The Complete Reference

Perl, Tcl/Tk, Expect, and Gawk

Perl, Tcl/Tk, Expect, and Gawk are scripting languages commonly used for customized applications on Linux. Your Red Hat Linux system installs these languages as part of its development package. Though beyond the scope of this book, a brief introduction to these languages is provided in this chapter.

Perl

The Practical Extraction and Report Language (Perl) is a scripting language originally designed to operate on files, generating reports and handling very large files. Perl was designed as a core program to which features could be easily added. Over the years, Perl's capabilities have been greatly enhanced. It can now control network connections, process interaction, and even support a variety of database management files. At the same time, Perl remains completely portable. A Perl script will run on any Linux system, as well as most other operating systems such as Unix, Windows, and Mac. Perl is also used extensively for implementing CGI scripts on Web sites. There are extensive and detailed man pages on Perl, discussing all aspects of the language with a great many examples. The man pages begin with the term **perl**; for example, **perlfunc** discusses the built-in Perl functions and **perlsyn** describes the different control structures. You can also use the **www.perl.com** site to access and search documentation including the reference manual, the online man pages, and FAQs.

There are extensive Internet resources for Perl. On the Perl Web site at **www.perl.com**, you can access documentation, software, newsgroups, and support. The site has programming and reference sections where you can access detailed FAQs on topics such as CGI, modules, and security. You can also access software archives and more detailed documentation.

Specialized Perl Web sites focus on programming, conferences, and reference resources. The Comprehensive Perl Archive Network (CPAN) maintains FTP sites that hold an extensive collection of utilities, modules, documentation, and current Perl distributions. You can also link to a CPAN site through the Perl Web sites. Several of the Perl Web sites are listed here:

www.perl.com
www.perlmongers.org
www.perl.org
republic.perl.com
www.perlreference.net
www.perl.com/CPAN/CPAN.html

The Perl advocacy group known as the perlmongers can be located at **www.perl.org** or **www.perlmongers.org**. The **republic.perl.com** site lets you join the Programming Republic of Perl. There are also several Usenet newsgroups that discuss different Perl

issues. You can use them to post questions and check out current issues. Here is a listing of the current newsgroups:

comp.lang.perl.announce
comp.lang.perl.misc
comp.lang.perl.modules
comp.lang.perl.tk

Perl Scripts

Usually, Perl commands are placed in a file that is then read and executed by the **perl** command. In effect, you are creating a shell in which your Perl commands are executed. Files containing Perl commands must have the extension **.pl**. This identifies a file as a Perl script that can be read by the **perl** command. There are two ways that you can use the **perl** command to read Perl scripts. You can enter the **perl** command on the shell command line, followed by the name of the Perl script. Perl will read and execute the commands. The following example executes a Perl script called **hello.pl**:

```
$ perl hello.pl
```

You can also include the invocation of the **perl** command within the Perl script file, much as you would for a shell script. This automatically invokes the Perl shell and will execute the following Perl commands in the script. The path **/usr/bin/perl** is the location of the **perl** command on the OpenLinux system. On other systems, it could be located in **/usr/local/bin** directory. The command **which perl** will return the location of Perl on your system. Place the following shell instruction on the first line of your file:

```
#!/usr/bin/perl
```

Then, to make the script executable, you would have to set its permissions to be executable. The **chmod** command with the **755** option sets executable permissions for a file, turning it into a program that can be run on the command line. You only have to do this once per script. You do not have to do this if you use the **perl** command on the command line, as noted previously. The following example sets the executable permissions for the **hello.pl** script:

```
$ chmod 755 hello.pl
```

As in C, Perl commands end with a semicolon. There is a **print** command for outputting text. Perl also uses the same escape sequence character to output newlines, **\n**, and tabs, **\t**. Comments are lines that begin with a **#**. The following is an example

of a Perl script. It prints out the word "hello" and a newline. Notice the invocation of the **perl** command on the first line:

```
helloprg

#!/usr/bin/perl

print "hello \n";

$ helloprg
hello
```

Perl Input and Output: <> and print

A Perl script can accept input from many different sources. It can read input from different files, from the standard input, and even from pipes. Because of this, you have to identify the source of your input within the program. This means that, unlike with Gawk but like with a shell program, you have to explicitly instruct a Perl script to read input. A particular source of input is identified by a file handle, a name used by programs to reference an input source such as a particular file. Perl already sets up file handles for the standard input and the standard output, as well as the standard error. The file handle for the standard input is **STDIN**.

The same situation applies to output. Perl can output to many different destinations, whether they be files, pipes, or the standard output. File handles are used to identify files and pipes when used for either input or output. The file handle **STDOUT** identifies the standard output, and **STDERR** is the file handle for the standard error. We shall first examine how Perl uses the standard input and output, and later discuss how particular files are operated on.

Perl can read input from the standard input or from any specified file. The command for reading input consists of the less-than (**<**) and greater-than (**>**) symbols. To read from a file, a file handle name is placed between them, **<MYFILE>**. To read from the standard input, you can simply use the **STDIN** file handle, **<STDIN>**, which is similar to the **read** command in the BASH shell programming language.

```
<STDIN>
```

To use the input that **<STDIN>** command reads, you assign it to a variable. You can use a variable you define or a default variable called **$_**, as shown in the next example. **$_** is the default for many commands. If the **print** command has no argument, it will print the value of **$_**. If the **chomp** command has no argument, it operates on **$_**, cutting off the newline. The **myread** script that follows illustrates the use of **$_** with the standard input:

```
myread

#!/usr/bin/perl
# Program to read input from the keyboard and then display it.

$_ = <STDIN>; #Read data from the standard input
print "This is what I entered: $_"; #Output read data as part of a string

$ myread
larisa and aleina
This is what I entered: larisa and aleina
```

You can use the **print** command to write data to any file or to the standard output. File handle names are placed after the **print** command and before any data such as strings or variables. If no file handle is specified, **print** outputs to the standard output. The following examples both write the "hello" string to the standard output. The explicit file handle for the standard output is **STDOUT**. If you do not specify an argument, **print** will output whatever was read from the standard input.

```
print STDOUT "hello";
print "hello";
```

Tip

*A null file handle, **<>**, is a special input operation that will read input from a file listed on the command line when the Perl script is invoked. Perl will automatically set up a file handle for it and read. If you list several files on the command line, Perl will read the contents of all of them using the null file handle. You can think of this as a **cat** operation in which the contents of the listed files are concatenated and then read into the Perl script.*

Perl File Handles

You use the **open** command to create a file handle for a file or pipe. The **open** command takes two arguments: the name of the file handle and the filename string. The name of the file handle is a name you make up. By convention, it is uppercase. The filename string can be the name of the file or a variable that holds the name of the file. This string can also include different modes for opening a file. By default, a file is opened for reading. But you can also open a file for writing, or for appending, or for both reading and writing. The syntax for **open** follows:

```
open ( file-handle, filename-string);
```

In the next example, the user opens the file reports, calling the file handle for it **REPS**:

```
open (REPS, "reports");
```

Often the filename will be held in a variable. You then use the **$** with the variable name to reference the filename. In this example, the filename "reports" is held in the variable **filen**:

```
filen = "reports";
open (REPS, $filen );
```

To open a file in a specific mode such as writing or appending, you include the appropriate mode symbols in the filename string before the filename, separated by a space. The different mode symbols are listed in Table 1. The symbol **>** opens a file for writing, and **+>** opens a file for both reading and writing. In the next example, the reports file is opened for both reading and writing:

```
open (REPS, "+> reports");
```

If you are using a variable to hold the filename, you can include the evaluated variable within the filename string, as shown here:

```
open (REPS, "+> $filen");
```

To read from a file using that file's file handle, you simply place the file handle within the **<** and **>** symbols. **<REPS>** reads a line of input from the reports file. In the **myreport** program, the reports file is opened and its contents are displayed.

myreport.pl

```
#!/usr/bin/perl
# Program to read lines from the reports file and display them

open(REPS, "< reports"); # Open reports file for reading only
while ( $ldat = <REPS> ) # Read a line from the reports file
 {
 print $ldat; # Display recently read line
 }
close REPS; # Close file
```

Perl also has a full set of built-in commands for handling directories. They operate much like the file functions. The **opendir** command opens a directory, much as a file is opened. A directory handle is assigned to the directory. The **readdir** command will read the first item in the directory, though, when in a list context, it will return all the file and directory names in that directory. **closedir** closes the directory, **chdir** changes directories, **mkdir** creates directories, and **rmdir** removes directories.

Perl Command Line Options	Description
-e	Enter one line of a Perl program.
-n	Read from files listed on the command line.
-p	Output to standard output any data read.
Perl File Commands	
open(*file-handle***,** *permission-with-filename***)**	Open a file.
close(*file-handle***)**	Close a file.
< *filename***>**	Read from a file.
<STDIN>	Read from the standard input.
<>	Read from files whose filenames are provided in the argument list when the program was invoked.
print <file-*handle*> *text***;**	Write to a file. If no file handle is specified, write to standard output. If no text is specified, write contents of **$_**.
printf < **handle>** " *Format-str* **",** *values* **;**	Write formatted string to a file. Use conversion specifiers to format values. If no file handle is specified, write to standard output. If no values are specified, use contents of **$_**.
sprintf *str-var* " *Format-str* **",** *values* **;**	Write formatted values to a string. Use conversion specifiers to format values. If no values are specified, use contents of **$_**.
Permissions for Opening Files	
< *filename*	Read-only.
> *filename*	Write-only.
+> *filename*	Read and write.
>> *filename*	Append (written data is added to the end of the file).
command **|**	An input pipe, reading data from a pipe.
| *command*	An output pipe, sending data out through this pipe.

Table 1. *Perl File Operations and Command Line Options*

Perl Variables and Expressions

Perl variables can be numeric or string variables. Their type is determined by context—the way they are used. You do not have to declare them. A variable that is assigned a numeric value and is used in arithmetic operations is a numeric variable. All others are treated as strings. To reference a variable in your program, you precede it with a **$** symbol, just as you would for a shell variable.

You can use the same set of operators with Perl variables as with C variables—with the exception of strings. Strings use the same special comparison terms as used in the Bourne shell, not the standard comparison operators. Those are reserved for numeric variables. However, other operators such as assignment operators work on both string and numeric variables. In the next example, the variable **myname** is assigned the string "Larisa". The assignment operator is the **=** symbol (see Table 2).

```
$myname = "Larisa";
```

For a numeric variable, you can assign a number. This can be either an integer or a floating-point value. Perl treats all floating-point values as double precision.

```
$mynum = 45;
$price = 54.72;
```

Perl also supports arithmetic expressions. All the standard arithmetic operators found in other programming languages are used in Perl. Expressions can be nested using parentheses (see Table 2). Operands can be numeric constants, numeric variables, or other numeric expressions. In the following examples, **$mynum** is assigned the result of an addition expression. Its value is then used in a complex arithmetic expression whose result is assigned to **$price**.

```
$mynum = 3 + 6;
$price = ( 5 * ($num / 3);
```

Perl supports the full range of assignment operators found in Gawk and C. The **++** and «— operators will increment or decrement a variable. The **+=** and **-=** operators and their variations will perform the equivalent of updating a variable. For example, **i++** is the same as **i = i + 1**, and **i += 5** is the same as **i= i + 5**. Increment operations such as **i++** are used extensively with loops.

You can easily include the value of a variable within a string by simply placing the variable within it. In the following example, the value of **$nameinfo** would be the string "My name is Larisa \n":

```
print "The number of items is $mynum \n"
$nameinfo = "My name is $myname \n"
```

To assign data read from a file to a variable, just assign the result of the read operation to the variable. In the next example, data read from the standard input is assigned to the variable **$mydata**:

```
$mydata = <STDIN>;
```

When reading data from the standard input into a variable, the carriage return character will be included with the input string. You may not want to have this carriage return remain a part of the value of the variable. To remove it, you can use the **chomp** command, which removes the last character of any string. With data input from the keyboard, this happens to be the carriage return.

```
chomp $myinput;
```

In the next example, the user inputs his or her name. It is assigned to the **myname** variable. The contents of **myname** is then output as part of a string. **chomp** is used to remove the carriage return from the end of the **$myname** string before it is used as part of another string.

```
readname.pl

#!/usr/bin/perl
$myname = <STDIN>;
chomp $myname;

print "$myname just ran this program\n";

$ myread.pl
larisa Petersen
larisa Petersen just ran this program
```

Arithmetic Operators	Function
*	Multiplication
/	Division
+	Addition

Table 2. *Arithmetic, Relational (Numeric), and Assignment Operators*

Arithmetic Operators	Function
-	Subtraction
%	Modulo—results in the remainder of a division
**	Power
Relational Operators	
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal in let
!=	Not equal
Increment Operators	
++	Increment variable by one
«—	Decrement variable by one
Arithmetic Assignment Operators	
+=	Increment by specified value
-=	Decrement by specified value
/=	Variable is equal to itself divided by value
*=	Variable is equal to itself multiplied by value
%=	Variable is equal to itself remaindered by value

Table 2. *Arithmetic, Relational (Numeric), and Assignment Operators* (continued)

Arrays and Lists

In Perl, you create an array by assigning it a list of values. A list in Perl consists of a set of values encased in parentheses and separated by colons. The following example is a list of four values:

```
( 23, 41, 92, 7)
```

You assign this list to the array you wish to create, preceding the array name with an **@** sign. This assignment will initialize the array sequentially, beginning with the first value in the list:

```
@mynums = (23, 41, 92, 7);
```

Once the array has been created, you can reference its individual elements. The elements start from 0, not 1. The **mynums** array has four elements, numbered from 0 to 3. You can reference individual elements using an index number encased within brackets. **[0]** references the first element, and **[2]** references the third element. The following example prints out the first element and then the fourth element. Notice that the array name is prefixed with a **$**.

```
print $mynums[0] ;
print $mynums[2] ;
```

You can change the value of any element in the array by assigning it a new value. Notice that you use a **$**, not an **@** sign, preceding an individual array element. The **@** sign references the entire array and is used when you are assigning whole lists of values to it. The **$** sign references a particular element, which is essentially a variable.

```
$mynums[2] = 40;
```

There is no limit to the number of elements in the array. You can add more by simply referencing a new element and assigning it a value. The following assignment will add a fifth element to the **mynums** array:

```
$mynums[4] = 63;
```

Each array will have a special variable that consists of a **#** and the name of the array. This variable is the number of elements currently in the array. For example, **#mynums** holds the number of elements in the **mynums** array. The following example prints out the number of elements. Notice the preceding **$**.

```
print "$#mynums";
```

When assigning a list to an array, the values in a list do not have to be of the same type. You can have numbers, strings, and even variables in a list. Similarly, elements of the array do not have to be of the same type. One element could be numeric, and another a string. In the next example, the list with varied elements is assigned to the **myvar** array:

```
@myvar = ( "aleina", 11, 4.5, "a new car");
```

You can reference the entire set of elements in an array as just one list of values. To do this, you use the array name prefixed by the **@** sign. The following example will output all the values in the **mynums** array:

```
print @mynums;
```

The **@** is used here instead of the **$**, because the array name is not a simple variable. It is considered a list of values. Only the individual elements are variables. This means that to just reference all the values in an array, you use the **@** sign, not the **$**. This is even true when you want to assign one array to another. In the next example, the values of each element in **mynums** are assigned to corresponding elements in **newnums**. Notice the **@** used for **mynums**. You can think of **@mynums** as evaluating to a list of the values in the **mynums** array, and this list being then assigned to **newnums**.

```
@newnums = @mynums;
```

Perl Control Structures

Perl has a set of control structures similar to those used in the Gawk, TCSH shell, and C programming languages. Perl has loops with which you can repeat commands, and conditions that allow you to choose among specified commands. For the test expressions, there are two different sets of operators for use with strings and numeric values. Table 2 lists the numeric relational operators, and Table 3 lists the string operators. You can also use pattern operations that allow the use of regular expressions. Table 4 lists the Perl control structures with their syntax.

Test Expressions

Perl has different sets of operators for numeric and string comparisons. You have to be careful not to mix up the different operators. The string operators are two-letter codes similar to those used in the BASH shell. For example, the **eq** operator tests for the equality of two strings, and the **gt** operator tests to see if one is greater than the other. Numeric comparisons, on the other hand, use as operators symbols similar to those found in programming languages. For example, **>** stands for greater than, and **==** tests for equality. These are essentially the same comparison operators used for the C programming language.

There are two important exceptions: patterns and string patterns. The string-pattern operator, **=~**, tests for a pattern in a string variable. The right-hand operand is the pattern, and the left-hand operand is the string. The pattern can be any regular expression, making this a very flexible and powerful operation.

Patterns perform pattern matching on either a string or the contents of the **_$** special variable. The pattern operator consists of two slashes that enclose the pattern searched for, ***/pattern/***. The pattern can be any regular expression. Regular expression are discussed in detail in the section on pattern matching.

Perl supports the AND, OR, and NOT logical operations. There are two implementations of these operations: the standard operators and those with list processing capabilities. The standard logical operators are **&&**, **||**, and **!**. The **&&** operator implements an AND operation, **||** an OR, and **!** a NOT. They take as their operands expressions, just as they do in the C programming language. Their syntax is as follows:

```
(expression) && (expression)
(expression) || (expression)
!(expression)
```

In the case of the logical AND, **&&**, if both commands are successful, the logical command is true. For the logical OR, **||**, if either command is successful, the OR is true. You can extend these commands with added **&&** or **||** commands, creating complex AND or OR operations. The logical commands allow you to use logical operations as your test commands in control structures. You can also use them independently.

The logical operators are usually used in test expressions for control structures such as **while** and **if**. But they can also be used independently as their own statements. In effect, they provide a simple way to write a conditional operation. In Perl scripts, you may often see an OR operation used with a file **open** command. In an OR operation, if the first expression fails, the second one is checked. If that second one is the **die** command to end the program, in effect there is an operation to end the program if the file **open** operation fails (the first expression fails).

```
open (REPS, "+> $filen") or die "Can't open $filen";
```

The AND operation works similarly, except that if the first expression is true, the second one is checked. The following looks for an empty line and, if it finds it, prints the message:

```
/^$/ && print "Found empty line";
```

String Comparisons	Function
gt	Greater than.
lt	Less than.
ge	Greater than or equal to.
le	Less than or equal to.
eq	Equal.
ne	Not equal.
Logical Operations	
expression **&&** *expression* *expression* **and** *expression*	The logical AND condition returns a true 0 value if both expressions return a true 0 value; if one returns a nonzero value, the AND condition is false and also returns a nonzero value. Execution stops at the first false expression. The **and** operation is the same as **&&** but has a lower precedence.
expression **||** *expression* *expression* **or** *expression*	The logical OR condition returns a true 0 value if one or the other expression returns a true 0 value; if both expressions return a nonzero value, the OR condition is false and also returns a nonzero value. Evaluation stops at the first true expression. The **or** operation is the same as **||** but has a lower precedence.
! *command* **not** *command*	The logical NOT condition inverts the true or false value of the expression. The **not** operation is the same as **!** but has a lower precedence.
File Tests	
-e	File exists.
-f	File exists and is a regular file.
-s	File is not empty.
-z	File is empty, zero size.
-r	File is readable.

Table 3. *String, Logical, File, and Assignment Operators*

String Comparisons	Function
File Tests *(continued)*	
`-w`	File can be written to, modified.
`-x`	File is executable.
`-d`	Filename is a directory name.
`-B`	File is a binary file.
`-T`	File is a text file.
Assignment Operator	
`=`	Assign a value to a variable.

Table 3. *String, Logical, File, and Assignment Operators* (continued)

Loops

Perl loops are the **`while`**, **`do-until`**, **`for`**, and **`foreach`** loops. The **`while`** loop is the more general-purpose loop, whereas the **`for`** and **`foreach`** loops provide special capabilities. The **`foreach`** is particularly helpful in processing lists and arrays. The **`while`**, **`do-until`**, and **`for`** loops operate much like their counterparts in the C programming language. The **`for`** loop, in particular, has the same three expression formats as the C **`for`** loop. The **`foreach`** loop is similar to its counterpart in the C shell, able to easily handle lists of items.

You can easily adapt the **`while`** loop for use with arrays. The variable used to control a loop can also be used, inside the loop, to index an array. In the next example, the elements of the **title** array are assigned the value of each title. Then, the contents of each element are printed out using a **`for`** loop. Notice that the **`$#num`** holds the count of elements in the array. **`$#num`** is used as the upper bound in the **`for`** loop test expression in order to check when the loop should stop.

```
titlearr.pl

#!/usr/bin/perl
# Program to define and print out a scalar array

@title = ( "Tempest", "Iliad", "Raven"); # define array with 3 elements

for($i = 0; $i <= $#title; $i++) # Loop through array, $#title is size
```

```
 {
 print "$title[$i] \n"; # Print an element of the title array
 }

$ titlearr.pl
Tempest
Iliad
Raven
```

The **foreach** loop is useful for managing arrays. In the **foreach** loop, you can use the array name to generate a list of all the element values in the array. This generated list of values then becomes the list referenced by the **foreach** loop. You can also specify a range of array elements, using only those values for the list, or a set of individual elements. In the next example, the array name **@mylist** is used to generate a list of its values that the **foreach** loop can operate on, assigning each one to **$mynum** in turn:

```
mynumlist.pl

#!/usr/bin/perl
# Program to use foreach to print out an array

@mylist = ( 34, 21, 96, 85); # define array of 4 elements

foreach $mynum ( @mylist ) # Assign value of each element to $mynum
 {
 print "$mynum \n";
 }
```

Using the **@ARGV** array, you can specify the command line arguments as a list of values. The arguments specified on the command line when the program was invoked become a list of values referenced by the **foreach** loop. The variable used in the **foreach** loop is set automatically to each argument value in sequence. The first time through the loop, the variable is set to the value of the first argument. The second time, it is set to the value of the second argument, and so on.

Tip

*The number of arguments that a user actually enters on the command line can vary. The **#ARGV** variable will always hold the number of arguments that a user enters. This is the number of elements that are in the **ARGV** array. If you want to reference all the elements in the **ARGV** array using their indexes, you will need to know the number of elements, **#ARGV**. For example, to use the **foreach** loop to reference each element in the **ARGV** array, you would use the **..** operator to generate a list of indexes. **0.. $#ARGV** generates a list of numbers beginning with 0 through to the value of **$#ARGV**.*

Conditions: if, elsif, unless, and switch

Perl supports if-else operations much as they are used in other programming languages. The **if** structure with its **else** and **elsif** components allows you to select alternative actions. You can use just the **if** command to choose one alternative, or combine that with **else** and **elsif** components to choose among several alternatives. The **if** structure has a test expression encased in parentheses followed by a block of statements. If the test is true, the statements in the block are performed. If not, the block is skipped. Unlike in other programming languages, only a block can follow the test, and any statements, even just one, must be encased with it. The following example tests to see if an **open** operation on a file was successful. If not, it will execute a **die** command to end the program. The NOT operator, **!**, will make the test true if **open** fails, thereby executing the **die** command.

```
if (!(open (REPS, "< $filen"))) {
 die "Can't open $filen";
}
else {
 print "Opened $filen successfully";
 }
```

Control Stucture	Description
LABEL: **{** *statements*; **}**	Block is a collection of statements enclosed within opening and closing braces. The statements are executed sequentially. The block can be labeled.
Conditional Control Structures: if, else, elsif, case	
if(*expression* **)** { *statements*; **}**	**if** executes statements if its test expression is true. Statements must be included within a block.
if(*expression* **)** { *statements*; **}** **else(** *expression* **)** { *statements*; **}**	**if-else** executes statements if the test expression is true; if false, the **else** statements are executed.

Table 4. *Perl Conditions, Loops, and Functions*

Control Stucture	Description
Conditional Control Structures: if, else, elsif, case *(continued)*	

if(*expression* **)** {
statements;
}
elsif(*expression* **)** {
statements;
}
else(*expression* **)** {
statements;
}
	elsif allows you to nest **if** structures, enabling selection among several alternatives; at the first true **if** expression, its statements are executed and control leaves the entire **elsif** structure.

unless(*expression* **)** { *statements*; **}**	**unless** executes statements if its test expression is false.
test **?** *stat1* **:** *stat2*	Conditional expression. If true, executes *stat1*, **else** *stat2*.
LABEL:**{** **if(***expr***){***statements***;last** *LABEL***}**; **}**	Simulate a **switch** structure by using listed **if** statements within a block with the **last** statement referencing a label for the block.
Loop Control Structures: while, until, for, foreach	
LABEL:**while(** *expression* **)** { *statements*; **}**	**while** executes statements as long as its test expression is true. LABEL is optional.
do{ *statements*; **}** until(*expression* **)**	**until** executes statements as long as its test expression is false.
foreach *variable* **(** *list-values***)** **{** *statements*; **}**	**foreach** is designed for use with lists of values such as those generated by an array; the variable operand is consecutively assigned the values in the list.

Table 4. *Perl Conditions, Loops, and Functions* (continued)

Control Stucture	Description
Loop Control Structures: while, until, for, foreach *(continued)*	
for(*init-expr*; *test-expr*; *incr-expr***)** **{** *statements*; **}**	The **for** control structure executes statements as long as *test-expr* is true. The first expression, *init-expr*, is executed before the loop begins. The third expression, *incr-expr,* is executed within the loop after the statements.
LABEL: *block-or-loop*	Label a block or loop. Used with the **next**, **last**, and **redo** commands.
Functions	
sub *function-name* ;	Declare a function.
sub *function-name* **{** *statements*; **}**	Define a function with the name *function-name.*
& *function-name***(***arg-list***)**	Call a function with arguments specified in the argument list.
@_	Holds the values of arguments passed to the current function. **$**_ and index references an argument. **$**_[0] is the first argument.
$#_	Number of arguments passed to the current function.

Table 4. *Perl Conditions, Loops, and Functions* (continued)

Tcl, Tk, and Expect

Tcl is general-purpose command language developed by John Ousterhout in 1987 at the University of California, Berkeley. Originally designed to customize applications, it has become a fully functional language in its own right. As with Perl and Gawk, you can write Tcl scripts, developing your own Tcl programs. Tcl is a very simple language to use.

TK and Expect are Tcl applications that extend the capabilities of the language. The Tk application allows easy development of graphical interactive applications. You can create your own windows and dialog boxes with buttons and text boxes of your choosing. The Expect application provides easy communication with interactive programs such as FTP and telnet.

Tcl is often used in conjunction with Tk to create graphical applications. Tk is used to create the graphical elements such as windows, and Tcl performs the programming actions such as managing user input. Like Java, Tcl and Tk are cross-platform applications. A Tcl/Tk program will run on any platform that has the Tcl/Tk interpreter installed. Currently, there are Tcl/Tk versions for Windows, the Mac, and Unix systems, including Linux. You can write a Tcl application on Linux and run the same code on Windows or the Mac. The new versions of Tcl and Tk 8.0 even support a local look and feel for GUI widgets using Mac-like windows on the Mac, but Windows-like windows under Windows 98.

Tcl is an interpreted language operating, like Perl, within its own shell; **tclsh** is the command for invoking the Tcl shell. Within this shell, you can then execute Tcl commands. You can also create files within which you can invoke the Tcl shell and list Tcl commands, effectively creating a Tcl program. A significant advantage to the Tcl language and its applications is the fact that it is fully compatible with the C programming language. Tcl libraries can be incorporated directly into C programs. In effect, this allows you to create very fast compiled versions of Tcl programs.

When you install Tk and Tcl on your system, Man pages for Tcl/Tk commands are also installed. Use the **man** command with the name of the Tcl or Tk command to bring up detailed information on that command. For example, **man switch** displays the manual page for the Tcl **switch** command, and **man button** displays information on the Tk button widget. Once you have installed Tk, you can run a demo program called **widget** that shows you all the Tk widgets available. The **widget** program uses Tcl/Tk sample programs and can display the source code for each. You can find the **widget** program by changing to the Tk **demos** directory as shown here. (**tk*** here matches on directory names consisting of **tk** and its version number, **tk4.1** for version 4.1, and **tk8.0** for version 8.0.)

```
cd /usr/lib/tk*/demos
```

From the Xterm window, just enter the command **widget**. You can also examine the individual demo files and modify them as you wish. If you have installed a version of Tk yourself into the **/usr/local/bin** directory rather than **/usr/bin**, the **demos** directory will be located in **/usr/local/lib/tk***.

Tcl/Tk Extensions and Applications

Currently, both Tcl and Tk are being developed and supported as open-source projects by Tcl Core Team. The current release of both Tcl and Tk is 8.0, though version 8.1 will

be ready soon. Current versions of Tcl and Tk are available free of charge from the Tcl Developer Xchange Web site, **http://dev.scriptics.com**. Also available on this site is extensive documentation for each product in PostScript, Adobe PDF, and HTML formats. The HTML documentation can be viewed online. RPM packaged versions can also be found at distribution sites such as **ftp.redhat.com**. You will need both Tcl and Tk RPM packages as well as the development packages for each.

Tcl/Tk has been enhanced by extensions that increase the capabilities of the language. Several commonly used extensions are TclX, [incr Tcl], and Oratcl. All these are currently available though links on the Tcl Developer Xchange Web site. Access the Tcl Software Resource page and from there, you can access the Tcl Extensions page, listing those currently available (**dev.scriptics.com/software**). Most you can download and install for free. Most are also available at **http://sourceforge.net**. TclX extends capabilities such as file access and time and date manipulation, many of which have been incorporated into recent Tcl releases. [incr Tcl] supports the easy implementation of higher-level widgets, using an object-oriented programming structure. BLT adds graph and bar widgets to Tk. Sybtcl and Oratcl implement database interfaces to the Sybase and Oracle databases. TclDP provides support for distributed programming across a network. With the Scotty Network Management Tools, you can develop network management applications. The TrfCrypt extension adds encryption that was removed from the standard Tcl/Tk release to make it exportable.

Numerous Tcl/Tk applications, development tools, and utilities are freely available for download from Internet sites. You can link to most of these through the Tcl Developer Xchange site or through **www.tcltk.com** site. Most products are also open-source projects available at **http://sourceforge.net**. If a site does not provide an RPM-packaged version of its software, be sure to check the appropriate distribution site, such as **ftp.redhat.com**. Of particular note is the Tcl/Tk Web page plug-in that allows you to embed Tcl/Tk programs within a Web page. Such embedded Tcl/Tk programs are called Tclets. There is also a Tcl/Tk Web server called TclHttpd that can be easily embedded in applications, making them Web capable. You can configure and modify the server using Tcl/Tk commands. Several GUI builders are also available that let you build graphical user interfaces (GUIs). Free Visual Tcl, SpecTcl, VisualGIPSY, and XF SpecTcl are GUI builders that have a window-based interface with menus and icons for easily creating Tcl/Tk widgets. They can generate both Tcl/Tk and Java code, and can be used to create standalone Tcl/Tk applications or Web Tclets. There are also editors (TextEdit), desktops (TK Desk), multimedia (MBone), and specialized applications like Tik, a Tcl/Tk implementation of AOL Instant Messenger.

Tcl

Tcl is a simple-to-use programming language. Its statements consist of a command followed by arguments, though it also has a complete set of control structures including **`while`** and **`for`** loops. Commands can be terminated either by a semicolon or by a newline. You can think of a Tcl command as a function call where the command name operates like a function name, followed by arguments to the function. However, unlike with a function call, there are no parentheses or commas encasing the arguments. You

simply enter the command name and then its arguments, separated only by spaces. A newline entered after the last argument will end the statement.

You can see the features in this format very clearly in the Tcl assignment command, **set**. To assign a value to a variable, you first enter the assignment command **set**. Then enter the name of the variable, followed by the value to be assigned. The command name, variable, and value are separated only by spaces. The newline at the end of the line ends the statement. The following statement assigns a string **"larisa"** to the variable **myname**, and the next statement assigns the integer value 11 to the variable **age**:

```
set myname "larisa"
set age 11
```

Note *Variable types are determined by their use. A variable assigned an integer will be considered an integer, and one assigned a string will be a character array.*

The Tcl Shell and Scripts: tclsh

You execute Tcl commands within the Tcl shell. You can do this interactively, entering commands at a Tcl shell prompt and executing them one by one, or you can place the commands in a script file and execute them all at once. To start up the Tcl shell, you enter the command **tclsh**. This starts up the Tcl shell with the % prompt. You can then enter single Tcl commands and have them evaluated when you press ENTER. You leave the Tcl shell by entering either an **exit** command or a CTRL-D.

```
$ tclsh
% set age 11
% puts $age
11
% exit
$
```

You can run a Tcl script either as a standalone program or as a file explicitly read by the Tcl shell command **tclsh**. A Tcl script has the extension **.tcl**. For example, the **myread.tcl** Tcl script would be read and executed by the following command:

```
$ tclsh myread.tcl
```

To create a standalone script that operates more like a command, you need to invoke the **tclsh** command within the script. You can do this using an explicit pathname for the **tclsh** command. This is placed on the first line of the script.

```
#!/usr/bin/tclsh
```

Expressions

Expressions are also handled as commands. The command **expr** evaluates an expression and returns its resulting value as a string. It takes as its arguments the operands and operator of an expression. Tcl supports all the standard arithmetic, comparison, and logical operators. The result of an arithmetic expression will be the same form as its operands. If the operands are real numbers, the result will be a real number. You can mix operands of different types, and Tcl will convert one to be the same as the other. In the case of real and integer operands, the integer will be converted to a real. In the next statement, the addition of 4 and 3 is evaluated by the **expr** command. The following statement multiplies 25 and 2:

```
expr 4 + 3
expr 25 * 2
```

Tip *The resulting value returned by any Tcl command is always a string. In the case of arithmetic operations, the arithmetic value is converted first to a string, which is then returned by the* ***expr*** *command.*

Embedded Commands

You can combine commands by embedding one within the other. Embedding is commonly used for assigning the result of an expression to a variable. This involves two commands, the **set** command to perform the assignment and the **expr** command to evaluate an expression. You embed commands using brackets. An embedded command is another Tcl command whose result is used as an argument in the outer Tcl command. The embedded command is executed first, and its result is used as the argument to the outer command. The following statement assigns the result of the arithmetic operation, 25 * 2, to the variable **num**. **expr 25 * 2** is a command embedded within the **set** command. First the embedded command is executed, and its result, "50", is assigned to the variable **num**.

```
set num [expr 25 * 2]
```

Variables

Tcl supports numeric and string variables as well as arrays, including associative arrays. All variables hold as their contents a string. However, though the content of a variable is a string, that string can be used as an integer or real value in an arithmetic expression, provided that the string consists of numbers. Whenever such a variable is used as an operand in an arithmetic expression, its contents are first converted to an integer or real value. The operation is performed on the arithmetic values, and the result returned by **expr** is then converted back to a string. This means that you do not have to worry about declaring the type of variable, or even defining a variable. All variables are automatically defined when they are first used in a statement.

As we have seen, variables can be assigned values using the **set** command. The **set** command takes as its argument the variable name and the value assigned. A variable's name can be any set of alphabetic or numeric characters and the underscore. Punctuation and other characters are not allowed.

When you need to use the value of a variable within a statement, you need to evaluate it. Evaluating a variable substitutes its name with its value. The **$** placed before a variable name performs such an evaluation. To use a variable's value as an operand in an expression, you need to evaluate the variable by preceding its name with the **$**. In the next example, the value 5 is assigned to the **mynum** variable. Then **mynum** is evaluated in an expression, **$mynum**, providing its value, 5, as an operand in that expression.

```
set mynum 5
expr 10 * $mynum
```

Should you want to make the value of a variable part of string, you only need to evaluate it within that string. The value of the variable becomes part of the string. In the following statement, the value of the variable **myname** is used as part of a string. In this case, the string will be **"My name is Larisa"**.

```
set myname "Larisa"
set mystr "My name is $myname"
```

Certain commands are designed to operate on variables. The **append** command concatenates a string to a variable. The **incr** command will increment an integer, and the **unset** command will undefine a variable. The different commands that operate on variables are listed in Table 5.

Commands	Description
set	Assign a value to a variable
global	Declare global variables
incr	Increment a variable by an integer value
unset	Delete variables
upvar	Reference a variable in a different scope
variable	Declare namespace variables
array	Array access operations like searches
expr	Math expressions

Table 5. *Assignments and Variables*

Arrays

Array elements are defined and assigned values using the **set** command with the index encased in parentheses. The following example assigns the string **"rain"** as the second element in the **weather** array:

```
set weather(2) rain
```

You can then reference the element using its index encased in parentheses with the array name preceded with a **$**.

```
puts $weather(2)
rain
```

Tcl allows the use of any word string as an index, in effect supporting associative arrays. The string used as the index is encased within parentheses next to the array name. The following statements add two elements to the **city** array with the index strings **Napa** and **Alameda**:

```
set city(Napa) 34
set city(Alameda) 17
```

Control Structures

Tcl has a set of control structures similar to those used in the Perl, Gawk, C shell, and C programming languages. Tcl has loops with which you can repeat commands and conditions that allow you to choose among specified commands. Table 6 lists the Tcl control structures. Control structures in Tcl often make use of a block of Tcl commands. A block of commands consists of Tcl commands enclosed in braces. The opening brace ({) will begin on the same line as that of the control structure that uses it. On following lines, there can be several Tcl commands, each on its own line. The block ends with a closing brace (}) on a line by itself. A block is literally an argument to a Tcl command. The block is passed to a control structure command, and the control structure will execute the commands in that block.

if

The **if** control structure allows you to select alternative actions. The **if** command takes two arguments, a test expression and a Tcl command or block of commands. Each is encased in its own set of braces. The test expression is used to determine if the Tcl commands will be executed. If the test expression is true, the commands are

Control Structures	Description
if	Conditional command, extend with else and **elseif** blocks
switch	Switch selection structure
while	The **while** loop
for	The **for** loop, like the C **for** loop
foreach	Loop through a list, or lists, of values
break	Forced loop exit

Table 6. *Tcl Control Structures*

performed. If not, the commands are skipped. Below is the syntax for the **if** structure. You can specify alternative commands to execute if the expression is false by attaching an **else** block with those Tcl commands. You can nest **if** structures using the **elseif** command.

```
if {test-expression} {
 Tcl commands
 } elseif {test-expression} {
 Tcl commands
 } else {
 Tcl commands
 }
```

Note *Keep in mind that the* **else** *and* **elseif** *keywords must be on the same line as the closing brace of the previous block, as in* **} elseif { test-expression} {** *. The opening brace of the next block must also be on the same line.*

switch

The **switch** structure chooses among several possible alternatives. The choice is made by comparing a string value with several possible patterns. Each pattern has its own block of Tcl commands. If a match is found, the associated block is executed. The **default** keyword indicates a pattern that matches anything. If all of the other matches fail, the block associated with the **default** keyword is executed. The **switch** structure begins with the keyword **switch**, the options prefixed with **-**, and the string pattern to

be matched, followed by a block containing all the patterns with their blocks. The syntax for the **switch** structure is described next:

```
switch -options string-pattern {
pattern {
Tcl commands
}
pattern {
Tcl commands
}
default {
Tcl commands
}
}
```

Options specify the pattern-matching capabilities. The following options are supported:

-exact	Use exact matching when comparing string to a pattern. This is the default.
-glob	When matching string to the patterns, use **glob** style matching.
-regexp	When matching string to the patterns, use regular expression matching (i.e., the same as implemented by the **regexp** command).
«—	Marks the end of options. The argument following this one will be treated as a string even if it starts with a -.

The **-regexp** option lets you match any regular expression, whereas **-glob** lets you use the shell filename matching methods. With **-glob**, the shell special characters *****, **[]**, **?** let you easily match on part of a string. With the **-regexp** option, you can match on complex alternative strings, specifying multiple instances of characters, the beginning or end of a string, and classes of characters. For example, to match on all filenames with the **.c** extension, you would use the following command:

```
switch -glob *.c
```

To match on words that end with a number, like "report17" and "report32," you could use the following command:

```
switch -regexp report[0-9]*
```

while

The **while** loop repeats commands. In Tcl, the **while** loop begins with the **while** command and takes two arguments, an expression, and either a single Tcl command or a block of Tcl commands. The expression is encased in braces. A block of Tcl commands begins with an opening brace on the same line as the **while** command. Then, on following lines are the Tcl commands that will be repeated in the loop. The block ends with a closing brace, usually on a line by itself. The syntax for the **while** loop with a single statement is described here:

```
while {expression } {
Tcl commands
}
```

for

The **for** loop performs the same tasks as the **while** loop. However, it has a different format. The **for** loop takes four arguments, the first three of which are expressions and the last of which is a block of Tcl commands. The first three arguments are expressions that incorporate the initialization, test, and increment components of a loop. These expressions are each encased in braces. The last argument, the block of Tcl commands, begins with an opening brace and then continues with Tcl commands on the following lines, ending with a closing brace:

```
for {expression1} {expression2} {expression3} {
Tcl commands;
}
```

foreach

The **foreach** structure is designed to sequentially reference a list of values. It is very similar to the C shell's **for-in** structure. The **foreach** structure takes three arguments: a variable, a list, and a block of Tcl commands. Each value in the list is assigned to the variable in the **foreach** structure. Like the **while** structure, the **foreach** structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. As in the **while** loop, the block of Tcl commands is encased in braces. The syntax for the **foreach** loop is described here:

```
foreach variable ( list of values ) {
tcl commands
}
```

Tcl Input and Output: gets and puts

Tcl can read input from the standard input or a file using the **gets** command and write output to the standard output with the **puts** command. The following command

reads a line from the standard input, usually the keyboard. The input is placed in the variable line.

```
gets line
```

The **puts** command outputs a string to the standard output or to a file. It takes as its argument the string to be output.

```
puts $line.
```

gets reads a line into the variable specified as its argument. You can then use this variable to manipulate whatever has been read. For example, you can use **line** in a **puts** command to display what was input.

```
myread

#!/usr/bin/tclsh
gets line
puts "This is what I entered: $line"

$ myread
larisa and aleina
This is what I entered: larisa and aleina
```

You can use the **puts** command to write data to any file or to the standard output. File handle names are placed after the **puts** command and before any data such as strings or variables. If no filename is specified, then **puts** outputs to the standard output.

To output formatted values, you can use the results of a format command as the argument of a **puts** command. **format** performs a string conversion operation on a set of values, formatting them according to conversion specifiers in a format string.

```
puts [format "%s" $myname]
```

If you want to output different kinds of values in a **puts** operation, you can use the **format** command to first transform them into a single string. The **puts** command will only output a single string. In the following example, the contents of the **$firstname** and **$age** variables are output as a single string by first using the **format** command with two string specifiers, **"%s %d"**, to make them one string. **%d** will transform a numeric value into its corresponding character values.

```
puts [format "%s %d" $firstname $age]
```

For string values, you can just make them part of the single string by placing them within double quotes. In this example, **firstname** and **lastname** are made part of the same string:

```
puts "$firstname $lastname"
```

Tcl File Handles

You use the **open** command to create a file handle for a file or pipe (see Table 7 for a list of Tcl file commands). The **open** command takes two arguments, the filename and the file mode, and returns a file handle that can then be used to access the file. The filename argument can be the name of the file or a variable that holds the name of the file. The file mode is the permissions you are opening the file with. This can be **r** for read-only, **w** for write-only, and **a** for append only. To obtain both read and write permission for overwriting and updating a file, you attach a **+** to the file mode. **r+** gives you read and write permission. The syntax for **open** follows:

```
open ( filename-string, file-mode );
```

You would usually use the **open** command in a **set** command so that you can assign the file handle returned by **open** to a variable. You can then use that file handle in that variable in other file commands to access the file. In the next example, the user opens the file **reports** with a file mode for reading, **r**, and assigns the returned file handle to the **myfile** variable.

```
set myfile [open "reports" r]
```

Often, the filename will be held in a variable. You then use the **$** with the variable name to reference the filename. In this example, the filename **reports** is held in the variable **filen**:

```
set myfile [open $filen r]
```

Once you have finished with the file, you close it with the **close** command. **close** takes as its argument the file handle of the file you want to close.

```
close $myfile
```

With the **gets** and **puts** commands, you can use a file handle to read and write from a specific file. **gets** takes two arguments: a file handle and a variable. It will read a line from the file referenced by the file handle and place it as a string in the variable.

If no file handle is specified, then **gets** reads from the standard input. The following command reads a line from a file using the file handle in the **myfile** variable. The line is read into the **line** variable.

```
gets $myfile line
```

The **puts** command also takes two arguments: a file handle and a string. It will write the string to the file referenced by the file handle. If no file handle is specified, then **puts** will write to the standard output. In the following example, **puts** writes the string held in the **line** variable to the file referenced by the file handle held in **myfile**. Notice that there is a **$** before **line** in the **puts** command, but not in the previous **gets** command. **puts** operates on a string, whereas **gets** operates on a variable.

```
puts $myfile $line

myreport

#!/usr/bin/tclsh
 set reps [open "reports" r ]
 while ( gets $reps line)
 {
 puts $line;
 }
 close reps
```

You can use the **file** command to check certain features of files, such as whether they exist or if they are executable. You can also check for directories. The **file** command takes several options, depending on the action you want to take. The **exist** option checks whether a file exists, and the **size** option tells you its size. The **isdirectory** option determines whether the file is a directory, and **isfile** checks to see whether it is a file. With the **executable**, **readable**, and **writable** options, you can detect whether a file is executable, can be read, or can be written to. The **dirname** option displays the full pathname of the file, and the **extension** and **root** name options show the extension or root name of the file, respectively. The **atime**, **mtime**, and **owned** options display the last access time and the modification time, and whether it is owned by the user.

```
file exits reps
file isfile reps
file size reps
file executable myreport
```

Often filenames will be used as arguments to Tcl programs. In this case, you can use the **argv** list to obtain the filenames. The **argv** command lists all arguments entered on the command line when the Tcl script was invoked. You use the **lindex** command to extract a particular argument from the **argv** list. Many programs use filenames as their arguments. Many also specify options. Remember that the **lindex** command indexes a list from 0. So the first argument in the **argv** list would be obtained by the following (be sure to precede **argv** with the **$**):

```
lindex $argv 0
```

You can, if you wish, reference an argument in the **argv** list within the **open** command. Here, the **lindex** operation is enclosed in braces, in place of the filename. The **lindex** command will return the filename from the **argv** list.

```
set shandle [ open {lindex $argv 1} r ]
```

File Access Commands	Description
file	Obtain file information
open	Open a file
close	Close a file
eof	Check for end of file
fcopy	Copy from one file to another
flush	Flush output from a file's internal buffers
glob	Match filenames using **glob** pattern characters
read	Read blocks of characters from a file
seek	Set the seek offset of a file
tell	Return the current offset of a file
socket	Open a TCP/IP network connection

Table 7. *Tcl File Access and Input/Output Commands*

File Access Commands	Description
Input/Output Commands	
`format`	Format a string with conversion specifiers, like `sprintf` in C
`scan`	Read and convert elements in a string using conversion specifiers, like `scanf` in C
`gets`	Read a line of input
`puts`	Write a string to output

Table 7. *Tcl File Access and Input/Output Commands* (continued)

Tk

The Tk application extends Tcl with commands for creating and managing graphic objects such as windows, icons, buttons, and text fields. Tk commands create graphic objects using the X Window System. It is an easier way to program X Window objects than using the X11 Toolkit directly. With Tk, you can easily create sophisticated window-based user interfaces for your programs.

The Tk language is organized according to different types of graphic objects such as windows, buttons, menus, and scroll bars. Such objects are referred to as *widgets.* Each type of widget has its own command with which you can create a widget. For example, you can create a button with the **`button`** command or a window with the **`window`** command. A type of widget is considered a class, and the command to create such a widget is called a class command. The command will create a particular instance of that class, a particular widget of that type. **`button`** is the class command for creating a button. Graphical objects such as buttons and frames are also often referred to as widgets. Table 8 lists the different widgets available in Tk.

Note *Several currently available Tcl/Tk GUI builders are Free Visual Tcl, SpecTcl, VisualGIPSY, and XF. These are freely available, and you can download them from their Web sites or from the Tcl Developer Xchange*

Widget	Description
button	A button
canvas	A window for drawing objects
checkbutton	A check button
entry	An input box
frame	A frame is a simple widget. Its primary purpose is to act as a spacer or container for complex window layouts
image	Create image objects for displaying pictures
label	A label
listbox	A list box with a selectable list of items
menu	A menu bar
menubutton	A menu button to access the menu
message	Create and manipulate message widgets
radiobutton	A radio button
scrollbar	A scroll bar
text	An editable text box
scale	A scale

Table 8. *Standard TK Widgets*

The wish Shell and Scripts

Tk operates under the X Window System. Within the X Window System, Tk uses its own shell, the wish shell, to execute Tk commands. To run Tk programs, you first start up your X-Window System and then start up the wish shell with the command **`wish`**. This will open up a window in which you can then run Tk commands.

You execute Tk commands within the wish shell interactively, entering commands and executing them one by one, or you can place the commands in a script file and execute them all at once. Usually, Tk commands are placed in a script that is then run with the invocation of the **`wish`** command. Like Tcl scripts, Tk scripts usually have the

extension **.tcl**. For example, a Tk script called **mydir.tcl** would be read and executed by the following command entered in an Xterm window:

```
$ wish mydir.tcl
```

To create a standalone script that operates more like a command, you need to invoke the **wish** command within the script. Ordinarily the **wish** command will open an interactive Tk shell window whenever executed. To avoid this, you should invoke **wish** with the **-f** option.

```
#!/usr/bin/wish -f
```

Note *When creating a standalone script, be sure to change its permissions with the* ***chmod*** *command to allow execution. You can then just enter the name of the script to run the program.*

```
$ chmod 755 mydir1
$ ./mydir1
```

Tk Widgets

Tk programs consist of class commands that create various graphic widgets. The class command takes as its arguments the name you want to give the particular widget followed by configuration options with their values (see Table 9). Tk commands have a format similar to Tcl. You enter a Tk class command on a line, beginning with the name of the command followed by its arguments. Tk commands are more complicated than Tcl commands. Graphic interface commands require a significant amount of information about a widget to set it up. For example, a button requires a name, the text it will display, and the action it will take.

Many Tk commands can take various options indicating different features of a widget. Table 10 lists several options commonly used for Tk widgets. In the following example, a button is created using the **button** command. The **button** command takes as its first argument the name of the button widget. Then, different options define various features. The **-text** option is followed by a string that will be the text displayed by the button. The **-command** option is followed by the command that the button executes when it is clicked. This **button** command will display a button with the text "Click Me". When you click it, the Tk shell will exit.

```
button .mybutton -text "Click Me" -command exit
```

Event Operations	Description
Bind	Associate Tcl scripts with X events
Bindtags	Bind commands to tags
Selection	Object or text selected by mouse
Geometry Managers	
Pack	Pack widgets next to each other
Place	Place widgets in positions in frame
Grid	Place widgets in a grid of rows and columns
Window Operations	
Destroy	Close a TK window
Toplevel	Select the top-level window
Wm	Set window features
Uplevel	Move up to previous window level

Table 9. *Tk Commands*

Button	Description
`-activebackground`	Specifies background color to use when drawing active elements
`-activeborderwidth`	Width of the 3-D border drawn around active elements
`-activeforeground`	Foreground color to use when drawing active elements
`-anchor`	How information is displayed in the widget; must be one of the values n, ne, e, se, s, sw, w, nw, or center
`-background`	The normal background color to use when displaying the widget

Table 10. *Tk Commonly Used Standard Options*

Button	Description
`-font`	The font to use when drawing text inside the widget
`-foreground`	The normal foreground color to use when displaying the widget
`-geometry`	Specifies the desired geometry for the widget's window
`-image`	Specifies an image to display in the widget
`-insertbackground`	Color to use as background in the area covered by the insertion cursor
`-insertborderwidth`	Width of the 3-D border to draw around the insertion cursor
`-insertofftime`	Number of milliseconds the insertion cursor should remain "off" in each blink cycle
`-relief`	Specifies the 3-D effect desired for the widget
`-selectbackground`	Specifies the background color to use when displaying selected items
`-text`	String to be displayed inside the widget
Button Options	
`-command`	Specifies a Tcl command to associate with the button
`-selectimage`	Image to display when the check button is selected
`-height`	Height for the button
`-state`	Specifies one of three states for the radio button: normal, active, or disabled
`-variable`	Global variable to set to indicate whether or not this button is selected
`-width`	Width for the button

Table 10. *Tk Commonly Used Standard Options* (continued)

To set up a working interface, you need to define all the widgets you need to perform a given task. Some widgets are designed to manage other widgets; for instance, scroll bars are designed to manage windows. Other widgets, such as text input fields, may interact with a Tcl program. A menu choice may take the action of running part of a Tcl program.

Widgets are organized hierarchically. For example, to set up a window to input data, you may need a frame, within which may be text field widgets as well as buttons. Widget names reflect this hierarchy. The widget contained within another widget is prefixed with that widget's name. If the name of the frame is **report** and you want to call the text input field **monday**, the text input field will have the name **report.monday**. A period separates each level in the hierarchy. A button that you want to call **ok** that is within the **report** frame would be named **report.ok**.

Once you have created your widgets, their geometry has to be defined. The geometry determines the size of each widget in relation to the others, and where they are placed in the window. Tk has three geometry managers, **pack**, **place**, and **grid**. The **pack** command is used in these examples. When you have defined your widgets, you issue a geometry manager command on them to determine their size and shape on the screen.

Note *Your widgets cannot be displayed until their geometry is determined.*

The following determines the geometry of the **.mybutton** widget using the **pack** command:

```
pack .mybutton
```

The **mydir1** program is a simple Tcl/Tk program to display a list of file and directory names in a Tk listbox widget with an attached scroll bar. Figure 1 shows this list box. With a listbox widget, you can display a list of items that you can then easily scroll through. Using the mouse, you can select a particular item. If a scroll bar is attached, you can scroll through the displayed items if there are more than can fit in the designated size of the list box. First the scroll bar is defined using the **scrollbar** command, giving it the name **.scroll** and binding it with the command **.list yview**. This instructs the scroll bar to be attached to the list box on a y-axis, vertical.

```
scrollbar .scroll -command ".list yview"
```

Then, the list box is defined with the **listbox** command, giving it the name **.list** and a y-axis scroll capability provided by the **.scroll** widget. The list box will appear sunken with the specified width and height.

```
listbox .list -yscroll ".scroll set" -relief sunken \
 -width 15 -height 15 -setgrid yes
```

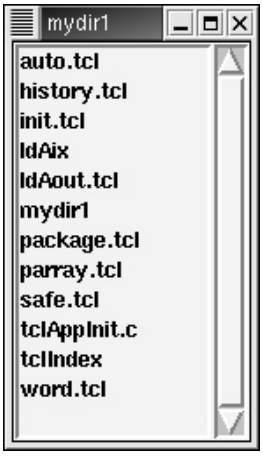

Figure 1. *The mydir1 Tk list box*

The two widgets are then created with the **pack** command and positioned in the window. They are placed on the left side of the window and will always expand to fill the window. The anchor is on the west side of the window, **w**. The list box, **.list**, is placed first, followed by the scroll bar, **.scroll**.

```
pack .list .scroll -side left -fill both -expand yes -anchor w
```

A Tcl **if** test then follows that checks if the user entered an argument when the program was invoked. The **if** test checks to see if there is a first element in the **argv** list where any arguments are held. If there are no arguments, the current directory is used, as represented by the period. This chosen directory is assigned to the **dir** variable. A Tcl **foreach** operation is then used to fill the list box. The shell **ls** command, as executed with the **exec** command, obtains the list of files and directories. Each is then placed in the list box with the Tk **insert** operation for the **.list** widget. The **insert** command takes a position and a value. Here, the value is a filename held in **$i** that is placed at the **end** of the list.

```
.list insert end $i
```

The CTRL-C character is then bound to the **exit** command to allow you to easily close the window. A listing of the **mydir1** program follows.

mydir1

```
#!/usr/bin/wish -f
# Create a scroll bar and listbox
scrollbar .scroll -command ".list yview"
listbox .list -yscroll ".scroll set" -relief sunken -width 15 -height
15 -setgrid yes
pack .list .scroll -side left -fill both -expand yes -anchor w
# If user enters a directory argument use that, otherwise use current
directory.
if {$argc > 0} then {
 set dir [lindex $argv 0]
 } else {
 set dir "."
 }
# Fill the listbox (.list) with the list of files and directories
obtained from ls
 cd $dir
 foreach i [exec ls -a ] {
 if [file isfile $i] {
 .list insert end $i
 }
 }
# Set up bindings for the file manager. Control-C closes the window.
bind all <Control-c> {destroy .}
```

To run the **mydir1** program, first make it executable using the **chmod** command to set the executable permissions, as shown here:

```
chmod 755 mydir1
```

Then, within a terminal window on your desktop or window manager, just enter the **mydir1** command at the prompt. You may have to precede it with a **./** to indicate the current directory.

```
./mydir1
```

A window will open with a list box displaying a list of files in your current working directory (see Figure 1). Use the scroll bar to display any filenames not shown. Click the window Close box to close the program.

Events and Bindings

A Tk program is event driven. Upon running, it waits for an event such as a mouse event or a keyboard event. A mouse event can be a mouse click or a double-click, or

even a mouse down or up. A keyboard event can be a CTRL key or meta key, or even the ENTER key at the end of input data. When the program detects a particular event, it takes an action. The action may be another graphical operation such as displaying another menu, or it may be a Tcl, Perl, or shell program.

Bindings are the key operational component of a Tk program. Bindings detect the events that drive a Tk program. You can think of a Tk program as an infinite loop that continually scans for the occurrence of specified events (bindings). When it detects such an event, such as a mouse click or control key, it executes the actions bound to that event. These actions can be any Tcl/Tk command or series of commands. Usually, they call functions that can perform complex operations. When finished, the program resumes its scanning, looking for other bound events. This scanning continues indefinitely until it is forcibly broken by an **`exit`** or **`destroy`** command, as is done with the CTRL-C binding. You can think of bindings as multiple entry points where different parts of the program begin. It is not really the same structure as a traditional hierarchical sequential program. You should think of a binding as starting its own sequence of commands, its own program. This means that to trace the flow of control for a Tk program, you start with the bindings. Each binding has its own path, its own flow of control.

Actions are explicitly bound to given events using the **`bind`** command. The **`bind`** command takes as its arguments the name of a widget or class, the event to bind, and the action to bind to that event. Whenever the event takes place within that widget, the specified action is executed.

```
bind .myframe <CTRL-H> {.myframe delete insert }
```

You use the **`bind`** command to connect events in a Tk widget with the Tcl command you want executed. In a sense, you are dividing your Tcl program into segments, each of which is connected to an event in a Tk widget. When an event takes place in a Tk widget, its associated set of Tcl commands is executed. Other Tk commands, as well as Tcl commands, can be associated with an event bound to a Tk widget. This means that you can nest widgets and their events. The Tcl commands executed by one Tk event may, in turn, include other Tk commands and widgets with events bound to yet other Tcl commands.

Expect

Expect has several commands that you can use to automatically interact with any Unix program or utility that prompts you for responses. For example, the login procedure for different systems using FTP or telnet can be automatically programmed with Expect commands. Expect is designed to work with any interactive program. It waits for a response from a program and will then send the response specified in its script. You can drop out of the script with a simple command and interact with the program directly.

Three basic Expect commands are the **send**, **expect**, and **interact** commands. The **expect** command will wait to receive a string or value from the application you are interacting with. The **send** command will send a string to that application. The **interact** command places you into direct interaction with the application, ending the Expect/Tcl script. In the following script, Expect is used to perform an anonymous login with FTP. The **spawn** command starts up the FTP program. The Internet address of the FTP site is assumed to be an argument to this script, and as such will be held in the **argv** list. In place of **$argv**, you could put the name of a particular FTP site. The **myftp.expect** script that follows will set up an ftp connection automatically.

myftp.expect

```
#!/usr/bin/expect
spawn ftp
send "open $argv\r"
expect "Name"
send "anonymous\r"
expect "word:"
send "richlp@turtle.mytrek.com\r"
interact
```

To run Expect commands, you have to first enter the Expect shell. In the previous **myftp.expect** script, the Expect shell is invoked with the command **#!/usr/bin/expect**. Be sure to add execute permission with **chmod** 755 `myftp.expect`:

```
$myftp ftp.calderasystems.com
```

The **expect** command can take two arguments: the pattern to expect and an action to take if the pattern is matched. **expect** can also take as its argument a block of pattern/action arguments. In this case, **expect** can match on alternative patterns, executing the action only for the pattern it receives. For example, the **ftp** command may return a "connection refused" string instead of a "name" string. In that case, you would want to issue this message and exit the Expect script. If you want more than one action taken for a pattern, you can encase them in braces, separated by semicolons.

Another useful Expect command is **timeout**. You can set the **timeout** command to a number of seconds, then have Expect check for the **timeout**. To set the number of seconds for a **timeout**, you use **set** to assign it to the **timeout** variable (the default is 10 seconds). To have the **expect** command detect a **timeout**, you use the word **timeout** as the **expect** command's pattern. With the **timeout**, you can add an action to take. An example of an Expect script follows:

```
set timeout 20
end "open $argv\r"
expect {
```

```
 timeout {puts "Connection timed out\n"; exit }
 "Connection refused" {puts "Failed to connect\n"; exit}
 "Unknown host" {puts "$argv is unknown\n"; exit}
 "Name"
}
```

Expect can run with any kind of program that requires interaction. All you need to do is to determine the sequence of prompts and responses you want.

Gawk

Gawk is a programming language designed to let Linux users create their own shell filters. A filter operates within a Linux shell such as BASH or TCSH. It reads information from an input source such as a file or the standard input, modifies or analyzes that information, and then outputs the results. Results can be a modified version of the input or an analysis. For example, the **sort** filter reads a file and then outputs a sorted version of it, generating output that can be sorted alphabetically or numerically. The **wc** filter reads a file and then calculates the number of words and lines in it, outputting just that information. The **grep** filter will search a file for a particular pattern, outputting the lines the pattern is found on. With Gawk, you can design and create your own filters, in effect creating your own Linux commands. You can instruct Gawk to simply display lines of input text much like **cat**, or to search for patterns in a file like **grep**, or even count words in a file like **wc**. In each case, you could add your own customized filtering capabilities. You could display only part of each line, or search for a pattern in a specific field, or count only words that are capitalized. This flexibility lets you use Gawk to generate reports, detecting patterns and performing calculations on the data.

You can use Gawk directly on the shell command line, or you can place Gawk within a shell file that you can then execute. The name of the shell file can be thought of as a new filter that you have created. In effect, with Gawk, you can define your own filters. In this sense there are two ways of thinking about Gawk. Gawk is itself a filter that you can invoke on the command line like any other filter, and Gawk is a programmable filter that you can use to create your own filters. This section will examine both aspects of Gawk. First we will examine Gawk as a filter, with all its different features. Then, we will see how you can use Gawk to define your own filters.

The Gawk utility has all the flexibility and complexity of a programming language. Gawk has a set of operators that allow it to make decisions and calculations. You can also declare variables and use them in control structures to control how lines are to be processed. Many of the programming features are taken from the C programming language and share the same syntax. All of this makes for a very powerful programming tool.

Gawk is the GNU version of the Unix Awk utility. Awk was originally created as a standard utility for the Unix operating system. One of its creators is Brian Kernighan, who developed the Unix operations system. An enhanced version of Awk called Nawk was developed later to include file handling. With Nawk, you can access several files in the same program. Gawk is a further enhancement, including the added features of Nawk as well as the standard capabilities of Awk.

Gawk has a full set of arithmetic operators. You can perform multiplication, division, addition, subtraction, and modulo calculations. The arithmetic operators are the same as those used in the C programming language and Perl. Gawk also supports both scalar and associative arrays. Gawk has control structures similar to those in the C programming language, as well as pattern matching and string functions similar to those in Perl and Tcl/Tk. You can find out more about Gawk at **www.gnu.org/software/gawk**.

The gawk Command

The **gawk** command takes as its arguments a Gawk instruction and a list of filenames. The Gawk instruction is encased in single quotes and is read as one argument. The Gawk instruction itself consists of two segments: a pattern and an action. The action is enclosed in brackets. The term "pattern" can be misleading. It is perhaps clearer to think of the pattern segment as a condition. The pattern segment can be either a pattern search or a test condition of the type found in programming languages. The Gawk utility has a full set of operators with which to construct complex conditions. You can think of a pattern search as just one other kind of condition for retrieving records. Instead of simply matching patterns as in the case of **grep**, the user specifies a condition. Records that meet that condition are then retrieved. The actions in the action segment are then applied to the record. The next example shows the syntax of a Gawk instruction, which you can think of as **condition** {action}:

```
pattern {action}
```

The Gawk utility operates on either files or the standard input. You can list filenames on the command line after the instruction. If there are no filenames listed, input is taken from the standard input. The example below shows the structure of the entire Gawk instruction. The invocation of Gawk consists of the **gawk** keyword followed by a Gawk instruction and filenames. As with the **sed** commands, the instruction should be placed within single quotes to avoid interpretation by the shell. Since the condition and action are not separate arguments for Gawk, you need to enclose them both in one set of quotes. The next example shows the syntax of a **gawk** command:

```
$ gawk 'pattern action { }' filenames
```

You can think of the pattern in a Gawk instruction as referencing a line. The Gawk action is then performed on that line. The next two examples below print all lines with the pattern "Penguin". The pattern segment is a pattern search. A pattern search is denoted by a pattern enclosed in slashes. All records with this pattern are retrieved. The action segment in the first example contains the **print** command. The **print** command outputs the line to the standard output.

books

```
Tempest Shakespeare 15.75 Penguin
Christmas Dickens 3.50 Academic
Iliad Homer 10.25 Random
Raven Poe 2.50 Penguin

$ gawk '/Penguin/{print}' books
Tempest Shakespeare 15.75 Penguin
Raven Poe 2.50 Penguin
```

Tip

Both the action and pattern have defaults that allow you to leave either of them out. The print action is the default action. If an action is not specified, the line is printed. The default pattern is the selection of every line in the text. If the pattern is not specified, the action is applied to all lines.

In the second example, there is no action segment. The default action is then used, the **print** action.

```
$ gawk '/Penguin/' books
Tempest Shakespeare 15.75 Penguin
Raven Poe 2.50 Penguin
```

Pattern Searches and Special Characters

Gawk can retrieve lines using a pattern search that contains special characters. The pattern is designated with a beginning and ending slash, and placed in the pattern segment of the Gawk instruction.

```
/pattern/ {action}
```

The pattern search is performed on all the lines in the file. If the pattern is found, the action is performed on the line. In this respect, Gawk performs very much like an editing operation. Like **sed**, a line is treated as a line of text and the pattern is searched

for throughout the line. In the next example, Gawk searches for any line with the pattern "Poe". When a match is found, the line is output.

```
$ gawk '/Poe/{print}' books
Raven Poe 2.50 Penguin
```

You can use the same special characters for Gawk that are used for regular expressions in the **sed** filter and the Ed editor. The first example below searches for a pattern at the beginning of the line. The special character **^** references the beginning of a line. The second example searches for a pattern at the end of a line using the special character **$**:

```
$ gawk '/^Christmas/{print}' books
Christmas Dickens 3.50 Academic

$ gawk '/Random$/{print}' books
Iliad Homer 10.25 Random
```

As in Ed and **sed**, you can use special characters to specify variations on a pattern. The period matches any character, the asterisk matches repeated characters, and the brackets match a class of characters: **.**, *****, and **[]**. In the first example below, the period is used to match any pattern in which a single character is followed by the characters "en":

```
$ gawk '/.en/{print}' books
Tempest Shakespeare 15.75 Penguin
Christmas Dickens 3.50 Academic
Raven Poe 2.50 Penguin
```

The next example uses the brackets and asterisk special characters to specify a sequence of numbers. The set of possible numbers is represented by the brackets enclosing the range of numeric characters [0–9]. The asterisk then specifies any repeated sequence of numbers. The context for such a sequence consists of the characters ".50". Any number ending with .50 will be matched. Notice that the period is quoted with a backslash to treat it as the period character, not as a special character.

```
$ gawk '/[0-9]*\.50/ {print}' books
Christmas Dickens 3.50 Academic
Raven Poe 2.50 Penguin
```

Gawk also uses the extended special characters: **+**, **?**, and **|**. The **+** and **?** are variations on the ***** special character. The **+** matches one or more repeated instances of a character. The **?** matches zero or one instance of a character. The **|** provides alternative

patterns to be searched. In the next example, the user searches for a line containing either the pattern "Penguin" or the pattern "Academic":

```
$ gawk '/Penguin|Academic/ {print}' books
Tempest Shakespeare 15.75 Penguin
Christmas Dickens 3.50 Academic
Raven Poe 2.50 Penguin
```

Variables

Gawk provides for the definition of variables and arrays. It also supports the standard kind of arithmetic and assignment operators found in most programming languages such as C. Relational operators are also supported.

In Gawk, there are three types of variables: field variables, special Gawk variables, and user-defined variables. Gawk automatically defines both the field and special variables. The user can define his or her own variables. You can also define arithmetic and string constants. Arithmetic constants consist of numeric characters, and string constants consist of any characters enclosed within double quotes.

Field variables are designed to reference fields in a line. A field is any set of characters separated by a field delimiter. The default delimiter is a space or tab. As with other database filters, Gawk numbers fields from 1. This is similar to the number used for arguments in shell scripts. Gawk defines a field variable for each field in the file. A field variable consists of a dollar sign followed by the number of the field. **$2** references the second field. The variable **$0** is a special field variable that contains the entire line.

Tip

A variable may be used in either the pattern or action segment of the Gawk instruction. If more than one variable is listed, they are separated by commas. Notice that the dollar sign is used differently in Gawk than in the shell.

In the next example, the second and fourth fields of the books file are printed out. The **$2** and **$4** reference the second and fourth fields.

books

```
Tempest Shakespeare 15.75 Penguin
Christmas Dickens 3.50 Academic
Iliad Homer 10.25 Random
Raven Poe 2.50 Penguin

$ gawk '{print $2, $4}' books
Shakespeare Penguin
Dickens Academic
Homer Random
Poe Penguin
```

In the next example, the user outputs the line with the pattern "Dickens" twice—first reversing the order of the fields and then with the fields in order. The **$0** is used to output all the fields in order, the entire line.

```
$ gawk '/Dickens/ {print $4, $3, $2, $1; print $0}' books
Academic 3.50 Dickens Christmas
Christmas Dickens 3.50 Academic
```

Gawk defines a set of special variables that provide information about the line being processed. The variable **NR** contains the number of the current line (or record). The variable **NF** contains the number of fields in the current line. There are other special variables that hold the field and record delimiters. There is even one, **FILENAME**, that holds the name of the input file. The Gawk special variables are listed in Table 11.

Both special variables and user-defined variables do not have a dollar sign placed before them. To use such variables, you only need to specify their name. The next example combines both the special variable **NR** with the field variables **$2** and **$4** to print out the line number of the line followed by the contents of fields two and four. The **NR** variable holds the line number of the line being processed.

```
$ gawk '{print NR, $2, $4}' books
1 Shakespeare Penguin
2 Dickens Academic
3 Homer Random
4 Poe Penguin
```

Variables	Description
NR	Record number of current record
NF	Number of fields in current record
$0	The entire current record
$n	The fields in the current record, numbered from 1—for example, **$1**
FS	Input field delimiter; default delimiter is space or tab
FILENAME	Name of current input file

Table 11. *Gawk Special Variables*

You can also define your own variables, giving them any name you want. Variables can be named using any alphabetic or numeric characters as well as underscores. The name must begin with an alphabetic character. A variable is defined when you first use it. The type of variable is determined by the way it is used. If you use it to hold numeric values, the variable is considered arithmetic. If you use it to hold characters, the variable is considered a string. You need to be consistent in the way in which you use a variable. String variables should not be used in arithmetic calculations and vice versa.

You assign a value to a variable using the assignment operator, **=**. The left-hand side of an assignment operation is always a variable and the right-hand side is the value assigned to it. A value can be the contents of a variable such as a field, special, or other user variable. It can also be a constant. In the next example, the user assigns the contents of the second field to the variable **myfield**:

```
$ gawk '{myfield = $2; print myfield}' books
Shakespeare
Dickens
Homer
Poe
```

By default, Gawk separates fields by spaces or tabs. However, if you want to use a specific delimiter, you need to specify it. The **-F** option allows Gawk to detect a specific delimiter. The **-F** option actually sets a Gawk special variable called **FS**, which stands for field separator. With the **-F** option, you can use any character you want for your delimiter.