background information you will need to understand how **ed** works.

To help illustrate the discussion to follow, log into your COHERENT system and type the following commands:

```
ed
a
this is a sample
ed session
.
w test
q
```

This example calls **ed**, then uses the **a** command to add lines to the text kept in memory. The period signals the end of the additions. The **w** command writes the lines of text to file **test**, and the command **q** tells **ed** to return to COHERENT. You will notice that after you type the **w** command, **ed** will respond with

```
28
```

which is the number of characters in the file.

Thus, to enter **ed**, simply type

```
ed
```

and to exit, type

```
q
```

You can also exit by typing **<ctrl-D>**: that is, hold down the **control** key on your keyboard, and at the same time strike the **D** key.

Notice that you are issuing two different kinds of commands in the above example: the command **ed** is given to the COHERENT shell, to invoke the editor; the rest of the commands are given to the editor. After **ed** is given the **q** command, it exits, and following commands are processed by COHERENT.

ed, Files, and Text

ed works with one file at a time. With **ed**, you can create a file, add to a file, or change a file previously created.

As you use **ed** to create or change files, you will type both *text* and controlling *commands* into the editor. Text is, of course, the matter that you are creating or changing. Commands, on the other hand, tell **ed** what you want it to do. As you will see shortly, there is a simple way to tell **ed** whether what you are typing is text or commands.

ed has about two dozen commands. Almost every one is only one letter long. Although they may seem terse, they are easy to learn. You will appreciate the brevity of the commands once you begin to use **ed** regularly.

You must end each command to **ed** by striking the **<return>** key. This key is present on all terminals. However, the labeling of the key may vary. It may be called **newline**, **linefeed**, **enter**, or **eol**, and is larger than any key on the keyboard except for the space bar. This key will be called the **<return>** key in the remainder of this document.

Creating a File

The example shown above created a file. Here is another example of file creation — here, creating a file called **twoline**:

```
ed
a
Two line Example,
thank you.
.
w twoline
q
```

The letter **a** tells **ed** to add lines to the file. You are creating a new file with this example; and when **ed** creates a new file, it is initially empty. The **w** command writes the lines you have added to file **twoline**. The command **q** tells the editor that you are finished, whereupon it returns to COHERENT. You can use the COHERENT command **cat** to list the contents of the new file:

```
cat twoline
```

the reply will be:

```
Two line Example,
thank you.
```

Each command used here will be described in detail in later sections.

Changing an Existing File

Suppose that a manuscript file of yours needs a few spelling corrections. **ed** will help you make them. To begin, simply name the file to correct when you issue the COHERENT command:

ed *filename*

where *filename* stands for the name of the file that you wish to edit. For example, the following adds a line to the file *twoline*, which we just created:

```
ed twoline
$a
This is the third line of the file.
.
w
q
```

Listing the file with **cat** gives:

```
Two line Example,
thank you.
This is the third line of the file.
```

The command **$a** tells **ed** to add one or more lines at the end of the file.

Correcting the spelling of a misspelled word is easy with **ed**. You can rearrange groups of words in a manuscript, and you can move or copy larger portions of text, such as a paragraph, from one spot to another.

Working on Lines

ed uses the *line* as the basic unit of information; for this reason, it is called a *line-oriented* editor. A line is defined as a group of characters followed by an end-of-line character, which is invisible. When you type out a file on your terminal, each line in the file will be shown on your terminal as one line. The commands for **ed** are based upon lines. When you add material to a file, you will be adding lines. If you remove or change items, you will do so to groups of lines.

ed knows each line by its number. A line's number, in turn, indicates its position within the file: the first line is number 1, the second line is number 2, and so on.

ed remembers the line you worked on most recently. This can help shorten the commands you type, as well as reduce the need for you to remember line numbers. The line most recently worked on is called the *current* line. **ed** commands use a shorthand symbol for the current line: the period '**.**'.

Another shorthand symbol used in **ed** commands is **$**, which represents the number of the last line in the file.

Many of the **ed** commands operate on more than one line at a time. Groups of lines are denoted by a range of line numbers, which appears as a prefix to the command.

Error Messages

If you type a command to **ed** incorrectly, **ed** respond with:

```
?
```

This indicates that it has detected an error. Many times, this error will be evident to you when you review the command that you just typed.

If you do not see what the error is, you can get a more lengthy description by typing to **ed**:

```
?
```

It will reply with an error message.

Basic Editing Techniques

This section discusses in more detail the elementary techniques and commands that you need to use **ed**. With the material presented in this section, you will be able to do most basic editing tasks.

Again, it is recommended that you type each example. This will help you understand each example, as well as remember the technique it demonstrates.

Creating a New File

To begin, let us presume that you need to create an entirely new file named **first**. Perhaps you only want one line in the file, and it is to read

```
This is my first example
```

These are the steps that you will need to go through to create this file.

The first step is to invoke the **ed** program. To do this, simply type

```
ed
```

Remember that you must end each line of commands or text line by pressing the **<return>** key, because **ed** will not act upon it until you do. Thus, you invoke the editor by typing **ed** and a **<return>**. Notice that these two characters must be lower case.

ed is now ready for commands. The first command that you will use is the append command **a**. This tells **ed** to add lines to the text in memory, which will later be written to the file. The number of lines that **ed** can hold in memory depends upon the amount of memory in your computer. For editing very large files, you should use **sed**, the COHERENT stream editor, which is described in its own tutorial.

ed will continue to add lines until you type a line that contains *only* a period. While it is adding lines, **ed** does not recognize commands.

After you issue the **a** command, you can type the lines to be included, concluding with a line that consists only of a period. This special line signals **ed** that you want to stop appending lines. The information that you have typed so far is:

```
ed
a
This is my first example
.
```

Next, you must tell **ed** to write the edited text into a file. Do so by issuing the write command **w**, plus the name of the file that is to hold the edited text. For example, if you wish to store this example in a file named **first**, issue the command:

```
w first
```

ed will write the file and tell you how many characters were written, in this case 25.

Finally, to quit the editor issue the quit command:

```
q
```

The commands you type after this will be interpreted and acted upon by COHERENT.

