Shell Scripting Primer

2006-05-23

Contents

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 took for bootstrapping other technologies. For example, the `autoconf(1)` 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 assumes 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.

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 9)—introduces basic concepts of shell scripting, including variables, control statements, file I/O, pipes, and redirection.

- "Result Codes, Subroutines, Scoping, and Sourcing" (page 19)—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 25)—explains how to use integer math in shell scripts. This section also explains how to use the `bc(1)` command-line utility or Perl to handle more complex math, such as floating-point calculations.
- "Regular Expressions Unfettered" (page 31)—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 43)—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 45)—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	C-shell-compatible shells
sh	csh
bash	tcsh

Bourne-compatible shells	C-shell-compatible shells
zsh	bcsh (C shell to bourne shell translator)
ksh	

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, this 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.

I/O and Shell Variables

What follows is a very basic shell script:

```
#! /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.

Of course, this script isn’t particularly useful. To make this more interesting, we’ll throw in a few variables.

```
#! /bin/sh

FIRST_ARGUMENT=”$1”
echo “Hello, world $FIRST_ARGUMENT!”
```

This is an example of a variable assignment. The variable $1 contains the first argument passed to the shell script. In this example, we’re making a copy and storing it into a variable called FIRST_ARGUMENT, then printing 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. At that point, the contents of the variable will be inserted. 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

```
=”myfirstcommandlineargument”
```

which is clearly not what you want.

You should also notice the double quotation marks. These are important, as any command-line argument may contain spaces. 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 will be treated as part of a single argument even if the value stored in the variable contains a space.

Warning: Shell scripts also allow 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.

Save this script as `test.sh`. Then type ‘`chmod a+x test.sh`’ to make it executable. Finally, run it with ‘`./test.sh leaders`’. You should see “Hello, world leaders!” printed to your screen.

Now 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
echo -n "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 `-n` flag to the `echo` command. This flag causes echo to suppress the newline that it would otherwise add to the end of the line of output. This is useful when you need to use multiple echo statements to output a single line of text. It also just happens to be handy for prompts.

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 separator). The default behavior is to split inputs everywhere there is a space, but by changing this variable, you can make the shell split the input fields by tabs, semicolons, semicolons, or even the letter 'q'. This is demonstrated in the following example:

```
#!/bin/sh
echo -n "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`.

Bulk I/O Using the cat Command

For small I/O, the `echo(1)` 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(1)` command can be particularly useful.

By itself, the `cat(1)` 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(1)` 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(1)` 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(1)` in action is the `.shar` file format, created by the tool `shar(1)` (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.

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 19). For now, though, it suffices to say that variables 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.

A classic example of a shell variable that is significant to scripts and tools is the `PATH` variable. This variable specifies a list of locations where the shell will search 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).

However, when you modify the `PATH` variable in a shell script, the shell is creating a new, *local* copy of this shell variable that is specific to your script. Any tool executed from your script will be passed the original, global `PATH` that was passed into your shell script from whatever script, tool, or shell that launched it.

To pass a modified version of a shell variable to any script or tool that your shell script calls, you must use the `export(1)` command. For example:

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

Either of these will have the same effect—specifically, they will 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 original value of the `PATH` environment variable is stored, a new version is exported, a command is executed, and the old version is restored.

```
ORIGPATH="$PATH"
PATH="/usr/local/bin:$PATH"
export PATH
# Execute some command here
export PATH="$ORIGPATH"
```

Various Bourne shells also offer a number of other built-in commands that you may find useful, one of the more useful for command-line users being `alias(1)`. This command allows you to assign a short name to replace a longer command. 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(1)`, or for ZSH, `zshbuiltins(1)`.

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.

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`.

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(1)`. The `test` executable is rarely run directly, however. Generally, it is invoked by running `[(1)`, which is just a symbolic link or hard link to `/bin/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 in this example, we have introduced 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.

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(1)`.

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 control features in the Bourne shell scripting: the `$()` operator. This operator, along with its cousin, the back-tick (‘) operator (not to be confused with a normal single quote), will result in the in-line execution of a command. In the case above, the `cat` command is executed, and its standard output is stored in the variable `FOO`.

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)
```

The for Statement

The final control structure in this chapter is the `for` statement. The `for` statement is completely unlike its C equivalent (which requires numeric computation, as described in "Paint by Numbers" (page 25)), 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(1)` command (this is described in more detail in "Regular Expressions Unfettered" (page 31)), the following script renames every file in the current directory that ends with `.JPG` to end in `.jpg`.

