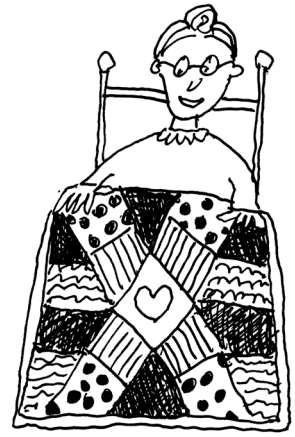
chapter

# 8

# Regular Expressions—Pattern Matching



## 8.1 What Is a Regular Expression?

If you are familiar with UNIX utilities, such as *vi, sed, grep*, and *awk*, you have met face-to-face with the infamous regular expressions and metacharacters used in delimiting search patterns. Well, with Perl, they're back!

What is a regular expression, anyway? A **regular expression** is really just a sequence, or pattern, of characters that is matched against a string of text when performing searches and replacements. A simple regular expression consists of a character or set of characters that matches itself. The regular expression is normally delimited by forward slashes.[1] The special scalar $_ is the default search space where Perl does its pattern matching. $_ is like a shadow. Sometimes you see it; sometimes you don't. Don't worry; all this will become clear as you read through this chapter.

**EXAMPLE 8.1**

```
1   /abc/
2   ?abc?
```

**EXPLANATION**

1 The pattern *abc* is enclosed in forward slashes. If searching for this pattern, for example, in a string or text file, any string that contained the pattern *abc* would be matched.

2 The pattern *abc* is enclosed in question marks. If searching for this pattern, only the first occurrence of the string is matched. (See the *reset* function in Appendix A.)

1. Actually, any character can be used as a delimiter. See Table 8.1 on page 210 and Example 8.12 on page 211.

## 8.2 Expression Modifiers and Simple Statements

A **simple statement** is an expression terminated with a semicolon. Perl supports a set of modifiers that allow you to further evaluate an expression based on some condition. A simple statement may contain an expression **ending** with a single modifier. The modifier and its expression are always terminated with a semicolon. When evaluating regular expressions, the modifiers may be simpler to use than the full-blown conditional constructs (discussed in Chapter 7, "If Only, Unconditionally, Forever").

The modifiers are

*if*
*unless*
*while*
*until*
*foreach*

### 8.2.1 Conditional Modifiers

**The *if* Modifier.** The *if* modifier is used to control a simple statement consisting of two expressions. If *Expression1* is true, *Expression2* is executed.

**FORMAT**

```
Expression2 if Expression1;
```

**EXAMPLE 8.2**

```
(In Script)
1   $x = 5;
2   print $x  if $x == 5;

(Output)
5
```

**EXPLANATION**

1 $x is assigned 5. The value of $x is printed only if $x is equal to 5.

2 The *if* modifier must be placed at the end of a statement and, in this example, controls the *print* function. If the expression $x == 5 is true, then the value of $x is printed.
It could be written *if ($x == 5) {print $x;}*.

## EXAMPLE 8.3

```
(In Script)
1   $_ = "xabcy\n";
2   print if /abc/;    # Could be written: print $_ if $_ =~ /abc/;

(Output)
xabcy
```

## EXPLANATION

1 The $_ scalar variable is assigned the string *xabcy*.

2 When the *if* modifier is followed directly by a regular expression, Perl assumes that the line being matched is $_, the default placeholder for pattern matching. The value of $_, *xabcy*, is printed if the regular expression *abc* is matched anywhere in the string.[a] The expression could have been written as *if $_ =~ /abc/*. (The =~ match operator will be discussed at the end of this chapter.)

a. $_ is the default output for the *print* function.

## EXAMPLE 8.4

```
(In Script)
1   $_ = "I lost my gloves in the clover.";
2   print "Found love in gloves!\n" if /love/;
                      # Long form: if $_ =~ /love

(Output)
Found love in gloves!
```

## EXPLANATION

1 The $_ is assigned the string *I lost my gloves in the clover.*

2 The regular expression *love* is matched in the $_ variable, and the string *Found love in gloves!* is printed; otherwise, nothing will be printed. The regular expression *love* is found in both *gloves* and *clover*. The search starts at the left-hand side of the string, so that matching *love* in *gloves* will produce the true condition before *clover* is reached. If $_ (or, for that matter, any other scalar) is used explicitly after the *if* modifier, then the =~ pattern matching operator is necessary when evaluating the regular expression.

### 8.2.2 The *DATA* Filehandle

In the following examples, the special filehandle called *DATA* is used as an expression in a *while* loop. This allows us to directly get the data from the same script that is testing it, rather than reading input from a separate text file. (You will learn all about filehandles

in Chapter 10, "Getting a Handle on Files.") The data itself is located after the _ _*DATA*_ _[2] special literal at the bottom of each of the example scripts. The _ _*DATA*_ _ literal marks the logical end of the script and opens the *DATA* filehandle for reading. Each time a line of input is read from <*DATA*>, it is assigned by default to the special $_ scalar. Although $_ is implied, you could also use it explicitly, or even some other scalar. The format used is shown in the following examples.