Now, review the example in its entirety. First you invoked **ed** by typing **ed** at the COHERENT prompt. Then you issued the add command **a** to add lines to the file. added lines with the **a** command, and finished the adding by typing a line that consists only of a period. You then wrote the editing text into a file by issuing the write command **w**, and finally you exited from **ed** by issuing the quit command **q**. The complete example is:

```
ed
a
This is my first example
.
w first
q
```

ed replied to the **w** command by printing the number of characters it wrote into the file. After you typed **q** COHERENT prompted you for a command again.

Changing a File

Suppose that you wish to change the file that you have just created: you want to add two more lines to the file so that the original line will be sandwiched between the new lines. You want the file to contain:

```
Example two, added last
This is my first example
Example two, added first
```

You will do this with **ed** using two new commands.

Again, you start by telling COHERENT to run **ed**. This time, however, you must type the name of the file that you are changing after the characters **ed**:

```
ed first
```

ed will remember this file name for later use with the **w** command.

ed reads the file in preparation for editing, and tells you the number of characters that it read in, again 25.

After reading the file, **ed** automatically sets the current line to the last line read in.

Now, add the third line shown in the second example by entering:

```
a
Example two, added first
.
```

This resembles the first example. In that case, however, the file had no information, whereas now it does. How did **ed** know where to add the lines?

The **a** command adds lines after the *current line*. When **ed** reads a file, it initially sets the current line to the last line read in; therefore, the **a** command added the new line after the last line.

The current line is used implicitly or explicitly by most commands, so it is helpful to know where it is. In general, the current line is left at the last line **ed** has processed. If you lose track of the current line, you can ask **ed** to tell you where it is, as you will see shortly.

To add the very first line to the second example, you will use yet another command, the insert command **i**. This command is identical to the **a** command, except that it inserts lines *before* the current line rather than after it.

Another word about the current line. After an **a** command finishes, the current line is the last line added. Thus, after the addition of "Example two, added first" above, the current line is now the last line in the file. So, if you were to do the **i** command immediately, you would be adding lines just before the last line, which is not what you want to do.

Nearly every **ed** command is flexible enough to allow you to specify the line upon which the command is to operate. Now you can complete the second example:

```
1i
Example two, added last
.
```

The numeral **1** before the **i** tells **ed** to insert lines before the first line in the file. The line-number prefix is used frequently, and applies to nearly every command.

Now, to finish the second example and save it into the same file, type:

```
w
q
```

Note that the file name was left off the **w** command. **ed** remembers the name of the file that you began with, and uses that name if none is used with the **w** command. Therefore, the edited text is written back into file **first**. Note, too, that the previous contents of the file **first** are lost when you write the new file **first**. Alternatively, you can type:

```
w second
```

This leaves the contents of **first** unchanged and creates a new file called **second**.

In case you forget, **ed** can tell you the name of the file with which you began. Simply type the command:

```
f
```

If you had used **f** any time while working on this second example, **ed** would have replied:

```
first
```

Remember to use the **q** command to leave **ed** and return to COHERENT.

Printing Lines

As you use **ed** to edit a file, you will find it most useful to print sections of the file on your terminal. This helps you see what you have done (and sometimes what you have not done), and helps you pinpoint where you wish to make changes.

The print command **p** prints the current line unless you specify a line number.

Continuing with the example begun above, when you type the commands

```
ed first
p
```

ed replies by printing

```
Example two, added first
```

which is the last line in the file named **first** from the previous example.

Again, like the commands **i** and **a**, if you want **ed** to print a line other than the current one, just prefix the **p** command with a line number. Thus, if you want to print the second line in the file, type:

```
2p
```

ed will reply with:

```
This is my first example
```

If you wish to print more than one line of a file, you can tell **ed** to print a *range* of line numbers: type the numbers of the first and last lines you wish to see, separated by a comma. For example, to print all three lines in the second example, type:

```
1,3p
```

ed responds by printing all lines. This same principle applies to other commands. The print command can also appear after other commands such as **s** or **d**, which are discussed later in this section.

Abbreviating Line Numbers

ed recognizes some shorthand descriptions for certain line numbers. The number of the last line can be represented by the dollar sign **$**. Thus, the command

```
1,$p
```

prints every line in the file. The advantage of this shorthand is that the command as typed works for any file, regardless of its size. This construct of **1,$p** is used often enough that it has an abbreviation of its own:

```
*p
```

The number of the current line can also be abbreviated by using the period or dot in the place of a line number. To print all lines from the beginning of the file through the current line, type:

```
1,.p
```

To print all lines from the current line through the end of the file, type:

```
.,$p
```

The special symbol **&** prints one screenful of text. Simply type:

```
&
```

This is equivalent to:

```
.,.+22p
```

If there are fewer than 23 lines between the current line and the end of the file, it is equivalent to

```
.,$p
```

All forms of the **p** command change the current line to the last line printed. The command

```
.,$p
```

after printing changes the current line to the last line of the file.

How Many Lines?

You can easily see the current line with **p**. Type:

```
p
```

This tells **ed** to print the current line. On your terminal, try the command:

```
.p
```

You will see that it does the same thing as **p**.

To discover how large your file is, just type:

```
=
```

ed will reply by typing the number of lines in the file.

To find the number of the current line, use the **dot equals** command:

```
.=
```

ed responds with the number of the current line.

Removing Lines

Editing means removing lines of text, as well as adding them. To illustrate how **ed** lets you remove lines of text, create another example file with **ed**:

```
ed
a
This is the first line.
The second line is good.
However, line three is bad.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
.
w example3
q
```

This creates a file named **example3**.

Now, you can practice removing lines that you no longer want. Begin editing the file by typing:

```
ed example3
```

Now, print the contents of the file by typing:

```
1,$p
```

Our first task is to delete lines 3 through 6. First, delete line 3, then print the entire file again.

```
3d
1,$p
```

and **ed** will respond with

```
This is the first line.
The second line is good.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
```

Notice that the original file's third line is no longer there. Line 3 is now what used to be line 4. Remember that the line numbers *always* begin with 1 for the first line of the file and progress consecutively even after the file has been changed. Thus, deleting a line will change the line number of each line from the deleted line to the the last line in the file.

You still need to remove three more lines. You can do this with one command:

```
3,5d
```

Again, type ***p** to print the contents of the file:

```
This is the first line.
The second line is good.
the next to last line stays.
as does the last line in the file.
```

Finally, write the updated file and quit:

```
w
q
```

This illustrates how to delete lines, both singly and in a group.

