sed

The name sed is short for *stream editor*. It performs text editing on a stream of text, either a set of specified files or standard input. sed is a powerful and somewhat complex program (there are entire books about it), so we will not cover it completely here.

In general, the way sed works is that it is given either a single editing command (on the command line) or the name of a script file containing multiple commands, and it then performs these commands upon each line in the stream of text. Here is a simple example of sed in action:

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

In this example, we produce a one-word stream of text using echo and pipe it into sed. sed, in turn, carries out the instruction s/front/back/ upon the text in the stream and produces the output “back” as a result. We can also recognize this command as resembling the “substitution” (search-and-replace) command in vi.

Commands in sed begin with a single letter. In the previous example, the substitution command is represented by the letter s and is followed by the search-and-replace strings, separated by the slash character as a delimiter. The choice of the delimiter character is arbitrary. By convention, the slash character is often used, but sed will accept any character that immediately follows the command as the delimiter. We could perform the same command this way:

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_'
back
```

By using the underscore character immediately after the command, it becomes the delimiter. The ability to set the delimiter can be used to make commands more readable, as we shall see.

Most commands in sed may be preceded by an *address*, which specifies which line(s) of the input stream will be edited. If the address is omitted, then the editing command is carried out on every line in the input stream. The simplest form of address is a line number. We can add one to our example.

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

Adding the address `1` to our command causes our substitution to be performed on the first line of our one-line input stream. If we specify another number and we see that the editing is not carried out, since our input stream does not have a line 2.

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

Addresses may be expressed in many ways. Table 20-7 lists the most common.

Table 20-7: `sed` Address Notation

Address	Description
n	A line number where *n* is a positive integer.
$	The last line.
/regexp/	Lines matching a POSIX basic regular expression. Note that the regular expression is delimited by slash characters. Optionally, the regular expression may be delimited by an alternate character, by specifying the expression with `\c`*regexp*`c`, where *c* is the alternate character.
addr1,*addr2*	A range of lines from *addr1* to *addr2*, inclusive. Addresses may be any of the single address forms listed earlier.
first~*step*	Match the line represented by the number *first*, then each subsequent line at *step* intervals. For example 1~2 refers to each odd numbered line, and 5~5 refers to the fifth line and every fifth line thereafter.
addr1,+n	Match *addr1* and the following *n* lines.
addr!	Match all lines except *addr*, which may be any of the forms listed earlier.

We'll demonstrate different kinds of addresses using the `distros.txt` file from earlier in this chapter. First, here's a range of line numbers:

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE    10.2     12/07/2006
Fedora  10       11/25/2008
SUSE    11.0     06/19/2008
```

```
Ubuntu  8.04     04/24/2008
Fedora  8    11/08/2007
```

In this example, we print a range of lines, starting with line 1 and continuing to line 5. To do this, we use the `p` command, which simply causes a matched line to be printed. For this to be effective, however, we must include the option `-n` (the "no auto-print" option) to cause `sed` not to print every line by default.

Next, we'll try a regular expression.

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE    10.2     12/07/2006
SUSE    11.0     06/19/2008
SUSE    10.3     10/04/2007
SUSE    10.1     05/11/2006
```

By including the slash-delimited regular expression `/SUSE/`, we are able to isolate the lines containing it in much the same manner as `grep`.

Finally, we'll try negation by adding an exclamation point (`!`) to the address.

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora  10       11/25/2008
Ubuntu  8.04     04/24/2008
Fedora  8        11/08/2007
Ubuntu  6.10     10/26/2006
Fedora  7        05/31/2007
Ubuntu  7.10     10/18/2007
Ubuntu  7.04     04/19/2007
Fedora  6        10/24/2006
Fedora  9        05/13/2008
Ubuntu  6.06     06/01/2006
Ubuntu  8.10     10/30/2008
Fedora  5        03/20/2006
```

Here we see the expected result: all the lines in the file except the ones matched by the regular expression.

So far, we've looked at two of the `sed` editing commands, `s` and `p`. Table 20-8 provides a more complete list of the basic editing commands.