## FORMAT

```
while(<DATA>){
    Do something with the data here
}
_ _DATA_ _
    The actual data is stored here
```

Or you could use the $_ explicitly as follows:

```
while($_=<DATA>){
    Do something with the data here
}
_ _DATA_ _
    The actual data is stored here
```

Or use another variable instead of $_ as follows:

```
while($inputline=<DATA>){
    Do something with the data here
}
_ _DATA_ _
    The actual data is stored here
```

## EXAMPLE 8.5

```
(The Script)
1   while(<DATA>){
2       print if /Norma/;      # Print the line if it matches Norma
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich
```

2. Instead of _ _*DATA*_ _, you can use _ _*END*_ _, but _ _*END*_ _ opens the *DATA* filehandle in the *main* package and _ _*DATA*_ _ in any package.

## EXAMPLE 8.5 (CONTINUED)

```
(Output)
Norma Cord
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. When the *while* loop is entered, a line of input is stored in the $_ scalar variable. The first line stored in $_ is *Steve Blenheim*. The next time around the loop, *Betty Boop* is stored in $_, and this continues until all of the lines following the _ _*DATA*_ _ token are read and processed.

2 Only the lines containing the regular expression *Norma* are printed. $_ is the default for pattern matching; it could also have been written as *print $_ if $_ =~ /Norma/;*.

3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.6

```
(The Script)
1   while(<DATA>){
2       if /Norma/ print;     # Wrong!
    }

3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Execution of script aborted due to compilation errors.
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop iterates through each line of text. Each line of input is assigned to $_, the default scalar used to hold a line of input and to test pattern matches.

2 The modifier must be at the end of the expression, or a syntax error results. This statement should be *print if /Norma/* or *if(/Norma/) {print;}*. (Similar to the *grep* command for UNIX.)

**The *unless* Modifier.** The *unless* modifier is used to control a simple statement consisting of two expressions. If *Expression1* is false, *Expression2* is executed. Like the *if* modifier, *unless* is placed at the end of the statement.

## FORMAT

```
Expression2 unless Expression1;
```

## EXAMPLE 8.7

```
(The Script)
1   $x=5;
2   print $x unless $x == 6;

(Output)
5
```

## EXPLANATION

The *unless* modifier controls the *print* statement. If the expression *$x == 6* is false, then the value of *$x* is printed.

## EXAMPLE 8.8

```
(The Script)
1   while(<DATA>){
2       print unless /Norma/;  # Print line if it doesn't match Norma
    }

3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Jon DeLoach
Karen Evich
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line below the _ _*DATA*_ _ token is read in and assigned to $_, and so on.

2 All lines that don't contain the pattern *Norma* are matched and printed. (Similar to the *grep -v* command for UNIX.)

3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

### 8.2.3 Looping Modifiers

**The *while* Modifier.** The *while* modifier repeatedly executes the second expression as long as the first expression is true.

**FORMAT**

```
Expression2 while Expression1;
```

**EXAMPLE 8.9**

```
(The Script)
1   $x=1;
2   print $x++,"\n" while $x != 5;

(Output)
1
2
3
4
```

**EXPLANATION**

Perl prints the value of *$x* while *$x* is not *5*.

**The *until* Modifier.** The *until* modifier repeatedly executes the second expression as long as the first expression is false.

**FORMAT**

```
Expression2 until Expression1;
```

**EXAMPLE 8.10**

```
(The Script)
1   $x=1;
2   print $x++,"\n" until $x == 5;

(Output)
1
2
3
4
```

**EXPLANATION**

1 *$x* is assigned an initial value of *1*.
2 Perl prints the value of *$x* until *$x* is equal to *5*. The variable *$x* is set to 1 and then incremented. Be careful that you don't get yourself into an infinite loop.

**The *foreach* Modifier.** The *foreach* modifier evaluates once for each element in its list, with $_ aliased to each element of the list, in turn.

**EXAMPLE 8.11**

```
(The Script)
1   @alpha=(a .. z, "\n");
2   print foreach @alpha;

(Output)
abcdefghijklmnopqrstuvwxyz
```

**EXPLANATION**

1 A list of lowercase letters is assigned to array *@alpha*.
2 Each item in the list is aliased to $_ and printed, one at a time, until there are no more items in the list.

## 8.3 Regular Expression Operators

The regular expression operators are used for matching patterns in searches and for replacements in substitution operations. The *m* operator is used for matching patterns, and the *s* operator is used when substituting one pattern for another.

### 8.3.1 The *m* Operator and Matching

The *m* operator is used for matching patterns. The *m* operator is optional if the delimiters enclosing the regular expression are forward slashes (the forward slash is the default) but required if you change the delimiter. You may want to change the delimiter if the regular expression itself contains forward slashes (e.g., when searching for birthdays, such as *3/15/93*, or pathnames, such as */usr/var/adm*).

**FORMAT**

```
/Regular Expression/      default delimiter
m#Regular Expression#     optional delimiters
m{regular expression}     pair of delimiters
```

**Table 8.1** Matching Modifiers

| *Modifier* | *Meaning* |
|---|---|
| *i* | Turn off case sensitivity. |
| *m* | Treat a string as multiple lines. |

**Table 8.1** Matching Modifiers (continued)

| *Modifier* | *Meaning* |
|---|---|
| *o* | Compile pattern only once. Used to optimize the search. |
| *s* | Treat string as a single line when a newline is embedded. |
| *x* | Permit comments in a regular expression and ignore whitespace. |
| *g* | Match globally; i.e., find all occurrences. Return a list if used with an array context, or true or false if a scalar context. |

## EXAMPLE 8.12

