Introduction to the sed Stream Editor

This is a tutorial for the COHERENT editor **sed**. It describes in elementary terms what **sed** does.

This guide is meant for two types of reader: the one who wants a tutorial introduction to **sed**, and the one who wants to use specific sections as references.

Related tutorials include *Using the COHERENT System*, which presents the basics of using COHERENT and introduces many useful programs, and the tutorials for the interactive line editor **ed** and for the screen editor MicroEMACS.

In a nutshell, **sed** edits files non-interactively; that is, **sed** applies your set of commands to every line of the file being edited. It is not meant to create a text, as you can do with **ed**, **me**, or **vi**. Rather, it lets you perform large, intricate transformations on a file of text, using commands that resemble those used by **ed** or **vi**'s colon-command mode.

Although **sed** is not as easy to control as **ed** or MicroEMACS, both of which are interactive, it can edit a large file very quickly. You can use **sed** to change computer programs, natural language manuscripts, command files, electronic mail messages, or any other type of text file.

One last point: **sed** normally writes its output to the standard output, which by default is your screen. To save its output into a file, use the shell's '>' operator to redirect the standard output into a file.

Getting to Know sed

sed is a text editor. It reads a text file one line at a time, and applies your set of editing commands to each line as it is read. Because it does not ask you for instructions after it executes each command, **sed** is a *non-interactive* text editor.

The advantages of **sed** are that it can readily apply the same editing commands to many files; it can edit a large file quickly; and it can readily be used with *pipes*. A pipe takes the product of one program and feeds it into another program for further processing. If you are unsure of how a pipe works, refer to *sh Shell Command Language Tutorial*.

sed resembles closely **ed**. **sed** and **ed** use almost all of the same commands, and locate lines in much the same way. However, there are important differences between **ed** and **sed**. **ed** is interactive: when you give **ed** a command from the keyboard, it executes that command immediately and then waits for you to enter the next command. **sed**, on the other hand, accepts your editing commands all at once, either from the keyboard or, more often, from a file you prepare; then, as it reads your text file one line at a time, it applies every command to every line of text. Therefore, *addressing* (that is, telling the program what commands should be applied to which lines) is much more important with **sed** than with **ed**.

Keep in mind, too, that **sed** does not change your original text file; rather, **sed** copies it, changes it, and sends the edited file either to the standard output or to another file that you name in the command line.

Getting Started

Here are a few exercises to introduce you to **sed**. Type them into your COHERENT system to get a feel for how **sed** works.

As explained above, **sed** applies a set of editing commands to your text file. To edit a file with **sed**, you must prepare three elements: (1) the text file that you wish to edit; (2) a command file (or *script*) that contains the **sed** commands you want to apply to the text file; and (3) a command line that tells the COHERENT system what you want done and with which files.

To begin, then, type the following text into your computer using the **cat** command. (Remember that **<ctrl-D>** is typed by holding down the *ctrl* key and simultaneously typing *D*.)

```
cat >exercise1
No man will be a sailor who has contrivance enough
to get himself into a gaol; for being in a ship is
being in a gaol, with the chance of being drowned.
<ctrl-D>
```

Now, type in the following **sed** script. This script will substitute *jail* for *gaol*:

```
        cat >script1
        s/gaol/jail/g
        <ctrl-D>
```

The last step is to prepare the command line. The command line consists of the **sed** command, the options that tell **sed** where its instructions will be coming from (either from a file or directly from the command line), the name of the file to be edited, and where the edited file should be send. The general form of the command line is as follows:

```
        sed [-n] [-e commands] [-f scriptname] textfile [>file]
```

The **-n** option will be explained below, in the section on *Output*. The **-e** option tells **sed** that *commands* follow immediately. The **-f** option tells **sed** that the commands are contained in the file *scriptname*. *textfile* is the name of the text file to be edited. The greater-than symbol '>' followed by a file name redirects the edited version of the text file into *file*; if this option is not used, the edited copy of the text file will be sent to the standard output.

In this example, a command script has been prepared, so the **-f** option will be used. Also, the edited text should appear on the terminal screen, so the '>' will not be used. Type the command line as follows:

```
        sed -f script1 exercise1
```

The following text will appear on your screen:

```
        No man will be a sailor who has contrivance enough
        to get himself into a jail; for being in a ship is
        being in a jail, with the chance of being drowned.
```

You can use **sed** not only to substitute one word for another, but to add lines to files, delete lines, and perform more involved editing. No matter how complex your **sed** editing becomes, though, **sed** will always use the basic format just described.

The next few sections describe **sed**'s basic commands.

Simple Commands

Type in the exercises exactly as shown and examine the results. Use the **cat** command to enter the command file as well as the input file. The edited text will appear on your terminal. Usually when you edit, you will want to redirect the edited text to a new file; however, for the exercises presented here, let the edited text appear on your terminal so you can examine the results immediately.

Substituting

The substitution command is used very often when editing. **sed**'s substitution command **s** resembles the same command in **ed**. Its form is as follows:

```
        s/term1/term2/
```

This tells **sed** to substitute *term2* for *term1*. To correct a misspelled word, for example, use this command form:

```
        s/mispel/misspell/
```

As written, this command changes only the first occurrence of **mispel** in each line of your text file. To change *every* occurrence of *mispel* in each line, add **g** (the **g**lobal option) at the end of the command:

```
        s/mispel/misspell/g
```

If you want to change only the *third* occurrence of **mispel** on every line, put a **3** after the **s**:

```
        s3/mispel/misspell/
```

When no digit follows the **s** and no **g** follows the command, only the first occurrence of the term in each line (should there be one) will be changed.

To practice the substitution, type the following file into your system (please include the misspellings):

```
        cat >exercise2
        From the Devils Dictionary:
        Hemp, n. A plant from whose fiberous bark is made
        an article of neckware which is frequently put on
        after public speaking in the open air and prevents
        the wearer from tking cold.
        <ctrl-D>
```

Now, prepare the following **sed** script to correct the misspellings:

```
        cat >script2
        s/Devils/Devil’s/
        s/fiberous/fibrous/
        s/tking/taking/
        <ctrl-D>
```

Invoke **sed** with the following command:

```
        sed -f script2 exercise2
```

The following will appear on your screen:

```
        From the Devil’s Dictionary:
        Hemp, n. A plant from whose fiberous bark is made
        an article of neckwear which is frequently put on
        after public speaking in the open air and prevents
        the wearer from taking cold.
```

To see how the **g** command and the number option work, prepare the following text file:

```
        cat >exercise3
        sd      sd      sd      sd
        sd      sd      sd      sd
        sd      sd      sd      sd
        <ctrl-D>
```

The following **sed** script changes the *third* **sd** in each line to **sed**:

```
        cat >script3
        s3/sd/sed/
        <ctrl-D>
```

Invoke **sed** with the following command line:

```
        sed -f script3 exercise3
```

The following will appear on your screen:

```
        sd      sd      sed     sd
        sd      sd      sed     sd
        sd      sd      sed     sd
```

To change *every* **sd** to **sed**, use the **g** option.   Prepare the following **sed** script:

```
        cat >script3a
        s/sd/sed/g
        <ctrl-D>
```

The following will appear on your screen:

```
        sed     sed     sed     sed
        sed     sed     sed     sed
        sed     sed     sed     sed
```

The **g** command will be most useful for editing prose, when you have no way to tell how many times a given error will appear on a line. The numeric option will be most useful for editing tables and lists.

Selecting Lines

Each of the substitution commands given above will be applied to every input line. Unlike **ed**, there is no error message if no line of text contains *term1*.

In certain instances, however, you may wish to apply a particular command only to specific lines. Lines can be specified (or *addressed*) by *preceding* the command with the identifying line number. The following exercise demonstrates line selection. First, prepare the following text file:

```
cat >exercise4
When a man is tired of London,
he is tired of life; for there
is in London all that life can afford.
<ctrl-D>
```

To change the word **tired** to **fatigued** on line 2 only, prepare the following **sed** script:

```
cat >script4
2s/tired/fatigued/
<ctrl-D>
```

Begin the editing of your text file by typing the following command line:

```
sed -f script4 exercise4
```

The following will appear on your screen:

```
When a man is tired of London,
he is fatigued of life; for there
is in London all that life can afford.
```

Remember that to specify a line number, you must place the number *before* the command; but to specify the numeric option (that is, position within the line), you must place the number *after* the command.

You can define a *range* of lines to be edited. One way to do this is to list the first and last line numbers, separated by commas, of the block of text in question. For example, the following script will change **is** to **was** only in the first two lines of the text file you just prepared:

```
cat >script4a
1,2s/is/was/
<ctrl-D>
```

Entering the command line

```
sed -f script4a exercise4
```

will bring the following text to your screen:

```
When a man was tired of London,
he was tired of life, for there
is in London all that life can afford.
```

Note that the word **is** in line 3 was unaffected by the substitution command, because it lay outside the range of lines specified by the command.

You can also select lines by *patterns*. Patterns are *strings* (any collection of letters and numbers, such as a word) that can be combined with commands. A fuller description of *patterns* can be found in the tutorial for **ed**. Later on, when we show you other commands, you will see that line selection by pattern rather than by line number is quite useful.

