Shell Scripting Primer

Open Source > Scripting & Automation

2008-04-08

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Introduction

Shell scripts are a fundamental part of the Mac OS X programming environment. As a ubiquitous feature of UNIX-based, Linux, and UNIX-like operating systems, they represent a way of writing certain types of command-line tools in a way that works on a fairly broad spectrum of computing platforms.

Because shell scripts are written in an interpreted language whose power comes from executing external programs to perform processing tasks, their performance can be somewhat limited. However, because they can execute without any additional effort on nearly any modern operating system, they represent a powerful tool for bootstrapping other technologies. For example, the `autoconf` tool, used for configuring software prior to compilation, is a series of shell scripts.

You should read this document if you are interested in learning the basics of shell scripting. This document assumes that you already have some basic understanding of at least one procedural programming language such as C. It does not assume that you have very much knowledge of commands executed from the terminal, though, and thus should be readable even if you have never run the Terminal application before.

The techniques in this document are not specific to Mac OS X, although this document does note various quirks of certain command-line utilities in various operating systems. In particular, it includes information about some cases where the Mac OS X versions of command-line utilities behave differently than other commonly available versions such as the GNU equivalents commonly used in Linux and some BSD systems.

This document is not intended to be a complete reference for shell scripting, as such a subject could fill entire libraries. However, it is intended to provide enough information to get you started writing and comprehending shell scripts. Along the way, it provides links to documentation for various additional tools that you may find useful when writing shell scripts.

For your convenience, many of the scripts in this document are also included in the companion files zip archive. You can find this archive in the table of contents when viewing this document in HTML form on the ADC Reference Library website.

Organization of This Document

This document is organized in a series of topics. These topics can be read linearly as a tutorial, but are also organized with the intent to be a quick reference on key subjects.

- "Shell Script Basics" (page 11)—introduces basic concepts of shell scripting, including variables, control statements, file I/O, pipes, and redirection.
- "Result Codes, Subroutines, Scoping, and Sourcing" (page 35)—describes how to obtain result codes from outside executables, how to write and call subroutines, subroutine variable scoping, and how to include one shell script inside another.
- "Paint by Numbers" (page 45)—explains how to use integer math in shell scripts. This section also explains how to use the `bc` command-line utility or Perl to handle more complex math, such as floating-point calculations.
- "Regular Expressions Unfettered" (page 51)—describes basic and extended regular expressions and how to use them. This section also describes the differences between these regular expression dialects and the dialect supported by Perl, and shows how to use Perl regular expressions through inline scripting.
- "Other Tools and Information" (page 121)—provides a basic summary of various commands that may be useful to shells script developers, including links to Mac OS X documentation on each of them.
- "An Extreme Example: The Monte Carlo (Bourne) Method for Pi" (page 127)—this appendix provides a complex example to showcase the power of shell scripts to perform complex tasks (slowly). The code example shows a shell script implementation of the Monte Carlo method for approximating the value of Pi. The code example takes advantage of all of the techniques described in the previous chapters. By showing some of the same calculations written in multiple ways, it also illustrates why it is often beneficial, performance-wise, to embed scripts written in other languages such as Perl or Awk when attempting tasks that suit those languages better.

Happy scripting!

Shell Script Basics

Writing a shell script is like riding a bike. You fall off and scrape your knees a lot at first, but once you are sufficiently experienced, you'll understand why people drive cars. If you have ever successfully trued a bicycle wheel, that's similar to knowing the basics of shell scripting. If you don't true your scripts, they wobble.

This chapter introduces basic concepts of shell scripting. It was not intended to be a complete reference on writing shell scripts, nor could it be. It does, however, provide a good starting point for beginners first learning this black art.

Shell Script Dialects

There are many different dialects of shell scripts, each with their own quirks, and some with their own syntax entirely. Because of these differences, the road to good shell scripting can be fraught with peril, leading to script failures, misbehavior, and even outright data loss.

To that end, the first lesson you must learn before writing a shell script is that there are two fundamentally different sets of shell script syntax: the Bourne shell syntax and the C shell syntax. The C shell syntax is more comfortable to many C programmers because the syntax is somewhat similar. However, the Bourne shell syntax is significantly more flexible and thus more widely used. For this reason, this document only covers the Bourne shell syntax.

The second hard lesson you will invariably learn is that each dialect of Bourne shell syntax differs slightly. This document includes only pure Bourne shell syntax and a few BASH-specific extensions. Where BASH-specific syntax is used, it is clearly noted.

The terminology and subtle syntactic differences can be confusing—even a bit overwhelming at times; had Dorothy in *The Wizard of Oz* been a programmer, you might have heard her exclaim, "BASH and ZSH and CSH, Oh My!" Fortunately, once you get the basics, things generally fall into place as long as you avoid using shell-specific features. Stay on the narrow road and your code will be portable.

Some common shells are listed below, grouped by script syntax:

Bourne-compatible shells

- sh
- bash
- zsh

- ksh

C-shell-compatible shells

- csh
- tcsh
- bcsh (C shell to Bourne shell translator)

Many of these shells have more than one variation. Most of these variations are denoted by prefixing the name of an existing shell with additional letters that are short for whatever differentiates them from the original shell. For example:

- The shell `pdksh` is a variant of `ksh`. Being a public domain rewrite of AT&T's `ksh`, it stands for "Public Domain Korn SHell." (This is a bit of a misnomer, as a few bits are under a BSD-like open source license. However, the name remains.)
- The shell `tcsh` is an extension of `csh`. It stands for the TENEX C SHell, as some of its enhancements were inspired by the TENEX operating system.
- The shell `bash` is an extension of `sh`. It stands for the Bourne Again SHell. (Oddly enough, it is not a variation of `ash`, the Almquist SHell, though both are Bourne shell variants.)

And so on. In general, with the exception of `csh` and `tcsh`, it is usually safe to assume that any shell will be compatible with Bourne shell syntax.

Note: Because the C shell syntax is not well suited to scripting beyond a very basic level, this document does not cover C shell variants in depth. For more information, see "She Sells C Shells" (page 12).

She Sells C Shells

The C shell is popular among some users as a shell for interacting with the computer because it allows simple scripts to be written more easily. However, the C shell scripting language is limited in a number of ways, many of which are hard to work around. For this reason, use of the C shell scripting language for writing significant scripts is not recommended. For more information, read "CSH Programming Considered Harmful" at http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/.

However, the C shell scripting language has its uses, particularly when writing scripts that set up environment variables for interactive shell environments. Where possible, the C shell syntax is presented alongside the Bourne shell syntax within this "basics" chapter.

Apart from this chapter, however, this document does not generally cover the C shell syntax. If after reading this, you still want to write a more complex script using the C shell programming language, you can find more information in on the C shell in the manual page for `csh`.

Tips for Shell Users

While this document is primarily focused on writing shell scripts, there are a few helpful tips that can be useful to shell users and programmers alike. This section includes a few of those tips.

The alias Builtin

Various Bourne shells also offer a number of other builtin commands that you may find useful, one of the more useful for command-line users being `alias`. This command allows you to assign a short name to replace a longer command. While this is not frequently used in shell scripts (unless you are intentionally trying to obfuscate your code), it is very convenient when using the shell interactively. For example:

```
alias listsource="ls *.c *.h"
```

Typing the command listsource after entering this line will result in listing all of the `.c` and `.h` files in the current directory.

For more information, see the man page `builtins`, or for ZSH, `zshbuiltins`.

C Shell Note: The C shell syntax is similar, but not identical. In the C shell, the equals sign is replaced with a space. For example:

```
alias listsource "ls *.c *.h"
```

Entering Special Characters

Some shells treat tabs and other control characters in special ways. When writing a script in a text file, this is not generally an issue. However, when entering commands on the command line, it may get in the way if you need to enter these special characters as part of a command for some reason.

To enter a tab or other control character on the command line, type control-v followed by the tab key or other control character. The control-v tells the shell to treat whatever character comes next literally without interpreting it in any way during entry.

For example, to enter the ASCII bell character (control-G), you can type the following:

```
echo "control-V control-G"
```

This will be seen on your screen as:

```
echo "^G"
```

When you press return, your computer should beep.

Shell Variables and Printing

What follows is a very basic shell script that prints "Hello, world!" to the screen:

```
#! /bin/sh

echo "Hello, world!"
```

The first thing you should notice is that the script starts with '`#!`'. This is known as an interpreter line. If you don't specify an interpreter line, the default is usually the Bourne shell (`/bin/sh`). However, it is best to specify this line anyway for consistency.

The second thing you should notice is the echo statement. This is nearly universal in shell scripting as a means for printing something to the user's screen.

If you'd like, you can try this script by saving those lines in a text file (say "hello_world.sh") in your home directory. Then, in Terminal, type:

```
chmod u+x hello_world.sh
./hello_world.sh
```

Of course, this script isn't particularly useful. It just prints the words "Hello, world!" to your screen. To make this more interesting, the next script throws in a few variables.

```
#! /bin/sh

FIRST_ARGUMENT="$1"
echo "Hello, world $FIRST_ARGUMENT!"
```

Type or paste this script into the text editor of your choice (TextEdit, for example) and save the file in your home directory in a file called `test.sh`. Then type '`chmod a+x test.sh`' in Terminal to make it executable. Finally, run it with '`./test.sh leaders`'. You should see "Hello, world leaders!" printed to your screen.

This script provides an example of a variable assignment. The variable $1 contains the first argument passed to the shell script. In this example, the script makes a copy and stores it into a variable called FIRST_ARGUMENT, then prints that variable.

You should immediately notice that, depending on use, variables may or may not begin with a dollar sign. If you are dereferencing the variable, you precede it with a dollar sign. The shell inserts the contents of the variable at that point in the script. You would not want to do this on the left side of the assignment statement, however, since FIRST_ARGUMENT starts out empty, and the first line would then evaluate to the following:

```
="myfirstcommandlineargument"
```

This is clearly not what you want.

You should also notice the double quotation marks. These are important, as any command-line argument could potentially contain spaces. Normally, when executing a command, spaces are treated as separating multiple arguments, and thus, shell scripts can behave differently when variables containing spaces are involved. By enclosing the variables in double quotes, they are treated as part of a single argument even if the value stored in the variable contains a space.

Had you omitted the quotes, if the first argument contained spaces (the phrase "This is a test", for example), the statement might evaluate to something like this:

```
FIRST_ARGUMENT=This is a test
```

The result would be that the shell would try to execute a command called `is`, passing it two arguments, `a` and `test`, and would set the variable `FIRST_ARGUMENT` to `This` in the environment it passes to that command. Again, while this behavior is useful, it is clearly not what you intended to do. (This behavior and its uses are described more in "Overriding Environment Variables for Child Processes (Bourne Shell)" (page 17).)

Note: Expansion of variables, occurs *after* the statement itself is fully parsed by the shell. Thus, as long as the variable is enclosed in double quote marks, you will not get any execution errors even if the variable's value contains double-quote marks.

However, if you are using double quote marks within a literal string, you must quote that string properly. For example:

```
MYSTRING="The word of the day is \"sedentary\"."
```

The details are explained in "Variables, Expansion, and Quoting" (page 31).

Shell scripts also allow the use of single quote marks. Variables between single quotes are not replaced by their contents. Be sure to use double quotes unless you are intentionally trying to display the actual name of the variable. You can also use single quotes as a way to avoid the shell interpreting the contents of the string in any way. These differences are described further in "Variables, Expansion, and Quoting" (page 31).

C Shell Note: The syntax for assignment statements in the C shell is rather different. Instead of an assignment statement, the C shell uses the `set` and `setenv` builtins to set variables as shown below:

```
set VALUE = "Four"
# or...
setenv VALUE "Four"

echo "$VALUE score and seven years ago...."
```

The functional difference between `set` and `setenv` is described in "Exporting Shell Variables" (page 15).

Exporting Shell Variables

One key feature of shell scripts is that they are limited in their scope to the currently running script. The scoping of variables is described in more detail in "Result Codes, Subroutines, Scoping, and Sourcing" (page 35). For now, though, it suffices to say that variables generally do not get passed on to scripts or tools that they execute.

Normally, this is what you want. Most variables in a shell script do not have any meaning to the tools that they execute, and would thus just represent clutter and the potential for variable namespace collisions if they are exported. Occasionally, however, you will find it necessary to make a variable's

value available to a tool. To do this, you must export the variable. These exported variables are commonly known as environment variables because they are outside the scope of the script itself, but affect its execution.

A classic example of an environment variable that is significant to scripts and tools is the `PATH` variable. This variable specifies a list of locations that the shell searches when executing programs by name (without specifying a complete path). For example, when you type `ls` on the command line, the shell searches in the locations specified in `PATH` (in the order specified) until it finds an executable called `ls` (or runs out of locations, whichever comes first).

The details of exporting shell variables differ considerably between the Bourne shell and the C shell. Thus, the following sections explain these details in a shell-specific fashion.

Using the export Builtin (Bourne Shell)

Generally speaking, the first time you assign a value to an environment variable such as the `PATH` variable, the Bourne shell creates a new, *local* copy of this shell variable that is specific to your script. Any tool executed from your script is passed the original value of `PATH` inherited from whatever script, tool, or shell that launched it.

With the BASH shell, however, any variable inherited from the environment is automatically exported by the shell. Thus, in some versions of Mac OS X, if you modify inherited environment variables (such as `PATH`) in a script, your local changes will be seen automatically by any tool or script that your script executes. Thus, in these versions of Mac OS X, you do not have to explicitly use the `export` statement when modifying the `PATH` variable.

Because different Bourne shell variants handle these external environment variables differently (even among different versions of Mac OS X), this creates two minor portability problems:

- A script written without the `export` statement may work on some versions of Mac OS X, but will fail on others. You can solve this portability problem with the `export` builtin, as described in "Using the export Builtin (Bourne Shell)" (page 16).
- A shell script that changes variables such as `PATH` will alter the behavior of any script that it executes, which may or may not be desirable. You can solve this problem by overriding the `PATH` environment variable when you execute each individual tool, as described in "Overriding Environment Variables for Child Processes (Bourne Shell)" (page 17).

To guarantee that your modifications to a shell variable are passed to any script or tool that your shell script calls, you *must* use the `export` command. You do not have to use this command every time you change the value; the variable remains exported until the shell script exits.

For example:

```
export PATH="/usr/local/bin:$PATH"
# or
PATH="/usr/local/bin:$PATH"
export PATH
```

Either of these statements has the same effect—specifically, they export the local notion of the `PATH` environment variable to any command that your script executes from now on. There is a small catch, however. You cannot later undo this export to restore the original global declaration. Thus, if you need to retain the original value, you must store it somewhere yourself.

In the following example, the script stores the original value of the PATH environment variable, exports an altered version, executes a command, and restores the old version.

```
ORIGPATH="$PATH"
PATH="/usr/local/bin:$PATH"
export PATH
# Execute some command here---perhaps a
# modified ls command....
ls
PATH="$ORIGPATH"
```

If you need to find out whether an environment variable (whether inherited by your script or explicitly set with the export directive) was set to empty or was never set in the first place, you can use the printenv command to obtain a complete list of defined variables and use grep to see if it is in the list. (You should note that although printenv is a csh builtin, it is also a standalone command in /usr/bin.)

For example:

```
DEFINED=`printenv | grep -c '^VARIABLE='`
```

The resulting variable will contain 1 if the variable is defined in the environment or 0 if it is not.

Overriding Environment Variables for Child Processes (Bourne Shell)

Because the BASH Bourne shell variant automatically exports all variables inherited from its environment, any changes you make to preexisting environment variables such as PATH are automatically inherited by any tool or script that your script executes.

While this is usually convenient, you may sometimes wish to change a preexisting environment variable without modifying the environment of any script or tool that your script executes. For example, if your script executes a number of tools in /usr/local/bin, it may be convenient to change the value of PATH to include /usr/local/bin. However, you may not want child processes to also look in /usr/local/bin.

This is easily solved by overriding the environment variable PATH on a per-execution basis. Consider the following script:

```
#!/bin/sh

echo $MYVAR
```

This script prints the value of the variable MYVAR. Normally, this would be a blank line. Save the script as printmyvar.sh, then type the following commands:

```
chmod a+x printmyvar.sh
MYVAR=7 ./printmyvar.sh
echo "MYVAR IS $MYVAR"
```

The initial assignment statement applies to the command that follows it. You should notice that the echo statement afterwards prints an empty string for the value of MYVAR. The value of MYVAR is only altered in the environment of the command ./printmyvar.sh, and the original (empty) value is restored after executing that command.

Thus, to modify the PATH variable locally but execute a command with the original PATH value, you can write a script like this:

```
#!/bin/sh
GLOBAL_PATH="$PATH"
PATH=/usr/local/bin

PATH="$GLOBAL_PATH" /bin/ls
```

Using the setenv Builtin (C shell)

In the C shell, variables are exported if you set them with setenv, but not it you set them with set. Thus, if you want your shell variable modifications to be seen by any tool or script that you call, you should use the setenv builtin. This is equivalent to an assignment statement with the export command in the Bourne shell.

```
setenv VALUE "Four"
echo "VALUE is '$VALUE'."
```

If you want your shell variables to only be available to your script, you should use the set builtin (described in "Shell Variables and Printing" (page 14)). The set builtin is equivalent to a simple assignment statement in the Bourne shell.

```
set VALUE = "Four"
echo "VALUE is '$VALUE'."
```

Notice that the local variable version requires an equals sign (=), but the exported environment version does not (and produces an error if you put one in).

To remove variables in the C shell, you can use the unsetenv or unset builtin. For example:

```
setenv VALUE "Four"
unsetenv VALUE

set VALUE = "Four"
unset VALUE

echo "VALUE is '$VALUE'."
```

This will generate an error message. In the C shell, it is not possible to print the value of an undefined variable, so if you think you may need to print the value later, you should set it to an empty string rather than using unset or unsetenv.

If you need to test an environment variable (*not* a shell-local variable) that may or may not be part of your environment (a variable set by whatever process called your script), you can use the printenv builtin. This prints the value of a variable if set, but prints nothing if the variable is not set, and thus behaves just like the variable would behave in the Bourne shell.

For example:

```
set X = `printenv VALUE`
echo "X is "\"$X\"
```

This prints X is "" if the variable is either empty or undefined. Otherwise, it prints the value of the variable between the quotation marks.

If you need to find out if a variable is simply empty or is actually not set, you can also use `printenv` to obtain a complete list of defined variables and use `grep` to see if it is in the list. For example:

```
set DEFINED = `printenv | grep -c '^VARIABLE='`
```

The resulting variable will contain 1 if the variable is defined in the environment or 0 if it is not.

Overriding Environment Variables for Child Processes (C Shell)

Unlike the Bourne shell, the C shell does not provide a built-in syntax for overriding environment variables when executing external commands. However, it is possible to simulate this either by using the `env(1)` command.

The best and simplest way to do this is by using the `env` command. For example:

```
env PATH="/usr/local/bin" /bin/ls
```

As an alternative, you can use the `set` builtin to make a temporary copy of any variable you need to override, change the value, execute the command, and restore the value from the temporary copy.

You should notice, however, that whether you use the `env` command or manually make a copy, the `PATH` variable is altered *prior to* searching for the command. Because the `PATH` variable controls where the shell looks for programs to execute, you must therefore explicitly provide a *complete* path to the `ls` command or it will not be found (unless you have a copy in `/usr/local/bin`, of course). The `PATH` environment variable is explained in "Special Shell Variables" (page 96).

As a workaround, you can determine the path of the executable using the `which` command prior to altering the `PATH` environment variable.

```
set GLOBAL_PATH = "$PATH"
set LS = `which ls`
setenv PATH "/usr/local/bin"
$LS
setenv PATH "$GLOBAL_PATH"
unset GLOBAL_PATH
```

Or, using `env`:

```
set LS = `which ls`
env PATH='/usr/local/bin' $LS
```

The use of the back-tick (`) operator in this fashion is described in "Inline Execution" (page 34).

Security Note: If your purpose for overriding an environment variable is to prevent disclosure of sensitive information to a potentially untrusted process, you should be aware that if you use `setenv` for the copy, the called process will have access to that temporary copy just as it would have access to the original variable.

To avoid this, be sure to create the temporary copy using the `set` builtin instead of `setenv`.

Shell Script Input and Output Using printf and read

Next, you might ask what to do if you need to get input from the user. The Bourne shell syntax provides basic input with very little effort.

```
#!/bin/sh
printf "What is your name?  -> "
read NAME
echo "Hello, $NAME.  Nice to meet you."
```

You will notice two things about this script. The first is that it introduces the `printf` command. This command is used because, unlike `echo`, the `printf` command does not automatically add a newline to the end of the line of output. This is useful when you need to use multiple lines of code to output a single line of text. It also just happens to be handy for prompts.

Note: In most operating systems, you can tell `echo` to suppress the newline. However, the syntax for doing so varies. Thus, `printf` is recommended for printing prompts. See "Designing Scripts for Cross-Platform Deployment" (page 77) for more information and other alternatives.

The second thing you'll notice is the `read` command. This command takes a line of input and separates it into a series of arguments. Each of these arguments is assigned to the variables in the `read` statement in the order of appearance. Any additional input fields are appended to the last entry.

You can modify the behavior of the `read` command by modifying the shell variable `IFS` (short for internal field separators). The default behavior is to split inputs everywhere there is a space, tab, or newline. By changing this variable, you can make the shell split the input fields by tabs, newlines, semicolons, or even the letter 'q'. This is demonstrated in the following example:

```
#!/bin/sh
printf "Type three numbers separated by 'q'. -> "
IFS="q"
read NUMBER1 NUMBER2 NUMBER3
echo "You said: $NUMBER1, $NUMBER2, $NUMBER3"
```

If, for example, you run this script and enter `1q3q57q65`, the script would reply with `You said: 1, 3, 57q65`. The third value contains `57q65` because only three values are requested in the read statement.

Note: The `read` statement *always* stops reading at the first newline encountered. Thus, if you set `IFS` to a newline, you cannot read multiple entries with a single `read` statement.

Warning: Changing IFS may cause unexpected consequences for variable expansion. For more information, see "Variable Expansion and Field Separators" (page 31).

But what if you don't know how many parameters the user will specify? Obviously, a single read statement cannot split the input up into an arbitrary number of variables, and the Bourne shell does not contain true arrays. Fortunately, the `eval` command can be used to simulate an array using multiple shell variables. This is described in "Data Structures, Arrays, and Indirection" (page 87).

Alternatively, you can use the `for` statement, which splits a single variable into multiple pieces based on the internal field separators. This statement is described in "The for Statement" (page 28).

C Shell Note: In the C shell, the syntax for reading is completely different. The following script is the C shell equivalent of the script earlier in this section:

```
printf "What is your name?  -> "
set NAME = $<
echo "Hello, $NAME.  Nice to meet you."
```

The C shell does not provide a way to read multiple values in a single command, though you can approximate this with careful use of `sed` as described in "Regular Expressions Unfettered" (page 51).

Bulk I/O Using the cat Command

For small I/O, the `echo` command is well suited. However, when you need to create large amounts of data, it may be convenient to send multiple lines to a file simultaneously. For these purposes, the `cat` command can be particularly useful.

By itself, the `cat` command really doesn't do anything that can't be done using redirect operators (except for printing the contents of a file to the user's screen). However, by combining it with the special operator `<<`, you can use it to send a large quantity of text to a file (or to the screen) without having to use the `echo` command on every line.

For example:

```
cat > mycprogram.c << EOF
#include <stdio.h>
int main(int argc, char *argv[])
{
    char array[] = { 0x25, 115, 0 };
    char array2[] = { 68, 0x61, 118, 0x69, 0144, 040,
                      0107, 97, 0x74, 119, 0157, 0x6f,
                      100, 0x20, 0x72, 117, 'l', 0x65,
                      115, 041, 012, 0 };
    printf(array, array2);
}
EOF
```

In this example, the text leading up to (but not including) the line that *begins* with `EOF` is stored in the file `mycprogram.c`. Note that the token `EOF` can be replaced with any token, so long as the following conditions are met:

- The token must not contain spaces or quotation marks. (Shell variables will *not* be expanded, so the `$` character is allowed.)
- The token after the `<<` in the starting line must match the token at the beginning of the last line.
- The end-of-block token must be at the very beginning of the line. If it appears after any other characters (*including* whitespace), it will be treated as part of the text to be output.
- The end-of-block token you choose must never appear at the start of a line in the intended output string.

This technique is also frequently used for printing instructions to the user from an interactive shell script. This avoids the clutter of dozens of lines of `echo` commands and makes the text much easier to read and edit in an external text editor (if desired).

Another classic example of this use of `cat` in action is the `.shar` file format, created by the tool `shar` (short for SHell ARchive). This tool takes a list of files as input and uses them to create a giant shell script which, when executed, recreates those original files. To avoid the risk of the end-of-block token appearing in the input file, it prepends each line with a special character, then strips that character off on output.

C Shell Note: The multi-line `cat` syntax is the same in the C shell and the Bourne shell.

Pipes and Redirection

As you may already be aware, the true power of shell scripting lies not in the scripts themselves, but in the ability to read and write files and chain multiple programs together in interesting ways.

Each program in a UNIX-based or UNIX-like system has three basic file descriptors (normally a reference to a file or socket) reserved for basic input and output: standard input (often abbreviated stdin), standard output (stdout), and standard error (stderr).

The first, standard input, normally takes input from the user's keyboard (when the shell window is in the foreground, of course). The second, standard output, normally contains the output text from the program. The third, standard error, is generally reserved for warning or error messages that are not part of the normal output of the program. This distinction between standard output and standard error is a very important one, as explained in "Pipes and File Descriptor Redirection (Bourne Shell)" (page 23).

Basic File Redirection

One of the most common types of I/O in shell scripts is reading and writing files. Fortunately, it is also relatively simple to do. Reading and writing files in shell scripts works exactly like getting input from or sending output to the user, but with the standard input redirected to come from a file or with the standard output redirected to a file.

For example, the following command creates a file called `MyFile` and fills it with a single line of text:

```
echo "a single line of text" > MyFile
```

Appending data is just as easy. The following command appends another line of text to the file `MyFile`.

```
echo "another line of text" >> MyFile
```

You should notice that the redirect operator (`>`) creates a file, while the append operator (`>>`) appends to the file. There is a third operator in this family, the merging redirect operator (`>&`). The most common use of this merging redirect operator is to redirect standard error and standard output simultaneously to a file. For example:

```
ls . THISISNOTAFILE >& filelistwitherrors
```

This creates a file called `filelistwitherrors`, containing both a listing of the current directory and an error message about the nonexistence of the file `THISISNOTAFILE`. The standard output and standard error streams are merged and written out to the resulting file.

Note: The merging redirect operator (`>&`) is a very powerful operator. Additional uses beyond basic use are described in more detail in "Pipes and File Descriptor Redirection (Bourne Shell)" (page 23).