Table 20-8: `sed` Basic Editing Commands

Command	Description
`=`	Output the current line number.
`a`	Append text after the current line.
`d`	Delete the current line.
`i`	Insert text in front of the current line.
`p`	Print the current line. By default, `sed` prints every line and only edits lines that match a specified address within the file. The default behavior can be overridden by specifying the `-n` option.
`q`	Exit `sed` without processing any more lines. If the `-n` option is not specified, output the current line.
`Q`	Exit `sed` without processing any more lines.
`s/regexp/replacement/`	Substitute the contents of *replacement* wherever *regexp* is found. *replacement* may include the special character `&`, which is equivalent to the text matched by *regexp*. In addition, *replacement* may include the sequences `\1` through `\9`, which are the contents of the corresponding subexpressions in *regexp*. For more about this, see the discussion of *back references* below. After the trailing slash following *replacement*, an optional flag may be specified to modify the `s` command's behavior.
`y/set1/set2`	Perform transliteration by converting characters from *set1* to the corresponding characters in *set2*. Note that unlike `tr`, `sed` requires that both sets be of the same length.

The `s` command is by far the most commonly used editing command. We will demonstrate just some of its power by performing an edit on our `distros.txt` file. We discussed earlier how the date field in `distros.txt` was not in a "computer-friendly" format. While the date is formatted MM/DD/YYYY, it would be better (for ease of sorting) if the format were YYYY-MM-DD. Performing this change on the file by hand would be both time consuming and error prone, but with `sed`, this change can be performed in one step.

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\
)$/\3-\1-\2/' distros.txt
SUSE    10.2     2006-12-07
Fedora  10       2008-11-25
SUSE    11.0     2008-06-19
Ubuntu  8.04     2008-04-24
Fedora  8        2007-11-08
SUSE    10.3     2007-10-04
Ubuntu  6.10     2006-10-26
Fedora  7        2007-05-31
Ubuntu  7.10     2007-10-18
Ubuntu  7.04     2007-04-19
SUSE    10.1     2006-05-11
Fedora  6        2006-10-24
Fedora  9        2008-05-13
Ubuntu  6.06     2006-06-01
Ubuntu  8.10     2008-10-30
Fedora  5        2006-03-20
```

Wow! Now that is an ugly looking command. But it works. In just one step, we have changed the date format in our file. It is also a perfect example of why regular expressions are sometimes jokingly referred to as a “write-only” medium. We can write them, but we sometimes cannot read them. Before we are tempted to run away in terror from this command, let’s look at how it was constructed. First, we know that the command will have this basic structure.

```
sed 's/regexp/replacement/' distros.txt
```

Our next step is to figure out a regular expression that will isolate the date. Because it is in MM/DD/YYYY format and appears at the end of the line, we can use an expression like this:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

This matches two digits, a slash, two digits, a slash, four digits, and the end of line. So that takes care of *regexp*, but what about *replacement*? To handle that, we must introduce a new regular expression feature that appears in some applications that use BRE. This feature is called *back references* and works like this: if the sequence *n* appears in *replacement* where *n* is a number from 1 to 9, the sequence will refer to the corresponding subexpression in the preceding regular expression. To create the subexpressions, we sim-

ply enclose them in parentheses like so:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

We now have three subexpressions. The first contains the month, the second contains the day of the month, and the third contains the year. Now we can construct *replacement* as follows:

```
\3-\1-\2
```

This gives us the year, a dash, the month, a dash, and the day.

Now, our command looks like this:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

We have two remaining problems. The first is that the extra slashes in our regular expression will confuse `sed` when it tries to interpret the `s` command. The second is that since `sed`, by default, accepts only basic regular expressions, several of the characters in our regular expression will be taken as literals, rather than as metacharacters. We can solve both these problems with a liberal application of backslashes to escape the offending characters.

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' dis
tros.txt
```

And there you have it!

Another feature of the `s` command is the use of optional flags that may follow the replacement string. The most important of these is the `g` flag, which instructs `sed` to apply the search-and-replace globally to a line, not just to the first instance, which is the default. Here is an example:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

We see that the replacement was performed, but only to the first instance of the letter `b`, while the remaining instances were left unchanged. By adding the `g` flag, we are able to

change all the instances.