```
1   m/Good morning/
2   /Good evening/
3   /\/usr\/var\/adm/
4   m#/usr/var/adm#
5   m(Good evening)
6   m'$name'
```

## EXPLANATION

1 The *m* operator is not needed in this example, since forward slashes delimit the regular expression.
2 The forward slash is the delimiter; therefore, the *m* operator is optional.
3 Each of the forward slashes in the search path is quoted with a backslash so it will not be confused with the forward slash used for the pattern delimiter—a messy approach.
4 The *m* operator is required because the pound sign (#) is used as an alternative to the forward slash. The pound sign delimiter clarifies and simplifies the previous example.
5 If the opening delimiter is a parenthesis, square bracket, angle bracket, or brace, then the closing delimiter must be the corresponding closing character, such as *m(expression)*, *m[expression]*, *m<expression>*, or *m{expression}*.
6 If the delimiter is a single quote, then variable interpolation is turned off; in other words, *$name* is treated as a literal.

## EXAMPLE 8.13

```
(The Script)
1   while(<DATA>){
2       print if /Betty/;      # Print the line if it matches Betty
    }
```

## EXAMPLE 8.13 (CONTINUED)

```
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Betty Boop
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to $_.
2 All lines that match the pattern *Betty* are matched and printed.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.14

```
(The Script)
1   while(<DATA>){
2       print unless /Evich/;   # Print line unless it matches Evich
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Norma Cord
Jon DeLoach
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line from under the _ _*DATA*_ _ token is read in and assigned to $_.
2 All lines that don't match the pattern *Evich* are printed.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.15

```
(The Script)
1   while(<DATA>){
2       print if m#Jon#      # Print the line if it matches Jon
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Jon DeLoach
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to $_.

2 The *m* (match) operator is necessary because the delimiter has been changed from the default forward slash to a pound sign (#). The line is printed if it matches *Jon*.

3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.16

```
(The Script)
1   while(<DATA>){
2       print if m(Karen E);   # Print the line if it matches Karen E
    }
3   $name="Jon";
4   $_=qq/$name is a good sport.\n/;
5   print if m'$name';
6   print if m"$name";

7   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
2   Karen Evich
5   <No output>
6   Jon is a good sport.
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line below the _ _*DATA*_ _ token is read in and assigned to $_.
2 The *m* (match) operator is necessary because the delimiter has been changed from the default forward slash to a set of opening and closing parentheses. Other pairs that could be used are square brackets, curly braces, angle brackets, and single quotes. If single quotes are used, and the regular expression contains variables, the variables will not be interpolated. The line is printed if it matches *Karen E*.
3 The scalar *$name* is assigned *Jon*.
4 $_ is assigned a string including the scalar *$name*.
5 When the matching delimiter is a set of single quotes, variables in the regular expression are not interpolated. The literal value *$name* is not found in $_; therefore, nothing is printed.
6 If double quotes enclose the expression, the variable *$name* will be interpolated. The string assigned to $_ is printed if it contains *Jon*.
7 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

**The *g* Modifier—Global Match.** The *g* modifier is used to cause a global match; in other words, all occurrences of a pattern in the line are matched. Without the *g*, only the first occurrence of a pattern is matched. The *m* operator will return a list of the patterns matched.

## FORMAT

```
m/search pattern/g
```

## EXAMPLE 8.17

```
(The Script)
   #!/usr/bin/perl
1  $_ = "I lost my gloves in the clover, Love.";
2  @list=/love/g;
3  print "@list.\n";

(Output)
3  love love.
```

## EXPLANATION

1 The $_ scalar variable is assigned a string of text.
2 If the search is done with the *g* modifier, in an array context, each match is stored in the *@list* array. The regular expression *love* was found in the string twice, once in *gloves* and once in *clover*. *Love* is not matched, since the *L* is uppercase.
3 The list of matched items is printed.

**The *i* Modifier—Case Insensitivity.** Perl is sensitive to whether characters are upper- or lowercase when performing matches. If you want to turn off case sensitivity, an *i* (insensitive) is appended to the last delimiter of the match operator.

## FORMAT

```
m/search pattern/i
```

## EXAMPLE 8.18

```
1   $_ = "I lost my gloves in the clover, Love.";
2   @list=/love/gi;
3   print "@list.\n";

(Output)
3   love love Love.
```

## EXPLANATION

1 The $_ scalar variable is assigned the string.
2 This time the *i* modifier is used to turn off the case sensitivity. Both *love* and *Love* will be matched and assigned to the array *@list*.
3 The pattern was found three times. The list is printed.

**Special Scalars for Saving Patterns.** The *$&* special scalar is assigned the string that was matched in the last successful search. *&`* saves what was found preceding the pattern that was matched, and *&'* saves what was found after the pattern that was matched.

## EXAMPLE 8.19