You can use the end-of-file symbol '$' for line selection. When you use this symbol, you do not have to know the exact number of lines in your text file. For example, if you want to apply a substitution command from line 10 through the end of your text file, the command form is:

`10,$s/`*term1*`/`*term2*`/`

p: Print Lines

When **sed** edits a text file, the edited text is by default sent to the *standard output*, which usually is your terminal's screen. (As noted above, the edited text can be optionally redirected to another file by using the shell's '>' operator.) Normally, **sed** prints every line in the text file, whether the line is changed or not.

The next exercise will demonstrate these defaults. First, type in the following text file:

```
cat >exercise5
Bill          g7            r115
Nora          g8            r115
Steve         g7            r120
Ella          g8            r120
Dave          g7            r115
Robert        g8            r120
<ctrl-D>
```

Next, create a script that contains no commands, by typing:

```
cat >script5
<ctrl-D>
```

Now, execute this empty script:

```
sed -f script5 exercise5
```

Note that **sed** simply copied your text file to the screen, without changing it in any way.

This default, however, can be inconvenient if you want to print only a selected portion of a file. Fortunately, with a couple of commands you can control **sed**'s printing.

The command line option **-n** changes **sed**'s printing behavior. When you invoke **-n**, the text file no longer is printed automatically. **sed** prints only the lines specified by the **p** command. The **p** command makes **sed** print whatever line (or lines) to which it is applied. Use **-n** on the command line to stop **sed** from printing every line automatically; then use the **p** command in the script to target the lines you want to print. The following exercise will help you grasp this point. First, type in the following **sed** script:

```
cat >script5a
/g7/p
<ctrl-D>
```

Enter the command line:

```
sed -n -f script5a exercise5
```

and the following text will appear on your terminal:

```
Bill          g7            r115
Steve         g7            r120
Dave          g7            r115
```

sed prints only the records of the students in grade 7 (**g7**).

It is important to note the order, or *syntax*, of the **-n** and **-f** command line options. The correct order is to enter **-n**, then **-f**. (**-nf** or **-fn** are also acceptable.) If you type **-f** and then **-n**, however, all you will get is an error message.

When you use the **p** option with a **sed** command, **sed** will print every line of text in which that command makes a substitution. This can be useful, but if you are not careful it can also create some problems. **sed** normally prints every line in your text file, whether or not it is changed by your script, unless you specify the **-n** option in your command line. Therefore, if you *do not* use the **-n** option in your command line and you **do** use the **p** option with your **s** commands, every line that **sed** edits will be printed more than once.

The following script illustrates this point:

```
cat >script5b
s/g7/g8/gp
s/r115/r120/gp
<ctrl-D>
```

Now, execute it with the following command:

```
sed -f script5b exercise5
```

The result will look like this:

```
Bill          g8             r115
Bill          g8             r120
Bill          g8             r120
Nora          g8             r120
Nora          g8             r120
Steve         g8             r120
Steve         g8             r120
Ella          g8             r120
Dave          g8             r115
Dave          g8             r120
Dave          g8             r120
Robert        g8             r120
```

Bill and **Dave** were printed three times: the first time because the **p** option was specified after editing the grade number, the second time because the **p** option was specified after editing the room number, and the third time because the **-n** option was *not* used on the command line. **Steve** and **Nora** were printed twice: the first time because their lines were edited once each, and the second time because the **-n** option was not used on the command line. **Ella** and **Robert** appeared once because their lines were not edited at all and the **-n** option was not specified in the command line.

To get around this problem, use the **-n** option and use **p** only once, on the last substitution:

```
cat >script5c
s/g7/g8/g
s/r115/r120/gp
<ctrl-D>
```

When you enter the following command line

```
sed -n -f script5c exercise5
```

the new result will be:

```
Bill          g8             r120
Nora          g8             r120
Dave          g8             r120
```

The **w** command acts like the **p** command, except that matched lines are written to the file whose name follows the **w**. The following script shows the correct form:

```
cat >script5d
s/g8/g9/w grade.9
s/g7/g8/w grade.8
<ctrl-D>
```

When you execute script5d with this command:

```
sed -f script5d exercise5
```

the normal product will be seen produced at your terminal, and the edited lines will be written to files **grade.8** and **grade.9**. File **grade.8** will contain:

```
Bill          g8             r115
Steve         g8             r120
Dave          g8             r115
```

Note the order in which the two **s** commands were given. If their order were reversed, every text line with **g7** in it would have **g7** changed to **g8** by the first **s** command, then have this newly created **g8** changed to **g9** by the second **s** command. Thus, *all* the students would be shown to be in **g9**, and every text line would be printed into the file **grade.9**.

Line Location

When you edit a file with **sed**, it is hard to keep track of line numbers. As noted earlier, you can locate specific lines with **sed** by using patterns as *line locators*. To see how this works, type the following text file into your system:

```
        cat >exercise6
        From the Book of Proverbs:
        As a door turneth upon his hinges, so the
        slothful man turneth upon his bed.
        A soft answer turneth away wrath:  but grievous
        words stir up anger.
        <ctrl-D>
```

Now, prepare the following **sed** script:

```
        cat >script6
        /door/,/bed/s/turneth/turns/
        <ctrl-D>
```

Execute it by entering the following command line:

```
        sed -f script6 exercise6
```

The text will appear on your terminal this way:

```
        From the Book of Proverbs:
        As a door turns upon his hinges, so the
        slothful man turns upon his bed.
        A soft answer turneth away wrath:  but grievous
        words stir up anger.
```

Note that the word *turns* was substituted for the word *turneth* only in the first proverb, not the second. The reason is that the **s** command in this instance was preceded by the *patterns* **door** and **bed**. These told **sed** to begin making the substitution on the first line in which the word **door** appears, and to stop making the substitution with the first line in which the word **bed** appears. In the text file, the fourth line also contained the word **turneth**, but because it lay outside the range of line specified by the line locators, no substitution was made.

When **sed** locates the last line of a block of text that you have defined, it will immediately look for the next occurrence of the first line locator. If it finds that first line locator, it will then resume making the substitution to your file until it again finds the second line locator or comes to the end of the file, whichever occurs first. In this example, when **sed** found the word **bed**, it began to look again for the word **door**; and if it had found the word **door**, it would have resumed substituting **turns** for **turneth**.

Remember that, as explained earlier, line numbers can also be used as line locators. For example, the **sed** script

```
        2,3s/turneth/turns/
```

would have produced the same changes as did the script with the pattern line locators prepared earlier.

Add Lines of Text

sed can add lines to your text file. To see how **sed** does this, first prepare the following text file:

```
        cat >exercise7
        From the Devil's Dictionary:
        Syllogism, n. A logical formula consisting of a major
        and a minor assumption and an inconsequent.
        <ctrl-D>
```

Now, type in the following script:

```
        cat >script7
        3a\
        Economy, n. Purchasing the barrel of whiskey you do not \
        need for the price of the cow you cannot afford.
        <ctrl-D>
```

When you implement the script:

```
        sed -f script7 exercise7
```

you will see this result:

```
From the Devil's Dictionary:
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
Economy, n. Purchasing the barrel of whiskey you do not
need for the price of the cow you cannot afford.
```

The append command **a** added text *after* the third line of the file. You defined where the text went. Notice the backslash '\' at the end of the line with the **a** command. This indicates that the next line is part of the command. When you append several lines of text, each line but the last one to be added must end with a '\' as in our example.

Note that no other editing command, such as **s**, can affect any line added with **a**. These lines go directly to your screen, or to a file, should you be sending the edited text there, and are invisible to all other **sed** commands.

The insert command **i** works like the **a** command, except that it adds its lines *before* the addressed line, rather than after. The following script shows how the **i** command works:

```
cat >script7a
2i\
Peace, n. In international affairs, a period of cheating\
between two periods of fighting.
<ctrl-D>
```

Invoking it with this command:

```
sed -f script7a exercise7
```

produces this:

```
From the Devil's Dictionary:
Peace, n. In international affairs, a period of cheating
between two periods of fighting.
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
```

As with the **a** command, no substitutions or other changes are performed on lines added with **i**.

Note, too, that you can *bracket* a text line by using the **a** and **i** commands at the same time. Adding a line with either **a** or **i** does not change line numbers of the text file you are editing (although it does, of course, change the line numbers of the file **sed** writes).

Delete Lines

The **d** command deletes lines that you do not want in the edited text. The original file stays unchanged, of course.

Lines that match the address (be it a line number, range, or pattern) of a **d** command do not appear in the output. Exercise 8 illustrates the **d** command:

```
cat >exercise8
The sun was shining on the sea,
Shining with all his might.
He did his very best to make
The billows smooth and bright --
And this was odd, because it was
The middle of the night.
<ctrl-D>
```

Now, you have to define the lines to be deleted by matching them with a unique pattern or a line number. To delete lines 3 through 6, prepare this script:

```
cat >script8
/best/,/night/d
<ctrl-D>
```

The command:

```
sed -f script8 exercise8
```

generates this result:

```
The sun was shining on the sea,
Shining with all his might.
```

Note that when a line is deleted, no other commands are applied to it. Usually, if a **sed** script holds a number of commands, every one of those commands is applied to every line read from your text file; however, **sed** is logical enough to read the next text line immediately, should a **d** command delete the current line before the series of commands has finished.