Pipes and File Descriptor Redirection (Bourne Shell)

The simplest example of the use of pipes is to pipe the standard output of one program to the standard input of another program. Type the following on the command line:

```
ls -l | grep 'rwx'
```

You will see all of the files whose permissions (or name) contain the letters `rwx` in order. The `ls` command lists files to its standard output, and the `grep` command takes its input and sends any lines that match a particular pattern to its standard output. Between those two commands is the pipe operator (`|`). This tells the shell to connect the standard output of `ls` to the standard input of `grep`.

Where the distinction between standard output becomes significant is when the `ls` command gives an error.

```
ls -l THISFILEDOESNOTEXIST | grep 'rwx'
```

You should notice that the `ls` command issued an error message (unless you have a file called `THISFILEDOESNOTEXIST` in your home directory, of course). If the `ls` command had sent this error message to its standard output, it would have been gobbled up by the `grep` command, since it does not match the pattern `rwx`. Instead, the `ls` command sent the message to its standard error descriptor, which resulted in the message going directly to your screen.

In some cases, however, it can be useful to redirect the error messages along with the output. You can do this by using a special form of the combining redirection operator (`>&`).

Before you can begin, though, you need to know the file descriptor numbers. Descriptor 0 is standard input, descriptor 1 is standard output, and descriptor 2 is standard error. Thus, the following command redirects standard error to standard output, then pipes the result to grep:

```
ls -l THISFILEDOESNOTEXIST 2>&1 | grep 'rwx'
```

It is also often useful for your script to send something to standard error. The following command sends an error message to standard error:

```
echo "an error message" 1>&2
```

This works by taking the standard output (descriptor 1) of the echo command and redirects it to standard error (descriptor 2).

You should notice that the ampersand (`&`) appears to behave somewhat differently than it did in "Basic File Redirection" (page 22). Because the ampersand is followed immediately by a number, this causes the output of one data stream to be merged into another stream. In actuality, however, the behavior is the same.

The redirect (`>`) operator implicitly redirects standard output unless combined with an ampersand, in which case it implicitly merges standard error into standard output and writes the result to a file. By specifying numbers, your script is simply overriding which file descriptor to use as its source and specifying a file descriptor to receive the result instead of a file.

Pipes and File Descriptor Redirection (C Shell)

The C shell does not support the full set of file descriptor redirection that the Bourne shell supports. In some cases, alternatives are provided. For example, you can pipe standard output and standard error to the same process using the `|&` operator as shown in the following snippet:

```
ls -l THISFILEDOESNOTEXIST |& grep 'rwx'
```

Some other operations, however, are not possible. You cannot, for example, redirect standard error without redirecting standard output. At best, if you can determine that your standard output will always be `/dev/tty`, you can work around this by redirecting standard output to `/dev/tty` first, then redirecting both the now-empty standard output and standard error using the `>&` operator. For example, to redirect only standard error to `/dev/null`, you could do this:

```
(ls > /dev/tty) >& /dev/null
```

This technique is *not* recommended for general use, however, as it will send output to your screen if anyone runs your script with standard output set to a file or pipe.

You can also work around this using a file, but not in an interactive way. For example:

```
(ls > /tmp/mytemporarylslisting) >& /dev/null
cat /tmp/mytemporarylslisting
```

It is not possible to redirect messages to standard error using the C shell unless you write a Bourne shell script or C program to do the redirection for you.

Basic Control Statements

The examples up to this point have been very basic, linear programs. This section will introduce some control flow statements to allow for more complex programs.

Note: C shell programming is not covered by this section in detail. The syntax for control statements in C shell programming is essentially the same as in C except for the following caveats:

- The C shell adds a `foreach` statement. Syntax is `foreach` *variable* (*list of values*) *statement* and executes the statement specified by *statement* for each of the values in the list, storing each value in the variable specified by *variable*. As in C, multiple statements can be combined with curly braces.
- The C shell adds the =~ and !~ operators. These are similar to string comparison operators, except that the right side is treated using filename globbing rules (for example, foo* matches files named foo, foot, fool, and so on). See "Special Characters Explained" (page 32) for more information about globbing.
- The C shell treats the == and != operators as string comparisons. To test for numeric equality or inequality, you can combine multiple comparison operators—`if ($A <= $B && !($A<B))` tests for numeric equality, for example.

For more information, see the manual page for `csh`.

The if Statement

The first control statement you should be aware of in shell scripting is the `if` statement. This statement behaves very much like the `if` statement in other programming languages, with a few subtle distinctions.

The first distinction is that the test performed by the `if` statement is actually the execution of a command. When the shell encounters an `if` statement, it executes the statement that immediately follows it. Depending on the return value, it will execute whatever follows the `then` statement. Otherwise, it will execute whatever follows the else statement.

The second distinction is that in shell scripts, many things that look like language keywords are actually programs. For example, the following code executes `/bin/true` and `/bin/false`.

```
# always execute
if true; then
    ls
else
    echo "true is false."
fi
# never execute
if false; then
    ls
fi
```

In both of these cases, an executable is being run—specifically, `/bin/true` and `/bin/false`. Any executable could be used here.

A return of zero (0) is considered to be true (success), and any other value is considered to be false (failure). Thus, if the executable returns zero (0), the commands following the `then` statement will be executed. Otherwise, the statements following the `else` clause (if one exists) will be executed.

The reason for this seemingly backwards definition of `true` and `false` is that most UNIX tools exit with an exit status of zero upon success and a non-zero exit status on failure, with positive numbers usually indicating a user mistake and negative numbers usually indicating a more serious failure of some sort. Thus, you can easily test to see if a program completed successfully by seeing if the exit status is the same as that of `true`.

One related statement that you should be familiar with is `elif`. This is similar to saying `else if` except that it does not require an additional `fi` at the end of the conditional, and thus results in more readable code.

For example:

```
#/bin/sh

read A
if [ "x$A" = "xfoo" ] ; then
    echo "Foo"
elif [ "x$A" = "xbar" ] ; then
    echo "Bar"
else
    echo "Other"
fi
```

This example reads a string from standard input and prints one of three things, depending on whether you typed “foo”, “bar’, or anything else. (The bracket syntax used in this example is explained in the next section, “The test Command And Bracket Notation” (page 26).)

The test Command And Bracket Notation

While the `if` statement can be used to run any executable, the most common use of the `if` statement is to test whether some condition is true or false, much like you would in a C program or other programming language. For example, the `if` statement is commonly used to see if two strings are equal.

Because the `if` statement runs a command, in order to use the `if` statement in this fashion, you will need a program to run that performs the comparison desired. Fortunately, one is built into the OS: `test`. The `test` executable is rarely run directly, however. Generally, it is invoked by running `[`, which is just a symbolic link or hard link to `/bin/test`.

Note: While the open bracket is a command, and there is a man page, you will have a hard time getting to it on the command line. Use:

```
man \\[
```

to see it (or just look at the man page for `test`).

In this form, the syntax of an `if` statement more closely resembles other languages. Consider the following example:

```
#! /bin/sh

FIRST_ARGUMENT="$1"
if [ x$FIRST_ARGUMENT = "xSilly" ] ; then
    echo "Silly human, scripts are for kiddies."
```

```
else
    echo "Hello, world $FIRST_ARGUMENT!"
fi
```

There are two things you should notice. First, the space before the equals sign is critical. This is the difference between assignment (no space) and comparison (space). The spaces around the brackets are also critical, as failure to include these spaces will result in a syntax error. (Remember, the open bracket is really just a command, and it expects that its last argument will be a close bracket by itself.)

Second, you should notice that the two arguments to the comparison are preceded by an 'x'. The reason for this is that the variable substitution occurs before this statement is executed. If you omit the 'x' and the value in `$FIRST_ARGUMENT` is empty this would evaluate to "`if [ = "Silly" ]`", which would be a blatant syntax error.

Another way to solve the empty variable problem is through the use of double quote marks. That way, even if the variable is empty, there is a placeholder. The following example uses double quote marks to test to see if a variable is empty:

```
if [ "$VARIABLE" = "" ] ; then
    echo "Empty variable \$VARIABLE"
fi
```

Now this example introduces another special character, the backslash. This is also known as a quote character because the character immediately after it is treated as though it were within quotes. Thus, in this case, the snippet prints the name of the variable `$VARIABLE` rather than its contents. This is described further in "Quoting Special Characters" (page 33).

The `test` command can also be used for various other tests, including the existence of a file, whether a path points to a directory, an executable, or a symbolic link, and so on. For example:

```
if [ -d "/System/Library/Frameworks" ] ; then
    echo "/System/Library/Frameworks is a directory."
fi
```

A complete list of switches for the test command can be found in the man page `test`.

The while Statement

In addition to the `if` statement, the Bourne shell also supports a `while` statement. Its syntax is similar.

```
while true; do
    ls
done
```

Like the `if` statement's `then` and `fi`, the while statement is bracketed by `do` and `done`. Much like the `if` statement, the while statement takes a single argument, which contains a command to execute. Thus, it is common to use the bracket command with `while` just as you would with `if`. For example:

```
while [ "x$FOO" != "x" ] ; do
    FOO="$(cat)";
done
```

Of course, this is a rather silly example. However, it does demonstrate one of the more powerful features in the Bourne shell scripting language: the `$()` operator, which inserts the output of one command into the middle of a statement. In the case above, the `cat` command is executed, and its standard output is stored in the variable `FOO`. This is described more in "Inline Execution" (page 34).

The for Statement

The most unusual control structure in this chapter is the `for` statement. The `for` statement in shell scripts is completely unlike its C equivalent (which requires numeric computation, as described in "Paint by Numbers" (page 45)), and actually behaves much like the `foreach` statement in various languages. It iterates through each of the items in a list. In the next example, the list is `*.JPG`, which the shell replaces with a list of files in the current directory that end in `.JPG`.

Without going into details about the regular expression syntax used by the `sed` command (this is described in more detail in "Regular Expressions Unfettered" (page 51)), the following script renames every file in the current directory that ends with `.JPG` to end in `.jpg`.

```
#!/bin/sh
for i in *.JPG ; do
    mv "$i" "$(echo $i | sed 's/\.JPG$/.x/')"
    mv "$(echo $i | sed 's/\.JPG$/.x/')" "$(echo $i | sed 's/\.JPG$/.jpg/')"
done
```

The `for` statement (by default) splits the file list on unquoted spaces. For example, the following script will print the letters "a" and "b" on separate lines, then print "c d" on a third line:

```
#!/bin/sh
for i in a b c\ d ; do
    echo $i
done
```

Under certain circumstances, you can change the way that the `for` statement splits lists by changing the contents of the variable `IFS`. The details of when this does and does not work are described in "Variable Expansion and Field Separators" (page 31).

The case statement

The final control statement in this chapter is the `case` statement. The `case` statement in shell scripts is similar to the C `switch` statement. It allows you to execute multiple commands depending on the value of a variable. The syntax is as follows:

```
case expression in
    [(] value | value | value | ... ) command; command; ... ;;
    [(] value | value | value | ... ) command; command; ... ;;
    ...
esac
```

You should notice three things about this syntax. First, each case is terminated by a double semicolon. Second, the opening parenthesis is optional and is frequently dropped by script authors. Third, a single set of commands can be applied to any number of values separated by the pipe (vertical bar) character (`|`).

For example, the following code sample prints the English names for the numbers 0–9, then prints them again.

```
#!/bin/sh

LOOP=0

while [ $LOOP -lt 20 ] ; do
        # The next line is explained in the
        # math chapter.
        VAL=`expr $LOOP % 10`

        case "$VAL" in
                ( 0 ) echo "ZERO" ;;
                ( 1 ) echo "ONE" ;;
                ( 2 ) echo "TWO" ;;
                ( 3 ) echo "THREE" ;;
                ( 4 ) echo "FOUR" ;;
                ( 5 ) echo "FIVE" ;;
                ( 6 ) echo "SIX" ;;
                ( 7 ) echo "SEVEN" ;;
                ( 8 ) echo "EIGHT" ;;
                ( 9 ) echo "NINE" ;;
                ( * ) echo "This shouldn't happen." ;;
        esac

        # The next line is explained in the
        # math chapter.
        LOOP=$((LOOP + 1))
done
```

You should notice the `( * )` case at the end. This is the equivalent of the `default` case in C. While that case will never be reached in this example, if you change the value of the modulo from 10 to any larger value, you will see that this case executes when no previous case matches the value of the expression.

The expr Command

No discussion of tests and comparisons would be complete without mentioning the `expr(1)` command. This command can perform various string comparisons and basic integer math. The math portions of the `expr` command are described in "The expr Command Also Does Math" (page 45).

The `expr` command is fairly straightforward. Each expression or token passed to the command must be surrounded by quotes if it may contain multiple words or characters that the shell considers special. For example, to compare two strings alphabetically, you could use the following command:

```
expr "This is a test" '<' "I am a person"
```

The following version would fail miserably because the shell would interpret the less-than sign as a redirect and would try to read from a file called "I am a person":

```
expr "This is a test" < "I am a person"
```

The details of quoting are described further in "Variables, Expansion, and Quoting" (page 31).

The `expr` command supports the usual complement of string comparisons (equality, inequality, less-than, greater-than, less-than-or-equal, and greater-than-or-equal).

In addition to these comparisons, the `expr` command can do several other tests: a logical "or" operator, a logical "and" operator, and a (fairly limited) basic regular expression matching operator.

The "or" operator (`|`) prints the first expression (`"$1"`) if it is nonempty and contains something other than the number zero (0). Otherwise, if the second string is nonempty and contains something other than the number zero, it prints the second expression (`"Untitled"`). If both strings are empty or zero, it prints the number zero.

You can use the "or" operator to substitute a default string using the or operator like this:

```
#!/bin/sh

NAME=`expr "$1" '|' "Untitled"`
echo "The chose name was $NAME"
```

Note: Because the `expr` command does not distinguish between the number zero (0) and an empty string, you should not use `expr` to test for an empty string if there is a possibility that the string might be `"0"`.

The "and" operator (`&`) is similar, returning either the first string (if both strings are nonempty) or zero (if either string is empty).

Finally, the `expr` command can work with basic regular expressions (*not* extended regular expressions) to a limited degree.

To count the number of characters from the beginning of the string (all expressions are implicitly anchored to the start of the string) up to and including the last letter 'i', you could write an expression like this:

```
STRING="This is a test"
expr "$STRING" : ".*i"
```

If the string does not match the expression, the `expr` command returns zero (0), which corresponds with the number of characters matched.

The most common use for this syntax is obtaining the length of a string, as shown in this snippet:

```
STRING="This is a test"
expr "$STRING" : ".*"
```

This same syntax can be used to return the text captured by the first set of parentheses in a basic regular expression. For example, to print the four characters immediately prior to the last occurrence of "est", you could write an expression like this one:

```
STRING="This is a test" expr "$STRING" : '.*\(....\)est'
```

Because this expression contains capturing parentheses, if the first string does not match the expression, the `expr` command returns an empty string.

For more information about writing basic regular expressions, read "Regular Expressions Unfettered" (page 51).

Variables, Expansion, and Quoting

In both the Bourne shell and the C shell, lines of code are processed in multiple passes. The first pass is a parsing pass in which the basic structure of the line of code is extracted. In this pass, quotation marks serve as delimiters between individual pieces of information. For example, you can print a letter immediately after the contents of a variable without a space by closing (and reopening if necessary) the enclosing double quotes immediately after the variable name.

The second pass is an expansion pass. In this pass, any variable is expanded and any inline execution is performed. If a variable contains special characters, the resulting text is further expanded unless that variable is surrounded by double quotes. This may cause unexpected behavior if, for example, a variable contains a wildcard character.

Note: While the expansion of a variable or command inline will not cause a syntax error by itself, it can change the behavior of the `eval` command. See "Data Structures, Arrays, and Indirection" (page 87) for more information.

Finally, the third pass is an execution pass. In this pass, the code is actually executed.

In some cases, you may need to change the way variable expansion takes place. You might want to use a nonstandard character to split a variable containing a list, change the way the shell handles special characters, or execute a command and substitute its output in the middle of another command. These techniques are described in the sections that follow.

Variable Expansion and Field Separators

In Bourne shell scripts, two operations are affected by the value of the `IFS` (internal field separators) shell variable: the `read` statement and variable expansion. The effect on the `read` statement is described separately in "Shell Variables and Printing" (page 14). In C shell scripts, only variable expansion is affected.

Whenever the shell expands a variable, the value of `IFS` comes into play. For example, the following script will print "a" and "b" on separate lines, then "c d" on a third line:

```
#!/bin/sh

IFS=":"
LIST="a:b:c d"
for i in $LIST ; do
        echo $i
done
```

This occurs *only* because the value on the right side of the `for` statement contains a variable (`LIST`) that is expanded by the shell. When the shell expands the variable, it replaces the colon with a space and quotes any spaces in the original string. In effect, by the time the `for` statement sees the values, the right side of the for statement contains `a b c\ d`, just as in the example shown in "The for Statement" (page 28).

If you insert the exact contents of `LIST` on the right side of the variable, this script will instead print "a:b:c" on one line and "d" on the other. This is why it is very important to choose record separators correctly.

Cross-Platform Compatibility Note: This treatment of record separators is specific to BASH. Some Bourne shell variants use `IFS` when the shell splits a list even if no expansion is involved.

To avoid unexpected behavior, you should avoid setting nonstandard values for `IFS` except when you are expanding a shell variable that depends on this.

As an exception, it is safe to modify IFS during a `read` statement. Be sure to save the original value in another variable and restore it afterwards, however, to avoid unexpected behavior elsewhere in the script.

Special Characters Explained

There are eight special characters in shell scripts: an asterisk (*), a question mark (?), curly braces ({ and }), square brackets ([and]) and single- and double-quote marks (‘ and “). You can use them as follows:

- Dollar sign ($)—the first character in variable expansion, shell builtin math, and inline execution.
- Asterisk (*)—a wildcard character that matches any number of characters. For example, `ls *.jpg` matches all files that end with the extension `.jpg`.
- Question mark (?)—a wildcard character that matches a single character. For example, `ls a?t.jpg` would match both `ant.jpg` and `art.jpg`.
- Curly braces—matches any of a series of options. For example, `ls *.{jpg,gif}` would match every file ending with either `.jpg` or `.gif`.
- Square brackets—matches any of a series of characters. For example, `ls a[rn]t.jpg` would match `art.jpg` and `ant.jpg`, but would not match `aft.jpg`. The syntax of these character classes is similar to character classes in regular expressions, but there are a number of subtle differences. For more information, see the Open Group’s page on pattern matching notation at http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html#tag_02_13.
- Double-quote marks—disables argument splitting on word boundaries (spaces) and shell expansion of most special characters within the quote marks. As a special exception, variables *are* expanded within double-quote marks. The contents of variables, however, are *not* expanded.
- Single-quote marks—disables argument splitting on word boundaries (spaces) and disables *all* shell expansion (including variables).

If your script accepts user input, these characters can produce unexpected results if you do not quote them properly. Consider the following example:

```
#!/bin/sh
echo "Filename?"
read NAME
ls $NAME
ls "$NAME"
```

If a user types `*.jpg` at the prompt, the first command lists all files ending in `.jpg` because the variable is expanded first, and then the expression within it is expanded. The second command lists a single file (or prints an error if you don’t have a file named `*.jpg`).

Quoting Special Characters

Sometimes, when writing shell scripts, you may need to explicitly include quotation marks, dollar signs, or other special characters in your output. The way that you do this depends on the context.

If the character you wish to quote is not within quote marks, you probably should be. Otherwise, you have to deal with *all* of the shell special characters (described in "Special Characters Explained" (page 32)) plus any new special characters that might be added in the future. This is particularly important if your script takes arbitrary user input and passes it as an argument.

However, if your script is not handling user input, you can quote a character by simply preceding it with a backslash (\). This tells the shell to treat it as a literal character instead of interpreting it normally. For example, the following code sample prints the word "Hello" enclosed in double-quotation marks.

```
echo \"Hello\"
```

If the character you wish to quote is within double quotes, the same rules apply. The only difference is that with the exception of dollar signs and the double-quote marks themselves, you don't need to quote special characters in this context. For example, to print the name of a variable followed by its value, you could write a statement like the following, which prints "The value of $VAR is 3" (with no quotes):

```
VAR=3
echo "The value of \$VAR is $VAR"
```

Similarly, you can quote a backslash with another backslash if you need to print it. For example, the following statement prints "This \ is a backslash." (again, without quotes):

```
echo "This \\ is a backslash."
```

If the character you wish to quote is within single quotes, shell expansion of special characters is disabled entirely. Thus, the only characters that are special are the single-quote marks themselves, because they terminate the single-quote context.

Because special character handling is disabled, a backslash does *not* quote anything between single-quote marks. Instead, a backslash is interpreted as literal text. Thus, to include a literal single quote within a double-quote context, you must terminate the single-quote context, then include the single quote (either by quoting it with a backslash or by surrounding it with double quotes), then start a new single-quote context.

For example, the following lines of code both print a popular phrase from an American children's television show:

```
echo 'It'\''s a beautiful day in the neighborhood.'
echo 'Won'"'"'t you be my neighbor?'
```

C Shell Note: The C shell does not support using a backslash to quote a character within a double-quoted string. Thus, in the C shell, you print a backslash like this:

```
echo "This \ is a backslash."
```

To print a literal dollar sign for a variable name, you must either put the dollar sign in single quotes or quote it with a backslash *outside* of any quote marks. For example:

```
echo "This is "'$'"FOO"
echo "This is "\$"FOO"
```

Both statements print the words “This is $FOO”.

Inline Execution

The Bourne shell provides two operators for executing a command and placing its output in the middle of another command or string. These operators are the `$()` operator and the back-tick (`) operator (not to be confused with a normal single quote).

One common way to use this is for generating a list of filenames inline. For example, the `grep` command, when passed the `-l` flag, returns a list of files that match. This is often combined with the `-r` flag, which makes grep search recursively for files within any directories that it encounters in its file list. Thus,if you want to edit any files containing "`myname`" with vi, for example, you could do it like this:

```
vi $(grep -rl myname directory_of_files)
```

You can, however, use this to execute any command. There is one small caveat you should be aware of, however. The back-tick operator cannot be nested. For example, the following command produces an error:

```
FOO=1; BAR=3
echo "Try this command: `echo $FOO + "`expr $BAR + 1`"`"
```

This fails because the echo command ends at the second back-tick. Thus, the command executed is `echo $FOO + "`. If you need to nest inline execution, you can use the `$()` operator for the nested command. For example, the previous example can be written correctly as follows:

```
FOO=1; BAR=3
echo "Try this command: `echo $FOO + "$(expr $BAR + 1)"`"
```

You should notice that double quotation marks can be safely nested within a command enclosed by either back-ticks or `$()`.

Note: Evaluation of inline commands, much like expansion of variables, occurs *after* the statement itself is fully parsed. Thus, it is safe to use either the back-tick or `$()` operator even if the command may produce double-quote marks in its output. You do not need to quote the resulting content in any way.

C Shell Note: The C shell only partially supports this. It does not support the `$()` syntax. Its support for the back-tick syntax is somewhat limited; newlines in the result are always stripped and replaced with spaces. If you need to preserve newlines, you should store the results in a temporary file instead of in a shell variable.

Result Codes, Subroutines, Scoping, and Sourcing

No procedural programming language would be complete without some notion of subroutines, functions, or other such constructs. The Bourne shell is no exception.

In the Bourne shell, there are two basic ways to approach subroutines. The first is through executing outside tools (which may include a script executing itself recursively). This was described briefly in "Basic Control Statements" (page 24). However, there are other techniques for obtaining result code information from external scripts. These are described in "Working With Result Codes" (page 35).

The second way to approach subroutines (and one which generally results in better performance) is through the use of actual subroutines. These are described in "Basic Subroutines" (page 36).

The scoping rules for shell subroutines differ from the scoping rules for most other programming languages. Shell script variable scoping is explained in "Variable Scoping" (page 38).

Finally, you may find it useful to include one entire shell script inside another. This subject is covered in "Including One Shell Script Inside Another" (page 38).

Working With Result Codes

Result codes, also known as return values, exit statuses, and probably several other names, are one of the more critical features of shell scripting, as they play a role in almost every aspect of script execution.

Whenever a command executes (including the open bracket shell builtin used as part of the `if` and `while` statements), a result code is generated. If the command exits successfully, the result is usually zero (`0`). If the command exits with an error, the result code will vary according to the tool. (See the documentation for the tool in question for a list of result codes.) The possible range of result codes is 0-255.

There are two ways of testing to see if a script executes correctly. The first is with an immediate test using the if statement. For example:

```
if ls mysillyfilename ; then
    echo "File exists."
fi
```

Note: This is not the best way of testing whether a file exists. This is only an example of a tool that returns a different exit status depending on whether it was successful at performing a task.

For more information about how to test for file existence using the if statement, see "The test Command And Bracket Notation" (page 26).

The second way is by testing the last exit status returned. The exit status is stored in the shell variable `$?`. For example:

```
ls mysillyfilename
if [ $? = 0 ] ; then
    echo "File exists."
fi
```

These two code examples should generate the same output.

Basic Subroutines

Subroutines in the Bourne shell look very much like C functions without the argument list. You call these subroutines just like you would run a program, and subroutines can be used anywhere that you would use an executable.

Here is a simple example that prints "Arg 1: This is an arg" using a shell subroutine:

```
#!/bin/sh

mysub()
{
        echo "Arg 1: $1"

}

mysub "This is an arg"
```

Just as shell script arguments are stored in shell-local variables named $1, $2, and so on, so too are the arguments to shell subroutines. In fact, in most ways, shell subroutines behave exactly like executing an external script. One place where they behave differently is in variable scoping. See "Variable Scoping" (page 38) for more information.

In general, a subroutine can do anything that a shell script can do. It can even return an exit status to the calling part of the shell script. For example:

```
#!/bin/sh

mysub()
{
        return 3
}

mysub "This is an arg"
echo "Subroutine returned $?"
```

Note: Be careful *not* to use `exit` in the subroutine. If you do, the entire script will exit, not just the subroutine. This is one area in which subroutines behave differently than separate scripts behave.

Anonymous Subroutines

The Bourne shell allows you to group more than one command together and treat them both as a separate command. In effect, this is creating an anonymous subroutine inline.

For example, if you want to copy a large number of files from one place to another, you could use `cp`, but this may not be semantically ideal for any number of reasons. Another option is to use `tar` to create an archive on standard output, then pipe that to a second instance of `tar` that extracts the archive.

The basic commands needed are show below. The first command in this example archives the listed files and prints the archive contents to standard output. The second command takes an archive form standard output and extracts the files.

```
tar -cf - file1 file2 file3 ...
tar -xf -
```

Thus, to copy files from one place to another, you could pipe the first `tar` command to the second one. However, there's a problem with that: because the second `tar` is running in the same directory, you are extracting the files on top of themselves. If you're lucky, nothing happens at all. In the worst case scenario, you could lose files this way.

