Table of Contents

Open Source Network Administration By James Kretchmar
ISBN: 0-13-046210-1 Publisher: Prentice Hall Print Publication Date: 9/22/2003

Chapter 10. Custom Tools

Although many tools are available to aid in the tasks of network administration, there will often be some task for which you need a tool that does not yet exist. In this situation, you can either find a way to live without the tool, pay someone to make it for you, or create the tool yourself.

Most of the tools described in this book are relatively complicated and take a large effort to create. They require a mastery of the language the tool is written in, the ability to design and implement a large project, and experience with network programming. However, you can write a great number of simple tools using languages that were designed for just such a purpose. This chapter presents a brief introduction to two such scripting languages: the Bourne shell and Perl. Both of these languages are in very wide use. The Bourne shell in particular is present on every standard Unix system, and the Perl language is now installed on most modern systems.

In this chapter you will be expected to already have familiarity with programming in some language; most any of the commonly used languages will serve as a sufficient background for you to quickly pick up the basics of Perl and the Bourne shell.

10.1. Basics of Scripting

The world of programming languages can be divided in to two major categories: compiled languages and interpreted languages.[1] The source for a compiled program is stored in one or more files, and then a special program called the **compiler** translates those files into instructions for the processor. Those instructions are stored in a binary file. The text files can be thrown away if you wish; the binary file is all that is required to execute the program. The program has to be compiled only once, at what’s called “compile time.” After that the binary executable can be run repeatedly.

[1] Actually, the distinction between compiled and interpreted does not have to be inherent in the language itself, just to the way the program is run. Some languages can be either compiled or interpreted, and yet others, like Java, are somewhere in between. Typically, though, a given language tends to be considered a compiled language or an interpreted language based on the most common use.

Most of the programs you run on a Unix system are compiled programs. You can check with the `file` command:

```
Solaris% file /usr/bin/date
/usr/bin/date:  ELF 32-bit MSB executable SPARC Version 1, ...
```

The text “ELF” and “SPARC” are tipoffs that the program is compiled.

An interpreted program, in contrast, is stored in one more files that are fed to a program called an **interpreter**, which decodes the program lines and executes the appropriate instructions on the fly. The program is interpreted every time it is run, and both the interpreter and script file must be present each time you wish to run the program. The term “script” or “shell script” always implies an interpreted language. The Perl and Bourne shell languages described here are both interpreted languages.

If you run the `file` program on a script, it will produce output like this:

```
Solaris% file test.pl
test.pl:       executable /usr/bin/perl script
```

What are the practical differences between compiled and interpreted languages? Compiled languages tend to run faster while interpreted languages are usually slower. Compiled languages produce a binary file that can be run on only machines of the same processor architecture and operating system, whereas an interpreted file can run on any machine. If the tool you need requires a good deal of performance (speed, memory, or anything else), a script is probably not the best way to create it. But if you need a low-performance program that can be created quickly and run anywhere, a script is just the right thing.

10.1.1. Running a Script

On Unix operating systems, there are two ways to cause a script to be interpreted. One is to invoke the interpreter program, passing it the filename of the script to be interpreted. For example, if you have a file called `testscript` with the following one line:

```
echo "Hello world"
```

it can be run as:

```
Solaris% sh testscript
Hello world
```

Note that `sh` is the interpreter program for the Bourne shell. In this example, the `sh` program reads the file `testscript` and takes action based on its contents. It would be nice if a script could be run like a normal program, though, without requiring you to specify the name of the interpreter first. This way, people using the script do not need to remember which language it was written in and they can type one less command. The operating system will let you do this if you place a special syntax at the beginning of the file. If `testscript` contains the two lines:

```
#!/bin/sh
echo "Hello world"
```

and you give it execute permissions:

```
Solaris% chmod u+x testscript
```

it can now be run directly:

```
Solaris% ./testscript
Hello world
```

When the kernel tries to execute `testscript`, it notices the `#!` as the first two characters, and upon finding them, it runs the interpreter named later on the line, feeding it all following lines of the file. It is also perfectly acceptable to give arguments to the interpreter on that special line:

```
#!/bin/sh -n
echo "Hello World"
```

The `-n` option is used simply for illustration; it instructs the Bourne shell to read the commands but not execute any of them. This is useful for checking the syntax of a script.

In many languages, including the Bourne shell and Perl, any line beginning with `#` is considered a comment and ignored by the interpreter. The first line is always treated as a special case by the operating system. Be aware that if you create an executable script file but do not include an initial `#!` line, the operating system will default to using the Bourne shell interpreter. You should not engage in this practice yourself, but you should at least recognize it when you come across it. For example, operators will sometimes give execute privileges to system startup scripts so that they can be invoked without the initial `sh` on the command line.

Open Source Network Administration By James Kretchmar
ISBN: 0-13-046210-1 Publisher: Prentice Hall Print Publication Date: 9/22/2003

10.1.2. Naming Conventions

Often you will find that scripts are named with a suffix that reflects the language the script is written in. Bourne shell scripts commonly end in `.sh` and Perl scripts commonly end in `.pl`. This is not a requirement, and just as often, you may find scripts named without a suffix.

10.1.3. Local and Environment Variables

Every running program has an "environment" associated with it. The environment is a list of variables and their values. The command `env` will print the current environment:

```
Linux% env
PWD=/var/tmp/
XUSERFILESEARCHPATH=/usr/athena/lib/X11/app-defaults/%N
PAGER=less
VERBOSELOGIN=1
...
```

Scripting languages typically have a way to modify environment variables, but it is important to understand the distinction between modifying an environment variable and modifying a variable local to the program. If you modify an environment variable, the value will be passed on in the environment of other programs run from your script. If you modify a local variable, it has no impact on other programs.