Abandoning Changes

Sometimes, you may make a mistake; rather than damage your file with badly edited text, you may wish to abandon what you have done and begin all over again. You can do so by using the **q** command in a different fashion than is shown above.

If you tell **ed** to **q** before you tell it to write the file with **w**, you abandon any changes made since beginning editing. However, to prevent you from accidentally selecting this option, **ed** checks to see if you have made any changes to the file; and if you have, it responds with a question mark '**?**'. To tell **ed** that you know what you are doing and really do wish to abandon the edited file, reply with a second **q**. **ed** will then quit and return you to COHERENT.

You can avoid the question mark prompt by typing the upper-case **Q** rather than lower-case **q**: **ed** will exit without regard to unsaved changes. You can also exit from **ed** by typing the end-of-file key **<ctrl-D>**.

Substituting Text Within a Line

If you type a line incorrectly, or later wish to rearrange some words or symbols within it, you know enough about **ed** now to do so. You only need to delete the line with the delete command **d** and re-type the line with the insert command **i**. To see how this is done, prepare the file **example4**, as follows:

```
ed
a
Software technology today has
adbanced to the point that large
software projects unherd of in
earlier times are undertaken and
.
w example4
q
```

This example has two misspelled words. We will correct each of them using different **ed** features.

The first method will be the direct way that you probably can anticipate. Give the following commands to the editor exactly as shown:

```
ed example4
2d
i
advanced to the point that large
.
```

These commands use the delete command **d** to delete the second line, and then uses the insert command **i** to insert the correct new line in its place.

Use the command

```
*p
```

to verify that the file now contains:

```
Software technology today has
advanced to the point that large
software projects unherd of in
earlier times are undertaken and
```

You can also use a second method to change the spelling of a word. This is the substitute command **s**. This command is very powerful, and probably is used more frequently than any other **ed** command.

The substitute command **s** is more complex than commands we have discussed so far, in that it has more elements, as follows: First is a line number or optional range of line numbers. Then comes the letter **s**, to invoke the substitute command itself. Third comes two *patterns* or *strings*, which are set off from the rest of the command and from each other with the **slash** character. For example:

```
1,$s/pattern1/pattern2/
```

Here, *pattern1* represents the string that you want **ed** to replace, and *pattern2* is the string that **ed** is to substitute in place of *pattern1*. Note that three slashes separate the two patterns from the **s**, from each other, and from the end of the line. These slashes must always be present.

With this command, you can correct the second spelling error in the example4:

```
3s/herd/heard/
p
```

ed replies:

```
software projects unheard of in
```

Note that these two command lines can be condensed to one:

```
3s/herd/heard/p
```

The meaning of these commands is: on the third line of the file, change **herd** to **heard** and, when finished, print the entire line. Without the **p** command, **ed** will change the line as you direct, but will not show you the new line. It is a good idea to print lines that you substitute in this manner until you gain in confidence with **ed**. Some **ed** experts always print the lines after substitution.

Type

```
.
w sample.text
q
```

to stop entering text, then save the newly typed text into file **sample.text** and exit ("quit") from **ed**.

After these two changes, the file looks like this:

```
Software technology today has
advanced to the point that large
software projects unheard of in
earlier times are undertaken and
```

Although the above example substitutes one word for another, note that the **s** command can replace any consecutive group of characters with any other: it may be one word, several words (including the space characters that separate them), or a fragment of a word.

Because **ed** looks for patterns rather words, you should keep in mind that it may find the wrong pattern. For example, assume that the current line in a file is

```
let not rain fall on a parade
```

and instead you want to say:

```
let not rain fall on the parade
```

You command **ed** to:

```
s/a/the/p
```

and are shocked to discover that the result is:

```
let not rthein fall on a parade
```

A better command to give **ed** would have been a substitute command that substituted the letter **a** preceded and followed by a space:

```
s/ a / the /p
```

Another correct way to do this task is to indicate within the substitution command which of several possible matches within the line is to be substituted. In our example, it is actually the third **a** that we are trying to match, so we could have used the special form of the command

```
s3/a/the/p
```

to get **ed** to select the one we wanted.

Undoing Substitutions

If you did change **a** to **the** inappropriately, you can retract the substitution by issuing the undo command

```
u
```

before you move on to another current line.

To illustrate this, enter this example:

```
ed
a
let not rain fall on a parade
.
w undo
q
```

Now, perform the substitution with

```
ed undo
s/a/the/p
```

which will result in:

```
let not rthein fall on a parade
```

To retract the substitution, simply type:

```
u
p
```

This undoes the substitution and prints the result.

Note that the undo command undoes the substitution only on the current line. Remember that if your substitution command operated over a range of lines, when it finishes the current line is the last one upon which the substitution was made. Thus, if you made an inappropriate substitution over a range of lines, the undo command will fix only the last line.

Global Substitutions

As you saw with the above examples, the **s** command substitutes only the *first* occurrence of the requested pattern on a given line.

A different form of the substitute command finds every occurrence of the indicated string on a line. Simply add the letter **g** for *global* after the third slash in the substitute command, and **ed** finds and changes every one:

```
s/pattern1/pattern2/g
```

So, if the current line contains a phrase:

```
a rose is a rose is a rose
```

and we tell **ed** to substitute

```
s/a/the/g
```

the line is changed to:

```
the rose is the rose is the rose
```

Again, be careful that your command does not inadvertently match all or part of a word that you wish to keep untouched.

Special Characters

In its first two parts, the substitute command uses some special punctuation characters. They will be discussed below in detail. However, you should be aware of these characters and avoid them until you progress to the advanced section, for unless used properly, they will give you undesired results. The characters are:

```
[ ^ $ * . \ &
```

They are used in **ed** and other COHERENT programs to form complex patterns.

Ranges of Substitution

Perhaps you need to change several lines that have the same misspelling or need the same editorial change. **s** can do that for you also. Simply prefix the command **s** with the line-number range as you would do with **p**. Borrowing the “rose” example again, if the saying were typed:

```
a rose is
a rose is
a rose
```

then you could do the same change as before, but across the entire file by typing

```
1,$s/a/the/
```

Note that the **g** after the **s** command has been omitted here, because you know that the string that you want to change appears only once on each line.

If some of the lines do not have the string you want to change, **ed** will not complain that the string is missing. However, if none of the lines in the range has the requested string, **ed** will print a **?**.

Intermediate Editing

This section introduces the more advanced command features of **ed**. Although you have already learned enough about **ed** to become productive, this section covers additional features that will increase your editing power considerably.

This section discusses the following topics: relative line numbering, moving blocks of text, finding strings, using special characters in substitution and search commands, processing global commands, and marking lines.

Relative Line Numbering

As discussed in the previous section, most commands allow you to use line numbers to control their range of operation. Before the command you can enter a single line number; for example:

```
1p
```

This, of course, prints the first line of the file. You may also specify a *range* of line numbers, by entering two numbers separated by a comma. For example, if the file contains at least ten lines, the command

```
1,10p
```

prints the first ten lines of the file.

The period (dot) always represents the number of the current line. For example, to print the file from the first line through the current line, just type:

```
1,.p
```

A command used without a line number always acts on the current line only. For example, typing

```
p
```

is equivalent to typing:

```
.p
```

There is yet another level of shorthand to line numbering — the plus and minus characters. These characters indicate *offsets* from the current line. For example, the command

```
.+3p
```

prints the third line after the current line. Likewise, the command

```
.-1p
```

prints the line that precedes the current line. Note that using a line offset changes the current line to the one addressed. Thus, after the above command is executed, the current line will be the one that preceded the original current line.

You can abbreviate this notation still further by leaving out the dot. The commands

```
+p
-p
```

do the following: First, **ed** advances to the next line and prints it; then it backs up to the previous line (which was the original current line) and printing it.

You can place several of these commands on one line to move the current line multiple lines. To back up three lines and then print, type:

```
---p
```

Note that in the absence of any other command, **ed** defaults to the **p** command. Thus

```
---
```

is equivalent to

```
---p
```

and

```
5
```

is identical to:

```
5p
```

The print command has one more abbreviation. If **ed** is expecting a command from you and you type nothing except **<return>**, **ed** interprets this as a command to advance the current line to the next line and print it. This action is equivalent to

```
+
```

or

```
.+1
```

<return> is the shortest command in **ed**.

All of the abbreviations for line numbers can be used by other commands that expect a range of line numbers. For example, if you want to delete five lines centered about the current line, you could type:

```
.-2,.+2d
```

and you would get your wish.

Note that **ed** does not allow you to specify a line number that is beyond the range of the file; this is regardless of whether you are typing a line number or any form of abbreviated line numbering. For example, suppose the current line is the last line in the file and you type:

```
+
```

This tells **ed** to "advance one line then print"; however, this is impossible because you are at the last line of the file, so there is no next line to print. When you request an impossible line number, **ed** replies by printing a question mark. Note, however, that the current line is always be valid so long as the file has at least one line in it. Thus, unless the file is empty, the command

```
.
```

will never give an error message.

Changing Lines

Earlier, an example of spelling correction was solved two ways. The first way was the clumsy way of deleting a line and retyping the entire line. This strategy means much work to change a single letter, so the substitute command was introduced instead.

On occasion, however, it is handy to be able to change lines en masse — as was done by deleting then inserting. **ed** provides this power with the change command **c**. In general terms,

```
m,nc
new lines
to be inserted
.
```

removes lines *m* through *n*, and insert new lines up to the period in place of them.

Moving Blocks of Text

When handling text, you will often need to shift a block of text from one position to another. In a manuscript, for example, you may need to rearrange the order of paragraphs to increase clarity. In a program, you may need to rearrange the order in which procedures appear.

To allow you to do this easily, **ed** provides a move command **m** that moves a block of text from one point in the file to another.

m is different from the other commands that we have discussed so far, in that line numbers follow as well as precede the **m** command itself. The line number that follows the command gives the line *after* which the text is to be moved. So, the general form of the move command is

b,*emd*

which means "move lines *b* through *e* to after line *d*".

To see how this works, first build the following file:

```
ed
a
    This is a paragraph of natural language
text. Due to stylistic considerations, it
really should be the second paragraph.
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
.
w example5
q
```

The file **example5** contains two paragraphs, each three lines long. We will now move the first paragraph to after the second paragraph.

You can do this in either of two ways: you can move the first paragraph to after the second paragraph, or you can move the second paragraph to before the first paragraph. Either gives the same result, but the commands are somewhat different. To shift the first paragraph to after the second paragraph, type:

```
ed example5
1,3m$
*p
Q
```

Remember that **$** always represents the last line in the file. The result is:

```
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
    This is a paragraph of natural language
text. Due to stylistic considerations, it
really should be the second paragraph.
```

To move the second paragraph to before the first, type:

```
4,6m0
```

Note that the destination is 0, which means that the text is to be moved to immediately after line 0. Because there is no line number 0, the move command interprets this to mean the beginning of the file.

Of course, in our small example, line number abbreviations and knowledge of the current line may be used in a number of different ways to perform exactly the same action. For example,

```
1,3m.
```

says to move lines 1 through 3 of the file to the line after the current line. When you invoke **ed**, it always sets the line number to the last line in the file. Thus, this form of the command has the same effect as the previous forms.

If the destination of a move command is not specified, **ed** assumes the current line. Therefore, the command

```
1,3m
```

also repositions the first paragraph correctly.

The move command changes the line numbers in the file, although the number of lines in the file remains the same. The different forms of the move command will, however, yield different settings for the current line.

After a move command, the current line becomes the number of the last line moved. Thus, if you moved the first paragraph to after the second paragraph, the current line will be reset to the last line in the file — the original line 3. However, if you moved the second paragraph to before the first paragraph, the current line would be reset to line 3 — which was originally the last line in the file.

Copying Blocks of Text

The transfer command **t** resembles the move command, except that it copies text rather than moving it. When you move text, it is erased from its original position. When you copy text, however, the text then appears both in its original position and in the position to which you copied it. **ed** uses the term *transfer* rather than *copy* because the command **c** is already used as the change command.

The form of the transfer command is as follows:

```
b,etd
```

This means to transfer (copy) the group of lines that begins with *b* and that ends with *e* (inclusive) to after line *d*.

After copying the text, **ed** sets the current line to the last line copied.

String Searches

The methods of line location that have been discussed to this point all involve line numbers. They specified an absolute line number, a relative line number, or a shorthand symbol such as **.** or **$**.

Often, however, line numbers are not useful, because there is no easy way to tell what number a line has, how many lines ago a block of text began, and so on.

ed's solution to this problem is to locate a line by asking **ed** to search for a pattern of text. **ed** begins searching on the line that follows the current line, and looks for a line that matches the specified pattern. If it finds a line that contains the requested pattern, **ed** resets the current line to that line.

If **ed** encounters the end of the file before it finds a match, **ed** jumps to the first line in the file, and continues its search from there. If it finds no match by the time it returns to the line where the search began, **ed** gives up and issues an error message — the question mark **?**. Remember, if you type a question mark in response to an error message, **ed** will tell you in more detail what the error is.

What does it mean to “match” a pattern? The simplest meaning is that two patterns are the same — the strings have exactly the same characters in exactly the same order. To see how this works, type the following to create file **example6**:

```
ed
a
    This is an example that we will
use for string searching.  There
is much natural language here as well
as some genuine arbitrary strings.
890,;+   foxtrot
qwertyuiop ##
.
w example6
q
```

Now, to locate and print any line contains the pattern **fox**, type:

```
ed example6
/fox/p
```

In response, **ed** prints the line:

```
890,;+   foxtrot
```

Also, you can use string expressions to print a range of lines. For example:

```
/This/,/much/p
```

This prints:

```
  This is an example that we will
use for string searching. There
is much natural language here as well
```

That is, it printed all lines from the first line that contains the pattern **This** through the first line that contains the pattern **much**.

Pattern searches can also be combined with relative line numbers. If you have a Pascal program file with several procedures in it, but you find that you need to rearrange the procedures, you can combine the power of the move command with the string searches.

```
PROCEDURE A;
...
...
PROCEDURE B;
...
...
PROCEDURE C;
```

Assume that the section of text that begins with **PROCEDURE A** should follow the line that contains **PROCEDURE B**. The following command moves the text properly:

```
/PROCEDURE A/,/PROCEDURE B/-1m/PROCEDURE C/-1
```

This commands **ed** (1) to locate the chunk of text that begins with a line containing the pattern **PROCEDURE A** and ends with the line just before the first line that contains the pattern **PROCEDURE B**, and then (2) move that text to just before the first line that contains the pattern **PROCEDURE C**. As you can see, you can pack a lot of information into one **ed** command.

Let's look at this command in more detail, to see exactly how it works. First, remember that the move command **m** is defined as

b,*e*m*d*

where *b* indicates the first line of the text to be moved, *e* indicates the last line of the text to be moved, and *d* indicates the line that the moved text is to follow. Thus, *b* corresponds to the number of the line that contains **PROCEDURE A** and is the first line of the procedure in question. *e*, however, corresponds to the line before the **PROCEDURE B** begins, by virtue of the -1. Here is an example of mixing pattern searches with relative line numbers, as mentioned above. Thus, you have found the beginning and ending lines of procedure A.

The final string search locates the first line of subroutine C. The move command normally moves text to **after** the given line; and because we wish to move the text to *before* the line that contains **PROCEDURE C**, we must include the -1 to move the text up one line.

Remembered Search Arguments

As discussed earlier, line numbers may be abbreviated in many ways. They may be entered as **.**, or **+**, or **-**, and certain combinations of these. With some commands, pressing **<return>** tells **ed** to use the current line number.

ed encourages you to abbreviate the search string. If you enter no string between the slashes in a search or substitution, then **ed** uses the last-used search string. A common use is in the global substitution command (which will be discussed in detail later in this section):

```
g/please remove this string/s// /p
```

This does not quite remove it, but replaces it with a blank. The last-used string can be specified by a string search, a substitute command, or a reverse string search (also discussed later in this section). Also, the remembered search argument may also be used in any one of these. You can use the remembered search feature to "walk" through the file, finding the next occurrence of a remembered search pattern.

Uses of Special Characters

As powerful as the line locator seems, some features are even even more powerful. These will be discussed in the Expert Editing section, below. However, these more powerful capabilities depend upon certain punctuation marks used in a special way. As you use the line locator (as well as the substitute command), be aware of these following characters:

```
[ ^ $ * . \ &
```

They have special significance to **ed** when they appear in a string search or a substitution pattern.

If you need to use one of these characters without invoking its special meaning, precede it with a backslash '\'. This tells **ed** not to interpret the character in a special way.

For example, to find a backslash character, type the search command:

```
/\\/
```

If any of these characters is to be used in another context, for example, within lines that you are adding with the **a** command, it should *not* be preceded with the backslash. Only use the backslash to hide the meaning when it appears within the string search command, or within the first part of the substitution command.

Global Commands

The global commands **g** and **v** let you repeat commands on all lines within a specified range. For example, to print all lines that contain the word **example**, type:

```
g/example/p
```

The global command can prefix almost any command. For example, the following command deletes all lines that contain three consecutive plus signs:

```
g/+++/d
```

Likewise, the command

```
g/foxtrot/.-2,.+2p
```

prints the five lines that surrounds any line that contains the word **foxtrot**.

A common use of the global command is to perform global substitution. The command

```
g/PROCEDURE/s/PROCEDURE/PROC/gp
```

performs the substitution on each line that contains the string **PROCEDURE** and prints the resulting line.

This may appear similar to the command

```
1,$s/PROCEDURE/PROC/gp
```

but is different in that the global command prints each of the changed lines, whereas the substitute command prints only the last line changed. Also, the method of operation of these two commands is different.

A related command **v** performs much the same task, but executes the commands only for lines that do *not* contain the specified string. Thus, to print all the lines that do not have the letter **w**, use:

```
v/w/p
```

For more sophisticated uses of the **g** and **v** commands and how they work, see the section on Expert Editing.

Joining Lines

What do you do if you inadvertently hit **<return>** as you are adding lines and need to combine the two lines?

```
ed
a
Look out, I seem to have hit ret
urn in the
middle of a word and don't know
what to do!
.
w rid
q
```

Rather that retyping the entire line, you can use the join command **j**:

```
ed rid
1,2j
1,$p
```

This gives:

```
Look out, I seem to have hit return in the
middle of a word and don't know
what to do!
```

If no line number is specified, **j** joins the current line and the following line. If a single line number is specified, join operates on that and the following line.