```
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
```

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" (page 18).

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 will create a file called `MyFile` and fill it with a single line of text:

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

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

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

You will 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 will create 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" (page 18).

Pipes and File Descriptor Redirection

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(1)` command lists files to its standard output, and the `grep(1)` 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 will 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 input. Thus, the following command redirects standard error to standard output, then pipes the result to grep:

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

It will also often be useful for your script to send something to standard error. The following command will send 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 will notice that the ampersand (`&`) appears to behave somewhat differently than it did in "Basic File Redirection" (page 17). 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.

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 14). However, there are other techniques for obtaining result code information from external scripts. These are described in "Working With Result Codes" (page 19).

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 20).

The scoping rules these subroutines differs from most other programming languages. Shell script variable scoping is explained in "Variable Scoping" (page 21).

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 22).

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 built-in 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.)

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 more complex uses of the if statement, see "The test Command And Bracket Notation" (page 15).

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 environment 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 21) 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.

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 `global` 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 21). 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.

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 28).

The expr Command

In shell scripts, numeric calculations are done using the command `expr(1)`. 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(1) and false(1). 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 control statement in a shell script. What follows is a simple example of a for-next loop written in a shell script:

```
COUNT=0
while [ `expr "$COUNT" '<' '4'` = 1 ] ; 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);
}
```

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(1)), and later adopted by the Z shell (zsh(1)) and the Bourne Again shell (bash(1)). 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 25).

Other Comparisons

As mentioned in,"Shell Script Basics" (page 9), the shell scripting language contains basic equality testing without the use of the `expr(1)` 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 attempt to do inequality testing of strings. 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 less 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.

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` to the number `1`, as expected.

For more information on regular expressions, see "Regular Expressions Unfettered" (page 31).

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(1)` 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(1)` 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(1)`.

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(1)` 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 exampel:

```
# 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: /Marr?y/
grep "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 Classees

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.

Forrtunately, 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(1)`, `awk(1)`, and `grep(1)` 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`.

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 38).

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 38)).

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:

```
/[]\\]/
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 is quoted, which makes it match a 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 38) and "Perl and Python Extensions" (page 39). In addition, quoting parentheses may not do what you would expect in some dialects, as described in "Capture Operators and Variables" (page 38).

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 33), 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
```

Note: The capture operator differs depending on which program is processing the regular expression. The tools `sed(1)`, `awk(1)`, and `grep(1)` (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 40).

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 41)), 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 40).
- 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 38) for information about capturing parts of a regular expression.)
- Addition of non-greedy matching. See "Non-Greedy Wildcard Matching" (page 40) for more information.

- Non-capturing parentheses. See "Non-Capturing Parentheses" (page 41) 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 33). 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-ilne tools.

Other Tools and Information

The final piece to understanding shell scripting is comprehending the gratuitous use of text processing tools. 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 provides information about other places to go to find additional information about them.

Commonly Used Tools

You will commonly find many of the following tools used in shell scripts. Unless otherwise noted, these commands take input from standard input (if applicable) and print the result to standard output.

- `ls(1)`—lists the files in the current directory.
- `find(1)`—lists or searches for files in a directory and its subdirectories.
- `tr(1)`—replaces one character with another.
- `sed(1)`—short for stream editor; performs more complex text substitutions.
- `grep(1)`—short for Global [search for] Regular Expressions and Print; prints lines matching an input pattern. Common variants include `agrep` and `egrep`. The `grep` command can take input from standard input or from files.
- `awk(1)`—short for Aho, Weinberger, and Kernighan; a programming language in itself, used for text processing using regular expressions.
- `perl(1)`—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, making it popular for this sort of use.
- `true(1)`—returns a successful exit status (0).
- `false(1)`—returns a successful exit status (1).
- `expect(1)`—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(8)` over a `tty(4)` or a similar mechanism to log into a remote machine. In general, scripting that requires two-way interaction between the script and a program is most easily done with an `expect` script.
- `expr(1)`—evaluates a numerical expression. This shell builtin supports basic integer math, and is commonly used for looping.

- `bc(1)`—basic calculator. This tool can do floating point math and various other useful calculations that are not practical with built-in shell math support.

Many of these commands use regular expressions. The syntax of regular expressions is described in "Regular Expressions Unfettered" (page 31). For additional usage notes specific to individual applications, see the manual page for the command itself.

Other Tools And Documentation

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(1)` 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(1)` 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 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(1)` 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 46).

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.
VAL1=$(ord "$RAWVAL1") # more on this function later
```

```
VAL2=$(ord "$RAWVAL2") # more on this function later
VAL3=$(ord "$RAWVAL3") # more on this function later
VAL4=$(ord "$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(1)`, inline `awk(1)`, and a more purist (read "slow") version using only sed and `tr(1)`.

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(1)` 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(1)` program described in "Finding Ordinal Rank Using awk" (page 46), 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 A-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(1)`, 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 < 256)) = 1 ] ; 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 -E "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))
        # 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(1)` implementation. The second way uses the `perl(1)` 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(1)`. 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 A-1 (page 47).

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))
        # echo "COUNT: $COUNT"
    done

    COUNT=$(($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

```
#!/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 < 256)) = 1 ] ; 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 -E "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))
    # 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))
    # echo "COUNT: $COUNT"
    done

    COUNT=$(($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 < $ITERATIONS)) = 1 ] ; do      # use expr for older shells.
    # Read four random bytes.
    RAWVAL1=$(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 <= 65535)) = 1 ] ; then
        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
2006-05-23	First version.