10.2. The Bourne Shell

The Bourne shell serves a twofold purpose, as most shells do. It can be used as a command line interpreter, just like `bash` (Bourne-again shell) and `tcsh`, and it can be used to write simple programs. Of course, `bash` and `tcsh` can also be used for simple programming, but the Bourne shell is the preferred language. Though `csh` and `tcsh` were designed with a syntax similar to that in the C programming language, certain idiosyncrasies can lead to problems with even moderately interesting scripts. The Bourne shell and `bash` are closely related, and most programs written for one will work with the other. On many Linux systems, the Bourne shell program is really just a symlink to the `bash` binary anyway. Regardless, if you write a program with the Bourne shell in mind, it will work even on systems that do not have the `bash` program.

10.2.1. Basics of the Bourne Shell

Fundamentally, a Bourne shell script is a list of commands just as you would type on the command line. Unless the command is a reserved keyword, the shell uses the PATH environment variable to locate the command and then it is executed. For example the script:

```
#!/bin/sh
hostname
date
uptime
who
```

would run the commands `hostname`, `date`, `uptime`, and `who`, printing the output of each to the screen. If a command is not in your path, you can either specify an explicit path or modify the PATH environment variable. Thus:

Open Source Network Administration By James Kretchmar
ISBN: 0-13-046210-1 Publisher: Prentice Hall Print Publication Date: 9/22/2003

```
#!/bin/sh
/usr/local/bin/myprogram
```

and

```
#!/bin/sh
PATH=${PATH}:/usr/local/bin; export PATH
myprogram
```

are both acceptable solutions.

Multiple commands can be listed on a single line if they are separated by semicolons. The preceding example could have been written as:

```
#!/bin/sh
hostname; date; uptime; who
```

10.2.2. Using Variables

Here is an example of setting and using a variable in the Bourne shell:

```
#!/bin/sh
a=foobar
echo $a
```

Notice that when a variable is set, no dollar sign is used, but when the variable is referenced, the dollar sign is used.

Because of the way quoting works in the Bourne shell, you will need to use quotation marks if you want to set a variable to a string with spaces in it:

```
#!/bin/sh
a="This is a test"
echo $a
```

But note that even though the variable `a` contains all four words and the spaces between them, when the variable is given to the `echo` command, each word is given as a separate argument. That is, the `echo` command is called here with four arguments. If instead you need to pass an argument containing spaces as a *single* argument to a command, you must use quotation marks when you reference the variable as well as when you set it. For example, if you want to use the `grep` command to search for the text "network administration," including the space, in the file `book.tex` it would look like:

```
#!/bin/sh
a="network administration"
grep "$a" book.tex
```

Here, only two arguments are passed to `grep`: first the search text and then the file name.

10.2.3. Local and Environment Variables

The Bourne shell has a slightly odd way of dealing with environment variables. The variable `a`, set and referenced above, is a local variable. Environment variables are accessed exactly the same way local variables are, except that they have the property of already being set for you. So:

```
#!/bin/sh
echo $EDITOR
```

prints the value of the EDITOR environment variable. Notice the convention that environment variable names are in uppercase while local variable names are in lowercase. This is not enforced but it is highly recommended so that you do not confuse others who read the program later. How did you know the EDITOR variable already had a value by virtue of its being in the environment? Only by convention. It is possible to figure out what variables have values set from the environment, but it is not usually necessary or worthwhile.

So does setting the value of an environment variable work just as above? The answer is no. The EDITOR variable as you accessed it is not really an environment variable at all. If you set its value with

```
EDITOR=vi
```

it will change the value of $EDITOR, but it is really only a local variable. This can be demonstrated with this script:

```
#!/bin/sh
echo $EDITOR
EDITOR=vi
/usr/bin/env | grep EDITOR
```

When run, it produces:

```
Solaris% ./testscript.sh
emacs
EDITOR=emacs
```

The value of the EDITOR environment variable is emacs even though you changed the value of the local variable to “vi.” The reason for this is that when the Bourne shell starts up, *every* variable is a local variable, and it places a copy of all environment variable *values* into a local variable of the same name.

You can declare a variable to be an environment variable with the `export` command. Once you do that, that variable is bound to the environment, meaning that any value you set for it will automatically be reflected in the environment and thus passed on to other programs. If we modify the preceding example by adding an `export` command:

```
#!/bin/sh
echo $EDITOR
EDITOR=vi
export EDITOR
/usr/bin/env | grep EDITOR
```

the output now becomes:

```
Solaris% ./testscript.sh
emacs
EDITOR=vi
```

Often in scripts you will see the export command placed on the same line as the variable is set, either with a semicolon:

```
EDITOR=vi; export EDITOR
```

or without:

```
EDITOR=vi export EDITOR
```

Both are valid Bourne shell syntax.

10.2.4. Exit Status

After a program is finished running, it returns an integer as its **exit status**. Often a program returns an exit status of zero if it exits normally and an exit status of one if it does not. This is not always the case, though; check the documentation for the program in question to find out its behavior.