```
1   $_="San Francisco to Hong Kong\n";

2   /Francisco/;      # Save 'Francisco' in $& if it is found
3   print $&,"\n";

4   /to/;
5   print $`,"\n";    # Save what comes before the string 'to'

6   /to\s/;           # \s represents a space
7   print $', "\n";   # Save what comes after the string 'to'

(Output)
3   Francisco
5   San Francisco
7   Hong Kong
```

**EXPLANATION**

1 The $_ scalar is assigned a string.
2 The search pattern contains the regular expression *Francisco*. Perl searches for this pattern in the $_ variable. If found, the pattern *Francisco* will be saved in another special scalar, *$&*.
3 The search pattern *Francisco* was successfully matched, saved in *$&*, and printed.
4 The search pattern contains the regular expression *to*. Perl searches for this pattern in the $_ variable. If the pattern *to* is matched, the string to the **left** of this pattern, *San Francisco*, is saved in the $` scalar (note the backquote).
5 The value of $` is printed.
6 The search pattern contains the regular expression *to*\s (*to* followed by a space; \s represents a space). Perl searches for this pattern in the $_ variable. If the pattern *to*\s is matched, the string to the **right** of this pattern, *Hong Kong*, is saved in the *$'* scalar (note the straight quote).
7 The value of *&'* is printed.

**The *x* Modifier—The Expressive Modifier.** The *x* modifier allows you to place comments within the regular expression and add whitespace characters (spaces, tabs, newlines) for clarity without having those characters interpreted as part of the regular expression; in other words, you can *express* your intentions within the regular expression.

**EXAMPLE 8.20**

```
1   $_="San Francisco to Hong Kong\n";
2   /Francisco  # Searching for Francisco
    /x;
3   print "Comments and spaces were removed and \$& is $&\n";

(Output)
3   Comments and spaces were removed and $& is Francisco
```

**EXPLANATION**

1 The $_ scalar is assigned a string.
2 The search pattern consists of *Francisco* followed by a space, comment, and another space. The *x* modifier allows the additional whitespace and comments to be inserted in the pattern space without being interpreted as part of the search pattern.
3 The printed text illustrates that the search was unaffected by the extra spaces and comments. *$&* holds the value of what was matched as a result of the search.

## 8.3.2 The *s* Operator and Substitution

The *s* operator is used for substitutions. The substitution operator replaces the first regular expression pattern with the second. The delimiter can also be changed. The *g* modifier

placed after the last delimiter stands for **global change** on a line. The return value from the *s* operator is the number of substitutions that were made. Without it, only the first occurrence of the pattern is affected by the substitution.

The special built-in variable *$&* gets the value of whatever was found in the search string.

## FORMAT

```
s/old/new/;
s/old/new/i;
s/old/new/g;
s+old+new+g;
s(old)/new/;   s[old]{new};
s/old/expression to be evaluated/e;
s/old/new/ige;
s/old/new/x;
```

## EXAMPLE 8.21

```
s/Igor/Boris/;
s/Igor/Boris/g;
s/norma/Jane/i;
s!Jon!Susan!;
s{Jon} <Susan>;
s/$sal/$sal * 1.1/e
s/dec/"Dec" . "ember"      # Replace "dec" or "Dec" with "December"
 /eigx;
```

**Table 8.2** Substitution Modifiers

| *Modifier* | *Meaning* |
|---|---|
| *e* | Evaluate the replacement side as an expression. |
| *i* | Turn off case sensitivity. |
| *m* | Treat a string as multiple lines.[a] |
| *o* | Compile pattern only once. Used to optimize the search. |
| *s* | Treat string as single line when newline is embedded. |
| *x* | Allow whitespace and comments within the regular expression. |
| *g* | Replace globally; i.e., find all occurrences. |

a. The *m*, *s*, and *x* options are defined only for Perl 5.

## EXAMPLE 8.22

```
(The Script)
1   while(<DATA>){
2       s/Norma/Jane/;     # Substitute Norma with Jane
3       print;
    }
4   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Jane Cord
Jon DeLoach
Karen Evich
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to $_.

2 In lines where $_ contains the regular expression *Norma*, the substitution operator, *s*, will replace *Norma* with *Jane* for the first occurrence of *Norma* on each line. (Similar to vi and *sed* commands for UNIX.)

3 Each line will be printed, whether or not the substitution occurred.

4 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.23

```
(The Script)
1   while($_= <DATA>){
2       print if s/Igor/Ivan/;     # Substitute Igor with Ivan
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich
```

## EXAMPLE 8.23 (CONTINUED)

```
(Output)
Ivan Chevsky
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to $_.
2 In lines where $_ contains the regular expression *Igor*, the substitution operator, s, will replace *Igor* with *Ivan* for the first occurrence of *Igor* on each line. Only if the substitution is successful will the line be printed.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

**Changing the Substitution Delimiters.** Normally, the forward slash delimiter encloses both the search pattern and the replacement string. Any nonalphanumeric character following the *s* operator can be used in place of the slash. For example, if a # follows the *s* operator, it must be used as the delimiter for the replacement pattern. If pairs of parentheses, curly braces, square brackets, or angle brackets are used to delimit the search pattern, any other type of delimiter may be used for the replacement pattern, such as *s(John) /Joe/*;

## EXAMPLE 8.24