Thus, you need run two commands on the right side of the pipe: a `cd` command to change directories before extracting the archive and the `tar` command itself. You can do this with an anonymous subroutine.

Here is a simple example:

```
tar -cf - file1 file2 file3 | \
    { cd "/destination" ; tar -xf - ; }
```

Notice the semicolon before the close curly brace. This is required. Also notice the space after the opening curly brace. This is also required. Forgetting either of these will cause a syntax error.

Of course, as written, there is still some risk involved in using this code. If the destination directory does not exist, the `cd` command fails, and the `tar` command executes in the wrong directory. To solve this problem, you should check the exit status of the first command before running the second one.

For example:

```
tar -cf - file1 file2 file3 | \
    { if cd "/destination" ; then tar -xf - ; fi; }
```

This version will execute the `cd` command, then execute the second `tar` command *only* if the `cd` command was successful.

Variable Scoping

Subroutines execute within the same shell instance as the main shell script. (This is not always the case in other shells like C shell, but it is true for Bourne shell scripts.) The result is that all shell variables are, by default, shared between subroutines and the main program body. This creates a bit of a problem when writing recursive code.

Fortunately, variables do not have to remain global. To declare a variable local to a given subroutine, use the `local` statement.

```
#!/bin/sh

mysub()
{
        local MYVAR
        MYVAR=3
        echo "SUBROUTINE: MYVAR IS $MYVAR";
}

MYVAR=4
echo "MYVAR INITIALLY $MYVAR"
mysub "This is an arg"
echo "MYVAR STILL $MYVAR"
```

This script will tell you that the initial value is 4, the value was changed to 3 in the subroutine, and remains 4 when the subroutine returns. Were it not for a `local` declaration of `MYVAR` in the subroutine, the subsequent change to `MYVAR` would have propagated back to the main body of the script.

Much like the `export` statement, the `local` statement can be used at the beginning of an assignment statement as well. For example, the previous subroutine could have contained the following line instead:

```
local MYVAR=3
```

In either case, any changes to the variable `MYVAR` would remain local to the subroutine through to the end of the subroutine. If the subroutine calls itself recursively, a new copy of `MYVAR` will be created for each call to the subroutine, resulting in a call stack much like local variables in C or other languages.

Note: Changes to this variable in other subroutines without a `local` declaration of `MYVAR` will still result in modifications to the global copy of `MYVAR`.

Including One Shell Script Inside Another

As with any programming language that includes subroutines, it is often useful to build up a library of common functions that your scripts can use. To avoid duplicating this content, the Bourne shell scripting language supports a mechanism to include one shell script inside another by reference. This process is referred to as "sourcing".

For example, create a file containing the subroutine `mysub` from "Variable Scoping" (page 38). Call it `mysub.sh`. To use this subroutine in another script, you can do the following:

```
#!/bin/sh
MYVAR=4
source /path/to/mysub.sh
echo "MYVAR INITIALLY $MYVAR"
mysub "This is an arg"
echo "MYVAR STILL $MYVAR"
```

This script will do exactly the same thing as the script in the previous section. The only difference is that the subroutine used is in a different file.

There is another, shorter, way to write the same thing using the period (`.`) character. For example:

```
#!/bin/sh
MYVAR=4
. /path/to/mysub.sh
echo "MYVAR INITIALLY $MYVAR"
mysub "This is an arg"
echo "MYVAR STILL $MYVAR"
```

This code does exactly the same thing as the previous example. The `source` command is more popular among former C shell programmers, while the period (`.`) version is more popular among Bourne shell purists. Both versions are perfectly cromulent, however.

These examples are not as straightforward as they seem, however. While this works very well for including subroutines, you cannot always use this in place of executing an outside script, as execution and sourcing behave very differently with respect to variables. The following example demonstrates this:

```
#!/bin/sh
# Save as sourcetest1.sh
MYVAR=3
source sourcetest2.sh
echo "MYVAR IS $MYVAR"

#!/bin/sh
# Save as sourcetest2.sh
MYVAR=4
```

You will notice that the second script changed the value of a variable that was local to the first script. Unlike executing a script as a normal shell command, executing a script with the `source` command results in the second script executing within the same overall context as the first script. Any variables that are modified by the second script will be seen by the calling script. While this can be very powerful, it is easy to clobber variables if you aren't careful.

Background Jobs and Job Control

For end-user convenience in the days of text terminals before the advent of tools like `screen`, the C shell contains job control features that allow you to start a process in the background, then go off and work on other things, bringing these background tasks into the foreground, suspending foreground tasks to complete them later, and continuing these suspended tasks as background tasks.

Over the years, many modern Bourne shell variants including `bash` and `zsh` have added similar support. The details of using these commands from the command line is beyond the scope of this document, but in brief, control-Z suspends the foreground process, `fg` brings a suspended or background job to the foreground, and `bg` causes a job to begin executing in the background.

Up until this point, all of the scripts have involved a single process operating in the foreground. Indeed, most shell scripts operate in this fashion. Sometimes, though, parallelism can improve performance, particularly if the shell script is spawning a processor-hungry task. For this reason, this section describes programmatic ways to take advantage of background jobs in shell scripts.

Note: All Bourne shell variants support running a command in the background. However, the information obtained about these jobs varies from shell to shell, and pure Bourne shell implementations do not provide this information at all. Thus, when writing scripts that use this functionality, you should be aware that you are significantly limiting the portability of your script when you use BASH-specific or Zsh-specific builtins.

Also note that these examples are specific to BASH. For Zsh, there are subtle differences in the formatting of job status that will require changes to various bits of code. Making this code work in other shells is left as an exercise for the reader.

To start a process running in the background, add an ampersand at the end of the statement. For example:

```
sleep 10 &
```

This will start a sleep process running in the background and will immediately return you to the command line. Ten seconds later, the command will finish executing, and the next time you hit return after that, you will see its exit status. Depending on your shell, it will look something like this:

```
[1]+  Done                    sleep 10
```

This indicates that the sleep command completed execution. A related feature is the `wait` builtin. This command causes the shell to wait for a specified background job to complete. If no job is specified, it will wait until all background jobs have finished.

The next example starts several commands in the background and waits for them to finish.

```
#!/bin/bash

function delayprint()
{
    local TIME;
    TIME=$1
    echo "Sleeping for $TIME seconds."
    sleep $TIME
    echo "Done sleeping for $TIME seconds."
}

delayprint 3 &
delayprint 5 &
delayprint 7 &
wait
```

This is a relatively simple example. It executes three commands at once, then waits until all of them have completed. This may be sufficient for some uses, but it leaves something to be desired, particularly if you care about whether the commands succeed or fail.

The following example is a bit more complex. Because the job control mechanism in most Bourne shell variants was designed primarily for interactive use rather than programmatic use, obtaining information about background jobs can be somewhat clumsy. Fortunately, there are few things that a well-written regular expression can't fix.

Note: Regular expressions are described in "Regular Expressions Unfettered" (page 51). For the purposes of this example, it is sufficient to understand that the function `jobidfromstring` takes a job string like the one shown previously and prints out the first single digit or multi-digit number by itself.

```
#!/bin/bash

function jobidfromstring()
{
        local STRING;
        local RET;

        STRING=$1;
        RET="$(echo $STRING | sed 's/^[^0-9]*//' | sed 's/[^0-9].*$//')"

        echo $RET;
}

function delayprint()
{
        local TIME;
        TIME=$1
        echo "Sleeping for $TIME seconds."
        sleep $TIME
        echo "Done sleeping for $TIME seconds."
}

delayprint 3 &
DP3=`jobidfromstring $(jobs %%)`

delayprint 5 &
DP5=`jobidfromstring $(jobs %%)`

delayprint 7 &
DP7=`jobidfromstring $(jobs %%)`

echo "Waiting for job $DP3";
wait %$DP3

echo "Waiting for job $DP5";
wait %$DP5

echo "Waiting for job $DP7";
wait %$DP7

echo "Done."
```

In this example, you pass a job number argument to the `jobs` builtin to tell it which job you want to find out information about. Job numbers begin with a percent (%) sign and are normally followed by a number. In this case, however, a second percent sign is used. This is one of a number of special job "numbers" that the shell provides. In this case, it tells the jobs command to output information about the last command that was executed in the background.

The result of this `jobs` command is a status string like the one shown earlier. This is passed as a series of arguments to the `jobidfromstring` function, which then prints the job ID by itself, which gets stored into one of the variables `DP3`, `DP5`, or `DP7`.

Finally, the script ends with a series of `wait` commands. Like the `jobs` command, these commands can take a job ID. This job ID consists of a percent sign followed by the number (obtained from one of the `DP*` variables).

The final example shows how to execute a limited number of concurrent jobs in which the order of job completion is not important.

```
#!/bin/bash

MAXJOBS=3


function jobidfromstring()
{
        local STRING;
        local RET;

        STRING=$1;
        RET="$(echo $STRING | sed 's/^[^0-9]*//' | sed 's/[^0-9].*$//')"

        echo $RET;
}

function spawnjob()
{
        echo $1 | bash
}

function clearToSpawn
{
    local JOBCOUNT="$(jobs -r | grep -c .)"
    if [ $JOBCOUNT -lt $MAXJOBS ] ; then
        echo 1;
        return 1;
    fi

    echo 0;
    return 0;
}

JOBLIST=""

COMMANDLIST='ls
echo "sleep 3"; sleep 3; echo "sleep 3 done"
echo "sleep 10"; sleep 10 ; echo "sleep 10 done"
echo "sleep 1"; sleep 1; echo "sleep 1 done"
echo "sleep 5"; sleep 5; echo "sleep 5 done"
```

```
echo "sleep 7"; sleep 7; echo "sleep 7 done"
echo "sleep 2"; sleep 2; echo "sleep 2 done"
'

IFS="
"

for COMMAND in $COMMANDLIST ; do
        while [ `clearToSpawn` -ne 1 ] ; do
                sleep 1
        done
        spawnjob $COMMAND &
        LASTJOB=`jobidfromstring $(jobs %%)`
        JOBLIST="$JOBLIST $LASTJOB"
done

IFS=" "

for JOB in $JOBLIST ; do
        wait %$JOB
        echo "Job $JOB exited with status $?"
done

echo "Done."
```

Most of the code here is straightforward. It is worth noting, however, that in the function `clearToSpawn`, the `-r` flag must be passed to the `jobs` command to restrict output to currently running jobs. Since this shell script does not actually wait for any process until after it has finished starting processes, the `jobs` builtin would otherwise return a list that included completed jobs.

Warning: While it is tempting to put the while loop inside the `clearToSpawn` subroutine, if you do so, the program will wait forever. The status of jobs does not get updated by the shell until script execution returns to the main body of the program.

The `-c` flag to `grep` causes it to return the number of matching lines rather than the lines themselves, and the period causes it to match on any nonblank lines (those containing at least one character). Thus, the `JOBCOUNT` variable contains the number of currently running jobs, which is, in turn, compared to the value `MAXJOBS` to determine whether it is appropriate to start another job or not.

Paint by Numbers

Using math in shell scripts is one area that is often ignored by shell scripting documentation—probably because so few people actually understand the subject. Shell scripts were designed more for string-based processing, with numerical computation as a bit of an afterthought, so this should come as no surprise.

This chapter mainly covers basic integer math operations in shell scripts. More complicated math is largely beyond the ability of shell scripting in general, though you can do such math through the use of inline Perl scripts or by running the `bc` command. These two techniques are described in "Beyond Basic Math" (page 48).

The expr Command Also Does Math

In shell scripts, numeric calculations are done using the command `expr`. This command takes a series of arguments, each of which must contain a single token from the expression to be evaluated. Each number, or symbol must thus be a separate argument.

For example, the expression `(3*4)+2` would be written as:

```
expr '(' '3' '*' '4' ')' '+' '2'
```

The command will print the result (`14`) to its standard output,

Note: Each argument in this example is surrounded by single quotes. This prevents the shell from trying to interpret the contents of the argument. Certain things like parentheses and comparison operators have special meaning to the shell, so without these single quotes, the command would not behave as expected.

If an argument contains a shell variable, double quotes must be used, since otherwise the shell variable would not be expanded at all. Thus in some cases, you will see examples in this chapter containing double quotes. However, for simplicity, the examples in this chapter will generally use single quotes unless there is a specific reason that double quotes are necessary.

For numerical comparisons, the same basic syntax is used. To test the truth of the inequality `3 < -2`, you would use the following statement:

```
expr '3' '<' '-2'
```

This will return a zero (0) because the statement is not true. If it were true, it would return a one (1).

> **Warning:** This mathematical expression of true is exactly the opposite of that returned by the commands `true` and `false`. This is often confusing to people who are new to shell scripting. The values returned by `true` and `false` are intended to represent return values for shell scripts and command-line tools, not numerical computation. Command-line tools and scripts typically return 0 on success, 1 on an invalid argument, or a negative value for serious failures. You should avoid comparing the results returned by `expr` with the return value of `true` or `false`.

The most common place to use this command is as part of a loop in a shell script. What follows is a simple example of a for-next loop written in a shell script:

```
COUNT=0
while [ $COUNT -lt '4' ] ; do
    echo "COUNT IS $COUNT"
    COUNT="$(expr "$COUNT" '+' '1')"
done
```

This is equivalent to the following bit of C:

```
int i;
for (i=0; i<4; i++) {
    printf("COUNT IS %d\n", i);
}
```

> **Note:** The `expr` command can also be used for string comparison. This use is described in the similarly titled section "The expr Command" (page 29) in "Shell Script Basics" (page 11).

The Easy Way: Parentheses

Another way to do math operations in `some` Bourne shell dialects is with double parentheses inline. The example below illustrates this technique:

```
echo $((3 + 4))
```

This form is much easier to use than the `expr` command because it is somewhat less strict in terms of formatting. In particular, with the exception of variable decoding, shell expansion is disabled. Thus, operators like less than and greater than do not need to be quoted.

This form is not without its problems, however. In particular, it is not as broadly compatible as the use of `expr`. This form is an extension added by the Korn shell (`ksh`), and later adopted by the Z shell (`zsh`) and the Bourne Again shell (`bash`). In a pure Bourne shell environment, this syntax will probably fail.

While most modern UNIX-based and UNIX-like operating systems use BASH to emulate the Bourne shell, if you are trying to write scripts that are more generally usable, you should use `expr` to do integer math, as described in "The expr Command" (page 45).

Other Comparisons

As mentioned in,"Shell Script Basics" (page 11), the shell scripting language contains basic equality testing without the use of the `expr` command. For example:

```
if [ 1 = 2 ] ; then
    echo "equal"
else
    echo "not equal"
fi
```

This code will work as expected. However, it isn't doing what you might initially think it is doing; this is performing a string comparison, *not* a numeric comparison. Thus the following code will not behave the way you would expect if you assumed it was comparing the numerical values:

```
if [ 1 = "01" ] ; then
    echo "equal"
else
    echo "not equal"
fi
```

It will print the words "not equal", as the strings "1" and "01" are not the same string.

Warning: Do not inadvertently perform a redirect instead of an inequality test. Take the following code for example:

```
if [ 2 > 3 ] ; then
    echo greater
fi
```

This will be true even though the comparison should be false. This is because no comparison is taking place. This is actually redirecting the output of the bracket command (an empty string) into a file called 3, which is probably not what you want.

The same thing occurs if you use the `expr` command without enclosing the less than or greater than operators in quotes.

This can also be a problem even when working with the `expr` command if your script takes user input. The `expr` command expects a number or symbol per argument. If you feed it something that isn't just a number or symbol, it will treat it as a string, and will perform string comparison instead of numeric comparison.

The following code demonstrates this in action:

```
expr '1' '+' '2'
expr ' 1' '+' '2'
expr '2' '<' '1'
expr ' 2' '<' '1'
```

The first line will print the number `3`. The second line will give an error message. When doing addition, this is easy to detect. When doing comparisons, however, as shown in the following two lines, the results are more insidious. The number `2` is clearly greater than the number `1`. In string comparison, however, a space sorts before any letter or number. Thus, the third line prints a `0`, while the fourth line prints a `1`. This is probably not what you want.

As with most things in shell scripting, there are many ways to solve this problem, depending on your needs. If you are only worried about spaces, and if the purpose for the comparison is to control shell execution, you can use the numeric evaluation routines built into `test`, as described in the `test` man page.

For example:

```
MYNUMBER=" 2" # Note this is a string, not a number.
# Force an integer comparison.
if [ "$MYNUMBER" -gt '1' ] ; then
    echo 'greater'
fi
```

However, while this works for trivial cases, there are a number of places where this is not sufficient. For example, this cannot be used if:

- Floating point comparison is needed (as described in "Beyond Basic Math" (page 48)).
- The value is preceded by a dollar sign or similar.
- The intended use is as a numerical truth value in a more complicated mathematical expression (without splitting the expression).

A common way to solve such problems is to process the arguments with a regular expression. For example, to strip any non-numeric characters from a number, you could do the following:

```
MYRAWNUMBER=" 2" # Note this is a string, not a number.

# Strip off any characters that aren't in the range of 0-9
MYNUMBER="$(echo "$MYRAWNUMBER" | sed 's/[^0-9]//g')"

expr "$MYNUMBER" '<' '1'
```

This would result in a comparison between the number `2` and the number `1`, as expected.

For more information on regular expressions, see "Regular Expressions Unfettered" (page 51).

Beyond Basic Math

The shell scripting language provides only the most basic mathematical operations on integer values. In most cases, this is sufficient. However, sometimes you will need to exceed those limitations to perform more complicated mathematical operations.

There are two main ways to do floating point math (and other, more sophisticated math). The first is through the use of inline Perl code, the second is through the use of the `bc` command. This section presents both forms briefly.

Floating Point Math Using Inline Perl

The first method of doing shell floating point math, inline Perl, is the easiest to grasp. To use this method, you essentially write a short perl script, then substitute shell variables into the script, then pass it to the perl interpreter, either by writing it to a file or by passing it in as a command line argument.

Note: Length limitations apply when passing in a Perl script by way of a command line argument. The exact limitations vary from one OS to another, but are generally in the tens of kilobytes. If your script needs to be longer, it should be written out to a file.

The following example demonstrates basic floating point math using inline Perl. It assumes a basic understanding of the Perl programming language.

```
#!/bin/sh
PI=3.141592654
RAD=7
AREA=$(perl -e "print \"The value is \".($PI * ($RAD*$RAD)).\"\n\";")
echo $AREA
```

Under normal circumstances, you probably would not want to print an entire string when doing this. However, the use of the string was to demonstrate an important point. Perl evaluates strings between single and double quote marks differently, so when doing inline Perl, it is often necessary to use double quotes. However, the shell only evaluates shell variables within double quotes. Thus, the double quote marks in the script had to be quoted so that they would actually get passed to the Perl interpreter instead of ending or beginning new command-line arguments.

This need for quoting can prove to be a challenge for more complex inline code, particularly when regular expressions is involved. In particular, it can often be tricky figuring out how many backslashes to use when quoting the quoting of a quotation mark within a regular expression. Such issues are beyond the scope of this document, however.

Floating Point Math Using the bc Command

The `bc` command, short for basic calculator, is a POSIX command for doing various mathematical operations. The `bc` command offers arbitrary precision floating point math, along with a built-in library of common mathematical functions to make programming easier.

Note: The most common version of `bc` (and the one included in Mac OS X) is GNU `bc`, which offers a number of extensions beyond those available in the POSIX version. For cross-platform compatibility, you should generally avoid these extensions if possible. If you specify the `-s` flag to GNU `bc`, it will disable the GNU extensions and will thus emulate the POSIX version.

The `bc` command takes its input from its standard input, not from the command line. If you pass it command line arguments, they are interpreted as file names to be executed, which is probably not what you want to do when executing math operations inline in a shells script.

Here is an example of using `bc` in a shell script:

```
#!/bin/sh
```

```
PI=3.141592654
RAD=7
AREA=$(echo "$PI * ($RAD ^ 2)" | bc)
echo "The area is $AREA"
```

The `bc` command offers much more functionality than described in this section. This section is only intended as a brief synopsis of the available functionality. For full usage notes, see the man page for `bc`.

Regular Expressions Unfettered

Regular expressions are a powerful mechanism for text processing. You can use regular expressions to search for a pattern within a block of text, to replace bits of that text with other bits of text, and to manipulate strings in various other subtle and interesting ways.

There are three basic types of regular expressions: basic regular expressions, extended regular expressions, and Perl regular expressions. This chapter explains the three at a high level, then points out areas in which they diverge.

For the purposes of this chapter, you should paste the following lines of text into a text file with UNIX line endings (newline):

```
Mary had a little lamb,
its fleece was white as snow,
and everywhere that Mary went,
the lamb was sure to go.
A few more lines to confuse things:
Marylamb had a little.
This is a test.  This is only a test.
Mary was married.  A lamb was nearby.
Mary, a little lamb, and my grocer's freezer...
Mary a lamb.
Marry a lamb.
Mary had a lamb looked like a lamb.
I want chocolate for Valentine's day.
This line contains a slash (/).
This line contains a backslash (\).
This line contains brackets ([]).
Why is mary lowercase?
What about Mary, Mary, and Mary?
const people fox
constant turtles bear
constellation Libra
The quick brown fox jumped over the lazy dog.
```

Save this into a file called `poem.txt`.

Regular Expression Syntax

The fundamental format for regular expressions is one of the following, depending on what you are trying to do:

```
/search_pattern/modifiers
command/search_pattern/modifiers
command/search_pattern/replacement/modifiers
```

The first syntax is a basic search syntax. In the absence of a command prefix, such a regular expression returns the lines matching the search pattern. In some cases, the slash marks may be (or must be) omitted—in the pattern argument to the `grep` command, for example.

The second syntax is used for most commands. In this form, some operation occurs on lines matching the pattern. This may be a form of matching, or it may involve removing the portions of the line that match the pattern.

The third syntax is used for substitution commands. These can be thought of as a more complex form of search and replace.

For example, the following command will search for the word 'test' within the specified file:

```
# Expression: /test/
grep 'test' poem.txt
```

Note: Note that `grep` expects the leading and trailing slashes in the regular expression to be removed.

The availability of commands and flags varies somewhat between regular expression variants, and is described in the relevant sections.

Positional Anchors and Flags

A common way to significantly alter regular expression matching is through the use of positional anchors and flags.

Positional anchors allow you to specify the position within a line of text where an expression is allowed to match. There are two positional anchors that are regularly used: caret (^) and dollar ($). When placed at the beginning or end of an expression, these match the beginning and end of a line of text, respectively.

For example:

```
# Expression: /^Mary/
grep "^Mary" < poem.txt
```

This will match the word "Mary", but only when it appears at the beginning of a line. Similarly, the following will match the word "fox," but only at the end of a line:

```
# Expression: /fox$/
grep "fox$" < poem.txt
```

The other common technique for altering the matching behavior of a regular expression is through the use of flags. These flags, when placed at the end of a regular expression, can change whether a regular expression is allowed to match across multiple lines, whether the matching is case sensitive or insensitive, and various other aspects of matching.

Note: Different tools support different flags, and not all flags are supported with all tools. The `grep` command-line tool uses command-line flags instead of flags in the expression itself.

The most commonly used flag is the global flag. By default, only the first occurrence of a search term is matched. This is mainly an issue when doing substitutions. The global flag changes this to alter every match instead of just the first.

For example:

```
# Expression: s/Mary/Joe/
sed "s/Mary/Joe/" < poem.txt
```

This will replace only the first occurrence of "Mary" with "Joe." By adding the global flag to the expression, it will instead replace every occurrence, as shown in the following example:

```
# Expression s/Mary/Joe/g
sed "s/Mary/Joe/g" < poem.txt
```

Wildcards and Repetition Operators

One of the most common ways to enhance searching through regular expressions is with the use of wildcard matching.

A wildcard is a symbol that takes the place of any other symbol. In regular expressions, a period (`.`) is considered a wildcard, as it matches any single character. For example:

```
# Expression: /wa./
grep 'wa.' poem.txt
```

This matches lines containing both "was" and "want" because the dot can match any character.

Wildcards are typically combined with repetition operators to match lines in which only a portion of the content is known. For example, you might want to search for every line containing "Mary" with the word "lamb" appearing later. You might specify the expression like this:

```
# Expression: /Mary.*lamb/
grep "Mary.*lamb" poem.txt
```

This would search for Mary followed by zero or more characters, followed by lamb.

Of course, you probably want at least one character between those to avoid matches for strings containing "Marylamb". You can construct this expression in one of the following ways:

```
# Expression (Basic): /Mary.\+lamb/
# Expression (Extended): /Mary.+lamb/
# Expression: /Mary..*lamb/
grep "Mary.\+lamb" poem.txt
grep -E "Mary.+lamb" poem.txt    # extended regexp
```

```
grep "Mary..*lamb" poem.txt
```

Note: The appearance of the plus operator differs depending on whether you are using basic or regular expressions.

The first dot in the third expression indicates that there will be at least one character. The dot-asterisk afterwards indicates that there will be zero or more characters. Thus, these three statements are equivalent.

The final useful repetition operator is the question mark operator. This operator will match zero or one repetitions of whatever comes before it.

Note: Like the plus operator, this differs in appearance depending on whether you are using basic or extended regular expressions.

For example, if you want to match both Mary and Marry, you might use an expression like this:

```
# Expression (Basic): /Marr\?y/
# Expression (Extended): /Marr?y/
grep "Marr\?y" poem.txt
grep -E "Marr?y" poem.txt
```

The question mark causes the preceding r to be optional, and thus, this expression will match lines containing either "Mary" or "Marry."

In summary, the basic wildcard and repetition operators are:

period (`.`)—wildcard; matches a single character.

question mark (`\?` or `?`)—matches 0 or 1 of the previous character, grouping, or wildcard. (This operator differs depending on whether you are using basic or extended regular expressions.)

asterisk(`*`)—matches zero or more of the previous character, grouping, or wildcard.

plus(`\+` or `+`)—matches one or more of the previous character, grouping, or wildcard. (This operator differs depending on whether you are using basic or extended regular expressions.)

Character Classes and Groups

Searching for certain keywords can be useful, but it is often not enough. It is often useful to search for the presence or absence of key characters at a given position in a search string.

For example, assume that you require the words Mary and lamb to be within the same sentence. To do this, you would need to only allow certain characters to appear between the two words. This can be achieved through the use of character classes.

There are two basic types of character classes: predefined character classes and custom, or user-defined character classes. These are described in the following sections.

Predefined Character Classes

Most regular expression languages support some form of predefined character classes. When used between square braces, these define commonly-used sets of characters.

- `:alnum:`—all alphanumeric characters (a-z, A-Z, and 0-9).
- `:cntrl:`—all control characters (ASCII 0-31).
- `:lower:`—all lowercase letters (a-z).
- `:space:`—all whitespace characters (space, tab, newline, carriage return, form feed, and vertical tab).
- `:alpha:`—all alphabetic characters (a-z, A-Z).
- `:digit:`—all numbers.
- `:print:`—all printable characters (opposite of :cntrl:).
- `:upper:`—all uppercase letters.
- `:blank:`—all whitespace within a line (spaces or tabs).
- `:graph:`—all alphanumeric or punctuation characters.
- `:punct:`—all punctuation characters
- `:xdigit:`—all hexadecimal digits (0-9, a-f, A-F).

For example, the following would be another way to match any sentence containing Mary and lamb:

```
/Mary[:alpha::digit::blank:][:alpha::digit::blank:]*lamb/
```

Custom Character Classes

In addition to the predefined character classes, regular expression languages also allow custom, user-defined character classes. These custom character classes just look like a list of characters surrounded by square brackets.

For example, if you only want to allow spaces and letters, you might create a character class like this one:

```
# Expression: /Mary[a-z A-Z]*lamb/
grep "Mary[a-z A-Z]*lamb" poem.txt
```

In this example, there are two ranges ('a' through 'z' and 'A' through 'Z') allowed, as well as the space character. Thus, any letter or space would match this pattern, but other things including period will not. Thus, this line matches the first line of the poem, but does not match the later line that begins with "Mary was married."

However, this pattern also did not match the line containing a comma, which was not really the intent. Listing every reasonable range of characters with a single omission would be prohibitively large, particularly if you want to include high ASCII characters, control characters, and other potentially unprintable characters.

Fortunately, there is another special operator, the caret (^). When placed as the first character of a character class, matching is reversed. Thus, the following expression would match any character other than a period:

```
# /Mary[^.]*lamb/
grep "Mary[^.]*lamb" poem.txt
```

Grouping Operators

As mentioned previously, regular expressions also have a notion of grouping. The purpose of grouping is to treat multiple characters as a single entity, usually for the purposes of modifying that entity with a repeat operator. This grouping is done using parentheses or quoted parentheses, depending on the regular expression dialect being used.

For example, say that you want to search for any string that contains the word "Mary" followed optionally by the word "had", followed by the word "a". You might write this expression like this:

```
#Expression (Basic): /Mary \(had \)\?a/
#Expression (Extended): /Mary (had )?a/
grep "Mary \(had \)\?a" poem.txt
grep -E "Mary (had )?a" poem.txt
```

Note: The grouping operator and optional operator differ depending on which program is processing the regular expression. The tools `sed`, `awk`, and `grep` use basic regular expressions (by default), and thus, these operators must be quoted. Any tools that use extended regular expressions use the bare operators.

Also note that the `-E` flag enables extended regular expressions in `grep`.

The flag to enable extended regular expressions in `sed` differs among different versions of the tool. For this reason, you should use basic regular expressions if at all possible when working with `sed`.

Extended regular expression extensions extend the capture syntax in such a way that expressions enclosed in parentheses will match any one of a series of smaller expressions separated by a vertical bar (`|`) operator. For example, to search for Mary, lamb, or had, you might use this expression:

```
#Expression (Extended): /(Mary|had|lamb)/
grep -E '(Mary|had|lamb)' poem.txt
```

Note: The syntax for grouping also results in a capture. This process is described in "Capture Operators and Variables" (page 58).

Using Empty Subexpressions

Sometimes, when working with groups, you may find it necessary to include an optional group. It may be tempting to write such an expression like this:

```
# Expression (Extended): /const(ant|ellation|) (.*)/
```

In an odd quirk, however, some command-line tools do not appreciate an empty subexpression. There are two ways to solve this.

The easiest way is to make the entire group optional like this:

```
# Expression (Extended): /const(ant|ellation)? (.*)/
grep -E 'const(ant|ellation)? (.*)'
```

Alternately, an empty expression may be inserted after the vertical bar.

```
# Expression (Extended): /const(ant|ellation|()) (.*)/
```

```
grep -E "const(ant|ellation|()) (.*)" poem.txt
```

Note: If you are mixing capturing with grouping, this method creates an empty capture, which ends up in the buffer following the capture buffer for this group (more on this in "Capture Operators and Variables" (page 58)).

Quoting Special Characters

As seen in previous sections, a number of characters have special meaning in regular expressions. For example, character classes are surrounded by square brackets, and the dash and caret characters have special meaning. You might ask how you would search for one of these characters. This is where quoting comes in.

In regular expressions, certain non-letter characters may have some special meaning, depending on context. To treat these characters as an ordinary character, you can prefix them with a backslash character (\). This also means that the backslash character is special in any context, so to match a literal backslash character, you must quote it with a second backslash.

There is one exception, however. To make a close bracket be a member of a character class, you do not quote it. Instead, you make it be the first character in the class.

Note: Perl rules for extended regular expressions allow you to quote a close bracket anywhere within a character class. Perl also recognizes the syntax shown here, however.

For example, to search for any string containing a backslash or a close bracket, you might use the following regular expression:

```
# Expression: /[]\\]/
grep '[]\\]' poem.txt
```

It looks a bit cryptic, but it is really relatively straightforward. The outer slashes delimit the regular expression. The brackets inside that are character class delimiters. The first close bracket immediately follows the open bracket, which makes it match an actual close bracket character instead of ending the character class. The two backslashes afterwards are, in fact, a quoted backslash, which makes this character class match the literal backslash character.

As a general rule, at least in extended regular expressions, any non-alphanumeric character can safely be quoted whether it is necessary to do so or not. If quoting it is not necessary, the extra backslash will simply be ignored. However, it is not always safe to quote letters or numbers, as these have special meanings in certain regular expression dialects, as described in "Capture Operators and Variables" (page 58) and "Perl and Python Extensions" (page 59). In addition, quoting parentheses may not do what you would expect in some dialects, as described in "Capture Operators and Variables" (page 58).

In basic regular expressions the behavior when quoting characters other than parentheses, curly braces, numbers, and characters within a character class is undefined.

Capture Operators and Variables

In "Wildcards and Repetition Operators" (page 53), this chapter described ways to create more complicated patterns to match for the search portion of a search and replace operation. This section describes more powerful operations for the replacement portion of a search and replace operation.

Capture operators and variables are used to take pieces of the original input text, capture them while searching, and then substitute those bits into the middle of the replacement text.

The easiest way to explain capture operators and variables is by example. Suppose you want to swap the words quick and lazy in the string, "The quick brown fox jumped over the lazy dog." You might write an expression like this:

```
# Expression (Basic): s/The \(.*\) brown \(.*\) the \(.*\) dog/The \3 brown \2
 the \1 dog/
# Expression (Extended): s/The (.*) brown (.*) the (.*) dog/The \3 brown \2 the
 \1 dog/
```

When you pass these expressions to `sed`, the last line of `poem.txt` should become "The lazy brown fox jumped over the quick dog."

```
sed "s/The \(.*\) brown \(.*\) the \(.*\) dog/The \3 brown \2 the \1 dog/" <
poem.txt
sed -E "s/The (.*) brown (.*) the (.*) dog/The \3 brown \2 the \1 dog/" < poem.txt
```

Portability Note: The use of the `-E` flag with `sed` to enable extended regular expressions varies from one operating system to another. For maximum portability, you should avoid using extended regular expressions with `sed`.

Note: The capture operator differs depending on which program is processing the regular expression. The tools `sed`, `awk`, and `grep` (by default) use quoted parentheses (a backslash followed by a parenthesis) to capture. Tools that use extended regular expressions by default use bare parentheses.

The content between each pair of parentheses (in this case—see note) is captured into its own buffer, numbered consecutively. Thus, in this expression, the content between "the" and "brown" is captured into a buffer. Then, the content between "brown" and "the" is captured. Finally, the content between "the" and "dog" is captured.

In the replacement string, the delimiter words ("The", "brown", "the", and "dog") are inserted, and the contents of the capture buffers are inserted in the opposite order.

Note: By default, repetition operators (except the question mark operator) are greedy. They will, by default, match the longest possible string that matches the expression as a whole. For example:

```
# s/Mary.*lamb/Joe/
sed "s/Mary.*lamb/Joe/" < poem.txt
```

In the poem, the line "Mary had a lamb looked like a lamb." will become simply "Joe."

If you want to only match up to the *first* occurrence of "lamb," you will need to use a Perl regular expression dialect extension, as described in "Non-Greedy Wildcard Matching" (page 60).

Mixing Capture and Grouping Operators

Since parentheses serve both as capture and grouping operators, use of grouping may result in unexpected consequences when capturing text in the same expression. For example, the following expression will behave very differently depending on input:

```
# Expression /const(ant)? (.*)/
```

The text you probably intended to capture is in the second buffer, not the first.

Note: In the Perl version of extended regular expressions (as described in "Non-Capturing Parentheses" (page 61)), you can use non-capturing parentheses to prevent the capture of the first portion, as show below:

```
/const(?:ant)? (.*)/
```

However, if you are using most command-line tools, this extended syntax is not supported.

Perl and Python Extensions

The regular expression dialect used in Perl, Python, and many other languages, are an extension of basic regular expressions. Some of the major differences include:

- Bare parentheses for capture—instead of using quoted parentheses, Perl uses parentheses by themselves. Quoted parentheses are treated as literals.
- Addition of shortcuts for character classes. See "Character Class Shortcuts" (page 60).
- Addition of quotation operators. In a regular expression, anything appearing between `\Q` and `\E` will be treated as literal text even if it contains characters that would ordinarily have special meaning in a regular expression. This is useful when user input, stored in a Perl variable, is used as part of a regular expression.
- Addition of "one-or-more" operator, represented by the plus (+) character.
- Support for retrieving captured values outside the scope of the extension through the variables `$1`, `$2`, and so on. (See "Capture Operators and Variables" (page 58) for information about capturing parts of a regular expression.)
- Addition of non-greedy matching. See "Non-Greedy Wildcard Matching" (page 60) for more information.

- Non-capturing parentheses. See "Non-Capturing Parentheses" (page 61) for more information.

Character Class Shortcuts

Perl regular expressions add a number of additional character class shortcuts. Some of these are listed below:

\b—word boundary (see note).
\B—non-word boundary (see note).
\d—equivalent to [:digit:].
\D—equivalent to [^:digit:].
\f—form feed.
\n—newline.
\p—character matching a Unicode character property that follows. For example, \p{L} matches a Unicode letter.
\P—character not matching a Unicode property that follows. For example, \P{L} matches any Unicode character that is not a letter.
\r—carriage return.
\s—equivalent to [:space:].
\S—equivalent to [^:space:].
\t—tab.
\v—vertical tab.
\w—equivalent to [:word:].
\W—equivalent to [^:word:].
\x—start of an ASCII character code (in hex). For example, \x20 would be a space.
\X—a single Unicode character (not supported universally).

These can be used anywhere on the left side of a regular expression, including within character classes.

Note: Word boundaries do not exist in basic regular expressions. These actually match the position between two characters rather than an actual character.

A word boundary occurs before the first character of a line (if it is a word character), at the end of the line (if it ends in a word character), and between any word character and non-word character that occur consecutively.

Non-Greedy Wildcard Matching

By default, repeat operators are greedy, matching as many times as possible before attempting to match the next part of the string. This will generally result in the longest possible string that matches the expression as a whole. In some cases, you may want the matching to stop at the shortest possible string that matches the entire expression.

To support this, Perl regular expressions (along with many other dialects) supports non-greedy wildcard matching. To convert a greedy wildcard to a non-greedy wildcard, you just add a question mark after it.

For example, consider the nursery rhyme "Mary had a little lamb, its fleece was white as snow, and everywhere that Mary went, the lamb was sure to go." Assume that you apply the following expression:

```
/Mary.*lamb/
```

That expression would match "Mary had a little lamb, its fleece was white as snow, and everywhere that Mary went, the lamb".

Suppose that instead, you want to find the shortest possible string beginning with Mary and ending with lamb. You might instead use the following expression:

```
/Mary.*?lamb/
```

That expression would match only the words "Mary had a little lamb".

Non-Capturing Parentheses

You may notice that the syntax for capture is identical to the syntax for grouping described in "Wildcards and Repetition Operators" (page 53). In most cases, this is not a problem. However, in some cases, you may wish to avoid capturing content if you are using parentheses merely as a grouping tool.

To turn off capturing for a given set of parentheses (or quoted parentheses), you should add a question mark followed by a colon after the open parenthesis.

Consider the following example:

```
# Expression (Perl and Similar ONLY): /Mary (?:had)* a little lamb\./
perl -e "while (\$line = <STDIN>) {
    \$line =~ s/Mary (?:had )*a little lamb\./Lovely day, isn't it?/;
    print \$line;
}" < poem.txt
```

This expression will match "Mary", followed by zero (0) or more instances of "had" followed by "a little lamb", followed by a literal period, and will replace the offending line with "Lovely day, isn't it?".

Note: Non-capturing parentheses are a Perl extension to regular expressions, and are not supported by most command-line tools.

How awk-ward

This chapter is a primer to help you learn how to use `awk`. The `awk` tool, along with `sed`, `grep`, and `perl`., are commonly used text processing tools based on regular expressions.

For more detailed reference material, see the manual page for `awk` and the GNU AWK manual (http://www.gnu.org/software/gawk/manual/).

This chapter uses the file `poem.txt` from "Regular Expressions Unfettered" (page 51) as the basis for most of its examples. Be sure to create that file before attempting any of these examples.

A Simple awk Script

At its most basic, an `awk` script is very similar to C. The major differences are:

- It is an interpreted language, so it is not as fast as C.
- Semicolons at the end of a statement are generally optional. (They are required only if you need to put more than one statement on a single line).
- A newline (line break) ends a statement. Much like shell scripts or C preprocessor macros, if you put a backslash at the end of one line, the statement continues onto the next line.
- Instead of having a `main` function, the main body of code is divided into a series of filter actions surrounded by curly braces. These filters are applied sequentially for *each record* in an input file. This means that the code between curly braces may execute more than once.
- Variables are all in the global scope except for parameters to functions. (Function-local variables are described more in "Functions in awk" (page 70).)
- Variables maintain their value across multiple records and files. They are set until explicitly cleared.

Unlike shell scripts (but like C), variables in `awk` are not preceded by dollar signs when you use them. This means that they cannot be inserted in the middle of strings.

There are a few special variables that are preceded by a dollar sign, however. The variable `$0` represents an entire record read from the input file. Similarly, awk divides each record up into fields, which are represented by special variables starting with `$1` and numbering upwards.

Here is a simple `awk` script:

```
{
        a=$0;
        print "This is a test: a is " a;
}
```

Save this file as `01_simple.awk`, then run it by typing:

```
awk -f 01_simple.awk poem.txt
```

This executes the awk script `01_simple.awk` and passes the file `poem.txt` as its input. For each record (a single line, by default) in the file, this will print the following:

```
This is a test: a is line from file
```

You should notice four things about this script:

- Strings separated by spaces are concatenated automatically just as they are in C.
- The `print` statement is much like the `print` statement in Perl. (The `awk` language also supports `printf`, whose syntax is like the command-line version, `printf`, except that the arguments are separated by commas instead of spaces.)
- The `awk` interpreter always requires an input file even if your script does not actually read anything from it. If you want `awk` to read from standard input, you must pass a hyphen (`-`) as the filename.
- The `awk` interpreter can take either a string of raw code or a file to execute. If you pass in a string of code as the first argument, that code is executed. If you want `awk` to execute code from a file, you must pass the `-f` flag followed by the path of the script file.

Conditional Filter Rules in awk

You don't always want to take an action based on every record in a file. Adding a pattern to a filter action is the most efficient way to limit its scope. In `awk`, the action specified by such a conditional filter occurs only if the specified pattern matches the record in question.

The format for a conditional filter rule is as follows:

```
pattern { action }
```

The *action* here is a series of statements just like any other filter rule. The *pattern* can be blank (in which case it matches every record), or it can contain any combination of regular expressions or relational expressions. These two types of expressions are briefly explained in the following sections.

Regular Expressions in awk

Conditional filter rules in `awk` may contain one or more regular expressions. These expressions *must* be a simple search-style regular expression (beginning and ending with a slash). It cannot include a command switch or modifier switches. For example, the following will not work the way you would expect:

`/mary/i`—Case-insensitive match for "mary" will actually match either the word "mary" or the letter "i", which is probably not what you want.

`s/lamb//`—Substitutions are not allowed here and will cause a syntax error.

The following `awk` script will print every line that contains “lamb”.

```
/lamb/ {
        a=$0;
        print "This is a test: a is " a;
}
```

Save this file as `02_conditional_regex.awk`, then run it by typing:

```
awk -f 02_conditional_regex.awk poem.txt
```

As with conditionals in C, you can combine multiple regular expressions with the Boolean operators `!` (not), `||` (or), and `&&` (and). For example, the following rule searches for any line that contains “Mary” but contains neither “lamb” nor “had”:

```
/Mary/ && !(/lamb/ || /had/){
        a=$0;
        print "This is a test: a is " a;
}
```

Save this file as `03_conditional_multiregex.awk`, then run it by typing:

```
awk -f 03_conditional_multiregex.awk poem.txt
```

It prints the following text:

```
This is a test: a is and everywhere that Mary went,
This is a test: a is What about Mary, Mary, and Mary?
```

For more information about regular expressions, read “Regular Expressions Unfettered” (page 51).

Relational Expressions in awk

In addition to regular expressions, `awk` supports relational expressions. You can use relational expressions to perform more fine-grained matching, such as matching based on the content of a particular field or variable.

The `awk` command supports five basic forms of relational expression:

- *expression* `~` /*regexp*/—Expression matches the regular expression.
- *expression* `!~` /*regexp*/—Expression does not match the regular expression.
- *expression comparison_operator expression*—Basic string or numeric comparison between two expressions.
- *expression* `in` *array_name*—Expression is a key in the specified array.

The *comparison_operator* can be any of the standard C comparison operators, such as `==`, `!=`, and so on.

The *expression* is generally either one of the fields or the result of an operation on one of the fields. For example, the following `awk` filter rules show, respectively, how to compare the first field to "mary" in a case-insensitive fashion, how to match all records that do not contain "Mary", and how to do an exact comparison of the first field against "Mary":

```
tolower($1) ~ /mary/ { print "CI Record: " $0; }
$0 !~ /Mary/ { print "Not Mary: " $0; }
$1 == "Mary" { print "Mary Record: " $0; }
```

Save this file as `04_conditional_insensitive.awk`, then run it by typing:

```
awk -f 04_conditional_insensitive.awk poem.txt
```

It outputs a series of lines beginning with the following:

```
CI Record: Mary had a little lamb,
Mary Record: Mary had a little lamb,
Not Mary: its fleece was white as snow,
Mary Record: Mary fleece was white as snow,
Mary Record: Mary everywhere that Mary went,
```

When `awk` sees two expressions combined with a comma (`,`), it applies the action to all records beginning with a record that matches the first pattern and continuing through a record that matches the second one.

Consider the following `awk` script:

```
/married/,/lowercase/{ print $0; }
```

Save this file as `05_conditional_range.awk`, then run it by typing:

```
awk -f 05_conditional_range.awk poem.txt
```

It prints every line beginning with the line containing "married" and ending with the line containing "lowercase".

Note: For examples using arrays, see "Working with Arrays in awk" (page 71).

Special Patterns in awk: BEGIN and END

The `awk` command supports two special patterns: `BEGIN` and `END`.

Any action associated with the `BEGIN` pattern executes before the first record is read from the file. You should, for example, make any changes to the record or field separators in a `BEGIN` action, as described in "Changing the Record and Field Separators in awk" (page 68).

Similarly, any action associated with the `END` pattern executes after the last record is read and processed. You could use this to output a special end of data record, for example.

The following example shows the use of `BEGIN` and `END` patterns.

```
BEGIN { print "Here is the line we care about."; }
/chocolate/ { print "Mmm.  Chocolate.  " $0; }
END { print "That's all that matters."; }
```

Save this file as `06_beginend.awk`, then run it by typing:

```
awk -f 06_beginend.awk poem.txt
```

It prints the following:

```
Here is the line we care about.
Mmm.  Chocolate.  I want chocolate for Valentine's day.
That's all that matters.
```

Note: The position of the `BEGIN` and `END` rules is not important. In this example, they were placed at the beginning and end for ease of readability. You can have as many `BEGIN` or `END` rules as needed. The `awk` tool executes these rules in the order in which they appear in the file.

Conditional Pattern Matching with Variables

In addition to matching against input fields, you can also specify variables in conditional pattern matches. Consider the following script:

```
BEGIN { lastwasmary = 0; }
(tolower($1) ~ /mary/ && !lastwasmary) { print "Mary appeared."; lastwasmary =
 1; }
(tolower($1) ~ /mary/ && lastwasmary) { print "Mary appeared again"; lastwasmary
 = 1; }
(tolower($1) !~ /mary/ && lastwasmary) { print "No Mary."; lastwasmary = 0; }
```

This script prints the words “Mary appeared” on the first line in which “Mary” is the first word. It prints “Mary appeared again” for each consecutive line in which “Mary” appears as the first word.

If “Mary” does not appear as the first word in a line, it prints “No Mary” and the variable `lastwasmary` is reset to zero. Thus, the next time “Mary” appears after that, it prints “Mary appeared” instead of “Mary appeared again”.

Of course, in this particular case, you may be better off conditionalizing the pattern using an `if/then` statement as described in “Control Statements in awk” (page 69).

You can also use variables to store the pattern for matching by replacing the entire pattern (including slashes) with the name of a variable. For example:

```
BEGIN { maryword = "mary"; keyword=maryword "lamb"; }
(tolower($1) ~ keyword) { print "Mary appeared."; }
(tolower($1) !~ keyword) { print "No mary."; }
```

This searches for any string in which “marylamb” appears as the first word (in a case-insensitive comparison).

You should notice that strings (and variables containing strings) separated by a space are concatenated automatically in the assignment statement. This effectively allows you to synthesize patterns containing variables.

You can also do the concatenation inline if desired. For example:

```
BEGIN { maryword = "mary"; }
(tolower($1) ~ maryword "lamb" ) { print "Mary appeared."; }
```

```
(tolower($1) !~ maryword "lamb" ) { print "No mary."; }
```

This code behaves identically to the previous example, but without the intermediate variable assignment.

Changing the Record and Field Separators in awk

By default, the record separator in `awk` is a newline, but you can change this by modifying the `awk` variable `RS`. Likewise, the default field separator is a regular expression that matches spaces and tabs.

By the time the first filter rule executes, `awk` has already read the first record and divided it into fields, using whatever record and field separators were in place at the time. Thus, if you change the record or field separator in a normal rule, that new record separator is not active until the *next* record is processed.

Thus, unless you are doing something particularly unusual, you should generally change the record separator *before* the first record is read. To do this, you use the special pattern `BEGIN`, as mentioned in "Special Patterns in awk: BEGIN and END" (page 66).

For example, the following script sets the record separator to the letter "i" and then prints each record:

```
BEGIN {RS="i"; FS=/r/}
{
        print "Record is: " $0;
        print "First field is " $1;
}
```

The first filter rule is evaluated before the first record in the file, thus setting the record separator to the letter "i" and the field separator to the letter "r". Then, after the first record is read, the second filter rule is evaluated against it based on the altered record separator.

Note: Both `RS` and `FS` can be regular expressions if desired.

The `awk` language also supports separate output separators for both records and fields. The output record and field separator variables are `ORS` and `OFS`, respectively.

The output field separator is automatically printed between fields whenever you print the value of `$0` (the "whole record" variable), and the output record separator is similarly printed at the end of `$0`.

Skipping Records and Files

At any point in your filter rules, you can skip processing of all remaining rules (effectively skipping to the next record) by using the `next` statement. For example:

```
if (i > 4) next;
```

Likewise, at any time, you can skip processing of the remainder of an input file by using the `nextfile` statement. For example:

```
if (i > 4) nextfile;
```

Control Statements in awk

Control statements in `awk` scripts are almost identical to the syntax of C control statements. As in C, the `if` statement looks like this:

```
if (expression) statement;
```

Note: The expression format is described in "Relational Expressions in awk" (page 65).

Just as in C, you can create compound statements by wrapping them in curly braces. For example, if you want to execute two statements when a given record contains the word Mary, you might write an `awk` script that looks like this:

```
{
        if ($0 ~ /Mary/) {
                print "Mary is in this line:";
                print $0;
        } else {
                print "NOMATCH: " $0;
        }
}
```

The `while` statement looks just like the `if` statement. For example:

```
{
        i=4
        if ($0 ~ /Mary/) {
                while (i) {
                        print i ":" $0;
                        i--;
                }
        }
}
```

The `for` syntax has aspects of both the C syntax and the shell script syntax. The C language form of the `for` statement is as follows:

```
for (pre_expression; while_expression; post_expression) statement
```

This statement is equivalent to the following:

```
pre_expression;
while (while_expression) {
                    statement;
                post_expression;
}
```

The first expression, which executes before entering the `while` loop, usually initializes one or more loop iterators. The second expression is then tested for truth. While it is true, the statement executes. After each iteration through the loop, the third expression executes. This usually increments or decrements the loop iterator.

As in C, you can skip the remaining code in the body of a `while` or `for` loop by calling the `continue` function.

For example, the following code prints each line that matches "Mary" three times. These are numbered 1, 2, and 4. It skips the case where `i==2`, and thus the number 3 is never printed.

```
{
        if ($0 ~ /Mary/) {
                for (i=0; i<4; i++) {
                        if (i==2) continue;
                        print i+1 ":" $0;
                }
        }
}
```

In addition, `awk` supports a shell-like (really, Perl-like) version of the `for` loop, in which it acts as an array iterator. The array iteration syntax is:

```
for (key_variable in array) statement
```

This syntax is described in more detail in "Working with Arrays in awk" (page 71).

Functions in awk

In addition to providing a number of standard functions (described in the manual page for `awk`), the `awk` language allows you to define your own custom functions. The syntax for a function declaration is:

```
function function_name(parameter1 [, parameter2, ...]) {
                    action
}
```

Because variables are in the global scope except for function parameters, if you want to define a local variable in a function, you *must* declare it as an extra parameter to the function. You do not have to pass in a value. If you do not declare the variable as a parameter, it affects execution outside of the function and its value is persistent across multiple invocations of the function.

For example, this function takes two parameters, subtracts them, and then adds one (`1`):

```
function subtractAndAddOne(a, b, c) {
        c = 1
        return (a-b+c);
}
BEGIN {
        print subtractAndAddOne(3, 2);
}
```

Important: When you call a function, you must not put a space before the opening parenthesis. In `awk`, a space is used for string concatenation, so adding a space is likely to cause a syntax error. However, it might instead result in rather strange behavior in certain contexts.

Working with Arrays in awk

The syntax for arrays in `awk` is very similar to that of arrays in C. Don't let that fool you, though. Under the hood, they behave very differently.

Arrays in `awk` are associative. This means that each array element is stored as a key-value pair, resulting in three major differences when compared to C:

- Arrays are allocated and grow dynamically as space is needed.
- Arrays can be sparse; you can have an array with a value at index `711` and a value at index `1116` with nothing between them.
- You cannot populate an array in a single operation except by splitting a string.

There are two ways to create an array. The first is by simply using it. The second is by using the `split` function. These methods are described in the sections that follow, along with useful tips about working with arrays.

Array Basics

The following code creates and prints an array called `my_array` containing the values "Partridge", "tree", "pear", and "Cassidy":

```
BEGIN {
        my_array[0] = "Partridge";
        my_array[1] = "pear";
        my_array[2] = "tree";
        my_array["David"] = "Cassidy";

        for ( my_index in my_array ) {
                print my_index "=" my_array[my_index];
        }
}
```

The first thing you will notice is that the array is not printed in order. In fact, it is printed in the order in which the underlying data is stored internally. If you want to print the values in key order, you must walk through the index numerically instead.

The second thing you will notice is that the `for` statement can be used to iterate through all of the keys in the array. In this usage, the `for` statement in `awk` is like the `for` statement in a shell script. The `for` statement array-iterator usage is:

`for (`*`key_variable`* `in` *`array_name`*`)` *`statement`*

Note: Unlike the `for` or `foreach` statements in most other languages, the array-iterator-style `for` statement in `awk` iterates through the array *keys* (indices) rather than through the array values. Thus, it is similar to the following Perl statement:

```
foreach my $key_variable (keys %assoc_array) { ... }
```

Because *key_variable* contains the key from each key-value pair rather than the value, you must explicitly use the key as an array index if you want to to obtain the values in the array. For example:

```
for ( i in arr ) {
        print arr[i];
}
```

The third thing you will notice is that, unlike C, array elements can take arbitrary strings as their key (array index). If you need to iterate through the array in key order, however, you should limit yourself to numeric keys.

As a side effect, the keys are *always* stored as a string even if they only contain numbers. Thus, if you want to compare them numerically to each other (for example, to find the smallest key for which a value exists), you must add zero (0) to the key prior to making the comparison.

For example, the following code iterates through this sparse array in key order by finding the minimum and maximum key values and then iterating from the minimum to the maximum:

```
BEGIN {
        my_array[0] = "Partridge";
        my_array[1] = "pear";
        my_array[2] = "tree";
        my_array[13] = "Cassidy";

        min = 0; max = 0;
        for ( my_index in my_array ) {
                if (my_index+0 < min) min = my_index;
                if (my_index+0 > max) max = my_index;
        }
        for (i=min; i<= max; i++) {
                if (i in my_array) {
                        print i "=" my_array[i];
                }
        }
}
```

In this example, you should note the `if` statement syntax near the end. Before printing an array value, the example checks to see if a value has ever been stored for that key value:

```
if (i in my_array) { ... }
```

Note: Generally speaking, awk assumes that you will do any array sorting externally (after awk has finished) using the sort tool or similar tools; for performance reasons, you should generally do so.

Creating Arrays with split

Assigning array elements individually can be very tedious. A more common (read "less painful") way to create an array is with the split function. The split syntax is as follows:

```
count = split( string, array_name, regexp );
```

For example, the following code splits the string "Mary lamb freezer" into words separated by spaces.

```
BEGIN {
        arr_len = split( "Mary lamb freezer", my_array, / / );
}
```

The result is that arr_len contains the number three (3). The variable my_array[1] contains "Mary", my_array[2] contains "lamb", and so on.

Copying and Joining an Array

The awk language does not support assignment of arrays. Thus, to copy an array, you must copy the individual values from one array to the next. For example, the following code initializes my_array and then copies its contents to copy_array before printing the array:

```
BEGIN {
        arr_len = split( "Mary lamb freezer", my_array, / / );
        for (word in my_array) {
                copy_array[word] = my_array[word];
        }
        for (word in copy_array) {
                print copy_array[word];
        }
}
```

Similarly, the awk language does not provide functions to join an array. To join an array, you should write a simple function like this one:

```
function join(input_array, separator) {
        string = "";
        first = 1;

        # Note: the array items are in no particular
        # order when joined with this function.
        for (i in input_array) {
                if (first) first = 0;
                else string = string separator;
                string = string input_array[i];
        }
        return string;
}
BEGIN {
        arr_len = split( "foo bar baz", my_array, / /);
```

```
        for (word in my_array) {
                print my_array[word];
        }

        print join(my_array, " ");
}
```

Like all array functions written using the array-iterator form of the `for` statement, this join does not occur in any particular order. If you need to join the array values in a particular order, you must write your own custom `join` function either using a numeric iterator or a manually specified list of fields. For example:

```
function join(input_array, separator) {
        string = "";
        first = 1;

        # Note: this preserves order, but does not
        # work with non-numeric or sparse arrays.
        for (i=1; i<=length(input_array); i++) {
                if (first) first = 0;
                else string = string separator;
                string = string input_array[i];
        }
        return string;
}
BEGIN {
        arr_len = split( "foo bar baz", my_array, / /);

        for (word in my_array) {
                print my_array[word];
        }

        print join(my_array, " ");
}
```

Deleting Array Elements

As you saw in "Array Basics" (page 71), you can add values to an array using arbitrary keys. You can also check to see if a value exists for a given key using the `if (key in array)` syntax.

If you need to delete a key-value pair, you could assign an empty value. However, the `if (key in array)` syntax still evaluates to true because there is still a value for that key (albeit an empty value). Thus, you probably want to remove the key entirely.

The `awk` programming language solves this problem with the `delete` function. The syntax for `delete` is:

```
delete array_name[key];
```

For example, the following script prints only the key-value pairs "purple = Partridge" and "majesties = tree".

```
BEGIN {
        my_array["purple"] = "Partridge";
        my_array["mountain"] = "pear";
```

```
        my_array["majesties"] = "tree";
        my_array["fruited"] = "Cassidy";

        mykey = "fruited";
        delete my_array["mountain"];
        delete my_array[mykey];

        for (i in my_array) {
                print i "=" my_array[i];
        }
}
```

If you need to clear all values from an array simultaneously, though, you don't have to delete them one at a time. Instead, you can simply do the following:

```
delete array_name;
```

This statement leaves the array specified by *array_name* empty for future use. You might do this if, for example, you want an array to be reset for each record.

File Input and Output

The `awk` programming language was primarily intended as a filter between one or more input files (or standard input) and standard output. However, it does provide some basic input and output capability.

As in shell scripts, any print statement can be written to a file using the redirection (`>`) operator (which destroys any previous contents of the file) or concatenated onto the end of an existing file using the concatenation (`>>`) operator.

Also, as in shell scripts, any print statement can be piped to an outside tool using the pipe (`|`) operator.

Pipes and redirections, however, behave differently in `awk` than in shell scripts; they remain open for future use until you explicitly close them or `awk` exits. This means, among other things, that the concatenation (`>>`) operator is only necessary if you want to retain an existing file and is not necessary to continue adding to a file that you create in `awk`.

For example, this script does the following:

- Sends two strings to `/bin/tail -n 1`. The `tail` tool prints the last line sent (which would be the second line). This shows that the two lines were sent to the same instance of `tail`.
- Closes the output to that pipe and sends another message to tail. This shows that a new instance of `tail` processed this command (because otherwise, the previous line would not have been printed).
- Writes two lines to the file `/tmp/testfile-awk`. If this file exists, it is overwritten. By using the redirect operator, the script demonstrates that additional output (after the first redirect) is appended to the file until the file is closed (regardless of whether you use the redirect or concatenation operator).

```
BEGIN {
        print "This is a test." | "/usr/bin/tail -n 1";
        print "This is only a test." | "/usr/bin/tail -n 1";
```

```
        close("/usr/bin/tail -n 1");
        print "Yikes!" | "/usr/bin/tail -n 1";

        print "This is another test" > "/tmp/testfile-awk"
        print "This is yet another test entirely" > "/tmp/testfile-awk"
}
```

Note: In `awk` (unlike in shell scripts), paths for redirects and pipes are considered strings. Thus, paths should be surrounded by double quotes so that they do not resemble regular expressions.

In a similar way, you can read input from a file using the redirection or pipe operator by combining the operator with the `getline` function. The `getline` reads a record from an outside file or pipe under programmatic control.

When you call `getline`, `awk` sets the variable `$0` to the next record from the specified file. The function returns `1` if a record was read, `0` if the end of file was reached, or `-1` if an error occurred (for example, if the file does not exist).

For example, the following program reads a record from `/tmp/testfile-awk`, and then reads a record from the output of the `echo` command:

```
BEGIN {
        getline < "/tmp/testfile-awk";
        print "The record was " $0;

        "/bin/echo 'This is a test line'" | getline
        print "The second record was " $0;
}
```

Warning: The `getline` function overwrites any value of `$0` read from the input file. Be sure you don't need it again before you call this function.

Designing Scripts for Cross-Platform Deployment

For the most part, scripts that run on other UNIX-based or UNIX-like platforms (Linux, for example) also run correctly on Mac OS X and vice versa. There are differences, however.

In addition to finding subtle variations in the file system hierarchy and the behavior of common command-line tools, you will also find different tools and technologies for device I/O and for adding and removing users and groups.

Bourne Shell Version

Mac OS X provides `bash` as its Bourne shell implementation. When executed as `/bin/sh`, it should be fully compatible with other implementations. However, at least in theory, differences could arise. The same is true of other operating systems that use BASH or `zsh` as their `sh` implementation.

For maximum compatibility, you should carefully avoid using any BASH-specific extensions in shell scripts. If you cannot avoid BASH extensions, you should explicitly make the script execute in BASH by changing the first line to the following:

```
#!/bin/bash
```

Managing Users and Groups

In the default configuration of Mac OS X, users and groups are not stored in a password file on disk. Thus, you cannot modify the password file directly.

Mac OS X supports a number of data stores for user and group information, including LDAP and flat files. Depending on the configuration, users could potentially be stored locally or remotely and accessed through any of these methods. Thus, to add users and groups through shell scripts in a general way, you *must* use the Directory Service command-line utility, `dscl` (or the Directory Service API upon which that utility is based).

Because the `dscl` tool is specific to Mac OS X, if you are writing scripts for cross-platform deployment, you should test for its existence and fall back to traditional password file modification if it is not there. To learn how to do this, read "The if Statement" (page 25).

For sample code that shows how to add a new user from the command line, read the Additional Features chapter of *Porting UNIX/Linux Applications to Mac OS X*.

To learn more about Directory Service records at a high level, read *Open Directory Programming Guide*. To learn how to use the Directory Service command line utility to alter those records, read the manual page for `dscl`.

Working with Device I/O

Mac OS X uses the I/O Kit for device drivers. Unlike most UNIX-based and UNIX-like operating systems, most devices are not exposed through device files in `/dev`. (Disks and serial ports are notable exceptions.)

In general, device I/O must be written in a C-derived language using the functionality in the I/O Kit framework. However, if you are writing your own device driver, you can expose a device file in `/dev` if desired.

Note: Devices *cannot* be accessed through `/dev/mem` in Mac OS X.

See *I/O Kit Fundamentals* for general information, *Accessing Hardware From Applications* to learn how to write an application to access device drivers from user space, or *Kernel Programming Guide* to learn how to support device files and the `ioctl` system call in the kernel.

Disk Management and Partitioning

Disk management and partitioning tools vary widely from one UNIX-based or UNIX-like OS to the next. It is impractical for this document to cover the subject in depth.

For information on other UNIX-based and UNIX-like operating systems, a good place to start is the *UNIX System Administration Handbook* by Nemeth and others.

For information about Mac OS X command-line tools for disk management and partitioning, see section 8 of *Mac OS X Man Pages*. In particular, you should look at the man pages for `hdiutil`, `pdisk`, and `diskutil`.

File System Hierarchy

A number of files are in different places in Mac OS X than in other operating systems. For more information about the Mac OS X layout, read *File System Overview*. For more information about other operating systems, read the following:

- `hier`—The Mac OS X manual page hier(7) describes the Mac OS X file-system hierarchy.

- http://www.FreeBSD.org/cgi/man.cgi?query=hier&sektion=7—The FreeBSD manual page `hier(7)` describes the FreeBSD file-system hierarchy. It is similar to the hierarchy used by most BSD-based operating systems. (No, the spelling of section is not a typo.)
- http://www.pathname.com/fhs/—The Filesystem Hierarchy Standard describes the file system hierarchy used by Linux-based operating systems, and is derived from the hierarchy used by AT&T UNIX-based operating systems.
- http://docs.hp.com/en/B2355-60130/hier.5.html—This appendix from the HP-UX documentation describes the hierarchy of AT&T UNIX-based operating systems.
- http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.baseadmn/doc/baseadmndita/fs_tree_org.htm—This page in the IBM pSeries and AIX Information Center describes the hierarchy of AIX.

General Command-Line Tool Differences

A number of command-line tools behave differently across various UNIX-based and UNIX-like operating systems. This chapter explains some of the key differences in those tools.

UNIX-based and UNIX-like operating systems generally fall into one of three camps:

- **AT&T UNIX:** Also known as UNIX System V (in its latest incarnation), AT&T UNIX was the original UNIX operating system. Its descendants include most operating systems that are commonly referred to as UNIX.
- **BSD:** Short for Berkeley Software Distribution, BSD is the name given to a family of operating systems descended from a derivative of UNIX that was originally distributed by the University of California, Berkeley, in the 1970s.

  Over the years, the Berkeley distribution and the AT&T distribution continued to diverge. The result is that there are a number of subtle syntax differences between shell scripts written for systems that follow AT&T semantics versus those that follow BSD semantics.

  In the 1990s, BSDi (a commercial company formed as a result of the UC Berkeley research) released the BSD operating system as open source. Most modern BSD operating systems are derived from this source base, known as 4.4BSD-Lite.

  Because of licensing limitations, most of the source code that was originally written by AT&T had to be rewritten from scratch under a more permissive license in order to release it as open source. This contributed further to the differences in syntax between BSD-based and AT&T UNIX-based operating systems.
- **Linux and GNU:** During the 1990s, a new operating system, Linux, was born. Combining a kernel written by Linus Torvalds and a number of utilities written by the Free Software Foundation (FSF) for their own operating system project, this operating system quickly grew into a very important third UNIX-like operating system.

  Adding to the importance of Linux and the GNU tools was the advent of MacBSD, FreeBSD, NetBSD, OpenBSD, and other BSD variants. Although BSD-based operating systems had many common utilities, they had no replacements for a few of the missing AT&T pieces. For this reason, many of these tools have also made their way into these BSD-based operating systems. In a similar way, BSD-derived tools frequently appear as part of Linux distributions.

In recent years, a number of standards have emerged to mitigate the differences in syntax between these operating systems, including POSIX and the Single Unix Specification (SUS). For this reason, many of the differences in syntax are gradually fading into irrelevance. However, for true cross-platform compatibility, you should still be aware of these differences.

Mac OS X prior to version 10.5 provided tools that generally follow BSD semantics (or, in some cases, Linux or GNU semantics). Beginning in Mac OS X v10.5, many of these tools instead generally obey AT&T semantics. Thus, some tools behave differently depending on the version of Mac OS X. These differences are described in the manual pages for the individual tools.

Note: While tools in Mac OS X v10.5 generally obey AT&T semantics, this is not always true. In particular, when executed from installer scripts or startup items, they still obey BSD semantics. You can also change which semantics are supported by setting certain environment variables as described in `compat`.

For more information on legacy-mode command support, see *Unix 03 Conformance Release Notes*.

As a convenience to script developers, you can obtain legacy behavior from most command-line tools by setting certain environment variables correctly. Read the manual page for `compat` to learn how to set these environment variables.

awk

In operating systems that follow AT&T semantics, the `awk` command supports certain forms of extended regular expressions (such as `{n,m}`, `[[==]]`, and `[[..]]`) without explicitly setting flags to enable extended regular expression support. Because this behavior is not portable, you should not depend on it.

Because of this difference, if you find a regular expression that `awk` cannot handle, you should first try enabling extended regular expression support and then see if the problem goes away. This will usually break other parts of the expression, however. If so, you must rewrite the regular expression using the extended regular expression syntax.

To learn about basic and extended regular expressions, read "Regular Expressions Unfettered" (page 51).

chown

If you pass the `-P` flag to `chown`, it does not follow symbolic links. Thus, the file that the the symbolic link points at is *never* modified if you specify the `-P` flag.

However, in operating systems that follow AT&T semantics, when you issue the command `chown -RP directory_name`, the user ID of the symbolic link itself *is* modified. In operating systems that follow BSD semantics, the symbolic link itself is *not* modified.

cp

If you pass both the `-i` and `-f` flags to `cp`, the flag that takes precedence varies among operating systems. These flags specify opposite behavior, so you should never use them together.

Also, the `-f` option has different behavior depending on the operating system:

Flags	BSD semantics	AT&T semantics
`-f` without `-p`	Destination file permissions unchanged.	Destination file permissions set to default permissions.
`-f` with `-p`	Destination file permissions set to permissions of source file.	Destination file permissions set to permissions of source file.

Finally, in operating systems that follow AT&T semantics, when copying recursively, the copy operation stops as soon as any error occurs. In operating systems that follow BSD semantics, copy operation completes to the maximum extent possible. In either case, the command exits with a nonzero result code.

If you need to ensure that a copy operation does not stop on first failure, you can use `tar` instead. For an example of how to use `tar` to copy files, see "Anonymous Subroutines" (page 37).

date

The result codes returned by `date` vary depending on the operating system. For cross-platform compatibility, you can only assume that a result code of zero (`0`) indicates success and any other value indicates some sort of failure.

echo

Of particular interest is the difference in behavior of the `echo` command. If you want to issue a prompt, in BSD-derived operating systems you can leave off the trailing newline by typing the following:

```
echo -n "Prompt: "
```

In AT&T UNIX-derived operating systems, the equivalent is:

```
echo "Prompt: \c"
```

Unfortunately, this difference makes it very difficult to write scripts that depend on this behavior in a cross-platform way. For portability, you should avoid either of these constructions. As an alternative, you can either use the `printf` command instead of `echo` or use the `tr` command to remove the newline.

For example, the following lines both print "Prompt: " followed by the word "newline" immediately afterward on the same line:

```
echo "Prompt: " | tr -d '\n'; echo "newline"
printf "Prompt: "; printf "newline\n";
```

The `echo` command also varies in the way it handles control-character escape sequences such as `\r`. Because these are handled differently in different operating systems, you should avoid using them with `echo`. As an alternative, use the `printf` command to print these sequences, or store the desired control character in a shell variable using `tr`.

For example, the following code sends an XON (Control-Q) byte to standard output:

```
XON=$(echo 'x' | tr 'x' "\\021");
echo "Here is an XON: $XON"
```

Note: The behavior of `-n`, `\c`, and other escape sequences may also vary between shell builtin versions of `echo` and the `/bin/echo` executable, depending on the operating system and the shell you are using.

file

The `file` command has two switches that behave differently in different operating systems: `-i` and `-r` (or `--raw`). For consistent behavior, you should avoid these switches.

In AT&T UNIX-based operating systems, the `-i` option tells the `file` command to not classify the contents of regular files using the external `mime.types` file. This results in faster performance but provides less detailed analysis.

In BSD-derived operating systems, the `-i` flag tells the `file` command to output raw mime type strings rather than the more traditional human readable ones. For this behavior, you should use the `--mime` flag instead, though that option is also not supported universally.

The `-r` and `--raw` options are supported only in BSD-derived operating systems. These flags tell the file command not to translate unprintable characters to their octal representations. AT&T-derived operating systems never do this.

grep

In some operating systems, `grep` fails silently if you try to match a caret in the middle of a line, while other versions of `grep` warn about the mistake. This is not a legal regular expression, of course, but if your script depends on getting an error in this case (or not getting an error), the script is not fully portable.

join

The `-e` option tells the `join` command to insert the specified string into empty fields. In operating systems that follow BSD semantics, substitution occurs *only* if there are no nonempty fields after the empty field. In operating systems that follow AT&T UNIX semantics, substitution always occurs.

Not all `join` flags are supported on all operating systems. For portability, you should limit yourself to `-a`, `-e`, `-o`, `-t`, `-v`, `-1`, and `-2`.

mkfifo

In operating systems that follow BSD semantics, the `mkfifo` command applies a mask of `0666` to the mode passed in for the `-m` option. In operating systems that follow AT&T semantics, no mask is applied.

mv

If you tell the `mv` command to move a subdirectory into its current parent directory (by typing `mv foo/bar foo`, for example), the behavior varies in a subtle way. No action occurs in any operating system because you are effectively moving a directory on top of itself. However, operating systems that follow BSD semantics exit with a zero (success) result code, whereas operating systems that follow AT&T semantics display an error message and exit with a nonzero (failure) result code.

pr

In AT&T UNIX semantics, the last space before the tab stop is replaced with a tab character. This replacement does not occur in most open source (BSD or Linux) implementations. For cross-platform consistency, you can globally replace the tab with a space by piping the output to `tr` with appropriate arguments. For example:

```
pr [arguments...] | tr '\t' ' '
```

ps

While not frequently used in shell scripts, the `ps` command behaves very differently between operating systems that follow BSD and AT&T semantics. The differences are summarized in the following table:

Flag	AT&T	BSD
`-e`	Display information about other users' processes, including those without controlling terminals; same as `-A`.	Display the environment variable settings for each process; same as `-E`.
`-g`	Display information about processes with the specified session leaders.	Unused option.
`-l`	"Long" display format; includes the `paddr` field.	"Long" display format; does not include the `paddr` field.
`-u`	Display processes belonging to a particular user. For example, `ps -u root` displays all processes belonging to the root user.	Display the fields `user`, `pid`, `%cpu`, `%mem`, `vsz`, `rss`, `tt`, `state`, `start`, `time` and `command`. Also implies the `-r` option (sort by CPU usage).

Note: For the most part, the information available from `ps` is similar in all variants (with the exception of the `-u` flag). The headings themselves, however, differ somewhat among BSD, AT&T, and Linux variants of the `ps` command. Similarly, column order is not guaranteed to be consistent across platforms. For this reason, programmatic use of ps is generally discouraged.

Most BSD and Linux variants have deprecated the use of BSD variants of flags when they are preceded by a dash. Passing these flags without a dash in these operating systems will generate the BSD behavior more consistently (at least on BSD and Linux-based operating systems). However, because this behavior is not portable, you should generally not depend on the specific quirks of a particular `ps` implementation.

sed

Most GNU versions of `sed` generate warnings for unused labels. Most other implementations do not.

When the `y` function is specified (for example, `sed y/string1/string2/`), most GNU versions convert double backslashes to single backslashes. This behavior is not portable, so you should not depend on it.

Because of this incompatibility, if you need to construct an expression containing user-entered strings that could potentially include a backslash, you should avoid the problem entirely by using the `s` function (for example, `sed s/string1/string2/`) instead of the `y` function.

sort

The form `sort +POS1 -POS2 ...` is a syntax specific to the GNU version of `sort` and is considered obsolete. This syntax is not portable.

For example:

```
$ cat data
b  a
a  b

$ sort data
a  b
b  a

$ sort +1 -2 data
sort: invalid option -- 2
Try `sort --help' for more information.
```

Instead, you should use the `-k` flag to do the same thing. For example:

```
$ sort -k 2,3 data
b  a
a  b
```

Note: The field and character positions are numbered differently with this syntax. Numbering for the `-k` syntax starts at one (`1`), while the obsolete plus and minus syntax starts at zero (`0`).

For more information on compatibility issues with the `sort` command, see the manual page for `sort`.

stty

The Mac OS X `stty` command does not support the following control modes:

- `bs0` and `bs1`
- `cr0`, `cr1`, `cr2`, and `cr3`
- `ff0` and `ff1`
- `nl0` and `nl1`
- `tab0`, `tab1`, `tab2`, and `tab3`
- `vt0` and `vt1`

In addition, Mac OS X does not support the following options:

- `ocrnl` and `-ocrnl`
- `ofdel` and `-ofdel`
- `ofill` and `-ofill`
- `onlret` and `-onlret`
- `onocr` and `-onocr`

For more information, see the manual page for `stty`.

uudecode, uuencode

In most Linux and BSD-derived operating systems, `uudecode` applies a mask of `0666` to file modes, thus preventing the creation of executable files (or files with other special modes). In operating systems that follow AT&T semantics, no mask is applied.

For consistency, if you require the results of `uudecode` to be executable or have nonstandard modes, your script should set the execute flag explicitly with `chmod`.

In operating systems that follow AT&T semantics, if `uudecode` overwrites an existing file, it cannot necessarily change its mode unless the file is owned by the current user or `uudecode` is running as the root user.

who

In operating systems that follow AT&T semantics, if you use the `-u` flag, the `who` command displays the process ID of the corresponding `login` process. In operating systems that follow BSD semantics, it does not display the process ID.

xargs

If you pass the `-L` flag to the `xargs` command, `xargs` calls the specified utility every time a certain number of lines are read. However, some details differ slightly:

- **Counting:** In operating systems that follow BSD semantics, the number of lines is based on the number of newlines encountered. Every line (including blank lines) is counted. In operating systems that follow AT&T UNIX semantics, blank lines are ignored for counting purposes.
- **Concatenation:** In operating systems that follow AT&T UNIX semantics, any line ending with a space is combined with the lines that follow it, up to and including the first nonblank line. This concatenation does not occur in operating systems that follow BSD semantics.
- **Combining Options:** In operating systems that follow BSD semantics, the `-L` and `-n` options can be used together. In operating systems that follow AT&T UNIX semantics, the `-L` and `-n` options are mutually exclusive, and the last one given on the command line will be used.

Advanced Techniques

Shell scripts can be powerful tools for writing software. Graphical interfaces notwithstanding, they are capable of performing nearly any task that could be performed with a more traditional language. This chapter describes several techniques that will help you write more complex software using shell scripts.

Data Structures, Arrays, and Indirection

One of the more under-appreciated commands in shell scripting is the `eval` builtin. The `eval` command takes a series of arguments, concatenates them into a single command, then executes it.

For example, the following script assigns the value `3` to the variable `X` and then prints the value:

```
#!/bin/sh
eval X=3
echo $X
```

For such simple examples, the `eval` command is superfluous. However, the behavior of the `eval` command becomes much more interesting when you need to construct or choose variable names programmatically. For example, the next script also assigns the value `3` to the variable `X`:

```
#!/bin/sh

VARIABLE="X"
eval $VARIABLE=3
echo $X
```

When the `eval` command evaluates its arguments, it does so in two steps. In the first step, variables are replaced by their values. In the preceding example, the letter `X` is inserted in place of `$VARIABLE`. Thus, the result of the first step is the following string:

```
X=3
```

In the second step, the `eval` command executes the statement generated by the first step, thus assigning the value `3` to the variable `X`. As further proof, the `echo` statement at the end of the script prints the value `3`.

The `eval` command can be particularly convenient as a substitute for arrays in shell script programming. It can also be used to provide a level of indirection, much like pointers in C. Some examples of the `eval` command are included in the sections that follow.

A Complex Example: Printing Values

The next example takes user input, constructs a variable based on the value entered using `eval`, then prints the value stored in the resulting variable.

```
#!/bin/sh
echo "Enter variable name and value separated by a space"
read VARIABLE VALUE
echo Assigning the value $VALUE to variable $VARIABLE
eval $VARIABLE=$VALUE

# print the value
eval echo "$"$VARIABLE

# export the value
eval export $VARIABLE

# print the exported variables.
export
```

Warning: This script executes arbitrary user input. It is intended *only* as an example of the usage of the `eval` statement. In real-world code, you should *never* pass unsanitized user input directly to `eval` because doing so would provide a vector for arbitrary code execution.

Run this script and type something like `MYVAR 33`. The script assigns the value `33` to the variable `MYVAR` (or whatever variable name you entered).

You should notice that the echo command has an additional dollar sign (`$`) in quotes. The first time the `eval` command parses the string, the quoted dollar sign is simplified to merely a dollar sign. You could also surround this dollar sign with single quotes or quote it with a backslash, as described in "Quoting Special Characters" (page 33). The result is the same.

Thus, the statement:

```
eval echo "$"$VARIABLE
```

evaluates to:

```
echo $MYVAR
```

Note: If you forgot to quote the first dollar sign, you would get a very strange result. The variable `$$` is a special shell variable that contains the process ID of the current shell. Thus, without quoting the first dollar sign, the two dollar signs would be interpreted as a variable, and thus the statement would evaluate to something like:

```
echo 1492MYVAR
```

This is probably not what you want.

A Practical Example: Using eval to Simulate an Array

In "Shell Variables and Printing" (page 14), you learned how to read variables from standard input. This was limited to some degree by the inability to read an unknown number of user-entered values.

The script below solves this problem using `eval` by creating a series of variables to hold the values of a simulated array.

```
#!/bin/sh

COUNTER=0
VALUE="-1"
echo "Enter a series of lines of test.  Enter a blank line to end."

while [ "x$VALUE" != "x" ] ; do
        read VALUE
        eval ARRAY_$COUNTER=$VALUE
        eval export ARRAY_$COUNTER
        COUNTER=$(expr $COUNTER '+' 1) # More on this in Paint by Numbers
done
COUNTER=$(expr $COUNTER '-' 1) # Subtract one for the blank value at the end.

# print the exported variables.
COUNTERB=0;

echo "Printing values."
while [ $COUNTERB -lt $COUNTER ] ; do
        echo "ARRAY[$COUNTERB] = $(eval echo "$"ARRAY_$COUNTERB)"
        COUNTERB=$(expr $COUNTERB '+' 1) # More on this in Paint by Numbers
done
```

This same technique can be used for splitting an unknown number of input values in a single line as shown in the next listing:

```
#!/bin/sh

COUNTER=0
VALUE="-1"
echo "Enter a series of lines of numbers separated by spaces."

read LIST
IFS=" "
for VALUE in $LIST ; do
        eval ARRAY_$COUNTER=$VALUE
        eval export ARRAY_$COUNTER
        COUNTER=$(expr $COUNTER '+' 1) # More on this in Paint by Numbers
```

```
done

# print the exported variables.
COUNTERB=0;

echo "Printing values."
while [ $COUNTERB -lt $COUNTER ] ; do
        echo "ARRAY[$COUNTERB] = $(eval echo '$'ARRAY_$COUNTERB)"
        COUNTERB=$(expr $COUNTERB '+' 1) # More on this in Paint by Numbers
done
```

A Data Structure Example: Linked Lists

In a complex shell script, you may need to keep track of multiple pieces of data and treat them like a data structure. The `eval` command makes this easy. Your code needs to pass around only a single name from which you build other variable names to represent fields in the structure.

Similarly, you can use the `eval` statement to provide a level of indirection similar to pointers in C.

For example, the following script manually constructs a linked list with three items, then walks the list:

```
#!/bin/sh

VAR1_VALUE="7"
VAR1_NEXT="VAR2"

VAR2_VALUE="11"
VAR2_NEXT="VAR3"

VAR3_VALUE="42"

HEAD="VAR1"
POS=$HEAD
while [ "x$POS" != "x" ] ; do
        echo "POS: $POS"
        VALUE="$(eval echo '$'$POS'_VALUE')"
        echo "VALUE: $VALUE"
        POS="$(eval echo '$'$POS'_NEXT')"
done
```

Using this technique, you could conceivably construct any data structure that you need (with the caveat that manipulating large data structures in shell scripts is generally not conducive to good performance).

Nonblocking I/O

No discussion of programming would be complete without a way to approximate asynchronous timer events and asynchronous input and output.

First, a warning. Nonblocking I/O is not possible in a pure shell script. It requires the use of an external tool that sets the terminal to nonblocking. Setting the terminal to nonblocking can seriously confuse the shell, so you should not mix nonblocking I/O and blocking I/O in the same program.

With that caveat, you can perform nonblocking I/O by writing a small C helper such as this one:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int ch;
    int flags = fcntl(STDIN_FILENO, F_GETFL);
    if (flags == -1) return -1; // error

    fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK);

    ch = fgetc(stdin);
    if (ch == EOF) return -1;
    if (ch == -1) return -1;
    printf("%c", ch);
    return 0;
}
```

If you compile this tool and name it `getch`, you can then use it to perform nonblocking terminal input, as shown in the following example:

```
#!/bin/bash

stty -icanon -isig
while true ; do
        echo -n "Enter a character: "
        CHAR=`./getch`
        if [ "x$CHAR" = "x" ] ; then
                echo "NO DATA";
        else
                if [ "x$CHAR" = "xq" ] ; then
                        stty -cbreak
                        exit
                fi
                echo "DATA: $CHAR";
        fi
        sleep 1;
done

# never reached
stty -cbreak
```

This script prints “NO DATA” or “DATA: *[some character]*” depending on whether you have pressed a key in the past second. (To stop the script, press the Q key.) Using the same technique, you can write fairly complex shell scripts that can detect keystrokes while performing other tasks.

The `stty` command changes two settings on the controlling terminal (a device file that represents the current Terminal window, console, `ssh` session, or other communication channel).

The `-icanon` flag disables canonicalization of input. For example, if you press (in order) the keys A, Delete, and Return, normally your shell script would receive an empty line. With canonicalization disabled, your application will instead see three bytes: the letter A, a control character representing the Delete key, and a newline character representing the Return key.

The `-isig` flag disables automatic generation of signals based on input character. By specifying this flag, you can trap arbitrary control characters, including characters that would otherwise halt, pause, or resume execution (Control-C, for example). Because disabling these signals makes it harder to stop execution of a shell script, you should generally avoid using this flag unless you intend to capture these control characters as part of normal operation. If you merely need to execute cleanup code when these keys are pressed, you should trap the resulting signals instead, as described in "Trapping Signals" (page 94).

Depending on what you are doing, you may also find it useful to pass the `-echo` flag. This flag disables the automatic echo of typed characters to the screen.

Depending on what other flags you pass, you may want to reset the terminal at the end by issuing the command `stty sane`.

Timing Loops

On rare occasions, you may find the need to perform some operation on a periodic basis with greater than the one second precision offered by `sleep`. Although the shell does not offer any precision timers, you can closely approximate such behavior through the use of a calibrated delay loop.

The basic design for such a loop consists of two parts: a calibration routine and a delay loop. The calibration routine should execute approximately the same instructions as the delay loop for a known number of iterations.

The nature of the instructions within the delay loop are largely unimportant. They can be any instructions that your program needs to execute while waiting for the desired amount of time to elapse. However, a common technique is to perform nonblocking I/O during the delay loop and then process any characters received.

For example, Listing 7-1 shows a very simple timing loop that reads a byte and triggers some simple echo statements (depending on what key is pressed) while simultaneously echoing a statement to the screen about once per second.

Listing 7-1 A simple one-second timing loop

```
#!/bin/sh

ONE_SECOND=1000

function read_test()
{
    COUNT=0
    local ONE_SECOND=1000                       # ensure this never trips!
    while [ $COUNT -lt 200 ] ; do
        CHAR=`./getch`
        if [ $1 = "rot" ] ; then
                CHAR=","
        fi
```

```
        case "$CHAR" in
                ( "q" | "Q" )
                        CONT=0;
                        GAMEOVER=1
                ;;
                ( "" )
                        # Silently ignore empty input.
                ;;
                ( * )
                        echo "Unknown key $CHAR"
                ;;
        esac
        COUNT=`expr $COUNT '+' 1`
        while [ $COUNT -ge $ONE_SECOND ] ; do
                COUNT=`expr $COUNT - $ONE_SECOND`
                MODE="clear";
                draw_cur $ROT;
                VPOS=`expr $VPOS '+' 1`
                MODE="apple";
                draw_cur $ROT
        done
    done
}

function calibrate_timers()
{
    2>/tmp/readtesttime time $0 -readtest
    local READ_DUR=`grep real /tmp/readtesttime | sed 's/real.*//' | tr -d ' '`
    # echo "READ_DUR: $READ_DUR"

    local READ_SINGLE=`echo "scale=20; ($READ_DUR / 200)" | bc`
    ONE_SECOND=`echo "scale=0; 1.0  / $READ_SINGLE" | bc`

    # echo "READ_SINGLE: $READ_SINGLE";
    # exit

    echo "One second is about $ONE_SECOND cycles."
}

if [ "x$1" = "x-readtest" ] ; then
        read_test
        exit
fi

echo "Calibrating.  Please wait."
calibrate_timers

echo "Done calibrating.  You should see a message about once per second.  Press 'q' to
 quit."
stty -icanon -isig

GAMEOVER=0
COUNT=0
# Start the game loop.
while [ $GAMEOVER -eq 0 ] ; do
        # echo -n "Enter a character: "
        CHAR=`./getch`
        case "$CHAR" in
```

```
                ( "q" | "Q" )
                        CONT=0;
                        GAMEOVER=1
                ;;
                ( "" )
                        # Silently ignore empty input.
                ;;
                ( * )
                        echo "Unknown key $CHAR"
                ;;
        esac
        COUNT=`expr $COUNT '+' 1`
        while [ $COUNT -ge $ONE_SECOND ] ; do
                COUNT=`expr $COUNT - $ONE_SECOND`
                echo "One second elapsed (give or take)."
        done
done

stty sane
```

In a real-world timing loop, you will probably have keys that perform certain operations that take time—moving a piece on a checkerboard, for example. In that case, your calibration should also perform a series to tests to approximate the amount of time for each of those operations.

If you divide the time for the slow operation by the duration of a single read operation (`READ_SINGLE`), you can discern an approximate penalty for the move using iterations of the main program loop as the unit value. Then, when you perform one of those operations later, you simply add that penalty value to the main loop counter, thus ensuring that the "One second elapsed" messages will quickly catch up with (approximately) where they should be.

You can approximate this further by using larger numbers in your loop counter to achieve greater precision. For example, you might increment your loop counter by 100 instead of by 1. This will give a much more accurate approximation of the number of cycles stolen by a slow operation.

Warning: If you perform significant multiplication (for example, to increase game play speed on subsequent levels) to change the rate of your timer, using larger values means that you are much more likely to exceed the maximum value that shell math or `expr` math can handle during your interim calculations. In such cases, you may find it better to use `bc`, which works with floating-point quantities.

Trapping Signals

No discussion of advanced programming would be complete without an explanation of signal handling. In UNIX-based and UNIX-like operating systems, signals provide a primitive means of interprocess communication. A script or other process can send a signal to another process by either using the `kill` command or by calling the `kill` function in a C program. Upon receipt, the receiving process either exits, ignores the signal, or executes a signal handler routine of the author's choosing.

Signals are most frequently used to terminate execution of a process in a friendly way, allowing that process the opportunity to clean up before it exits. However, they can also be used for other purposes. For example, when a terminal window changes in size, any running shell in that window receives a

SIGWINCH (window change) signal. Normally, this signal is ignored, but if a program cares about window size changes, it can trap that signal and handle it in an application-specific way. With the exception of the SIGKILL signal, any signal can be trapped and handled by calling the function signal.

In much the same way, shell scripts can also trap signals and perform operations when they occur, through the use of the trap builtin.

The syntax of trap is as follows:

```
trap function signal [ signal ... ]
```

The first argument is the name of a function that should be called when the specified signals are received. The remaining arguments contain a space-delimited list of signal names or numbers. Because signal numbers vary between platforms, for maximum readability and portability, you should always use signal names.

For example, if you want to trap the SIGWINCH (window change) signal, you could write the following statement:

```
trap sigwinch_handler SIGWINCH
```

After you issue this statement, the shell calls the function sigwinch_handler whenever it receives a SIGWINCH signal. The script in Listing 7-2 prints the phrase "Window size changed." whenever you adjust the size of your terminal window.

Listing 7-2 Installing a signal handler trap

```
#!/bin/sh

function fixrows()
{
        echo "Window size changed."
}

echo "Adjust the size of your window now."
trap fixrows SIGWINCH

COUNT=0
while [ $COUNT -lt 60 ] ; do
        COUNT=$(($COUNT + 1))
        sleep 1
done
```

Sometimes, instead of trapping a signal, you may want to ignore a signal entirely. To do this, specify an empty string for the function name. For example, the code in Listing 7-3 ignores the "interrupt" signal generated when you press Control-C:

Listing 7-3 Ignoring a signal

```
#!/bin/sh
trap "" SIGINT

echo "This program will sleep for 10 seconds and cannot be killed with"
echo "control-c."
sleep 10
```

Finally, signals can be used as a primitive form of interscript communication. The next two scripts work as a pair. To see this in action, first save the script in Listing 7-4 as `ipc1.sh` and the script in Listing 7-5 as `ipc2.sh`.

Listing 7-4 `ipc1.sh`: Script interprocess communication example, part 1 of 2

```
#!/bin/sh

## Save this as ipc1.sh

./ipc2.sh &

PID=$!

sleep 1 # Give it time to launch.

kill -HUP $PID
```

Listing 7-5 `ipc2.sh`: Script interprocess communication example, part 2 of 2

```
#!/bin/sh

## Save this as ipc2.sh

function hup_handler()
{
        echo "SIGHUP RECEIVED."
        exit 0
}

trap hup_handler SIGHUP

while true ; do
        sleep 1
done
```

Now run `ipc1.sh`. It launches the script `ipc2.sh` in the background, uses the special shell variable `$!` to get the process ID of the last background process (`ipc2.sh` in this case), then sends it a hangup (`SIGHUP`) signal using `kill`.

Because the second script, `ipc2.sh`, trapped the hangup signal, its shell then calls a handler function, `hup_handler`. This function prints the words “SIGHUP RECEIVED.“ and exits.

Special Shell Variables

The Bourne shell has a number of special “automatic” variables that it maintains for informational purposes. These variables provide information such as the process ID of the shell, the exit status of the last command, and so on. This section provides a list of these special variables. For additional variables supported by specific Bourne shell variants such as `bash` and `zsh`, see the `bash` and `zshparam` manual pages, respectively.

Table 7-1 Special shell variables

Variable	Description
Process information	
`$$`	Process ID of shell
`$PPID`	Process ID of shell's parent process. **Quirk Warning:** for subshells, the value of PPID is inherited from the parent shell. Thus, PPID is only the parent of the outermost shell process.
`$?`	Exit status of last command.
`$_`	Name of last command.
`$!`	Process ID of last process run in the background using ampersand (&) operator.
`$PATH`	A colon-delimited list of locations where trusted executables are installed. Any executable in one of these locations can be executed without specifying a complete path.
Field and record parsing	
`$IFS`	Input Field Separators (uses are explained in "Variable Expansion and Field Separators" (page 31))
User information	
`$HOME`	The user's home directory.
Miscellaneous variables	
`$#`	Number of arguments passed to the shell.
`$@`	Complete list of arguments passed to the shell, separated by spaces..
`$*`	Complete list of arguments passed to the shell, separated by the first character of the `IFS` (input field separators) variable. **Portability warning:** In AIX, if you surround this variable with quotes, each argument will be enclosed in quotes.
`$-`	A list of all shell flags currently enabled.

Shell Text Formatting

One powerful technique when writing shell scripts is to take advantage of the terminal emulation features of your terminal application (whether it is Terminal, an xterm, or some other application) to display formatted content.

You can use the `printf` command to easily create columnar layouts without any special tricks. For more visually exciting presentation, you can add color or text formatting such as boldface or underlined display using ANSI (VT100/VT220) escape sequences.

In addition, you can use ANSI escape sequences to show or hide the cursor, set the cursor position anywhere on the screen, and set various text attributes, including boldface, inverse, underline, and foreground and background color.

Using the printf Command for Tabular Layout

Much like C and other languages, most operating systems that support shell scripts also provide a command-line version of `printf`. This command differs from the C `printf` function in a number of ways. These differences include the following:

- The `%c` directive does not perform integer-to-character conversion. The only way to convert an integer to a character with the shell version is to first convert the integer into octal and then print it by using the octal value as a switch. For example, `printf "\144"` prints the lowercase letter d.
- The command-line version supports a much smaller set of placeholders. For example, `%p` (pointers) does not exist in the shell version.
- The command-line version does not have a notion of long or double-precision numbers. Although flags with these modifiers are allowed (`%lld`, for example), the modifiers are ignored. Thus, there is no difference between `%d`, `%ld`, and `%lld`.
- Large integers may be truncated to 32-bit signed values.
- Double-precision floating-point values may be reduced to single-precision values.
- Floating point precision is not guaranteed (even for single-precision values) because some imprecision is inherent in the conversion between strings and floating-point numbers.

Much like the `printf` statement in other languages, the shell script `printf` syntax is as follows:

```
printf "format string" argument ...
```

Like the C `printf` function, the command-line `printf` format string contains some combination of text, switches (`\n` and `\t`, for example), and placeholders (`%d`, for example).

The most important feature of `printf` for tabular layouts is the padding feature. Between the percent sign and the type letter, you can place a number to indicate the width to which the field should be padded. For a floating-point placeholder (`%f`), you can optionally specify `two` numbers separated by a decimal point. The leftmost value indicates the total field width, while the rightmost value indicates the number of decimal places that should be included. For example, you can print pi to three digits of precision in an 8-character-wide field by typing `printf "%8.3f" 3.14159265`.

In addition to the width of the padding, you can add certain prefixes before the field width to indicate special padding requirements. They are:

- Minus sign (`-`)—indicates the field should be left justified. (Fields are right justified by default.)
- Plus sign (`+`)—indicates that a sign should be prepended to a numerical argument even if it has a positive value.

- Space—indicates that a space should be added to a numerical argument in place of the sign if the value is positive. (A plus sign takes precedence over a space.)
- Zero (0)—indicates that numerical arguments should be padded with leading zeroes instead of spaces. (A minus sign takes precedence over a zero.)

For example, if you want to create a four-column table of name, address, phone number, and GPA, you might write a statement like this:

Listing 7-6 Columnar printing with printf

```
#/bin/sh

NAME="John Doe"
ADDRESS="1 Fictitious Rd, Bucksnort, TN"
PHONE="(555) 555-5555"
GPA="3.885"
printf "%20s | %30s | %14s | %5s\n" "Name" "Address" "Phone Number" "GPA"
printf "%20s | %30s | %14s | %5.2f\n" "$NAME" "$ADDRESS" "$PHONE" "$GPA"
```

The `printf` statement pads the fields into neat columns and truncates the GPA to two decimal places, leaving room for three additional characters (the decimal point itself, the ones place, and a leading space). You should notice that the additional arguments are all surrounded by quotation marks. If you do not do this, you will get incorrect behavior because of the spaces in the arguments.

Note: The `printf` command, like its C function sibling, does not truncate values to fit within the specified field width. For examples of how to truncate strings, see "Truncating Strings" (page 100).

The next sample shows number formatting:

```
#!/bin/sh

GPA="3.885"

printf "%f | whatever\n" "$GPA"
printf "%20f | whatever\n" "$GPA"
printf "%+20f | whatever\n" "$GPA"
printf "%+020f | whatever\n" "$GPA"
printf "%-20f | whatever\n" "$GPA"
printf "%- 20f | whatever\n" "$GPA"
```

This prints the following output:

```
3.885000 | whatever
            3.885000 | whatever
           +3.885000 | whatever
+000000000003.885000 | whatever
3.885000             | whatever
 3.885000            | whatever
```

Most of the same formatting options apply to `%s` and `%d` (including, surprisingly, zero-padding of string arguments). For more information, see the manual page for `printf`.

Truncating Strings

To truncate a value to a given width, you can use a simple regular expression to keep only the first few characters. For example, the following snippet copies the first seven characters of a string:

```
STRING="whatever you want it to be"
TRUNCSTRING="`echo "$STRING" | sed 's/^\(.......\).*$/\1/'`"
echo "$TRUNCSTRING"
```

As an alternative, you can use a more general-purpose routine such as the one in Listing 7-7, which truncates a string to an arbitrary length by building up a regular expression.

Listing 7-7 Truncating text to column width

```
function trunc_field()
{
    local STR=$1
    local CHARS=$2
    local EXP=""
    local COUNT=0
    while [ $COUNT -lt $CHARS ] ; do
        EXP="$EXP."
        COUNT=`expr $COUNT + 1`
    done
    echo $STR | sed "s/^\($EXP\).*$/\1/"
}

printf "%10s | something\n" "`trunc_field "$TEXT" 20`"
```

Of course, you can do this much faster by either caching these strings or replacing most of the function with a single line of Perl:

```
echo "$STR" | perl -e "$/=undef; print substr(<STDIN>, 0, $CHARS);"
```

Finally, if you are willing to write code that is *extremely* nonportable (using a syntax that does not even work in `zsh`), you can use Bash-specific substring expansion:

```
echo "${STR:0:8}"
```

You can learn about similar operations in the manual page for `bash` under the "Parameter Expansion" heading. As a general rule, however, you should avoid such shell-specific tricks.

Using ANSI Escape Sequences

You can use ANSI escape sequences to add color or formatting to text displayed in the terminal, reposition the cursor, set tab stops, clear portions of the display, change scrolling behavior, and more. This section includes a partial list of many commonly used escape sequences, along with examples of how to use them.

Important: For the purposes of this section, the Esc (escape) key is represented by the notation `^[` because the ASCII character for the Esc key is the same as the ASCII character for Control-bracket (character 27). Thus, when you see `^[[`, it means Esc followed by a bracket. (Nearly all ANSI escape sequences begin with Esc-bracket, though there are a few exceptions.)

There are two ways to generate escape sequences: direct printing and using the terminfo database. Printing the sequences directly has significant performance advantages but is less portable because it assumes that all terminals are ANSI/VT100/VT220-compliant. A good compromise is to combine these two approaches by caching the values generated with a terminfo command such as `tput` at the beginning of your script and then printing the values directly elsewhere in the script.

Generating Escape Sequences using the terminfo Database

Generating escape sequences with the terminfo database is relatively straightforward once you know what terminal capabilities to request. You can find several tables containing capability information, along with the standard ANSI/VT220 values for each capability, in "ANSI Escape Sequence Tables" (page 103). (Note that not all ANSI escape sequences have equivalent terminfo capabilities, and vice versa.)

Once you know what capability to request (along with any additional arguments that you must specify), you can use the `tput` command to output the escape sequence (or capture the output of `tput` into a variable so you can use it later). For example, you can clear the screen with the following command:

```
tput cl
```

Some terminfo database entries contain placeholders for numeric values, such as row and column information. The easiest way to use these is to specify those numeric values on the command line when calling `tput`. However, for performance, it may be faster to substitute the values yourself. For example, the capability `cup` sets the cursor position to a row and column value. The following command sets the position to row 3, column 7:

```
tput cup 3 7
```

You can, however, obtain the unsubstituted string by requesting the capability without specifying row and column parameters. For example:

```
tput cup | less
```

By piping the data to `less`, you can see precisely what the `tput` tool is providing, and you can look up the parameters in the manual page for `terminfo`. This particular example prints the following string:

```
^[[%i%p1%d;%p2%dH
```

The `%i` notation means that the first two (and only the first two) values are one greater than you might otherwise expect. (For ANSI terminals, columns and rows number from 1 rather than from 0). The `%p1%d` means to push parameter 1 onto the stack and then print it immediately. The parameter `%p2%d` is the equivalent for parameter 2.

As you can see from even this relatively simple example, the language used for terminfo is quite complex. Thus, while it may be acceptable to perform the substitution for simple terminals such as VT100 yourself, you may still be trading performance for portability. In general, it is best to let `tput` perform the substitutions on your behalf.

Generating Escape Sequences Directly

To use an ANSI escape sequence without using `tput`, you must first be able to print an escape character from your script. There are three ways to do this:

- **Use `printf` to print the escape sequence.** In a string, the `\e` switch prints an escape character. This is the easiest way to print escape sequences.

  For example, the following snippet shows how to print the reset sequence (`^[c`):

  ```
  printf "\ec" # resets the screen
  ```

  > **Note:** In all versions of Mac OS X, `printf` is a shell builtin for `/bin/sh`. However, this is not necessarily true for other platforms. Thus, if cross-platform performance is an issue, you should avoid this usage.

- **Embed the escape character in your script.** The method of doing this varies widely from one editor to another. In most text-based editors and on the command line itself, you do this by pressing Control-V followed by the Esc key. Although this is the fastest way to print an escape sequence, it has the disadvantage of making your script harder to edit.

  For example, you might write a snippet like this one:

  ```
  echo "^[c" # Read the note below!!!
  ```

  > **Note:** You *must* enter this escape character manually; copying and pasting the text in this example will *not* work.
  >
  > To enter the above escape sequence, type `echo` followed by a space and double-quote mark. Then press Control-V followed by the Esc key to add the escape character. Next, type a lowercase `c`. Finally, close the double-quote mark and press Return.

- **Use `printf` to store an escape character into a variable.** This is the recommended technique because it is nearly as fast as embedding the escape character but does not make the code hard to read and edit.

  For example, the following code sends a terminal reset command (`^[c`):

  ```
  #!/bin/sh

  ESC=`printf "\e"`       # store an escape character
                          # into the variable ESC
  echo "$ESC""c"          # Echo a terminal reset command.
  ```

Because the terminal reset command is one of only a handful of escape sequences that do not start with a left square bracket, it is worth pointing out the two sets of double-quote marks after the variable in the above example. Without those, the shell would try to print the value of the variable `ESCc`, which does not exist.

ANSI Escape Sequence Tables

There are four basic categories of escape codes:

- Cursor manipulation routines (described in Table 7-2 (page 104)) allow you to move the cursor around on the screen, show or hide the cursor, and limit scrolling to only a portion of the screen.
- Attribute manipulation sequences (described in "Attribute and Color Escape Sequences" (page 105)) allow you to set or clear text attributes such as underlining, boldface display, and inverse display.
- Color manipulation sequences (described in "Attribute and Color Escape Sequences" (page 105)) allow you to change the foreground and background color of text.
- Other escape codes (described in Table 7-5 (page 108)) support clearing the screen, clearing portions of the screen, resetting the terminal, and setting tab stops.

Cursor and Scrolling Manipulation Escape Sequences

The terminal window is divided into a series of rows and columns. The upper-left corner is row 1, column 1. The lower-right corner varies depending on the size of the terminal window.

You can obtain the current number of rows and columns on the screen by examining the values of the shell variables `LINES` and `COLUMNS`. Thus, the screen coordinates range from `(1, 1)` to `($LINES, $COLUMNS)`. In most modern Bourne shells, the values for `LINES` and `COLUMNS` are automatically updated when the window size changes. This is true for both `bash` and `zsh`.

However, in `bash`, these variables are set only for interactive instances of the shell. This presents a small problem for shell scripts that care about window size. As a result, in versions of Mac OS X where the default shell is `bash` (Mac OS X v10.3 and newer), these variables are not defined in shell scripts that start with `#!/bin/sh`.

Of course, you could request `zsh` as the interpreter by changing the first line of your script to `#!/bin/zsh`, but this is not particularly portable. Fortunately, without changing shells, you can easily obtain the current row and column count with the code in Listing 7-8.

Listing 7-8 Obtaining terminal size using stty or tput

```
# If tput is available, this is the easy way:
MYLINES=`tput lines` # ROWS
MYCOLUMNS=`tput cols` # COLUMNS

# If not, you can do it the hard way.  This usually works.
MYLINES=`stty -a | grep rows | sed 's/^.*;\(.*\)rows\(.*\);.*$/\1\2/' | sed 's/;.*$//'
 | sed 's/[^0-9]//g'` # ROWS
MYCOLUMNS=`stty -a | grep columns | sed 's/^.*;\(.*\)columns\(.*\);.*$/\1\2/' | sed
's/;.*$//' | sed 's/[^0-9]//g'` # COLUMNS
```

If you want to be particularly clever, you can also trap the `SIGWINCH` signal and update your script's notion of lines and columns when it occurs. See "Trapping Signals" (page 94) for more information.

Once you know the number of rows and columns on your screen, you can move the cursor around with the escape sequences listed in Table 7-2. For example, to set the cursor position to row 4, column 5, you could issue the following command:

```
printf "\e[4;5H"
```

For other, faster ways to print escape sequences, see "Generating Escape Sequences Directly" (page 102).

Table 7-2 Cursor and scrolling manipulation escape sequences

Terminfo capability	Escape sequence	Description
`tivis` **Note:** The terminfo entry for Terminal does not support this option.	`^[[?25l`	Hides the cursor.
`tvvis` **Note:** The terminfo entry for Terminal does not support this option.	`^[[?25h`	Shows the cursor.
`cup` *row colum*	`^[[`*r*`;`*c*`H`	Sets cursor position to row *r*, column *c*.
(no equivalent)	`^[[6n`	Reports current cursor position as though typed from the keyboard (reported as `^[[`*r*`;`*c*`R`). Note: it is not practical to capture this information in a shell script.
`sc`	`^[7`	Saves current cursor position and style.
`rc`	`^[8`	Restores previously saved cursor position and style.
`cuu` *r*	`^[[`*r*`A`	Moves cursor up *r* rows.
`cud` *r*	`^[[`*r*`B`	Moves cursor down *r* rows.
`cuf` *c*	`^[[`*c*`C`	Moves cursor right *c* columns.
`cub` *c*	`^[[`*c*`D`	Moves cursor left *c* columns.
(no equivalent)	`^[[7h`	Disables automatic line wrapping when the cursor reaches the right edge of the screen.
(no equivalent)	`^[[7l`	Enables line wrapping (on by default).
(no equivalent)	`^[[r`	Enables whole-screen scrolling (on by default).
(no equivalent)	`^[[`*S*`;`*E*`r`	Enables partial-screen scrolling from row *S* to row *E* and moves the cursor to the top of this region.
`do`	`^[D`	Moves the cursor down by one line.
`up`	`^[M`	Moves the cursor up by one line.

Attribute and Color Escape Sequences

Attribute and color escape sequences allow you to change the attributes or color for text that you have not yet drawn. No escape sequence (scrolling notwithstanding) changes anything that has already been drawn on the screen. Escape sequences apply only to subsequent text.

For example, to draw a red "W" character, first send the escape sequence to set the foreground color to red (`^[[31m`), then print a "W" character, then send an attribute reset sequence (`^[[m`), if desired.

The attribute and color escape codes can be combined with other attribute and color escape codes in the form `^[[#;#;#;...#m`. For example, you can combine the escape sequences `^[[1m` (bold) and `^[[32m` green text) into the sequence `^[[1;32m`. Listing 7-9 prints a familiar phrase in multiple colors.

Listing 7-9 Using ANSI color

```
#!/bin/sh

printf '\e[41mH\e[42me\e[43ml\e[44;32ml\e[45mo\e[m \e[46;33m'
printf 'W\e[47;30mo\e[40;37mr\e[49;39ml\e[41md\e[42m!\e[m\n'
```

Note: For consistent formatting, you may add a leading zero to any single-digit attribute escape sequences, if desired. For example, `^[[1m` is equivalent to `^[[01m`.

Table 7-3 contains a list of capabilities and escape sequences that control text style.

Table 7-3 Attribute escape sequences

Terminfo capability	Escape sequence	Description
Resetting attributes		
`me`	`^[[m` or `^[[0m`	Resets all attributes to their default values.
Setting attributes		
`bold`	`^[[1m`	Enables "bright" display. This is basically boldface text. This code and code #2 (`dim`) are mutually exclusive.
`dim`	`^[[2m`	Enables "dim" display. This code and code #1 (`bold`) are mutually exclusive. Not supported in Terminal.
`so` **Note:** In the terminfo database entry for Terminal, this is mapped to inverse because the VT100 "standout" mode is not supported.	`^[[3m`	Enables "standout" display. Not supported in Terminal.
`us`	`^[[4m`	Enables underlined display.

Terminfo capability	Escape sequence	Description
`blink` **Note:** The terminfo entry for Terminal does not support this option.	`^[[5m`	<blink>.
(No equivalent.)	`^[[6m`	Fast blink or strike-through. (Not supported in Terminal; behavior inconsistent elsewhere.)
`mr`	`^[[7m`	Enables reversed (inverse) display.
`invis` **Note:** The terminfo entry for Terminal does not support this option.	`^[[8m`	Enables hidden (background-on-background) display.
	`^[[9m`	Unused.
	Codes `10m`–`19m`	Font selection codes. Unsupported in most terminal applications, including Terminal.
Clearing attributes		
(No equivalent.)	`^[[20m`	"Fraktur" typeface. Unsupported almost universally, and Terminal is no exception.
	`^[[21m`	Unused.
`se` **Note:** Technically, this capability is supposed to end standout mode, but it is overloaded to disable bold bright/dim mode as well.	`^[[22m`	Disables "bright" or "dim" display. This disables either code `1m` or `2m`.
`se`	`^[[23m`	Disables "standout" display. Not supported in Terminal.
`ue`	`^[[24m`	Disables underlined display.
(No equivalent. Use `me` to disable all attributes instead.)	`^[[25m`	</blink>. Also disables slow blink or strike-through (`6m`) on terminals that support that attribute.
	`^[[26m`	Unused.
(No equivalent. Use `me` to disable all attributes instead.)	`^[[27m`	Disables reversed (inverse) display.
(No equivalent. Use `me` to disable all attributes instead.)	`^[[28m`	Disables hidden (background-on-background) display.

Terminfo capability	Escape sequence	Description
	^[[29m	Unused.

Table 7-4 contains a list of capabilities and escape sequences that control text and background colors.

Table 7-4 Color escape sequences

Terminfo capability	Escape sequence	Description
Foreground colors		
setaf 0	^[[30m	Sets foreground color to black.
setaf 1	^[[31m	Sets foreground color to red.
setaf 2	^[[32m	Sets foreground color to green.
setaf 3	^[[33m	Sets foreground color to yellow.
setaf 4	^[[34m	Sets foreground color to blue.
setaf 5	^[[35m	Sets foreground color to magenta.
setaf 6	^[[36m	Sets foreground color to cyan.
setaf 7	^[[37m	Sets foreground color to white.
	^[[38m	Unused.
setaf 9	^[[39m	Sets foreground color to the default.
Background colors		
setab 0	^[[40m	Sets background color to black.
setab 1	^[[41m	Sets background color to red.
setab 2	^[[42m	Sets background color to green.
setab 3	^[[43m	Sets background color to yellow.
setab 4	^[[44m	Sets background color to blue.
setab 5	^[[45m	Sets background color to magenta.
setab 6	^[[46m	Sets background color to cyan.
setab 7	^[[47m	Sets background color to white.
	^[[48m	Unused.
setab 9	^[[49m	Sets background color to the default.

Other Escape Sequences

In addition to providing text formatting, ANSi escape sequences provide the ability to reset the terminal, clear the screen (or portions thereof), clear a line (or portions thereof), and set or clear tab stops.

For example, to clear all existing tab stops and set a single tab stop at column 20, you could use the snippet show in Listing 7-10.

Listing 7-10 Setting tab stops

```
#!/bin/sh
echo # Start on a new line
printf "\e[19C" # move right 19 columns to column 20
printf "\e[3g" # clear all tab stops
printf "\e[W" # set a new tab stop
printf "\e[19D" # move back to the left
printf "Tab test\tThis starts at column 20."
```

Table 7-5 contains a list of capabilities and escape sequences that perform other miscellaneous tasks such as cursor control, tab stop manipulation, and clearing the screen or portions thereof.

Table 7-5 Other escape codes

Terminfo capability	Escape sequence	Description
Resetting the terminal		
`reset` **Note:** This resets many more things than `^[c`. It is also technically not a single capability but rather the contatenation of `rs1`, `rs2`, and `rs3`.	`^[c`	Resets the background and foreground colors to their default values, clears the screen, and moves the cursor to the home position.
Clearing the screen		
`cd`	`^[[J` or `^[[0J`	Clears to the bottom of the screen using the current background color.
(no equivalent)	`^[[1J`	Clears to the top of the screen using the current background color.
`cl`	`^[[2J`	Clears the screen to the current background color. On some terminals, the cursor is reset to the home position.
Clearing the current line		
`ce`	`^[[K` or `^[[0K`	Clears to the end of the current line.
`cb`—Not supported in terminfo entry for Terminal.	`^[[1K`	Clears to the beginning of the current line.

Terminfo capability	Escape sequence	Description
(no equivalent)	`^[[2K`	Clears the current line.
Tab stops		
`hts`	`^[[W` or `^[[0W`	Set horizontal tab at cursor position.
(no equivalent)	`^[[1W`	Set vertical tab at current line. (Not supported in Terminal.)
	Codes `2W`–`6W`	Redundant codes equivalent to codes `0g`–`3g`.
(no equivalent)	`^[[g` or `^[[0g`	Clear horizontal tab at cursor position.
(no equivalent)	^[[1g	Clear vertical tab at current line. (Not supported in Terminal.)
(no equivalent)	^[[2g	Clear horizontal and vertical tab stops for current line *only*. (Not supported in Terminal.)
`tbc`	^[[3g	Clear all horizontal tabs.

Note: You can also set tab stops with the command-line utility `tabs`.

For More Information

The tables in this chapter provide only some of the more commonly used escape sequences and terminfo capabilities. You can find an exhaustive list of ANSI escape sequences at http://www.inwap.com/pdp10/ansicode.txt and an exhaustive list of terminfo capabilities in the manual page for `terminfo`.

Before using capabilities or escape sequences not in this chapter, however, you should be aware that most terminal software (including Terminal in Mac OS X) does not support the complete set of ANSI escape sequences or terminfo capabilities.

Performance Tuning

Shell scripts, when compared with compiled languages, generally do not perform well. However, most shell scripts also do not perform as well as they could with a bit of performance tuning. This chapter shows some common pitfalls of shell scripting and demonstrates how to fix these mistakes.

Avoiding Unnecessary External Commands

Every line of code in a shell script takes time to execute. This section shows two examples in which avoiding unnecessary external commands results in a significant performance improvement.

Finding the Ordinal Rank of a Character (More Quickly)

The Monte Carlo method sample code, found in "An Extreme Example: The Monte Carlo (Bourne) Method for Pi" (page 127), shows a number of ways to calculate the ordinal value of a byte. The version written using a pure shell approach is painfully slow, in large part because of the loops required.

The best way to optimize performance is to find an external utility written in a compiled language that can perform the same task more easily. Thus, the solution to that performance problem was to use `perl` or `awk` to do the heavy lifting. Although they are not compiled languages, both `perl` and `awk` have compiled routines (`ord` and `index`, respectively) to find the index of a character within a string.

However, when using outside utilities is not possible, you can still reduce the complexity by executing outside tools less frequently. For example, once you have an initialized array containing all of the characters from 1–255 (skipping null), you can reduce the number of iterations by removing more than one character at a time until the character disappears, then going back by one batch of characters and working your way forward again, one character at a time.

The following code runs more than twice as fast (on average) as the purely linear search:

```
function ord2()
{
    local CH="$1"
    local STRING=""
    local OCCOPY=$ORDSTRING
    local COUNT=0;
```

```
    # Delete ten characters at a time.  When this loop
    # completes, the decade containing the character
    # will be stored in LAST.
    CONT=1
    BASE=0
    LAST="$OCCOPY"
    while [ $CONT = 1 ] ; do
        LAST=`echo "$OCCOPY" | sed 's/^\(..........\)/\1/'`
        OCCOPY=`echo "$OCCOPY" | sed 's/^.........//'`
        CONT=`echo "$OCCOPY" | grep -c "$CH"`
        BASE=`expr $BASE + 10`
    done
    BASE=`expr $BASE - 10`

    # Search for the character in LAST.
    CONT=1;
    while [ $CONT = 1 ]; do
        # Copy the string so we know if we've stopped finding
        # non-matching characters.
        OCTEMP="$LAST"

        # echo "CH WAS $CH"
        # echo "ORDSTRING: $ORDSTRING"

        # If it's a close bracket, quote it; we don't want to
        # break the regexp.
        if [ "x$CH" = "x]" ] ; then
                CH='\]'
        fi

        # Delete a character if possible.
        LAST=$(echo "$LAST" | sed "s/^[^$CH]//");

        # On error, we're done.
        if [ $? != 0 ] ; then CONT=0 ; fi

        # If the string didn't change, we're done.
        if [ "x$OCTEMP" = "x$LAST" ] ; then CONT=0 ; fi

        # Increment the counter so we know where we are.
        COUNT=$((COUNT + 1)) # or COUNT=$(expr $COUNT '+' 1)
        # echo "COUNT: $COUNT"
    done

    COUNT=$(($COUNT + 1 + $BASE)) # or COUNT=$(expr $COUNT '+' 1)
    # If we ran out of characters, it's a null (character 0).
    if [ "x$OCTEMP" = "x" ] ; then COUNT=0; fi

    # echo "ORD IS $COUNT";

    # Return the ord of the character in question....
    echo $COUNT
    # exit 0
}
```

As you tune, you should be cognizant of the average case time. In the case of a linear search, assuming all possible character values are equally likely, the average time is half of the number of items in the list, or about 127 comparisons. Searching in units of 10, the average is about 1/10 of that plus half of 10, or about 17.69 comparisons, with a worst case of 34 comparisons. The optimal value is 16, with an average of 15.9375 comparisons, and a worst case of 30 comparisons.

Of course, you could write the code as a binary search. Because splitting a string is not easy to do quickly, a binary search works best with strings of known length in which you can cache a series of strings containing some number of periods. If you are searching a string of arbitrary length, this technique would probably be much, much slower than a linear search (unless you use `bash`-specific substring expansion, as described in "Truncating Strings" (page 100)).

Caching the split strings increases initialization time slightly but reduces execution time by about a factor of 2 compared to the "skip 16" version. Whether that tradeoff is appropriate depends largely on how many times you need to perform this operation. If the answer is once, then the extra initialization time will likely erase any performance gain from using the binary search. If the answer is more than once, the binary search is preferable.

Listing 8-1 contains the binary search version.

Listing 8-1 A binary search version of the Bourne shell `ord` function

```
# Initialize the split strings.  This code should be added to ord_init.

    SPLIT=128
    while [ $SPLIT -ge 1 ] ; do
        COUNT=$SPLIT
        STRING=""
        while [ $COUNT -gt 0 ] ; do
                STRING="$STRING"."
                COUNT=$((COUNT - 1))
        done
        eval "SPLIT_$SPLIT=\"$STRING\"";
        SPLIT=$((SPLIT / 2))
    done


function split_str()
{
        STR="$1"
        NUM="$2"
        SPLIT="$(eval "echo \"\$SPLIT_$NUM\"")"
        LEFT="$(echo "$STR" | sed "s/^\\($SPLIT\\).*$/\\1/")"
        RIGHT="$(echo "$STR" | sed "s/^$SPLIT//")"
}


function ord3()
{
    local CH="$1"
    OCCOPY="$ORDSTRING"
    FIRST=1;
    LAST=257

    ord3_sub "$CH" "$ORDSTRING" $FIRST $LAST
}
```

```
function ord3_sub()
{
    local CH="$1"
    OCCOPY="$2"
    FIRST=$3
    LAST=$4

    # echo "FIRST: $FIRST, LAST: $LAST"

    if [ $FIRST -ne $(($LAST - 1)) ] ; then
        SPLITWIDTH=$((($LAST - $FIRST) / 2))
        split_str "$OCCOPY" $SPLITWIDTH
        if [ $(echo "$LEFT" | grep -c "$CH") -eq 1 ] ; then
                # echo "left"
                ord3_sub "$CH" "$LEFT" $FIRST $(( $FIRST + $SPLITWIDTH ))
        else
                # echo "right"
                ord3_sub "$CH" "$RIGHT" $(( $FIRST + $SPLITWIDTH )) $LAST
        fi
    else
        echo $(( $FIRST + 1 ))
    fi
}
```

As expected, this performs significantly better, decreasing execution time by about ten percent in this case. The improved performance, however, is almost precisely offset by the extra initialization costs to enable you to split the list. That is why you should never assume that a theoretically optimal algorithm will perform better than a theoretically less optimal algorithm. In shell scripting, the performance impact of constant cost differences can easily outweigh improvements in algorithmic complexity.

Reducing Use of the eval Builtin

The `eval` builtin is a very powerful tool. However, it adds considerable overhead when you use it.

If you are executing the `eval` builtin repeatedly in a loop and do not need to use the results for intermediate calculations, it is significantly faster to store each expression as a series of semicolon-separated commands, then execute them all in a single pass at the end.

For example, the following code shifts the entries in a pseudo-array by one row:

```
function test1()
{
        X=1; XA=0
        while [ $X -lt 5 ] ; do
                Y=1;
                while [ $Y -lt 5 ] ; do
                        eval "FOO_$X""_$Y=FOO_$XA""_$Y"
                        Y=`expr $Y + 1`
                done
                X=`expr $X + 1`
                XA=`expr $XA + 1`
        done
```

```
}
```

You can speed up this function by about 20% by concatenating the assignment statements into a single string and running `eval` only once, as show in the following example:

```
function test3()
{
        X=1; XA=0
        LIST=""
        while [ $X -lt 5 ] ; do
                Y=1;
                while [ $Y -lt 5 ] ; do
                        LIST="$LIST$SEMI""FOO_$X""_$Y=\$FOO_$XA""_$Y"
                        SEMI=";"
                        Y=`expr $Y + 1`
                done
                X=`expr $X + 1`
                XA=`expr $XA + 1`
        done
        # echo $LIST
        eval $LIST
}
```

An even more dramatic performance improvement comes when you can precache these commands into a variable. If you need to repeatedly execute a fairly well-defined series of statements in this way (but don't want to waste hundreds of lines of space in your code), you can create the list of commands once, then use it repeatedly.

By caching the list of commands, the second and subsequent executions improve by about a factor of 200, which puts its performance at or near the speed of a function call with all of the assignment statements written out.

Another useful technique is to precache a dummy version of the commands, with placeholder text instead of certain values. For example, in the above code you could cache a series of statements in the form `ROW_X_COL_1=ROW_Y_COL_1;`, repeating for each column value. Then, when you needed to copy one row to another, you could do this:

```
eval `echo $ROWCOPY | sed "s/X/$DEST_ROW/g" | sed "s/Y/$SRC_ROW/g"`
```

If you don't have separate variables for source and destination rows, you might write something like the following:

```
eval `echo $ROWCOPY | sed "s/X/$ROW/g" | sed "s/Y/$(expr $ROW + 1)/g"`
```

By writing the code in this way, you have replaced several lines of iterator code and dozens of `eval` instructions with a single `eval` instruction and two executions of `sed`. The resulting performance improvement is dramatic.

Other Performance Tips

Here are a few more performance tuning tips.

Background or Defer Output

Output to files takes time, output to the console doubly so. If you are writing code where performance is a consideration, you should either execute output commands in the background by adding an ampersand (`&`) to the end of the command or group multiple output statements together.

For example, if you are drawing a game board, the fastest way is to store your draw commands in a single variable and output the data at once. In this way, you avoid taking multiple execution penalties. A very fast way to do this is to disable buffering and set newline to shift down a line without returning to the left edge (run `stty raw` to set both of these parameters), then store the first row into a variable, followed by a newline, followed by backspace characters to shift left to the start of the next row, followed by the next row, and so on.

Defer Potentially Unnecessary Work

If the results of a series of instructions may never be used, do not perform those instructions.

For example, consider code that uses `eval` to obtain the values from a series of variables in a pseudo-array. Suppose that the code returns immediately if any of the variables has a value of 2 or more.

Unless you are accumulating multiple assignment statements into a single `eval` statement (as described in "Reducing Use of the eval Builtin" (page 114)), you should call `eval` on the first statement by itself, make the comparison, run eval for the next statement, and so on. By doing so, you are reducing the average number of calls to `eval`.

Perform Comparisons Only Once

If you have a function that performs an expensive test two or more times, cache the results of that test and perform the most lightweight comparison possible from then on.

Also, if you have two possible execution paths through your code that share some code in common, it may be faster to use only a single `if` statement and duplicate the small amount of common code rather than repeatedly performing the same comparison. In general, however, such changes will only result in a single-digit percentage improvement in performance, so it is usually not worth the decrease in maintainability to duplicate code in this way.

The performance impact varies depending on the expense of the test. Tests that perform computations or outside execution are particularly expensive and thus should be minimized as much as possible. Of course, you can reduce the additional impact by performing the calculation once and doing a lightweight test multiple times.

A simple test case produced the results shown in Table 8-1.

Table 8-1 Performance (in seconds) impact of duplicating common code to avoid redundant tests

Test performed twice with one copy of shared code in-between	Test performed once with two copies of shared code
7.003	6.957

Choose Control Statements Carefully

In most situations, the appropriate control statement is obvious. To test to see whether a variable contains one of two or three values, you generally choose an `if` statement with a small number of `elif` statements. For larger number of values, you generally choose a `case` statement. This not only leads to more readable code, but also results in *faster* code.

For small numbers of cases (5), as expected, the difference between a series of `if` statements, an `if` statement with a series of `elif` statements, and a `case` statement is largely lost in the noise, performance-wise, even after 1000 iterations. Although the results shown in Table 8-2 are in the expected order, this was only true approximately half the time. For a smaller number of cases, the differences can largely be ignored.

Table 8-2 Performance (in seconds) comparisons of 1000 executions of various control statement sequences

	eval statement executing multiple functions	series of if statements	if, then series of elif statements	case statement
Five cases	6.945	6..846	6.831	6.807
Ten cases	7.094	7.224	6.980	6.903
Fifty cases	7.023	8.03	7.392	6.704

With a larger number of cases, the results more predictably resemble what one would expect. The `case` version is fastest, followed by the `elif` version, followed by the `if` version, with the `eval` version still coming in last. These results tended to be more consistent, though `eval` was often faster than the series of `if` statements.

Although the performance differences (shown in Table 8-2) are relatively small, in a sufficiently complex script with a large number of cases, they can make a sizable difference. In particular, the `case` statement tends to degrade more gracefully, whereas the series of `if` statements by themselves tends to cause an ever-increasing performance penalty.

Perform Computations Only Once

For example, if you have a function that includes `expr $ROW + 1` in two or more lines of code, you should define a local variable `ROW_PLUS_1` and store the value of the expression in that variable. Caching the results of computation is particularly important if you are using `expr` for more portable math, but doing so consistently results in a small performance improvement even when using shell math.

Table 8-3 Performance (in seconds) of 1000 iterations, performing each computation once or twice

Twice with expr	Once with expr	Twice with shell math	Once with shell math
23.744	12.820	6.596	6.486

Use Shell Builtins Wherever Possible

Using `echo` by itself is typically about 30 times faster than explicitly executing `/bin/echo`. This improved performance also applies to other builtins such as `umask` or `test`.

Of course, `test` is particularly important because it doubles as the bracket (`[`) command, which is essential for most control statements in the shell. If you explicitly wrote a control statement using `/bin/[`, your performance would degrade immensely, Fortunately, it is unlikely that anyone would ever do that accidentally.

Table 8-4 Relative performance (in seconds) of 1000 iterations of the echo builtin and the echo command

echo (builtin)	/bin/echo	printf (builtin)	/usr/bin/printf
0.285	6.212	0.230	6.359

On a related note, the `printf` builtin is significantly faster than the `echo` builtin. Thus, for maximum performance, you should use `printf` instead of `echo`.

For Maximum Performance, Use Shell Math, Not External Tools

Although significantly less portable, code that uses the `zsh`- and `bash`-specific `$(( $VAR + 1))` math notation executes up to 125 times faster than identical code written with the `expr` command and up to 225 times faster than identical code written with the `bc` command.

Use `expr` in preference to `bc` for any integer math that exceeds the capabilities of the shell's math capabilities. The floating-point math used by `bc` tends to be significantly slower.

Table 8-5 Relative performance (in seconds) of 1000 iterations of shell math, expr, and bc

shell math	expr command	bc command
0.111	14.106	25.008

Combine Multiple Expressions with sed

The `sed` tool, like any other external tool, is expensive to start up. If you are processing a large chunk of data, this penalty is lost in the noise, but if you are processing a short quantity of data, it can be a sizable percentage of script execution time. Thus, if you can process multiple regular expressions in a single instance of `sed`, it is much faster than processing each expression separately.

Consider, for example, the following code, which changes "This is a test" into "This is burnt toast" and then throws away the results by redirecting them to `/dev/null`.

```
function1()
{
    LOOP=0
    while [ $LOOP -lt 1000 ] ; do
        echo "This is a test." | sed 's/a/burnt/g' | sed 's/e/oa/g' > /dev/null
```

```
        LOOP=$((LOOP + 1))
    done
}
```

You can speed this up dramatically by rewriting the processing line to look like this:

```
echo "This is a test." | sed -e 's/a/burnt/g' -e 's/e/oa/g' > /dev/null
```

By passing multiple expressions to `sed`, it processes them in a single execution. In this case, the processing of the second expression can be reduced by more than 60% on a typical computer.

As explained in "Avoiding Unnecessary External Commands" (page 111), you can improve performance further by concatenating these strings into a single string and processing the output of all 1000 lines in a single invocation of `sed` (with two expressions). This change reduces the total execution time by nearly a factor of 20 compared with the original version.

For small inputs, the execution penalty is relatively large, so combining expressions results in a significant improvement. For large inputs, the execution penalty is relatively small, so combining expressions generally results in negligible improvement. However, even with large inputs, if the `sed` statements are executed in a loop, the cumulative performance difference could be noticeable.

Table 8-6 Relative performance (in seconds) of different use cases for sed

	Two calls per line (2000 calls total)	One call per line (1000 calls total)	Two calls on accumulated text	One call on accumulated text
Single-processor system	16.874	9.983	0.670	0.665
Dual-processor system	11.460	8.143	0.619	0.612

Other Tools and Information

The final piece to understanding shell scripting (and to understanding other people's shell scripts) is comprehending the use (and abuse) of command-line tools. The scripts listed in this section are commonly used in shell scripts.

Each of these tools has its own syntax and its own quirks. It would be impractical to explain them all in detail. However, this chapter briefly highlights some common tools and includes links to their manual pages for finding additional information about them.

General Tools

The tools in this section are general tools that don't fit into any broad categories.

Table A-1 Commonly-used general scripting tools

Tool	Description
`bc`	Short for "basic calculator", performs floating point math and various other useful calculations that are not practical with basic shell math support.
`expect`	Used to work with hard-to-handle command-line tools that require more complex interaction than is possible with a single pipe. For example, you could use an `expect` script to interact with `getty` over a `tty` or other bidirectional connection to log into a remote computer. In general, scripting that requires two-way interaction between the script and a program is most easily done with an `expect` script.
`expr`	Evaluates a numerical expression. This command supports basic integer math, and is frequently used for incrementing a loop iterator.
`false`	Returns a failure exit status (nonzero).
`sleep`	Pauses execution for a period of time (measured in seconds).
`true`	Returns a successful exit status (`0`).

Text Processing Tools

The tools listed in this section are commonly used for text processing. Unless otherwise noted, these commands take input from standard input (if applicable) and print the result to standard output.

Many of these commands use regular expressions. The syntax of regular expressions is described in "Regular Expressions Unfettered" (page 51). For additional usage notes specific to individual applications, see the manual page for the command itself.

Table A-2 Commonly-used text processing tools

Tool	Description
`awk`	Short for Aho, Weinberger, and Kernighan; a programming language in itself, used for text processing using regular expressions. This is described further in "How awk-ward" (page 63).
`grep`	Short for Global [search for] Regular Expressions and Print; prints lines matching an input pattern (optionally with a specified number of lines of leading and/or trailing context). The `grep` command can take input from standard input or from files. Common variants include `agrep` ("approximate grep" from the Univ. of AZ), `fgrep`, and `egrep`.
`head`	Prints the first few lines from a file (or standard input). The number of lines can be specified with the `-n` flag.
`perl`	A programming language whose scripts can be easily embedded in shell scripts using the `-e` flag. Perl's regular expression language is somewhat richer than basic regular expressions (and easier to read than character classes in extended regular expressions), making it popular for text processing use.
`sed`	Short for stream editor; performs more complex text substitutions using regular expressions.
`sort`	Sorts a series of lines. By default, `sort` reads these lines from its standard input. After its standard input is closed, it sorts them and prints the results to its standard output.
`tail`	Prints the last few lines from of a file (or standard input). The number of lines can be specified with the `-n` flag. Alternatively, you can specify the starting position as a byte or line offset from either the start or end of the file.
`tee`	Copies standard input to standard output, saving a copy into a file (or multiple files).
`tr`	Replaces one character with another.
`uniq`	Filters out adjacent lines that match.

File Commands

These commands are used to manipulate files, including renaming, moving, and deleting files, changing permissions, creating directories, listing files, and so on.

Table A-3 Commonly-used file manipulation tools

Tool	Description
`cd`	Changes the current working directory. The command `cd ..` moves up a directory, for example.
`chflags`	Changes flags on a file or directory. Most of these flags are relatively obscure. For changing permissions flags, use `chmod` instead.
`chgrp`	Changes the group ID associated with a file or directory.
`chmod`	Changes modes (permission bits) or access control lists (ACLs) on a file or directory.
`chown`	Changes the ownership of files or directories. This command can also change the group if desired.
`find`	Lists or searches for files in a directory and its subdirectories.
`ln`	Creates symbolic links and hard links to files or directories.
`ls`	Lists the files in the current directory.
`mkdir`	Creates new directories.
`mkfifo`	Creates named pipes for communication. This is useful in situations where pipes cannot be established while executing the commands, such as connecting two tools in a circular fashion.
`mv`	Moves or renames files and directories.
`rm` and `rmdir`	Removes files and directories
`stat`	Prints detailed file status information, such as the type of file, last modification date, and so on.
`GetFileInfo` and `SetFile`	These tools, installed as part of the Developer Tools installation, are useful for getting and manipulating things like extended attributes. Be aware that if you write a script that depends on these, it will require the Developer Tools to be installed.

Disk Commands

The tools listed in this section perform operations on disks, file systems, partition tables, and disk images.

Table A-4 Commonly-used disk-related and partition-related tools

Tool	Description
`diskutil`	Mounts and unmounts volumes and disks, checks disks for consistency, erases optical disks, wipes disks with a security wipe, partitions disks, manipulates RAID sets, and so on. This utility is the command-line counterpart to the Disk Utility application.
`fsck`, `fsck_msdos`, `fsck_hfs`	Checks a file system for consistency.
`hdiutil`	Creates and manipulates disk images, including attaching disk images for mounting.
`mount` and `umount` (Also `mount_afp`, `mount_autofs`, `mount_cd9660`, `mount_cddafs`, `mount_fdesc`, `mount_ftp`, `mount_hfs`, `mount_msdos`, `mount_nfs`, `mount_ntfs`, `mount_smbfs`, `mount_udf`, `mount_union`, `mount_url`, `mount_volfs`, and `mount_webdav`)	Mounts and unmounts volumes. If you unmount automounted volumes behind the back of the disk arbitrartion system, you can cause strange behavior in the GUI. Use these commands with care, and if you are trying to unmount an automounted volume, use `hdiutil` or `diskutil` instead.

Archiving and Compression Commands

The tools in this section allow you to create archive files that contain copies of multiple files for ease of distribution, to extract the contents of archive files, and compress and decompress files to reduce disk space or network utilization.

The compression tools can also generally be used with pipes to compress data without storing it in a file. The archive tools can generally use standard input or output for reading or writing the archive itself, but not the contents thereof. The `funzip` variant of the zip archiving tool can be used with two pipes, but can only extract the first file from an archive.

Table A-5 Commonly-used archiving and compression tools

Tool	Description
`bzip2`, `bunzip2`, and `bzip2recover`	Compresses and decompresses files using the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. This compression tool takes somewhat longer than other tools such as `gzip`, but tends to result in smaller files, and is thus growing in popularity for distributing large files. Files created with this tool end with the `.bz2` extension.
`compress` and `uncompress`	Compresses and decompresses files using the Lempel-Ziv-Welsh (LZW) compression algorithm. This compression format has largely fallen out of popularity. Files created by this tool end with the `.Z` extension.

Tool	Description
`gzip`, `gunzip`, and `zcat/gzcat`	Compresses, uncompresses, and prints the contents of files in the GNU Zip (LZ77-based) format. This is a compression is popular with UNIX and Linux users. While based on the same underlying compression scheme, the GNU Zip and ZIP file formats are not the same. The ZIP file format can contain multiple files, while the Gzip file format can only contain a single file (though this single file may be a `tar` archive). Files created by this tool end with the `.gz` extension.
`zip`, `unzip`, and `funzip`	Compresses and uncompresses files and directories using the ZIP file format (deflate, based on LZ77 and Huffman coding). This file format is commonly used for exchanging compressed files with Windows users. Files created by this tool end with the .zip extension.
`tar`	Creates, appends to, and extracts multi-file archives in the tar (short for "Tape ARchive") format. This is the standard format for storing multiple files in a single archive among UNIX and Linux users. The tar file format is usually seen in a compressed form, using either `gzip` or `bzip2`. Files created by this tool end with the `.tar` extension (or the `.tgz` or `.tbz` extensions for tar archives compressed with `gzip` or `bzip2`).

For More Information

There are a nearly unlimited number of tools that you might find useful when writing shell scripts. These are just a few of the more common ones. You can find out about the command-line tools that ship as part of Mac OS X by looking in the man pages, either online (*Mac OS X Man Pages*) or by using the `man` command on the command line.

For help finding a command to perform a particular task, you can either search the online version of the man pages or use the `apropos` command on the command line.

Happy scripting!

An Extreme Example: The Monte Carlo (Bourne) Method for Pi

The Monte Carlo method for calculating Pi is a common example program used in computer science curricula. Most CS professors do not force their students to write it using a shell script, however, and doing so poses a number of challenges.

The Monte Carlo method is fairly straightforward. You take a unit circle and place it inside a 2x2 square and randomly throw darts at it. For any dart that hits within the circle, you add one to the "inside" counter and the "total" counter. For any dart that hits outside the circle, you just add one to the "total" counter. When you divide the number of hits inside the circle by the number of total throws, you get a number that (given an infinite number of sufficiently random throws) will converge towards ∏/4 (one fourth of pi).

A common simplification of the Monte Carlo method (which is used in this example) is to reduce the square to a single unit in size, and to reduce the unit circle to only a quarter circle. Thus, the circle meets two corners of the square and has its center at the third corner..

The computer version of this problem, instead of throwing darts, uses a random number generator to generate a random point within a certain set of bounds. In this case, the code uses integers from 0-65,535 for both the x and y coordinates of the point. It then calculates the distance from the point (0,0) to (x,y) using the pythagorean theorem (the hypotenuse of a right triangle with edges of lengths x and y). If this distance is greater than the unit circle (65,535, in this case), the point falls outside the "circle". Otherwise, it falls inside the "circle".

Obtaining Random Numbers

To obtain random numbers, this code example uses the `dd` command to read one byte at a time from `/dev/random`. Then, it must calculate the numeric equivalent of these numbers. That process is described in "Finding The Ordinal Rank of a Character" (page 128).

The following example shows how to read a byte using `dd`:

```
# Read four random bytes.
RAWVAL1=$(dd if=/dev/random bs=1 count=1 2> /dev/null)
RAWVAL2=$(dd if=/dev/random bs=1 count=1 2> /dev/null)
RAWVAL3=$(dd if=/dev/random bs=1 count=1 2> /dev/null)
RAWVAL4=$(dd if=/dev/random bs=1 count=1 2> /dev/null)

# Calculate the ordinality of the bytes.
XVAL0=$(ord "$RAWVAL1") # more on this function later
```

```
XVAL1=$(ord "$RAWVAL2") # more on this function later
YVAL0=$(ord "$RAWVAL3") # more on this function later
YVAL1=$(ord "$RAWVAL4") # more on this function later

# We basically want to get an unsigned 16-bit number out of
# two raw bytes.  Earlier, we got the ord() of each byte.
# Now, we figure out what that unsigned value would be by
# multiplying the high order byte by 256 and adding the
# low order byte.  We don't really care which byte is which,
# since they're just random numbers.
XVAL=$(( ($XVAL0 * 256) + $XVAL1 ))   # use expr for older shells.
YVAL=$(( ($YVAL0 * 256) + $YVAL1 ))   # use expr for older shells.
```

Finding The Ordinal Rank of a Character

There are many ways to calculate the ordinal rank of a character. This example presents three of those: inline `perl`, inline `awk`, and a more purist (read "slow") version using only sed and `tr`.

Finding Ordinal Rank Using Perl

The easiest way to find the ordinal rank of a character in a shell script is by using inline Perl code. In the following example, the raw character is echoed to the `perl` interpreter's standard input. The short Perl script sets the record separator to undefined, then reads data until EOF, finally printing the ordinal value of the character that it retrieves using the `ord` function..

```
YVAL1=$(echo $RAWVAL4 | perl -e '$/ = undef; my $val = <STDIN>; print ord($val);')
```

Finding Ordinal Rank Using awk

This method for obtaining the ordinal rank of a character is slightly more complicated, but still relatively fast. Performance is only slightly slower than the Perl example.

```
YVAL0=$(echo $RAWVAL3 | awk '{
    RS="\n"; ch=$0;
    # print "CH IS ";
    # print ch;
    if (!length(ch)) { # must be the record separator.
            ch="\n"
    };
    s="";
    for (i=1; i<256; i++) {
            l=sprintf("%c", i);
            ns = (s l); s = ns;
    };
    pos = index(s, ch); printf("%d", pos)
}')
```

In this example, the raw character is echoed to an `awk` script. That script iterates through the numbers 1-255, concatenating the character (`l`) whose ASCII value is that number (`i`) onto a string (`ns`). It then asks for the location of that character in the string. If no value is found, index will return zero (0), which is convenient, as `NULL` (character 0) is excluded from the string.

The surprising thing is that this code, while seemingly far more complicated than the Perl equivalent, performs almost as well (less than half a second slower per 100 iterations).

Finding Ordinal Rank Using tr And sed

This example was written less out of a desire to actually use such a method and more out of a desire to prove that such code is possible. This is, by far, the most roundabout way to calculate the ordinal rank of a character that you are likely to ever encounter. It behaves much like the `awk` program described in "Finding Ordinal Rank Using awk" (page 128), but without using any other programming languages other than Bourne shell scripts.

The first part of this example is a small code snippet to convert an integer into its octal equivalent. This will be important later.

Listing B-1 An Integer to Octal Conversion Function

```
# Convert an int to an octal value.
inttooct()
{
        echo $(echo "obase=8; $1" | bc)
}
```

This code is relatively straightforward. It tells the basic calculator, `bc`, to print the specified number, converting the output to base 8 (octal).

The next part of this example is the code to initialize the string containing a list of all of the possible ASCII characters except `NULL` (character 0). This function is called only once at program initialization; the shell version of this code is very slow as it is, and calling this function each time you try to find the ordinal rank of a character would make this code completely unusable.

```
# Initializer for the scary shell ord function.
ord_init()
{
    I=1
    ORDSTRING=""
    while [ $I -lt 256 ] ; do
        # local HEX=$(inttohex $I);
        local OCT=$(inttooct $I);
        # The following should work with GNU sed, but
        # Mac OS X's sed doesn't support \x.
        # local CH=$(echo ' ' | sed "s/ /\\x$HEX/")
        # How about this?
        # local CH=$(perl -e  "\$/=undef; \$x = ' '; \$x =~ s/ /\x$HEX/g; print \$x;")
        # Yes, that works, but it's cheating.  Here's a better one.
        local CH=$(echo ' ' | tr ' ' "\\$OCT");
        ORDSTRING=$ORDSTRING$CH
        I=$(($I + 1)) # or I=$(expr $I '+' 1)
        # echo "ORDSTRING: $ORDSTRING"
    done
}
```

This version shows three possible ways to generate a raw character from the numeric equivalent. The first way works in Perl, and works with GNU `sed`, but does not work with Mac OS X's `sed` implementation. The second way uses the `perl` interpreter. While this way works, the intent was to avoid using other scripting languages if possible.

The third way is an interesting trick. A string containing a single space is passed to `tr`. The `tr` command, in its normal use, substitutes all instances of a particular character with another one. It also recognizes character codes in the form of a backslash followed by three octal digits. Thus, in this case, its arguments tell it to replace every instance of a space in the input (which consists of a single space) with the character equivalent of the octal number `$OCT`. This octal number, in turn, was calculated from the loop index (`I`) using the octal conversion function shown in Listing B-1 (page 129).

When this function returns, the global variable `$ORDSTRING` contains every ASCII character beginning with character 1 and ending with character 255.

The final piece of this code is a subroutine to locate a character within a string and to return its index. Again, this can be done easily with inline Perl code, but the goal of this code is to do it without using any other programming language.

```
ord()
{
    local CH="$1"
    local STRING=""
    local OCCOPY=$ORDSTRING
    local COUNT=0;

    # Delete the first character from a copy of ORDSTRING if that
    # character doesn't match the one we're looking for.  Loop
    # until we don't have any more leading characters to delete.
    # The count will be the ASCII character code for the letter.
    CONT=1;
    while [ $CONT = 1 ]; do
        # Copy the string so we know if we've stopped finding
        # non-matching characters.
        OCTEMP="$OCCOPY"

        # echo "CH WAS $CH"
        # echo "ORDSTRING: $ORDSTRING"

        # If it's a close bracket, quote it; we don't want to
        # break the regexp.
        if [ "x$CH" = "x]" ] ; then
                CH='\]'
        fi

        # Delete a character if possible.
        OCCOPY=$(echo "$OCCOPY" | sed "s/^[^$CH]//");

        # On error, we're done.
        if [ $? != 0 ] ; then CONT=0 ; fi

        # If the string didn't change, we're done.
        if [ "x$OCTEMP" = "x$OCCOPY" ] ; then CONT=0 ; fi

        # Increment the counter so we know where we are.
        COUNT=$((COUNT + 1)) # or COUNT=$(expr $COUNT '+' 1)
        # echo "COUNT: $COUNT"
```

```
    done

    COUNT=$(($COUNT + 1)) # or COUNT=$(expr $COUNT '+' 1)
    # If we ran out of characters, it's a null (character 0).
    if [ "x$OCTEMP" = "x" ] ; then COUNT=0; fi

    # echo "ORD IS $COUNT";

    # Return the ord of the character in question....
    echo $COUNT
    # exit 0
}
```

Basically, this code repeatedly deletes the first character from a copy of the string generated by the `ord_init` function unless that character matches the pattern. As soon as it fails to delete a character, the number of characters deleted (before finding the matching character) is equal to one less than the ASCII value of the input character. If the code runs out of characters, the input character must have been the one character omitted from the ASCII lookup string: `NULL` (character 0).

Complete Code Sample

Note: This complete code listing is also available in the companion files zip archive, which may be found in the table of contents when viewing this chapter in HTML form on the ADC Reference Library website.

```
#!/bin/sh

ITERATIONS=1000
SCALE=6

# Set FAST to "slow", "medium", or "fast".  This controls
# which ord() function to use.
#
# slow-use a combination of perl, awk, and shell methods
# medium-use only perl and awk methods.
# fast-use only perl

# FAST="slow"
# FAST="medium"
FAST="fast"

# 100 iterations - FAST
# real    0m9.850s
# user    0m2.162s
# sys     0m8.388s

# 100 iterations - MEDIUM
# real    0m10.362s
# user    0m2.375s
# sys     0m8.726s

# 100 iterations - SLOW
# real    2m25.556s
```

```
# user    0m32.545s
# sys     2m12.802s

# Calculate the distance from point 0,0 to point X,Y.
# In other words, calculate the hypotenuse of a right
# triangle whose legs are of length X and Y.
distance()
{
    local X=$1
    local Y=$2

    DISTANCE=$(echo "sqrt(($X ^ 2) + ($Y ^ 2))" | bc)

    echo $DISTANCE
}

# Convert an int to a hex value.  (Not used.)
inttohex()
{
    echo $(echo "obase=16; $1" | bc)
}

# Convert an int to an octal value.
inttooct()
{
    echo $(echo "obase=8; $1" | bc)
}

# Initializer for the scary shell ord function.
ord_init()
{
    I=1
    ORDSTRING=""
    while [ $I -lt 256 ] ; do
    # local HEX=$(inttohex $I);
    local OCT=$(inttooct $I);
    # The following should work with GNU sed, but
    # Mac OS X's sed doesn't support \x.
    # local CH=$(echo ' ' | sed "s/ /\\x$HEX/")
    # How about this?
    # local CH=$(perl -e  "\$/=undef; \$x = ' '; \$x =~ s/ /\x$HEX/g; print \$x;")
    # Yes, that works, but it's cheating.  Here's a better one.
    local CH=$(echo ' ' | tr ' ' "\\$OCT");
    ORDSTRING=$ORDSTRING$CH
    I=$(($I + 1)) # or I=$(expr $I '+' 1)
    # echo "ORDSTRING: $ORDSTRING"
    done
}

# This is a scary little lovely piece of shell script.
# It finds the ord of a character using only the shell,
# tr, and sed.  The variable ORDSTRING must be initialized
# prior to first use with a call to ord_init.  This string
# is not modified.
ord()
{
    local CH="$1"
    local STRING=""
```

```
    local OCCOPY=$ORDSTRING
    local COUNT=0;

    # Delete the first character from a copy of ORDSTRING if that
    # character doesn't match the one we're looking for.  Loop
    # until we don't have any more leading characters to delete.
    # The count will be the ASCII character code for the letter.
    CONT=1;
    while [ $CONT = 1 ]; do
    # Copy the string so we know if we've stopped finding
    # non-matching characters.
    OCTEMP="$OCCOPY"

    # echo "CH WAS $CH"
    # echo "ORDSTRING: $ORDSTRING"

    # If it's a close bracket, quote it; we don't want to
    # break the regexp.
    if [ "x$CH" = "x]" ] ; then
        CH='\]'
    fi

    # Delete a character if possible.
    OCCOPY=$(echo "$OCCOPY" | sed "s/^[^$CH]//");

    # On error, we're done.
    if [ $? != 0 ] ; then CONT=0 ; fi

    # If the string didn't change, we're done.
    if [ "x$OCTEMP" = "x$OCCOPY" ] ; then CONT=0 ; fi

    # Increment the counter so we know where we are.
    COUNT=$((COUNT + 1)) # or COUNT=$(expr $COUNT '+' 1)
    # echo "COUNT: $COUNT"
    done

    COUNT=$(($COUNT + 1)) # or COUNT=$(expr $COUNT '+' 1)
    # If we ran out of characters, it's a null (character 0).
    if [ "x$OCTEMP" = "x" ] ; then COUNT=0; fi

    # echo "ORD IS $COUNT";

    # Return the ord of the character in question....
    echo $COUNT
    # exit 0
}

# If we're using the shell ord function, we need to
# initialize it on launch.  We also do a quick sanity
# check just to make sure it is working.
if [ "x$FAST" = "xslow" ] ; then
    echo "Initializing Bourne ord function."
    ord_init

    # Test our ord function
    echo "Testing ord function"
    ORDOFA=$(ord "a")
    # That better be 97.
```

```
    if [ "$ORDOFA" != "97" ] ; then
        echo "Shell ord function broken.  Try fast mode."
    fi

    echo "ord_init done"
fi

COUNT=0
IN=0

# For the Monte Carlo method, we check to see if a random point between
# 0,0 and 1,1 lies within a unit circle distance from 0,0.  This allows
# us to approximate pi.
while [ $COUNT -lt $ITERATIONS ] ; do
    # Read four random bytes.
    RAWVAL1=$(dd if=/dev/random bs=1 count=1 2> /dev/null)
    RAWVAL2=$(dd if=/dev/random bs=1 count=1 2> /dev/null)
    RAWVAL3=$(dd if=/dev/random bs=1 count=1 2> /dev/null)
    RAWVAL4=$(dd if=/dev/random bs=1 count=1 2> /dev/null)

    # ord "$RAWVAL4";
    # exit 0;

    # The easy method for doing an ord() of a character: use Perl.
    XVAL0=$(echo $RAWVAL1 | perl -e '$/ = undef; my $val = <STDIN>; print ord($val);')
    XVAL1=$(echo $RAWVAL2 | perl -e '$/ = undef; my $val = <STDIN>; print ord($val);')

    # The not-so-easy way using awk (but still almost as fast as perl)
    if [ "x$FAST" != "xfast" ] ; then
        # Run this for FAST = medium or slow.
        echo "AWK ord"
        # Fun little awk program for calculating ord of a letter.
        YVAL0=$(echo $RAWVAL3 | awk '{
        RS="\n"; ch=$0;
        # print "CH IS ";
        # print ch;
        if (!length(ch)) { # must be the record separator.
            ch="\n"
        };
        s="";
        for (i=1; i<256; i++) {
            l=sprintf("%c", i);
            ns = (s l); s = ns;
        };
        pos = index(s, ch); printf("%d", pos)
        }')
        # Fun little shell script for calculating ord of a letter.
    else
       YVAL0=$(echo $RAWVAL3 | perl -e '$/ = undef; my $val = <STDIN>; print ord($val);')
    fi

    # The evil way---slightly faster than looking it up by hand....
    if [ "x$FAST" = "xslow" ] ; then
        # Run this ONLY for FAST = slow.  This is REALLY slow!
        YVAL1=$(ord "$RAWVAL4")
    else
       YVAL1=$(echo $RAWVAL4 | perl -e '$/ = undef; my $val = <STDIN>; print ord($val);')
    fi
```

```
    # echo "YV3: $VAL3"
    # YVAL1="0"

    # We basically want to get an unsigned 16-bit number out of
    # two raw bytes.  Earlier, we got the ord() of each byte.
    # Now, we figure out what that unsigned value would be by
    # multiplying the high order byte by 256 and adding the
    # low order byte.  We don't really care which byte is which,
    # since they're just random numbers.
    XVAL=$(( ($XVAL0 * 256) + $XVAL1 ))   # use expr for older shells.
    YVAL=$(( ($YVAL0 * 256) + $YVAL1 ))   # use expr for older shells.

    # This doesn't work well, since we can't seed awk's PRNG
    # in any useful way.
    # YVAL=$(awk '{printf("%d", rand() * 65535)}')

    # Calculate the difference.
    DISTANCE=$(distance $XVAL $YVAL)
    echo "X: $XVAL, Y: $YVAL, DISTANCE: $DISTANCE"

    if [ $DISTANCE -le 65535 ] ; then # use expr for older shells
        echo "In circle.";
        IN=$(($IN + 1))
    else
        echo "Outside circle.";
    fi

    COUNT=$(($COUNT + 1))               # use expr for older shells.
done

# Calculate PI.
PI=$(echo "scale=$SCALE; ($IN / $ITERATIONS) * 4" | bc)

# Print the results.
echo "IN: $IN, ITERATIONS: $ITERATIONS"
echo "PI is about $PI"
```

Document Revision History

This table describes the changes to *Shell Scripting Primer*.

Date	Notes
2008-04-08	Fixed a bug in an awk code sample.
2008-02-08	Added several useful commands to the "Other Tools" chapter.
2007-12-11	Updated for Mac OS X v10.5. Added some basic information about csh and additional awk samples.
2007-10-02	Fixed a typo in an awk code example.
2007-04-03	Added chapter on performance optimization and advanced scripting techniques. Made other minor enhancements.
2006-12-05	Clarified behavior of variable exports. Added explanation of eval command.
2006-11-07	Added chapters on cross-platform scripting and awk.
2006-10-03	Added a section on job control in bash and zsh.
2006-06-28	Fixed a number of typographical errors.
2006-05-23	First version.

Document Revision History