In the Bourne shell, the return value of the last program run is stored in the variable `$?`. For example, this short script uses the `$?` variable to print the exit status from the `touch` program. The `touch` program tries to create a file if it does not exist; if it does already exist, it updates the modification time:

```
#!/bin/sh
touch /tmp/mytestfile
echo $?
touch /foobar
echo $?
```

When you run it, the result is:

```
Solaris% ./testscript.sh
0
touch: /foobar cannot create
1
```

The first use of the `touch` program exits normally with exit status zero. In the second instance, `touch` is not able to create the file `/foobar`, so a warning is printed and the exit status is set to one.

10.2.5. Conditionals

This script demonstrates the use of conditionals in the Bourne shell:

```
#!/bin/sh
a="green"
if [ "$a" = "red" ]; then
  echo "Found red"
elif [ "$a" = "green" ]; then
  echo "Found green"
else
  echo "Found a color other than red or green"
fi
```

Note that some of the spacing is very important. There must be a space after the open bracket, before the closed bracket, and both before and after the equal signs. Placing quotes around `$a`, while not strictly necessary in this example, is a good habit to develop. If the value of `$a` had been empty (`""`), the quotes would be required.

The `if` statement begins the set of conditionals. The `elif` command stands for “else if.” The `else` statement catches any conditions not already met, and finally the `fi` (“if” backwards) command closes the set of conditionals.

There are a number of interesting details to be studied in how these statements are constructed and in shortcuts that can be taken. But the format above is clear and simple and will work wherever you need a conditional. However, it is worth mentioning that the brackets are not really Bourne shell syntax. The open bracket (`[`) is actually a separate program, usually a symlink to the `test` program:

```
Linux% ls -l /usr/bin/[
lrwxrwxrwx    1 root     root    4 Jul  1  2002 /usr/bin/[ -> test*
```

In the previous example, you compared strings to see if they were equal. Other tests are also possible, such as comparing numeric values:

```
#!/bin/sh
a=75
if [ $a -eq 0 ]; then
  echo "Zero"
elif [ $a -le 50 ]; then
  echo "Less than or equal to 50"
elif [ $a -lt 80 ]; then
  echo "Less than 80"
elif [ $a -ge 80 ]; then
  echo "Greater than or equal to 80"
fi
```

Also useful is the ability to test files. For example, you can use `-r` to test if a file is readable:

```
#!/bin/sh
a=/var/tmp/myfile
if [ ! -r "$a" ]; then
  echo "Warning: $a is not readable"
fi
```

The exclamation mark negates the test, which is to say that it will succeed when the opposite condition is true. Using `-r` alone will succeed when the file is readable; using `! -r` will succeed when the file is *not* readable.

The `and` and `or` operators are `-a` and `-o`, respectively:

```
#!/bin/sh
a=1
b=1
if [ $a -eq 1 -a $b -eq 1 ]; then
  echo "Both"
elif [ $a -eq 1 -o $b -eq 1 ]; then
  echo "One"
else
  echo "Neither"
fi
```

The full list of tests is available on the `test` man page (type `man test` at the prompt). On some systems, this man page lists tests for a number of different shells, so make sure to look at the ones for `/bin/sh`, or if that is not present, the tests for the `bash` shell.

Do note that conditionals are particularly useful in conjunction with the status variable:

```
#!/bin/sh
touch /foobar
if [ $? -ne 0 ]; then
  echo "Warning: touch failed"
fi
```

10.2.6. Arguments

Often you will want use arguments from the command line as variables in your script. Say you have a script that tests a server for a security vulnerability. You would like to be able to invoke it as:

```
Solaris% ./scanhost myserver.example.com
```

In the Bourne shell, the arguments are stored in order as the variables `$1`, `$2`, `$3`, and so on. The variable `$0` stores the name of the script itself, as invoked from the command line, and the variables `$*` and `$@` store all of the command line arguments separated by spaces.[2] Our vulnerability testing program above might begin with:

[2] The difference between these two variables is in how they behave when quoted.

```
#!/bin/sh
if [ "$1" = "" ]; then
   echo "You must supply a hostname"
   exit 1
fi
host="$1"
echo "Scanning $host"
```

Arguments can also be manipulated with the `shift` command. It will throw out the value of `$1`, store `$2` in `$1`, store `$3` in `$2`, and so on. A common example of argument processing using the `shift` command is presented in the next section, on Bourne shell loops.

10.2.7. Loops

The Bourne shell supports `for` loops as follows:

```
#!/bin/sh
servers="mail1.example.com mail2.example.com time.example.com"
for i in $servers; do
   echo $i
done
```

Support for `while` loops is also included. In this example, we use a while loop to perform argument processing:

```
#!/bin/sh
quiet=0
verbose=0
while [ "$1" != "" ]; do
  if [ "$1" = "-q" ]; then
     quiet=1
     shift
  elif [ "$1" = "-v" ]; then
     verbose=1
     shift
  else
     echo "Invalid argument"
     exit
  fi
done
```

10.2.8. Using Command Output

Many times you will want to use the output of a particular program in your script. Perhaps you wish to store the result of the `date` command in a variable so that it can be used several times later on in the script. You can do so by placing the command in single back quotes:

```
#!/bin/sh
date='date'
echo The date is $date
```

Many scripts use this technique to perform simple changes to text using `sed` or `awk`:

```
gecos='grep "$1" /etc/passwd | awk -F: '{print $5}''
```

Note that `$1` refers to the first argument on the command line, but the `$5` is something else entirely. When single quotes are used in the Bourne shell, as they are in `'print $5'`, variables values are not substituted. Instead, the text string `$5` is passed on to the `awk` program, which uses that syntax to represent the fifth field of text (in this case, the fifth field of the password file).

10.2.9. Working with Input and Output

Input and output redirection is straightforward in the Bourne shell. The standard output is redirected to a file with >*filename*:

```
#!/bin/sh
echo "Starting script at 'date'" > /var/tmp/log
```

This will overwrite an existing file; if you wish to append to the file instead, use >>*filename*:

```
echo "Ending script at 'date'" >> /var/tmp/log
```

You can redirect standard error by placing the number 2 in front of the greater-than sign.[3] The following will redirect both the standard out and standard error to the same file:

[3] It is 2 because that is the file descriptor number of standard error.

```
#!/bin/sh
make > /var/tmp/buildlog 2> /var/tmp/buildlog
```

The standard input can be redirected from a file with a less-than sign:

```
grep $user < /etc/passwd
```

It is also possible to read lines from the standard input and use them in your script. The following script, for example, can be used much like the cat program. Run it as `testscript.sh` *filename*:

```
#!/bin/sh
while read a; do
   echo $a
done < $1
```

The `read` command reads one line from the standard input and stores it in the named variable, here `a`. When the end of the file is reached, `read` returns a non-zero value. This allows the loop above to continue reading lines until no more are present.

10.2.10. Functions

Modern versions of the Bourne shell support user-created functions, though it is still possible to find some old versions that do not. Make sure to test out functions on your system before relying on them.

This program defines and uses a function called `logit`:

```
#!/bin/sh
logit () {
  echo "$*" >> /var/tmp/scriptlog
}
logit "Starting script at 'date'"
# do some work here
logit "Finished script at 'date'"
```

Notice that within a function, the variables `$1`, `$*`, etc., now refer to the arguments passed to the function instead of the arguments passed to the script command line. Functions can also return a value just as a program can. This allows you to check the success or failure of the function. As an example:

```
#!/bin/sh
myfunction () {
  return 1
}
myfunction
echo $?
```

This produces:

```
Solaris% ./testscript.sh
1
```

10.2.11. Other Miscellaneous Items

There are many other features available to you in the Bourne shell, all of which are documented in the manual page. The following sections describe a few odds and ends that you may find useful.

Interpreting Another File

You can cause a Bourne shell script in another file to be interpreted by placing the filename after a period and a space:

```
#!/bin/sh
echo This is my script
. /var/tmp/otherscript
echo Back to my script
```

All the commands in `/var/tmp/otherscript` will then be run just as if they were present our script. Note that this means you do not want `/var/tmp/otherscript` to start with a `#!/bin/sh` line.

Exiting

You can exit a Bourne shell script with the exit command. If you give it an integer argument, that will be the exit status of the program:

```
#!/bin/sh
if [ "$1" = "" ]; then
   echo "You must supply an argument"
   exit 1
fi
```

Traps

Unlike many other simple scripting languages, the Bourne shell lets you catch signals with the `trap` command:

```
#!/bin/sh
trap "echo Interrupted; exit 1" 2
sleep 30
```

This script, if run normally, will simply wait for 30 seconds and then exit. Instead, run the script, but before it finishes, type <ctrl>-C to send it an interrupt signal. The result is this:

```
Solaris% ./testscript.sh
^CInturrupted
```

The `trap` command takes two or more arguments. The first is the command to be run when the signal is caught. In this case, we wish to print "Interrupted" and then exit the script, so these two commands are placed in quotation marks so as to be one argument to the `trap` command. The second and following arguments to `trap` list the signals to be caught. Each signal is listed by number; the number assigned to the interrupt signal (SIGINT) is 2. The full list of signals available, in sequential order starting with signal 1, can be printed from the command line with:

```
Linux% kill -l
HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2 PIPE ALA...
CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH P...
RTMIN RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
```

This list may be different on different systems.

In a more practical setting, you might use the `trap` command to call a function that cleans up temporary files and the like, in case a script is interrupted unexpectedly.

The Process ID

The process ID of the script itself is stored in the variable `$$`. This can be handy in creating temporary files so that they do not conflict with other instances of the script running at the same time.

```
#!/bin/sh
tmpfile=/tmp/scripttmp.$$
```

Comments

Any line beginning with a pound sign (`#`) is a comment line and is ignored by the interpreter. The `#!/bin/sh` line at the beginning is an exception, as described earlier.

10.3. Perl

While the Bourne shell has many strengths, some things it is not particularly good at are manipulating text and dealing with lists and other complicated data structures. A programmer needing to manipulate text from a Bourne shell script usually calls other programs like `sed` and `awk`. But this can quickly become cumbersome. That's when many programmers turn to Perl. Perl is the ultimate language for text manipulation, and it has powerful tools for manipulation of lists and other data structures.

10.3.1. Basics of Perl

Using the Bourne shell, you can simply list Unix commands and they will be executed, unless the word you use happens to be one of the few reserved keywords like `if` or `while`. In Perl, this is not the case. Perl relies less on outside Unix commands, and as a result, it expects you to use Perl syntax by default. For example, instead of using the Unix echo command to print text to the screen, you will use Perl's print function:

```
#!/usr/bin/perl
print "Hello world\n";
```

Perl is often installed as `/usr/bin/perl`, but if it is installed elsewhere on your system, be sure to modify the `#!` line appropriately.

Notice that the line ends with a semicolon. Every line in a Perl program must end with a semicolon; it is not optional as it is in the Bourne shell.[4] Also notice that at the end of the print function, a newline is specified with `\n`. Unlike the Unix `echo` command, the Perl print function does not print a newline at the end by default.

[4] The term "line" here doesn't really mean a line of text. It would be more accurate to say that every command must end with a semicolon. A command can span multiple lines and the semicolon will be only at the end of the last line.

10.3.2. Using Variables

One major difference between Perl and the Bourne shell is that in Perl, variable names are always preceded by a dollar sign, whether they are being set or read. For example:

```
#!/usr/bin/perl
$a=foobar;
print "$a\n";
```

10.3.3. Local and Environment Variables

All variables in Perl are local to Perl. If you wish to access an environment variable, you can do it this way:

```
#!/usr/bin/perl
print "$ENV{HOME}\n";
```

This prints the value of the environment variable HOME. You can set it similarly:

```
$ENV{HOME}="/var/tmp/foobar";
```

The ENV variable here is really a special kind of data structure called a hash, which is described in more detail below.

10.3.4. Conditionals

Conditionals in Perl are similar to those C, except that the “else if” syntax is neither the `else if` used in C nor the `elif` used in the Bourne shell. Instead it is `elsif`:

```
#!/usr/bin/perl
$a=green;
if ($a eq "red") {
    print "Found red\n";
} elsif ($a eq "green") {
    print "Found green\n";
} else {
    print "Found a color other than red or green\n";
}
```

As you can see, strings in Perl are compared with the `eq` operator. Integer comparisons are as:

```
#!/usr/bin/perl
$a=75
if ($a == 0); then
    print "Zero\n";
} elsif ($a <= 50) {
    print "Less than or equal to 50\n";
} elsif ($a < 80) {
    print "Less than 80\n";
} elsif ($a >= 80) {
    print "Greater than or equal to 80\n";
}
```

Perl can also perform tests on files:

```
#!/usr/bin/perl
if (-r "/etc/passwd")  {
    print "Readable\n";
} else {
    print "Not readable\n";
}
```

The full list of tests is available in the Alphabetical Listing of Perl Functions section of the `perlfunc` man page.

The syntax for the `and` and `or` operators is `&&` and `||`, respectively:

```
#!/usr/bin/perl
if ( (12>5) && (3<2) ) {
    print "I bet this won't happen.\n";
}
if ( (12>5) || (3<2) ) {
    print "I bet this will.\n";
}
```

10.3.5. Text Manipulation

An in-depth discussion of the text manipulation capabilities of Perl is outside the scope of this book, but a sampling of the basic features gives you a good starting point. Perl uses regular expressions in a similar manner to `sed` and `awk`. The Perl operator for performing regular expression functions is =~ (an equal sign followed by a tilde). For example:

```
#!/usr/bin/perl
$a="foobar";
if ($a=~/oob/) {
    print "Contains oob\n";
} else {
    print "Does not contain oob\n";
}
```

The expression `$a=~/oob/` evaluates true when the string "oob" is present within the text of `$a`. It is also possible to replace text using *`s/text/replacement/`* syntax:

```
#!/usr/bin/perl
$a="foobar";
$a=~s/oo/abcde/;
print $a."\n";
```

The result of running this program is:

```
Solaris% ./test.pl
fabcdebar
```

This program also demonstrates another useful tool of text manipulation: Placing a period between two strings concatenates them. The print statement on the last line prints `$a` followed by an appended newline (`\n`).

Details on the many regular expression options available in Perl can be found in the `perlre` man page. Additionally, the `perlfunc` man page lists many Perl functions that can manipulate text. For example, the `chop` function removes the last character from a string:

```
#!/usr/bin/perl
$a="foobar";
chop $a;
print $a."\n";
```

The result is:

```
Solaris% ./test.pl
fooba
```

This is a handy function when you are reading lines from a file or from the standard input. You can use `chop` to remove the newline present at the end of each line.

10.3.6. Lists

Most shell scripting languages have little to no support for manipulating lists, but Perl has excellent functionality for this. A list in Perl is a variable like any other, except that it is preceded by an at sign instead of a dollar sign. Here is an example of initializing and using a list:

```
#!/usr/bin/perl
@dances=("Waltz", "Foxtrot", "Tango", "Quickstep");
print "The first dance in the list is $dances[0]\n";
```

Open Source Network Administration By James Kretchmar
ISBN: 0-13-046210-1 Publisher: Prentice Hall Print Publication Date: 9/22/2003

```
print "The entire list is:\n";
foreach $i (@dances) {
    print "  $i\n";
}
```

Note that when you refer to the variable `dances` in a place where a list is expected (**list context**), it is called `@dances`, but when it is referenced in a place where a normal variable is expected (**scalar context**), `$dances` is used.

Perl has several functions for manipulating lists. The `push` function appends a value to the end of a list, while the pop function removes a value from the end:

```
#!/usr/bin/perl
@dances=("Waltz", "Foxtrot", "Tango", "Quickstep");
push(@dances, "Viennese Waltz");
$a=pop(@dances);
print "The last dance is $a\n";
$a=pop(@dances);
print "The last dance is $a\n";
```

This script produces:

```
Solaris% /var/tmp/test.pl
The last dance is Viennese Waltz
The last dance is Quickstep
```

The `shift` function removes a value from the beginning of the list, and the `unshift` function prepends a value onto the beginning of the list:

```
#!/usr/bin/perl
@dances=("Waltz", "Foxtrot", "Tango", "Quickstep");
unshift(@dances, "Samba");
$a=shift(@dances);
print "The first dance is $a\n";
$a=shift(@dances);
print "The first dance is $a\n";
```

when you run this:

```
Solaris% ./test.pl
The first dance is Samba
The first dance is Waltz
```

The shift function is often used in parsing command line arguments, as demonstrated later.

One final trick that is useful to know: The variable `$#dances` will contain the index of the last element in the list `dances`. That is, it will be one less than the size of the list. The following code is another way to print the list of dances:

```
#!/usr/bin/perl
@dances=("Waltz", "Foxtrot", "Tango", "Quickstep");
for $i (0 .. $#dances) {
   print "$dances[$i]\n";
}
```

10.3.7. Hashes

Hashes are another useful data structure in Perl. A hash is a set of key/value pairs. For example, the environment variables passed to a Perl program are stored in a hash called ENV. The name of an environment variable in ENV is the key part of the key/value pair, and the environment variable's value is the value part of the pair.

Much as you denote a variable representing an entire list by preceding it with an at sign instead of a dollar sign, you begin a variable representing an entire hash with a percentage sign. The ENV hash is then %ENV. But just as with lists, when you refer to an element of a hash in a scalar context, use a dollar sign instead. The syntax for accessing an element of a hash uses braces:

```
#!/usr/bin/perl
$vendor{"switch1"}="cisco";
$vendor{"switch2"}="enterasys";
print "The vendor for switch1 is " . $vendor{"switch1"} . "\n";
```

And the familiar example for accessing and changing environment variables is:

```
#!/usr/bin/perl
print "$ENV{HOME}\n";
$ENV{HOME}="/var/tmp/foobar";
```

10.3.8. Reading from a File

Reading lines from a file is easy in Perl:

```
#!/usr/bin/perl
open(FILE, "/etc/passwd");
while (<FILE>) {
    print $_;
}
close(FILE);
```

This will print every line from the password file. The variable $_ is special; it is set to successive lines of the file each time through the loop. Note that $_ will include the newline character at the end of each line. If you wish to remove the final newline character, you can use the chop function described in Section 10.3.5. In general, the $_ variable in Perl refers to whatever the "current thing" is. In this case, it is lines from the file.

Of course, it would be better programming practice to check if the file can actually be opened before you try to read from it, and you can do this in a variety of ways. The following will exit the script and print an error message if the file cannot be opened:

```
#!/bin/sh
open(FILE, "/tmp/myfile") || die "Could not open file";
while (<FILE>) {
    print $_;
}
close(FILE);
```

If you wish to read from the standard input, you can skip the open and close lines and use an empty file handle:

```
#!/usr/bin/perl
while (<>) {
   print $_;
}
```

10.3.9. Writing to a File

Writing to a file is also easy in Perl:

```
#!/usr/bin/perl
open(FILE, ">/var/tmp/foobar") || die "unable to write to file";
print FILE "This is the first line of text\n";
close(FILE);
```

The greater-than sign before the filename indicates that you wish to open the file for writing. You can also open a file such that writes will append to any text already present instead of overwriting the file. Using two greater-than signs instead of one:

```
#!/usr/bin/perl
open(FILE, ">>/var/tmp/log") || die "unable to append to file";
print FILE "This is the next log message\n";
close(FILE);
```

10.3.10. Arguments

Command line arguments are stored in a hash named `ARGV`. Unlike in other languages, `$ARGV[0]` does not contain the name of the script itself; it is the first argument on the command line. `$ARGV[1]` is the second argument and so on. You can use `$#ARGV` as the index of the last argument:

```
#!/usr/bin/perl
if ($#ARGV < 0) {
    print "You must supply an argument to this program\n";
    exit 1;
}
print "The first argument is $ARGV[0]\n";
```

10.3.11. Loops

Perl has the same kinds of loops as other languages, including `for` loops and `while` loops. A for loop is constructed as:

```
#!/usr/bin/perl
for $i (1 .. 4) {
    print "$i\n";
}
```

which produces:

```
Solaris% ./test.pl
1
2
3
4
```

Here's example syntax for a while loop used to parse command line arguments:

```
#!/usr/bin/perl
$verbose=0;
$quiet=0;
```

```
while (@ARGV) {
   $arg=shift(@ARGV);
   if ($arg eq "-v") {
      $verbose=1;
   } elsif ($arg eq "-q") {
      $quiet=1;
   } else {
      print "Invalid argument $arg\n";
   }
}
```

10.3.12. Using Command Output

Command output can be obtained in Perl with the use of single back quotes, just as in the Bourne shell:

```
#!/usr/bin/perl
$a='date';
print $a;
```

Perl has built-in functions for retrieving the date and time, but this example illustrates the use of back quotes to obtain command output.

10.3.13. Subroutines

Subroutines in Perl are defined through use of the `sub` keyword:

```
#!/usr/bin/perl
sub max {
   my $a=shift, $b=shift;
   if ($a > $b) {
      return $a;
   } else {
      return $b;
   }
}
print "The max of 5 and 12 is " . max(5,12) . "\n";
```

Arguments to the subroutine are passed in the variable `@_`, which is just like the variable `$_` but now in the context of being a list, namely the list of arguments. Because this list of arguments is the "current thing," when the subroutine is entered you can simply call the shift function with no arguments to retrieve the first argument from the list. The word `my` is used to instruct Perl that the variables `$a` and `$b` are to be local to the procedure instead of visible to the entire script as Perl would have them be by default.

10.3.14. Exiting

There a number of ways to exit a Perl script. You can use the exit function:

```
#!/usr/bin/perl
exit 0;
```

Or you can use the `die` function, which prints an error message to standard error and exits with whatever the current error number is.

The script:

```
#!/usr/bin/perl
die "This script failed"
```

produces the output:

```
Solaris% ./test.pl
This script failed at ./test.pl line 2.
```

10.3.15. Perl for Network Monitoring Scripts

Because Perl has such good text and list manipulation abilities, it is often the perfect tool for collecting output from several other programs. Perl can extract the relevant data from the formatted output that those programs produce and store it in data structures such as lists and hashes. This data can then be sorted or manipulated as required and finally printed in any format you desire.

10.4. Programming Monitors

Many administration tools will perform a task and then exit. For example, you may write a tool that queries SNMP variables on a device and then prints the results. This is a relatively straightforward program. However, you may also find that you occasionally want to write a program that runs all the time, monitoring a network service.

Ideally, you will not need to write a program like this yourself but can instead use the capabilities of other monitoring software. It is risky to have too many separate pieces of software performing monitoring functions. If one of them should die because of a bug or other problem, would you notice? If you have one central piece of monitoring software, you probably will, especially if you have software monitoring the monitor itself. But if you have 10 different scripts monitoring conditions of your network, all of which report only once in a while, how long will it take you to notice that one of them has stopped reporting? Regardless, the following sections provide a few tips that will help if you do find you need to write a monitoring script.

10.4.1. Loop Timing

Pay careful attention to the parts of your program that have loops in them. When writing a monitoring script, you will probably have at least one loop: the large loop around the entire program that keeps it going. Be sure to add delays to the loop as needed! You can do this with the `sleep` command in both Perl and the Bourne shell, as demonstrated below. Remember that the computer will try to execute your script as quickly as it can. If your script is sending SNMP probes, sending email, or doing anything that interacts with the rest of the world, you will want to make sure you do not cause an operational problem by overwhelming some resource. Let’s say you are going to write a script that checks if a machine is responding to SNMP requests. If the machine does not respond to SNMP, the script will send a piece of email to the network administrators:

```
#!/bin/sh
host=www.example.com
to=admins@example.com
community=public
while [ 1 ]; do
  snmpget -v 1 $host $community system.sysDescr.0 > /dev/null 2> \
     /dev/null
  if [ $? -eq 1 ]; then
     echo "The host $host is not pinging" | /usr/lib/sendmail $to
     sleep 1800
  fi
```

```
    sleep 60
  done
```

This script has a number of shortcomings: It assumes `sendmail` is available and it will repeatedly send email while the host being probed is unresponsive. The latter point will be addressed in the next section. It also does not include a subject line, which is a problem addressed in Section 10.4.4.

The important thing to notice is that there are two `sleep` statements in this example. The first one is at the end of the while loop, and it causes the program to sit and do nothing for a full minute. This ensures that the program does not do anything more often than once a minute. If this `sleep` statement were not present, the script would send a flood of SNMP requests to the host you wish to probe. Also vital is the `sleep` statement after email is sent, which causes it to wait 30 minutes (1800 seconds) after sending mail before doing anything else. If it were not present, and www.example.com stopped responding to SNMP requests for an extended period of time, the script would send a piece of email once every minute. Just imagine what would happen if neither `sleep` statement were included and www.example.com was down.

10.4.2. State Machines

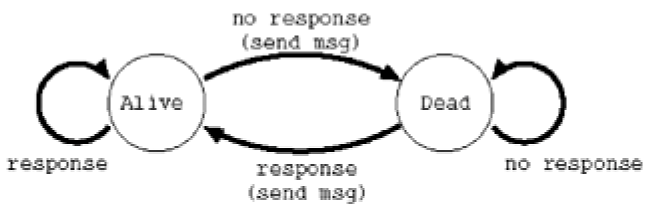
Instead of receiving a piece of email every half hour in the above example, it would be preferable to receive a piece of email only when the machine either has been responding but stops or has not been responding but begins to respond. To make this happen, you must introduce the concept of **state** into your script. You will use a variable that indicates the current state of the SNMP status of the machine (we can call the states “alive” and “dead”), and when the state changes, the program triggers a message:

```
#!/bin/sh
host="www.example.com"
to="admins@example.com"
community="public"
state="alive"
while [ 1 ]; do
  snmpget -v 1 $host $community system.sysDescr.0 > /dev/null 2> \
     /dev/null
  response=$?
  if [ "$state" = "alive" ]; then
     if [ $response -eq 1 ]; then
        echo "The host $host has stopped pinging" | \
   /usr/lib/sendmail $to
        sleep 300
        state="dead"
     fi
  elif [ "$state" = "dead" ]; then
     if [ $response -eq 0 ]; then
        echo "The host $host has started pinging" | \
   /usr/lib/sendmail $to
        sleep 300
        state="alive"
     fi
  fi
  sleep 60
done
```

If the SNMP state is “alive” but the machine is no longer responding, a notification is sent, and the state is changed to “dead.” If the state is “dead” and the machine is responding, notification is sent and the state is changed to “alive.” When the state stays the same (SNMP was dead and is still not responding or SNMP was alive and continues to respond), no message is sent.

A program that has different states like this is called a **finite state machine**. The possible states of the program, along with indications of how to transition between those states, can be represented in a simple **state transition diagram**, as in Figure 10.1. This program is simple and has only two states, making the state transition diagram useful only as an exercise. But add another state or two to your program, and it can easily become complicated enough that a state transition diagram will be a great help. It will ensure that you fully understand the structure of the state machine and that you do not forget to deal with any unexpected circumstances. It is surprisingly easy to forget about a state transition or two if you have even a modest number of states in your program.

Figure 10.1. Simple State Transition Diagram.



10.4.3. Keeping It Running

When writing a monitoring tool, especially one that will remain silent when there are no problems to report, you must take care to write the program so that it is not likely to exit unexpectedly. For scripting languages like the Bourne shell and Perl, this mostly this means keeping the scripts simple and avoiding calls to `exit` and other functions that terminate the program. In languages such as C, you will need to take many additional measures to ensure the program does not exit because of memory corruption or other bugs that are unlikely to occur in a scripting language.

10.4.4. Sending Nicer Mail with Sendmail

The `sendmail` program does not have a convenient mechanism for specifying a subject line or other message headers on the command line. In order to send a message that includes these headers, you can create an appropriately formatted file and run `sendmail` with the `-t` argument. The following program fragment demonstrates one way to do this in the Bourne shell:

```
tmpmsg=/tmp/msg.$$
echo "To: admins@example.com" > $tmpmsg
echo "From: root@server.example.com" >> $tmpmsg
echo "Subject: Monitor for $host" >> $tmpmsg
echo "" >> $tmpmsg
echo "$host is not responding" >> $tmpmsg

cat $tmpmsg | /usr/lib/sendmail -t
rm -f $tmpmsg
```

Or, if you want to impress your friends, you can use this fancy shortcut:

```
tmpmsg=/tmp/msg.$$
cat > $tmpmsg <<EOF
To: admins@example.com
From: root@server.example.com
Subject: Monitor for $host

$host is not responding
EOF

cat $tmpmsg | /usr/lib/sendmail -t
rm $tmpmsg
```

Make sure the blank line is present between the subject line and the message body; this is how `sendmail` determines where the headers end and the message begins.

In Perl, the process is the same: Create a file with the message you wish to send and feed it to the standard input of `sendmail -t`:

```
open(SENDMAIL, "|/usr/lib/sendmail -t");
print SENDMAIL "To: admins\@example.com\n";
print SENDMAIL "From: root\@server.example.com\n";
print SENDMAIL "Subject: Monitor for $host\n";
print SENDMAIL "\n";
print SENDMAIL "$host is not responding.\n";
close(SENDMAIL);
```

Note that at signs (@) need to be escaped with a backslash and that a newline character is explicitly included at the end of each line.

10.5. Running Programs from Cron

You may wish to run scripts or other administration tools in an automated fashion on a regular basis. The Unix `cron` program will allow you to do this. If you run `crontab -e` from the command line, it will bring up an editor.[5] In this editor, you can add or edit the names of programs to be run. The account from which you run the `crontab` program is the account that will execute the programs later on. If the program should be run as root, make sure to run `crontab` as root when you add the entry.

[5] If you are not happy with the default editor, which may be `ed`, you can override it by setting the EDITOR environment variable to the editor of your choice.

Each line of the file specifies one program to run. The first five fields denote the time the program should be started. They are, in order:

- The minute (0–59)
- The hour (0–23)
- The day of the month (1–31)
- The month (1–12)
- The day of the week (0–6, 0 is Sunday)

An asterisk can be used in any of these fields to indicate that every value is acceptable. For example, `00 1 * * *` would mean that the program should be run every day at 1:00 a.m.

Following the timing information is the name of the program to run and its arguments. For example:

```
Solaris# crontab -l
37 * * * *      /usr/lib/sendmail -q > /dev/null 2> /dev/null
```

As you can see, running the `crontab` program with the `-l` option will print the current configuration. Here the mail queue is run every hour at 37 minutes past the hour. The standard output and standard error are redirected to `/dev/null` because, otherwise, cron will send output in email, and in this case, it was decided that it would be preferable not to receive any error messages.

Here's another example:

```
Solaris# crontab -l
05 9 * * *      /usr/local/bin/report.pl -e > /dev/null
```

This runs a report program every day at 9:05 a.m. The program sends email to administrators about problem conditions that have arisen within the past day.

After you edit the `crontab` file, save and exit the editor, and the changes will be complete. You should test the script to make sure it works correctly from `cron`.

10.6. References and Further Study

The Bourne shell is well documented in the `sh` man page. There are many extensive references on the Perl language, both in books and on-line. The web site http://www.perldoc.com/ is a good, canonical source for on-line documentation and also contains links to other sites with information on Perl. There is so much information in the online Perl manuals that the man page has been split into many different man pages. Typing `man perl` will list an index of the other perl man pages. Particularly useful are `perlsyn` for Perl syntax, `perlfunc` for information about Perl functions, `perlop` for information about Perl operators and `perlre` for information about Perl regular expressions. The books *Programming Perl*, *3rd ed.,* (O'Reilly and Associates, 2000) by Larry Wall, Tom Christiansen, and Jon Orwant and *Learning Perl*, *3rd ed.,* (O'Reilly and Associates, 2001) by Randal L. Schwartz and Tom Phoenix are commonly used references for Perl.

Open Source Network Administration By James Kretchmar
ISBN: 0-13-046210-1 Publisher: Prentice Hall Print Publication Date: 9/22/2003