Change Lines

The **c** command combines the **i** and **d** options. Text is inserted before the addressed lines, which are then deleted. To see how this command works, prepare the following text file:

```
cat >exercise9
Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
<ctrl-D>
```

Now, type in the following script:

```
cat >script9
1,2c\
Twas brilliant, and the shining cove\
Did glare and glimmer in the wave;
<ctrl-D>
```

When you execute your script with the following command line:

```
sed -f script9 exercise9
```

the result is:

```
Twas brilliant, and the shining cove
Did glare and glimmer in the wave;
All mimsy were the borogoves,
And the mome raths outgrabe.
```

Like the **i** and **a** commands, the **c** command requires all added lines but the last to end with '\'.

Include Lines From a File

When you edit a file, you may wish to include, or *read in*, a second file as part of it. This is done with **r** command. To see how this works, type the following file into your computer, and call it **include**:

```
cat >include
        Then there comes the often-used refrain
        Whose repetitious writing dulls the brain.
<ctrl-D>
```

Now, prepare the file to be edited:

```
cat >exercise10
To write a poem doesn't take much time;
Just string some words to rhythm and a rhyme.
What poets do to language is a crime,
Words and syntax twisted for a rhyme.
<ctrl-D>
```

When you write your script, you must tell **sed** where to read in **include**. The form of the command should be familiar by now:

```
cat >script10
/rhyme/r include
<ctrl-D>
```

The result is of

```
sed -f script10 exercise10
```

is:

```
To write a poem doesn't take much time;
Just string some words to rhythm and a rhyme.
       Then there comes the often-used refrain
       Whose repetitious writing dulls the brain.
What poets do to language is a crime,
Words and syntax twisting for a rhyme.
       Then there comes the often-used refrain
       Whose repetitious writing dulls the brain.
```

Note that the **r** command inserted **include** *after* the addressed line. You can address lines by number, of course, as well as by pattern.

Quit Processing

The **q** command makes **sed** stop processing the text file. You will use this command most often to limit the application your **sed** script to a portion of your text file. For example, if you were editing a large file and you knew that your commands would be irrelevant to the last half of the file, you could insert an appropriately addressed **q** and save some computer time. You can also use this command to print portions of a file.

To see how this is done, prepare the following text file:

```
cat >exercise11
An hourglass has a very wide top,
       a very narrow
           middle
        and a bottom
that is also extremely wide.
<ctrl-D>
```

The following script will print the top of the text file. Note how the script uses **middle** to address the line where the file is to be split.

```
cat >script11
/middle/q
<ctrl-D>
```

The command:

```
sed -f script11 exercise11
```

produces:

```
An hourglass has a very wide top,
       a very narrow
           middle
```

To print out only the lines *after* the pattern **middle**, simply delete the first half of the file with the **d** command, as follows:

```
cat >script11a
1,/middle/d
<ctrl-D>
```

The result is the output:

```
        and a bottom
that is also extremely wide.
```

Next Line

The **n** command advances to the next line of the text file. The **n** command is useful for instances when you have two or more interrelated lines, and you want to ensure th a particular set of patterns is matched over the entire set of lines. To see how **n** works, prepare the following text file:

```
cat >exercise12
Alpha
One
Beta
Two
Gamma
Three
Delta
Four
Epsilon
Five
<ctrl-D>
```

To print a list of letters alone, type the following script:

```
cat >script12
n
d
<ctrl-D>
```

and execute it with the following command line:

```
sed -f script12 exercise12
```

The result will be the following:

```
Alpha
Beta
Gamma
Delta
Epsilon
```

Remember that **n** does *not* stop processing, go to the next text line, and begin processing all over again. Rather, it simply reads the next input line and continues processing from where it left off. For example, if your **sed** file consisted of three commands, the second of which was the **n** command, **sed** would apply the first command to the first line it read, then jump to the second line and apply the last commands. Then, it would read the third line and begin the pattern over again. To see how this works, prepare the following text file:

```
cat >exercise13
Alpha
Alpha
Alpha
Alpha
<ctrl-D>
```

Now type in this script:

```
cat >script13
s/Alpha/Apple/
/Apple/n
s/Alpha/Banana/
<ctrl-D>
```

When you execute the script with this command line:

```
sed -f script13 script13
```

the following will appear on your terminal:

```
Apple
Banana
Apple
Banana
```

Note that the first substitution command changed the first **Alpha** to **Apple**; the **n** command moved **sed** to the next line; and the second **s** command changed that **Alpha** to **Banana**.

Advanced sed Commands

The following sections discuss **sed**'s advanced features. They also discuss the method of operation.

Work Area

As described earlier, **sed** reads your text file one line at a time, and applies all of your editing commands to that line. After the editing commands have been applied, the edited line is either sent to the *standard output*, written to a file you have named, or thrown away, depending on what you have told **sed** to do.

When **sed** reads a line from your text file, it copies that line into a *work area*, where it actually executes your editing commands. **sed** notes the number of the line in the work area, then executes each editing command in turn, first checking to see if the patterns or line numbers specified in each command actually apply to that line. After each command is checked in turn and performed if indicated, **sed** prints the edited line (if it is supposed to be), and reads the next text line.

Add to Work Area

The exercises so far have used only one line in the work area. The **N** command, however, tells **sed** to read a second line into the work area. The following exercise illustrates the use of the work area and the **N** command.

```
cat >exercise14
This exercise has a brok
en word.
<ctrl-D>
```

Now, prepare the following **sed** script:

```
cat >script14
/brok$/N
s/brok\nen/broken/
s/has/had/
<ctrl-D>
```

and execute it with the following command line:

```
sed -f script14 exercise14
```

which produces the following text:

```
This exercise had a broken word.
```

You will find it helpful to review this exercise in some detail. The first command in the script

```
/brok$/N
```

tells **sed** to search for the pattern **brok** at the *end* of the line of text. (The dollar sign '$' in this instance indicates the end of the line; remember that when the '$' is used with a line number, it indicates the end of the *file*.) The **N** command tells **sed** to keep this line in the working space, and copy the *next* line into the working space as well.

When **sed** executes this command on the present text file, the work area will look like this:

```
This example has a brok<newline>en word.
```

Note that the two lines now appear to **sed** as though they formed one long line. The word **<newline>** represents the end of line character that tells your terminal or printer to jump to a new line when the text file is printed out. This character is invisible, but it is there, and it can be changed or deleted. You can describe this character to **sed** by using the characters **\n**. The first substitution in this script

```
s/brok\nen/broken/
```

replaces **brok<newline>en** with **broken**. Because the newline character is deleted from the text, what used to be printed out as two lines on your screen will now be printed out as one.

Note the difference, too, between the **n** and **N** commands. The **n** command will *replace* the text line in the work area with the next line from your text file. The **N** command, however, *appends* the next line from your text file to the end of the text already in the working area. The next exercise demonstrates this difference. First, create the following text file:

```
cat >exercise15
Apple
Apple
Apple
Apple
<ctrl-D>
```

Now, prepare the following two scripts:

```
cat >script15
/Apple/n
s/Apple/Banana/g
<ctrl-D>
```

```
cat >script15a
/Apple/N
s/Apple/Banana/g
<ctrl-D>
```

When script15 is executed with the following command line:

```
sed -f script15 exercise15
```

this will appear on your screen:

```
Apple
Banana
Apple
Banana
```

The **n** command told **sed** to print out the line already in the work area before reading in the next line from the text file. This meant that **sed** substituted **Banana** for **Apple** only on the *second* line of each pair.

Note what happens, however, when you run script15a, using this command line:

```
sed -f script15a exercise15
```

This text appears:

```
Banana
Banana
Banana
Banana
```

Because *both* lines of each pair were kept in the work area, the substitution command changed both of them.

Print First Line

The **P** command prints material from the work area. Unlike the **p** command, which prints *everything* in the work area, **P** prints only the *first* line in the work area. To see how this works, prepare the following text file:

```
cat >exercise16
Student:  George
Teacher:  Mr. Starzynski
Student:  Marian
Teacher:  Miss Peterson
Student:  Ivan
Teacher:  Mr. Starzynski
<ctrl-D>
```

Now, prepare the following scripts:

```
cat >script16
/Student/N
/Mr. Starzynski/p
<ctrl-D>
```

```
cat >script16a
/Student/N
/Mr. Starzynski/P
<ctrl-D>
```

When the first of these scripts is executed with the following command line (note the use of the **-n** option):

```
        sed -n -f script16 exercise16
```

the result is

```
        Student:  George
        Teacher:  Mr. Starzynski
        Student:  Ivan
        Teacher:  Mr. Starzynski
```

whereas script16a, when executed as follows:

```
        sed -n -f script16a exercise16
```

produces

```
        Student:  George
        Student:  Ivan
```

In **script16**, the **N** command lines pull both the name of the student and the name of the teacher into **sed**'s work area; the **p** command prints the student and teacher in each case where the teacher is Mr. Starzynski. In **script16a**, however, the **N** pulled both student and teacher into the work area, the **P** command printed only the *first* line of the work area — that is, the name of the student.

As you can see, **P** is a powerful tool that will allow you to select material from tables, lists, and other repetitive files.

Save Work Area

sed can create a second work area in addition to the primary work area in which **sed** performs its editing. **sed** does not execute any editing commands on the material stored in this secondary work area; rather, this work area can be used to store material that you want to edit or insert later.

The commands **h** and **H** copy material from the primary work area into the secondary work area. **h** and **H** differ in that **h** *displaces* any material in the secondary work area with the line being copied there, whereas **H** *appends* the line being copied onto the material already in the secondary work area. Note, too, that both **h** and **H** merely *copy* the primary work area into the secondary work area — after these commands have been executed, the material in the primary work area remains intact, and can be edited further, printed out, or deleted, whichever you prefer.

The commands **g** and **G** copy material back from the secondary work area into the primary work area. Again, these commands differ in that **g** *displaces* whatever is in the primary work area with the material from the secondary work area, whereas **G** *appends* the material from the secondary work area onto the material already in the primary work area.

The following exercises will demonstrate how these commands are used. First, create the following text file:

```
        cat >exercise17
        fruit:  apple
        berry:  gooseberry
        fruit:  orange
        berry:  raspberry
        fruit:  pear
        berry:  blueberry
        <ctrl-D>
```

The first script uses the **h** and **g** commands:

```
        cat >script17
        /fruit/h
        /fruit/d
        /berry/g
        <ctrl-D>
```

When you execute this script with the following command line:

```
        sed -f script17 exercise17
```

you receive the following text on your screen:

```
        fruit:  apple
        fruit:  orange
        fruit:  pear
```

Review the last script in detail. The first command, **/fruit/h**, copied the line beginning with “fruit” into the secondary work area, displacing whatever happened to be there. The command **/fruit/d** then deleted the line from the primary work area; if this were not done, it would then have been printed out. The third command, **/berry/g** then recopied the material from the secondary work area into the primary work area, displacing all lines in the primary work area that begin with “berry”. The result of all this shuffling and displacing was that the three lines that begin with **fruit** were printed out.

The next script demonstrates the **H** command:

```
cat >script17a
/fruit/H
/fruit/d
/berry/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17a exercise 17
```

you see:

```
fruit:  apple
fruit:  apple
fruit:  orange
fruit:  apple
fruit:  orange
fruit:  pear
```

Because the **H** command *appends* material into the secondary work area, rather than replacing it as **h** does, all three lines that began with **fruit** were cumulatively stored in the secondary work area. Because the **g** command was used for every line that began with **berry**, the contents of the secondary work area (that is, the **fruit** lines) were written over each of the three lines that began with **berry**.

The next script demonstrates the use of the **G** command:

```
cat >script17b
/fruit/H
/fruit/d
/berry/G
s/berry://g
s/fruit://g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17b exercise17
```

you will see:

```
gooseberry
apple
raspberry
apple
orange
blueberry
apple
orange
pear
```

The **H** command copies the lines that begin with **fruit** into the secondary work area. The **G** command then re-copies them from the secondary work area into the primary work area, and appends them to the material already in the primary work area — that is, to a line that begins with **berry**.

The two substitution commands then strip off the **fruit** and **berry** prefixes; obviously, these substitutions do not affect the operation of the **H** and **G** commands, but they do create a tidier result.

By the way, be sure you distinguish the **g** command from the **g** option used with the **s** command. If you do not, what **sed** finally prints out for you may appear very strange.

The final command that uses the secondary work area is **x**, which exchanges the two work areas. The following script shows how this is used:

```
cat >script17c
/fruit/H
/fruit/d
/blueberry/x
s/berry://g
s/fruit://g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17c exercise17
```

you see:

```
gooseberry
raspberry
apple
orange
pear
```

The text lines that began with **fruit** were moved into the secondary working area. The **x** command was executed when the line that contained the word **blueberry** was reached, and the two working areas exchanged their contents. The **fruit** lines were then printed out, while the **blueberry** line was simply left in the secondary working at the end of the program, and disappeared when the program concluded.

Note that **x** simply swaps the two working areas — there is no "**X**" command that appends the work areas onto each other.

Transform Characters

The **y** command is a special form of the **s** command. With the **y** command, you can replace a number of characters easily, without having to write a series of **s** commands.

The form of the command is:

```
y/123/abc/
```

In the above example, **1** will be replaced with **a**, **2** with **b**, and **3** with **c** throughout the document (no **g** option is needed). For **y** to work properly there must be a one-to-one relationship between the characters being replaced and the characters replacing them. Also, **y** cannot make exchanges that involve more than one character — it cannot, for example, replace **apple** with **banana**.

One useful task for the **y** command is to change all upper-case letters in a file to lower case. Prepare the following text file to see how this is done:

```
cat >exercise18
NOW IS THE TIME FOR ALL GOOD MEN TO COME
TO THE AID OF THE PARTY.
<ctrl-D>
```

And prepare the following script, which will change these capitals:

```
cat >script18
y/ABCDEFGHI/abcdefghi/
y/JKLMNOPQR/jlkmnopqr/
y/STUVWXYZ/stuvwxyz/
<ctrl-D>
```

The alphabet is entered here in three chunks, to prevent the command from being too long to type easily. Execute this script with the following command line:

```
sed -f script18 exercise18
```

The result is:

```
now is the time for all good men to come
to the aid of the party.
```

Command Control

sed gives you advanced control over the execution of commands. The next subsections describe how these command controls help you write compact, powerful scripts.

{ }: Command Grouping

In several of the exercises presented so far, more than one command specified the same line locator. By using braces '{' and '}', you can bundle commands, which makes writing your scripts easier and lessens the chance of making a typographical error.

To see how this is done, type the following exercise:

```
cat >exercise19
When my love swears that she is made of truth,
I do believe her, though I know she lies,
That she might think me some untutored youth,
Unlearned in the world's false subtleties.
<ctrl-D>
```

Now, prepare the following script:

```
cat >script19
/truth/{N
P
}
/lies/d
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script19 exercise19
```

the result on your terminal is:

```
When my love swears that she is made of truth,
That she might think me some untutored youth,
Unlearned in the world's false subtleties.
```

Note the syntax of this command. Each subsequent command must go on a line of its own, as must the right brace '}'. If this syntax is not observed, you will receive an error message.

!: All But

The **!** flag inverts a line selector; that is to say, the command will be performed on every line *but* the one named in the line selector. The following script will show how this works:

```
cat >script19a
2!d
<ctrl-D>
```

which, when run with the following command line:

```
sed -f script19a exercise19
```

produces

```
I do believe her, though I know she lies,
```

This script deleted every line *except* line 2. The **!** flag may also be used with a range of lines, as indicated by line numbers or line patterns; in either case, you must place the **!** flag *after* the line selectors and immediately *before* the command. Obviously, the **!** flag is very powerful, and can be used to sift out a few desired lines from a large file.

= : Print Line Number

You may wish to print only the *line number* of lines that contain a selected pattern. This is done with the **=** command. For example, you may wish to know the number of each line in the exercise that contains the word **she**. The following script:

```
cat >script19b
/she/=
<ctrl-D>
```

when executed with the following command line (note the **-n** option):

```
sed -n -f script19b exercise19
```

produces this result:

```
1
2
3
```

These numbers can be stored in a file and used in further editing, or included with the text of the fully edited file to provide a series of line markers.

Skipping Commands

sed normally processes editing commands in order, beginning with the first command and proceeding sequentially to the last. This behavior can be modified by the branching commands: **b**, **t**, and **:**.

These commands must be used with the colon (**:**) command, which defines a *label* point in the list of commands.

The **branch** command **b** allows you to skip unconditionally some editing commands in your script. The following exercise demonstrates how this can be used:

```
cat >exercise20
They went to sea in a sieve, they did;
In a sieve they went to sea;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a sieve they went to sea.
<ctrl-D>
```

The following script uses the **b** command to avoid making certain changes to the first line of the poem:

```
cat >script20
s/sea/drink/g
/They/bend
s/sieve/ship/g
:end
```

When you execute this script with the following command line:

```
sed -f script20 exercise20
```

you will see:

```
They went to drink in a sieve, they did;
In a ship they went to drink;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a ship they went to drink.
```

Note that the word **sea** is changed to **drink** throughout the file; however, when **sed** noted that the word **They** appeared in line 1, the **b** command forced it to seek the **:** command that was labeled with the word **end**, and to continue editing only *after* it found the labelled **:** command. In so doing, **sed** skipped the command to substitute **ship** for **sieve**, which is why that substitution was not made in line 1.

Note the syntax of the **b** command: the label follows it without a break. The text of the label is unimportant, just so long as it matches that used in the **b** command; however, the use of a label allows you to place several **b** or (as will be seen) **t** commands in the same script without mixing them up.

t: Test Command

The **test** command, **t**, also allows you to change the order in which editing commands are executed. Unlike the **b** command, which simply examines a line for a given pattern, the **t** command *tests* to see if a particular substitution has been performed.

The following script demonstrates the use of the **t** command:

```
        cat >script20a
        s/They/they/g
        tend
        s/sieve/ship/
        :end
        s/sea/drink/g
        <ctrl-D>
```

which, when executed with the following command line:

```
        sed -f script20a exercise20
```

produces:

```
        they went to drink in a sieve, they did;
        In a ship they went to drink;
        In spite of all their friends could say,
        On a winter’s morn, on a stormy day,
        In a ship they went to drink.
```

Note that the **t** command checked to see that **they** was substituted for **They** before branching to the ‘:’ command labeled with the word **end**.

Also note the syntax of the **t** command: Like the **b** command, the label immediately follows the command and is not separated by a space; unlike the **b** command, however, the **t** command appears on the line *below* the substitution command for which it is testing.

For More Information

The Lexicon entry for **sed** summarizes its command-line options and commands. The COHERENT line editor **ed** resembles **sed**, except that it works interactively instead of in a stream. For information on **ed**, see its tutorial or its entry in the Lexicon.