17

USING PERL TO FACILITATE BIOLOGICAL ANALYSIS

Lincoln D. Stein

The Cold Spring Harbor Laboratory
Cold Spring Harbor, New York

Consider a situation in which an investigator is studying genes that affect neuronal signaling in *C. elegans*, with a primary interest in identifying those with gene products that may be secreted. A Web site that reports the results of a large systematic study of predicted *C. elegans* genes using the RNA-induced inhibition of gene expression (RNAi) technique is available, and the investigator can download a summary file based on a few thousand experiments. The latest release of WormPep, which reports peptide sequences of over 19,000 known and predicted worm genes, is also available. What is now needed is the ability to search the RNAi results file for those genes that affect worm movement in some way (e.g., a common phenotype for genes affecting neuronal activity), and extract the sequence of those genes from WormPep. The ultimate plan is to submit these sequences of interest to SignalP, an E-mail-based signal peptide cleavage site predictor based at the Technical University of Denmark (discussed in Chapter 11).

How would one accomplish this? One way is to do the job by hand. First, one would need to read the RNAi summary file into a word processing program, cull it for experiments that affected locomotion in some way, and then assemble a list of all of the genes that produce relevant phenotypes. Next, one would open WormPep, search for the corresponding sequence for each of these genes, and cut-and-paste these sequences into another file. The last step would be to reformat the entries into the format required by the SignalP server, pasting all of the entries into an E-mail message.

It should be immediately apparent that there are some problems with this approach. If there are more than just a few genes of interest to analyze, the job quickly becomes rather tedious, if not overly time consuming. Worse yet, the next time that new RNAi results are released, the whole process will need to be repeated, determining which RNAi entries are new. The step involving loading WormPep into a word processor might not even be tenable due to the sheer size of the database, well over 10 megabases.

This is the type of problem that a Perl script can help with. The Perl programming language excels at slicing, dicing, and integrating data files and is the language of choice for the many bioinformatics researchers. This chapter will provide a gentle introduction to Perl, with examples designed to illustrate the usefulness of learning this language.

GETTING STARTED

During the course of this chapter, a solution will be developed for the data integration problem introduced in the first paragraph of this chapter. Before this problem is attacked, however, some very simple scripts that illustrate the basics of Perl programming will be discussed. In considering these examples, the reader is strongly encouraged to follow along by typing the examples into a Perl interpreter to get a better sense for what these short scripts actually do. Modifying and experimenting with the scripts for individual use is also encouraged.

Perl interpreters are available for the Macintosh, Windows, and UNIX operating systems and more often than not is made available as freeware. A number of download sites are listed at the end of this chapter. Because Perl program files are usually compressed to speed downloads, Macintosh users will need to use a utility like UnStuffIt to uncompress the package, whereas Windows users will need WinZip or PKZip. Perl should come as part of the standard software installation on most UNIX-based machines. Typing `perl -v` at the UNIX prompt will indicate whether the program is indeed installed and, if so, the version number that is available on that machine.

Perl consists of two essential parts:

the *interpreter*, called perl on UNIX systems, MacPerl on Macintoshes, and perl.exe on Windows machines; and

scripts, text files that are written by the user describing a discrete set of steps to be performed by the interpreter. The scripts are actually computer programs, and the words script and program can be used interchangeably.

The process of writing and running a Perl script is similar on Windows and UNIX systems but slightly different for Macintosh users. The basic steps are described below.

On a Windows or UNIX system, create a text-only file containing the following lines:

```
#!/usr/bin/perl
print "My first Perl script.";
```

Any word processing program may be used, as long as the file is saved as `text only`. The Windows Notepad program is good for this task, as it saves its files in text-only format by default. Name the file `first.pl`.

This newly created file contains two lines. The first line is a comment that identifies the file as a Perl script (indicated by the `#!` at the beginning of the line). The second line is a *print statement* that tells Perl to print out the text

```
My first Perl script
```

The name of the file ends in the extension `.pl`, which is a standard naming convention for Perl scripts.

To run this command, open a command-line window (the DOS window on Windows systems, a shell window on UNIX), change to the directory that contains the file, and type the command

```
% perl first.pl
My first Perl script.
```

where `%` represents the command-line prompts for both the Windows and UNIX systems, and boldface type represents input typed by the user at the keyboard.

What this particular command does is invoke the Perl command interpreter, passing to it the name of the file that should be run. The interpreter dutifully processes the script line-by-line, sees the single print command, and executes it. The output of the script appears in the command window.

If Perl is installed in the standard way on the computer, the `perl` command does not have to be explicitly typed. On UNIX systems, the script file can be marked as being directly executable using the `chmod` command:

```
% chmod + x first.pl
```

On Windows systems, the file does not have to be explicitly marked as executable because Perl is usually installed in such a way that any file ending in the `.pl` extension is associated with the Perl interpreter. Simply typing in the name of the program will run it:

```
% first.pl
My first Perl script.
```

Mac-based Perl does not have a command window, so the process of creating and running a Perl script is somewhat different. On Macintoshes, the MacPerl application is launched by double clicking on its icon (a pyramid with a camel). This will launch a special-purpose text editor from which the user can create, run, and debug Perl scripts. Select New from the File menu to bring up a text-editing window. Type the script shown above, beginning with the `#!/usr/bin/perl` line and choose `Save As` from the `File` menu. Give the script a name, such as `first.pl`, and from the list of Type options in the Save dialog, choose Droplet. This will save the script as a miniature application called a *droplet*, which has a distinctive pyramid-shaped icon (containing another camel). Double clicking on this new icon will produce a text window containing the output of the script, which simply reads `My first`

`Perl script`. The script can also be run directly through the MacPerl application by choosing Run Script from the Script pull-down menu.

HOW SCRIPTS WORK

A script consists of a series of commands, more formally called *statements*, that are meaningful to the Perl interpreter. Unless told otherwise, the interpreter starts at the top of the script file and works its way down to the bottom, executing each statement in turn.

Consider this new script:

```
#!/usr/bin/perl
# preamble...
print "I can do math!\n";
# do some calculations
$sum = 3 + 4;
# print the result
print "The sum of 3 + 4 is ",$sum,".\n";
```

This script consists of three statements; the first one (`print "I can do math!\n"`) tells the interpreter to print out the indicated text. As we shall discuss in more detail later, the special character sequence `\n` is not interpreted literally but instead prints out as a *newline* character. The second statement, `$sum = 3 + 4`, adds the numbers three and four together and stores the result in a variable named `$sum`. The last statement prints the text `The sum of 3 + 4 is`, followed by the contents of the variable `$sum`, followed by a period and a newline.

Notice that each statement ends with a semicolon. The semicolon tells Perl where one statement ends and another begins. Blank lines and other white space can help make the script more readable but are ignored by the interpreter. Any line that begins with a pound sign (#) is a comment. When Perl sees a pound sign it simply ignores everything between it and the end of the line. The use of comments is strongly encouraged, since it allows other users to better understand what the programmer was trying to accomplish in a particular block of code, as well as reminding the programmer themselves of the same when reexamining code written long before.

The topmost line is also a Perl comment, but on UNIX systems it serves double-duty as a directive to the UNIX shell to tell it to execute the command `/usr/bin/perl` when the script file is executed. On Macintosh and Windows systems, this line is extraneous, but it is better to include it to maintain portability with UNIX machines and as a matter of good form.

Unless otherwise instructed, the Perl interpreter starts at the first statement and works its way to the last. When this script is run, the following output is produced:

```
I can do math!
The sum of 3 + 4 is 7.
```

Notice that the user only sees the output from the two print statements. The statement that performs the addition acts silently, behind the scenes.

STRINGS, NUMBERS, AND VARIABLES

Perl can deal with an astonishing number of data types, including but not limited to text, integers, floating point numbers, complex numbers, and binary numbers.

Following a long computer science tradition of using obscure terms for simple concepts, the Perl term for text is *string*. Strings are surrounded by single or double quotation marks:

```
'I am a string.'
"I am another string."
```

Having two types of quotation marks available makes it easier to create strings that contain embedded quotation marks:

```
'"Anna," she wailed "come quickly! The tiara is gone!"'
```

There are also some more substantial differences between the two types of quotes that will be discussed later in this chapter under *Variable Interpolation*.

Numbers are written just as one would expect:

```
1
49
28.2
-109
6.04E23
```

The last example shows how one represents scientific notation. The `E` means exponent, and the number should be interpreted as 6.02 10^{23}.

If you want your strings to contain special characters, such as tabs or new lines, Perl provides special *escape sequences* to represent them. These escape sequences consist of a backslash followed by a single character. The two most commonly used are `\n`, which begins a new line, and `\t`, which inserts a tab. For example,

```
print "There is a newline\nhere and a tab\tthere.\n";
```

produces the following output:

```
There is a newline
here and a tab        there.
```

Perl only interprets escape sequences when they occur in double-quoted strings. In single-quoted strings, the backslash and the character that follows it are interpreted literally.

Variables provide temporary storage for strings, numbers, and other values. In Perl, variables are arbitrary names preceded by a dollar sign. Examples of valid variable names are shown below.

```
$x
$X
```

```
$i_am_a_variable
$LongVariableName
```

Perl variables are case sensitive. In the list above, `$x` is one variable, and `$X` is a different one entirely.

When first created, variables are empty or *undefined*. Values are assigned to variables using the = sign, also known as the assignment operator:

```
$x = 42;
print 'The value of $x is ',$x,"\n";
```

Assignment works from right to left. In the example above, the number 42 is assigned to the variable `$x`. Once assigned, the variable can be used in the place of a value, as the print statement above shows. The same variable can be used multiple times, using the assignment operator to change its contents.

```
print 'The value of $x is still ',$x,"\n";
$x = 'Mary had a little lamb';
print 'But now the value of $x is ',$x,"\n";
```

This code fragment will now print out the following:

```
The value of $x is still 42
But now the value of $x is Mary had a little lamb
```

Note that there is no restriction on the type of data a variable can hold. In this example, `$x` initially contained an integer and then held a string. Unlike some programming languages, Perl does not require the user to *declare* (formally describe) variables before using them, although this type of checking can be activated if desired. Perl actually has several types of variables. In addition to variables that hold a single value, which are technically called *scalar* variables, there are *arrays* and *hashes*, two types of variables that are capable of holding multiple values. These will be discussed in an upcoming example.

ARITHMETIC

Perl knows basic arithmetic. Symbols known as *operators* are responsible for the various arithmetic operations:

```
 +  addition
 -  subtraction
 *  multiplication
 /  division
 ** exponentiation
 () grouping
```

The following example does a little math and prints out the result:

```
$x = 4;
$y = 2;
$z = 3 +  $x * $y;
print $z,"\n";
```

The result that gets printed out is 11. However, a more succinct way to express this would be to combine the first three lines into a single expression passed to print:

```
print 3 + 4*2,"\n";
```

Note that the arithmetic expression is processed as 3 + (4*2) rather than (3 + 4)*2. When evaluating numeric expressions, Perl uses the standard rules of precedence. The precedence can be changed by explicitly using parentheses:

```
print (3 + 4)*2,"\n";
```

VARIABLE INTERPOLATION

Another interesting difference between double- and single-quoted strings is what happens when a variable is embedded inside a string. In double-quoted strings, the variable is expanded to its contents, a process known as *string interpolation*. This can aid readability considerably:

```
print "The value of $x is $x\n";
```

Assuming that `$x` again contains "Mary had a little lamb," the above statement outputs

```
The value of Mary had a little lamb is Mary had a little lamb
```

Single-quoted strings do not work in this fashion. If the print statement used a single-quoted string instead, it would print

```
The value of $x is $x\n
```

The user can precisely control whether variable interpolation occurs in double-quoted strings by placing a backslash in front of variables that should not be interpolated.

```
print "The value of \$x is $x\n";
```

In this statement, the first occurrence of the `$x` variable is protected against interpolation because of the backslash, but the second is not.

```
The value of $x is Mary had a little lamb
```

The backslash character can also be used to embed a double-quote character inside a double-quoted string. How is the following different from the outputs shown above? Run it through the Perl interpreter to check your conclusion:

```
print "The value of \$x is \"$x\"\n";
```

Variable interpolation only extends to the contents of the variable itself. Perl will not try to evaluate arithmetic expressions or other programming statements that are embedded in double-quoted strings. For example, the statements

```
$y = 19;
print "The result is $y+3\n";
```

produces the output

```
The result is 19+3
```

To evaluate the expression, put it outside the double quotes. Perl will do the arithmetic, and the print statement will output the result.

```
print "The result is ",$y + 3,"\n";
```

BASIC INPUT AND OUTPUT

Input, in programming parlance, is how data ‘‘get into’’ a script. Output is, of course, what comes out of the script. Most scripts will do both, inputting data from one source and outputting it to another.

The main way of producing output is to use the **print** function. Print takes a list of one or more arguments separated by commas and sends them to the current output device, which by default is the computer’s screen. As already shown, print can deal equally well with text, numbers, and variables:

```
$sidekick = 100;
print "Maxwell Smart's sidekick is ",$sidekick-1,".\n";
print "If she had a twin, her twin might be called"
2*($sidekick-1),".\n";
```

The result of this script is

```
Maxwell Smart's sidekick is 99.
If she had a twin, her twin might be called 198.
```

The main way to read input data is to use the angle bracket operator (<>), which reads a line of input from the current input device. You will usually call this operator in conjunction with the assignment operator to save the returned information into a variable:

```
$line = <>;
print "Got $line";
```

With these two operations, one can now write a fully interactive program named ‘‘dog years,’’ which converts your age in human years to your age in dog years:

```
#!/usr/bin/perl
print "Enter your age: ";
$age = <>;
print "Your age in dog years is ",$age/7,"\n";
```

When this program is run, the result will look like this:

```
% dog_years.pl
Enter your age: 42
Your age in dog years is 6
```

where the 42 is typed in at the keyboard. Another tiny program illustrates one of the idiosyncrasies of the <> operator:

```
#!/usr/bin/perl
print "Enter your name: ";
$name = <>;
print "Hello $name, happy to meet you!\n";
```

Running this program produces output that might not be what is expected:

```
% hello.pl
Enter your name: Lincoln
Hello Lincoln
, happy to meet you!
```

What’s going on? In fact, when the <> operator reads a line of input, the newline character at the end of the typed data is still there! More often than not, the newline character must be removed. Obligingly, Perl provides a function named **chomp** that will do exactly that, removing the terminal newline from a string. The rewritten program looks like this:

```
#!/usr/bin/perl
print "Enter your name: ";
$name = <>;
chomp $name;
print "Hello $name, happy to meet you!\n";
```

With the newline removed, the output looks the way it should:

```
% hello.pl
Enter your name: Lincoln
Hello Lincoln, happy to meet you!
```

The program can still be made a bit shorter by combining the input and chomp statements into a single statement, at the risk of making the program slightly harder to understand:

```
#!/usr/bin/perl
print "Enter your name: ";
chomp($name = <>);
print "Hello $name, happy to meet you!\n";
```

The parentheses control the precedence of the operation so that Perl does the input first and then passes `$name` to the chomp function.

FILEHANDLES

When lines of input are read, the data come from the keyboard by default. When the script writes lines of output, the output goes to the screen by default. What if one wants to change these defaults so that input comes from a file or output goes into one?

There are several ways to do this, but the most straightforward is to use *filehandles*. A filehandle is the connection between a script and a file. Scripts can read from a filehandle to get the contents of the file a line at a time and print to a filehandle to add data to a file.

To open a file for reading, use the **open** function:

```
open MYFILE,'data.txt';
```

The open function expects exactly two arguments. The first is a name for the filehandle (`MYFILE`). This is an arbitrary name that the user chooses, with the convention being to use all uppercase letters. The second argument is the name of the file to open. If just the file name is provided, as shown in the example, Perl will attempt to open a file by that name in the current directory (on Windows and UNIX systems, the directory in which the command to run the script was given; on Macintoshes, the folder that the script is located in). The full path, explicitly giving the location of the file, can also be given.

Unfortunately, the way one specifies a file path is different on the three different operating systems. On UNIX systems, a path begins with a forward slash and each directory is separated by additional slashes. On Windows systems, a path begins with the drive letter (e.g., `C:`) and uses backslashes to separate directories. On Macintoshes, the path begins with the name of the hard disk, and colons separate the name of each successive folder. Examples of fully qualified path names on UNIX, Windows, and Macintosh systems are shown below.

```
UNIX      /usr/local/blast/data/cosmids.txt
Windows   C:\Documents\Blast\Data\Cosmids.txt
Macintosh HD:Sequence Data:Blast:cosmids
```

The open command may not be able to open a file for reading if, for instance, the file does not actually exist. Ways to detect and handle this kind of error will be discussed in the section *Filehandle Errors*.

Once a filehandle is open, it can be read from using the <> operator:

```
$line = <MYFILE>;
chomp $line;
$next_line = <MYFILE>;
chomp $next_line;
```

The only difference between reading from the keyboard and reading from a file is that, instead of using an empty pair of angle brackets (<>), the open filehandle is placed between them (<MYFILE>).

Each time you call <MYFILE>, a new line of data will be read from the file. When the last line has been read, the operation will return the undefined value; detecting this type of error will be discussed later. When a filehandle is no longer needed, it can be closed using the **close** function:

```
close MYFILE;
```

Writing to a file works in much the same way. The main difference is that, when the file is opened, Perl is instructed to write to the file by placing a > sign before the filename:

```
open MYFILE,'>data.txt';
```

If the file does not already exist, Perl will create it and open it for writing. If the file already exists, then Perl will empty out its existing contents before opening it. This ensures that the new data written to the file will replace anything that was already there. To add data to the end of the file without disrupting its current contents, the file can be opened for appending using the >> sign:

```
open MYFILE,'>>data.txt';
```

Data written to the file will now be appended to the end of its current contents rather than writing over the file. If the file does not already exist, then an empty one is automatically created.

Once a filehandle is opened for writing, data can be sent to it, using the filehandle as the print command's first argument:

```
print MYFILE "Your age in dog years is ",$age/7,"\n";
```

This will write the indicated line of text to the file associated with MYFILE. Notice that there is no comma between the filehandle and the list of data arguments! The full, generalized syntax for print is

```
print [FILEHANDLE] $data1 [,$data2 [,$data3 ....]]
```

The square brackets mean an argument is optional. One or more spaces is used to separate the optional filehandle from the first data argument, and commas are used to separate the individual items to be written to the filehandle.

When finished writing to a filehandle, use the close function to `close` it as before. If the program ends without explicitly closing the filehandle, Perl will close the file automatically.

Nothing prevents a script from having multiple filehandles open at the same time. This odd little program will write the first two odd-numbered lines of input to the file `odd.txt` and even-numbered lines to the file `even.txt`:

```
#!/usr/bin/perl
open ODD,">odd.txt";
open EVEN,">even.txt";
$line = <>;
print ODD $line;
$line = <>;
print EVEN $line;
$line = <>
print ODD $line;
$line = <>
print EVEN $line;
close ODD;
close EVEN;
```

There is a certain amount of repetition in this script, for sake of clarity. This script will be revisited shortly, using a more elegant loop to perform the same function.

Each time <> is called, a new line of input is obtained either from the keyboard or the file. What happens when the end of the file is reached? Because there aren't any more lines to read, <> simply returns the undefined value.

MAKING DECISIONS

So far, the programs that have been discussed have been very linear (see Fig. 17.1). The Perl interpreter starts at the first statement and works its way to the last, executing each of them along the way.

Life is full of decisions, however, and so are Perl scripts. Often, there should be different paths followed if a particular condition is true, another if the condition is false. Considering the dog years calculator shown above, what would happen if the user had entered a negative number for the age or a number that's unreasonably large? It would be desirable to reject the input outright, rather than produce and output a preposterous answer. This simple modification to the original script achieves this goal.

```
#!/usr/bin/perl
print "Enter your age: ";
$age = <>;
die "Preposterous age" if $age <= 0 or $age >= 100;
print "Your age in dog years is ",$age/7,"\n";
```

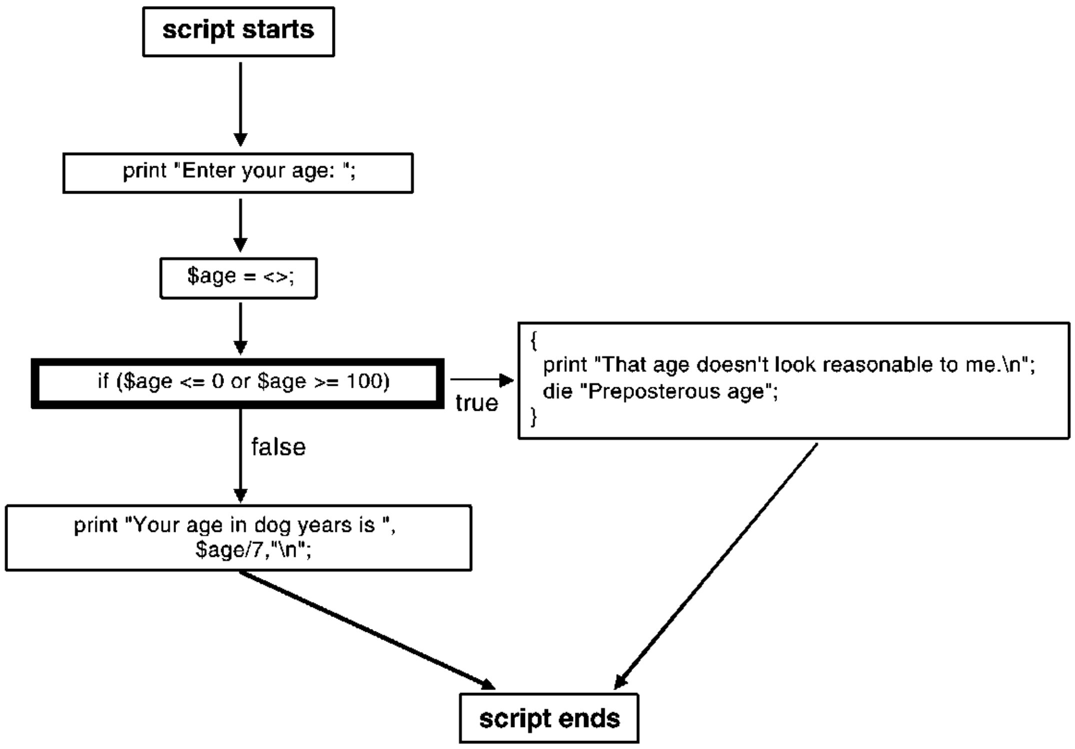
script starts
print "Enter your age: ";
$age = <>;
if ($age <= 0 or $age >= 100)
true
false
{
print "That age doesn't look reasonable to me.\n";
die "Preposterous age";
}
print "Your age in dog years is ",
$age/7,"\n";
script ends

Figure 17.1. The "dog years" calculator provides an example of entirely linear program flow.

The fourth line is the new one, consisting of two parts. The first part (`die "Preposterous age"`) introduces the **die** function, which is a lethal form of the `print` command. It prints out the error message given to it, along with information indicating the current line number. It then immediately terminates the program. It is great for handling those occasions when a fatal, unrecoverable error has occurred. The second part of the line (`if $age <= 0 or $age >= 100`) tests the `$age` variable's numeric value. If `$age` is either less than or equal to zero or greater than or equal to 100, then the test is true and the die function is executed. If `$age` satisfies neither test, then the `die` function is skipped and the program goes on as before. In plain English, the statement can be read as, "Die if the age is either less than or equal to zero or the age is greater than or equal to 100." If the age is invalid, the program will terminate at the die statement, and the last print statement will never be executed. Testing the potential scenarios of the now-modified program, we find

```
% dog_years.pl
Enter your age: -20
Preposterous age at dog_years.pl line 4, <> chunk 1.
% dog_years.pl
Enter your age: as old as Methuselah
Preposterous age at dog_years.pl line 4, <> chunk 1.
% dog_years.pl
Enter your age: 42
Your age in dog years is 6
```

TABLE 17.1. Numeric Comparison Operators

Operator	Description	Example
==	Equality	`$a == $b`
!=	Not equal	`$a != $b`
<	Less than	`$a < $b`
>	Greater than	`$a > $b`
<=	Less than or equal to	`$a <= $b`
>=	Greater than or equal to	`$a >= $b`
!	Logical not	`$ = !$b`

Perl has a complete set of comparison operators that work on numbers and strings. To compare numbers, use any of the operators shown in Table 17.1. To compare strings, use any of the operators shown in Table 17.2. The numeric comparison operators are straightforward because they look, for the most part, like conventional expressions used in algebra. The big trap is the == operator that is used to test the equality of two numbers. Two equal signs are used instead of just one.

If you forget and accidentally use the assignment operator (=), then you will not get the result you expect:

```
$a == $b; # compare $a to $b, return true if equal
$a = $b;  # assign contents of $b to $a
```

Read the ! operator as *not*. Unlike the other operators, it takes a single argument and reverses its truth. True expressions become false and vice versa. For example

```
print "the number is not greater than 0" if !($a > 0);
```

This statement first compares the current value of `$a` to zero and returns a true value if `$a` is greater than zero. The `!` operator then reverses the test so that the expression as a whole is true only if `$a` is less than or equal to zero.

The string comparison operators are funny two-letter commands. The one used most frequently is `eq`, for testing whether two strings are the same. Consider this new version of the `hello.pl` script.

TABLE 17.2. String Comparison Operators

Operator	Description	Example
`eq`	Equality	`$a eq $b`
`ne`	Not equal	`$a ne $b`
`lt`	Less than	`$a lt $b`
`gt`	Greater than	`$a gt $b`
`le`	Less than or equal to	`$a le $b`
`ge`	Greater than or equal to	`$a ge $b`
`=~`	Pattern match	`$a =~ /gattc/`

```
#!/usr/bin/perl
print "Enter your name: ";
chomp($name = <>);
print "Hello $name, happy to meet you!\n";
print "Hail great leader!\n" if $name eq 'Lincoln';
```

The output of the program uses `eq` to give Lincoln a special greeting:

```
% hello.pl
Enter your name: George
Hello George, happy to meet you!
% hello.pl
Enter your name: Lincoln
Hello Lincoln, happy to meet you!
Hail great leader!
```

Other handy string comparison operators are `lt`, which is true if the first string is less than the second string, and `gt` if the first is greater than the second. Perl compares strings alphabetically but uses criteria different from the telephone book. Among other subtle (and not-so-subtle) differences, the set of uppercase letters is less than the set of lowercase letters; in Perl, `Z` is less than `a`. Be careful not to use == to compare strings or `eq` to compare numbers. The handiest string comparison operator of them all is =~, the pattern matching operator. This is the most powerful of Perl operators, deserving its own discussion later in this chapter.

Two or more comparison operations can be combined using the operators `and` and `or`. An expression involving `and` is true only if *both* the right and left sides are true, whereas `or` expressions are true if *either* side is true. There is also a `not` operator, which reverses the sense of whatever comes to the right, making true expressions false and false ones true, just like Big Brother did in George Orwell's *1984*.

The use of the `if` statement has already been demonstrated, executing a statement only when the condition that follows is true. As always with Perl, the opposite operator exists, called `unless`, executing a statement only when the condition following is false. Returning to the program testing for preposterous ages, the test could be rewritten this way:

```
die "Preposterous age" unless 0 < $age and $age < 100;
```

Read the statement this way: "Die unless the age is greater than zero and less than 100." The effect is the same. Sometimes it will seem more natural to write the conditional with an `if` sometimes with an `unless`.

CONDITIONAL BLOCKS

How would one approach executing several statements conditionally? In this case, the statements can be grouped into a *block*, using curly braces. The grouped statements can then be executed altogether, using the block form of `if`. Returning again to the dog-age calculator

```
#!/usr/bin/perl
print "Enter your age: ";
$age = <>;
if ($age <= 0 or $age >= 100) {
  print "That age doesn't look reasonable to me.\n";
  die "Preposterous age";
}
print "Your age in dog years is ",$age/7,"\n"';
```

In this example, `if` controls a block of two statements surrounded by the curly braces. The first statement prints out a warning, and the second one terminates the program with die. If the age does not fall within the range of 0–100, then the statements within the block *will* execute and the program will end before reaching the last line of code. If the age does fall within the specified range, the two statements in the `if` block are ignored and the program goes on to print out the calculated result, as before. The effect is to create two alternative paths in the program, one of which leads to termination with an error statement (Fig. 17.2).

`If` blocks have this general form:

```
if (TEST) {
                          STATEMENT 1;
                          STATEMENT 2;
                          STATEMENT 3;
                               ...
}
```

The test itself must be enclosed by parentheses in the manner shown, but any comparison operation involving numbers or strings is allowed. The indentation is a matter of style—although Perl does not depend on the indentation to interpret the

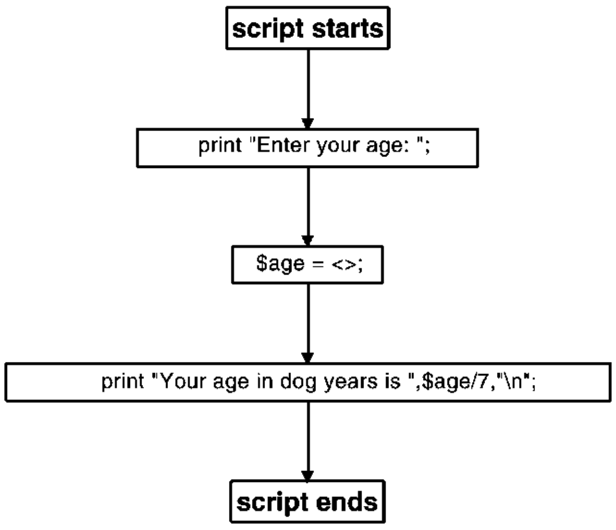
script starts
print "Enter your age: ";
$age = <>;
print "Your age in dog years is ",$age/7,"\n";
script ends

Figure 17.2. If blocks can change the flow of execution.

code, indenting code in this fashion is invaluable when debugging a program, as it is easier to see where a given block of code begins and ends.

Perl can also handle situations in which one set of statements should be executed if the test condition is true, and another set of statements if the test condition is false. To do so, an `else` block is added to the if structure, changing the construct from an *if* statement to an *if—else* statement.

```
#!/usr/bin/perl
print "Enter your age: ";
$age = <>;
if ($age <= 0 or $age >= 100) {
  print "That age doesn't look reasonable to me.\n";
} else {
  print "Your age in dog years is ",$age/7,"\n";
}
```

The if-else statement shown above has two blocks, each surrounded by curly braces. The contents of the first block is executed when the test is true. Otherwise the second block is executed. The result is that, if the entered age falls outside the acceptable range for the program, an error message is printed; if not, it prints the calculated results instead.

```
% dog_years.pl
Enter your age: eighteen
That age doesn't look reasonable to me.
% dog_years.pl
Enter your age: 21
Your age in dog years is 3
```

There happens to be a single statement in each block in this example, but there is no limit to the number of statements that can be enclosed within a block. Finally, for sake of completeness, code that specifies what is wrong with bad input data can be included in the program. A handy `elsif` block can test one string after another.

```
#!/usr/bin/perl
print "Enter your age: ";
$age = <>;
if ($age <= 0) {
  print "You are way too young to be using a computer.\n";
} elsif ($age >= 100) {
  print "Not in a dog's life!\n";
} else {
  print "Your age in dog years is ",$age/7,"\n";
}
```

There are now two `$age` checks in this program. The first test compares the age with zero and prints out a warning message if it is less than or equal to zero. If the test is false, then the program proceeds to the `elsif` block and tries the second test, which compares `$age` with 100. If *this* test is true, then the second error message is printed out. Otherwise, if both tests are false, the program falls through to the `else` block. The possible outcomes are now as follows:

```
% dog_years.pl
Enter your age: -20
You are way too young to be using a computer.
% dog_years.pl
Enter your age: 999
Not in a dog's life!
% dog_years.pl
Enter your age: 28
Your age in dog years is 4
```

WHAT IS TRUTH?

The words *true* and *false* have been tossed around rather blithely in this chapter. It is now appropriate to define these terms more pricisely. Regardless of the meaning of truth in the broader, philosophical sense, truth in Perl boils down to four very simple rules:

Zero (0) is *false*.
The empty string (" ") is *false*.
The undefined value is *false*.
Everything else is *true*.

The various numeric and string comparison tests that were illustrated in the previous section evaluate to 1 when true and to the undefined value when false.

LOOPS

Conditional statements allow the flow of a program to be modified so that sections of code can be executed or skipped over as needed. They cannot, however, make the program execute a particular section of code more than once. For this, Perl (and most programming languages) utilizes what are known as *loops*. Perl has a lot of different types of loops available to the user, but the most useful one is the **while** loop. The while loop looks very much like an **if** block, but instead of executing the contents of the block once if the test is true, it executes the statements repeatedly so long as the test is true. An example of this type of loop is illustrated by this simple counting program:

```
#!/usr/bin/perl
$count = 1;
while ($count <= 5) {
  print "$count potato\n";
  $count = $count + 1;
}
```

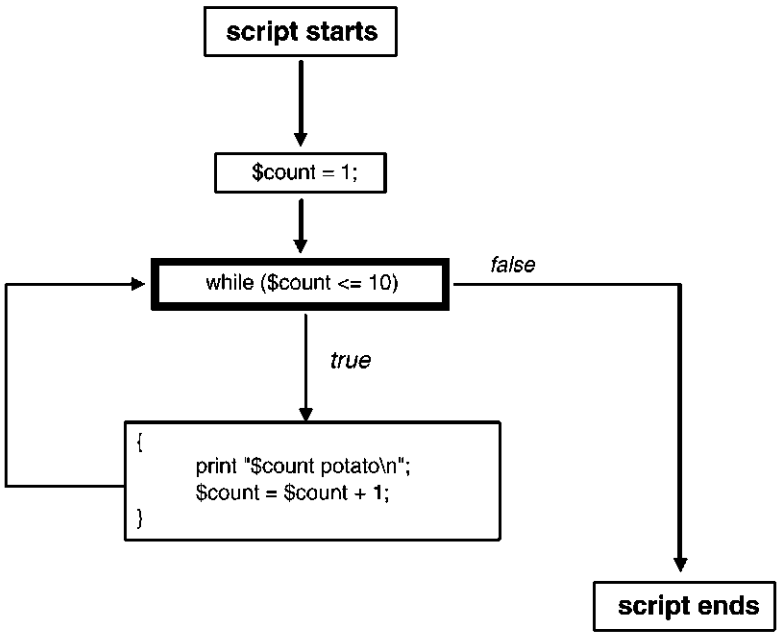

Figure 17.3. While loops execute the same block repeatedly until a defined condition is satisfied.

Before the loop begins, the program creates a variable named `$count` and sets its value to 1. The `while` loop test checks whether `$count` is less than or equal to 10 and executes the two statements contained in the curly braces as long as this condition holds true. The first statement prints out the current value of `$count`, and the second increments the variable by 1. The first time the while statement is encountered, `$count` was set to 1, so the block is executed. The second time, `$count` is 2, and the block is executed again. This continues until `$count` is 6, at which point the test is no longer true (because `$count` is greater than 5) and the loop terminates. The output for the program is quite simply

```
% count.pl
1 potato
2 potato
3 potato
4 potato
5 potato
```

In the same way that `unless` reverses the sense of the `if` statement, `until` can be used as an alternative to `while`. `Until` loops will execute the contents of a block until a certain test becomes true. The counting program could then be rewritten as follows, to yield the same output:

```
#!/usr/bin/perl
$count = 1;
```

```
until ($count > 5) {
  print "$count potato\n";
  $count = $count + 1;
}
```

COMBINING LOOPS WITH INPUT

Loops become very powerful when combined with input statements. Consider this simple example:

```
#!/usr/bin/perl
print "Type something> ";
while ( defined($line = <> ) ) {
  chomp $line;
  print "You typed '$line'\n\n";
  print "Type something> ";
  }
```

This script prints out the prompt `Type something>` and immediately enters a `while` loop. The `while` loop here is a little different from the ones shown before. Instead of doing a comparison, the while loop's test contains the expression `defined($line = <>)`. This expression looks a bit bizarre but can be explained very simply. The first thing that happens is that a line of input is read and assigned to the variable `$line`. Then, `$line` is then passed to a new function named `defined`, which returns true if the contents of a variable are defined, false otherwise. Recall that <> reads lines from a file (or keyboard) until it reaches the end of the file, at which point it returns undefined. This test is telling the while loop to read input lines one at a time until the end of the file is reached. The parentheses ensure that defined is called after the line is read into `$line`, not before.

For each line read in this way, the while loop executes three statements. The first statement calls chomp to remove the newline character from the end of `$line`. The second statement uses variable interpolation to insert the input line into the string `"You typed '$line'\n\n"` and then prints the resulting string out. The last statement displays the prompt again. The result looks like this:

```
% echo.pl
Type something> hi there
You typed 'hi there'
Type something> this is something
You typed 'this is something'
Type something> ^D
%
```

As you can see, this program echoes back everything typed in, like a particularly annoying child. Even if nothing were typed in, the program would simply echo back an empty string; this is because the script would read a single newline character, which it would then remove with chomp, yielding an empty string. To stop the program, an *end-of-file* character has to be sent. On UNIX and Macintosh systems, this is done by typing control-D (written ^D for short). On Microsoft Windows systems, this is done by typing control-Z (^Z). An alternative is to kill the program by typing control-C (^C). This stops the program dead in its tracks.

Because it can be inconvenient to remember obscure control characters, the program can be made a bit friendlier by allowing the user to type `quit` to exit the loop:

```
#!/usr/bin/perl
print "Type something. 'quit' to finish> ";
while ( defined($line = <>) ) {
  chomp $line;
  last if $line eq 'quit';
  print "You typed '$line'\n\n";
  print "Type something> ";
  }
print "goodbye!\n";
```

The main change is the line that immediately follows the `chomp`. This introduces a new function named `last`, which acts as a loop modifier. It is only allowed to occur within the body of a loop. When executed, `last` causes the script to exit the loop immediately, even though the loop test may still be true. This statement compares the contents of `$line` to the string `quit`. If they match, the `last` function is executed and the loop finishes. Running the program now, entering the word **quit** exits the loop so that the last line (which prints "goodbye!") can be executed.

Because the process of reading and processing incoming data a line at a time is so common, Perl provides a handy shortcut. If <> appears all alone in the test part of a `while` statement, Perl will read a line into an automatic variable with the odd-looking name `$_` (which is read as "dollar sign underscore") and then call `defined` on your behalf. Furthermore, many text-processing functions operate on `$_` by default, including `chomp`. Taking advantage of this shortcut, one can rewrite the previous example in this way:

```
#!/usr/bin/perl
print "Type something. 'quit' to finish> ";
while (<>) {
  chomp;
  last if $_ eq 'quit';
  print "You typed '$_'\n\n";
  print "Type something> ";
  }
print "goodbye!\n";
```

STANDARD INPUT AND OUTPUT

When a Perl script reads a line from <> and prints without using a specified filehandle, it instead uses two automatic filehandles called `STDIN` and `STDOUT`. The statements

```
print "The answer to life is...";
print STDOUT "The answer to life is...";
```

are exactly equivalent. In the same vein, the statements `$line = <>;` and `$line = <STDIN>;` are almost equivalent. The subtle differences between `<>` and `<STDIN>` will be discussed below.

The names `STDIN` and `STDOUT` are derived from *standard input* and *standard output*, an idea popularized by the UNIX operating system. Standard input and standard output are abstract files from which a script can accept input and send output, respectively. When a script is first launched, standard input corresponds to the keyboard and standard output corresponds to the computer screen. On Windows and UNIX systems, standard output appears in the command interpreter window. On Macs, the output appears in a small scrolling window that MacPerl creates specifically for this purpose. When a Perl script is launched, the user has the option of changing where standard input and output come from and go to. The user can also arrange for the standard output of one script to be sent to the standard input of another script in assembly-line fashion. This is actually a very powerful facility, but one that is beyond the scope of this chapter.

To redirect standard *input* from a file in either Microsoft Windows or UNIX systems, use the less-than (<) symbol to indicate the file:

```
% count_words.pl <C:\My Documents\cosmids\11_22_00cosmids.txt
```

To redirect standard *output* to a file on Windows or UNIX systems, use the greater-than (>) symbol:

```
% reverse_translate.pl > dna.txt
```

Standard input and output can be redirected simultaneously by using both symbols on the same command line:

```
% reverse_translate.pl < protein.txt > dna.txt
```

There is actually a third automatic filehandle called `STDERR`, for *standard error*. Perl sends its error messages and other diagnostics to standard error rather than standard output. When die is used to display error messages, the messages go to the `STDERR` filehandle automatically. `STDERR` is initially attached to the screen, just like `STDOUT`. On UNIX systems, `STDERR` can be redirected using `>&`. The following command will send the standard output of reverse_translate.pl to a file named `dna.txt`, and send any warnings or errors to a file named `errors.out`.

```
% reverse_translate.pl > dna.txt >& errors.out
```

Perl in Windows and Macintosh systems do not use the concept of a separate, standard error, so this type of redirection will not work on those systems.

Returning to the idea that `<STDIN>` and `<>` are almost—but not exactly—the same, the difference lies in the fact that `<>` contains some additional magic that makes it easy to process command line arguments. Given a file named unsorted.txt that is to be processed using a file named odd_even.pl, one way to process this file would be to redirect standard input, like so:

```
% odd_even.pl < unsorted.txt
```

However, because odd_even.pl contains a line using <> operator to read from standard input, the command can also be written in the following way:

```
% odd_even.pl unsorted.txt
```

The file name unsorted.txt is now being passed to the script as an *argument* and not as standard input. When <> is used, it looks for any filenames on the command line, opens them, and reads from them a line at a time. If no files are mentioned on the command line, <> reads from standard input. This feature can be used to process multiple files at once as well, reading from each of the files mentioned on the command line as is they were all a single, continuous file. Macintosh users, who do not have a command line available to them, can still take advantage of this <> feature. When a Perl script is saved as a droplet, text files can be dragged and dropped on top of its icon. Each of the dropped text files will be passed to <> for reading.

FINDING THE LENGTH OF A SEQUENCE FILE

Moving to a biologically based example, consider a text file containing a large DNA sequence in single-letter format. The sequence is of unknown length, and it would be desirable to quickly determine the number of bases in the sequence. Using the file size alone would not be appropriate, since the presence of end-of-line characters at the end of each line would artificially inflate the number. The Perl script that will be developed in this section will answer this question as well as count up the number of lines in the original text file. This short script will read in the file, one line at a time, removing each line's terminal newline character and determining the length of what is left. The length for that line is added to a running total, with a counter tracking the number of lines in the file.

```
#!/usr/bin/perl
# file_size.pl
$length = 0;  # set length counter to zero
$lines = 0;   # set number of lines to zero
while (<>) {  # read file one line at a time
  chomp;  # remove terminal newline
   $length = $length +  length $_;
  $lines = $lines + 1;
}
print "LENGTH = $length\n";
print "LINES = $lines\n";
```

A built-in function named `length` is invoked to determine the length of the line once the terminal newline is removed. When the script is done, it prints out the ultimate values of `$length` and `$lines`:

```
% file_size.pl dna.txt
LENGTH = 50649
LINES = 1387
```

PATTERN MATCHING

The script from the previous section was useful only for calculating DNA lengths in the unusual case of having a file that contains nothing but raw DNA sequence. More frequently, however, sequence data come in FASTA format. To tally up the length of all the sequences in a FASTA file, the lines that begin with > must be ignored. Perl's pattern-matching operations make this easy to do.

A pattern match is a special type of text comparison. It is something like `eq`, but, instead of testing for an exact match between two strings, it tests a string against a pattern, using a pattern description language known as a *regular expression*. A simple example of a pattern match would be

```
print "EcoRI site found!" if $dna =~/GAATTC/;
```

This if statement compares the contents of the variable `$dna` against the pattern `GAATTC`. The funny-looking =~ symbol is the pattern match comparison operator; think of it as an ''approximately equal'' comparison. If the string to the left of the pattern match operator contains the indicated pattern, it will return true. In the script fragment above, the program will print out `EcoRI site found`! if the string contains `GAATTC` anywhere along its length.

Regular expression patterns are delimited by forward slashes. The simplest ones contain a sequence of normal characters that must match somewhere within the body of a string. The *EcoR* I-site detector is one such example. Regular expressions are much more powerful than this, however. For example, square brackets can be used to specify a set of alternative characters in the manner shown here:

```
$dna =~/GGG[GATC]CCC/
```

This pattern matches a sequence of characters beginning with GGG, followed by any of the characters G, A, T, or C, followed by the sequence CCC. In other words, this pattern searches for GGGNCCC.

To search for a series of alternative patterns, you can use the | symbol to separate the alternatives. For example, this will search for either *EcoR* I sites or *Hind* III sites:

```
$dna =~/GAATTC|AAGCTT/;
```

This facility is greatly enhanced by *metacharacters* and *quantifiers*. A metacharacter represents a whole class of characters. For example, a single dot (.) will match any character except the end of a line, whereas `\d` signifies any digit. There

TABLE 17.3. Regular Expression Metacharacters

Metacharacter	Description
`.`	Any character except newline
`^`	The beginning of a line
`$`	The end of a line
`\w`	Any word character (non-punctuation, non-white space)
`\W`	Any non-word character
`\s`	White space (spaces, tabs, carriage returns)
`\S`	Non-white space
`\d`	Any digit
`\D`	Any non-digit

are also metacharacters that will match the beginning and ending of lines and match the boundaries between one word and the next. Table 17.3 lists some of the more common metacharacters. Notice that there are many cases in which a metacharacter representing a character set is paired with its complement. For example, `\s` matches white space, and `\S` matches nonwhite space (in other words, printing characters). Another frequently used pair, ^ and `$`, match the beginning and end of lines, respectively.

For example, to match a “ZIP+4” format ZIP code, the required regular expression would be written as follows:

```
$address =~ /\d\d\d\d\d-\d\d\d\d/;
```

For a regular expression to match a metacharacter *literally*, it must be preceded by a backslash. For example, to match DNA sequence IDs of the form “M58200.2,” where a dot is used literally, the regular expression should be written

```
$sequence_id =~ /\w+\.\d+/;
```

By default, any character or metacharacter in a regular expression matches exactly once. By placing a quantifier after the character, Perl can match a character a specific number of times or a range of times. The simplest type of quantifier is `{M}`, which tells Perl to match the pattern exactly M times. Using this notation, the ZIP+4 regular expression could be rewritten as

```
$address =~ /\d{5}-\d{4}/;
```

Similar quantifiers include the form `{M,N}`, which will match at least *M* times but no more than *N*; `{M,}`, which matches at least *M* times; and `{,N}`, which will match no more than *N* times. To bring the examples back into a biological context, the following expression will match the plant mcrBC methylation site Pu-C-X(40–80)-Pu-C, in which the center of the recognition site can be anywhere from 40 to 80 nucleotides long.

```
$sequence =~ /[AG]C[GATC]{40,80}[AG]C/;
```

Although the curly braces can be used to describe any quantification, there are some short-cut metacharacters that are used for the frequent cases (Table 17.4).

Parentheses can be used to group parts of a regular expression, and then a quantifier can be applied to the entire group. For example, this regular expression will match normal five-digit ZIP codes as well as the ZIP + 4 form:

```
$address =~ /\d{5}(-\d{4})?/;
```

The `\d{5}` part matches a digit repeated exactly five times. This is followed by an optional section containing `-\d{4}`, a hyphen followed by four digits. The optional section is completely surrounded by parentheses to group it, and the group is followed by a `?` symbol, meaning that it can match at most once and possibly not at all. Parenthesized groups can also be used to extract portions of regular expressions.

With regular expressions, the DNA length calculator can be rewritten so that it correctly ignores lines that begin with the > sign. The modified program looks like this:

```
#!/usr/bin/perl
# file_size2.pl
$length = 0;
$lines = 0;
while (<>) {
chomp;
$length = $length + length $_ if $_ =~ /^[GATCNgatcn]+$/;
$lines = $lines + 1;
}
print "LENGTH = $length\n";
print "LINES = $lines\n";
```

The key modification to the original script is the insertion of a conditional test on the statement that tallies the length of each line of text. The script tests `$_` for a pattern match with the regular expression `/^[GATCNgatcn]+$/`, which matches lines containing DNA sequence. The initial ^ character matches the beginning of the line, `[GATCNgatcn]+` matches one or more of the characters GATCN or their lower-case equivalents, and the $ matches the end of the line. Other lines in the FASTA file, including blank lines and the description lines, are ignored. The file length that is printed out corresponds now to the sequence only.

TABLE 17.4. Refular Expression Quantifiers

Quantifier	Description
?	0 or 1 occurrence
+	1 or more occurrences
•	0 or more occurrences
{N,M}	Between N and M occurrences
{N, }	At least N occurrences
{ ,M}	No more than M ocurrences

Before leaving this script, some syntactical tricks can be applied to make the script more concise. If a regular expression appears alone without a =~ operator, then Perl assumes that the variable to be tested for a pattern match is $_. The string length function also behaves this way, returning the length of $_ if no variable is explicitly specified. So, the length-tallying line can be rewritten as

```
$length = $length +  length if /^[GATCNgatcn]+$/;
```

A second shortcut is to append an `i` flag at the end of the regular expression right after the second slash. This puts the regular expression into *case-insensitive* mode, and allows the statement to be written as

```
$length = $length + length if /^[GATCN]+$/i;
```

Finally, because adding a value to a variable and storing the sum back into the same variable is such a common operation, Perl provides the shortcut operator += (read as "plus equals"). This operator takes a numeric value on its right side, adds it to the contents of the variable on its left side, and stores the result back into the same variable, all in one graceful step. Taking advantage of this feature gives statements of the form

```
$length + = length if /^[GATCN]+ $/i;
```

Similar assignment shortcuts are summarized in Table 17.5. Putting all these shortcuts together gives the final version of the DNA length-tallying script:

```
#!/usr/bin/perl
# file_size3.pl
$length = 0;
$lines = 0;
while (<>) {
  chomp;
  $length + = length if /^[GATCN]+ $/i;
  $lines + = 1;
}
print "LENGTH = $length\n";
print "LINES = $lines\n";
```

TABLE 17.5. Assignment Shortcut Operators

Operator	Example	Description
+=	$a += 3	Add a number to a variable
-=	$a -= 3	Subtract a number from a variable
*=	$a *= 10	Multiply variable by a number
/=	$a /= 2	Divide a variable by a number
.=	$txt .= "abc"	Append a string to a variable

EXTRACTING PATTERNS

Not only are regular expressions good for detecting patterns in text, but they can be used to extract matching portions of the text as well. To see how this might be useful, consider a FASTA description line like this one:

```
>M18580 Clone 305A4, complete sequence
```

In addition to the initial >, the description line contains two different fields that we might like to capture. The first is ‘‘M18580,’’ the mandatory sequence ID. The second is ‘‘Clone 305A4, complete sequence,’’ an optional human-readable comment. In regular expression terms, the description line looks like this:

```
/^>\S+ \s*.*$/
```

Reading from left to right is the beginning of the line (^), a > sign, one or more non-white-space characters corresponding to the sequence ID, zero or more spaces or other white space, zero or more of any character corresponding to the optional description, and the end of the line ($). This regular expression match can be made to extract the ID and description lines simply by putting parentheses around the parts that should be captured:

```
/^>(\S+)\s*(.*)$/;
$id = $1;
$description = $2;
```

When a string successfully matches a regular expression, any portions of the expression that are contained within parentheses are extracted and placed into the automatic variables `$1, $2, $3`, and so forth. The extraction works from left to right and only happens if the entire regular expression matches.

This trick can be used to fix a deficiency in the DNA length calculator from the previous section. Previous versions of this script naively treated the entire FASTA file as a single DNA sequence. However, a FASTA file usually contains entries for multiple sequences. With pattern matching, the description lines can be identified and the sequence IDs extracted, allowing for the length of each sequence to be printed out separately.

```
#!/usr/bin/perl
$id = '';                  # holds sequence ID of current sequence
$length = 0;               # holds length of current sequence
$total_length = 0;         # tallies aggregate length of all seqs
while (<>) {
  chomp;
  if (/^>(\S+)$/) { # found a new description line
    print "$id: $length\n" if $length > 0;
    $id = $1;
    $length = 0;
  } else {
    $length += length;
```

```
      $total_length + = length;
    }
}
print "$id: $length\n" if $length > 0; # last entry
print "TOTAL LENGTH = $total_length\n";
```

After initializing the three variables at the top of the program, the script enters a while loop. As before, it reads a line at a time into `$_` and removes the terminating newline. The new feature is an if-else block. The block performs a pattern match on the line, looking for a FASTA description line. If one is found, it signals the beginning of a new sequence. The following then occurs:

- If `$length` is non-zero, the ID and length of the previous sequence are printed. The check on `$length` prevents the program from printing out an empty line when it hits the very first sequence in the file.
- The matched sequence ID is copied into `$id`.
- `$length` is set to zero.

Otherwise, the program is in the middle of a sequence, in which case the length of the current line is added to the appropriate counters. After the last line is read, the ID and length of the last sequence in the file are printed, as well as the total length of all the sequences:

```
% fasta_length.pl ests.fasta
D28205: 1105
BCD207F: 402
BCD207R: 332
BCD386F: 192
BCD386R: 362
CDO98F: 374
TOTAL LENGTH = 2767
```

ARRAYS

Previous examples have worked with single-valued *scalar* variables only. However, Perl has the ability to work with multivalued variables as well. There are two basic multivalued variables, named *arrays* and *hashes*. An array is a list of data values indexed by number. A hash is a list of data values indexed by string. Both are very easy to use and incredibly handy. To understand arrays, consider how one might keep track of a large number of identifiers, such as clone names. With scalar variables, one approach could be to assign each clone name to a different variable:

```
$clone1 = '192a8';
$clone2 = '18c10';
$clone3 = '327h1';
...
```

The problem with this approach, besides being tedious, is that it does not offer any way to step through the entire list of clones one by one, performing the same operation on each one. Arrays circumvent this problem. An array can be defined as follows:

```
@clones = ('192a8','18c10','327h1','201e4');
```

This new array, named `@clones`, contains four strings. Array variables begin with an `@` sign to distinguish them from scalar variables, which begin with a `$`. Scalars and array variables are completely separate. In fact, a scalar variable can be named `$clones` and an array variable can be named `@clones` within the same program. They will not interact in any way. As alluded to above, operations can be applied to arrays as a whole. For example, the = operator can be used to copy one array into another:

```
@old_clones = @clones;
```

Items can be added to the end of an array using the **push** function:

```
push @clones,'281e3';
```

After this statement executes, `@clones` will contain five items. The opposite of push is **pop**, which removes the last item from the array, reducing it in size by one, and returns the removed item as its result. This statement will reduce `@clones` back to a length of four , assigning `281e3` to the scalar variable `$last_clone`:

```
$last_clone = pop @clones;
```

Two array operations are particularly common: accessing an arbitrary array element by its positions in the array using *indexing* and looping over each element of the array in order using the **foreach** loop. Considering indexing first, to copy the third element of `@clones` into a variable named `$third_clone`, the statement would be written as

```
$third_clone = $clones[2];
```

This will—and should—look strange at first. The numeral in the square brackets, `[2]`, is the *index*. Perl numbers its arrays starting with *zero*, so the first item is actually index 0, the second item is index 1, and the third item is index 2. Any expression can be placed within the square brackets, as long as that expression ultimately evaluates to an integer. As an example, if a scalar variable `$i` contains a number, to address the next element in the series, it could be referred to as `$clones[$i+1]`. More mystifying, however, is the `$` at the beginning of the array name. What happened to `@clones`?

When one indexes into an array, the symbol at the front refers to the individual array element, *not* the array as a whole. Because the element itself is a scalar, the symbol at the front should be a `$`. To clarify, look at the following two examples.

```
@old_clones = @clones;
$first_clone = $clones[0];
```

The first line would copy the *entire* array, whereas the second line copies just a single element within the array.

Arrays can be extremely long; ones with thousands of elements are not unusual. A common operation is to loop through each member of an array and do something with it. The `foreach` loop makes this possible. For instance, say that the array `@dna` contains a list of DNA sequences and that a printout of the length of each element would be helpful. The following small loop would accomplish this.

```
foreach $dna (@dna) {
  print length $dna,"\n";
}
```

The `foreach` loop has three parts: the name of a scalar variable known as the *loop variable*, an array name enclosed in parentheses, and a block containing the statements to be executed. `Foreach` steps through the array one element at a time, placing each element in the loop variable and executing the statements within the block. The statements may examine the contents of the loop variable and act on it or even change the loop variable to change the corresponding array element. After the loop is finished, the loop variable will again contain whatever it had before the loop began or be undefined if this is the only time it was used.

To illustrate how assigning to the loop variable changes the contents of the array, below is a fragment of code that will treat every element of an array as a DNA sequence, replacing it with its reverse complement:

```
foreach $dna (@dna) {
  $dna = reverse $dna; # reverse it
  $dna =~ tr/gatcGATC/ctagCTAG/; # complement it
}
```

The two statements in the block show off a pair of Perl functions that have not yet been discussed. The first of these, **reverse**, returns the reverse of a scalar variable, turning GGGGTTTT into TTTTGGGG. The reversed sequence is assigned back into the loop variable. The next statement uses the **tr** function (for *translate*) to substitute one set of characters with another. `tr` has an unusual syntax that is rooted in Perl's historical origins. It uses the slash as a delimiter, replacing the list of characters between the first set of delimiters with the characters in the second set. The replacement occurs on whatever variable `tr` is bound to using the =~ operator (the syntax should be reminiscent of pattern matching). Characters not mentioned in the list are left unchanged. In the example above, the list gatcGATC is replaced with ctagCTAG. What happens is that "g" is replaced with "c," "a" with "t", and so forth. Thus TTTTGGGG becomes AAAACCCC, which is the reverse complement of the original element, GGGGTTTT.

ARRAYS AND LISTS

Perl lists are closely related to arrays. Lists are a set of constants or variables enclosed in parentheses. An example of a list of strings would be (`"one", "two", "buckle my shoe"`), whereas an example of a list of variables would be (`$a, $b, $c`). A list that combines variables, constant strings, and constant numbers might be something like (`$a, "the Roman empire", 3.1415926, $ipath`). Lists can be thought of as being related to array variables in the same way that the constant 123.4 might be related to the scalar variable `$total`.

Lists are useful for performing operations in parallel. For example, lists can be assigned to array variables to make the array identical to the list. In fact, one example of this was shown earlier:

```
@clones = ('192a8','18c10','327h1','201e4');
```

Arrays can be assigned to lists, provided that each element of the list is a variable and not a constant. For example, to extract the first three elements of an array, one could write

```
($first,$second,$third) = @clones;
```

After this operation, `$first` will contain `18c10`, and so on. Naturally enough, lists can be assigned to lists as well, again provided that the list on the left contains variables only:

```
($one,$two,$three) = (1,2,3);
```

SPLIT AND JOIN

It is often very useful to transform strings into arrays and to join the elements of arrays together into strings. The **split** and **join** functions allow for these operations. Split takes two arguments: a delimiter and a string. It splits the string at delimiter boundaries, returning an array consisting of the split elements. The delimiters themselves are discarded. To illustrate this, consider a case requiring the manipulation of a long file containing comma-delimited files, such as the following:

```
192a8,The Sanger Centre,GGGTTCCGATTTCCAA,CCTTAGGCCAAATTAAGGCC
```

Split makes it easy to convert the long string into a more manageable array. To split on the comma, the comma is used as the delimiter:

```
chomp($line = <>);          # read the line into $line
@fields = split ',',$line;
```

`@fields` will now contain the five individual elements, which can now be indexed or looped over. Split is often used with a list on the left side instead of an array, allowing one to go directly to assigning to the list. For example, rather than creating an array named `@fields`, the result of a split command can assign values to a list of named, scalar variables:

```
($clone,$laboratory,$left_oligo,$right_oligo) = split ',',$line;
```

The `join` function has exactly the opposite effect of split, taking a delimiter and an array (or list) and returning a scalar containing each of the elements joined together by the delimiter. Thus, continuing the earlier example, the `@fields` array can be turned into a tab-delimited string by joining on the tab character (whose escape symbol is `\t`):

```
$tab_line = join "\t",@fields;
```

After this operation, `$tab_line` will look like this:

```
192a8 The Sanger Centre GGGTTCCGATTTCCAACCTTAGGCCAAATTAAGGCC
```

HASHES

The last Perl data type that will be considered is the **hash**. Hashes are similar to arrays in many respects. They hold multiple values, they can be indexed, and they can be looped over, one element at a time. What distinguishes hashes from arrays is that the elements of a hash are unordered, and the indexes are not numbers but strings. A few examples will clarify this:

```
%oligos = ();
$oligos{'192a8'} = 'GGGTTCCGATTTCCAA';
$oligos{'18c10'} = 'CTCTCTCTAGAGAGAGCCCC';
$oligos{'327h1'} = 'GGACCTAACCTATTGGC';
```

In this example, an empty hash named `%oligos` is created, and three elements are then added to it. Each element has an index named after the clone from which it was derived, and a value containing the sequence of the oligo itself. After the assignments, the values can be accessed by indexing into the hash with curly braces.

```
$s = $oligos{'192a8'};
print "oligo 192a8 is $\n";
print "oligo 192a8 is ",length $oligos{'192a8'}," base pairs long\n";
print "oligo 18c10 is $oligos{'18c10'}\n";
```

This will print out

```
oligo 192a8 is GGGTTCCGATTTCCAA
oligo 192a8 is 16 base pairs long
oligo 18c10 is CTCTCTCTAGAGAGAGCCCC
```

Just as a variable containing an integer can be used as an index into an array, a variable containing a string can be used as the index into a hash. The following example uses a loop to print out the sequence of each of the three oligos previously defined by `%oligos`:

```
foreach $clone ('327h1','192a8','18c10') {
          print "$clone: $oligos{$clone}\n";
}
```

As with arrays, there is a distinction between the hash as a whole and individual elements of a hash. When referring to an element of a hash using its index surrounded by curly braces, one is referring to the *scalar* value contained within the hash, so the $ symbol must be used as a prefix. To refer to the hash variable as a whole, use the % symbol as the prefix. This allows for one hash to be assigned to another, as well as the ability to perform other whole-hash operations.

For historical reasons, the indexes of a hash are called its *keys*. Calling the **keys** function produces a list of all the keys in the hash. Using a command of the form `@clones = keys %clones;` will assign to the array three string elements (`'327h1','192a8','18c10'`), *but in no predictable order*. The elements of a hash are unordered, and the order in which they are put into a hash has no effect on the order in which they are returned.

To get all the values of a hash, use the **values** function:

```
@oligos = values %clones;
```

The `@oligos` array will now contain a three-element list consisting of each of the oligo sequences that were placed into the hash. As with keys, the values are returned in an unpredictable order. However, the order of elements retrieved by keys will match the order retrieved by values. Hence, the position of clone 192a8 in the `@clones` array will match the position of its corresponding oligo in `@oligos`.

A REAL-WORLD EXAMPLE

At this point in the discussion, all the tools needed to solve the problem posed at the very beginning of this chapter are now at hand. The problem will be approached in steps. The first task is to scan through a file of RNAi results, collecting the genes that have anything to do with locomotion. Assume that the input file is named `rnai.txt` and contains lines in the following format; each field is separated by a tab.

Gene	Date	Status	Phenotype Summary
B0310.2	2/18/2000	complete	larval arrest
B0379.4	2/18/2000	complete	none
B0496.8	2/19/2000	incomplete	
ZK899.6	2/19/2000	complete	uncoordinated, coils and kinks
ZK945.6	2/19/2000	complete	hermaphrodites sterile
M6.1	2/19/2000	complete	flaccid paralysis
...			

```
#!/usr/bin/perl

# STEP 1: extract RNA inhibition data
open RNAi,"rnai.txt" or die "Couldn't open rnai.txt: $!";
%interesting_genes = ();
while (<RNAi>) {
        chomp;
        ($gene,$date,$status,$phenotype) = split "\t";
        $interesting_genes{$gene} = $phenotype
                if $phenotype =~ /uncoordinated|paraly|coil|movement|kink|jerk/;
}
close RNAi;

# STEP 2: extract protein sequence data
open WP,"WormPep.fasta" or die "Can't open WormPep: $!";
%sequences = ();
while (<WP>) {
        if (/^>(\S+)$/) { # found a new description line
                $id = $1;
        } else {
                $sequences{$id} .= $_ if $interesting_genes{$id};
        }
}
close WP;

# STEP 3: reformat for submission to SignalP
open SIGNALP,">signalP.txt" or die "Couldn't open signalP.txt: $!";
print SIGNALP "euk\n";
print SIGNALP "graphics\n";
foreach $gene (keys %sequences) {
        print SIGNALP ">$gene $interesting_genes{$gene}\n";
        print SIGNALP $sequences{$gene};
}
close SIGNALP;
```

Figure 17.4. This script reformats a set of neuron-related *C. elegans* genes for submission to the SignalP signal peptide prediction program.

The main challenge here is to search the Phenotype Summary field for results having to do with locomotion. This is a fuzzy sort of problem, best solved using Perl's pattern-matching facility. After scanning through the results for a while, we decide to search for any of the following keywords: uncoordinated, paralysis, paralyzed, coils, coiler, movement, kinky, and jerky.

With this decision made, the first part of the script can be written (Fig. 17.4, Step 1). The script attempts to open the file `rnai.txt`. If unsuccessful, it dies with an error message. If successful, it initializes the hash `%interesting_genes` to empty. This hash will be used to hold the list of locomotion-related genes and will be set up so that the keys are gene names and the values are the corresponding phenotypes. The program then steps through the input file, one line at a time. It chomps off the newline from the end of each line and then uses the `split` function to split each line into fields, using the tab character as the delimiter. This results in a four-element list, which is assigned to an array containing the variables `$gene`, `$date`, `$status`, and `$phenotype`.

Once this is done, the contents of `$phenotype` are compared to a regular expression. The regular expression is a series of alternatives, separated by the | character. Any phenotype that contains any of the strings listed will produce a match. Notice that some of the keywords have been shortened to reduce the length of the expression, as well as pull in some terms that may not have been anticipated. For example, `paraly` will match both `paralysis` and `paralyzed`, without much risk of matching something unintended. If the pattern matches, then the corresponding gene is recorded. By the end of the loop, `%interesting_genes` will be populated with all of the genes whose phenotypes matched the target list. Because `rnai.txt` will no longer be used, its filehandle is closed.

In Step 2 of Figure 17.4, the WormPep set of predicted *C. elegans* proteins will be stepped through, pulling out all the ones matching the collection of genes identified in Step 1. WormPep's format is similar to the standard FASTA format:

```
>2L52.1 CE20433 Zinc finger, C2H2 type (CAMBRIDGE) protein id:CAA21776.1
MSMVRNVSNQSEKLEILSCKWVGCLKSTEVFKTVEKLLDHVTADHIPEVIVNDDGSEEVV
CQWDCCEMGASRGNLQKKKEWMENHFKTRHVRKAKIFKCLIEDCPVVKSSSQEIETH...
```

The portion of the definition line that immediately follows the > is the name of the gene. The program needs to step through all of these entries, extracting the gene names and saving the sequences of those contained in the collection of interesting genes. The script begins Step 2 by opening the WormPep file using a filehandle named `WP`. If successful, it initializes a hash named `%sequences`. This hash will have keys corresponding to the names of the interesting genes, with values consisting of their peptide sequences. The script then enters a loop in which it retrieves each line of the WormPep file, pattern-matching it against a regular expression that examines the def lines. If the program detects a definition line, meaning the beginning of a new sequence, it puts the gene name into a scalar variable named `$id`, taking advantage of Perl's ability to extract parenthesized portions of regular expressions.

If the current line does *not* match the regular expression, then the program has hit a sequence line, with the name of the current gene being held in `$id`. The `if` statement tests whether the current gene is an element of `%interesting_genes`, and, if so, the sequence will be read in, growing one line at a time, until the next definition line is reached. At the end of the loop, `%interesting_genes` will be fully populated with the sequences of interest. Note that the newline has not been removed from the end of each line of input sequence data; in this case, it is desirable to keep the newlines, to facilitate later parts of the program.

In Step 3 of Figure 17.4, the gene sequences are formatted into an E-mail message for submission to the SignalP server. The server expects E-mail submissions in the following format:

```
euk
graphics
>ID1 Comments (ignored)
MLETLCYNYLPLCEQLEPVLNVRDKEDLATSLVRVMYKHNLAKEFLCDLIMKEVEKL...
>ID2 More comments (ignored)
MPARRHLSQPAREGSLRACRSHESLLSSAHSTHMIELNEDNRLHPVHPSIFEVPNCF...
.
```

The first two lines contain information required by SignalP to properly process the sequences; here, the server is being instructed that the sequences are from a eukaryote and that graphics of the predictions should be returned. Following these flags are the sequences, in FASTA format. The sequence ID is required, and anything following it is ignored by the server. A dot follows the last sequence in the file.

The script begins Step 3 by attempting to open the file `signalP.txt` for writing. If successful, it writes the top two lines of the outgoing E-mail message to the file. The program then enters a `foreach` loop, calling the `keys` function to recover all keys from the `%sequences` hash. This retrieves all of the names of all of the genes for which sequence information has been assembled. For each gene, a description line containing the gene name and its phenotype is printed to the file-handle. Although the phenotype will be ignored by SignalP, the information is retained for future reference. After this, the sequence of the gene is printed, newlines and all. At the end of the loop, the `signalP.txt` filehandle is closed. The final step in the analysis is for the researcher to take the newly created `signalP.txt` file and to e-mail it to the SignalP server. The most useful part of this script is that it can be automatically rerun each time WormPep is updated to repeat the analysis.

WHERE TO GO FROM HERE

Perl has many features that cannot be covered in a single chapter. For example, *subroutines* allow one to define customized functions that take arguments and return a result. *References* allow sophisticated data structures such as lists of lists to be created. *Objects* allow large, complex programs to be written so that code can be reused in different contexts. *Pipes and processes* allow for the control of external programs, perhaps to create an automated pipeline invoking commonly used programs.

Last, but not nearly least, there are *modules*, which are libraries of useful code routines put together by Perl programmers around the world and made available for public use. For example, the `Mail::Sendmail` module would enable the SignalP-processing script to E-mail its formatted message directly to the SignalP server without creating an intermediate file. Other modules allow for the creation of interactive, graphical front ends for programs or the creation of dynamic Web pages. Most relevant for biologists are the modules that form Bioperl, an extremely powerful collection of tools for searching biological databases, manipulating and processing sequences, and analyzing nucleotides and proteins.

INTERNET RESOURCES FOR TOPICS PRESENTED IN CHAPTER 17

Perl home page	*http://www.perl.com/*
Comprehensive Perl Archive Network	*http://www.cpan.org/*
BioPerl	*http://www.bioperl.org/*

SUGGESTED READING

Schwartz, R. L. (1998). Learning Perl, 2nd Ed. (O'Reilly & Associates).

Christiansen, T., and Torkington, N. (1999) Perl Cookbook (O'Reilly & Associates).