

19

REGULAR EXPRESSIONS



In the next few chapters, we are going to look at tools used to manipulate text. As we have seen, text data plays an important role on all Unix-like systems, such as Linux. But before we can fully appreciate all the features offered by these tools, first we have to examine a technology that is frequently associated with the most sophisticated uses of these tools—*regular expressions*.

As we have navigated the many features and facilities offered by the command line, we have encountered some truly arcane features, such as shell expansion and quoting, keyboard shortcuts, and command history, not to mention the `vi` editor. Regular expressions continue this “tradition” and may be (arguably) the most arcane feature of them all. This is not to suggest that the time it takes to learn about them is not worth the effort. Quite the contrary. A good understanding will enable us to perform amazing feats, though their full value may not be immediately apparent.

What Are Regular Expressions?

Simply put, regular expressions are symbolic notations used to identify patterns in text. In some ways, they resemble the shell’s wildcard method of matching file and pathnames but on a much grander scale. Regular expressions are supported by many command line tools and by most programming languages to facilitate the solution of text manipulation prob-

lems. However, to further confuse things, not all regular expressions are the same; they vary slightly from tool to tool and from programming language to language. For our discussion, we will limit ourselves to regular expressions as described in the POSIX standard (which will cover most of the command line tools), as opposed to many programming languages (most notably Perl), which use slightly larger and richer sets of notations.

grep

The main program we will use to work with regular expressions is our old pal `grep`. The name *grep* is actually derived from the phrase “global regular expression print,” so we can see that `grep` has something to do with regular expressions. In essence, `grep` searches text files for text matching a specified regular expression and outputs any line containing a match to standard output.

So far, we have used `grep` with fixed strings, like so:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

This will list all the files in the */usr/bin* directory whose names contain the substring *zip*.

The `grep` program accepts options and arguments this way, where *regex* is a regular expression:

```
grep [options] regex [file...]
```

Table 19-1 describes the commonly used `grep` options.

Table 19-1: `grep` Options

Option	Long op- tion	Description
--------	---------------------	-------------

Option	Long option	Description
-i	--ignore-case	Ignore case. Do not distinguish between uppercase and lowercase characters.
-v	--invert-match	Invert match. Normally, <code>grep</code> prints lines that contain a match. This option causes <code>grep</code> to print every line that does not contain a match.
-c	--count	Print the number of matches (or non-matches if the -v option is also specified) instead of the lines themselves.
-l	--files-with-matches	Print the name of each file that contains a match instead of the lines themselves.
-L	--files-without-match	Like the -l option, but print only the names of files that do not contain matches.
-n	--line-number	Prefix each matching line with the number of the line within the file.
-h	--no-file-name	For multifile searches, suppress the output of filenames.

To more fully explore `grep`, let's create some text files to search.

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt  dirlist-sbin.txt  dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

We can perform a simple search of our list of files like this:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

In this example, `grep` searches all the listed files for the string *bzip* and finds two matches, both in the file *dirlist-bin.txt*. If we were interested only in the list of files that contained matches rather than the matches themselves, we could specify the `-l` option.

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

Conversely, if we wanted to see only a list of the files that did not contain a match, we could do this:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Metacharacters and Literals

While it might not seem apparent, our `grep` searches have been using regular expressions all along, albeit very simple ones. The regular expression `bzip` is taken to mean that a match will occur only if the line in the file contains at least four characters and that somewhere in the line the charac-

ters *b*, *z*, *i*, and *p* are found in that order, with no other characters in between. The characters in the string *bzip* are all *literal characters*, in that they match themselves. In addition to literals, regular expressions may also include *metacharacters* that are used to specify more complex matches. Regular expression metacharacters consist of the following:

`^ $. [] { } - ? * + () | \`

All other characters are considered literals, though the backslash character is used in a few cases to create *metasequences*, as well as allowing the metacharacters to be escaped and treated as literals instead of being interpreted as metacharacters.

NOTE

As we can see, many of the regular expression metacharacters are also characters that have meaning to the shell when expansion is performed. When we pass regular expressions containing metacharacters on the command line, it is vital that they be enclosed in quotes to prevent the shell from attempting to expand them.

The Any Character

The first metacharacter we will look at is the dot (.) or period character, which is used to match any character. If we include it in a regular expression, it will match any character in that character position. Here's an example:

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
```

```
bunzip2
```

```
bzip2
```

```
bzip2recover
```

```
gunzip
```

```
gzip
```

```
funzip
```

gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx

We searched for any line in our files that matches the regular expression `.zip`. There are a couple of interesting things to note about the results. Notice that the `zip` program was not found. This is because the inclusion of the dot metacharacter in our regular expression increased the length of the required match to four characters, and because the name `zip` contains only three, it does not match. Also, if any files in our lists had contained the file extension `.zip`, they would have been matched as well, because the period character in the file extension would be matched by the “any character,” too.

Anchors

The caret (^) and dollar sign (\$) are treated as *anchors* in regular expressions. This means they cause the match to occur only if the regular expression is found at the beginning of the line (^) or at the end of the line (\$).

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
```

zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit

```
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
```

gunzip
gzip
funzip
gpg-zip

```
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

Here we searched the list of files for the string *zip* located at the beginning of the line, at the end of the line, and on a line where it is at both the beginning and the end of the line (i.e., by itself on the line). Note that the regular expression `^$` (a beginning and an end with nothing in between) will match blank lines.

A CROSSWORD PUZZLE HELPER

Even with our limited knowledge of regular expressions at this point, we can do something useful.

My wife loves crossword puzzles, and she will sometimes ask me for help with a particular question. Something like, “What’s a five-letter word whose third letter is *j* and last letter is *r* that means . . . ?” This kind of question got me thinking.

Did you know that your Linux system contains a dictionary? It does. Take a look in the `/usr/share/dict` directory, and you might find one or several. The dictionary files located there are just long lists of words, one per line, arranged in alphabetical order. On my system, the *words* file contains just over 98,500 words. To find possible answers to the crossword puzzle question above, we could do this:

```
[me@linuxbox ~]$ grep -i '^...j.r$' /usr/share/dict/words
Major
major
```

Using this regular expression, we can find all the words in our dictionary file that are five letters long and have a *j* in the third position and an

r in the last position.

Bracket Expressions and Character Classes

In addition to matching any character at a given position in our regular expression, we can also match a single character from a specified set of characters by using *bracket expressions*. With bracket expressions, we can specify a set of characters (including characters that would otherwise be interpreted as metacharacters) to be matched. In this example, using a two-character set, we match any line that contains the string *bzip* or *gzip*:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

A set may contain any number of characters, and metacharacters lose their special meaning when placed within brackets. However, there are two cases in which metacharacters are used within bracket expressions and have different meanings. The first is the caret (^), which is used to indicate negation; the second is the dash (-), which is used to indicate a character range.

Negation

If the first character in a bracket expression is a caret (^), the remaining characters are taken to be a set of characters that must not be present at the given character position. We do this by modifying our previous example, as follows:

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
```

preunzip
prezip
prezip-bin
unzip
unzipsfx

With negation activated, we get a list of files that contain the string *zip* preceded by any character except *b* or *g*. Notice that the file *zip* was not found. A negated character set still requires a character at the given position, but the character must not be a member of the negated set.

The caret character invokes negation only if it is the first character within a bracket expression; otherwise, it loses its special meaning and becomes an ordinary character in the set.

Traditional Character Ranges

If we wanted to construct a regular expression that would find every file in our lists beginning with an uppercase letter, we could do this:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHJKLMNOPQRSTUVWXYZ]' dirlist*.txt
```

It's just a matter of putting all 26 uppercase letters in a bracket expression. But the idea of all that typing is deeply troubling, so here is another way.

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
```

```
MAKEDEV  
ControlPanel  
GET  
HEAD  
POST  
X  
X11  
Xorg  
MAKEFLOPPIES
```

By using a three-character range, we can abbreviate the 26 letters. Any range of characters can be expressed this way including multiple ranges, such as this expression that matches all filenames starting with letters and numbers:

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

In character ranges, we see that the dash character is treated specially, so how do we actually include a dash character in a bracket expression? By making it the first character in the expression. Consider these two examples:

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

This will match every filename containing an uppercase letter. The following will match every filename containing a dash or an uppercase A or an uppercase Z:

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

POSIX Character Classes

The traditional character ranges are an easily understood and effective way to handle the problem of quickly specifying sets of characters. Unfortunately, they don't always work. While we have not encountered any problems with our use of `grep` so far, we might run into problems using other programs.

In [Chapter 4](#), we looked at how wildcards are used to perform path-name expansion. In that discussion, we said that character ranges could be used in a manner almost identical to the way they are used in regular expressions, but here's the problem:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*  
/usr/sbin/ModemManager  
/usr/sbin/NetworkManager
```

(Depending on the Linux distribution, we will get a different list of files, possibly an empty list. This example is from Ubuntu.) This command produces the expected result—a list of only the files whose names begin with an uppercase letter, but with the following command we get an entirely different result (only a partial listing of the results is shown):

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*  
/usr/sbin/biosdecode  
/usr/sbin/chat  
/usr/sbin/chgpasswd  
/usr/sbin/chpasswd  
/usr/sbin/chroot  
/usr/sbin/cleanup-info  
/usr/sbin/complain  
/usr/sbin/console-kit-daemon
```

Why is that? It's a long story, but here's the short version:

Back when Unix was first developed, it knew only about ASCII characters, and this feature reflects that fact. In ASCII, the first 32 characters (numbers 0–31) are control codes (things such as tabs, backspaces, and carriage returns). The next 32 (32–63) contain printable characters, including most punctuation characters and the numerals 0–9. The next 32 (numbers 64–95) contain the uppercase letters and a few more punctuation symbols. The final 31 (numbers 96–127) contain the lowercase letters and yet more punctuation symbols. Based on this arrangement, systems using ASCII used a *collation order* that looks like this:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

This differs from proper dictionary order, which is like this:

As the popularity of Unix spread beyond the United States, there grew a need to support characters not found in US English. The ASCII table was expanded to use a full eight bits, adding characters 128–255, which accommodated many more languages. To support this capability, the POSIX standards introduced a concept called a *locale*, which could be adjusted to select the character set needed for a particular location. We can see the language setting of our system using the following command.

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

With this setting, POSIX-compliant applications will use a dictionary collation order rather than ASCII order. This explains the behavior of the previous commands. A character range of `[A-Z]` when interpreted in dictionary order includes all of the alphabetic characters except the lowercase *a*, hence our results.

To partially work around this problem, the POSIX standard includes a number of character classes that provide useful ranges of characters, as described in [Table 19-2](#).

Table 19-2: POSIX Character Classes

Character class	Description
<code>[:alnum:]</code>	The alphanumeric characters. In ASCII, equivalent to: <code>[A-Za-z0-9]</code>
<code>[:word:]</code>	The same as <code>[:alnum:]</code> , with the addition of the underscore (<code>_</code>) character.

Character class	Description
<code>[:alpha:]</code>	The alphabetic characters. In ASCII, equivalent to: <code>[A-Za-z]</code>
<code>[:blank:]</code>	Includes the space and tab characters.
<code>[:cntrl:]</code>	The ASCII control codes. Includes the ASCII characters 0 through 31 and 127.
<code>[:digit:]</code>	The numerals 0 through 9.
<code>[:graph:]</code>	The visible characters. In ASCII, it includes characters 33 through 126.
<code>[:lower:]</code>	The lowercase letters.
<code>[:punct:]</code>	The punctuation characters. In ASCII, equivalent to: <code>[- ! " # \$ % & ' () * + , . / : ; < = > ? @ [\] _ ` { } ~]</code>
<code>[:print:]</code>	The printable characters. All the characters in <code>[:graph:]</code> plus the space character.
<code>[:space:]</code>	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed. In ASCII, equivalent to: <code>[\t\r\n\v\f]</code>
<code>[:upper:]</code>	The uppercase characters.
<code>[:xdigit:]</code>	Characters used to express hexadecimal numbers. In ASCII, equivalent to: <code>[0-9A-Fa-f]</code>

Even with the character classes, there is still no convenient way to express partial ranges, such as `[A-M]`.

Using character classes, we can repeat our directory listing and see an improved result.

```
[me@linuxbox ~]$ ls /usr/sbin/[:upper:]*  
/usr/sbin/MAKEFLOPPIES  
/usr/sbin/NetworkManagerDispatcher  
/usr/sbin/NetworkManager
```

Remember, however, that this is not an example of a regular expression; rather, it is the shell performing pathname expansion. We show it here because POSIX character classes can be used for both.

REVERTING TO TRADITIONAL COLLATION ORDER

You can opt to have your system use the traditional (ASCII) collation order by changing the value of the `LANG` environment variable. As we saw earlier, the `LANG` variable contains the name of the language and character set used in your locale. This value was originally determined when you selected an installation language as your Linux version was installed.

To see the locale settings, use the `locale` command.

```
[me@linuxbox ~]$ locale  
LANG=en_US.UTF-8  
LC_CTYPE="en_US.UTF-8"  
LC_NUMERIC="en_US.UTF-8"  
LC_TIME="en_US.UTF-8"  
LC_COLLATE="en_US.UTF-8"  
LC_MONETARY="en_US.UTF-8"  
LC_MESSAGES="en_US.UTF-8"  
LC_PAPER="en_US.UTF-8"  
LC_NAME="en_US.UTF-8"  
LC_ADDRESS="en_US.UTF-8"
```

```
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

To change the locale to use the traditional Unix behaviors, set the `LANG` variable to `POSIX`.

```
[me@linuxbox ~]$ export LANG=POSIX
```

Note that this change converts the system to use US English (more specifically, ASCII) for its character set, so be sure if this is really what you want.

You can make this change permanent by adding this line to your `.bashrc` file:

```
export LANG=POSIX
```

POSIX Basic vs. Extended Regular Expressions

Just when we thought this couldn't get any more confusing, we discover that POSIX also splits regular expression implementations into two kinds: *basic regular expressions (BRE)* and *extended regular expressions (ERE)*. The features we have covered so far are supported by any application that is POSIX compliant and implements BRE. Our `grep` program is one such program.

What's the difference between BRE and ERE? It's a matter of metacharacters. With BRE, the following metacharacters are recognized:

```
^ $ . [ ] *
```

All other characters are considered literals. With ERE, the following metacharacters (and their associated functions) are added:

However (and this is the fun part), the `(`, `)`, `{`, and `}` characters are treated as metacharacters in BRE *if* they are escaped with a backslash, whereas with ERE, preceding any metacharacter with a backslash causes it to be treated as a literal. Any weirdness that comes along will be covered in the discussions that follow.

Because the features we are going to discuss next are part of ERE, we are going to need to use a different `grep`. Traditionally, this has been performed by the `egrep` program, but the GNU version of `grep` also supports extended regular expressions when the `-E` option is used.

POSIX

During the 1980s, Unix became a very popular commercial operating system, but by 1988, the Unix world was in turmoil. Many computer manufacturers had licensed the Unix source code from its creators, AT&T, and were supplying various versions of the operating system with their systems. However, in their efforts to create product differentiation, each manufacturer added proprietary changes and extensions. This started to limit the compatibility of the software. As always with proprietary vendors, each was trying to play a winning game of “lock in” with their customers. This dark time in the history of Unix is known today as the *Balkanization*.

Enter the Institute of Electrical and Electronics Engineers (IEEE). In the mid-1980s, the IEEE began developing a set of standards that would define how Unix (and Unix-like) systems would perform. These standards, formally known as IEEE 1003, define the application programming interfaces (APIs), shell, and utilities that are to be found on a standard Unix-like system. The name POSIX, which stands for *Portable Operating System Interface* (with the *X* added to the end for extra snappiness), was suggested by Richard Stallman (yes, *that* Richard Stallman) and was adopted by the IEEE.

Alternation

The first of the extended regular expression features we will discuss is called *alternation*, which is the facility that allows a match to occur from among a set of expressions. Just as a bracket expression allows a single character to match from a set of specified characters, alternation allows matches from a set of strings or other regular expressions.

To demonstrate, we'll use `grep` in conjunction with `echo`. First, let's try a plain old string match.

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

This is a pretty straightforward example, in which we pipe the output of `echo` into `grep` and see the results. When a match occurs, we see it printed out; when no match occurs, we see no results.

Now we'll add alternation, signified by the vertical-bar metacharacter.

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

Here we see the regular expression `'AAA|BBB'`, which means “match either the string `AAA` or the string `BBB`.” Notice that since this is an extended feature, we added the `-E` option to `grep` (though we could have just used the `egrep` program instead), and we enclosed the regular expression in quotes to prevent the shell from interpreting the vertical-bar metacharacter as a pipe operator. Alternation is not limited to two choices.

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
```

```
AAA
```

To combine alternation with other regular expression elements, we can use `()` to separate the alternation.

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

This expression will match the filenames in our lists that start with either *bz*, *gz*, or *zip*. Had we left off the parentheses, the meaning of this regular expression changes to match any filename that begins with *bz* or contains *gz* or contains *zip*:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

Quantifiers

Extended regular expressions support several ways to specify the number of times an element is matched, as described in the sections that follow.

?—Match an Element Zero or One Time

This quantifier means, in effect, “make the preceding element optional.” Let’s say we wanted to check a phone number for validity and we considered a phone number to be valid if it matched either of these two forms, where *n* is a numeral:

- *(nnn) nnn-nnnn*
- *nnn nnn-nnnn*

We could construct a regular expression like this:

```
^\((?[0-9][0-9][0-9])\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

In this expression, we follow the parentheses characters with question marks to indicate that they are to be matched zero or one time. Again, be-

cause the parentheses are normally metacharacters (in ERE), we precede them with backslashes to cause them to be treated as literals instead.

Let's try it.

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

Here we see that the expression matches both forms of the phone number but does not match one containing non-numeric characters. This expression is not perfect as it still allows mismatched parentheses around the area code, but it will perform the first stage of a verification.

****—Match an Element Zero or More Times***

Like the `?` metacharacter, the `*` is used to denote an optional item; however, unlike the `?`, the item may occur any number of times, not just once. Let's say we wanted to see whether a string was a sentence; that is, it starts with an uppercase letter, then contains any number of uppercase and lowercase letters and spaces, and ends with a period. To match this (crude) definition of a sentence, we could use a regular expression like this:

```
^[[:upper:]][:upper:][:lower:]*\.
```

The expression consists of three items: a bracket expression containing the `[[:upper:]]` character class, a bracket expression containing both the `[[:upper:]]` and `[[:lower:]]` character classes and a space, and a period escaped with a backslash. The second element is trailed with an `*` metacharacter

so that after the leading uppercase letter in our sentence, any number of uppercase and lowercase letters and spaces may follow it and still match.

```
[me@linuxbox ~]$ echo "This works." | grep -E '^[:upper:][:upper:]
[:lower:] ]*\.'
```

This works.

```
[me@linuxbox ~]$ echo "This Works." | grep -E '^[:upper:][:upper:]
[:lower:] ]*\.'
```

This Works.

```
[me@linuxbox ~]$ echo "this does not" | grep -E '^[:upper:]
[:upper:][:lower:] ]*\.'
```

```
[me@linuxbox ~]$
```

The expression matches the first two tests, but not the third, since it lacks the required leading uppercase character and trailing period.

+—*Match an Element One or More Times*

The + metacharacter works much like the *, except it requires at least one instance of the preceding element to cause a match. Here is a regular expression that will match only the lines consisting of groups of one or more alphabetic characters separated by single spaces:

```
^([[:alpha:]]+ ?)+$
```

Let's try it.

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
```

This that

```
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
```

a b c

```
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
```

```
[me@linuxbox ~]$ echo "abc d" | grep -E '^([[:alpha:]]+ ?)+$'
```

```
[me@linuxbox ~]$
```

We see that this expression does not match the line `a b 9` because it contains a nonalphabetic character; nor does it match `abc d` because more than one space character separates the characters `c` and `d`.

***{ }*—Match an Element a Specific Number of Times**

The `{` and `}` metacharacters are used to express minimum and maximum numbers of required matches. They may be specified in four possible ways, as outlined in [Table 19-3](#).

Table 19-3: Specifying the Number of Matches

Specifier	Meaning
<code>{n}</code>	Match the preceding element if it occurs exactly <i>n</i> times.
<code>{n,m}</code>	Match the preceding element if it occurs at least <i>n</i> times but no more than <i>m</i> times.
<code>{n,}</code>	Match the preceding element if it occurs <i>n</i> or more times.
<code>{,m}</code>	Match the preceding element if it occurs no more than <i>m</i> times.

Going back to our earlier example with the phone numbers, we can use this method of specifying repetitions to simplify our original regular expression from the following:

```
^(\?[0-9][0-9][0-9])? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

to the following:

```
^(\?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

Let's try it.

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^([0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^([0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^([0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
[me@linuxbox ~]$
```

As we can see, our revised expression can successfully validate numbers both with and without the parentheses, while rejecting those numbers that are not properly formatted.

Putting Regular Expressions to Work

Let's look at some of the commands we already know and see how they can be used with regular expressions.

Validating a Phone List with grep

In our earlier example, we looked at single phone numbers and checked them for proper formatting. A more realistic scenario would be checking a list of numbers instead, so let's make a list. We'll do this by reciting a magical incantation to the command line. It will be magic because we have not covered most of the commands involved, but worry not. We will get there in future chapters. Here is the incantation.

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3})
${RANDOM:0:3}-${
{RANDOM:0:4}}" >> phonenumber.txt; done
```

This command will produce a file named *phonenumber.txt* containing 10 phone numbers. Each time the command is repeated, another 10 numbers are added to the list. We can also change the value 10 near the begin-

ning of the command to produce more or fewer phone numbers. If we examine the contents of the file, however, we see we have a problem.

```
[me@linuxbox ~]$ cat phonenumberlist.txt
```

```
(232) 298-2265
```

```
(624) 381-1078
```

```
(540) 126-1980
```

```
(874) 163-2885
```

```
(286) 254-2860
```

```
(292) 108-518
```

```
(129) 44-1379
```

```
(458) 273-1642
```

```
(686) 299-8268
```

```
(198) 307-2440
```

Some of the numbers are malformed, which is perfect for our purposes because we will use `grep` to validate them.

One useful method of validation would be to scan the file for invalid numbers and display the resulting list.

```
[me@linuxbox ~]$ grep -Ev '^([0-9]{3}) [0-9]{3}-[0-9]{4}$'
```

```
phonenumberlist.txt
```

```
(292) 108-518
```

```
(129) 44-1379
```

```
[me@linuxbox ~]$
```

Here we use the `-v` option to produce an inverse match so that we will output only the lines in the list that do not match the specified expression. The expression itself includes the anchor metacharacters at each end to ensure that the number has no extra characters at either end. This expression also requires that the parentheses be present in a valid number, unlike our earlier phone number example.

Finding Ugly Filenames with find

The `find` command supports a test based on a regular expression. There is an important consideration to keep in mind when using regular expressions in `find` versus `grep`. Whereas `grep` will print a line when the line *contains* a string that matches an expression, `find` requires that the pathname *exactly match* the regular expression. In the following example, we will use `find` with a regular expression to find every pathname that contains any character that is not a member of the following set:

```
[!_./0-9a-zA-Z]
```

Such a scan would reveal pathnames that contain embedded spaces and other potentially offensive characters.

```
[me@linuxbox ~]$ find . -regex '.*[!_./0-9a-zA-Z].*'
```

Because of the requirement for an exact match of the entire pathname, we use `.*` at both ends of the expression to match zero or more instances of any character. In the middle of the expression, we use a negated bracket expression containing our set of acceptable pathname characters.

Searching for Files with locate

The `locate` program supports both basic (the `--regex` option) and extended (the `--regexp` option) regular expressions. With it, we can perform many of the same operations that we performed earlier with our `dirlist` files.

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
```

```
/bin/bzcat
```

```
/bin/bzcmp
```

```
/bin/bzdiff
```

```
/bin/bzegrep
```

```
/bin/bzexe
```

```
/bin/bzfgrep
```

```
/bin/bzgrep
```

```
/bin/bzip2
```

```
/bin/bzip2recover
```

```
/bin/bzless
```



```
/bin/bzmore  
/bin/gzexe  
/bin/gzip  
/usr/bin/zip  
/usr/bin/zipcloak  
/usr/bin/zipgrep  
/usr/bin/zipinfo  
/usr/bin/zipnote  
/usr/bin/zipsplit
```

Using alternation, we perform a search for pathnames that contain either *bin/bz*, *bin/gz*, or */bin/zip*.

Searching for Text with less and vim

`less` and `vim` both share the same method of searching for text. Pressing the `/` key followed by a regular expression will perform a search. If we use `less` to view our *phonelist.txt* file, like so:

```
[me@linuxbox ~]$ less phonelist.txt
```

and then search for our validation expression, like this:

```
(232) 298-2265  
(624) 381-1078  
(540) 126-1980  
(874) 163-2885  
(286) 254-2860  
(292) 108-518  
(129) 44-1379  
(458) 273-1642  
(686) 299-8268  
(198) 307-2440  
~  
~
```

~

```
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

`less` will highlight the strings that match, leaving the invalid ones easy to spot.

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

~

~

~

```
(END)
```

`vim`, on the other hand, supports basic regular expressions, so our search expression would look like this:

```
/([0-9]{3}) [0-9]{3}-[0-9]{4}
```

We can see that the expression is mostly the same; however, many of the characters that are considered metacharacters in extended expressions are considered literals in basic expressions. They are treated only as metacharacters when escaped with a backslash. Depending on the particular configuration of `vim` on our system, the matching will be highlighted. If not, try this command mode command to activate search highlighting:

```
:hlsearch
```

NOTE

Depending on your distribution, *vim* may or may not support text search highlighting. Ubuntu, in particular, supplies a stripped-down version of *vim* by default. On such systems, you may want to use your package manager to install a more complete version of *vim*.

Summing Up

In this chapter, we saw a few of the many uses of regular expressions. We can find even more if we use regular expressions to search for additional applications that use them. We can do that by searching the man pages.

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

The `zgrep` program provides a front end for `grep`, allowing it to read compressed files. In our example, we search the compressed section 1 man page files in their usual location. The result of this command is a list of files containing either the string *regex* or the string *regular expression*. As we can see, regular expressions show up in a lot of programs.

There is one feature found in basic regular expressions that we did not cover. Called *back references*, this feature will be discussed in the next chapter.