```
(The Script)
1   while(<DATA>){
2       s#Igor#Boris#;        # Substitute Igor with Boris
3       print;
    }
4   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Boris Chevsky
Norma Cord
Jon DeLoach
Karen Evich
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to *$_*.
2 The delimiter following the *s* operator has been changed to a pound sign (#). This is fine as long as all three delimiters are pound signs. The regular expression *Igor* is replaced with *Boris*.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.25

```
(The Script)
1   while(<DATA>){
2       s(Blenheim){Dobbins};      # Substitute Blenheim with Dobbins
3       print;
    }
4   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Dobbins
Betty Boop
Igor Chevsky
Norma Cord
Jon DeLoach
Karen Evich
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to *$_*.
2 The search pattern *Blenheim* is delimited with parentheses and the replacement pattern, *Dobbins*, is delimited with forward slashes.
3 The substitution is shown in the output when it is printed. *Blenheim* is replaced with *Dobbins*.
4 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

**The *g* Modifier—Global Substitution.** The *g* modifier is used to cause a global substitution; that is, all occurrences of a pattern are replaced on the line. Without the *g*, only the first occurrence of a pattern on each line is changed.

## FORMAT

```
s/search pattern/replacement string/g;
```

## EXAMPLE 8.26

```
(The Script)
# Without the g option
    (The Script)
1   while(<DATA>){
2       print if s/Tom/Christian/;  # First occurrence of Tom on each
                                    # line is replaced with Christian
    }
3   _ _DATA_ _
    Tom Dave Dan Tom
    Betty Tom Henry Tom
    Igor Norma Tom Tom

(Output)
Christian Dave Dan Tom
Betty Christian Henry Tom
Igor Norma Christian Tom
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to *$_*.
2 The **first** occurrence of *Tom* will be replaced with *Christian* for each line that is read.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.27

```
(The Script)
# With the g option
1   while(<DATA>){
2       print if s/Tom/Christian/g; # All occurrences of Tom on each
                                    # line are replaced with Christian
    }
3   _ _DATA_ _
    Tom Dave Dan Tom
    Betty Tom Henry Tom
    Igor Norma Tom Tom

(Output)
Christian Dave Dan Christian
Betty Christian Dick Christian
Igor Norma Christian Christian
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to *$_*.
2 With the *g* option, the substitution is global. **Every** occurrence of *Tom* will be replaced with *Christian* for each line that is read.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

**The *i* Modifier—Case Insensitivity.** Perl is sensitive to upper- or lowercase characters when performing matches. If you want to turn off case sensitivity, an *i* (insensitive) is appended to the last delimiter of the match or substitution operator.

## FORMAT

```
s/search pattern/replacement string/i;
```

## EXAMPLE 8.28

```
(The Script)
    # Matching with the i option
1   while(<DATA>){
2       print if /norma cord/i;     # Turn off case sensitivity
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Norma Cord
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to *$_*.
2 Without the *i* option, the regular expression */norma cord/* would not be matched, because all the letters are not lowercase in the lines that are read as input. The *i* option turns off case sensitivity.
3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

## EXAMPLE 8.29

```
(The Script)
1   while(<DATA>){
2       print if s/igor/Daniel/i;    # Substitute igor with Daniel
    }

3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Daniel Chevsky
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to $_. Each time the loop is entered, the next line following _ _*DATA*_ _ is assigned to $_ until all the lines have been processed.

2 The regular expression in the substitution is also caseinsensitive, owing to the *i* option. If *igor* or *Igor* (or any combination of upper- and lowercase) is matched, it will be replaced with *Daniel*.

3 The *DATA* filehandle gets its data from the lines that follow the _ _*DATA*_ _ token.

**The *e* Modifier—Evaluating an Expression.** On the replacement side of a substitution operation, it is possible to evaluate an expression or a function. The search side is replaced with the result of the evaluation.

## FORMAT

```
s/search pattern/replacement string/e;
```

## EXAMPLE 8.30

```
(The Script)
    # The e and g modifiers
1   while(<DATA>){
2       s/6/6 * 7.3/eg;        # Substitute 6 with product of 6 * 7.3

3       print;
    }
```

## EXAMPLE 8.30 (CONTINUED)

```
    _ _DATA_ _
    Steve Blenheim   5
    Betty Boop       4
    Igor Chevsky     6
    Norma Cord       1
    Jon DeLoach      3
    Karen Evich      66


(Output)
Steve Blenheim     5
Betty Boop         4
Igor Chevsky       43.8
Norma Cord         1
Jon DeLoach        3
Karen Evich        43.843.8
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to $_. Each time the loop is entered, the next line after _ _*DATA*_ _ is assigned to $_ until all the lines have been processed.

2 If the $_ scalar contains the number *6*, the replacement side of the substitution is evaluated. In other words, the *6* is multiplied by *7.3* (*e* modifier); the product of the multiplication (*43.8*) replaces the number *6* each time the number *6* is found (*g* modifier).

3 Each line is printed. The last line contained two occurrences of *6*, causing each *6* to be replaced with *43.8*.

## EXAMPLE 8.31

```
(The Script)
    # The e modifier
1   $_=5;
2   s/5/6 * 4 - 22/e;
3   print "The result is: $_\n";

4   $_=1055;
5   s/5/3*2/eg;
6   print "The result is: $_\n";

(Output)
3   The result is: 2
6   The result is: 1066
```

## EXPLANATION

1 The $_ scalar is assigned *5*.
2 The *s* operator searches for the regular expression *5* in $_. The *e* modifier evaluates the replacement string as a numeric expression and replaces it with the result of the arithmetic operation, *6* 4 – 22*, which results in *2*.
3 The result of the evaluation is printed.
4 The $_ variable is assigned *1055*.
5 The *s* operator searches for the regular expression *5* in $_. The *e* modifier evaluates the replacement string as a numeric expression and replaces it with the product of *3*2*; i.e., every time *5* is found, it is replaced with *6*. Since the substitution is global, all occurrences of *5* are replaced with *6*.
6 The result of the evaluation is printed.

## EXAMPLE 8.32

```
(The Script)
1   $_ = "knock at heaven's door.\n";
2   s/knock/"knock, " x  2  . "knocking"/ei;
3   print "He's $_;

(Output)
He's knock, knock, knocking at heaven's door.
```

## EXPLANATION

1 The $_ variable is the string *knock at heaven's door.*\n;
2 The *s* operator searches for the regular expression *knock* in $_. The *e* modifier evaluates the replacement string as a string expression and replaces it with *knock x 2* (repeated twice) and concatenates (the dot operator) with the string *knocking*, ignoring case.
3 The resulting string is printed.

## EXAMPLE 8.33

```
(The Script)
    # Saving in the $& special scalar
1   $_=5000;
2   s/$_/$& * 2/e;
3   print "The new value is $_\.n";

4   $_="knock at heaven's door.\n";
5   s/knock/"$&," x 2 . "$&ing"/ei;
6   print "He's $_";
```

**EXAMPLE 8.33 (CONTINUED)**

```
(Output)
3   The new value is 10000.
6   He's knock,knock,knocking at heaven's door.
```

**EXPLANATION**

1 The $_ scalar is assigned *5000*.
2 The search string, *5000*, is stored in the *$&* variable. In the replacement side the expression is evaluated; in other words, the value of *$&* is multiplied by 2. The new value is substituted for the original value. $_ is assigned the new value.
3 The resulting value is printed.
4 The $_ scalar is assigned the string *knock at heaven's door.\n*.
5 If the search string (*knock*) is found, it is stored in the *$&* variable. In the replacement side, the expression is evaluated. So, the value of *$&* (*knock*) is replicated twice and concatenated with *$&ing* (*knocking*). The new value is substituted for the original value. $_ is assigned the new value and printed.

### 8.3.3 Pattern Binding Operators

The **pattern binding** operators are used to bind a matched pattern, substitution, or translation (see *tr* in Appendix A) to another scalar expression. In the previous examples, pattern searches were done implicitly (or explicitly) on the $_ variable, the default pattern space. That is, each line was stored in the $_ variable when looping through a file. In the previous example, the $_ was assigned a value and used as the search string for the substitution. But what if you store a value in some variable other than $_?

Instead of

```
$_ = 5000;
```

you would write

```
$salary = 5000;
```

Then if a match or substitution is performed on *$salary* instead of

```
print if /5/;  or  s/5/6;
```

you would write

```
print if $salary =~ /5/;  or  $salary =~ s/5/6/;
```

So, if you have a string that is not stored in the $_ variable and need to perform matches or substitutions on that string, the pattern binding operators =~ or !~ are used. They are also used with the *tr* function for string translations.

The pattern matching operators are listed in Table 8.3.

## FORMAT

```
Variable =~ /Expression/
Variable !~ /Expression/
Variable =~ s/old/new/
```

**Table 8.3** Pattern Matching Operators

| ***Example*** | ***Meaning*** |
|---|---|
| *$name =~ /John/* | True if *$name* contains pattern. Returns 1 for *true*, null for *false*. |
| *$name !~ /John/* | True if *$name* does not contain pattern. |
| *$name =~ s/John/Sam/* | Replace first occurrence of *John* with *Sam*. |
| *$name =~ s/John/Sam/g* | Replace all occurrences of *John* with *Sam*. |
| *$name =~ tr/a–z/A–Z/* | Translate all lowercase letters to uppercase. |
| *$name =~ /$pal/* | A variable can be used in the search string. |

## EXAMPLE 8.34

```
(The Script)
    # Using the $_ scalar explicitly
1   while($_=<DATA>){
2       print $_ if  $_ =~ /Igor/;  # $_ holds the current input line
3   #   print if /Igor/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Igor Chevsky
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to $_. Each time the loop is entered, the next line after _ _*DATA*_ _ is assigned to $_ until all the lines have been processed.
2 If the regular expression */Igor/* is matched in the $_ variable, the *print* function will print the value of $_. The =~ is necessary here only if the $_ scalar is explicitly used as an operand.
3 If the =~ pattern matching operator is omitted, the default is to match on $_, and if the *print* function is given no arguments, the value of $_ is also printed.

## EXAMPLE 8.35

```
(The Script)
    #!/usr/bin/perl
1   $name="Tommy Tuttle";
2   print "Hello Tommy\n"  if $name =~ /Tom/;
                                    # Prints Hello Tommy,if true
3   print "$name\n" if  $name !~ /Tom/;  # Prints nothing if false

4   $name =~ s/T/M/;                # Substitute first T with an M
5   print "$name.\n";

6   $name="Tommy Tuttle";
7   print "$name\n" if $name =~ s/T/M/g;  # Substitute every T with M
8   print "What is Tommy's last name? ";
9   print "You got it!\n" if <STDIN> =~ /Tuttle/;

(Output)
2   Hello Tommy
5   Mommy Tuttle.
7   Mommy Muttle
8   What is Tommy's last name? Tuttle
9   You got it!
```

## EXPLANATION

1 The scalar *$name* is assigned *Tommy Tuttle*.
2 The string *$name* is printed if *$name* contains the pattern *Tom*. The return value from a successful match is *1*.
3 The string *$name* is not printed if *$name* does **not** contain the pattern *Tom*. The return value from an unsuccessful match is null.
4 The first occurrence of the letter *T* in *$name* is replaced with the letter *M*.
5 *$name* is printed, reflecting the substitution.
6 *$name* is assigned *Tommy Tuttle*.

## EXPLANATION (CONTINUED)

7 **All** occurrences of the letter *T* in *$name* are replaced with the letter *M*. The *g* at the end of the substitution expression causes a global replacement across the line.
8 User input is requested.
9 The user input (<*STDIN*>) is matched against the regular expression *Tuttle*, and if there is a match, the *print* statement is executed.

## EXAMPLE 8.36

```
(The Script)
1   $salary=50000;
2   $salary =~ s/$salary/$& * 1.1/e;
3   print "\$& is $&\n";
4   print "The salary is now \$$salary.\n";

(Output)
3   $& is 50000
4   The salary is now $55000.
```

## EXPLANATION

1 The scalar *$salary* is assigned *50000*.
2 The substitution is performed on *$salary*. The replacement side evaluates the expression. The special variable *$&* holds the value found on the search side. To change the value in *$salary* after the substitution, the pattern matching operator **=~** is used. This binds the result of the substitution to the scalar *$salary*.
3 The *$&* scalar holds the value of what was found on the search side of the substitution.
4 The scalar *$salary* has been increased by *10%*.

## EXAMPLE 8.37

```
(The Script)
    # Using split and pattern matching
1   while(<DATA>){
2       @line = split(":", $_);
3       print $line[0],"\n"  if $line[1] =~ /408-/
                               # Using the pattern matching operator
    }
4   _ _DATA_ _
    Steve Blenheim:415-444-6677:12 Main St.
    Betty Boop:303-223-1234:234 Ethan Ln.
    Igor Chevsky:408-567-4444:3456 Mary Way
    Norma Cord:555-234-5764:18880 Fiftieth St.
    Jon DeLoach:201-444-6556:54 Penny Ln.
    Karen Evich:306-333-7654:123 4th Ave.
```

## EXAMPLE 8.37 (CONTINUED)

```
(Output)
Igor Chevsky
```

## EXPLANATION

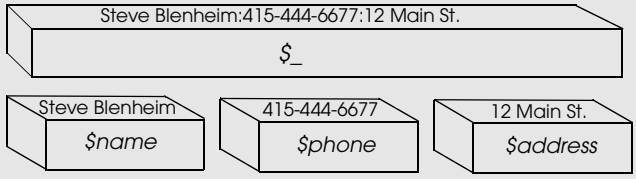
1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line following the _ _*DATA*_ _ token is read in and assigned to $_. Each time the loop is entered, the next line from _ _*DATA*_ _ is assigned to $_ until all the lines have been processed.

2 Each line from the file will be split at the colons and the value returned stored in an array, *@line*.

3 The pattern */408–/* is matched against the array element *$line[1]*. If that pattern is matched in *$line[1]*, the value of *$line[0]* is printed. Prints *Igor*'s name, *$line[0]*, because his phone, *$line[1]*, matches the *408* area code.

4 The text following _ _*DATA*_ _ is used as input by the special *DATA* filehandle.

## EXAMPLE 8.38




```
(The Script)
    # Using split, an anonymous list, and pattern matching
1   while(<DATA>){
2       ($name, $phone, $address) = split(":", $_);
3           print $name  if $phone =~ /408-/    # Using the pattern
                                                # matching operator
    }

4   _ _DATA_ _
    Steve Blenheim:415-444-6677:12 Main St.
    Betty Boop:303-223-1234:234 Ethan Ln.
    Igor Chevsky:408-567-4444:3456 Mary Way
    Norma Cord:555-234-5764:18880 Fiftieth St.
    Jon DeLoach:201-444-6556:54 Penny Ln.
    Karen Evich:306-333-7654:123 4th Ave.

(Output)
Igor Chevsky
```

## EXPLANATION

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to *$_*. Each time the loop is entered, the next line following _ _*DATA*_ _ is assigned to *$_* until all the lines have been processed.

2 Each line from the file will be split at the colons and the value returned stored in an anonymous list consisting of three scalars: *$name*, *$phone*, and *$address*. Using the anonymous list makes the program easier to read and manipulate than in the previous example where an array was used. With the array, you have to make sure you get the right index number to represent the various fields, whereas the named scalars are straightforward.

3 The pattern */408-/* is matched against the *$phone* variable. If that pattern is matched in *$phone*, the value of *$name* is printed. *Igor*'s name is printed because his phone matches the *408* area code.

4 The text following _ _*DATA*_ _ is used as input by the special *DATA* filehandle.

## EXAMPLE 8.39

```
(The Script)
1   while($inputline=<DATA>){
2       ($name, $phone, $address) = split(":", $inputline);
3       print $name if $phone =~ /^408-/;  # Using the pattern
                                           # matching operator
4       print $inputline if $name =~ /^Karen/;
5       print if /^Norma/;
    }

6   _ _DATA_ _
    Steve Blenheim:415-444-6677:12 Main St.
    Betty Boop:303-223-1234:234 Ethan Ln.
    Igor Chevsky:408-567-4444:3456 Mary Way
    Norma Cord:555-234-5764:18880 Fiftieth St.
    Jon DeLoach:201-444-6556:54 Penny Ln.
    Karen Evich:306-333-7654:123 4th Ave.

(Output)
3   Igor Chevsky
4   Karen Evich:306-333-7654:123 4th Ave.
5   < No output >
```

**EXPLANATION**

1 The special *DATA* filehandle gets its input from the text after the _ _*DATA*_ _ token. The *while* loop is entered and the first line after the _ _*DATA*_ _ token is read in and assigned to a user-defined variable, *$inputline*, rather than *$_*. Each time the loop is entered, the next line from _ _*DATA*_ _ is assigned to *$inputline* until all the lines have been processed.
2 Each line from the file, stored in *$inputfile*, will be split at the colons and the value returned stored in an anonymous list consisting of three scalars: *$name*, *$phone*, and *$address*.
3 The pattern */408-/* is matched against the *$phone* variable. If that pattern is matched in *$phone*, the value of *$name* is printed. Prints *Igor*'s name because his phone matches the *408* area code.
4 Each line is stored in *$inputline*, one after the other, until the end of the file is reached. The value of *$inputline* is displayed if it begins with the regular expression *Karen*.
5 Since the default line holder, *$_*, is no longer being used, nothing is assigned to it, and nothing is matched against it or displayed. The lines are now being stored and matched in the user-defined variable *$inputline*.
6 The text following _ _*DATA*_ _ is used as input by the special *DATA* filehandle.

## 8.4 What You Should Know

1. What is meant by a regular expression?
2. How are the *if* and *unless* modifiers used?
3. How do you change the forward slash delimiter used in the search pattern to something else?
4. What does the *s* operator do?
5. What is meant by a global search?
6. When do you need the pattern binding operators, =~ and !~?
7. What is the default pattern space holder?
8. What is the _ _DATA_ _ filehandle used for?
9. What do the *ieg* modifiers mean?

## 8.5 What's Next?

In the next chapter, you will harness the power of pattern matching by learning Perl's plethora of regular expression metacharacters. You will learn how to anchor patterns,

how to search for alternating patterns, whitespace characters, sets of characters, repeating patterns, etc. You will learn about greedy metacharacters and how to control them. You will learn about capturing and grouping patterns, to look ahead and behind. By the time you have completed this chapter, you should be able to search for data by regular expressions based on a specific criterion in order to validate the data and to modify the text that was found.

## EXERCISE 8
## A Match Made in Heaven

```
(sample.file found on CD)
Tommy Savage:408-724-0140:1222 Oxbow Court, Sunnyvale,CA 94087:5/19/66:34200
Lesle Kerstin:408-456-1234:4 Harvard Square, Boston, MA 02133:4/22/62:52600
JonDeLoach:408-253-3122:123 Park St., San Jose, CA 94086:7/25/53:85100
Ephram Hardy:293-259-5395:235 Carlton Lane, Joliet, IL 73858:8/12/20:56700
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
William Kopf:846-836-2837:6937 Ware Road, Milton, PA 93756:9/21/46:43500
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI 23874:3/28/45:245700
James Ikeda:834-938-8376:23445 Aster Ave., Allentown, NJ 83745:12/1/38:45000
Lori Gortz:327-832-5728:3465 Mirlo Street, Peabody, MA 34756:10/2/65:35200
Barbara Kerz:385-573-8326:832 Ponce Drive, Gary, IN 83756:12/15/46:268500
```

1. Print all lines containing the pattern *Street*.
2. Print lines where the first name matches a *B* or *b*.
3. Print last names that match *Ker*.
4. Print phone numbers in the *408* area code.
5. Print Lori Gortz's name and address.
6. Print Ephram's name in capital letters.
7. Print lines that do not contain a *4*.
8. Change William's name to Siegfried.
9. Print Tommy Savage's birthday.
10. Print the names of those making over $40,000.
11. Print the names and birthdays of those people born in June.
12. Print the zip code for Massachusetts.