Several lines can be joined by using the form of the command:

a,*b*j

This joins lines *a* through *b* into one line. Likewise, the command

```
1,$j
```

joins all the lines in the file into one line. Then, the command **.p** or **p** prints the entire file.

Note that the command

```
3j
```

does the same job as the command

```
3,4j
```

The join command generates its own second line number if none is specified, so that the command

*n*j

is equivalent to

n,*n*+1j

where **n** is a line number. This command is the only one that interprets a missing line number this way.

Splitting Lines

You can split one line into two with the substitute command **s**. To illustrate, suppose you typed in the following commands:

```
ed
a
This line wants to be two, with this second.
.
w split
q
```

To perform the split, type:

```
ed split
s/two, /two,\
/p
*p
w
q
```

The line split is caused by the backslash that precedes the **<return>**. This tells **ed** that the **<return>** does not terminate the command, but that it is part of the substitution. The contents of file **split** are now:

```
This line wants to be two,
with this second.
```

Marking Lines

As you edit a manuscript or program, it is sometimes handy to be able to leave a “bookmark” in the text for later reference. **ed** provides this feature with the mark command **k**. To mark the next line that has the word **find**, use

```
/find/ka
```

where the letter **a** is the mark. To print the line that has been so marked, use:

```
'ap
```

You can place these references anywhere that a line number is expected.

The mark must be one lower-case letter. Also, each mark is associated with one line. Marking a line with the **k** command does not change the current line.

Marks can be especially handy when you move paragraphs with the **m** command. They give you a chance to review the sections that you will be moving before you do the move.

For example, suppose that you have a manuscript with a paragraph that must be moved to a different part of the document. Create the following example:

```
ed
a
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
text
text
text
    This is the spot that we want the paragraph
to precede.
.
w example7
q
```

Now, place three marks to help with the move:

```
ed example7
/first line,/ka
/Next paragraph/kb
/is the spot/kc
```

This marks the first line to be moved with **a**, the line after the last to be moved with **b**, and the paragraph's destination with **c**. But you can see that the move command moves lines to the line *after* the third number specified, so let's change the third mark:

```
'c-1kc
```

Now we can use **c** in the move command without arithmetic. Now, print the paragraph to be moved to be sure that the marks are correct.

```
'a,'bp
```

ed replies with

```
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
```

You can see that we would move one line too many if we used the marks as they are. So, change **b** also.

```
'b-1kb
```

Now, do the move:

```
'a,'bm'c
1,$p
```

The file now contains:

```
    Next paragraph begins here.
text
text
text
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    This is the spot that we want the paragraph
to precede.
```

Marking sections of text can increase the ease with which you solve your complex **ed** problems.

Searching in Reverse Direction

All scanning, processing, and searching has been shown going from the beginning of the file toward the end. Sometimes it is useful to find some word that occurs **before** the current line.

You can get **ed** to do string searching in the reverse direction by specifying the search with question marks **?** rather than slashes **/**. To find the previous occurrence of the word **last**, use:

```
?last?
```

This form of searching can be useful in finding the beginning and end of a **repeat/until** statement. For example, if the current line is in the middle of a Pascal **repeat/until** group, you can print the group with the command:

```
?repeat?,/until/p
```

The reverse search is like the forward search in every way except the direction of search. The search begins one line before the current or specified line, and proceeds toward the beginning of the file. If the string is not found by the time that the search reaches the beginning of the file, the search resumes at the end of the file, and progresses towards the starting point of the search. If the string is not found when the search reaches the original starting point, the question-mark error message is issued signifying no match.

Also, the command

```
??
```

uses the remembered search argument.

Expert Editing

This section describes the most advanced **ed** commands.

File Processing Commands

Earlier, we discussed the commands

```
ed
```

and:

```
ed filename
```

ed also has file-handling commands that go beyond those already discussed.

Suppose that you entered the command

```
ed file1
```

only to discover when you examined the contents of **file1** that you really wish to edit file **file2**. You could correct this error by exiting from **ed** and then re-invoking it for **file2**. However, **ed** command **e** lets you close out the current file and begin to edit a new file without exiting from the editor. For example, to stop editing **file1** and begin to edit **file2**, simply issue the command:

```
e file2
```

If you had made any changes to **file1**, **ed** will prompt you with a **?**, which is its way of asking if you wish to throw away the changes you have made to this file. If you immediately repeat the command, **ed** proceeds even if there are unsaved changes. The command

```
E new
```

commands **ed** to edit the new file, whether or not there are unsaved changes.

ed's "read" command **r** also reads a new file, but adds it to the file being edited instead of replacing the current file with it. This can be handy for copying one file into another one. For example, if you have a manuscript prefix stored in the file **prefix** to include the prefix at the beginning of the file you are editing, type:

```
0r prefix
```

r inserts the file being read after the line number specified; in this case, line 0 means at the beginning of the file. If used without a line number, **r** appends the newly read lines to the end of the file.

ed's command **w** writes the entire file if no line number is specified; however, you can specify line numbers. For example

```
1,3w new
```

writes the first three lines to file **new**. If the file name is omitted, the lines are written to the remembered file name.

The **w** command is unique in that it never changes the current line. This is true regardless of what line numbers are specified in the range for the command, or how those line numbers were developed.

The **W** command resembles the **w** command, except that it appends lines to the end of the file, whereas **w** creates a new file and erases any previous contents.

The **f** command prints the remembered file name that was set in

```
ed filename
```

or

```
e filename
```

or

```
w filename
```

commands. You can also use **f** to reset the remembered name, by typing:

```
f newname
```

This form of the command tells you what the new remembered file name is, even though you just typed it in.

Note that the command

```
w filename
```

changes the remembered name only if there is currently no remembered name, as does the **r** command.

Patterns

Earlier, you were cautioned that certain punctuation characters have special effect in search and substitute commands. These characters are:

```
[ ^ $ * . \ &
```

They are used to form powerful substitute and locator commands. An orderly combination of these special characters is called a *pattern*, sometimes called a *regular expression*. You can use a pattern to find or *match* a variety of strings with one search argument.

The simplest patterns use alphabetic characters and numeric digits, which match themselves. For example,

```
/ab/
```

finds and prints the next line containing the string **ab**.

The next simplest character to use in a pattern is the period or dot. It matches any character except the **newline** character that separates lines. Two periods in succession match any two consecutive characters, and so on. For example, if you have a file that contains algebraic statements of the form

```
a+b
c+e
a-b
a/b
d*e
```

and wanted to find and print any line involving **a** and **b** (in that order), then use the search statement:

```
/a.b/
```

The **.** in this example matches **+**, **-**, and **/**.

Then, you ask, how do I find a string that contains a period? For example, if you want to find all the sentences that ended with "lost." (that is, the word **lost** followed by a period), you might first try:

```
/lost./p
```

This, however, also matches the string "lost " (the word **lost** followed by a space), which is not what you want.

This is where the special character backslash comes in handy. A backslash tells **ed** to treat the next character as a regular character, even if it usually is a special character. Thus, to find "lost.", you need only type:

```
/lost\./p
```

This will not incorrectly find "lost ". If you want to find backslashes in your file, simply say:

```
/\\/p
```

Matching Many With One Character

The asterisk * matches an indefinite number of characters. For example, to remove extra spaces between words in a document, type

```
g/##*/s//#/p
```

(The character **#** has been substituted here for the space character to make the example more readable.) This replaces each series of spaces by one space.

Note that there are two spaces before the * in the search string. This is necessary because the * matches any length of string, including zero. Therefore, searching for a space followed by any number of spaces finds strings that are at least one space long.

The * matches the longest possible string of the previous character. This requires careful attention on your part, because the string matched by * might be longer than your required string, or even zero in length. Either way could give you unexpected results.

If you have a line

```
a+b-c
```

in your file and want to change it to

```
a+c
```

type the command:

```
s/a.*c/a+c/p
```

However, if the line read instead

```
a+b-c*d+c
```

and you applied the command, the result would be

```
a+c
```

since the **.*** matches the longest string between any **a** and any **c**.

Beginning and Ending of Lines

The characters **^** and **$** match, respectively, the beginning and ending of a line. Thus, you can find and print all lines that end with a bang:

```
g/bang$/p
```

or those that begin with a whimper:

```
g/^whimper/p
```

These two characters can also help you find lines of specific length. If you need to see all lines exactly five characters long, the command

```
g/^.....$/p
```

does the trick. To find and delete all blank lines, type:

```
g/^ *$/d
```

Note that this time the * matches a string of zero spaces. However, this is correct, because a blank line includes lines that have nothing in them, as well as lines that contain only spaces.

Replacing Matched Part

In many cases of substituting, you find yourself extending a word, or adding information to the end of a phrase. This can lead to extensive retyping of characters. The special **&** character can help out.

This character is special only when used in the right part, or *pattern2* of the substitute command. It means "the string that matched the left part". For example, to add **ing** to the word **help** in the current line, use:

```
s/help/&ing/
```

The ampersand may appear more than once in the right side.

This can be more interesting if the left part has a non-trivial *pattern*. For every word in a line that is preceded by two or more spaces, double the number of spaces before it:

```
s/###*/&&/gp
```

(Again, spaces have been replaced with # for clarity.)

Replacing Parts of Matched String

A more sophisticated feature, which is similar to the ampersand, helps you to rearrange parts of a line. To see how this works, create a file by typing:

```
ed
a
first part=second part
.
w eql
q
```

Two special bracket symbols, **\(** and **\)** can be used to delineate patterns in the left part of a substitution expression. Then, you can use the special symbols **\1**, **\2**, etc., to insert the delimited parts. The symbol **\(** marks the beginnning of the pattern, and **\)** marks the end. For example, to delete everything in the line except the characters to the left of the **=**, type

```
ed eql
s/^\(.*\)=.*/\1/p
Q
```

In the substitute command, the **^** matches the beginning of the line, **.*** matches "first part", and **=.*** matches the rest of the line. The symbol **\1** signifies the matched characters between the first **\(** (the only one in this example) and **\)**. The **p** then prints the result, which will be:

```
first part
```

To interchange the two parts, type

```
ed eql
s/\(.*\)=\(.*\)/\2=\1/
p
wq
```

The result is

```
second part=first part
```

The first portion of the substitution expression,

```
\(.*\)=\(.*\)
```

can be thought of as being in three parts. The first part

```
\(.*\)
```

matches all characters up to but not including the **=**, which are

```
first part
```

The second part

```
=
```

matches the **=** in the line, and finally the third part

```
\(.*\)
```

matches all characters following the "=", or

```
second part
```

The remainder of the substitution expression

```
\2=\1
```

which is the replacement part, rebuilds the line in interchanged order. The symbol **\2** replaces the matched string enclosed in the second pair of **\(\)** delimiters, and the symbol **\1** inserts the matched string enclosed in the first pair of **\(\)**.

The right side of the substitution inserts the second matched expression (from **\2**), then inserts the **=** sign again, followed finally with the first part of the line from **\1**.

This may appear involved, but can be immensely valuable in situations that require rearrangement of a large number of lines.

The next special characters for patterns that we will consider are the bracket characters **[** and **]**. These are used to define the character class. Inside the brackets, put a group of characters; **ed** will match any of them if it appears. For example, to print a line that contains any odd digit, say:

```
g/[13579]/p
```

For even more power and flexibility, you can combine character classes with the asterisk. For example, the following command finds and prints all lines that contain a negative number followed by a period:

```
g/-[0123456789]*\./p
```

This matches lines containing the following example strings:

```
-1.
-666.
-3.7.77
```

You can also match all lower-case letters by listing them in brackets, but the following abbreviation simplifies this:

```
g/[a-z]/p
```

This can also be used for the negative number example above:

```
g/-[0-9]*\./p
```

Most special characters lose their original meaning within the brackets, but one of the special characters, **caret** ^, gets a new meaning. In this context, it matches all characters *except* those listed in the brackets. For example, the following pattern matches a string that begins with **K** and continues with any character except a number:

```
/K[^0-9]/
```

This matches:

```
KQ
KK
KK9
```

but not:

```
K7
kK0
```

Other special characters may be part of a character class, but lose their special meaning when used in that context. Remember, however, that if you want to match the right bracket, it must appear first in the list. So, to find all occurrences of special characters in the file, type:

```
g/[]^\.*[&]/p
```

Listing Funny Lines

The **p** command prints lines with graphic characters in them. It also prints lines with non-graphic (or *control*) characters, but these do not appear on the screen. For example, printing a line that contains the BEL character **<ctrl-G>** will ring your terminal's bell, but you will not see where the BEL character occurs within the line.

The **l** command behaves like the **p** command, except that it also decodes and prints control characters. For example, if you use the **l** command to print a line that containing the word **bell** followed by a BEL character, you would see:

```
bell\007\n
```

Note that "007" is the ASCII value for **<ctrl-G>**. (ASCII is the system of encoding characters within your computer; see **ASCII** in the Lexicon for the full ASCII table.) The **l** command displays the backspace character **<ctrl-H>** as a hyphen first overstruck with a **<** and then a newline, which appears as **\n** on your screen. It displays a tab character as a **-**, first overstruck with a **>** and followed by a newline character, which appears as **>\n**. If the line being listed with **l** is too long to be displayed on one line on your screen, **l** separates it into two lines, with the backslash character placed at the end of the first line to indicate the split.

All other features of the **p** command apply to the **l** command.

Keeping Track of Current Line

The most commonly used abbreviation in **ed** is the dot, or period, which stands for the current line. Many commands can change the value of the dot, and it is useful to you to be able to anticipate this change when using the abbreviation.

Different classes of commands affect the value of the dot in different ways; in general, however, the simple explanation is usually correct: the current line is the last line processed by the last command to be executed.

Consider, for example, how the substitution command **s** changes the current line:

```
1,$s/flow/change/
p
```

In this example, the current line will be the last line modified by the substitutions; and that will be the line that the **p** command prints.

The **w** command is an exception to this rule. It does not change the current line, regardless of any line range selection or how these ranges are developed.

The **r** command changes the current line to the last of the lines read.

The **d** command sets the current line to the line after the last line deleted unless the last line in the file was deleted, in which case the new last line becomes the current line.

The line insertion commands **i**, **c**, and **a** all leave the current line as the last line added. If no lines are added, however, their behaviors differ: **i** and **c** effectively back up the last line by one, whereas **a** leaves it the same.

When Current Line Is Changed

When the current line changes is also important. Normally, the current line does not change until the command is completed.

To illustrate, create a file **semi** by typing:

```
ed
a
begin
second
first
in between
second
last
.
w semi
q
```

Now, edit the file and type all lines from **first** to **second**:

```
ed semi
/first/,/second/p
Q
```

This will cause an error! The reason is that the search command begins with current line set to **$**, so "first" is found on line 3. But the search for "second" also begins with the current line set at **$**, and finds "second" on line 2. Thus, the command translates to

```
3,2p
```

which is clearly invalid.

To do what was intended, use the **semicolon ;** instead of the comma to separate the two searches. This forces **ed** to change the current line to be changed after the search for **first** rather than after the entire command. Thus, the commands

```
ed semi
/first/;/second/p
Q
```

are correct and will do what is intended. The result will be:

```
first
in between
second
```

The search for **first** still begins with the current line set at **$**. However, after it finds **first**, **ed** resets the current line to 3, and begins the search for **second** there, and succeeds on line 5.

Finally, to be sure of where the current line is, you can use the **p** command to show you the line; or you can have **ed** tell you the number of the current line by typing:

```
.=
```

To give you a perspective on where you are with respect to the end of the file, type

```
&=
```

and **ed** will tell you the number of the last line in the file.

You can put any line number expression before **=** and it will type the result. For example

```
/next/=
```

types the number of the next line to contain "next" (if there is one). The command **=** never changes the line number.

More About Global Commands

All the global commands discussed thus far have been followed by only one command — substitute, print, and delete. You can, however, put several commands after a global command, and execute each of those commands for each line that matches.

To change all occurrences of the word **cacophonous** to the word **noisy** and print the three lines that follow, issue the command:

```
g/cacophonous/s//noisy/\
.+1,.+3p
```

Here, the additional commands are separated by the backslash before the **<return>**. Several commands can be added, and all but the last need the backslash at the end.

This will work for the line-adding commands, as well. To insert a spelling warning before each line that contains the word **occurrance**, issue the command:

```
g/occurrance/i\
((the following line needs spelling check))\
.
```

Note that the last line of the **i** group can be entered without a backslash, in which case the line containing only the period must be omitted. This has the same effect as:

```
g/occurrance/i\
((the following line needs spelling check))
```

You should not depend upon the setting of the current line in any multiline global command. There are two reasons for this. First, if one of the commands is a substitute and the string is not found in the matched line, the current line will not be changed.

Second, the global command operates in two phases. The first part scans the file for lines that match the string argument. **ed** marks these lines internally in a manner similar to the **k** command. The second phase then executes the commands on each of the marked lines. Therefore, you cannot count on a string search following the **g** to set the current line number.

Again, the **v** command behaves in the same way, except that it selects lines that do *not* match the pattern.

Caution is advised when using remembered search arguments, for a similar reason. A search argument is remembered only if the search has been executed. Thus, in a command of the form

```
g/backup/s//reverse/\
s/backin /backing/
```

the first remembered search may use **backup** on some occasion, and “**backin**” on others. The reason for this is that the second phase of the **g** command begins with a remembered search argument of **backup.** After the second line of the multiline command executes, the remembered search argument is “**backin** ”. This remains throughout the remainder of the second **g** phase.

Thus, it is recommended that you avoid remembered search arguments when using multiline global commands.

Issuing COHERENT Commands Within ed

While you are using **ed**, you can issue COHERENT commands by prefixing them with the **!** command.

This can be useful if, for example, you need to discover a file name while in the middle of an edit, and you want to find it without leaving **ed**. Thus, to list your directory while in **ed**, type:

```
!lc
```

ed sends the command to COHERENT and echoes a **!** character when the command is finished.

There is no limitation on the type of command that you may issue with this feature. It is even plausible that you want to start another **ed**.

For More Information

The Lexicon article on **ed** summarizes its commands and options. The COHERENT system also includes three other useful editors: **sed**, the stream editor; MicroEMACS, the screen editor; and **vi**, a clone of the standard UNIX screen editor. MicroEMACS and **sed** are introduced with their own tutorials, and each is summarized in the Lexicon.