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

So far, we have only given `sed` single commands via the command line. It is also possible to construct more complex commands in a script file using the `-f` option. To demonstrate, we will use `sed` with our `distros.txt` file to build a report. Our report will feature a title at the top, our modified dates, and all the distribution names converted to uppercase. To do this, we will need to write a script, so we'll fire up our text editor and enter the following:

```
# sed script to produce Linux distributions report

1 i\
\
Linux Distributions Report\

s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

We will save our `sed` script as `distros.sed` and run it like this:

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt

Linux Distributions Report

SUSE    10.2     2006-12-07
FEDORA  10       2008-11-25
SUSE    11.0     2008-06-19
UBUNTU  8.04     2008-04-24
FEDORA  8        2007-11-08
SUSE    10.3     2007-10-04
UBUNTU  6.10     2006-10-26
FEDORA  7        2007-05-31
UBUNTU  7.10     2007-10-18
UBUNTU  7.04     2007-04-19
SUSE    10.1     2006-05-11
FEDORA  6        2006-10-24
FEDORA  9        2008-05-13
```

```
UBUNTU   6.06      2006-06-01
UBUNTU   8.10      2008-10-30
FEDORA   5         2006-03-20
```

As we can see, our script produces the desired results, but how does it do it? Let's take another look at our script. We'll use `cat` to number the lines.

```
[me@linuxbox ~]$ cat -n distros.sed
    1   # sed script to produce Linux distributions report
    2
    3   1 i\
    4   \
    5   Linux Distributions Report\
    6
    7   s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
    8   y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Line one of our script is a *comment*. Like many configuration files and programming languages on Linux systems, comments begin with the # character and are followed by human-readable text. Comments can be placed anywhere in the script (though not within commands themselves) and are helpful to any humans who might need to identify and/or maintain the script.

Line 2 is a blank line. Like comments, blank lines may be added to improve readability.

Many `sed` commands support line addresses. These are used to specify which lines of the input are to be acted upon. Line addresses may be expressed as single line numbers, line number ranges, and the special line number `$`, which indicates the last line of input.

Lines 3, 4, 5, and 6 contain text to be inserted at the address 1, the first line of the input. The `i` command is followed by the sequence of a backslash and then a carriage return to produce an escaped carriage return, or what is called a *line-continuation character*. This sequence, which can be used in many circumstances including shell scripts, allows a carriage return to be embedded in a stream of text without signaling the interpreter (in this case `sed`) that the end of the line has been reached. The `i`, and the `a` (which appends text, rather than inserting it) and `c` (which replaces text) commands allow multiple lines of text as long as each line, except the last, ends with a line-continuation character. The sixth line of our script is actually the end of our inserted text and ends with a plain carriage return rather than a line-continuation character, signaling the end of the `i` command.

Note: A line-continuation character is formed by a backslash followed *immediately* by a carriage return. No intermediary spaces are permitted.

Line 7 is our search-and-replace command. Since it is not preceded by an address, each line in the input stream is subject to its action.

Line 8 performs transliteration of the lowercase letters into uppercase letters. Note that unlike `tr`, the `y` command in `sed` does not support character ranges (for example, `[a-z]`), nor does it support POSIX character classes. Again, since the `y` command is not preceded by an address, it applies to every line in the input stream.

People Who Like sed Also Like...

`sed` is a capable program, able to perform fairly complex editing tasks to streams of text. It is most often used for simple, one-line tasks rather than long scripts. Many users prefer other tools for larger tasks. The most popular of these are `awk` and `perl`. These go beyond mere tools like the programs covered here and extend into the realm of complete programming languages. `perl`, in particular, is often used instead of shell scripts for many system management and administration tasks, as well as being a popular medium for web development. `awk` is a little more specialized. Its specific strength is its ability to manipulate tabular data. It resembles `sed` in that `awk` programs normally process text files line by line, using a scheme similar to the `sed` concept of an address followed by an action. While both `awk` and `perl` are outside the scope of this book, they are good skills for the Linux command line user to learn.

`aspell`

The last tool we will look at is `aspell`, an interactive spelling checker. The `aspell` program is the successor to an earlier program named `ispell` and can be used, for the most part, as a drop-in replacement. While the `aspell` program is mostly used by other programs that require spell-checking capability, it can also be used effectively as a standalone tool from the command line. It has the ability to intelligently check various types of text files, including HTML documents, C/C++ programs, email messages, and other kinds of specialized texts.

To spellcheck a text file containing simple prose, it could be used like this: