Shell Scripting Primer

Contents

Contents

Contents

Contents

Contents

Contents

Tables and Listings

Introduction

Shell scripts are a fundamental part of the OS X programming environment. As a ubiquitous feature of UNIX 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 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 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 File" Zip archive. You can find this archive in the heading area when viewing this document in HTML form on the developer.apple.com website.

Organization of This Document

This document is organized as 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.

- Before You Begin (page 15)—explains how to get a command prompt in OS X and other operating systems, provides pointers to documentation about using the command line interactively, and provides useful command-line tips (such as how to enter control characters).

- Shell Script Basics (page 21)—introduces basic concepts of shell scripting, including variables, control statements, file I/O, pipes, redirection, and argument handling.

- Subroutines, Scoping, and Sourcing (page 83)—describes how to obtain result codes from outside executables, how to write and call subroutines, subroutine variable scoping rules, how to include one shell script inside another (sourcing), and how to use job control to run tasks in the background.

- Paint by Numbers (page 93)—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 100)—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.

- How AWK-ward (page 122)—explains the AWK command, which provides a data-driven programming language based on regular expressions and tabular data.

- Designing Scripts for Cross-Platform Deployment (page 146)—describes key differences in the shell scripting environments provided by various operating systems and provides tips for writing portable scripts.

- Advanced Techniques (page 168)—shows you how to simulate data structures and pointers, perform nonblocking I/O, write timing loops, trap signals, use special built-in shell variables, draw styled text using ANSI color and formatting commands, find the absolute path of a script, use `osascript` to manipulate graphical applications, and use file descriptors and named pipes to treat command-line tools as filters.

- Performance Tuning (page 222)—describes techniques for improving the performance of complex scripts.

- Other Tools and Information (page 267)—provides a basic summary of various commands that may be useful to shell script developers, including links to OS X documentation for each of them.

- Starting Points (page 273)—provides several sample shell scripts and snippets that automate real-world tasks. This appendix also provides links to other complete examples elsewhere in the book.

- An Extreme Example: The Monte Carlo (Bourne) Method for Pi (page 327)—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 a number of numerical and string handling 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!

Before You Begin

Before you begin writing shell scripts, you should familiarize yourself a bit with the shell environment.

Obtaining a Shell Prompt

There are many ways to get shell access, depending on the operating system you are running.

In OS X

There are four ways to get a shell prompt in OS X:

- Run Terminal.

  This is, by far, the easiest way to get a shell prompt. It has the advantage of providing access to other GUI applications at the same time.This is the recommended way to get shell access.

  You can find Terminal in the Utilities folder inside your Applications folder.

- Connect via SSH (secure shell).

  First, enable "Remote Login" in the Sharing preferences pane.

  Next, use the SSH client of your choice to log in. For example, you might use the `ssh` command in Terminal to run scripts on a remote computer. For more information, see the documentation for `ssh`.

- Use the OS X (Mach) console.

  In System Preferences, open the Accounts preference pane (Users in OS X v10.1 and earlier), and set the "Display login window as" setting to "Name and Password". Then log out.

  Next, at the login window, Type `>console` as the username. (Leave the password field blank.)

  You will then see a text-based login prompt. Log in with your "short name" and password.

  Log out (type `exit` or `logout` and press return) to get back to GUI-land (or just enter a few wrong passwords in a row).

- Boot single user.

  This environment is *not* generally recommended for scripting. It takes considerable effort to enable networking, mount external disks, and enable other functionality. Also, the root volume is mounted read-only by default. As a result, this mode is mainly useful for disaster recovery.

In Other UNIX Variants or Linux Variants

In most other UNIX or Linux variants, you can gain access to a shell by running XTerm, GTerm, KTerm, Terminal, or some other similarly named application. Alternatively, if you log into such a machine remotely using `ssh`, you should get a shell prompt as soon as you log in.

Some UNIX or Linux variants provide a text-based login prompt. On these systems, you generally get a shell prompt as soon as you log in.

In Windows

Although Windows does not provide a shell, you can add one by installing Cygwin. Instructions for installing Cygwin are beyond the scope of this document. See http://www.cygwin.com/ for more information.

> **Note:** The Cygwin environment is not a complete UNIX shell scripting environment. The examples in this document have not been tested in Cygwin and are not guaranteed to work correctly in the Cygwin environment.

Familiarize Yourself With the Command Line

Read Command Line Primer (page 255) to get a good overview of how to get things done in a command line environment.

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 the `alias` builtin 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"
```

An alias is only active for the remainder of the current shell session. To make an alias permanent, you must add it to an appropriate script that gets run automatically whenever your shell starts up. See Login Scripts (page 17) to learn how.

For more information, see the manual page for your login shell (for example, `bash`, `csh`, `sh`, `tcsh`, or `zsh`).

Login Scripts

OS X provides support for login scripts and environment property lists to allow you to set environment variables and aliases that are automatically set whenever you run a new shell. There are two ways to do this:

- **Bourne shell (bash, zsh, and so on):**

  To persistently set environment variables and add aliases, you can add the appropriate `alias`, variable assignment, and `export` commands to the following files:

  `~/.profile`—executed automatically for all login shells.

  `~/.bash_profile`—similar to `.profile`, but only runs for `bash` login shells.

  `~/.bashrc` and `~/.zshrc`—executed automatically for all non-login `bash` or `zsh` shells (when you explicitly type `bash` or `zsh` on the command line or run a script that starts with `#!/bin/bash` or `#!/bin/zsh`).

  You may also find it useful to create a `.bashrc` file that sources your `.profile` file. For example:

  ```
  . $HOME/.profile
  ```

  Sourcing is described in more detail in Subroutines, Scoping, and Sourcing (page 83).

- **C shell (csh, tcsh, and so on):**

To persistently set environment variables and add aliases, you can add the appropriate `alias`, `set`, and `setenv` commands to the following files:

> `~/.login`—automatically executes for all login shells.

> `~/.cshrc`—automatically executes for all non-login shells (when you explicitly type `bash` on the command line or run a script that starts with `#!/bin/csh` or `#!/bin/tcsh`).

You may also find it useful to create a `.cshrc` file that sources your `.login` file. For example:

```
source $HOME/.login
```

Sourcing is described in more detail in Subroutines, Scoping, and Sourcing (page 83).

Entering Special Characters

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

Creating Text Files in Your Home Directory

In various parts of this document, you need to create a text file and save it into your home directory.

In Terminal, your home directory is the directory that you are in when you first open the Terminal window.

In the rest of OS X, your home directory can be found in the "PLACES" list in Finder window sidebars, Save dialog sidebars, and so on. It's the icon that looks like a house. Your home directory is also the default location if you create a new finder window by choosing **File** > **New Finder Window** in Finder.

Creating Text Files with TextEdit

Creating a text file in TextEdit is fairly straightforward.

1. Create a new file by choosing **File** > **New** (from the **File** menu).

2. Choose **Format** > **Make Plain Text**.

   By default, TextEdit saves files in Rich Text Format (RTF). Choosing Make Plain Text from the Format menu tells it that you want to work with a plain text file instead.

3. Type or paste in the script as directed in the text.

4. Choose **File** > **Save As**.

5. In the resulting Save dialog, scroll the sidebar on the left until you see the "PLACES" section, and click the house icon beside your username.

6. Name the file as directed in the text and save it.

> **Important:** If you are running OS X v10.7.3, any text files you create with TextEdit may fail to execute with the error "bad interpreter: Operation not permitted". To fix this problem, upgrade to OS X v10.7.4 or later and paste the script into a new file.

Creating Text Files with Xcode

Creating a text file in Xcode is fairly straightforward.

1. Create a new file by choosing **File** > **New** > **File...** (from the **File** menu).

2. Choose "Other" in the "OS X" section of the sidebar, then choose "Shell Script" as the file type.

3. Click the "Next" button.

4. In the resulting Save dialog, click the disclosure triangle so that the entire save panel is visible. Then, scroll the sidebar on the left until you see the "PLACES" section, and click the house icon beside your username.

5. Name the file as directed in the text and save it.

6. Type or paste in the script as directed in the text.

7. Choose **File** > **Save**.

Creating Text Files with pico or nano

If you are logging into a computer remotely using SSH, you must use a text editor that can be run on the command line (unless you use X11 forwarding and an X11-based editor).

The `pico` and `nano` commands are two very easy command-line text editors. At least one of these commands is available in most UNIX or Linux-based operating systems.

To create a text file in NANO or PICO:

1. Type `nano` *filename* or `pico` *filename* and press return. (Type the name of the file you want to create or edit instead of the word *filename*.)

2. Edit the file. Use arrow keys to navigate.

3. When you are finished editing, press Control-O. Adjust the name of the file (if desired), then press return to save the file to disk.

4. To exit the editor, press Control-X.

For other valid commands, see the list of control characters along the bottom of the screen or press Control-G for more complete documentation.

Shell Script Basics

Writing a shell script is like riding a bike. You fall off and scrape your knees a lot at first. With a bit more experience, you become comfortable riding them around town, but also quickly discover why most people drive cars for longer trips.

Shell scripting is generally considered to be a glue language, ideal for creating small pieces of code that connect other tools together. While shell scripts can be used for more complex tasks, they are usually not the best choice.

If you have ever successfully trued a bicycle wheel (or paid someone else to do so), that's similar to learning the basics of shell scripting. If you don't true your scripts, they wobble. Put another way, it is often easy to write a script, but it can be more challenging to write a script that consistently works well.

This chapter and the next two chapters introduce the basic concepts of shell scripting. The remaining chapters in this document provide additional breadth and depth. This document is 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/emulator)

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. This should not be confused with the `dash` shell—an `ash`-derived shell used in some Linux distributions—whose name stands for the Debian Almquist SHell.)

And so on. In general, with the exception of `csh` and `tcsh`, it is usually safe to assume that any modern login shell is 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

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 complex scripts is not recommended. For more information, read "CSH Programming Considered Harmful" at http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/. Although many of the language flaws it describes are fixed by some modern C shells, if you are writing a script that must work on multiple computers across different operating systems, you cannot always guarantee that the installed C shell will support those extensions.

However, the C shell scripting language has its uses, particularly for writing scripts that set up environment variables for interactive shell environments, execute a handful of commands in order, or perform other relatively lightweight chores. To support such uses, the C shell syntax is presented alongside the Bourne shell syntax within this "basics" chapter where possible.

Outside of this chapter, 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`.

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` command. The `echo` command is nearly universal in shell scripting as a means for printing something to the user's screen. (Technically speaking, `echo` is generally a shell builtin, but it also exists as as standalone command, `/bin/echo`. You can read more about the difference between the builtin version and the standalone version in echo (page 154) and Use Shell Builtins Wherever Possible (page 231).)

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 (see Creating Text Files in Your Home Directory (page 18) for help creating a text file) and save the file in your home directory in a file called `test.sh`.

Once you have saved the file in your home directory, 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 variables may or may not begin with a dollar sign, depending on how you are using them. If you want to dereference a variable, you precede it with a dollar sign. The shell then inserts the contents of the variable at that point in the script. For all other uses, you do not precede it with a dollar sign.

> **Important:** You generally do *not* want to prefix the variable on the left side of an assignment statement with a dollar sign. Because `FIRST_ARGUMENT` starts out empty, if you used a dollar sign, the first line:
>
> ```
> $FIRST_ARGUMENT="$1" # DO NOT DO THIS!
> ```
>
> would be expanded by the shell into the following complete gibberish:
>
> ```
> ="myfirstcommandlineargument"
> ```
>
> This is clearly not what you want (and produces an error). Because of the order in which the statement is evaluated, the above assignment statement would still fail with an error even if `FIRST_ARGUMENT` were nonempty. (If you really want to assign a value to a variable whose name is in a different variable, use `eval`, as described in Using the eval Builtin for Data Structures, Arrays, and Indirection (page 168).)

You should also notice that the argument to echo is surrounded by double quotation marks. This is explained further in the next section, Using Arguments And Variables That Contain Spaces (page 25).

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 28).

Using Arguments And Variables That Contain Spaces

Take a second look at the script from the previous section:

```
#!/bin/sh


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

Notice that the echo statement is followed by a string surrounded by quotation marks. Normally, the shell uses spaces to separate arguments to commands. Outside of quotation marks, the shell would treat "Hello," and "world" as separate arguments to `echo`.

By surrounding the string with double quote marks, the shell treats the entire string as a single argument to `echo` even though it contains spaces.

To see how this works, save the script above as `test.sh` (if you haven't already), then type the following commands:

```
./test.sh leaders and citizens
./test.sh "leaders and citizens"
```

The first line above prints "Hello, world leaders!" because the space after "leaders" ends the first argument ($1). Inside the script, the variable $1 contains "leaders", $2 contains "and", and $3 contains "citizens".

The second line above prints "Hello, world leaders and citizens!" because the quotation marks on the command line cause everything within them to be grouped as a single argument.

Notice also that there are similar quotation marks on the right side of the assignment statement:

```
FIRST_ARGUMENT="$1"
```

With most modern shells, these double quotation marks are not required for this particular assignment statement (because there are no literal spaces on the right side), but they are a good idea for maximum compatibility. See Historical String Parsing (page 341) in Historical Footnotes and Arcana (page 341) to learn why.

When assigning literal strings (rather than variables containing strings) to a variable, however, you must surround any spaces with quotation marks. For example, the following statement does not do what you might initially suspect:

```
STRING2=This is a test
```

If you type this statement, the Bourne shell gives you an error like this:

```
sh: is: command not found
```

The reason for this seemingly odd error is that the assignment statement ends at the first space, so the next word after that statement is interpreted as a command to execute. See Overriding Environment Variables for Child Processes (Bourne Shell) (page 30) for more details.

Instead, write this statement as:

```
STRING2="This is a test"
```

Using quotation marks is particularly important when working with variables that contain filenames or paths. For example, type the following commands:

```
mkdir "/tmp/My Folder"
FILENAME="/tmp/My Folder"
ls "$FILENAME"
ls $FILENAME
```

The above example creates a directory in `/tmp` called "My Folder". (Don't worry about deleting it because `/tmp` gets wiped every time you reboot.) It then attempts to list the files in that directory. The first time, it uses quotation marks. The second time, it does not. Notice that the shell misinterprets the command the second time as being an attempt to list the files in `/tmp/My` and the files in `Folder`.

Handling Quotation Marks in Strings

In modern Bourne shells, expansion of variables, occurs *after* the statement itself is fully parsed by the shell. (See Historical String Parsing (page 341) in Historical Footnotes and Arcana (page 341) for more information.) Thus, as long as the variable is enclosed in double quote marks, you do 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\"."
```

> **C Shell Note:** The C shell handling of backslashes within double-quoted strings is different. In the C shell, the previous example should be changed to:
>
> ```
> MYSTRING="The word of the day is "\""sedentary"\""."
> ./test.sh \""leaders"\"
> ```
>
> to achieve the desired effect. This difference is described further in Parsing, Variable Expansion, and Quoting (page 61).

This quoting technique also applies to literal strings within commands entered on the command line. For example, using the script from earlier in Shell Variables and Printing (page 23), the command:

```
./test.sh "\"leaders\""
```

prints the phrase "Hello, world "leaders"!"

The details of quotes as they apply to variable expansion are explained in Parsing, Variable Expansion, and Quoting (page 61). (Variable safety with shells that predate this behavior is generally impractical. Fortunately, the modern behavior has been the norm since the mid-1990s.)

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 Parsing, Variable Expansion, and Quoting (page 61).

Exporting Shell Variables

One key feature of shell scripts is that variables are typically limited in their scope to the currently running script. The scoping of variables is described in more detail in Subroutines, Scoping, and Sourcing (page 83). 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 thus 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 an outside tool. To do this, you must export the variable. These exported variables are commonly known as environment variables because they affect the execution of every script or tool that runs but are not part of those scripts or tools themselves.

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 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 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 OS X), this creates two minor portability problems:

- A script written without the `export` statement may work on some versions of OS X, but will fail on others. You can solve this portability problem by using the `export` builtin, as described in this section.

- 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 30).

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` builtin. 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. (This is not true for other Bourne shell variants; see Using the export Builtin (Bourne Shell) (page 29) for further explanation.)

While automatic export 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 problem 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 variable is empty, so this script just prints a blank line. Save the script as `printmyvar.sh`, then type the following commands:

```
chmod a+x printmyvar.sh      # makes the script executable
MYVAR=7 ./printmyvar.sh      # runs the script
echo "MYVAR IS $MYVAR"       # prints the variable
```

Notice that the assignment statement MYVAR=7 applies `only` to the command that follows it. The value of MYVAR is altered in the environment of the command `./printmyvar.sh`, so the script prints the number 7. However, the original (empty) value is restored after executing that command, so the echo statement afterwards prints an empty string for the value of MYVAR.

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 if 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 builtin is the C shell equivalent to issuing an assignment statement with the `export` builtin 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 23)). 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 behaves 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 command.

The best and simplest way to do this is with 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 265).

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 backtick (`) operator in this fashion is described in Inline Execution (page 68).

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 has access to that temporary copy just as it had access to the original variable. To avoid this, be sure to create the temporary copy using the `set` builtin instead of `setenv`.

Deleting Shell Variables

For the most part, in Bourne shell scripts, when you need to get rid of a variable, setting it to an empty string is sufficient. However, in long-running scripts that might encounter memory pressure, it can be marginally useful to delete the variable entirely. To do this, use the `unset` builtin.

For example:

```
MYVAR="this is a test"
unset MYVAR
echo "MYVAR IS \"$MYVAR\""
```

The `unset` builtin can also be used to delete environment variables.

C Shell Note: The C shell `unset` builtin is identical except that it cannot be used to delete environment variables. Use `unsetenv` instead, as shown in Overriding Environment Variables for Child Processes (C Shell) (page 33).

Also, in C shell, if you try to use a deleted variable, it is considered an error. (In Bourne shell, an unset variable is treated like an empty string.)

Shell Input and Output

The Bourne shell provides a number of ways to read and write files, display text, and get information from the user, including `echo` (described previously in Shell Script Basics (page 21)), `printf`, `read`, `cat`, pipes, and redirection. This chapter describes these mechanisms.

Shell Script Input and Output Using printf and read

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 behavior 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 146) 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 change in behavior 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 replies 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 62).

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` builtin can be used to simulate an array using multiple shell variables. This technique is described in Using the eval Builtin for Data Structures, Arrays, and Indirection (page 168).

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 52).

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 100) or cut. For example:

```
#!/bin/csh
printf "Type three numbers separated by 'q'. -> "

set LINE = "$<"

set NUMBER1 = `echo "$LINE" | cut -f 1 -d 'q'`
set NUMBER2 = `echo "$LINE" | cut -f 2 -d 'q'`
set NUMBER3 = `echo "$LINE" | cut -f 3 -d 'q'`

echo "You said: $NUMBER1, $NUMBER2, $NUMBER3"
```

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
```

This example script takes the text after the line containing the `cat` command up to (but not including) the line that *begins* with `EOF` and stores it into 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 unless you surround it with quotation marks. (These outer quotation marks are not considered part of the token unless you quote them.)

- Shell variables in the name of the token are *not* expanded, so the `$` character is just like any other ordinary character.

- 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 the only thing that appears on the line. If it shares the line with 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 as 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).

Note: Although shell variables cannot be used to define the token itself, by default, shell variables are expanded within the string to be printed. To disable this expansion, surround the token with single or double quote marks. For example:

```
cat << 'EOF'
The variable in this line will not be expanded: $PATH
EOF
```

Notice that EOF does not appear in quotes in the actual text. This is a key difference between the Bourne shell and C shell behavior. If you want to explicitly look for EOF within single quotes, you would write it like this:

```
cat << "'EOF'"
...
'EOF'
```

or

```
cat << \''EOF'\'
...
'EOF'
```

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 multiline `cat` syntax in the C shell is the same as in the Bourne shell, with one key difference: the entire token is treated as literal text for matching purposes, including backslashes and quotation marks. For example:

```
cat << 'EOF'
The variable in this line will not be expanded: $PATH
'EOF'
```

For another example:

```
cat << \''EOF'\'
The variable in this line will not be expanded: $PATH
\''EOF'\'
```

In both cases, the quotation marks still behave as a switch to control whether or not to expand variables within the output.

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 42).

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.

Many (but not all) Bourne-compatible shells support a third operator in this family, the merging redirect operator (>&) that redirects 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.

Compatibility Note: Not all Bourne shell variants support the >& operator when used in this way. This simplified behavior is not specified by POSIX, and a few shells (most notably `ash` and its Debian derivative, `dash`) generate an error if you try to use this operator without specifying a file descriptor number after the >&. For maximum portability, you should redirect standard output to a file, then separately combine standard error into standard output like this:

```
ls . THISISNOTAFILE > filelistwitherrors 2>&1
```

See Pipes and File Descriptor Redirection (Bourne Shell) (page 42) for more information about using file descriptor redirection to combine file descriptors.

Note: The >& operator is also very powerful when used for file descriptor redirection. Additional uses beyond basic use are described in more detail in Pipes and File Descriptor Redirection (Bourne Shell) (page 42) and Scripting Interactive Tools Using File Descriptors (page 211).

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 combines standard error into standard output, then pipes the result to grep:

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

This operator is also often useful if your script needs to send a message 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 40). 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 effect is the same (assuming your shell supports the use of >& by itself).

The redirect (>) operator implicitly redirects standard output. When combined with an ampersand and followed by a filename, in some shells, it merges standard output and standard error and writes the result to a file, though this behavior is not portable. By specifying numbers, your script is effectively overriding which file descriptor to use as its source and specifying a file descriptor to receive the result instead of a file.

Note: Be careful when mixing normal redirection with file descriptor merging. The following command combines standard output and standard error into a single output file.

```
ls . BOGUSFILENAME > filelistwitherrors 2>&1
```

If you reverse the order of the redirects, however, only standard output is written into the file.

```
ls . BOGUSFILENAME 2>&1 > just the file
```

Further, if you pipe the result of the second version above into another utility, it will receive the standard error output from the `ls` command.

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, however, possible to discard standard output and capture standard input. For example:

```
(ls / /bogusfile > /dev/null) |& more
```

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.

Flow Control, Expansion, and Parsing

The topics of flow control, expansion, and parsing may seem somewhat disparate, but they are closely related in the context of Bourne shell scripts.

In particular, because of the token splitting rules, parsing and expansion are most likely to make a behavioral difference in the context of control statements (`if`, `while`, and so on).

Similarly, to fully understand variable expansion, you must understand how it interacts with parsing, including when the contents of variables undergo further token splitting.

Because of the complex relationship between these topics, they are described together in a single chapter.

Basic Control Statements

The examples in previous chapters have been very basic, linear programs. This section shows how to add flow control statements that 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 nonzero 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 statement 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 [ "$A" = "foo" ] ; then
    echo "Foo"
elif [ "$A" = "bar" ] ; 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 48).)

C Shell Note: The C shell syntax is similar to C. There are two forms:

```
#!/bin/csh

set A = "$<"

if ( "x${A}" == "xfoo" ) echo "Foo (single line)"

if ( "x${A}" == "xfoo" ) then
    echo "Foo"
else if ( "x${A}" == "xbar" ) then
    echo "Bar"
else
    echo "Other"
endif
```

Note that the `echo` or `then` statement *must* appear on the same line as the `if` statement. If it does not, you get an "empty if" error and the script terminates.

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`. (For more information about using other commands with the `if` statement, see Working with Result Codes (page 70).)

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:** Although 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="$1"
if [ "$FIRST_ARGUMENT" = "Silly" ] ; then
    echo "Silly human, scripts are for kiddies."
else
    echo "Hello, world $FIRST_ARGUMENT!"
fi
```

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

Second, you should notice the use of double quote marks. This serves two purposes. First, it ensures that even if the variable or string is empty, there is a placeholder. This also ensures that the code will function correctly if the variable's value contains spaces.

If you are looking at older code, you may also see the empty variable problem solved in another way:

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

In this older style, the two arguments to the comparison are preceded by an 'x' (and in this example, on the right side, the 'x' precedes nothing, thus comparing the value to an empty string).

The reason this is needed is because variable substitution occurs before this statement is executed. If you omit the 'x' on the left side and the value in $VARIABLE is empty, then this statement evaluates to "if [= x]", which is a blatant syntax error.

This style is not recommended for new code. It does not handle spaces inside variables, and provides a significant attack vector for arbitrary code injection. See Shell Script Security (page 234) for more information.

> **Note:** This example introduces another special character, the backslash. It 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 use of backslash (and other similar characters) is described further in Quoting Special Characters (page 66).

The test command can also be used for various other tests, including the testing for the existence of a file, basic numerical comparisons, checking whether a path points to a directory, an executable, or a symbolic link, and so on. For example, the −d flag checks whether its argument is a directory, as shown in this snippet:

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

A complete list of flags and operators supported by the test command can be found in the man page test.

> **C Shell Note:** While the test command can be used in the C shell, it is somewhat unusual to do so; the if and while statements in the C shell do not use it as part of their normal syntax.

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 that contains a command to execute. The loop terminates when this command's exit status is false (nonzero).

As with the `if` statement, the most common command used to control looping is the bracket command (as described in The test Command and Bracket Notation (page 48)).

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 technique is described more in Inline Execution (page 68).

At any time during a loop, you can terminate the loop early with the `break` statement or skip ahead to the next iteration of the loop with the `continue` statement. When working with nested loops, these statements may be followed by an optional numerical argument to alter execution of the enclosing loops.

For example, consider the following statements:

```
break 2
continue 2
```

The first statement above (`break 2`) breaks out of not only the top level `while` loop, but also the `while` or `for` loop that contains it. The second statement above (`continue 2`) not only causes the remainder of the current loop to be skipped, but also causes the remainder of the loop that encloses it to be skipped.

C Shell Note: The C shell syntax is similar:

```
set FOO = "x"
while (${FOO} != "")
    set FOO = `cat`
end
```

Just as in C, the `break` and `continue` statements are also supported for further loop control. However, the C shell does not support breaking or continuing at any nesting level other than the topmost level.

The for Statement

The most unusual control structure in this chapter is the `for` statement. It can take two very different forms depending on what you want to do.

In a standard Bourne shell, the `for` statement in shell scripts is completely unlike its C equivalent (which requires numerical computation, as described in Paint by Numbers (page 93)), and actually behaves much like the `foreach` statement in various languages.

In some modern Bourne shell variants, you can also do a numerical version of a for loop. The syntax is nearly identical to the C syntax for for loops.

The two syntaxes are covered in the following sections.

Standard for Loops

The `for` statement in Bourne shell scripts iterates through the items in a list. For each item, it sets the loop variable to the item, then executes a series of statements.

In the next example, the list is `*.JPG`. When the shell performs globbing on this (see Special Characters Explained (page 63) for more information), it replaces the `*.JPG` 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 syntax is described in more detail in Regular Expressions Unfettered (page 100)), 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 62).

At any time during a loop, you can terminate the loop early with the `break` statement or skip ahead to the next iteration of the loop with the `continue` statement. When working with nested loops, these statements may be followed by an optional numerical argument to alter execution of the enclosing loops.

For example, consider the following statements:

```
break 2
continue 2
```

The first statement above (`break 2`) breaks out of not only the top level `for` loop, but also the `while` or `for` loop that contains it. The second statement above (`continue 2`) not only causes the remainder of the current loop to be skipped, but also causes the remainder of the loop that encloses it to be skipped.

C Shell Note: The C shell `foreach` statement is similar.

```csh
#!/bin/csh


foreach i ( *.JPG )
    mv "${i}" `echo ${i} | sed 's/\.JPG$/.x/'`
    mv `echo ${i} | sed 's/\.JPG$/.x/'` `echo ${i} | sed 's/\.JPG$/.jpg/'`
end
```

While the C shell supports the `break` and `continue` statements in a `foreach` loop, it does not support breaking or continuing at any nesting level other than the topmost level.

Extended for Loops

Most modern Bourne shells (including BASH) provide an extension for numerical for loops using a variant of the built-in math operator (double parentheses). You can see this style of `for` loop in the following script. It takes a single argument and counts from 1 up to the number specified in that argument. To demonstrate the concept as succinctly as possible, it makes no attempt to validate its input. You, however, should always do so in your scripts.

```bash
#!/bin/bash


# This is an extension that is supported in
# bash, zsh, and many other recent sh variants,
# but is not always valid.
#
# Usage: for5.sh <number>


for (( i = 1 ; i <= $1 ; i++ )) ; do
        echo "I is $i"
done
```

For maximum portability, however, you should use a `while` loop, as shown below:

```bash
i=1
while [ $i -le $1 ] ; do
```

```
    echo "I is $i"

    i=`expr $i '+' 1`

done
```

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. It is equivalent to 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.

C Shell Note: The C shell `switch` statement is functionally equivalent, but behaves somewhat differently.

Like in C, each `case` statement falls through into the following case statement until the shell encounters a `breaksw` statement, which causes execution to immediately jump out of the entire `switch` statement.

```csh
#!/bin/csh

set LOOP = 0

while ( ${LOOP} <= 20 )
    set VAL = `expr ${LOOP} % 10`
    switch (${VAL})
        case 0:
            echo "ZERO" ; breaksw
        case 1:
            echo "ONE" ; breaksw
        case 2:
            echo "TWO" ; breaksw
        case 3:
            echo "THREE" ; breaksw
        case 4:
            echo "FOUR" ; breaksw
        case 5:
            echo "FIVE" ; breaksw
        case 6:
            echo "SIX" ; breaksw
        case 7:
            echo "SEVEN" ; breaksw
        case 8:
            echo "EIGHT" ; breaksw
        case 9:
            echo "NINE" ; breaksw
        default:
            echo "This shouldn't happen."
    endsw
```

```
    set LOOP = `expr ${LOOP} + 1`


end
```

The expr Command

No discussion of tests and comparisons would be complete without mentioning the `expr` 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 93).

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 fails miserably because the shell interprets the less-than sign as a redirect and tries 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 Parsing, Variable Expansion, and Quoting (page 61).

> **Note:** Be careful when using the `expr` command. Any expression that generates a numerical value (including the string comparison in the previous example) effectively generates two seemingly contradictory results. It returns one value through its exit status and a different numerical value by way of its standard output.
>
> The exit status is zero if a logical expression evaluates to true and one if the expression evaluates to false. The output printed to standard output is one if a logical expression evaluates to true and zero if the expression evaluates to false. Notice that these values are reversed. Be sure to use the exit status when comparing the result to the output of commands like `true`, not the value printed to standard output.
>
> This disparity is only really confusing for computations that return a logical true or false value, of course. The behavior can be explained fairly simply: the `expr` command returns a "success" exit status, zero, if the command prints a value other than zero or an empty string. If it prints a zero or an empty string, its exit status is one (failure).

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.

While normally used for logic purposes, 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 chosen name was $NAME"
```

The "or" operator (|) prints the value of the first expression (`"$1"` in this example) 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"` in this example). If both strings are empty or zero, it prints the number zero. The exit status of the command is zero on success, one if both strings are empty or zero.

> **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"
```

The string to the right side of the colon is a relatively simple regular expression. The period character matches a single character. The asterisk modifies the behavior of the period so that it matches zero or more characters. (Read Regular Expressions Unfettered (page 100) for further explanation.) 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 prints an empty string.

For more information about writing basic regular expressions, read Regular Expressions Unfettered (page 100).

C Shell Note: This behaves the same in C shell as it does in the Bourne shell (apart from the usual syntax differences). For example:

```
#!/bin/csh


set NAME = `expr "${1}" '|' "Untitled"`


echo "The chosen name was ${NAME}"
```

Parsing, Variable 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` builtin. See Using the eval Builtin for Data Structures, Arrays, and Indirection (page 168) 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 Script Input and Output Using printf and read (page 35).

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 52).

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 demonstrates why it is very important to choose record separators correctly.

Cross-Platform Compatibility Note: This treatment of record separators is consistent in all modern Bourne shell variants (ASH, BASH, DASH, KSH, ZSH, newer versions of the sh interpreter, and so on). Some earlier 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.

C Shell Note: Most versions of csh do not allow you to alter the field separator. If you need more precise control over field separators, you can use the cut command in a while loop, incrementing a counter.

```
#!/bin/csh

set IFS = ":"
set LIST = "a:b:c d"
set POS = 1
set i = `echo "${LIST}" | cut -f ${POS} -d ':'`

# Repeat until you get an empty field.  This only works if
# you know you should never encounter an empty field.  Otherwise,
# you must know the number of fields.

while ( "x${i}" != "x" )
    echo $i
    set POS = `expr ${POS} '+' 1`
    set i = `echo "${LIST}" | cut -f ${POS} -d ':'`
end
```

If you cannot guarantee that there are no empty fields in the list, you must first count the fields and use a counter in your loop test. To learn how to count the fields, see The expr Command (page 58). To learn how to use counters, read The expr Command Also Does Math (page 93), substituting the C shell syntax as described in Shell Variables and Printing (page 23) and Inline Execution (page 68) as appropriate.

Special Characters Explained

There are several special characters in shell scripts: a dollar sign ($), an asterisk (∗), a question mark (?), curly braces ({ and }), square brackets ([and]), parentheses ((and)), single and double quote marks (' and "), the backtick mark (`, sometimes called the left single quote mark), and the backslash (\). These characters are treated differently by the shell.

Most of these special characters are used in filename expansion, also known as *globbing*. Globbing characters obey different expansion rules than other characters.

The characters behave as follows:

- Dollar sign ($)—the first character in variable expansion, shell builtin math, and inline execution. Variable names beginning with a dollar sign are expanded regardless of whether they appear inside double quotes. If used outside of double quotes, any globbing characters within the contents of the variable are also expanded. Variable names within the contents are not expanded, however.

- Asterisk (*)—a wildcard character that matches any number of characters in a filename. For example, `ls *.jpg` matches all files that end with the extension `.jpg`. The asterisk is used in globbing.

- Question mark (?)—a wildcard character that matches a single character in a filename. For example, `ls a?t.jpg` matches both `ant.jpg` and `art.jpg`. The question mark is used in globbing.

- Curly braces—matches any of a series of options in a filename. For example, `ls *.{jpg,gif}` matches every file ending with either `.jpg` or `.gif`. Curly braces are used in globbing.

- Square brackets—matches any of a series of characters in a filename. For example, `ls a[rn]t.jpg` matches `art.jpg` and `ant.jpg`, but does not match `aft.jpg`. If the first character is a caret (^), it matches every character except for the characters listed.

  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.

  Square brackets are used in globbing.

- Parentheses—these characters serve multiple purposes, depending on context:

  - Used to mark the beginning of a new subroutine. This is described in Subroutines, Scoping, and Sourcing (page 83).

  - Used to group a chain of operations. This is described in Chaining Execution (page 71).

  - Used for math in some Bourne shell variants. This is described in The Easy Way: Parentheses (page 94).

  - Used in `for` loop iterators supported by some Bourne shell variants. This is described in Extended for Loops (page 54).

- Double-quote marks—disables argument splitting on word boundaries (spaces) and shell expansion of most special characters within the quote marks, with a few exceptions:

    - Variables *are* expanded within double quote marks. The contents of variables, however, are *not* expanded in any way even if they contain globbing characters.

    - Inline execution is also expanded within double quote marks.

    - The backslash character still functions within double quote marks in the Bourne shell and variants thereof, but not in C shell variants.

    Note: Although globbing-related characters are not generally expanded within double quotes, expansion of globbing characters within strings enclosed in double quotes may still occur if the double quotes are on the right side of a variable assignment and the variable is later used without double quotes. For example:

    ```
    FOO="*.c"    # *.c does not get expanded here
    ls $FOO      # *.c DOES get expanded here
    ```

- Single-quote marks—disables argument splitting on word boundaries (spaces) and disables *all* shell expansion (including variables). The backslash is treated just like any other literal character when it appears within single quotes. For example, `'\"'` is a string that contains a backslash and a double quote mark.

- Backtick marks—roughly equivalent to `$()`, these are used to delimit code for inline execution. This technique is described in Inline Execution (page 68).

- Backslash—causes the next character to be treated as a literal character, overriding the special behaviors explained in this section. This technique is described further in Quoting Special Characters (page 66).

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`).

C Shell Note: In Bourne shell variants, globbing occurs anywhere a variable is expanded or a globbing character appears as literal text outside of quotation marks. In the C shell, it is slightly more limited.

Within expressions such as the right half of an `if` statement, the C shell provides two additional operators: 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). Although this operator visually resembles the regular expression operator in Perl, this C shell operator does *not* perform a regular expression comparison.

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 string you wish to quote is not within quote marks, it probably should be. Otherwise, you have to deal with *all* of the shell special characters (described in Special Characters Explained (page 63)) plus any new special characters that might be added in the future. Protecting against special characters is particularly important if your script takes arbitrary user input and passes it as an argument to a command.

However, if your script is not handling user input, you can quote a single 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 single-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".

Similarly, to print a quotation mark, you must either surround it with the opposite type of quotation mark or quote it outside of quotation marks. For example, the following statement will not work:

```
echo "This is \"wrong\" and will cause csh to exit with an error"
```

This fails because the first backslash is treated as part of the string, which is terminated with the quotation mark immediately after it. Because the third quotation mark is not within a string, however, the backslash quotes it, turning it into a literal character. Thus, it does not start a new string. The fourth quotation mark (at the end of the line) then begins a string. As a result, there is no matching double quote mark to end the string and CSH exits with an unmatched quotation mark error.

Instead, you can use either of the following syntaxes:

```
echo "You probably meant "\""this"\"" or "'"'" this"'"'"."
```

In the first part, the string is terminated with a double quote mark followed by a quoted double quote mark (displayed literally), followed by opening a new string with a double quote mark. In the second part, the string is terminated with a double quote mark, followed by a double quote mark within single quotes, followed by opening a new string with a double quote mark.

The construction of code that takes advantage of this parsing difference to execute different code depending on whether it is executing in a Bourne shell or a C shell is left as an exercise for the reader.

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 backtick (`` ` ``) operator (not to be

confused with a normal single quote).

These operators are often used with commands that generate a list of filenames to pass them as the argument list to another command. For example, the `grep` command, when passed the `-l` flag, returns a list of files that match. This technique 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 whose contents contain the word "`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 backtick 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 backtick. 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 backticks or the `$()` operator.

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 backtick (`) 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 inline execution:

- The C shell does not support the `$()` syntax.

- The C shell support for the backtick syntax is somewhat limited in that newline characters 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, then operate on the resulting file.

Result Codes, Chaining, and Flags

This chapter covers concepts related to the arguments that scripts take and the results that they return to their caller. It consists of three parts:

- Working with Result Codes (page 70) explains the numeric result codes that scripts and tools return to the calling scripts or tools. It further explains how scripts can use those values to find out whether a tool succeeded or failed.

  For example, the `if` statement and the `test` command work together to control program flow (as described in Flow Control, Expansion, and Parsing (page 46)). This section explains how this interaction works under the hood.

- Chaining Execution (page 71) takes the concept of result codes one step further, demonstrating how you can make a series of commands execute conditionally depending on whether the previous commands succeeded or failed.

- Handling Flags and Arguments (page 74) tells how to write scripts that take complex flags and arguments.

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 three 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 example is not the best way of testing whether a file exists. It is only intended as 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 48).

C Shell Note: The C shell also supports this technique (with a different syntax) as described in The if Statement (page 46).

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
```

C Shell Note: The C shell exit status variable is called $status.

The third way is by taking advantage of the "and" operator:

```
ls mysillyfilename && echo "File exists."
```

These three code examples should generate the same output. The third technique is explained further in Chaining Execution (page 71).

Chaining Execution

The shell provides three operators for chaining execution:and (&&), or (||) and not (!).

And (&&)

> If the command to the left succeeds (has a zero exit status), the command to the right executes. Otherwise, it does not. The result code returned by this operation is success (zero) only if both commands return zero. Otherwise, its result code is whatever was returned by whichever command failed.

Or (||)

> If the command to the left succeeds (has a zero exit status), the command to the right does not execute. If the command to the left fails, the command to the right does execute. If the leftmost command succeeds, the exit status returned by this operator is zero. Otherwise, the exit status returned is the exit status of the command to the right of the operator.

Not (!)

> Executes the command to the right of the operator. If the command returns a zero exit status, the operator returns a nonzero exit status. If the command returns a nonzero exit status, the operator returns a zero exit status.

The three operators are shown in the following snippet:

```
ls / || ! ls mysillyfilename && echo "Whatever."
```

The operator precedence rules in Bourne shell scripts are very different from those in C. Parentheses are evaluated first, as they can be used to override grouping of operators. After that, however, evaluation of operators occurs in order from left to right.

For example, the following line lists all of the files in the root directory, then echoes "It's a boy":

```
ls / || ls /xy && echo "It's a boy"
```

The || operator takes precedence over the && operator because of left-to-right evaluation rules. The shell shortcuts evaluation of the || operator. Thus, because `ls /` always succeeds, the || operator causes the second `ls` to be skipped entirely, and the statement up to the && operator evaluates to `true` (0). This value is then combined with the echo statement after it by the && operator. Thus, the echo statement executes afterwards.

Note: These rules are very different from the rules in C or most other programming languages. If you substitute function calls in C with the same return values (`true`, `false`, and `true`), the resulting statement behaves very differently. Consider the following statement:

```
if (a() || b() && c()) { ... }
```

If functions a and c return `true` and function b returns `false`, the && operator takes precedence over the || operator. Thus, when the first function call (a) executes and returns true, the || operator shortcuts the rest of the statement. However, the expression as a whole still evaluates to `true` in this case. The reason for this is easier to see if you rewrite the statement with parentheses to show the operator precedence like this:

```
if  (a() ||
        (b() && c())
    ) { ... }
```

You can modify the order of operations (or clarify it to avoid confusing people who are not used to languages without operator precedence) by adding parentheses, as shown in the next snippet:

```
ls / || ( ls /nonexistentfile && echo "file exists" )
```

In this case, because the first `ls` statement is successful, the remainder of the statement is skipped. If you replace the `ls /` with `false`, the failed listing of `nonexistentfile` generates an error message and a nonzero exit status, which in turn causes the `echo` statement to still be skipped.

Of course, the existence of these operators also means that you could write an `if` statement without actually using the `if` keyword, as shown in the following snippet:

```
FOO=3
[ $FOO -eq 3 ] && echo "three"
```

Because this decreases readability, however, this syntax is not recommended. This form is presented here only to help with comprehension of existing scripts.

C Shell Note: The C shell syntax for chaining is identical to the Bourne shell syntax. However, you should be aware that some versions of the C shell have subtle bugs in their logic behavior. If you run into these bugs, adding parentheses around single statements can sometimes help.

Handling Flags and Arguments

Throughout this chapter and previous chapters, examples have shown basic argument handling with variables such as $1, $2, and so on. This is fine for simple scripts, but some scripts call for more advanced argument processing. This section describes several techniques for processing arguments.

Special Multi-argument Variables

The shell provides a number of special variables associated with argument lists:

$#.

Contains the number of arguments.

$*.

Expands to the list of arguments, starting from $1.

If this variable appears outside double quotes, each argument is treated as a single indivisible field for field splitting purposes. For example, if used in the argument list to a command, each original argument is passed to that command as a separate argument.

If this variable appears within double quotes, each argument is separated by the value of the IFS variable, and no field splitting occurs within the resulting block. Thus, if this variable is used as part of the argument list to a command, this entire IFS-delimited string is passed in as a single argument. See Variable Expansion and Field Separators (page 62) for more information about the IFS variable.

Compatibility Note: In AIX, if you surround this variable with quotes, the shell wraps each individual argument with quotes when it expands the variable.

$@.

Expands to the list of arguments, starting from $1.

If this variable appears outside double quotes, argument splitting behavior is not defined by the specification. However, in most shells, text is split as though the entire contents of each argument were inserted as-is, separated by spaces, and without any quotes.

If this variable appears within double quotes, each argument is treated as a single indivisible field for field splitting purposes. Thus, if this variable is used within double quotes as part of the argument list to a command, each original argument is passed as a separate argument to that command.

In addition, if this variable appears within double quote marks along with other text ("BLAH$@BLAH", for example), the portion of the string prior to the $@ is prepended to the first argument, and the portion of the string after the $@ is appended to the last argument.

C Shell Note: This variable does not exist in C shell. Use $* instead.

The following code listings demonstrate the use of these arguments and the subtle differences between them.

Listing 5-1   00_listargs.sh

```
#!/bin/sh


for i in "$@" ; do
echo ARG $i
done
```

Listing 5-2   01_testargs.sh

```
#!/bin/sh


IFS="
"


echo "COUNT: $#"
echo
```

```
echo '\$*'
./00_listargs.sh $*
echo
echo '"\$*"'
./00_listargs.sh "$*"
echo
echo '$@'
./00_listargs.sh $@
echo
echo '"$@"'
./00_listargs.sh "$@"
echo
echo '"foo bar$*bar foo"'
./00_listargs.sh "foo bar$*bar foo"
echo
echo '"foo bar$@bar foo"'
./00_listargs.sh "foo bar$@bar foo"
```

Save these scripts with the filenames shown, then run them by typing `./01_testargs.sh This is a "silly test"` and note the differences in the way these variables behave.

The shift Builtin

The `shift` builtin provides a way to remove arguments from the argument list. Each time you call the `shift` builtin, the first argument is deleted and the remaining arguments are shifted down by one. You can also specify an optional numeric argument to indicate how many times you want to shift the argument list.

The following script demonstrates the shift builtin:

Listing 5-3   02_shift.sh

```
#!/bin/sh


echo "\$1: $1 \$2: $2 \$3: $3 \$4: $4 \$5: $5 \$6: $6"


shift
```

```
echo "\$1: $1 \$2: $2 \$3: $3 \$4: $4 \$5: $5 \$6: $6"


shift 2


echo "\$1: $1 \$2: $2 \$3: $3 \$4: $4 \$5: $5 \$6: $6"
```

Run this script by typing `./02_shift.sh The quick brown fox jumped over the lazy dog.` and notice how the arguments change. Initially, the first six arguments are "`The quick brown fox jumped over`". After the first `shift` statement, the first six arguments are "`quick brown fox jumped over the`". After the second shift statement, the first six arguments are "`fox jumped over the lazy dog`".

> **C Shell Note:** The C shell implementation of the `shift` builtin is somewhat different, though the most basic form is the same. The C shell version does not take a numeric parameter to indicate the number of times to shift, however. Instead, if you pass it an array variable as an argument; the contents of the array are shifted similarly.

The getopts builtin and the getopt command

The `getopts` builtin and the `getopt` command both process a list of arguments in a manner that is similar to the `getopt` function in C. If you are writing a Bourne shell script, the `getopts` builtin is strongly recommended because it is faster, safer, and more flexible. (If you are writing a C shell script, the `getopts` builtin is not available.)

Both `getopt` and `getopts` take an option string as an argument. This option string is constructed as follows:

Simple flag
    Just use the letter of the flag. For example, to add the "`-f`" flag, add the letter "`f`" to the option string.

Flag with argument
    Use the letter of the flag followed by a colon. For example, if you want to accept something like "`-o filename`", you would add "`o:`" to the option string.

As a special option, the `getopts` built-in supports detection of unknown flags and missing arguments. To enable this option, add a colon (`:`) as the first character of the option string.

The getopts Builtin

The `getopts` builtin puts your script in control of the argument parsing process. Each call to `getopts` returns a single flag and, where applicable, the argument to that flag. The syntax is as follows:

```
getopts opt_string user_specified_variable [args]
```

The option string is described above in The getopts builtin and the getopt command (page 77). The user-specified variable is described below. The `getopts` builtin can also optionally take a list of arguments to process. You should generally omit this.

The `getopts` builtin modifies the values of the following variables:

user_specified_variable

    The first option you pass to `getopts` is the name of a variable. The `getopts` variable puts the flag itself into the specified variable (without the leading hyphen).

OPTARG

    The argument value associated with the current flag (if applicable).

OPTERR

    In some shells, if this variable is set to `1`, error reporting by the underlying `getopt` function is enabled. If set to `0`, error reporting is disabled. This is not portable, but it is relatively harmless to set this variable "just in case". This variable is ignored if the first character of the option string is a colon (`:`), which tells `getopts` that the script knows how to handle and report errors.

OPTIND

    The index of the current argument being processed. You should set this to 1 before calling the `getopts` builtin for the first time (or to start over, processing the arguments again using a different set of options).

For example, the following script is a crude variant of the `ls` command. It takes an optional −l flag that enables long listings and an optional −o flag that contains the name of a file into which it writes its output. If no output file is specified, it writes its output to standard output. It also takes an optional path or list of paths that are passed to `ls` as-is.

Listing 5-4   03_getopts.sh

```
#!/bin/sh


DO_LONG=""


# Start processing options at index 1.
OPTIND=1
# OPTERR=1
OUTPUT_FILE=""
```

```
while getopts ":hlo:" VALUE "$@" ; do

    echo "GOT FLAG $VALUE"

    if [ "$VALUE" = "h" ] ; then
        echo "Usage: $0 [-l] [-o outputfile] [path ...]"
        exit 1
    fi
    if [ "$VALUE" = "l" ] ; then
        DO_LONG="-l"
    fi
    if [ "$VALUE" = "o" ] ; then
        echo "Set output file to \"$OPTARG\""
        OUTPUT_FILE="$OPTARG"
    fi
    # The getopt routine returns a colon when it encounters
    # a flag that should have an argument but doesn't.  It
    # returns the errant flag in the OPTARG variable.
    if [ "$VALUE" = ":" ] ; then
        echo "Flag -$OPTARG requires an argument."
        echo "Usage: $0 [-l] [-o outputfile] [path ...]"
        exit 1
    fi
    # The getopt routine returns a question mark when it
    # encounters an unknown flag.  It returns the unknown
    # flag in the OPTARG variable.
    if [ "$VALUE" = "?" ] ; then
        echo "Unknown flag -$OPTARG detected."
        echo "Usage: $0 [-l] [-o outputfile] [path ...]"
        exit 1
    fi
done


# The first non-flag argument is at index $OPTIND, so shift one fewer
# to move it into $1
```

```
shift `expr $OPTIND - 1`


if [ "$OUTPUT_FILE" = "" ] ; then

    ls $DO_LONG "$@"
else
    ls $DO_LONG "$@" > $OUTPUT_FILE
fi


exit $?
```

You should notice two things about this script. First, it takes advantage of the leading colon in the option string. This tells `getopts` that the script knows how to handle errors. Second, it provides two additional options—one for the colon (`:`) flag and one for the question mark (`?`) flag. The colon flags is returned when `getopts` encounters a flag with a missing argument. The question mark flag is returned when `getopts` encounters an unknown flag. These two additional cases are enabled by the leading colon in the option string.

> **Note:** The $? variable is explained further in Working with Result Codes (page 70).

The getopt Command

The `getopt` command takes a different approach than the `getopts` builtin. It processes the entire argument list at once and lets you know whether the argument list matches the list of valid flags or not. If the argument list matches, `getopt` canonicalizes the argument list, putting the flags and their optional arguments first (prior to any non-flag arguments), followed by a single "--" argument to indicate that there are no more flags to process.

> ⚠️ **Warning:** The `getopt` command does not support arguments that contain spaces because of the way it reconstructs the argument list. If at all possible, use the `getopts` builtin instead.
> Because of this limitation, using `getopt` in Bourne shell scripts is strongly discouraged. To avoid encouraging bad behavior, the code snippet in this section is presented exclusively in the C shell dialect.

The syntax of the `getopt` command is as follows:

```
getopt opt_string args
```

The following snippet behaves much like the one in Listing 5-4 (page 78). Unlike in that example, it is not possible to programmatically detect the nature of errors (missing arguments or invalid flags).

Also, as noted previously, filenames containing spaces are not handled correctly by `getopt`. This is not a problem with the script. It is a fundamental limitation of the `getopt` tool and the way its output is parsed.

Cross-Platform Compatibility Note: The GNU (Linux) version of `getopt` provides additional flags that cause it to output a string quoted for a particular shell to work around this limitation. That usage is not portable, however, and is not compatible with the OS X `getopt` implementation.

Listing 5-5   01_getopt.csh

```csh
#!/bin/csh

set OUTPUT_FILE=""
set DO_LONG=""

set argv=`getopt "hlo:" $*`

if ( $status != 0 ) then
    echo "Usage: $0 [-l] [-o outputfile] [path ...]"
    exit 1
endif

while ( "$1" != "--" )
        echo "GOT FLAG $1"
    switch($1)
        case "-h":
            echo "Usage: $0 [-l] [-o outputfile] [path ...]"
            exit 1
        case "-o":
            set OUTPUT_FILE="$2"
            shift
            breaksw
        case "-l":
            set DO_LONG="-l"
            breaksw
    endsw
    shift
```

```
end

shift # remove trailing --

# echo "ARGS: $*"

if ( "$OUTPUT_FILE" == "" ) then
    ls $DO_LONG $*
else
    ls $DO_LONG $* > $OUTPUT_FILE
endif

exit $status
```

Note: The `$status` variable is explained further in Working with Result Codes (page 70).

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 46). However, there are other techniques for obtaining result code information from external scripts. These are described in Working with Result Codes (page 70). You can also make execution of one command be conditional upon the result code returned by another command as described in Chaining Execution (page 71).

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 Subroutine Basics (page 83). You can also write short, simple subroutines inline as described in Anonymous Subroutines (page 84).

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 86).

You may find it useful to include one entire shell script inside another. This subject is covered in Including One Shell Script Inside Another (Sourcing) (page 89).

Finally, you may find it useful to execute outside scripts in the background and check their status at a later time. You can learn about this in Background Jobs and Job Control (page 197).

Subroutine Basics

Subroutines in the Bourne shell look very much like C functions without the argument list. You call these subroutines just like you run a program, and subroutines can be used anywhere that you can 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 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 86) 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 way in which subroutines behave differently than separate scripts behave.

> **C Shell Note:** The C shell does not support subroutines. You can, however, use additional external scripts to simulate them. For very simple subroutines, you can also approximate the functionality with aliases as described in The alias Builtin (page 16).

Anonymous Subroutines

The Bourne shell allows you to group more than one command together and treat them both as a separate command. In effect, you are 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 semicolon is required. Also notice the space after the opening curly brace. This space is also required. Forgetting either of these results in 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.

C Shell Note: The C shell does not support anonymous subroutines. You can, however, use additional external scripts to simulate them. You can also roughly approximate this functionality through careful use of chaining as described in Chaining Execution (page 71). For example:

```
( cd / && ls ) | more
```

Unfortunately, if you need the second command to execute even if the first command fails, you can quickly end up with very unreadable code.

```
((ls /boguslocation || true) && (ls || true)) | more
```

Variable Scoping

Subroutines execute within the same shell instance as the main shell script. As a result, all shell variables are, by default, shared between the 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.

Declaring a Local Variable

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 subsequent changes to the variable MYVAR remain local to this subroutine.

If this subroutine calls itself recursively, a new copy of MYVAR is created for each call to this subroutine, resulting in a call stack much like local variables in C or other languages.

Unlike most other languages, however, if this subroutine calls other subroutines, the local copy of MYVAR is also used by those other subroutines (unless they also declare a local copy of MYVAR). In effect, it is as though the global variable MYVAR were replaced with a new global variable that gets destroyed and replaced with the original when the subroutine returns.

> **Important:** Changes to this variable in subroutines that do not have a `local` declaration of MYVAR will still result in modifications to the global copy of MYVAR except when those subroutines are called from this one.

Using Global Variables in Subroutines

In general, you can freely read and modify global variables within any subroutine. However, there are two situations in which this is not the case:

- Changes to variables previously declared as `local` in the current call stack. This is described further in Declaring a Local Variable (page 86).
- Changes made in subroutines called through inline execution.

If you call a subroutine using inline execution, that subroutine gets a local copy of all shell variables. Changes made to those variables are not propagated back into the main script context because the subroutine gets executed in a separate shell.

The following script demonstrates these concepts:

```
#!/bin/sh


# Demonstrates scoping rules.


changevalue()
{
    NAME="$1"


    eval "$NAME=\"\$(expr \"\$$NAME\" \"+\" \"1\")\""
    eval echo "\$$NAME"
}


localchange()
{
    local X=17


    printf "Local variable X: $X + 1 is: "
    changevalue X
    echo "which is also $X"
}


A=3
printf "$A + 1 is "
changevalue A
echo "which is also $A"


B=3
printf "$B + 1 is "
RESULT="$(changevalue B)"
echo $RESULT
echo "which is NOT $B"


localchange
echo "X in a global context is \"$X\""
```

Note: The use of `eval` is explained in Using the eval Builtin for Data Structures, Arrays, and Indirection (page 168).

Notice that when `changevalue` is called directly, the changes it makes to global variables are propagated back to the main script body. When it is called using inline execution, the changes are lost.

This can cause problems for any subroutine that returns a string and also has side effects. There are two straightforward design patterns that can be used to solve this:

- The subroutine could store its output string in a variable instead of printing it. The caller would then use that variable instead of using inline execution to capture the subroutine's output in a variable.

  If desired, one argument to the subroutine could be the name of the variable to use. By designing it in this way, the caller can specify a variable that is local to the calling subroutine, thus avoiding global namespace pollution.

- The caller can redirect the subroutine's output to a file and subsequently use inline execution with the `cat` command to copy the subroutine's output into a variable.

Both methods are functionally equivalent.

Including One Shell Script Inside Another (Sourcing)

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

To source one script from another, you use the `.` builtin.

For example, create a file containing the subroutine `mysub` from Variable Scoping (page 86). Call it `mysub.sh`. To use this subroutine in another script, you can do the following:

```
#!/bin/sh
MYVAR=4


# The next line sources the external script.
. /path/to/mysub.sh


echo "MYVAR INITIALLY $MYVAR"
```

```
mysub "This is an arg"
echo "MYVAR STILL $MYVAR"
```

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

In addition to using the period (`.`) character, many shells provide a `source` builtin that does the same thing. For example:

```
# This form is less compatible.
source /path/to/mysub.sh
```

The `source` builtin is more popular among former C shell programmers, while the period (`.`) version is more popular among Bourne shell purists. The period version is considered portable.

> **Compatibility Note:** The `source` builtin is a BASH extension that is also supported by ZSH. Other Bourne shell variants do not support this builtin. For maximum portability, you should always use the period (`.`) builtin instead.

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
. 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` builtin 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.

C Shell Note: The C shell supports the `source` builtin, but does not support the period form (`.`).

Finding the Absolute Path of the Current Script

Occasionally, you may write a script that needs to execute itself or needs to source a subroutine library in the same directory. When you do, it can be useful to obtain the absolute path of the script itself.

The shell variable `$0` contains the name passed in on the command line. If the script was executed with an absolute path, this is all you need. However, if the script is in a directory contained in the PATH environment variable, this may contain nothing more than the name of the script.

To obtain the actual path of the script, you must take advantage of the shell's ability to search through the locations in the PATH variable. The following snippet returns the path of the executing script. This path may be relative to the current working directory.

```
SCRIPT="$(which $0)"
```

Your script can then execute itself like this:

```
"$SCRIPT" arguments go here
```

You can get a complete absolute path by adding a few more lines:

```
SCRIPT="$(which $0)"
if [ "x$(echo $SCRIPT | grep '^\/')" = "x" ] ; then
    SCRIPT="$PWD/$SCRIPT"
fi
```

If the path starts with a leading slash (`/`), it is already an absolute path, so you don't need to do anything to it. If it does not, prepending the current working directory turns it into one.

Note: This result is not a minimized absolute path; it may contain references to the current (`.`) or enclosing (`..`) directories. It is, however, an absolute path that is will not break even if your script changes directories or modifies its PATH environment variable.

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 98).

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` is 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 because shell variables inside single quotes are not 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 $3 < -2$, 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 difference 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 script 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 58) in Shell Script Basics (page 21).

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 Also Does Math (page 93).

Common Mistakes

As mentioned in, Shell Script Basics (page 21), 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; it is performing a string comparison, *not* a numeric comparison. Thus the following code will not behave the way you might expect if you assumed a numerical comparison:

```
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 because no comparison is taking place. Instead, this line of code 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.

C Shell Note: The C shell makes this even more difficult, as it does not provide operators for numerical equality at all. Instead, you must do a test like this:

```
if ($A <= $B && !($A < B))
```

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 produces an error message. When doing addition, this mistake 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 98)).

- 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 nonnumeric 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 results in a comparison between the number 2 and the number 1, as expected.

For more information on regular expressions, see Regular Expressions Unfettered (page 100).

Beyond Basic Math

The shell scripting language provides only the most basic mathematical operations on integer values. In most cases, integer operations are sufficient. However, sometimes you may 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 do 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 must be quoted so that they 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.

> **Cross-Platform Compatibility Note:** The most common version of bc (and the one included in 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.

The shell itself does not support regular expressions natively. To use regular expressions, you must invoke an external tool.

Some tools that support regular expressions include:

- `awk`—A scripting language in and of itself. Described further in How AWK-ward (page 122).
- `grep`—Returns the list of lines that match an expression (or the lines that do not match with the `-v` flag). Exits with a status of true (0) if a match occurred or false (1) if no match occurred.
- `perl`—A scripting language with more advanced regular expression functionality.
- `sed`—A tool that performs text substitutions based on regular expressions.

You will see these commands used throughout this chapter.

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

How about 9 * 9?

The quick brown fox jumped over the lazy dog.
```

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

Where Can I Use Regular Expressions?

Regular expressions are most commonly used for text filtering. For example, to change every occurrence of the letter "a" in a string to a capital "A", you might echo the string and pipe the result to `sed` like this:

```
echo "This is a test, this is only a test" | sed 's/a/A/g'
```

You can also use regular expressions to search for strings in a file or a block of text by using the `grep` command. For example, to look for the word `"bar"` in the file `foo.txt`, you might do this:

```
grep "bar" foo.txt
# or
cat foo.txt | grep "bar"
```

Finally, on occasion, it can be useful to use regular expressions in control statements. This advanced usage is described further in .

Types of Regular Expressions

There are three basic types of regular expressions: basic regular expressions, extended regular expressions, and Perl regular expressions. Throughout this chapter, the sections points out areas in which they diverge. This section is just a summary of the differences. For more detail, see the appropriate section.

Basic regular expressions and extended regular expressions differ in the following areas:

- Basic regular expressions use a backslash prior to grouping/capturing parentheses (and prior to pipe operators within these parentheses). Extended regular expressions do not. These operators are described in Grouping Operators (page 108).

- Basic regular expressions use a backslash prior to a plus sign when used to mean "one or more of the previous character or group". Extended regular expressions do not. This operator is described in Wildcards and Repetition Operators (page 104).

- Basic regular expressions use a backslash prior to a question mark when used to mean "zero or one of the previous character or group". Extended regular expressions do not. This operator is described in Wildcards and Repetition Operators (page 104).

Perl regular expressions are equivalent to extended regular expressions with a few additional features:

- Perl can (optionally) use a dollar sign instead of a backslash to represent variables in substitution patterns, as described in Capturing Operators and Variables (page 112).

- Perl supports noncapturing parentheses, as described in Noncapturing Parentheses (page 119).

- The order of multiple options within parentheses can be important when substrings come into play, as described in Grouping Operators (page 108).

- Perl allows you to include a literal square bracket anywhere within a character class by preceding it with a backslash, as described in Quoting Special Characters (page 111).

- Perl adds a number of additional switches that are equivalent to certain special characters and character classes. These are described in Character Class Shortcuts (page 117).

- Perl supports a broader range of modifiers. These are described in Using Modifiers (page 115).

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 searches 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 matches the word "Mary", but only when it appears at the beginning of a line. Similarly, the following matches 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 of concern when performing substitutions. The global flag changes this so that a substitution alters every match in the line instead of just the first one.

For example:

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

This replaces only the first occurrence of "Mary" with "Joe." By adding the global flag to the expression, it instead replaces 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 searches 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". The most common way to solve this is with the plus (+) operator. However, you can construct this expression in several 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 extended regular expressions; in basic regular expressions, it must be preceded by a backslash.

The first dot in the third expression matches a single character. The dot-asterisk afterwards matches be zero or more additional characters. Thus, these three statements are equivalent.

The final useful repetition operator is the question mark operator (?). This operator matches zero or one repetitions of whatever precedes it.

Note: Like the plus operator, this differs in appearance depending on whether you are using basic or extended regular expressions; in basic regular expressions, it must be preceded by a backslash.

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 matches 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 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 brackets, these define commonly used sets of characters. The most broadly supported set of predefined character classes are the POSIX character classes:

[:alnum:]—all alphanumeric characters (a-z, A-Z, and 0-9).

[:alpha:]—all alphabetic characters (a-z, A-Z).

[:blank:]—all whitespace within a line (spaces or tabs).

[:cntrl:]—all control characters (ASCII 0-31).

[:digit:]—all numbers.

[:graph:]—all alphanumeric or punctuation characters.

[:lower:]—all lowercase letters (a-z).

[:print:]—all printable characters (opposite of [:cntrl:], same as the union of [:graph:] and [:space:]).

[:punct:]—all punctuation characters

[:space:]—all whitespace characters (space, tab, newline, carriage return, form feed, and vertical tab). (See note below about compatibility.)

[:upper:]—all uppercase letters.

[:xdigit:]—all hexadecimal digits (0-9, a-f, A-F).

For example, the following is another way to match any sentence containing Mary and lamb (but not if there are punctuation marks between them):

```
# Expression: /Mary[[:alpha:][:digit:][:blank:]][[:alpha:][:digit:][:blank:]]*lamb/
grep 'Mary[[:alpha:][:digit:][:blank:]][[:alpha:][:digit:][:blank:]]*lamb' poem.txt
```

Compatibility Note: Not all tools fully support POSIX character classes. In particular:

- The grep tool does not support [:space:] because this character class includes line break characters, which makes no sense in a tool that is designed to print lines that match a pattern.

- The sed tool accepts [:space:] but treats it like [:blank:] for the same reason.

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 matches this pattern, but other things (including the period character) do 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 matches 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.

> **Note:** The syntax for grouping also results in a capture. This process is described in Capturing Operators and Variables (page 112).

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`.

You can also use the grouping syntax to provide multiple options, any one of which is treated as a match. Expressions enclosed in parentheses match any one of a series of smaller expressions separated by a pipe (|) operator. For example, to search for Mary, lamb, or had, you might use this expression:

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

Because regular expressions generally match from left to right, you should be careful when working with multiple options that are substrings of one another during substitution and be sure to place the larger of the possible matches first. Some regular expression engines always take the longer match, while other regular expression engines always take the leftmost match.

For example, the following lines give the same result:

```
sed -E 's/(lamb|lamb,)/orange/' poem.txt
sed -E 's/(lamb,|lamb)/orange/' poem.txt
```

However the following lines do not:

```
perl -pi.bak -e 's/(lamb|lamb,)/orange/' < poem.txt
perl -pi.bak -e 's/(lamb,|lamb)/orange/' < poem.txt
```

In Perl, when the input contains the word "lamb" followed by a comma, the regular expression engine matches the word "lamb" first because it is the leftmost option. It replaces it with the word "orange" and leaves the comma. In the second option, because the version with a comma matches first, the comma is deleted if it is there.

You can, of course, also avoid this problem by writing the expression as:

```
perl -pi.bak -e 's/lamb,?/orange/' < poem.txt
```

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 Capturing Operators and Variables (page 112)).

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 can search for one of these characters. This is where quoting comes in.

In regular expressions, certain nonletter 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 immediately inside the outer slashes 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 nonalphanumeric character can safely be quoted whether it is necessary to do so or not. If quoting it is not necessary, the extra backslash is simply 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 Capturing Operators and Variables (page 112) and Perl and Python Extensions (page 116). In addition, quoting parentheses may not do what you might expect in some dialects, as described in Capturing Operators and Variables (page 112).

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

Capturing Operators and Variables

In Wildcards and Repetition Operators (page 104), 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.

Capturing 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 capturing 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."

```
# Expression (Basic): s/The (.*) brown (.*) the (.*) dog/The \3 brown \2 the \1
dog/
sed "s/The \(.*\) brown \(.*\) the \(.*\) dog/The \3 brown \2 the \1 dog/" <
poem.txt


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


# Perl supports extended form, but also supports
# using a dollar sign for the variable name.  (Note
```

```
# the use of single quotes to prevent the shell from

# doing variable substitution on $1, $2, and $3.)

perl -pi.bak -e "s/The (.*) brown (.*) the (.*) dog/The \3 brown \2 the \1 dog/"
< poem.txt

perl -pi.bak -e 's/The (.*) brown (.*) the (.*) dog/The $3 brown $2 the $1 dog/'
< poem.txt
```

Note: The syntax of the capturing operator differs depending on whether you are using basic, extended, or Perl regular expressions.

Compatibility 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.

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. By default, they 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." becomes simply "Joe.".

If you want to only match up to the *first* occurrence of "lamb", you must either use a Perl regular expression dialect extension, as described in Nongreedy Wildcard Matching (page 118) or use a greedy regular expression from the other end of the string to replace the word "lamb" with another word that is known to not occur elsewhere in the input.

For example:

```
sed 's/lamb\(.*\)$/UNMATCHABLE\1/' < poem.txt | sed 's/^.*UNMATCHABLE/Joe/'
```

This statement produces the line "Joe looked like a lamb."

Mixing Capturing and Grouping Operators

Since parentheses serve both as capturing 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 Noncapturing Parentheses (page 119)), you can use noncapturing 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.

Using Modifiers

The overall behavior of a regular expression can be tuned using a number of modifiers. For example:

```
/foo/i
```

In this example, the `/i` modifier makes the regular expression match in a case-insensitive fashion. Thus, this matches both "Foo" and "fOo".

Not all commands and languages support all modifiers. For example, most versions of the `sed` command support only the `/g` modifier.

The basic modifiers are:

- `/g`—replace globally. Without this flag, a substitution command replaces only the first matching occurrence per line. With this flag, a substitution command also replaces subsequent matches.

- `/i`—use case insensitive matching (Perl extension; equivalent to `grep -i`).

- `/m`—multiline matching (Perl extension). the `$` and `^` anchors should match at newline boundaries in addition to matching at the beginning an end of the string as a whole. The dot (`.`) does not match newline characters.

- `/o`—compile once (Perl extension). In Perl, if a regular expression includes a variable as part of the pattern, the regular expression engine must recompile the expression every time it is used because the variable contents might have changed.

  If you know that the contents will not change after they are set the first time, the `/o` flag disables recompilation of the expression. For regular expressions that do not contain variables, this switch has no effect.

- `/s`—single-line matching (Perl extension). The `$` and `^` anchors should *not* match at newline boundaries. With this modifier, they only match at the very beginning and end of the string as a whole. The dot (`.`) matches newline characters just like any other character.

- /x—extend readability (Perl extension). This mode causes matching to ignore all whitespace between tokens in the expression unless quoted or wrapped in brackets (in most languages) and to treat a hash mark (#) as the start of a single-line comment.

> **Note:** Not all whitespace is ignored; multicharacter tokens like \d must not be split or they will be interpreted differently.

The purpose of this mode is to allow you to split complex regular expressions into multiple lines. For example, in Perl, you might detect a date like this:

```
if ($foo =~ /(\d\d\d\d) # year
    \s*-\s* # separator
    (\d\d) # month
    \s*-\s* # separator
    (\d\d) # day
    /x) {
        print "Date detected\n";
}
```

The syntactical details vary from language to language.

Perl and Python Extensions

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

- Addition of shortcuts for character classes. See Character Class Shortcuts (page 117).

- Addition of quotation operators. In a regular expression, the contents of variables appearing between \Q and \E are automatically quoted, and thus treated as literal text even if the variable contains characters that ordinarily have special meaning in a regular expression. These operators are useful when user input, stored in a Perl variable, is used as part of a regular expression.

- Support for retrieving captured values outside the scope of the expression; the captured values are stored in the variables $1, $2, and so on. (See Capturing Operators and Variables (page 112) for information about capturing parts of a regular expression.)

> **Note:** In PHP, these captured values are passed back in an array that you can provide as an optional argument.

- Addition of nongreedy matching. See Nongreedy Wildcard Matching (page 118) for more information.

- Noncapturing parentheses. See Noncapturing Parentheses (page 119) for more information.

You can find links to additional resources that describe these extensions in For More Information (page 119).

Character Class Shortcuts

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

\A—anchors matching to the beginning of the string as a whole (but not the beginning of lines within the string).

This shortcut is not broadly supported outside of Perl. In other languages, use ^ and add the /s modifier (or do not specify the /m modifier, depending) to specify line-at-once matching.

\b—word boundary (see note).

\B—nonword 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.

\u—a single Unicode character in JavaScript regular expressions. This shortcut must be followed by four hexadecimal digits.

\v—vertical tab.

\w—equivalent to [:word:].

\W—equivalent to [^:word:].

\x—start of an ASCII character code (in hex). For example, \x20 is a space.

\X—a single Unicode character (not supported universally). This shortcut must be followed by four hexadecimal digits.

\z—anchors matching to the end of the string as a whole (but not the end of lines within the string).

This shortcut is not broadly supported outside of Perl. In other languages, use $ and add the /s modifier (or do not specify the /m modifier, depending) to specify line-at-once matching.

\Z—anchors matching to the end of the string as a whole (but not the end of lines within the string). In some languages (including Perl), this matches prior to the closing line break if the string ends with a line break. To avoid this, use \z instead.

This shortcut is not broadly supported outside of Perl. In other languages, use $ and add the /s modifier to specify line-at-once matching.

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

> **Note:** Word boundaries (the \b and \B switches) do not exist in basic or non-Perl extended regular expressions. These 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 nonword character that occur consecutively.
>
> For substitution purposes, "replacing" a word boundary with text is equivalent to inserting that text, much like replacing other anchors such as ^ or $.

Nongreedy 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 nongreedy wildcard matching. To convert a greedy repeat operator to a nongreedy repeat operator, 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 matches "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 matches only the words "Mary had a little lamb". The +? operator behaves similarly.

Noncapturing Parentheses

You may notice that the syntax for capture is identical to the syntax for grouping described in Wildcards and Repetition Operators (page 104). In most cases, the additional captures are not a problem. However, in some cases (particularly when splitting strings into arrays in Perl), 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, 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 -pi.bak -e "s/Mary (?:had )*a little lamb\./Lovely day, isn't it?/" < poem.txt
```

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

For More Information

This chapter covers regular expressions as they apply to shell scripts. While it covers some of the more interesting extensions provided by languages such as Perl, it is by no means a complete reference to Perl regular expressions.

For a thorough explanation of Perl regular expressions and additional features and quirks in various programming languages, see http://perldoc.perl.org/perlre.html and http://www.regular-expressions.info/.

Using Regular Expressions in Control Statements

The shell's `test` command (described in The test Command and Bracket Notation (page 48)) does not natively support regular expressions, so in order to use regular expressions in control statements, you must take advantage of the ability to execute arbitrary external commands (more specifically, the `grep` command) instead of using bracket notation.

As shown throughout this chapter, the `grep` command takes a stream of text (or a path or list of paths) and prints every line that matches the specified regular expression. What you may not have noticed, however, is that its exit status changes depending on whether the input matches the specified expression.

The `grep` command exits with a successful exit status (`0`) if the input matches the specified expression at least once or a failed exit status (generally `1`) if the pattern does not match. Thus, you can easily use it to control an `if` statement.

For example:

```
if (echo "$MYVAR" | grep "bar" > /dev/null) ; then
    echo "The value of MYVAR ($MYVAR) contains \"bar\"."
fi
```

In the above example, the rightmost exit status (from `grep`) is treated as the exit status for the group of commands (assuming that the `echo` command succeeds, which it always should). The redirect to `/dev/null` prevents the text output from being printed to the user's screen.

> **Performance Note:** Regular expressions should not be used if standard shell tests can do the same thing. Regular-expression-based tests are much slower than built-in shell tests because of the need to execute multiple external commands.

Regular expressions can also be used in other control statements such as `while` loops. For example, the following snippet counts the occurrences of the letter 'x' in a single-line string:

```
MYVAR="xxxxxx"
while (echo "$MYVAR" | grep 'x' > /dev/null) ; do
    # Be sure to change MYVAR here!
    echo "got x"
    MYVAR="$(echo "$MYVAR" | sed -E 's/x//')"
done
```

Of course, this contrived snippet is a good example of when you should avoid regular expressions; testing for an empty string makes this snippet run roughly twice as fast.

How AWK-ward

This chapter is a primer to help you learn how to use the AWK programming language and the `awk` interpreter. The `awk` interpreter, much like `sed`, `grep`, and `perl`, is a commonly used text processing tool based on regular expressions.

For more detailed reference material, see the manual page for `awk`, the GNU AWK manual (http://www.gnu.org/software/gawk/manual/), and Brian Kernighan's book, *The AWK Programming Language*.

This chapter uses the file `poem.txt` from Regular Expressions Unfettered (page 100) as the basis for most of its examples. Be sure to create that file before attempting any of these examples.

These examples are tested primarily on the OS X version of AWK, which is derived from "The One True AWK" by Brian Kernighan. Please report any compatibility problems with other versions of AWK using the feedback links at the bottom of each page.

What Is AWK?

AWK is a language designed primarily for processing structured data records containing text. This language is executed by the `awk` interpreter.

The design of AWK centers around dividing the input text into records, each one containing a number of fields. Each time the `awk` interpreter encounters a record separator, it begins a new record. By default, the record separator is a newline character, though you can change this as described in Changing the Record and Field Separators in AWK Scripts (page 129).

After the `awk` interpreter has read a complete record from the input, it divides that record into fields. The fields are delimited by a field separator, similar to the field separators described in Variable Expansion and Field Separators (page 62).

An AWK script is divided into a series of rules. Once the `awk` interpreter has divided a record into fields, it executes these rules in sequence. Each rule has access to variables that contain the record as a whole and the individual fields of that record. The rules can then perform various modifications to that data, print the data, and so on.

A Simple AWK Script

At its most basic, the syntax of 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 133).)

- Variables maintain their value across multiple records and files. They are set until explicitly cleared.

Unlike shell scripts (but like C), variables in AWK scripts 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
```

> **Important:** Be sure to save this file with UNIX-style line endings (newline) and not Mac-style (carriage return) or Windows-style (carriage return and line feed). AWK splits records on newline characters by default. For more information, see Cross-Platform Line Endings (page 147).

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 scripts, 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 scripts 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 might 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 using the `awk` interpreter 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 100).

Expression Ranges in awk

In AWK scripts, when you combine two expressions with a comma (`,`), the action is applied 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
```

The `awk` interpreter prints every line in the poem file 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 133).

Relational Expressions in AWK

In addition to regular expressions, AWK scripts support 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.

AWK scripts support four 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. (See Working with Arrays in AWK (page 133) for more information on working with arrays.)

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 with the awk interpreter by typing:

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

The script 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,
```

Special Patterns in AWK: BEGIN and END

AWK scripts support 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 Scripts (page 129).

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 with the awk interpreter 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, AWK scripts also allow you to use arbitrary 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, but performs the matching in a case-insensitive fashion. 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 130).

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 Scripts

In AWK scripts, the default record separator is a newline, but you can change this by modifying the regular expression stored in the variable RS. Likewise, the default field separator, stored in the variable FS, is a regular expression that matches spaces and tabs.

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 described in Special Patterns in AWK: BEGIN and END (page 127).

By the time any other filter rule executes, the awk interpreter 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.

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 `BEGIN` 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 contain either a regular expression or a literal string 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`.

Control Statements in AWK

Control statements in AWK scripts are syntactically almost identical to C control statements.

The if Statement

As in C, the `if` statement looks like this:

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

Note: The expression format is described in Relational Expressions in AWK (page 126).

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

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

```
{

        i=4

        if ($0 ~ /Mary/) {

                while (i) {

                        print i ":" $0;

                        i--;

                }

        }

}
```

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

The for Statement

The `for` statement 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 `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 133).

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;
```

The `if` statement syntax is described in Control Statements in AWK (page 130).

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 scripts, 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

Arrays in AWK scripts are syntactically very similar to arrays in C. Don't let that fool you, though. Under the hood, they behave very differently.

Arrays in AWK scripts 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 scripts 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 scripts 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];
                }
                if (!(i in my_array)) {
                        print i " is unset.";
                }
        }
}
```

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) { ... }
```

As with any expression, you can invert matching with an exclamation point. For example, to check to see if a particular index has never been stored in an array, you could write the following:

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

> **Note:** Generally speaking, the AWK language is designed under the assumption that you will do any array sorting externally (after the `awk` interpreter 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 count_elements(input_array)
{
    counter=0;
    for (word in input_array) {
        counter++;
    }
    return counter;
}
function join(input_array, separator) {
        string = "";
        first = 1;


        # Note: this preserves order, but does not
        # work with nonnumeric or sparse arrays.
        for (i=1; i<=count_elements(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, " ");

}
```

Compatibility Note: Previous versions of this script used the built-in `length` function to obtain the number of elements in an array (instead of the `count_elements` function). While this technique works in most versions of AWK released since 2002, it does not work in GNU AWK or its derivatives within the context of a function if the array was passed as one of the function's arguments.

Although this bug has been fixed in the official GNU AWK source repository and should be fixed in versions of GNU AWK after version 3.1.6, for maximum portability, you should still avoid using the `length` function in this way.

Deleting Array Elements

As you saw in Array Basics (page 134), 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 scripts 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 contains the second string). This demonstrates that the first two print statements both sent their output 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 scripts (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`, the `awk` interpreter 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).

The following AWK script 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.

Integrating AWK Scripts with Shell Scripts

It is often useful to combine AWK scripts with shell scripts to perform various tasks. This creates two challenges: getting information into an AWK script (beyond the bulk data read via standard input) and getting information back out in a form that is usable by the shell. These topics are covered in the sections that follow.

Accepting Arguments from Shell Scripts

Much like the similarly named C variables, the `ARGV` variable is an array of arguments passed to an AWK script, and the `ARGC` variable contains the number of arguments in `ARGV`. These variables are demonstrated in Listing 9-1.

Listing 9-1   Test script for arguments (`23_arguments.awk`)

```
{
    for (i=0; i<ARGC; i++) {
        print "ARGUMENT " i " is " ARGV[i];
    }
}
```

Save this script as `23_arguments.awk` and then issue the following commands:

```
echo > myinputfile
awk -f 23_arguments.awk myinputfile
```

You should see the following output:

```
ARGUMENT 0 is awk
ARGUMENT 1 is myinputfile
```

> **Note:** All arguments passed to AWK scripts must be the names of files that actually exist. This cannot be used for passing arbitrary data.

Reading Environment Variables

As in shell scripts, AWK scripts have access to environment variables. The AWK interpreter stores a copy of its environment in the `ENVIRON` associative array, indexed by the name of the variable.

> **Note:** It is not possible to set the environment passed to programs that an AWK script executes except by using the env tool as an intermediary.

For example, to print the value of the `PATH` environment variable, you would write code like the following:

```
{
    print "PATH IS: " ENVIRON["PATH"];
}
```

Extracting Output from AWK Scripts

When writing shell scripts, one of the trickiest things to get right is handling the output of tools that your scripts call. Fortunately, the tabular data format commonly used by AWK scripts is also easy to read in shell scripts. The UNIX command-line environment provides the `cut` tool, which is specifically designed to extract tabular data from lines of text.

Consider the following AWK script. It reads a file containing five tab-delimited data fields, then outputs three of those fields (also in a tab-delimited format).

```
BEGIN {
    RS="\n";
    FS="\t";
}
{
    print $1 "\t" $3 "\t" $5;
}
```

You can parse its output as shown in Listing 9-2.

Listing 9-2   Parsing the output of an AWK script

```
#!/bin/sh


# Store the output in a variable.


OUTPUT="$(awk 'BEGIN { \
        RS="\n"; \
        FS="\t"; \
    } \
    { \
        print $1 "\t" $3 "\t" $5; \
    }' tab_delimited_file)"


# Set the field separator to a newline so that
# the "for" statement below will put one line
# at a time in the "LINE" variable.
IFS="
"


# Parse and print the records.
RECORD=1
for LINE in $OUTPUT ; do
        # By default, cut uses tab as its delimiter,
        # so these commands take the first,
        # second, and third tab-delimited fields
        # from a single line of input, respectively.
        FIELD_1="$(echo "$LINE" | cut -f 1)"
        FIELD_2="$(echo "$LINE" | cut -f 2)"
        FIELD_3="$(echo "$LINE" | cut -f 3)"

        echo "RECORD $RECORD"
        echo "    FIELD 1: $FIELD_1"
        echo "    FIELD 2: $FIELD_2"
        echo "    FIELD 3: $FIELD_3"
        echo
```

```
        RECORD="$(expr $RECORD '+' 1)"
done
```

Another useful technique when dealing with complex result sets is to write different pieces of data to different files. Parsing several simple files can sometimes be easier than parsing a single complex result set, particularly when parsing it in a shell script.

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 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

OS X provides BASH as its Bourne shell implementation. When executed as `/bin/sh`, it should be fully compatible with other implementations. However, occasionally differences may arise. The same is true of other operating systems that use BASH or ZSH as their Bourne shell 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
```

You should use a similar first line for scripts written using ZSH extensions.

> **Compatibility Note:** For detailed lists of places where BASH and ZSH differ from pure Bourne shell variants, see http://www.gnu.org/software/bash/manual/bashref.html#Major-Differences-From-The-Bourne-Shell and http://zsh.dotsrc.org/FAQ/zshfaq02.html.
>
> For more information about BASH and ZSH, see the manual pages for `bash` and `zsh`.
>
> For maximum cross-platform compatibility, you should test your code using several shells, including `dash` and/or `ash`. For more information about DASH, see http://gondor.apana.org.au/~herbert/dash/.

Cross-Platform Line Endings

Different operating systems use different characters to indicate the end of each line in text files. This can cause strange and unusual behavior if you aren't expecting it:

- Command-line tools in OS X (and other UNIX or Linux variants) use UNIX-style line endings. This means that each line in a text file ends with a newline character (character `10/0xA`, often abbreviated LF).

- Many older Mac applications use "Mac-style" line endings. This means that each line in a text file ends with a carriage return character (character `13/0xD`, often abbreviated CR).

  When processed with command-line utilities in UNIX or Linux variants, files with legacy Mac-style line endings show up as a single line on the screen; as each line printed to the screen, it overwrites the previous line. This is because UNIX and Linux move the cursor to the left edge of the screen when they encounter a carriage return, but do not move the cursor down a line.

- Windows applications and many network services use Windows-style line endings. This means that each line in a text file ends with both a carriage return and a line feed (character `13/0xD` followed by character `10/0xA`, often abbreviated CR/LF or CRLF).

  When processed with command-line utilities in UNIX or Linux variants, content with Windows-style line endings looks right, but may behave in unexpected ways due to the extra carriage return at the end of each line. For example, the extra carriage return can perturb the splitting behavior in awk, can cause patterns that use the end-of-line anchor in regular expressions to fail, and so on.

- Occasionally, you may also encounter a file that ends with a newline followed by a carriage return (the reverse of Windows line endings, abbreviated LF/CR or LFCR).

  When processed with command-line utilities in UNIX or Linux variants, as with Windows-style line endings, everything will appear right, but you will get strange behavior, including field splitting problems, misbehavior of patterns containing the start-of-line anchor in regular expressions, and so on.

It is generally straightforward to detect the line ending type of a text file and read it correctly. The following code snippet demonstrates one way:

Listing 10-1 Converting line endings to UNIX-style newlines

```
TYPE="$(file "$1" | sed 's/.*with //' | sed 's/ .*//')"


if [ "$TYPE" = "CR" ] ; then

        DATA="$(tr '\r' '\n' < "$1")"

else

        # Most versions of the "file" command can't detect

        # LFCR line endings, so do this even if the file

        # appears to have UNIX line endings.

        DATA="$(tr -d '\r' < "$1")"

fi
```

Converting between these formats is also relatively easy once you have determined that you need to do so.

Listing 10-2 Converting between line ending formats

```
# Convert from legacy Mac-style CR line endings

# to UNIX-style LF line endings for use with

# command-line tools

tr '\r' '\n' < mac_text_file > unix_text_file


# Convert from UNIX-style LF to legacy Mac-style CR

# line endings

tr '\n' '\r' < unix_text_file > mac_text_file


# Convert from Windows-style CR/LF line endings (or

# LF/CR line endings) to UNIX line endings

tr -d '\r' < windows_text_file > unix_text_file


# Convert from UNIX-style LF line endings to

# Windows-style CR/LF line endings

CR="$(printf "\r")"

sed "s/$/$CR/" < unix_text_file > windows_text_file
```

Working with Device I/O

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 OS X.

See *IOKit 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.

File System Hierarchy

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

- `hier`—The OS X manual page hier(7) describes the 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://h20000.www2.hp.com/bc/docs/support/SupportManual/c02255645/c02255645.pdf—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.

System Administration Tasks

This section provides an overview of a few common system administration tasks. Complete coverage of system administration tasks is beyond the scope of this document. For a more thorough treatment, read *Introduction to Command-Line Administration* at http://manuals.info.apple.com/en_US/IntroCommandLine_v10.6.pdf.

Managing Users and Groups

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

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 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 46).

To learn more about managing users and groups from the command line, read *Introduction to Command-Line Administration* at http://manuals.info.apple.com/en_US/IntroCommandLine_v10.6.pdf.

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`.

To see how to manually add a new user from the command line, read the Additional Features chapter of *Porting UNIX/Linux Applications to OS X*. For scripts to help you add new users and groups programmatically, see User and Group Management (page 312) in the Starting Points (page 273) chapter of this document.

Access Control List (ACL) Management

Some UNIX-based and UNIX-like operating systems provide `setfacl`, `chacl`, or `acledit`/`aclget`/`aclput` for setting file and directory ACLs. OS X does not. Instead, OS X provides file ACL modification through the `chmod` command.

Regrettably, there is no standardized syntax for getting and setting ACLs on the command line (nor even a standard set of supported rights across operating systems). Currently, the only way to portably handle ACLs is to avoid them entirely or to require your users to write an OS-specific plug-in.

If you must use ACLs in a cross-platform script, you must special-case the code on a per-OS basis. The easiest way to do this is to use the output of the `uname` command. (See the `uname` manual page for more information.)

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 OS X command-line tools for disk management and partitioning, read *Introduction to Command-Line Administration* at http://manuals.info.apple.com/en_US/IntroCommandLine_v10.6.pdf, and see section 8 of *OS X Man Pages*. In particular, you should look at the man pages for `hdiutil`, `pdisk`, `fdisk`, `gpt`, and `diskutil`.

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 release 2.

  Because of licensing restrictions on the original UNIX source code, the portions that were originally written by AT&T had to be rewritten 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 (GNU Hurd), 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.

Over the 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). As operating systems work towards compliance with these specifications, 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.

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

> **Note:** While tools in OS X v10.5 and later generally obey AT&T semantics, this is not always true. In particular, when executed from installer scripts or startup items, they obey BSD semantics for backwards compatibility with existing scripts.
>
> As a convenience to script developers, you can also obtain legacy behavior from most command-line tools by setting certain environment variables as described in the `compat` manual page.
>
> For more information on legacy-mode command support, see *Unix 03 Conformance Release Notes*, the `compat` manual page, and the manual pages for individual commands.

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 a particular `awk` interpreter 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 to fully use the extended regular expression syntax.

To learn about basic and extended regular expressions, read Regular Expressions Unfettered (page 100). To learn more about the `awk` interpreter, read the manual page for `awk`. To learn more about the AWK scripting language, read How AWK-ward (page 122).

chown

If you pass the −P flag to `chown`, it does not follow symbolic links. Thus, the file that a symbolic link points to 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 84).

crontab

In AT&T-based UNIX systems, the `crontab` command reads from standard input by default, but on BSD-based systems, it does not. For cross-platform compatibility, you should specify a hyphen (−) for the filename instead. This works on with versions of `crontab` that obey both AT&T and BSD semantics.

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.

df

The `df` command has two different meanings for the `–t` flag beginning in OS X v10.5. They are as follows:

- If you include a value afterwards (for example, `–t hfs`), it behaves like the `–T` flag. This usage is deprecated.

- Without an argument, it tells `df` to print the total allocated space. Because this option is the default, this use of the `–t` flag is unnecessary.

The default block size varies on different operating systems. Linux and most BSD-based operating systems default to a 1k block size, while AT&T UNIX-based operating systems default to a 512-byte block size.

For consistent behavior across multiple operating systems, you should always specify a block size explicitly. For example, the -k flag specifies that the block size should be reported in kilobytes.

Finally, the capacity percentage reported by `df` may be rounded differently in different operating systems.

dos2unix and unix2dos

Linux provides these two utilities for converting between UNIX-style and DOS-style line endings. Using these tools is not portable, and OS X does not provide these utilities.

Instead of using `dos2unix` or `unix2dos`, you should instead use `tr` or `sed` as described in Cross-Platform Line Endings (page 147).

du

Operating systems that follow AT&T semantics allow you to pass a combination of the `–L`, `–H`, and `–P` options to `du`. The last flag encountered determines the command's behavior. In operating systems that follow BSD semantics, specifying more than one of these options results in an error. To fix this problem, delete all but the last of these options.

Also, many BSD-based operating systems cannot detect symbolic link loops. For cross-platform compatibility, you should generally not tell `du` to follow symbolic links unless you are certain that no cycles can occur.

echo

Of particular interest is the difference in behavior of the `echo` builtin and the corresponding standalone 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 `printf` or `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. Such an expression 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.

head

The `head` command exists across most operating systems. However, different versions provide several flags that are nonstandard.

The only flag that can be used portably is the `-n` flag, which takes a line count.

Most operating systems (including OS X) also support the `-c` flag, which specifies a byte count, but this support is not guaranteed to be portable. It is possible to emulate this functionality portably with the help of an AWK script, however, as follows:

Listing 10-3   Emulating head -c using AWK: 01_head_c.sh

```
#!/bin/sh

# Usage: ./head_c filename bytecount
FILENAME=$1
COUNT=$2

SCRIPT="$(mktemp '/tmp/head_c.XXXXXXXXXX')"

cat << EOF > "$SCRIPT"
BEGIN {
        FS="";
        my_string = ""
```

```
}
{
        my_string = my_string "\n" \$0;
}
END {
        # Start from character 2 to skip the bogus leading newline.
        print substr(my_string, 2, $COUNT);
}
EOF


awk -f "$SCRIPT" "$FILENAME"


rm "$SCRIPT"
```

You may also run into a minor compatibility problem when porting scripts from Linux to OS X. When you pass multiple filenames to the head command, it prints a heading line for each file name in the form

```
==> filename <==
```

The Linux version of head provides a –q flag that disables printing the header marker even if you specify multiple files. It also provides a –v flag that forces header printing even when only one file is specified.

As an alternative to the –v flag, you can output the filename marker in your script with a simple echo statement like this one:

```
echo "==> $FILENAME <=="
```

As an alternative to the –q flag, provided that there is no possibility of your files' contents actually matching the pattern, you can strip out the markers with grep like this:

```
head -n 1 file1 file2 ... | grep -v '^==>.*<==$'
```

In addition to these flag differences, POSIX specifies that the input files for head must be valid text files, which means that all byte sequences must be valid for the current locale. Although not all versions of head enforce this restriction, versions that do may fail when used with binary files in some operating systems unless you change the local settings.

If your scripts must process binary files, be sure to specify the "C" locale before executing commands that work with these binary files. To change the locale, issue the following statement:

```
export LANG="C"
```

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.

less

See .

ls

When −H is specified (and is not overridden by −L or −P) and a file argument is a symbolic link that resolves to a non-directory file, the output reflects the nature of the link, rather than that of the file. In operating systems that follow BSD semantics, the output describes the file.

The −f option turns on the −a option (show files whose names have a period (`.`) as the first character). In operating systems that follow BSD semantics, it does not.

The −o option causes the listing to be in long format, but to omit the group id. In operating systems that follow BSD semantics, the −o option modifies the −l option, causing file flags to be listed.

The −g, −n, and −o options turn on the −l option (causing the listing to be in long format). In operating systems that follow BSD semantics, they do not.

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.

more or less

Different operating systems handle the −n and −p flags to the `more` command differently.

In operating systems that follow the BSD and AT&T semantics, the −n option specifies the number of lines per screen, and the −p option allows you to specify commands (such as :p) to execute each time a new screenful of text is displayed.

In operating systems that follow Linux semantics (and for the less command on all operating systems), the −n flag tells the more command to to suppress line numbering, and the −p flag specifies a search pattern.

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.

Flag	AT&T	BSD
–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.

rename

The `rename` command is a command that exists on some Linux distributions. To add further confusion, there are two separate commands that have this name, depending on the distribution, and the syntax for the two commands is completely different:

- In some Linux distributions, `rename` is a command from the util-linux-ng package, found at http://user-web.kernel.org/~kzak/util-linux-ng/.

- In other Linux distributions, `rename` is a Perl script, also known in various incarnations as `prename` or `perl–rename` that ships as part of the Perl distribution. This script is available from CPAN.

Because the use of the `rename` tool is not portable even across Linux distributions, you should generally use the `find` command, if possible.

If `find` is insufficient, you can easily install the Perl `rename` command using the `cpan` tool. To do this, first log in in as an admin user, then run Terminal, then type:

```
sudo cpan File::Rename
```

The `sudo` command then asks you to enter your admin password.

Once the `File::Rename` CPAN package is installed, the rename command is in `/usr/local/bin`.

Be sure to document this nonstandard dependency appropriately in your script, along with an explanation of how to install the module.

sed

Different versions of `sed` use different flags for enabling extended regular expressions. GNU `sed` (commonly used in Linux) uses the `−r` flag. BSD versions of `sed` (including the OS X version) use the `−E` flag. If your script must run on both platforms, you must test for compatibility first. For example:

```
STRING="$(echo 'xy' | sed −E 's/(x)y/\1/' 2> /dev/null)"
if [ "$STRING" = "x" ] ; then
    SEDERE="−E"
else
    SEDERE="−r"
fi
...
sed $SEDERE ...
```

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

Also, 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 and is not supported in OS X beginning in version 10.5.

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).

> **Compatibility Note:** OS X v10.5 and later does not support this legacy GNU sort syntax. However, as a temporary workaround while you rewrite the offending scripts, you can set the _POSIX2_VERSION environment variable as show in the following snippet:
>
> ```
> export _POSIX2_VERSION=200111
> # or in CSH
> setenv _POSIX2_VERSION 200111
> ```
>
> Do not rely on this workaround for production code; its continued support is not guaranteed.

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

stty

Prior to OS X v10.5, the stty command did 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, prior to OS X v10.5, `stty` did not support the following options:

- `ocrnl` and `–ocrnl`

- `ofdel` and `–ofdel`

- `ofill` and `–ofill`

- `onlret` and `–onlret`

- `onocr` and `–onocr`

In legacy mode, these modes and options are still not accepted. For more information, see the manual page for `stty`.

tail

The `tail` command differs significantly between Linux and OS X. The GNU variant of `tail` provides options that the OS X version does not and vice versa. Both provide features that are not part of the POSIX specification, and thus may not be portable.

According to the POSIX specification, the following flags are portable: `–f` (continue to wait for the file to grow or for the FIFO to provide additional data), `–c` (byte count), and `–n` (line count).

Further, POSIX only explicitly requires the `tail` command to accept a single filename as an argument. Any use with multiple files is inherently not portable.

`–b` (OS X)
    OS X provides a `–b` flag that allows you to specify a location in 512-byte block increments. For maximum portability, multiply the number by 512 yourself and use the `–c` flag instead.

`–F` (OS X and Linux)
    Both Linux and OS X provide a `–F` flag that is equivalent to `–f --retry`. This is easily avoided with the workarounds described as part of the entries for the individual `--follow` and `--retry` flags.

`--follow` (Linux)

Linux also provides a `--follow` flag, which is equivalent to `-f` except when used with file descriptors.

When working with files, use `-f` instead.

The file descriptor syntax is not portable and is not supported except in Linux. Use a named pipe (FIFO) instead.

`--max-unchanged-stat` (Linux)

Linux provides a `--max-unchanged-stat` that tries reopening a file if you are using the `-f` flag and the file hasn't changed in a while. This allows it to handle the case here the file is renamed and a new file with the same name is created as often happens with log files. There is no easy portable replacement for this feature.

`--pid` (Linux)

Linux provides a `--pid` flag that terminates the `tail` command after the specified process ID dies.

There is no easy portable replacement for this feature, though it could be replaced in a not-so-portable fashion by a script running as a background job that uses the `ps` command to verify the existence of the process.

Assuming the process being watched was originally started by the shell script in the background, it could also be replaced by running the `tail` command in the background and using the `wait` shell builtin to wait for the process ID to exit, then killing the `tail` command. For more information, see Background Jobs and Job Control (page 197).

`-q` (Linux)

As with the `head` command, Linux provides `-v` and `-q` flags. See head (page 156) earlier in this section for explanation of these flags and suggested alternatives.

`-r` (OS X)

OS X provides a `-r` flag that reverses the order of the lines printed. It also changes the behavior of the leading plus (+) and minus (−) symbols when passed as part of arguments to the `-b`, `-c`, and `-n` flags.

It is possible to write an AWK script to emulate this behavior by pushing each line in the input file into an array, then printing the lines in reverse order and either skipping a given number of entries in the array to skip lines or using `substr` call to skip a given number of bytes. The head (page 156) section of this chapter provides an example of how to emulate `head -c` using an AWK script; this example provides a good starting point for writing a script that emulates this `tail` feature.

`--retry` (Linux)

> Linux provides a `--retry` flag to keep trying to open a file if it does not exists.
>
> This is commonly used, with the `-f` flag, and in that usage, is equivalent to the `-F` flag, which OS X supports.
>
> By itself, however, OS X has no equivalent flag, though you can trivially approximate it in a more portable fashion by writing a while loop in a shell script that repeatedly checks for the file until it finds it, then runs the `tail` command.

`-s` and `--sleep-interval` (Linux)

> Linux provides `-s` and `--sleep-interval` flags to lower CPU use by adding a delay between checks to see if a file you are watching with `-f` has grown.

`-v` (Linux)

> As with the `head` command, Linux provides `-v` and `-q` flags. See head (page 156) earlier in this section for explanation of these flags and suggested alternatives.

In addition to these flag differences, POSIX specifies that the input files for `tail` must be valid text files, which means that all byte sequences must be valid for the current locale. Although not all versions of `tail` enforce this restriction, versions that do may fail when used with binary files in some operating systems unless you change the local settings.

If your scripts must process binary files, be sure to specify the "C" locale before executing commands that work with these binary files. To change the locale, issue the following statement:

```
export LANG="C"
```

Finally, unlike the `head` command, POSIX does not require that the `tail` command be able to store and print a text block of arbitrary length. It requires only that the buffer size be at least 10 times the value of `LINE_MAX`. The value of `LINE_MAX` is implementation dependent, but must be at least 2048 bytes.

While this theoretical 20,480 byte limit in the output of the `tail` command is not commonly enforced in modern operating systems, the only *guaranteed* portable way to generate larger results from `tail` is to use another tool such as AWK.

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.

which

In OS X, the `which` command can take the -s flag for "silent" behavior. In this mode, it does not output any text and returns an exit status of `0` if the command exists in any of the paths listed in the `PATH` environment variable or `1` if it does not (or `2` if you pass an invalid flag).

This flag does not exist in many operating systems that obey AT&T semantics. The GNU version of `which` used in Linux also does not support this flag. As an alternative, you can redirect the output of `which` to `/dev/null` as described in Pipes and Redirection (page 40).

Also, some (not all) Linux distributions come with the GNU `which` command. This command differs significantly in its behavior from other UNIX-like operating systems. In order to support searching for multiple commands in a single `which` statement, its exit status contains the number of commands that were not found, or -1 if you pass it unknown flags. (It also supports a number of formatting flags that are not broadly available.)

For reliable cross-platform use, you should specify exactly one command argument at a time, pass no flags (except the ubiquitous –a flag, if desired), and assume that an exit status of either –1 or 2 indicates a usage error.

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.

> **Compatibility Note:** You can get the BSD semantics in OS X v10.5 by enabling legacy mode as described in the `compat` manual page.

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.

- Using the eval Builtin for Data Structures, Arrays, and Indirection (page 168) describes how to create complex data structures in shell scripts.

- Shell Text Formatting (page 176) tells how to do tabular layouts and use ANSI escape sequences to add color and styles to your terminal output.

- Trapping Signals (page 173) tells how to write signal handlers in shell scripts.

- Nonblocking I/O (page 191) and Timing Loops (page 194) show one way to write complex interactive scripts such as games.

- Background Jobs and Job Control (page 197) explains how to do complex tasks in the background while your script continues to execute, including how to perform some basic parallel computation. It also explains how to obtain the result codes from these jobs after they exit.

- Application Scripting With osascript (page 204) describes how your script can interact with OS X applications using AppleScript.

- Scripting Interactive Tools Using File Descriptors (page 211) describes how you can make bidirectional connections to command-line tools.

- Networking With Shell Scripts (page 216) describes how to use the `nc` tool (otherwise known as netcat) to write shell scripts that take advantage of TCP/IP sockets.

Using the eval Builtin for Data Structures, Arrays, and Indirection

One of the more under-appreciated commands in shell scripting is the `eval` builtin. The `eval` builtin 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` builtin is superfluous. However, the behavior of the `eval` builtin 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` builtin 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` builtin 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` builtin 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` builtin are included in the sections that follow.

A Complex Example: Setting and Printing Values of Arbitrary Variables

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` builtin. In real-world code, you should *never* pass unsanitized user input directly to `eval` because doing so can 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` builtin 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 66). The result is the same.

Thus, the statement:

```
eval echo "$"$VARIABLE
```

evaluates to:

```
echo $MYVAR
```

Note: If you forget to quote the first dollar sign, you 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 are interpreted as a variable, and thus the statement evaluates 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 23), 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` builtin 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` builtin 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:

```sh
#!/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).

A Powerful Example: Binary Search Trees

Working with Binary Search Trees (page 287) in Starting Points (page 273) provides a ready-to-use binary search tree library written as a Bourne shell script.

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 C 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 subroutine signal [ signal ... ]
```

The first argument is the name of a subroutine 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 subroutine `sigwinch_handler` whenever it receives a `SIGWINCH` signal. The script in Listing 11-1 prints the phrase "Window size changed." whenever you adjust the size of your terminal window.

Listing 11-1   Installing a signal handler trap

```
#!/bin/sh


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 subroutine name. For example, the code in Listing 11-2 ignores the "interrupt" signal generated when you press Control-C:

Listing 11-2   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 11-3 as `ipc1.sh` and the script in Listing 11-4 as `ipc2.sh`.

Listing 11-3   `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 11-4   `ipc2.sh`: Script interprocess communication example, part 2 of 2

```
#!/bin/sh


## Save this as ipc2.sh


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 subroutine, `hup_handler`. This subroutine prints the words "SIGHUP RECEIVED." and exits.

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 11-5   Columnar printing using `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 179).

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 11-6, which truncates a string to an arbitrary length by building up a regular expression.

Listing 11-6 Truncating text to column width

```
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 subroutine 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 183). (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 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 tries 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 11-1 (page 185)) 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 186)) 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 186)) allow you to change the foreground and background color of text.

- Other escape codes (described in Table 11-4 (page 190)) 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 shells.

Compatibility Note: In BASH, the LINES and COLUMNS 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 OS X where the default shell is BASH (OS X v10.3 and newer), these variables are not defined in shell scripts that start with #!/bin/sh.

Of course, you could request that ZSH interpret the script by changing the first line of your script to #!/bin/zsh, but doing so is not particularly portable. Fortunately, without changing shells, you can easily obtain the current row and column count with the code in Listing 11-7.

Listing 11-7 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 173) 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 11-1. 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 182).

Table 11-1   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` *r c*	`^[[`*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.

Terminfo capability	Escape sequence	Description
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 11-8 prints a familiar phrase in multiple colors.

Listing 11-8   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 11-2 contains a list of capabilities and escape sequences that control text style.

Table 11-2   Attribute escape sequences

Terminfo capability	Escape sequence	Description
Resetting attributes		
me	^[[m or ^[[0m	Resets all attributes to their default values.
Setting attributes		

Terminfo capability	Escape sequence	Description
`bold`	`^[[1m`	Enables "bold" display. 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 attribute 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.
`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.

Terminfo capability	Escape sequence	Description
`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.
	`^[[29m`	Unused.

Table 11-3 contains a list of capabilities and escape sequences that control text and background colors.

Table 11-3   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.

Terminfo capability	Escape sequence	Description
`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 11-9.

Listing 11-9 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 11-4 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 11-4   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 concatenation 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.
(no equivalent)	`^[[2K`	Clears the current line.
Tab stops		

Terminfo capability	Escape sequence	Description
`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/ansi-code.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 OS X) does not support the complete set of ANSI escape sequences or terminfo capabilities.

Nonblocking I/O

Most shell scripts do not need to accept user input at all during execution, and scripts that do require user input can generally request it a line at a time. However, if you are writing a shell script that needs to interact with the user while performing background activity, it can be convenient to simulate 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:

```c
#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:

```bash
#!/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. For example, you might write a game of ping pong that checks for a keystroke at the beginning of each ball drawing loop and if it detects one, moves the user's paddle by a few pixels.

This script also illustrates another useful technique: disabling input buffering. The `stty` command changes three 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 receives an empty line. With canonicalization disabled, your application instead sees 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 173).

- The `−cbreak` flag sets some reasonable defaults for interactive shell use.

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. If you are capturing characters for a full-screen game, for example, echoing the typed characters to the screen tends to be disastrous, depending on how unlucky the user's timing is when pressing the key.

Depending on what other flags you pass, you may want to reset the terminal more fully at the end by issuing the command `stty sane`. In OS X, this flag is identical to `−cbreak`, but in Linux and some other operating systems, the `sane` flag is a superset of the `−cbreak` flag.

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 11-10 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 11-10   A simple one-second timing loop

```
#!/bin/sh


ONE_SECOND=1000


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
}


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.

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

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 script 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. It shows two different techniques for waiting for jobs. You should generally use the process ID when waiting for a child process. You can obtain the process ID of the last command using the $! shell variable.

If, however, you need to inspect a job using the jobs builtin, you must use the job ID. It can be somewhat clumsy to obtain a job ID because the job control mechanism in most Bourne shell variants was designed primarily for interactive use rather than programmatic use. Fortunately, there are few things that a well-written regular expression can't fix.

> **Note:** Regular expressions are described in Regular Expressions Unfettered (page 100). For the purposes of this example, it is sufficient to understand that the subroutine jobidfromstring takes a job string like the one shown previously and prints out the first single digit or multidigit number by itself.

```
#!/bin/bash


jobidfromstring()
{
        local STRING;
        local RET;

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

```
        echo $RET;
}


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


# Use the job ID for this one.
delayprint 3 &
DP3=`jobidfromstring $(jobs %%)`


# Use the process ID this time.
delayprint 5 &
DP5=$!


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


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


echo "Waiting for process ID $DP5";
# No percent because it is a process ID
wait $DP5


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

```
echo "Done."
```

This example passes a job number or process ID 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 the case, however, a second percent sign is used. The `%%` job is one of a number of special job "numbers" that the shell provides. It tells the `jobs` builtin 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 string is passed as a series of arguments to the `jobidfromstring` subroutine, which then prints the job ID by itself. The output of this subroutine, in turn, is stored into either the variable DP3 or DP7.

This example also demonstrates how to wait for a job based on process ID using a special shell variable, `$!`, which contains the process ID of the last command executed. This value is stored into the variable DP5. Process IDs are generally preferred over job IDs when using the `jobs` command in scripts (as opposed to hand-entered use of the `jobs` command).

Finally, the script ends with a series of calls to the `wait` builtin. These commands tell the shell to wait for a child process to exit. When a child process exits, the shell reaps the process, stores its exit status in the `$?` variable, and returns control to the script..

Like the `jobs` command, the `wait` builtin can take a job ID or process ID. If you specify a job or process ID, the shell does not return control to the script until the specified job or process exits. If no process or job ID is specified, the `wait` builtin returns as soon as the first child exits.

A job ID consists of a percent sign followed by the job number (obtained from either the variable DP3 or DP7). A process ID is just the number itself.

> **C Shell Note:** The C shell does not allow you to query the last job or wait for a single job or process ID. You can, however, wait for all outstanding jobs to finish by running the `wait` builtin with no arguments.

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


spawnjob()
```

```
{
    echo $1 | bash
}

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 $COMMAND &
    LASTJOB=$!
    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 subroutine `clearToSpawn`, the `-r` flag must be passed to the `jobs` builtin to restrict output to currently running jobs. Without this flag, the `jobs` builtin would otherwise return a list that included completed jobs, thus making the count of running jobs incorrect.

> ⚠️ **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.

C Shell Note: A C shell version of this script is included in the accompanying Companion Files download. To obtain this archive, see the web version of this document at http://developer.apple.com/.

Application Scripting With osascript

OS X provides a powerful application scripting environment called AppleScript. With AppleScript, you can launch an application, tell a running application to perform various tasks, query a running application in various ways, and so on. Shell script programmers can harness this power through the `osascript` tool.

Note: Although this section describes use of `osascript` for executing AppleScript for application scripting, the `osascript` tool provides a command-line interface to any scripting language with an interpreter that conforms to the Open Scripting Architecture (OSA). For example, if you install the third-party JavaScript OSA freeware package, you can use `osascript` to execute JavaScript code.

The `osascript` tool executes a program in the specified language and prints the results via standard output. If no program file is specified, it reads the program from standard input.

The first example is fairly straightforward. It opens the file `poem.txt` in the directory above the directory where the script is located:

Listing 11-11   Opening a file using AppleScript and osascript: 07_osascript_simple.sh

```
#!/bin/sh


POEM="$PWD/../poem.txt"


cat << EOF | osascript -l AppleScript
launch application "TextEdit"
tell application "TextEdit"
        open "$POEM"
end tell
EOF
```

You should notice that the path to the file `poem.txt` is specified as an absolute path here. This is crucial when working with `osascript`. Because the current working directory of a launched application is always the root of the file system (the / directory) rather than the shell script's working directory, a script must pass an absolute path to AppleScript rather than a path relative to the script's working directory.

The next example shows how to query an application. In this case, it launches TextEdit, opens two files, asks TextEdit for a list of open documents, and uses that list to help it ask TextEdit to return the first paragraph of text in the document that corresponds with the `poem.txt` file.

Listing 11-12   Working with a file using AppleScript and osascript: 08_osascript_para.sh

```
#!/bin/sh


# Get an absolute path for the poem.txt file.
POEM="$PWD/../poem.txt"


# Get an absolute path for the script file.
SCRIPT="$(which $0)"
if [ "x$(echo $SCRIPT | grep '^\/')" = "x" ] ; then
    SCRIPT="$PWD/$SCRIPT"
fi


# Launch TextEdit and open both the poem and script files.
cat << EOF | osascript -l AppleScript > /dev/null
launch application "TextEdit"
tell application "TextEdit"
    open "$POEM"
end tell


set myDocument to result
return number of myDocument
EOF


cat << EOF | osascript -l AppleScript > /dev/null
launch application "TextEdit"
tell application "TextEdit"
        open "$SCRIPT"
```

```
end tell

set myDocument to result
return number of myDocument
EOF



# Tell the shell not to mangle newline characters, tabs, or whitespace.
IFS=""


# Ask TextEdit for a list of open documents.  From this, we can
# obtain a document number that corresponds with the poem.txt file.
# This query returns a newline-deliminted list of open files. Each
# line contains the file number, followed by a tab, followed by the
# filename
DOCUMENTS="$(cat << EOF | osascript -l AppleScript


    tell application "TextEdit"
        documents
    end tell


    set myList to result        -- Store the result of "documents" message into
variable "myList"
   set myCount to count myList  -- Store the number of items in myList into myCount
   set myRet to ""              -- Create an empty string variable called "myRet"


   (* Loop through the myList array and build up a string in the myRet variable
      containing one line per entry in the form:

      number tab_character name
    *)
   repeat with myPos from 1 to myCount
       set myRet to myRet & myPos & "\t" & name of item myPos of myList & "\n"
   end repeat
   return myRet
```

```
EOF

)"


# Determine the document number that corresponds with the poem.txt

# file.

DOCNUMBER="$(echo $DOCUMENTS | grep '[[:space:]]poem\.txt' | grep -v ' poem\.txt'
 | head -n 1 | sed 's/\([0-9][0-9]*.\).*/\1/')"

SECOND_DOCNUMBER="$(echo $DOCUMENTS | grep '[[:space:]]poem\.txt' | grep -v '
poem\.txt' | tail -n 1 | sed 's/\([0-9][0-9]*.\).*/\1/')"


if [ $DOCNUMBER -ne $SECOND_DOCNUMBER ] ; then

    echo "WARNING: You have more than one file named poem.txt open.  Using the"
1>&2

    echo "most recently opened file." 1>&2

    echo "DOCNUMBER $DOCNUMBER != $SECOND_DOCNUMBER"

fi


echo "DOCNUMBER: $DOCNUMBER"


if [ "x$DOCNUMBER" != "x" ] ; then

    # Query poem.txt by number

    FIRSTPARAGRAPH="$(cat << EOF | osascript -l AppleScript

        tell application "TextEdit"

            paragraph 1 of document $DOCNUMBER

        end tell

EOF

    )"

    echo "The first paragraph of poem.txt is:"

    echo "$FIRSTPARAGRAPH"

fi


# Query poem.txt by name

FIRSTPARAGRAPH="$(cat << EOF | osascript -l AppleScript

        tell application "TextEdit"

            paragraph 1 of document "poem.txt"

        end tell
```

```
EOF
)"
echo "The first paragraph of poem.txt is:"
echo "$FIRSTPARAGRAPH"
```

This script illustrates three very important concepts.

- It shows how to refer to a document by number and how to iterate through a list of documents, associating the name with a particular document number.

- It demonstrates a limitation in AppleScript—specifically, that you cannot always uniquely identify a particular document with a given name if two open files have the same name. When writing scripts, you should carefully avoid opening two files with the same name using the same application.

- It demonstrates how to reference a document by its name. The results from the `documents` message are transient; document numbers change as new windows are opened and old windows are closed. Thus, you should generally address documents using their names rather than using document numbers unless you are very careful.

The final example shows how to manipulate images using shell scripts and AppleScript. It scales the image to be as close to 320x480 or 480x320 (depending on the orientation of the image) as possible.

Listing 11-13   Resizing an image using Image Events and osascript: 09_osascript_images.sh

```
#!/bin/sh

# Get an absolute path for the poem.txt file.

MAXLONG=480
MAXSHORT=320

URL="http://images.apple.com/macpro/images/design_smartdesign_hero20080108.png"
FILE="$PWD/my design_smartdesign_hero20080108.png"
OUTFILE="$PWD/my design_smartdesign_hero20080108-mini.png"

if [ ! -f "$FILE" ] ; then
    curl "$URL" > "$FILE"
fi
```

```
# Tell the shell not to mangle newline characters, tabs, or whitespace.
IFS=""


# Obtain image information
DIM="$(cat << EOF | osascript -l AppleScript
tell application "Image Events"
    launch
    set this_image to open "$FILE"
    copy dimensions of this_image to {W, H}
    close this_image
end tell
return W & H
EOF
)"


W="$(echo "$DIM" | sed 's/ *, *.*//' )"
H="$(echo "$DIM" | sed 's/.* *, *//' )"


echo WIDTH: $W HEIGHT: $H


if [ $W -gt $H ] ; then
    LONG=$W
    SHORT=$H
else
    LONG=$H
    SHORT=$W
fi


# echo "LONG: $LONG SHORT: $SHORT"
# echo "MAXLONG: $MAXLONG MAXSHORT: $MAXSHORT"


NEWLONG=$LONG
NEWSHORT=$SHORT
# NEWSCALE=1
```

```
if [ $NEWLONG -gt $MAXLONG ] ; then
    # Long direction is too big.
    NEWLONG="$(echo "scale=20; $LONG * ($MAXLONG/$LONG)" | bc | sed 's/\..*//')";
    NEWSHORT="$(echo "scale=20; $SHORT * ($MAXLONG/$LONG)" | bc | sed 's/\..*//')";
    NEWSCALE="$(echo "scale=20; ($MAXLONG/$LONG)" | bc)";
fi


# echo "PART 1: NEWLONG: $NEWLONG NEWSHORT: $NEWSHORT"


if [ $NEWSHORT -gt $MAXSHORT ] ; then
    # Short direction is till too big.
    NEWLONG="$(echo "scale=20; $LONG * ($MAXSHORT/$SHORT)" | bc | sed 's/\..*//')";
    NEWSHORT="$(echo "scale=20; $SHORT * ($MAXSHORT/$SHORT)" | bc | sed 's/\..*//')";
    NEWSCALE="$(echo "scale=20; ($MAXSHORT/$SHORT)" | bc)";
fi


# echo "PART 2: NEWLONG: $NEWLONG NEWSHORT: $NEWSHORT"


if [ $W -gt $H ] ; then
    NEWWIDTH=$NEWLONG
    NEWHEIGHT=$NEWSHORT
else
    NEWHEIGHT=$NEWLONG
    NEWWIDTH=$NEWSHORT
fi


echo "DESIRED WIDTH: $NEWWIDTH NEW HEIGHT: $NEWHEIGHT (SCALE IS $NEWSCALE)"


cp "$FILE" "$OUTFILE"


DIM="$(cat << EOF | osascript -l AppleScript
tell application "Image Events"
    launch
```

```
    set this_image to open "$OUTFILE"
    scale this_image by factor $NEWSCALE
    save this_image with icon
    copy dimensions of this_image to {W, H}
    close this_image
end tell
return W & H
EOF
)"


GOTW="$(echo "$DIM" | sed 's/ *, *.*//' )"
GOTH="$(echo "$DIM" | sed 's/.* *, *//' )"


echo "NEW WIDTH: $GOTW NEW HEIGHT: $GOTH"
```

Of course, you could just as easily perform these calculations in AppleScript itself, but this demonstrates how easy it is for shell scripts to exchange information with AppleScript code, manipulate image files, and tell applications to perform other complex tasks.

For more information about manipulating images with Image Events, see http://www.apple.com/applescript/imageevents/. You can also find many other AppleScript examples at http://www.apple.com/applescript/examples.html.

Scripting Interactive Tools Using File Descriptors

Most of the time, you should use `expect` scripts or C programs to control interactive tools. However, it is sometimes possible, albeit sometimes difficult, to script such interactive tools (if their output is line-based). This section explains the techniques you use.

> **C Shell Note:** The lack of file descriptor redirection is one of the more serious flaws in the C shell. The techniques described in this section are not possible in C shell or its variants.

Creating Named Pipes

Before you can communicate with a tool in a continuous round-trip fashion, you must create a pair of FIFOs (short for first-in, first-out, otherwise known as named pipes) using the `mkfifo` command. For example, to create named pipes called `/tmp/infifo` and `/tmp/outfifo`, you would issue the following commands:

```
mkfifo /tmp/infifo
mkfifo /tmp/outfifo
```

To see this in action using the `sed` command as a filter, type the following commands:

```
mkfifo /tmp/outfifo
sed 's/a/b/' < /tmp/outfifo &
echo "This is a test" > /tmp/outfifo
```

Notice that `sed` exits after receiving the data and printing `This is b test` to the screen. The `echo` command opens the output FIFO, writes the data, and closes the FIFO. As soon as it closes the FIFO, the `sed` command gets a `SIGPIPE` signal and (usually) terminates. To use a command-line tool as a filter and keep passing data to it, you must make sure that you don't close the FIFO until you are finished using the filter. To achieve this, you must use file descriptors, as described in the next section.

Opening File Descriptors for Reading and Writing

As explained in Creating Named Pipes (page 212), sending data to a named pipe with command-line tools causes the command to terminate after the first message. To prevent this, you must open a file descriptor in the shell to provide continuous access to the named pipe.

You can open a file descriptor for writing to the output FIFO as follows:

```
exec 8> /tmp/outfifo
```

This command opens file descriptor 8 and redirects it to the file /tmp/outfifo.

> **Note:** You must choose a file descriptor number that is unused. Typically your script has three file descriptors open initially—descriptor 0 (standard input), descriptor 1 (standard output), and descriptor 2 (standard error). Just to be safe, this example uses descriptor 8.

Similarly, you can open a descriptor for reading like this:

```
exec 9<> /tmp/infifo
```

You can write data to an open descriptor like this:

```
# Write a string to descriptor 8
echo "This is a test." >&8
```

You can read a line from an open descriptor like this:

```
# Read a line from descriptor 9 and store the result in variable MYLINE
read MYLINE <&9
```

When you have finished writing data to the filter, you should close the pipes and delete the FIFO files as follows:

```
exec 8>&-
exec 9<&-
rm /tmp/infifo
rm /tmp/outfifo
```

Table 11-5 (page 213) summarizes the operations you can perform on file descriptors. The next section contains a complete working example.

Table 11-5   Shell file descriptor operators

Operator	Equivalent C code

n <> "filename"	fd = open("*filename*", O_RDWR|O_CREAT);
	dup2(fd, *n*);
	close(fd);

Operator	Equivalent C code

n> "filename"	`fd = open("`*filename*`", O_WRONLY|O_CREAT|O_TRUNC);`
	
	`dup2(fd, `*n*`);`
	
	`close(fd);`

n>> "filename"	`fd = open("`*filename*`", O_WRONLY|O_APPEND|O_CREAT);`
	
	`dup2(fd, `*n*`);`
	
	`close(fd);`

n<&*o* *n*>&*o*	`dup2(`*o*`, `*n*`);` **Note:** Although these operators behave identically, for readability, you should use the <& operator for read-only or read-write descriptors and the >& for write-only descriptors.
n<&− *n*<&−	`close(`*n*`);`

Using Named Pipes and File Descriptors to Create Circular Pipes

There's just one more problem. The `sed` command buffers its input by default. This can cause problems when using it as a filter. Thus, you must tell the `sed` command to not buffer its input by specifying the −l flag (or the −u flag for GNU `sed`).

The following listing demonstrates these techniques. It runs `sed`, then sends two strings to it, then reads back the two filtered strings, then sends a third string, then reads the third filtered string back, then closes the pipes.

Listing 11-14   Using FIFOs to create circular pipes

```
#!/bin/sh


# Create two FIFOs (named pipes)
INFIFO="/tmp/infifo.$$"
OUTFIFO="/tmp/outfifo.$$"
mkfifo "$INFIFO"
mkfifo "$OUTFIFO"


# OS X and recent *BSD sed uses −l for line−buffered mode.
```

```
BUFFER_FLAG="-l"

# GNU sed uses -u for "unbuffered" mode (really line-buffered).
if [ "x$(sed --version 2>&1 | grep GNU)" != "x" ] ; then
    BUFFER_FLAG="-u"
fi

# Set up a sed substitution input from the input fifo otput to
sed $BUFFER_FLAG 's/a test/not a test/' < $INFIFO > $OUTFIFO &
PID=$!

# Open a file descriptor (#8) to write to the input FIFO
exec 8> $INFIFO

# Open a file descriptor (#9) to read from the output FIFO.
exec 9<> $OUTFIFO

# Send two lines of text to the running copy of sed.
echo "This is a test." >&8
echo "This is maybe a test." >&8

# Read the first two lines from sed's output.
read A <&9
echo "Result 1: $A"
read A <&9
echo "Result 2: $A"

# Send another line of text to the running copy of sed.
echo "This is also a test." >&8

# Read it back.
read A <&9
echo "Result 3: $A"

# Show that sed is still running.
```

```
ps -p $PID


# Close the pipes to terminate sed.
exec 8>&-
exec 9<&-


# Show that sed is no longer running.
ps -p $PID


# Clean up the FIFO files in /tmp
rm "$INFIFO"
rm "$OUTFIFO"
```

Networking With Shell Scripts

By building on the concepts in Using Named Pipes and File Descriptors to Create Circular Pipes (page 214), you can easily write scripts that communicate over the Internet using TCP/IP using the netcat utility, nc. This utility is commonly available in various forms on different platforms, and the available flags vary somewhat from platform to platform.

The following listing shows how to write a very simple daemon based on netcat that works portably. It listens on port 4242. When a client connects, it reads a line of text, then sends the client the same line, only backwards. It repeats this process until the client closes the connection.

Listing 11-15   A simple daemon based on netcat

```
#!/bin/sh


INFIFO="/tmp/infifo.$$"
OUTFIFO="/tmp/outfifo.$$"


# /*! Cleans up the FIFOs and kills the netcat helper. */
cleanup_daemon()
{
```

```
    rm -f "$INFIFO" "$OUTFIFO"


    if [ "$NCPID" != "" ] ; then
        kill -TERM "$NCPID"
    fi


    exit
}


# /*! @abstract Attempts to reconnect after a sigpipe. */
reconnect()
{
        PSOUT="$(ps -p $NCPID | tail -n +2 | tr -d '\n')"
        if [ "$PSOUT" = "" ] ; then
                cleanup_shttpd
        fi
        closeConnection 8 "$INFIFO"
}


trap cleanup_daemon SIGHUP

trap cleanup_daemon SIGTERM

trap reconnect SIGPIPE

trap cleanup_daemon SIGABRT

trap cleanup_daemon SIGTSTP

# trap cleanup_daemon SIGCHLD

trap cleanup_daemon SIGSEGV

trap cleanup_daemon SIGBUS

trap cleanup_daemon SIGQUIT

trap cleanup_daemon SIGINT


mkfifo "$INFIFO"

mkfifo "$OUTFIFO"


# /*! Reverses a string. */
reverseit()
```

```
{
    STRING="$1"

    REPLY=""

    while [ "$STRING" != "" ] ; do
        FIRST="$(echo "$STRING" | cut -c '1')"
        STRING="$(echo "$STRING" | cut -c '2-')"
        REPLY="$FIRST$REPLY"
    done

    echo "$REPLY"
}

while true ; do
    CONNECTED=1
    nc -l 4242 < $INFIFO > $OUTFIFO &
    NCPID=$!

    exec 8> $INFIFO
    exec 9<> $OUTFIFO

    while [ $CONNECTED = 1 ]  ; do
            read -u9 -t1 REQUEST

        if [ $? = 0 ] ; then
            # Read didn't time out.
            reverseit "$REQUEST" >&8
            echo "GOT REQUEST $REQUEST"
        fi

        CONNECTED="$(jobs -r | grep -c .)"
    done
done
```

This daemon is designed to be portable, which limits the flags it can use. As a result, it can only handle a single client at any given time, with a minimum of a one second period between connection attempts. This is the easiest way to use the netcat utility. For a more complex example, see A Shell-Based Web Server (page 284).

You can also use netcat as a networking client in much the same way. You might send a request to a web server, a mail server, or other daemon. Of course, you are generally better off using existing clients such as `curl` or `sendmail`, but when that is not possible, netcat provides a solution.

The following listing connects to the daemon shown in Listing 11-15 (page 216), requests input from the user, sends the input to the remote daemon, reads the result, and prints it to standard output.

Listing 11-16   A simple client based on netcat

```
#!/bin/sh


INFIFO="/tmp/infifo.$$"
OUTFIFO="/tmp/outfifo.$$"


INFIFO="/tmp/infifo.$$"
OUTFIFO="/tmp/outfifo.$$"


# /*! Cleans up the FIFOs and kills the netcat helper. */
cleanup_client()
{
    rm -f "$INFIFO" "$OUTFIFO"

    if [ "$NCPID" != "" ] ; then
        kill -TERM "$NCPID"
    fi

    exit
}


# /*! @abstract Attempts to reconnect after a sigpipe. */
reconnect()
```

```
{
        PSOUT="$(ps -p $NCPID | tail -n +2 | tr -d '\n')"
        if [ "$PSOUT" = "" ] ; then
                cleanup_shttpd
        fi
        closeConnection 8 "$INFIFO"
}

trap cleanup_client SIGHUP
trap cleanup_client SIGTERM
trap reconnect SIGPIPE
trap cleanup_client SIGABRT
trap cleanup_client SIGTSTP
trap cleanup_client SIGCHLD
trap cleanup_client SIGSEGV
trap cleanup_client SIGBUS
trap cleanup_client SIGQUIT
trap cleanup_client SIGINT

mkfifo "$INFIFO"
mkfifo "$OUTFIFO"

nc localhost 4242 < $INFIFO > $OUTFIFO &
NCPID=$!

exec 8> $INFIFO
exec 9<> $OUTFIFO

while true ; do
    printf "String to reverse -> "
        read STRING
    echo "$STRING" >&8
    read -u9 REVERSED
    echo "$REVERSED"
```

```
done
```

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 327), 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 the `perl` or `awk` interpreter 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:

```
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 [ $CONT = 1 ]; do
        # Copy the string so we know if we've stopped finding
        # nonmatching characters.
        OCTEMP="$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 179)).

Caching the strings of periods used in the splitting process increases initialization time slightly, but after that, the execution time of the search itself improves 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 12-1 contains the binary search version.

Listing 12-1   A binary search version of the Bourne shell `ord` subroutine

```
# Initialize the split strings.
# This block of code should be
# added to the end of 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

 # End of content to add to ord_init


split_str()
{
        STR="$1"
        NUM="$2"
        SPLIT="$(eval "echo \"\$SPLIT_$NUM\"")"
        LEFT="$(echo "$STR" | sed "s/^\\($SPLIT\\).*$/\\1/")"
        RIGHT="$(echo "$STR" | sed "s/^$SPLIT//")"
}
```

```
ord3()
{
    local CH="$1"

    OCCOPY="$ORDSTRING"

    FIRST=1;

    LAST=257


    ord3_sub "$CH" "$ORDSTRING" $FIRST $LAST
}



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 and often do easily outweigh improvements in algorithmic complexity.

Of course, using a Perl or AWK script to find the ordinal rank is much faster than any of these methods. The purpose of this example is to demonstrate methods for improving efficiency of similar operations, not to show the best way to find the ordinal rank of a character.

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:

```
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 subroutine by about 20% by concatenating the assignment statements into a single string and running `eval` only once, as show in the following example:

```
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 subroutine 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 the `eval` builtin 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 call to `eval` (as described in Reducing Use of the eval Builtin (page 227)), 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 subroutine 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 12-1.

Table 12-1   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 12-2 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 12-2   Performance (in seconds) comparisons of 1000 executions of various control statement sequences

	eval builtin executing multiple subroutines	series of if statements	if, then series of elif statements	casestatement
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 might 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 12-2) 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 subroutine 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 12-3   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 write a control statement using `/bin/[`, the script's performance degrades immensely, Fortunately, it is unlikely that anyone would ever do that accidentally.

Table 12-4   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 if your shell provides it (most do). 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 12-5   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 222), 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 12-6   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

Shell Script Security

Security is often overlooked when writing shell scripts. Many programmers ignore shell script security under the assumption that anything an attacker can do by attacking a script can be achieved more easily by simply executing the commands themselves. This is not true, however, when the script takes input from an untrusted third party:

- Shell scripts running as CGI scripts on a web server take input from the network.

- Shell scripts that read files and take actions based on their contents may take input from untrusted files.

- Shell scripts that perform web queries (with `curl`, for example) or other network requests may take input from untrusted servers or clients.

Further, most security problems are also correctness bugs even if someone is not trying to attack your code.

This chapter describes a few common mistakes in scripting, shows how these vulnerabilities can be exploited, and explains how to prevent these attacks in your scripts.

This chapter also describes how UNIX permissions and POSIX access control lists (ACLs) affect your scripts and how to manipulate those permissions and ACLs in your scripts.

Environment Attacks

Environment variable attacks are the most common way to manipulate script behavior. By manipulating the environment of a script, you can change its behavior if the script depends on the values of those environment variables.

Although they are less harmful for scripts these days (because scripts cannot be run setuid in any modern OS), they can still cause incorrect behavior. For setuid binaries, they are even more dangerous. These attacks can also be harmful in a multiuser setting if one user gains the ability to modify the login scripts of another user through a bug or incorrect configuration.

The most common environment attack is modifying the PATH environment variable. This variable controls what gets executed when you type a command without giving the full path.

Consider the following code:

```
#!/bin/sh


ls /tmp
```

The attack:

Create an executable binary or script that does something harmful and name it "ls". Then do this:

```
export PATH=/path/to/malicious/binary:$PATH

/path/to/above/script
```

Because the path to the malicious binary is first in the search path, the malicious `ls` command gets executed instead of the real one.

Mitigation:

Always specify absolute or relative paths when executing binaries or other scripts. If your script runs other scripts or binaries that do not use absolute or relative paths internally, you should explicitly set the value of the PATH environment variable in your scripts to prevent problems.

Attacks On Files In Publicly Writable Directories

Files in publicly writable directories, including temporary files, are vulnerable to attack by substituting a malicious file in place of the file your script intended to read or write.

Temporary File Attack

The simplest example of this attack is a tool storing secret information into a temporary file.

Consider the following code:

```
#!/bin/sh


SECRETDATA="My password is 12345."

echo > /tmp/mysecretdata

chmod og-rwx /tmp/mysecretdata

echo "$SECRETDATA" >> /tmp/mysecretdata
```

The attack:

Create a tool that watches for the file `/tmp/mysecretdata` to appear. (Although this can be done with a shell script, it probably won't be fast enough to work very often. Use the File System Events API in C instead.)

Upon detecting the existence of the path, do this:

```
FILE *fp=fopen("/tmp/mysecretdata", "r");
```

If the attacker manages to open the file before the script executes the `chmod` command, it can continue to read data from the file for as long as it keeps the file open.

Mitigation:

There are two things you must do to fix this:

- Always use the `umask` command to specify initial permissions on the file when you create it.

- Always create temporary files with the `mktemp` command. This creates a new file with the specified template, ensuring that a file or symbolic link with that name does not already exist.

For example:

```
#!/bin/sh


SECRETDATA="My password is 12345."
umask 0177
FILENAME="$(mktemp /tmp/mytempfile.XXXXXX)"
echo "$SECRETDATA" >> "$FILENAME"
```

However, assuming you actually intend to use the data again in the future, this mitigation is probably not sufficient either, for the reasons described in the next attack.

Input File Attack

A similar attack can be performed on files used as inputs to shell scripts.

Consider a script that executes the following code:

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

```
echo "My password is secret!" > /tmp/mypublicdata


...


PUBLICDATA="$(cat /tmp/mypublicdata)"


echo "$PUBLICDATA" | nc 192.168.1.102 3333
```

This script sends the contents of a temporary file to port 3333 of another computer at IP number 192.168.1.102 using the `nc` utility.

The attack:

Create a tool that watches for the file `/tmp/mydata` to appear. (Although this can be done with a shell script, it probably won't be fast enough to work very often. Use the File System Events API in C instead.)

Upon detecting the existence of the path, do this:

```
unlink("/tmp/mypublicdata");
unlink("/etc/myscretdata", "/tmp/mypublicdata");
```

If the attacker manages to do this before the script reads the file, then your secret password (presumably 12345, from the previous script) is sent unencrypted over port 3333. The attacker can then sniff for traffic on that port, and can log into your account (or at least unlock your luggage).

Mitigation:

This is particularly troublesome to mitigate because UNIX tools inherently follow symbolic links. The only way to solve the problem is to avoid writing the actual files into public directories. You should do this as follows:

- Always create temporary *directories* with the `mktemp` command, then create your actual temporary files *inside* those directories. By doing this, you can set restrictive permissions on the directory that will prevent an attacker from deleting your files and replacing them.

  If you specify the `-d` flag, the `mktemp` command creates a new directory with the specified template, ensuring that a file or directory with that name does not already exist.

- Always use the `umask` command to specify initial permissions on files and directories when you create them.

For example:

```
#!/bin/sh

umask 0177
TMPDIR="$(mktemp -d /tmp/mytempfile.XXXXXX)"
echo "My password is secret!" > "$TMPDIR"/mypublicdata


...


PUBLICDATA="$(cat "$TMPDIR"/mypublicdata)"


echo "$PUBLICDATA" | nc 192.168.1.102 3333
```

Injection Attacks

The most common type of attack in shell scripts is the injection attack. This type of attack occurs when arguments stored in user-provided variables are passed to commands without proper quoting.

Simple Example

Consider the following example:

```
read FOO
read BAR
if [ x$FOO = xfoo ] ; then
    echo $FOO
    eval $BAR
fi
```

This code has two security holes. Can you spot them?

- `if [ x$FOO = xfoo ] ; then`

  This statement allows for an injection attack on `FOO`.

  The attack:

  Pass "`foo = xfoo -o x`" as the value for `FOO`.

Despite the fact that the value of `FOO` is not "foo", the statement executes anyway. Depending on what this test does, this could potentially cause unexpected behavior.

Mitigation:

To fix this bug, change the if statement to read:

```
if [ "$FOO" = "foo" ] ; then
```

- `eval $BAR`

    This is a no-no. Never run eval on data passed in by a user unless you have very, very carefully sanitized it (and if possible, use a whitelist to limit the allowed values).

    The attack:

    Pass a dangerous command for `BAR`.

    Mitigation:

    Just don't do that.

Subtle Example

The following example is more subtle. Instead of running `eval`, it writes data to a script, but does so without protecting the values:

```
#!/bin/sh

read FOO

# ...

echo ls $FOO >> myscript.sh

# ...

chmod a+x myscript.sh
./myscript.sh
```

The attack:

Pass the value "`; rm randomfile`" to cause this script to delete a file.

The Wrong Mitigation:

You might be tempted to fix this bug by changing the echo and execution lines to read:

```
echo ls "\"$FOO\"" >> myscript.sh
export FOO
```

However, this still does not solve the problem because FOO is expanded immediately, which means that if the value of FOO contains a quotation mark—for example, `"'; rm randomfile ; echo '"`, you now have a different (but equally bad) security hole.

Correct Mitigation #1:

One way to fix this bug is to change the echo line to read:

```
echo ls "\"\$FOO\"" >> myscript.sh
```

This causes the variable FOO to be expanded when the script is executed. However, this works only if the variable FOO is exported, because otherwise the variable FOO would expand to nothing in the second script.

Correct Mitigation #2:

Another way to fix this bug is to change the echo line to read:

```
QUOTFOO="$(echo "$FOO" | sed "s/'/'\"'\"'/g")"
echo ls "'$QUOTFOO'" >> myscript.sh
```

By using single quotes around the string in the secondary script, the only character relevant to the shell is the single quote character. The `sed` command then replaces any single quote characters in the string with a closing single quote followed by a single quote wrapped in double quotes followed by an opening single quote.

Backwards Compatibility Example

The following example is not dangerous in modern shells, but is dangerous in older Bourne shells:

```
#!/bin/sh


read FOO
echo $FOO
```

The attack:

Pass the value "`; rm randomfile`" to cause this script to delete a file in older shells.

Most modern shells parse the statement prior to any variable substitution, and are thus unaffected by this attack. However, for proper security when your script is run on older systems (not to mention avoiding a syntax error if the filename contains spaces), you should still surround the variable with double quotes.

Mitigation:

To fix this bug, change the echo line to read:

```
echo "$FOO"
```

Authentication Attacks

In general, you should not rely on a script to determine whether a user does or does not have permission to do something. It is clumsy and error-prone. It is possible to do so, however, and there are right and wrong ways to do it.

The wrong way:

```
if [ $UID = 100 –a $USER = "myusername" ] ; then
    cd $HOME
fi
```

This code has three security bugs, and they're all caused by using variables in ways that are unsafe. For historical compatibility, the OS provides the UID, USER, and HOME environment variables. They are quite useful as long as you aren't using them for security reasons.

The attack:

```
$ tcsh
% setenv UID 100
% setenv USER myusername
% setenv HOME $HOME/.ssh
% /path/to/script.sh
```

Even though most modern Bourne shells protect against modifying `UID`, the `USER` variable is unprotected, and not all shells protect the `UID` variable, either.

Fortunately, the script just changed into a directory. Combined with another exploitable attack such as an injection attack, however, this could be exploited in bad ways.

Mitigation:

To obtain the user ID:

```
# Effective UID
MYEUID="$(/usr/bin/id -u)"


# Real UID
MYUID="$(/usr/bin/id -u -r)"
```

To obtain the username:

```
MYUID="$(/usr/bin/id -u -n)"
```

To obtain the actual home directory:

```
HOMEDIR="$(dscl . -read /Users/dg NFSHomeDirectory | sed 's/^NFSHomeDirectory: //')"
```

Note that this method for obtaining the home directory is specific to OS X.

Permissions and Access Control Lists

OS X uses the UNIX permissions model, extended by POSIX access control lists. These permissions models are described in detail in the OS X File System Security section of *File System Programming Guide* . This section assumes that you are already at least peripherally familiar with the concept of users and groups.

Examining File Permissions

UNIX permissions are visible to users in Terminal and in the Finder's Get Info window. In Terminal, you can easily look at the permissions in a human-readable format by using the `ls` command as follows:

```
$ ls -ld filename dirname

drwxr-xr-x  2 username  groupname  68 Jun 16 13:40 dirname

-rw-r--r--  1 username  groupname   0 Jun 16 13:40 filename
```

The left character indicates whether the file system object is a file (−), directory (d), symbolic link (l), block (b) or character (c) special file, named pipe (p), or UNIX domain socket (s).

The next three characters show the Owner permissions, followed by the Group permissions, and finally, the Other permissions as listed in the following table:

Permissions flag	Octal Bit Value	Meaning
-	n/a	No permission
r	4	Read permission
w	2	Write permission
x	1	Execute permission

s	In the optional first octal digit:
	    4—setuid
	    2—setgid
		Setuid or setgid with execute permission

S	See above.	Setuid or setgid `without` execute permission
t	In optional first octal digit: 1	Sticky bit

The complete set of permissions is often expressed in octal, as defined by the bits in the table above. The first digit includes the sticky bit and setuid and setgid bits. If zero, you may omit it when passing the value to most commands. The remaining three digits contain the Owner (user), Group, and Other permissions, respectively.

For example, a file that is setuid and setgid, with read/write/execute Owner permissions and read/execute Group and Other permissions, the octal equivalent is 6755:

- The leading special permissions value is 6, which is the bitwise OR of setuid (4) and setgid (2).

- The Owner permission is 7, which is the bitwise OR of the read (4), write (2), and execute (1) bits.

- The Group and Other permissions are both 5, which is the bitwise OR of the read (4) and execute (1) permissions.

To show the UNIX permissions of a file, use the stat command as follows:

```
stat -f "%p" filename
```

Ignore all but the last four digits returned.

Changing File Ownership and Permissions

The ability to change file ownership and permissions is limited by the operating system for security and quota reasons. Users can:

- Change the permissions for any file that they own.

- Change the group for any file that they own to any group that they are a member of.

Non-root users cannot:

- Change permissions on files owned by anyone else.

- Change the group of a file to a group that they are not a member of.

- Change the owner of any file.

The root user can change permissions and ownership arbitrarily except when blocked by BSD file system flags.

With those restrictions in mind, the sections that follow describe how to change permissions and change user and group ownership of files and directories.

Use chown and chgrp to Change User and Groups Ownership

You can change the owner of a file or directory with the chown command:

```
# Change the owner of a file or directory
sudo chown newowner filename_or_dirname


# Change the owner of a directory and everything in it recursively
sudo chown -R newowner dirname
```

You can change the group for a file with either the chown command or the chgrp command:

```
# Change the group by itself
```

```
chown :newgroup filename_or_dirname

chgrp newgroup filename_or_dirname


# Change the group of a directory and everything in it recursively

chown -R :newgroup dirname

chgrp -R newgroup dirname
```

You can also change both owner and group simultaneously:

```
# Change the owner and the group

sudo chown newowner:newgroup filename_or_dirname


# Change the group of a directory and everything in it recursively

sudo chown -R newowner:newgroup dirname
```

For more information, see the manual pages for `chown` and `chgrp`.

Use chmod to Change File and Directory Permissions

OS X (and other UNIX-based operating systems) provide the `chmod` command for changing the permissions of files and directories.

The `chmod` command, short for "change mode," is so named because it allows you to modify file or directory **modes**. A mode is a three-digit or four-digit octal representation of the UNIX permissions for a file (or 4-5 digits in languages that require a leading zero, such as C).

There are two basic ways you can use the `chmod` command: numeric modes and human-readable flags.

Most users use `chmod` in its human-readable form:

```
chmod a+rw world_writable_file
```

This command tells `chmod` to add read (`r`) and write (`w`) access to the existing set of permissions for all users (`a`). So if the permissions were originally `r-x--x-w-`, the resulting permissions would be `rwxrwxrw-`.

You can also add and subtract permissions for the owning user (`u`), the group (`g`), or other users (`o`) separately. For example, to add read (`r`), write (`w`), and execute (`x`) permission for the owning user and take it away from members of the owning group and everyone else, you could issue either of the following commands:

```
chmod u+rwx,g-rwx,o-rwx filename

chmod u+rwx,go-rwx filename

chmod a-rwx,u+rwx filename
```

Similarly, you can set the User, Group, or Other permissions without regard to what bits were set before by using equals. For example, to set group permissions to read, no-write, no-execute, you could issue the following command:

```
chmod g=r filename
```

Finally, to make an executable run setuid (u+s) and setgid (g+s), you might execute a command like one of the following:

```
chmod a+rx,ug+s filename

chmod a+rxs filename    # Note: o+s is ignored.
```

Alternatively, if you know the numeric file mode you want to apply (see Examining File Permissions (page 242) for details), you can pass the chmod command either a three-digit or four-digit mode value:

```
chmod 666 world_writable_file

chmod 0666 world_writable_file
```

The chmod command can also be used to modify POSIX access control lists (ACLs). This use is described later, in Use chmod to Modify Access Control Lists (page 247).

Use chflags to Set Special File Permission Flags

In addition to the standard permission flags, OS X has a few special permission flags that can be set using the chflags or lchflags command (or with the chflags or fchflags API in C). These flags are described in the OS X File System Security section of *File System Programming Guide*.

The permissions flags set with chflags take precedence over any permissions granted by normal UNIX permissions or access control lists.

The usage of the chflags command is fairly straightforward. For example, to make a file immutable (so that it cannot be moved, renamed, deleted, or modified), you can issue one of the following commands:

```
chflags uchg filename # user flag
```

```
sudo chflags schg filename # system flag
```

Notice that the flag comes in two variants: the user flag and the system flag. The user flag can be changed by the file's owner and `root` (just like normal permissions). The system flag can be changed solely by `root`.

To undo this change, you would issue one of the following commands:

```
chflags nouchg filename # user flag
sudo chflags noschg filename # system flag
```

For cross-platform compatibility and readability reasons, OS X supports two other variations on each of these flags: `uchange`, `uimmutable`, `schange`, and `simmutable`. These variants behave identically to their shortened forms.

There are several other flags you can set with the `chflags` command, the most common being the user and system append-only flags (`uappnd`/`uappend` and `sappnd`/`sappend`, respectively).

For more information, read the `chflags` and `lchflags` manual pages and the OS X File System Security section of *Security Overview* .

Use chmod to Modify Access Control Lists

The `chmod` command is most commonly known for its ability to modify UNIX permissions. However, in OS X, it also does double duty, providing the scripting interfaces for modifying a file's POSIX access control lists (ACLs).

The basic concept of ACLs is fairly straightforward. An access control list is a list of rules (access control entries, or ACEs).

- Each entry grants or denies the right to access a file or directory in a particular way (the right to read the file, for example).

- For any given right, the first entry in the list that matches against the current user's user ID or group membership wins.

- If the end of the list is reached without matching anything, the file or directory's UNIX permissions are used to determine access.

This is a greatly simplified explanation; for full details, read the OS X File System Security section of *Security Overview* .

Each ACL entry looks like this:

```
username grant rightname

groupname grant rightname

username deny rightname

groupname deny rightname
```

where *username* and *groupname* are the names of a user or group, respectively, and *rightname* is the name of an access right (read, for example).

You can add an access control entry with the +a flag to chmod. For example, to deny read access on a file to the MySQL user, you would type:

```
chmod +a "_mysql deny read" filename
```

To see the results of your changes, type:

```
ls –le filename
```

By default, new access control list entries are appended to the end of the list. If you need to insert an access control elsewhere in the list, you can use the +a# flag. For example, to insert a new rule at position zero (the top of the list), you would issue a command like this one:

```
chmod +a# 0 "_www deny read" filename
```

You can delete an access control entry with the –a flag like this:

```
chmod –a "_mysql deny read" filename
```

This command deletes any entry that is an exact match for the specified rule.

Finally, you can replace an entry with another entry using the =a# flag. For example, to change the username in the rule inserted above from _www to _mdnsresponder, you would type:

```
chmod =a# 0 "_mdnsresponder deny read" filename
```

In addition to the basic rules described above, the ACL system in OS X supports inheritance. Any inherited ACL entries for a directory are automatically copied to any new files created within that directory at the time of creation.

You can specify:

- whether an ACL should be inherited by:

    enclosed files—`file_inherit` right

    directories—`directory_inherit` right

    both—`file_inherit,directory_inherit` right

    neither (the default).

- whether an ACL should be inherited by the children of enclosed directories (the default) or not (`limit_inherit` right).

- whether an ACL should apply to the directory itself (the default) or merely be inherited by things inside it (`only_inherit` right).

You can specify any combination of these flags in an access control entry for a directory by passing the flags as part of the rights list.

For example:

```
chmod +a "_www deny list,search,directory_inherit" dirname
```

This rule prevents the _www user from listing the directory's contents. It also prevents the _www user from accessing any files within the specified directory even with an exact name lookup (search). The rule is inherited by any new directory created inside the specified directory (and any directory created inside that one, and so on), but is not inherited by ordinary files.

> **Note:** Inheritance flags apply exclusively to access control entries for directories. You cannot set these flags on files.

> **Cross-platform Compatibility Note:** Command-line tools behavior for modifying access control lists is not standardized. For tips on handling this across multiple platforms, see Access Control List (ACL) Management (page 150) in Designing Scripts for Cross-Platform Deployment (page 146).

For more information about the ACL scheme in OS X is described in OS X File System Security section of *Security Overview*. For more information about the command-line flags for getting and setting ACLs, see the manual page for `chmod`.

Securing Temporary Files

Because the temporary directories in OS X and other UNIX-based operating systems are world-writable, you must take care to ensure that you are modifying the file you think you are modifying.

For example, the following code has two serious bugs:

```
if [ ! -f /tmp/mytempfile ] ; then

    # Race condition here


    touch /tmp/mytempfile


    chmod u=rw,og= /tmp/mytempfile
    # Missing error check here


    echo My secret password is omnibus > /tmp/mytempfile
fi
```

An application that happens to get the timing right can create a file called `/tmp/mytempfile` right after the script checks for its existence, wait for the script to write data into it, and subsequently steal the password. The `chmod` command would produce an error in this case, but because the script doesn't check the result code, the error is moot.

To solve this problem, always use the `mktemp` command to create temporary files. The `mktemp` command creates files with initial permissions of 0600, and never returns an existing file. (Using `mktemp` also provides an easy way to obtain a known-unique filename, potentially avoiding unexpected behavior caused by temp file collisions.)

> **Important:** Although OS X does not use a privileged helper to clean up temporary files (except during a reboot), some operating systems do. If a script could potentially take a long time to execute without modifying a temporary file, such privileged cleanup helpers can open up a security vulnerability by deleting the existing temp file out from under your script.
>
> Because of this risk, system-provided temporary directories should only be used to store sensitive data *briefly*. You should do as little work as possible between creating the file and using it, and should clean up the file as soon as possible afterwards.
>
> Further, if you suspend your scripts for any significant period of time, your scripts *must* create any sensitive temporary files in a non-world-writable directory.
>
> You should avoid writing sensitive data out to temporary files at all if you can possibly avoid it.

Flags That Affect Security (and Correctness)

The `set` builtin (described in the `sh` man page) sets a number of shell features that can be used to reduce the risk posed by certain types of common programming mistakes. These flags allow your scripts to automatically exit if an unset variable is expanded, automatically exit if any simple commands fail, or automatically export variables.

In addition, the BASH shell provides a flag that causes pipes to return a nonzero exit status when any command in the chain of pipes exits with an error instead of always returning the exit status of the last command. It also supports a flag that limits the effect of environment variables on the interpreter, intended for use in scripts that are expected to be run as a privileged user (for example, the `root` user).

Detecting Unset Variables

By default, the Bourne shell treats unset variables as empty (unlike `csh`). If your script expects that unset variable to contain a value, this can lead to incorrect script execution and, depending on the script, may even result in a security hole. To guard against this, you can issue the following command:

```
set -u
```

With this flag set, if your script tries to use an empty variable, the shell prints an error message, and the entire script exits immediately with a nonzero exit status.

> **Note:** If your script changes its behavior deliberately based on the presence or absence of one or more environment variables, you should typically perform those tests before you set this flag.

If desired, you can later restore the default behavior with the following command:

```
set +u
```

Checking Exit Status Automatically

For very simple scripts, checking the exit status of each command can be tedious. You can greatly simplify these scripts by instead issuing the following command:

```
set –e
```

With this flag set, if any simple command exits with a nonzero exit status, the shell terminates with that command's exit status. A simple command is defined as a command that includes no pipes or lists, that is not executed as part of a control statement, and whose exit status is not inverted with an exclamation point.

> **Important:** Because there are many situations in which errors can be masked (particularly in pipes and lists), this flag is *not* a substitute for proper error checking in complex scripts.

If desired, you can later restore the default behavior with the following command:

```
set +e
```

Exporting Variables Automatically

It is not always necessary to export variables that your script uses internally. However, if a child process depends on the values of those variables, they must be exported. In some cases, failing to export a variable could even result in a security hole if it causes the child to grant a user access that he or she would otherwise not have. For example, if a CGI script running in a web server environment provides additional limits on what files a remote user can access, a bug in that script might give the user access to other files.

You can, if desired, tell the shell to automatically export any variable that your script sets by issuing the following command:

```
set –a
```

> ⚠️ **Warning:** Automatically exporting variables can also *cause* a security hole by exporting variables containing sensitive data, such as internal passwords and application keys, into the environments of every command that your script executes. If the output of those commands could be seen by an untrusted user—commands executed by a CGI script, for example—then you risk leaking sensitive data. For this reason, you should avoid setting this flag if your script contains any sensitive data, such as internal passwords or application keys.

If desired, you can later restore the default behavior with the following command:

```
set +a
```

Retrieving the Exit Status of Piped Commands in BASH

The exit status of a series of commands connected by pipes is, by default, the exit status of the rightmost command. If you do not examine the output from the final command to ensure that it makes sense, this default behavior can potentially mask errors that might lead to security problems.

For example, consider the following code:

```
ls nonexistentfile | cat
echo $?
```

In the first command, even though the `ls` command fails, the cat command does not care whether it received any input or not, and thus exits with a zero exit status. As a result, the pipe's exit status is zero. If it is critical to know whether the first command failed (for example, if it performs an operation with an important side effect, such as removing a file on disk), then this is potentially unsafe.

There are many ways that you can fix this problem. The most obvious fix is to store the results of the first command into a variable temporarily, check the result code of the first command, and then use `echo` to pipe the results to the second command. This technique is often less than ideal for commands that take a long time to execute or produce large amounts of output, however, because the second command does not receive any data until after the first command exits. The performance impact is particularly noticeable if the output of the final command is expected to be read by the user.

As an alternative, in BASH, you can issue the following command before issuing the commands above:

```
set -o pipefail
```

After issuing this command, the pipe's exit status is provided by the rightmost command that failed with a nonzero exit status, or zero if every command in the chain of pipes exited successfully. In the earlier example, the final `echo` command would print the number `1` (the exit status of the `ls` command).

> **Note:** This feature is specific to BASH and is not supported by other Bourne shell implementations. If you use this feature, you should change the interpreter line to the following:
>
> ```
> #!/bin/bash
> ```
>
> If you are writing a script that must be portable to other `sh` implementations, you cannot use this setting. Instead, either store the results in an intermediate variable or file, or check the final result carefully to ensure that it makes sense.

If desired, you can later restore the default behavior with the following command:

```
set +o pipefail
```

Sanitizing the Environment in BASH

For BASH shell scripts (or Bourne shell scripts running in BASH) that must run in a privileged environment (as the `root` user, for example), it is a good idea to tell the shell to not automatically execute any "run commands" files (`.bashrc`, `.profile`, and so on) that may contain alias commands that affect script execution, functions that may override commands in your script, or even malicious commands that an attacker wants your script to execute while running as the `root` user.

To sanitize the script's environment in this way, you should change your script's interpreter line to the following:

```
#!/bin/bash -p
```

In this mode, the scripts referenced by the ENV and BASH_ENV environment variables are not executed, shell functions are not inherited, and the `SHELLOPTS` environment variable is ignored.

> **Note:** Although you can theoretically set this value with the `set` builtin, by the time your script actually starts running commands, the damage is already done. For this reason, you should always set this flag in the interpreter line.
>
> Also, you should be aware that this flag is specific to BASH, and is not broadly available in other shells.

Command Line Primer

Historically, the command line interface provided a way to manipulate a computer over simple, text-based connections. In the modern era, in spite of the ability to transmit graphical user interfaces over the Internet, the command line remains a powerful tool for performing certain types of tasks.

As described previously in Before You Begin (page 15), most users interact with a command-line environment using the Terminal application, though you may also use a remote connection method such as secure shell (SSH). Each Terminal window or SSH connection provides access to the input and output of a shell process. A shell is a special command-line tool that is designed specifically to provide text-based interactive control over other *command-line tools*.

In addition to running individual tools, most shells provide some means of combining multiple tools into structured programs, called shell scripts (the subject of this book).

Different shells feature slightly different capabilities and scripting syntax. Although you can use any shell of your choice, the examples in this book assume that you are using the standard OS X shell. The standard shell is `bash` if you are running OS X v10.3 or later and `tcsh` if you are running an earlier version of the operating system.

The following sections provide some basic information and tips about using the command-line interface more effectively; they are not intended as an exhaustive reference for using the shell environments.

Note: This appendix was originally part of *Mac Technology Overview* .

Basic Shell Concepts

Before you start working in any shell environment, there are some basic features of shell scripting that you should understand. Some of these features are specific to OS X, but most are common to all platforms that support shell scripting.

Running Your First Command-Line Tool

In general, you run command-line tools that OS X provides by typing the name of the tool. (The syntax for running tools that you've added is described later in this appendix.)

For example, if you run the `ls` command, by default, it lists the files in your home directory. To run this command, type `ls` and press Return.

Most tools also can take a number of flags (sometimes called switches). For example, you can get a "long" file listing (with additional information about every file) by typing `ls -l` and pressing Return. The `-l` *flag* tells the `ls` command to change its default behavior.

Similarly, most tools take arguments. For example, to show a long listing of the files on your OS X desktop, type `ls -l Desktop` and press Return. In that command, the word `Desktop` is an argument that is the name of the folder that contains the contents of your OS X desktop.

In addition, some tools have flags that take flag-specific arguments in addition to the main arguments to the tool as a whole.

Specifying Files and Directories

Most commands in the shell operate on files and directories, the locations of which are identified by paths. The directory names that make up a path are separated by forward-slash characters. For example, the Terminal program is in the `Utilities` folder within the `Applications` folder at the top level of your hard drive. Its path is `/Applications/Utilities/Terminal.app`.

The shell (along with, for that matter, all other UNIX applications and tools) also has a notion of a current working directory. When you specify a filename or path that does not start with a slash, that path is assumed to be relative to this directory. For example, if you type `cat foo`, the `cat` command prints the contents of the file `foo` in the current directory. You can change the current directory using the `cd` command.

Finally, the shell supports a number of directory names that have a special meaning.

Table A-1 lists some of the standard shortcuts used to represent specific directories in the system. Because they are based on context, these shortcuts eliminate the need to type full paths in many situations.

Table A-1   Special path characters and their meaning

Path string	Description
`.`	The `.` directory (single period) is a special directory that, when accessed, points to the current working directory. This value is often used as a shortcut to eliminate the need to type in a full path when running a command. For example, if you type `./mytool` and press return, you are running the `mytool` command in the current directory (if such a tool exists).

Path string	Description
`..`	The `..` directory (two periods) is a special directory that, when accessed, points to the directory that contains the current directory (called its *parent directory*). This directory is used for navigating up one level towards the top of the directory hierarchy. For example, the path `../Test` is a file or directory (named `Test`) that is a sibling of the current directory. **Note:** Depending on the shell, if you follow a symbolic link into a subdirectory, typing `cd ..` directory will either take you back to the directory you came from or will take you to the parent of the current directory.
`~` or `$HOME`	At the beginning of a path, the tilde character represents the home directory of the specified user, or the currently logged in user if no user is specified. (Unlike `.` and `..`, this is not an actual directory, but a substitution performed by the shell.) For example, you can refer to the current user's `Documents` folder as `~/Documents`. Similarly, if you have another user whose short name is `frankiej`, you could access that user's Documents folder as `~frankiej/Documents` (if that user has set permissions on his or her Documents directory to allow you to see its contents). The `$HOME` environment variable can also be used to represent the current user's home directory. In OS X, the user's home directory usually resides in the `/Users` directory or on a network server.

File and directory names traditionally include only letters, numbers, hyphens, the underscore character (_), and often a period (`.`) followed by a file extension that indicates the type of file (`.txt`, for example). Most other characters, including space characters, should be avoided because they have special meaning to the shell.

Although some OS X file systems permit the use of these other characters, including spaces, you must do one of the following:

- "Escape" the character—put a backslash character (\) immediately before the character in the path.

- Add single or double quotation marks around the path or the portion that contains the offending characters.

For example, the path name `My Disk` can be written as `"My Disk"`, `'My Disk'`, or `My\ Disk`.

Single quotes are safer than double quotes because the shell does not do any interpretation of the contents of a single-quoted string. However, double quotes are less likely to appear in a filename, making them slightly easier to use. When in doubt, use a backslash before the character in question, or two backslashes to represent a literal backslash.

For more detailed information, see Quoting Special Characters (page 66) in Flow Control, Expansion, and Parsing (page 46).

Accessing Files on Additional Volumes

On a typical UNIX system, the storage provided by local disk drives is presented as a single tree of files descending from a single root directory. This differs from the way the Finder presents local disk drives, which is as one or more volumes, with each volume acting as the root of its own directory hierarchy. To satisfy both worlds, OS X includes a hidden directory, `Volumes`, at the root of the local file system. This directory contains all of the volumes attached to the local computer.

To access the contents of other local (and many network) volumes, you prefix the volume-relative path with `/Volumes/` followed by the volume name. For example, to access the `Applications` directory on a volume named `MacOSX`, you would use the path `/Volumes/MacOSX/Applications`.

> **Note:** To access files on the boot volume, you are not required to add volume information, since the root directory of the boot volume is `/`. Including the volume information still works, though, so if you are interacting with the shell from an application that is volume-aware, you may want to add it, if only to be consistent with the way you access other volumes. You must include the volume information for all volumes other than the boot volume.

Input And Output

Most tools take text input from the user and print text out to the user's screen. They do so using three standard file descriptors, which are created by the shell and are inherited by the program automatically. These standard file descriptors are listed in Table A-2.

Table A-2   Input and output sources for programs

File descriptor	Description
stdin	The standard input file descriptor is the means through which a program obtains input from the user or other tools. By default, this descriptor provides the user's keystrokes. You can also redirect the output from files or other commands to `stdin`, allowing you to control one tool with another tool.
stdout	The standard output file descriptor is where most tools send their output data. By default, standard output sends data back to the user. You can also redirect this output to the input of other tools.

File descriptor	Description
`stderr`	The standard error file descriptor is where the program sends error messages, debug messages, and any other information that should not be considered part of the program's actual output data. By default, errors are displayed on the command line like standard output. The purpose for having a separate error descriptor is so that the user can redirect the actual output data from the tool to another tool without that data getting corrupted by non-fatal errors and warnings.

To learn more about working with these descriptors, including redirecting the output of one tool to the input of another, read Shell Input and Output (page 35).

Terminating Programs

To terminate the currently running program from the command line, press Control-C. This keyboard shortcut sends an abort (ABRT) signal to the currently running process. In most cases this causes the process to terminate, although some tools may install signal handlers to trap this signal and respond differently. (See Trapping Signals (page 173) in Advanced Techniques (page 168) for details.)

In addition, you can terminate most scripts and command-line tools by closing a Terminal window or SSH connection. This sends a hangup (HUP) signal to the shell, which it then passes on to the currently running program. If you want a program to continue running after you log out, you should run it using the nohup command, which catches that signal and does not pass it on to whatever command it invokes.

Frequently Used Commands

Shell scripting involves a mixture of built-in shell commands and standard programs that run in all shells. Although most shells offer the same basic set of commands, there are often variations in the syntax and behavior of those commands. In addition to the shell commands, OS X also provides a set of standard programs that run in all shells.

Table A-3 lists some commands that are commonly used interactively in the shell. Most of the items in this table are not specific to any given shell. For syntax and usage information for each command, see the corresponding man page. For a more in-depth list of commands and their accompanying documentation, see *OS X Man Pages*.

Table A-3   Frequently used commands and programs

Command	Meaning	Description
`cat`	(con)catenate	Prints the contents of the specified files to `stdout`.
`cd`	change directory	Changes the current working directory to the specified path.
`cp`	copy	Copies files (and directories, when using the -r option) from one location to another.
`date`	date	Displays the current date and time using the standard format. You can display this information in other formats by invoking the command with specific flags.
`echo`	echo to output	Writes its arguments to `stdout`. This command is most often used in shell scripts to print status information to the user.
`less` and `more`	pager commands	Used to scroll through the contents of a file or the results of another shell command. This command allows forward and backward navigation through the text. The `more` command got its name from the prompt "Press a key to show more…." commonly used at the end of a screenful of information. The `less` command gets its name from the idiom "less is more".
`ls`	List	Displays the contents of the specified directory (or the current directory if no path is specified). Pass the `–a` flag to list all directory contents (including hidden files and directories). Pass the `–l` flag to display detailed information for each entry. Pass `–@` with `–l` to show extended attributes.
`mkdir`	Make Directory	Creates a new directory.
`mv`	Move	Moves files and directories from one place to another. You also use this command to rename files and directories.
`open`	Open an application or file.	You can use this command to launch applications from Terminal and optionally open files in that application.
`pwd`	Print Working Directory	Displays the full path of the current directory.

Command	Meaning	Description
`rm`	Remove	Deletes the specified file or files. You can use pattern matching characters (such as the asterisk) to match more than one file. You can also remove directories with this command, although use of `rmdir` is preferred.
`rmdir`	Remove Directory	Deletes a directory. The directory must be empty before you delete it.
Ctrl-C	Abort	Sends an abort signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.
Ctrl-Z	Suspend	Sends the SIGTSTP signal to the current command. In most cases this causes the command to be suspended, although commands may install signal handlers to trap this command and respond differently. Once suspended, you can use the `fg` builtin to bring the process back to the foreground or the `bg` builtin to continue running it in the background.
Ctrl-\	Quit	Sends the SIGQUIT signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.

Environment Variables

Some programs require the use of environment variables for their execution. Environment variables are variables inherited by all programs executed in the shell's context. The shell itself uses environment variables to store information such as the name of the current user, the name of the host computer, and the paths to any executable programs. You can also create environment variables and use them to control the behavior of your program without modifying the program itself. For example, you might use an environment variable to tell your program to print debug information to the console.

To set the value of an environment variable, you use the appropriate shell command to associate a variable name with a value. For example, to set the environment variable `MYFUNCTION` to the value `MyGetData` in the global shell environment you would type the following command in a Terminal window:

```
# In Bourne shell variants
export MYFUNCTION="MyGetData"
```

```
# In C shell variants
setenv MYFUNCTION "MyGetData"
```

When you launch an application from a shell, the application inherits much of its parent shell's environment, including any exported environment variables. This form of inheritance can be a useful way to configure the application dynamically. For example, your application can check for the presence (or value) of an environment variable and change its behavior accordingly. Different shells support different semantics for exporting environment variables, so see the man page for your preferred shell for further information.

Child processes of a shell inherit a copy of the environment of that shell. Shells do not share their environments with one another. Thus, variables you set in one Terminal window are not set in other Terminal windows. Once you close a Terminal window, any variables you set in that window are gone.

If you want the value of a variable to persist between sessions and in all Terminal windows, you must either add it to a login script or add it to your environment property list. See Before You Begin (page 15) for details.

Similarly, environment variables set by tools or subshells are lost when those tools or subshells exit.

Running User-Added Commands

As mentioned previously, you can run most tools by typing their name. This is because those tools are located in specific directories that the shell searches when you type the name of a command. The shell uses the PATH environment variable to control where it searches for these tools. It contains a colon-delimited list of paths to search—`/usr/bin:/bin:/usr/sbin:/sbin`, for example.

If a tool is in any other directory, you must provide a path for the program to tell it where to find that tool. (For security reasons, when writing scripts, you should always specify a complete, absolute path.)

For security reasons, the current working directory is *not* part of the default search path (PATH), and should not be added to it. If it were, then another user on a multi-user system could trick you into running a command by adding a malicious tool with the same name as one you would typically run (such as the `ls` command) or a common misspelling thereof.

For this reason, if you need to run a tool in the current working directory, you must explicitly specify its path, either as an absolute path (starting from `/`) or as a relative path starting with a directory name (which can be the `.` directory). For example, to run the `MyCommandLineProgram` tool in the current directory, you could type `./MyCommandLineProgram` and press Return.

With the aforementioned security caveats in mind, you can add new parts (temporarily) to the value of the PATH environment variable by doing the following:

```
echo "$PATH"


# In Bourne shell variants
export PATH="$PATH:/my/new/path/part"


# In C shell variants
setenv PATH "$PATH:/my/new/path/part"
```

If you want the additional path components to persist between sessions and in all Terminal windows, you must either add it to a login script or add it to your environment property list. See Before You Begin (page 15) for details.

Running Applications

To launch an application, you can generally either:

- **Use the** open **command.**

  ```
  open /path/to/MyApp.app
  ```

- **Run the application binary itself.**

  Type the pathname of the executable file inside the package.

  ```
  /path/to/MyApp.app/Contents/MacOS/MyApp
  ```

> **Note:** As a general rule, if you launch a GUI application from a script, you should run that script only within Terminal or another GUI application. You cannot necessarily launch an GUI application when logged in remotely (using SSH, for example). In general, doing so is possible only if you are also logged in using the OS X GUI, and in some versions of OS X, it is disallowed entirely.

Learning About Other Commands

At the command-line level, most documentation comes in the form of man pages (short for manual). Man pages provide reference information for many shell commands, programs, and POSIX-level concepts. The manual page `manpages` describes the organization of manual, and the format and syntax of individual man pages.

To access a man page, type the `man` command followed by the name of the thing you want to look up. For example, to look up information about the `bash` shell, you would type `man bash`. The man pages are also included in the OS X Developer Library (*OS X Man Pages*).

You can also search the manual pages by keyword using the `apropos` command.

> **Note:** Not all commands and programs have man pages. For a list of available man pages, look in the `/usr/share/man` directory or see *OS X Man Pages* in the OS X Developer Library.

Most shells have a command or man page that displays the list of commands that are built into the shell (builtins). Table A-4 lists the available shells in OS X along with the ways you can access the list of builtins for the shell.

Table A-4   Getting a list of shell builtins

Shell	Command
bash	`help` or `bash` -c help
sh	`man sh`
csh	`builtins`
tcsh	`builtins`
zsh	`man zshbuiltins`

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 B-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. This is commonly used in conjunction with the `wait` builtin.
`$PATH`	A colon-delimited list of locations where trusted executables are installed. Any executable in one of these locations can be executed without specifying a complete path.
Field and record parsing	
`$IFS`	Input Field Separators (uses are explained in Variable Expansion and Field Separators (page 62))
User information	
`$HOME`	The user's home directory.
`$UID`	The user's ID. **Security Warning:**This value can be modified by the calling script, so it should not be used for authentication purposes.

Variable	Description
$USER	The user's (short) login name. **Security Warning:**This value can be modified by the calling script, so it should not be used for authentication purposes.
Miscellaneous Variables	
$#	Number of arguments passed to the shell. This variable is described further in Handling Flags and Arguments (page 74).
$@	Complete list of arguments passed to the shell, separated by spaces.. This variable is described further in Handling Flags and Arguments (page 74).
$*	Complete list of arguments passed to the shell, separated by the first character of the IFS (input field separators) variable. This variable is described further in Handling Flags and Arguments (page 74).
$–	A list of all shell flags currently enabled.
$PWD	The current working directory. Equivalent to executing the pwd command.

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 is 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 C-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 100). For additional usage notes specific to individual applications, see the manual page for the command itself.

Table C-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 tool is described further in How AWK-ward (page 122).
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 C-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 tool 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 C-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_cd9660`, `mount_cddafs`, `mount_fdesc`, `mount_ftp`, `mount_hfs`, `mount_msdos`, `mount_nfs`, `mount_ntfs`, `mount_smbfs`, `mount_udf`, `mount_url`, and `mount_webdav`)	Mounts and unmounts volumes. If you unmount automounted volumes behind the back of the disk arbitration 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 C-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.
`gzip`, `gunzip`, `zcat`, and `gzcat`	Compresses, uncompresses, and prints the contents of files in the GNU Zip (LZ77-based) format. This compression scheme 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 multifile archives in the `tar` (short for "Tape ARchive") format. This format 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 OS X by looking in the man pages, either online (*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!

Starting Points

This appendix provides a number of short script snippets that simplify common tasks and provides links to a few other scripts in other chapters.

Files and Directories

Copying Files and Directories

The first script demonstrates how to copy a folder full of files and folders to a different location using `cp`.

> ⚠️ **Warning:** Warning: Do not put a slash at the end of the name of `folder_to_copy`. In some operating systems, this causes the contents of `folder_to_copy` to be copied into `destination_directory` instead of the whole folder.

Listing D-1   Copying a folder recursively

```
cp –R –p folder_to_copy destination_directory
```

The next script shows how to copy a tree of files and folders, preserving the source directory structure using `tar`. For example, this results in `destination/file1`, `destination/dir2/file2`, and so on.

Listing D-2   Copying multiple files and directories to another location, preserving the directory structure

```
tar –czf – file1 dir2/file2 dir3/file3 | \
    { cd /destination ; tar –xzf – ; }
```

The next two scripts show how to copy entire trees of files from one server to another securely using `tar` and `ssh`.

Listing D-3   Copying a tree of files and folders from the current directory to a remote computer

```
# Copies directory_or_file_name on the local machine
```

```
# to /path/to/destination/directory_or_file_name on

# a remote machine.


tar –czf – directory_or_file_name | ssh username@hostname \

    "cd /path/to/destination; tar –xzf –"
```

Listing D-4   Copying a tree of files and folders from a remote computer to the current directory

```
# Copies the directory called directory_name from

# /path/to/source/directory_name on a remote server

# to the current directory on the local machine.


ssh username@hostname "cd /path/to/source; \

    tar –czf – directory_name" | tar –xzf –
```

The following script recovers from a failed `tar` copy. Normally, you would just use `rsync`, but occasionally you may have to copy lots of files to or from an ISP that disallows `rsync` and sets an unreasonably low maximum CPU time for executables, causing `tar` to die repeatedly.

Note: This script uses the `stat` command-line tool, which uses completely nonstandard flags across different operating systems. The variables `LOCALFORMATFLAG`, `LOCALFORMAT`, `REMOTEFORMATFLAG`, and `REMOTEFORMAT` must be adjusted for the operating system on the local and remote systems, respectively. The examples given cover OS X and Linux. See the manual page for `stat` on each machine to determine the correct flags. The format string should contain the path of the file, followed by a space, followed by the length of the file (in bytes).

Listing D-5   Code to recover from a truncated tar copy

```
#!/bin/sh


USERNAME="remoteuser"
REMOTEHOST="remotehost.example.org"
SRCDIR="/path/to/testdir"
OUTDIR="/remote/path/here"
```

```
# Format is "path bytecount"
LOCALFORMATFLAG="-f"  # OS X
LOCALFORMAT="%N %z"   # OS X
REMOTEFORMATFLAG="-c" # Linux
REMOTEFORMAT="%n %s"  # Linux


OUTDIRQUOTED="$(echo "$OUTDIR" | sed 's/"/\\"/g')"


IFS="
"


BACKUPLIST=""


cd "$SRCDIR"


# Generate a list of files and their length in bytes on the local
# and local machines.
LOCALFILELIST="$(cd "$SRCDIR" ; find . -exec stat "$LOCALFORMATFLAG" \
    "$LOCALFORMAT" {}  \; | sort)"
REMOTEFILELIST="$(ssh $USERNAME@$REMOTEHOST "cd \"$OUTDIRQUOTED\" ; \
    find . -exec stat "$REMOTEFORMATFLAG" '$REMOTEFORMAT' {}  \; | sort")"


# echo "RFL: $REMOTEFILELIST"


# Loop until there are no more local files to check.
while true ; do
    LNFILES="$(echo "$LOCALFILELIST" | grep -c .)"
    LNFM1="$(expr "$LNFILES" '-' '1')"
    RNFILES="$(echo "$REMOTEFILELIST" | grep -c .)"
    RNFM1="$(expr "$RNFILES" '-' '1')"


    # echo "@TOP LNFM1: $LNFM1 RNFM1 $RNFM1"


    # If there are no more local files, break out of the outer loop.
    # Otherwise, pop the first filename from the list.
```

```
    if [ $LNFM1 -lt 0 ] ; then
        break;
    else
        LOCALLINE="$(echo "$LOCALFILELIST" | head -n 1)"
        LOCALFILE="$(echo "$LOCALLINE" | sed 's/ [0-9][0-9]*$//')"
        LOCALQUOTED="$(echo "$LOCALFILE" | sed 's/"/\\"/g')"
        LOCALLENGTH="$(echo "$LOCALLINE" | \sed 's/.* \([0-9][0-9]*\)$/\1/')"
        LOCALFILELIST="$(echo "$LOCALFILELIST" | tail -n $LNFM1)"
    fi


    # If there are no more remote files, every local file must
    # be added to the list of files to copy.
    # Otherwise, pop the first filename from the list.
    if [ $RNFM1 -lt 0 ] ; then
        REMOTELINE=""
        REMOTEFILE=""
        REMOTELENGTH=0
        REMOTEFILELIST=""
    else
        REMOTELINE="$(echo "$REMOTEFILELIST" | head -n 1)"
        REMOTEFILE="$(echo "$REMOTELINE" | sed 's/ [0-9][0-9]*$//')"
        REMOTELENGTH="$(echo "$REMOTELINE" | sed 's/.* \([0-9][0-9]*\)$/\1/')"
        REMOTEFILELIST="$(echo "$REMOTEFILELIST" | tail -n $RNFM1)"
    fi


    # echo "OLOOP LOCALFILE: $LOCALFILE REMOTEFILE: $REMOTEFILE"
    # echo "LOCALFILELIST: $LOCALFILELIST"
    # echo "REMOTEFILELIST: $REMOTEFILELIST"


    # If the filenames do not match, then the local file does
    # not exist on the remote server (because the lists are sorted).
    if [ "$LOCALFILE" != "$REMOTEFILE" ] ; then


        # Until they do match, keep adding files to the list of stuff to copy.
```

```
while [ "$LOCALFILE" != "$REMOTEFILE" -a "$LOCALFILE" != "" ] ; do
    # echo "NOMATCHLOOP LOCALFILE: $LOCALFILE REMOTEFILE: $REMOTEFILE"


    # echo "ADDED \"$LOCALQUOTED\" TO BACKUP LIST"


    BACKUPLIST="$BACKUPLIST \"$LOCALQUOTED\""


    # If it is a directory, adding the directory to the archive
    # adds everything in it, so skip everything in it.
    if [ -d "$LOCALFILE" ] ; then
        # echo "ISDIR"


        DIRLOOP=1
        LList2="$LOCALFILELIST"


        # Loop until we run out of files or the names do not match.
        while [ $DIRLOOP = 1 ] ; do
            LOCALFILE="$(echo "$LOCALFILE" | sed 's/\/$//')"
            LOCALQUOTED="$(echo "$LOCALFILE" | sed 's/"/\\"/g')"


            LNFILES2="$(echo "$LList2" | grep -c .)"
            LNFM1_2="$(expr "$LNFILES2" '-' '1')"


            # echo "LList2: $LList2"
            if [ $LNFM1_2 -lt 0 ] ; then
                # We ran out of files, so stop looking for files in
                # the directory.

                LLine2=""
                LF2=""
                LLen2=0
                LList2=""
                DIRLOOP=0
            else
                # Grab the next file in the list.
```

```
                    LLine2="$(echo "$LList2" | head -n 1)"
                    LF2="$(echo "$LLine2" | sed 's/ [0-9][0-9]*$//')"
                    LLen2="$(echo "$LLine2" | \
                        sed 's/.* \([0-9][0-9]*\)$/\1/')"
                    LList2="$(echo "$LList2" | tail -n $LNFM1_2)"


                    # echo "INDIRLOOP: FILE IS $LF2"


                    # Repeatedly strip off the last part of the path
                    # until it matches or the path is empty.
                    INDIR="NO"
                    while [ "$LF2" != "" -a "$LF2" != "." ] ; do
                        # echo "LF2: \"$LF2\""
                        LF2="$(dirname "$LF2" | sed 's/\/$//')";
                        if [ "$LF2" = "$LOCALFILE" ] ; then
                            # It matches.  The file is in the directory.
                            INDIR="YES"; LF2="";
                        fi
                    done
                    if [ $INDIR = "YES" ] ; then
                        # Because this file is in the directory, commit
                        # the changes to the local file list (thus
                        # removing this file from the list).


                        # echo "INDIR"
                        LOCALFILELIST="$LList2"
                    else
                        # This file is not in the directory.  Don't take it
                        # off the list, and stop looking for files in the
                        # directory.


                        # echo "NOTINDIR"
                        DIRLOOP=0
                    fi
```

```
                fi

        done


        # Recount the number of files in the local list because it may
        # have changed significantly.
        LNFILES="$(echo "$LOCALFILELIST" | grep -c .)"
        LNFM1="$(expr "$LNFILES" '-' '1')"


    else
        # It is not a directory.  Pop the file from the list.


        # echo "@BOTTOM LOCALFILELIST: $LOCALFILELIST"


        # Recount the number of files in the local list.
        LNFILES="$(echo "$LOCALFILELIST" | grep -c .)"
        LNFM1="$(expr "$LNFILES" '-' '1')"


        # echo "@BOTTOM LNFM1: $LNFM1 RNFM1 $RNFM1"


        # Grab the next file.  This is the middle loop iterator
        # testing to see if the filename matches.
        if [ $LNFM1 -lt 0 ] ; then
            LOCALLINE=""
            LOCALFILE=""
            LOCALQUOTED=""
            LOCALLENGTH=0
            LOCALFILELIST=""
        else
            LOCALLINE="$(echo "$LOCALFILELIST" | head -n 1)"
            LOCALFILE="$(echo "$LOCALLINE" | sed 's/ [0-9][0-9]*$//')"
            LOCALQUOTED="$(echo "$LOCALFILE" | sed 's/"/\\"/g')"
            LOCALLENGTH="$(echo "$LOCALLINE" | \
                sed 's/.* \([0-9][0-9]*\)$/\1/')"
            LOCALFILELIST="$(echo "$LOCALFILELIST" | tail -n $LNFM1)"
        fi
```

```
            fi
        done
    fi


    # When the script reaches this point,
    if [ "$LOCALFILE" = "$REMOTEFILE" -a "$LOCALFILE" != "" \
            -a $LOCALLENGTH != $REMOTELENGTH ] ; then
        if [ ! -d "$LOCALFILE" ] ; then
            # echo "ADDED \"$LOCALQUOTED\" TO BACKUP LIST"
            BACKUPLIST="$BACKUPLIST \"$LOCALQUOTED\""
        fi
    fi
done


echo "BACKUPLIST $BACKUPLIST"


if [ "$BACKUPLIST" != "" ] ; then
        eval tar -czf - $BACKUPLIST  | ssh $USERNAME@$REMOTEHOST \
                "cd \"$OUTDIRQUOTED\" ; tar -xzf -"
fi
```

Renaming Files

The following example shows how to standardize the case of the file extension on image files.

```
find photo_directory -iname '*.jpg' -exec \
    mv {} `echo {} | sed 's/\.[jJ][pP][gG]$/.jpg/'` \;
```

Converting File Line Endings

show how to convert between the line ending formats used for text files on various platforms.

Image Manipulation

In Advanced Techniques (page 168), Listing 11-13 (page 208) shows how to resize an image using `osascript`.

In addition to the `osascript` interface, OS X includes the `sips` command, which provides a direct shell interface to some of the image processing features in OS X.

The following snippet shows how to use `sips` to scale an image to a maximum of 250 pixels horizontally or vertically and convert the image to JPEG format.

```
sips -s format jpeg --resampleHeightWidthMax 250 myphoto.tif --out mythumb.jpg
```

You can also combine `sips` with `exiftool` (available from http://www.sno.phy.queensu.ca/~phil/exiftool/) for even greater power and control. The following script uses `sips` and `exiftool` to automatically rotate a photograph based on the encoded orientation information, and allows you to specify an offset (in 90 degree increments) to adjust the rotation further.

Listing D-6   Rotating an image using sips

```sh
#!/bin/sh


# Adjust paths as needed
EXIFTOOL=/usr/local/bin/exiftool
SIPS=/usr/bin/sips


INPUTFILE="$1"
OUTPUTFILE="$2"
OFFSET="$3"


# If the user doesn't specify an offset, assume zero.
if [ "$OFFSET" = "" ] ; then
    OFFSET=0
fi


# Use exiftool to read the EXIF orientation tag as a raw numeric value.
ORIENTATION="$($EXIFTOOL -b -Orientation $INPUTFILE)"


# If no orientation tag is found, assume no rotation is needed.
```

```
if [ "$ORIENTATION" = "" ] ; then

    ORIENTATION=1

fi


# This table determines the rotation (in 90 degree increments)

# based on the EXIF orientation tag and determines whether a

# coordinate transformation is needed.

case $ORIENTATION in

    (1)    ROT=0; FLIP=0;; # No rotation or flip needed.

    (2)    ROT=0; FLIP=1;; # Flip horizontal.

    (3)    ROT=2; FLIP=0;; # Rotate 180, no flip.

    (4)    ROT=2; FLIP=1;; # Rotate 180, flip.

    (5)    ROT=3; FLIP=1;; # Rotate 270, flip.

    (6)    ROT=1; FLIP=0;; # Rotate 90, no flip.

    (7)    ROT=1; FLIP=1;; # Rotate 90, flip.

    (8)    ROT=3; FLIP=0;; # Rotate 270, no flip.

    (*)    echo "BAD ORIENTATION $ORIENTATION" ; exit -1;;

esac


# Calculate the number of degrees to rotate the image

# based on the above table and the user-entered adjustment.

DEGREES="$(expr 90 '*' '(' $OFFSET '+' $ROT ')')"


# Generate the additional flags for sips if flipping is required.

FLIPSTR=""

if [ $FLIP = 1 ] ; then

    FLIPSTR="--flip horizontal"

else

    FLIPSTR=""

fi


# Perform the transformation.

$SIPS $FLIPSTR --rotate $DEGREES $INPUTFILE --out $OUTPUTFILE
```

```
# Delete the orientation keys so that sips and other tools

# won't get confused when doing auto-rotation.

$EXIFTOOL -Orientation= $OUTPUTFILE
```

Networking

Using SIGSTOP And SIGCONT To Manage Long-Lived Daemons

This trick prevents FTP servers on DSL connections from hopelessly clogging up the upstream link by using the `killall` command. It also traps Control-C and other likely signals so that if you break out of the script, the FTP processes are restarted correctly.

Listing D-7   Slowing down an FTP server

```
#!/bin/sh

SECONDS_TO_RUN=5
SECONDS_TO_PAUSE=20

handler() {
    killall -CONT ftpd
    exit 0
}

trap handler SIGHUP SIGTERM SIGQUIT SIGINT
# This must be run as root or the ftp user.
while true ; do
    killall -STOP ftpd
    sleep $SECONDS_TO_PAUSE
    killall -CONT ftpd
    sleep $SECONDS_TO_RUN
done
```

A Shell-Based Web Server

The Networking With Shell Scripts (page 216) section in Advanced Techniques (page 168) described how to write a simple daemon using netcat. It is possible to write remarkably complex daemons using this technique.

The first step in an HTTP daemon is parsing the initial request. For simple GET requests without query strings, this is fairy trivial. The following snippet takes the request line as an argument and sets global variables containing the request type, the URL, and the HTTP version.

```
parseRequest()
{
    local REQUEST="$(echo "$1" | tr -d '\r')"


    TYPE="$(echo "$REQUEST" | cut -f 1 -d ' ')"
    URL="$(echo "$REQUEST" | cut -f 2 -d ' ')"
    VERSION="$(echo "$REQUEST" | cut -f 3 -d ' ')"


    echo "GOT REQUEST: $REQUEST" 1>&2
}
```

Before you can actually interpret the request, however, you must split off the query string if it is there. For example, the URL `http://example.org/foo.cgi?bar` contains a host part (`example.org`), a path part (`/foo.cgi`), and a query string (`bar`). This code does not split off the host part because it is sent separately from the HTTP query string in HTTP/1.1 and is omitted entirely in HTTP/1.0.

```
splitURL()
{
        URL="$1"
        PATHPART="$(echo "$URL" | sed 's/?.*$//g')"
        local PATHLEN="$(strlen "$PATHPART")";
        local CUTPOS="$(expr "$PATHLEN" "+" "2")"


        PARMPART="$(echo "$URL" | cut -c "$CUTPOS-")"
}
```

Finally, you must parse the headers that the client sends so you can search for the `Host:` header to know what domain's contents to serve to the client (and to possibly send some of these headers back to the client). The first snippet reads the data from the client.

```
parseHeaders()
{
        local FD="$1"
        local TREENAME="$2"
        local HEADERLINE

        if [ "$TREENAME" = "" ] ; then
                TREENAME="HEADERTREE"
        fi

        # Creates a new tree head object with the specified name.
        newTree "$TREENAME"
        eval $TREENAME=\"\$\(getLastNodeName\)\"

        # echo "TN: $TREENAME" 1>&2

        # Reads headers from the specified file descriptor until
        # it gets a blank line, pasing each one to a parser..
        while true ; do
                eval read -u$FD HEADERLINE
                HEADERLINE="$(echo "$HEADERLINE" | tr -d '\r')"
                # echo "GOT HEADER LINE: \"$HEADERLINE\"" 1>&2

                if [ "$HEADERLINE" = "" ] ; then
                        # End of headers reached.
                        # echo "End of headers" 1>&2
                        break;
                fi

                addHeaderLine "$HEADERLINE" "$TREENAME"
        done
        LAST_TREE_NODE_INSERTED="$TREENAME"
```

```
}
```

The next part, addHeaderLine, trivially parses the header line by splitting the string on the first colon (:) character and stripping off any leading whitespace after it. Then, it calls another function to add it to the binary tree.

```
addHeaderLine()
{
        local HEADERLINE="$1"
        local TREE="$2"

        local FIELDNAME="$(echo "$HEADERLINE" | cut -f 1 -d ':')"
        local FIELDVALUE="$(echo "$HEADERLINE" | cut -f 2- -d ':' | \
            sed 's/^[[:space:]]//g')"

        addHeader "$FIELDNAME" "$FIELDVALUE" "$TREE"
}
```

The final snippet adds the header to a binary tree using the tree library described in Working with Binary Search Trees (page 287).

```
addHeader()
{
        local FIELDNAME="$1"
        local FIELDVALUE="$2"
        local TREE="$3"

        # echo "Inserting $FIELDNAME with value $FIELDVALUE into $TREE" 1>&2
        insertKey "$TREE" "$FIELDNAME"
        NODE="$(getLastNodeName)"
        setTreeField "$NODE" "Contents" "$FIELDVALUE"
}
```

All that remains is to tie the code together and actually handle the requests. To see the code in action, download the Companion Files zip archive associated with this document. (See the table of contents in the HTML version of this document at developer.apple.com.)

Within the Companion Files archive, you can find the sample at `scripts/BB_Starting_Points/networking/shttpd`.

This script requires a modified version of the OS X version of netcat that provides enhanced functionality and error recovery capabilities beyond what standard netcat versions provide. The Makefile (in the Companion Files archive) downloads, builds, and installs this modified version of netcat. The patch should also be easy to apply to the OpenBSD version of netcat.

> ⚠️ **Warning:** This script is not suitable for use in a production environment.

Text Manipulation

Data Management

Working with Binary Search Trees

Occasionally, it is useful to keep an array of dictionaries of key-value pairs and to be able to rapidly search through that array. Listing D-9 (page 290) provides such functionality in the form of a binary tree.

> **Note:** You can find the complete version of this script in the
> `BB_Starting_Points/networking/shttpd/shttpd/shttpd_subs` directory in the companion
> files archive.
>
> You can find complete reference documentation in the
> `BB_Starting_Points/networking/shttpd/shttpd_docs` directory in the companion files
> archive.

This binary tree library contains a number of key functions:

General tree functions:

`newTree(optional_tree_name)`
> Creates a new binary tree.

`deleteTree(tree_name)`
> Deletes a binary tree, freeing resources associated with it.

`iterateTree(tree_name, callback, call_on_root=0)`
> Iterates through a subtree, calling a function for each node.

`mergeTrees(source_tree_name, dest_tree_name)`
> Copies all of the keys in one tree into another. In the event of a collision for a given key, the new
> values take precedence.

Insertion Functions:

`insertKey(tree_name, key)`
> Inserts a new key into a binary tree using string comparisons.

`insertKeyNumeric(tree_name, key)`
> Inserts a new key into a binary tree using numerical comparisons.

`getLastNodeName()`
> Retrieves the last node inserted.

Node Functions:

`treeKey(node_name)`
> Retrieves the key associated with a node object.

`treeField(node_name, field_name)`
> Retrieves a field value for a node in the tree.

`setTreeField(node_name, field_name, new_value)`
> Sets a field value for a node in the tree.

Search Functions:

```
treeSearch(tree_name, key)
```
> Searches a binary tree for a given key using string comparisons.

```
treeSearchNumeric(tree_name, key)
```
> Searches a binary tree for a given key using numerical comparisons.

The following code demonstrates how to use this binary tree library:

Listing D-8   Binary tree example

```
# Tell the binary tree library to not run its tests.
DISABLE_TESTS=true
. binary_tree.sh


# Create a new binary tree and obtain its name.
newTree
TESTTREE="$(getLastNodeName)"


# Insert three nodes into the tree
# with keys 1, 3, and 7.
insertKeyNumeric "$TESTTREE" 3
insertKeyNumeric "$TESTTREE" 7
insertKeyNumeric "$TESTTREE" 1


# Add an attribute to the last node inserted (1)
ONENODE="$(getLastNodeName)"
setTreeField "$ONENODE" "MyFieldName" "42"


# Takes a node and prints the key value and
# the value of MyFieldName
echokeyandmyfieldname()
{
    echo "$(treeKey "$1") -> $(treeField "$1" "MyFieldName")"
}


# Iterate the tree in key order and call
```

```
# echokeyandmyfieldname on each node
iterateTree "$TESTTREE" "echokeyandmyfieldname"
```

Without further introduction, here is the binary tree code library. (The version in the companion files archive also includes some test code.)

Listing D-9   binary_tree.sh from shttpd

```sh
#!/bin/sh

# /*!
#       @header
#               A binary tree algorithm written in a shell script.  The main
#               functions of interest are {@link newTree}, {@link deleteTree},
#               {@link insertKey}, {@link insertKeyNumeric}, {@link treeSearch},
#               {@link treeSearchNumeric}, {@link iterateTree}, and
#               {@link mergeTrees}.
#
#               This is a minimal binary tree implementation that does not support
#               removing existing values from the tree once inserted.  However, such
#               functionality can be trivially retrofitted on top by adding or
#               clearing a "deleted" attribute on nodes using {@link setTreeField} if
#               desired.
#
#               To use this shell script, source it after setting DISABLE_TESTS to
#               "true".  To run tests, execute the script directly.
#   */

# /*! @group Global Variables
#               Variables used internally.  No user-serviceable parts inside.
#   */

# /*!
#       @abstract The starting object ID.  This is an internal counter.
#   */
```

```
OID=0

# /*!
#     @abstract A newline character.
#   */
NEWLINE="
"

# /*!
#     @abstract
#         Field separator.  Do not change.
#   */
IFS="$NEWLINE"

# /*! @group Node Functions
#         Functions that operate on a single node in the tree.
#   */

# /*!
#     @abstract Retrieves the key associated with a node object.
#     @result
#         Returns the key via <code>stdout</code>.
#     @param NODE
#         The node object.
#   */
treeKey()
{
        local NODE="$1"

        eval echo "\$$NODE"_KEY
}

# /*!
#     @abstract
#         Retrieves the left subtree for a node in the tree.
```

```
#     @result
#          Returns the node name of the left subtree via <code>stdout</code>.
#     @discussion
#          This is mainly an internal function, though you can use
#          it for debugging purposes.
#     @param NODE
#          The node object.
#  */
treeLeft()
{
     local NODE="$1"


     eval echo "\$$NODE"_LEFT
}


# /*!
#     @abstract
#          Sets the left subtree for a node in the tree.
#     @discussion
#          This is an internal function.  Do not call it directly.  Use
#          {@link insertKey} or {@link insertKeyNumeric} instead.
#     @param NODE
#          The node object.
#     @param VAL
#          The new left value.
#  */
setTreeLeft()
{
     local NODE="$1"
     local VAL="$2"


     eval "$NODE"_LEFT=\"$VAL\"
}
```

```
# /*!
#     @abstract
#         Retrieves the right subtree for a node in the tree.
#     @result
#         Returns the node name of the right subtree via <code>stdout</code>.
#     @discussion
#         This is mainly an internal function, though you can use
#         it for debugging purposes.
#     @param NODE
#         The node object.
# */
treeRight()
{
        local NODE="$1"

        eval echo "\$$NODE"_RIGHT
}


# /*!
#     @abstract
#         Sets the right subtree for a node in the tree.
#     @discussion
#         This is an internal function.  Do not call it directly.  Use
#         {@link insertKey} or {@link insertKeyNumeric} instead.
#     @param NODE
#         The node object.
#     @param VAL
#         The new right value.
# */
setTreeRight()
{
        local NODE="$1"
        local VAL="$2"

        eval "$NODE"_RIGHT=\"$VAL\"
```

```
}

# /*!
#     @abstract
#         Retrieves a field value for a node in the tree.
#     @result
#         Returns the requested field value via <code>stdout</code> or
#         an empty string.
#     @seealso setTreeField
#     @param NODE
#         The node object.
#     @param FIELDNAME
#         The field name.
#  */
treeField()
{
        local NODE="$1"
        local FIELDNAME="$2"

        eval echo "\$$NODE"_DATAFIELD_"$FIELDNAME"
}

# /*!
#     @abstract
#         Sets a field value for a node in the tree.
#     @discussion
#         This function allows you to store arbitrary attributes in a tree node.
#         If a value already exists for the specified field name, the value is
#         overwritten.
#     @param NODE
#         The node object.
#     @param FIELDNAME
#         The field name.
#     @param VAL
```

```
#          The new field value.
#   */
setTreeField()
{
        local NODE="$1"
        local FIELDNAME="$2"
        local VAL="$3"


        eval "$NODE"_DATAFIELD_"$FIELDNAME"=\"$VAL\"
        local DATAFIELDS="$(eval echo "\$$NODE"_DATAFIELDS)"
        eval "$NODE"_DATAFIELDS="\"$DATAFIELDS$NEWLINE$FIELDNAME\""
}



# /*! @group General Tree Functions
#          Operations that create, delete, iterate, and merge trees.
#   */


# /*!
#      @abstract
#          Iterates through a subtree, calling a function for each node.
#      @discussion
#          For each node in the tree (in sorted order), the function
#          specified by ACTION is called with a single parameter
#          containing the node name of the node being traversed.
#      @param TREE
#          The tree to traverse.
#      @param ACTION
#          The function to call on each node.
#      @param CALLONROOT
#          Set to 1 if you want to also call ACTION on the (bogus) root node.
#          This is usually only set for debug printing purposes.
#   */
iterateTree()
{
```

```
        local TREE="$1"

        local ACTION="$2"

        local CALLONROOT="$3"


        # echo "NAME IS $TREE"


        if [ "$CALLONROOT" = "1" ] ; then

                eval "$ACTION" "$TREE"

        fi


        iterateSubtree "$(treeLeft "$TREE")" "$ACTION"

}


# /*!
#     @abstract
#         Copies all of the keys in one tree into another.
#     @discussion
#         For each key in TREE_SRC, an equivalent key is
#         inserted in TREE_DST, including any field values
#         associated with it.  In the event of a collision
#         for a given key, the resulting set of field values
#         for that key is the union of the two sets of field
#         values, with the new values from TREE_SRC taking
#         precedence.
#     @param TREE_SRC
#         The source tree to copy.
#     @param TREE_DST
#         The destination tree into which the source tree is copied.
#  */
mergeTrees()
{
        local TREE_SRC="$1"
        local TREE_DST="$2"
```

```
        # echo "Here SRC: $TREE_SRC (left is $(treeLeft "$TREE_SRC"))" 1>&2

        # echo "     DST: $TREE_DST" 1>&2


        iterateSubtree "$(treeLeft "$TREE_SRC")" reinsert
}


# /*!
#     @abstract
#         Deletes a binary tree.
#     @param TREE
#         The name of the tree to delete.
#  */
deleteTree()
{
        local TREE="$1"


        if [ "$TREE" = "" ] ; then
                return;
        fi


        deleteTree "$(treeLeft "$TREE")"
        deleteTree "$(treeRight "$TREE")"
        deleteNode "$TREE"
}


# /*!
#     @abstract
#         Creates a new binary tree.
#     @result
#         Obtain the name of the tree using {@link getLastNodeName}.
#     @param TREE
#         The name of the tree to create.
#  */
newTree()
{
```

```
        local TREE="$1"


        newTreeNode "" "" "" "$TREE"
}


# /*! @group Search Functions
#         Functions used for searching for a key in a tree.  Be sure to
#         choose whether you want to use numerical or string key comparisons
#         for the search and choose the appropriate function accordingly.
#         The comparison type usde for searching must match the comparison
#         type used during insertion or the results are undefined.
#  */


# /*!
#     @abstract
#         Searches a binary tree for a given key.
#     @discussion
#         This tree search uses string comparisons.  You must use
#         {@link insertKey} with this function (and not
#         {@link insertKeyNumeric}.  For numeric searches, use
#         {@link treeSearchNumeric}.
#     @result
#         Returns the node name of the matching node through <code>stdout</code>
#         if found or an empty string otherwise.
#     @param TREE
#         The tree to search.
#     @param KEY
#         The key to search for.
#  */
treeSearch()
{
        local TREE="$1"
        local KEY="$2"
```

```
        subtreeSearch "$(treeLeft "$TREE")" "$KEY"

}


# /*!
#     @abstract
#         Searches a binary tree for a given key.
#     @result
#         Returns the node name of the matching node through <code>stdout</code>
#         if found or an empty string otherwise.
#     @discussion
#         This tree search uses numeric comparisons.  You must use
#         {@link insertKeyNumeric} with this function (and not
#         {@link insertKey}.  For string searches, use {@link treeSearch}.
#     @param TREE
#         The tree to search.
#     @param KEY
#         The key to search for.
#   */
treeSearchNumeric()
{
        local TREE="$1"
        local KEY="$2"


        subtreeSearchNumeric "$(treeLeft "$TREE")" "$KEY"
}


# /*! @group Insertion Functions
#         Functions used for inserting a key into a tree.  Be sure to
#         choose whether you want to use numerical or string key comparisons
#         during insertion and choose the appropriate function accordingly.
#
#         After inserting, you can use {@link getLastNodeName} to get the
#         node name of the resulting node if desired.
#   */
```

```
# /*!
#     @abstract
#         Retrieves the last node inserted.
#     @result
#         Returns the node name of the last node inserted via
#         <code>stdout</code>.
#     @discussion
#         After creating a new node with {@link insertKey} or a
#         new tree with {@link newTree}, call this to obtain its
#         note ID.
#  */
getLastNodeName()
{
        echo "$LAST_TREE_NODE_INSERTED"
}


# /*!
#     @abstract
#         Inserts a new key into a binary tree.
#     @discussion
#         If a node already exists with this value, the
#         existing node is returned.
#
#         This tree insertion uses string comparisons.  You must use
#         {@link treeSearch} with this function (and not
#         {@link treeSearchNumeric}.  For numeric searches, use
#         {@link insertKeyNumeric}.
#     @result
#         Obtain the node name of the newly created node using
#         {@link getLastNodeName}.
#     @param TREE
#         The name of the binary tree.
#     @param KEY
#         The key to insert.
```

```
#   */

insertKey()

{

        local TREE="$1"

        local KEY="$2"


        local LASTTREE="$TREE"

        local DIRECTION="LEFT"


        while [ "$TREE" != "" -a "$LASTTREE" != "" ] ; do

                if [ $DIRECTION = "LEFT" ] ; then

                        TREE="$(treeLeft "$TREE")"

                else

                        TREE="$(treeRight "$TREE")"

                fi

                local TREEKEY="$(treeKey "$TREE")"


                if [ "$TREE" != "" ] ; then

                        if [ "$KEY" \< "$TREEKEY" ] ; then

                                DIRECTION="LEFT"

                                LASTTREE="$TREE"

                        elif [ "$KEY" \> "$TREEKEY" ] ; then

                                DIRECTION="RIGHT"

                                LASTTREE="$TREE"

                        else

                                # Matching node already exists.  Return its name.

                                LAST_TREE_NODE_INSERTED="$NODE"

                                return

                        fi

                fi

        done

        newTreeNode "" "" "$KEY"

        local NODE="$(getLastNodeName)"


        if [ $DIRECTION = "LEFT" ] ; then
```

```
                setTreeLeft "$LASTTREE" "$NODE"

        else

                setTreeRight "$LASTTREE" "$NODE"

        fi

}


# /*!

#     @abstract

#         Inserts a new key into a binary tree.

#     @discussion

#         If a node already exists with this value, the

#         existing node is returned.

#

#         This tree insertion uses string comparisons.  You must use

#         {@link treeSearch} with this function (and not

#         {@link treeSearchNumeric}.  For numeric searches, use

#         {@link insertKeyNumeric}.

#     @result

#         Obtain the node name of the newly created node using

#         {@link getLastNodeName}.

#     @param TREE

#         The name of the binary tree.

#     @param KEY

#         The key to insert.

# */

insertKeyNumeric()

{

        local TREE="$1"

        local KEY="$2"


        # echo "IN INSNUM"


        local LASTTREE="$TREE"

        local DIRECTION="LEFT"
```

```
        while [ "$TREE" != "" -a "$LASTTREE" != "" ] ; do
                if [ $DIRECTION = "LEFT" ] ; then
                        TREE="$(treeLeft "$TREE")"
                else
                        TREE="$(treeRight "$TREE")"
                fi
                local TREEKEY="$(treeKey "$TREE")"


                if [ "$TREE" != "" ] ; then
                        if [ "$KEY" -lt "$TREEKEY" ] ; then
                                DIRECTION="LEFT"
                                LASTTREE="$TREE"
                        elif [ "$KEY" -gt "$TREEKEY" ] ; then
                                DIRECTION="RIGHT"
                                LASTTREE="$TREE"
                        else
                                # Matching node already exists.  Return its name.
                                LAST_TREE_NODE_INSERTED="$NODE"
                                return
                        fi
                fi
        done
        newTreeNode "" "" "$KEY"
        local NODE="$(getLastNodeName)"


        if [ $DIRECTION = "LEFT" ] ; then
                setTreeLeft "$LASTTREE" "$NODE"
        else
                setTreeRight "$LASTTREE" "$NODE"
        fi
}


# /*! @group Debug Functions
```

```
#       Functions that print debug information about binary trees,
#       tree nodes, and so on.
#  */

# /*!
#     @abstract
#         Prints a node structure for debugging purposes.
#     @param NODE
#         The node to print.
#  */
printNode()
{
        local NODE="$1"

        echo "NAME:  $NODE"
        echo "KEY:   $(treeKey "$NODE")"
        echo "LEFT:  $(treeLeft "$NODE")"
        echo "RIGHT: $(treeRight "$NODE")"
        echo "DATA:"

        local DATAFIELDS="$(eval echo "\$$NODE"_DATAFIELDS)"
        local FIELDNAME
        for FIELDNAME in $DATAFIELDS ; do
                # Skip the empty first field.
                if [ "$FIELDNAME" != "" ] ; then
                        eval echo "    $NODE""_DATAFIELD_$FIELDNAME"":" \
                                "\$$NODE""_DATAFIELD_$FIELDNAME"
                fi
        done
        echo "-=-=-=-=-=-=-=-=-=-"
}

# /*!
#     @abstract
```

```
#          Prints out the contents of a tree for debugging purposes.
#   */
printTree()
{
        local TREE="$1"


        # echo "NAME IS $TREE"
        iterateTree "$TREE" "printNode" 1
}


# /*!
#       @abstract
#          Prints a line of text in red letters.
#   */
echored()
{
        printf "\e[1;31m%s\e[0;30m\n" $@
}


# /*!
#       @abstract
#          Prints a line of text in green letters.
#   */
echogreen()
{
        printf "\e[1;32m%s\e[0;30m\n" $@
}


# /*!
#       @abstract
#          Prints a line of text in blue letters.
#   */
echoblue()
{
        printf "\e[1;34m%s\e[0;30m\n" $@
```

```
}


# /*! @group Internal Functions
#         No user-serviceable parts inside.  These functions are used
#         internally by the other functions and should generally not
#         be called from outside unless you really know what you are
#         doing.
#  */


# /*!
#     @abstract
#         Iterates through a subtree, calling a function for each node.
#     @discussion
#         Do not call this directly.  Call {@link iterateTree} instead.
#  */
iterateSubtree()
{
        local TREE="$1"
        local ACTION="$2"


        if [ "$TREE" = "" ] ; then
                return;
        fi
        # echo "IN IST: TREE $TREE" 1>&2


        iterateSubtree "$(treeLeft "$TREE")" "$ACTION"
        eval "$ACTION $TREE"
        iterateSubtree "$(treeRight "$TREE")" "$ACTION"


}


# /*!
#     @abstract
```

```
#        Internal helper function.
#     @discussion
#        This function is used by {@link mergeTrees} to take a node from
#        one tree and duplicte it in another.
# */
reinsert()
{
        local NODE="$1"

        # echo "GOT NODE \"$NODE\"" 1>&2

        # echo "TREE_DST: $TREE_DST" 1>&2

        if [ "$NODE" = "" ] ; then
                return;
        fi

        local VAL="$(treeKey "$NODE")"

        if [ "$VAL" = "" ] ; then
                return;
        fi

        # local NEWNODE="$(treeSearch "$TREE_DST" "$VAL")"
        # echo "NN1: $NEWNODE"

        insertKey "$TREE_DST" "$VAL"
        local NEWNODE="$(getLastNodeName)"

        # print "NN: $NEWNODE" 1>&2

        local DATAFIELDS="$(eval echo "\$$NODE"_DATAFIELDS)"
        local FIELDNAME

        for FIELDNAME in $DATAFIELDS ; do
```

```
                    # Skip the empty first field.

                    if [ "$FIELDNAME" != "" ] ; then

                            # eval echo setting
"$NEWNODE""_DATAFIELD_$FIELDNAME""=\"\$$NODE""_DATAFIELD_$FIELDNAME\"" 1>&2

                            eval "$NEWNODE""_DATAFIELD_$FIELDNAME""=\
                                    \"\$$NODE""_DATAFIELD_$FIELDNAME\""

                    fi

            done


            # printNode "$NODE"
}


# /*!
#     @abstract
#         Creates a new node in the tree.
#     @discussion
#         This is an internal function.  Do not call it directly.  Use
#         {@link insertKey} or {@link insertKeyNumeric} instead.
#     @param LEFT
#         The initial left value for the node (usually empty).
#     @param RIGHT
#         The initial right value for the node (usually empty).
#     @param KEY
#         The key for the new node.
#     @param TREE
#         The desired name for the node (usually empty).
#  */
newTreeNode()
{
        local LEFT="$1"
        local RIGHT="$2"
        local KEY="$3"
        local TREE="$4"


        if [ "$TREE" = "" ] ; then
```

```
            TREE="TREENODE_$OID"

            OID="$(expr "$OID" "+" "1")"

            # echo "$TREE"

    # else

            # echo "Using explicit name \"$TREE\"" 1>&2

    fi


    eval "$TREE"_LEFT=\"$LEFT\"

    eval "$TREE"_RIGHT=\"$RIGHT\"

    eval "$TREE"_KEY=\"$KEY\"

    LAST_TREE_NODE_INSERTED="$TREE"
}



# /*!
#     @abstract
#         Searches a binary tree for a given key.
#     @discussion
#         This is an internal function.  Do not call it directly.  Use
#         {@link treeSearch} instead.
#     @result
#         Returns the node name of the matching node through <code>stdout</code>
#         if found or an empty string otherwise.
#     @param TREE
#         The subtree to search.
#     @param KEY
#         The key to search for.
# */
subtreeSearch()
{
    local TREE="$1"
    local KEY="$2"

    if [ "$TREE" = "" ] ; then
            return;
```

```
        fi


        local TREEKEY="$(treeKey "$TREE")"


        if [ "$KEY" \< "$TREEKEY" ] ; then
                subtreeSearch "$(treeLeft "$TREE")" "$KEY"
        elif [ "$KEY" \> "$TREEKEY" ] ; then
                subtreeSearch "$(treeRight "$TREE")" "$KEY"
        else
                echo $TREE
        fi
}


# /*!
#     @abstract
#         Searches a binary tree for a given key.
#     @discussion
#         This is an internal function.  Do not call it directly.  Use
#         {@link treeSearch} instead.
#     @result
#         Returns the node name of the matching node through <code>stdout</code>
#         if found or an empty string otherwise.
#     @param TREE
#         The subtree to search.
#     @param KEY
#         The key to search for.
#  */
subtreeSearchNumeric()
{
        local TREE="$1"
        local KEY="$2"


        if [ "$TREE" = "" ] ; then
                return;
```

```
        fi


        local TREEKEY="$(treeKey "$TREE")"


        if [ "$KEY" -lt "$TREEKEY" ] ; then
                subtreeSearchNumeric "$(treeLeft "$TREE")" "$KEY"
        elif [ "$KEY" -gt "$TREEKEY" ] ; then
                subtreeSearchNumeric "$(treeRight "$TREE")" "$KEY"
        else
                echo $TREE
        fi
}


# /*!
#     @abstract
#         Deletes a node in a tree.
#     @discussion
#         This algorithm does not support deleting arbitrry nodes.
#         This is an internal function that is used by {@link deleteTree}.
#     @param NODE
#         The node to delete.
#  */
deleteNode()
{
        local NODE="$1"


        local DATAFIELDS="$(eval echo "\$$NODE"_DATAFIELDS)"


        local FIELDNAME
        for FIELDNAME in $DATAFIELDS ; do
                # Skip the empty first field.
                if [ "$FIELDNAME" != "" ] ; then
                        eval unset "$NODE"_DATAFIELD_$FIELDNAME
                fi
        done
```

```
        eval unset "$NODE"_LEFT

        eval unset "$NODE"_RIGHT

}
```

User and Group Management

OS X provides significant GUI tools for managing users and groups. Sometimes, however, you may need to do things the hard way (from the command line). For the occasional hand addition, you can manually add a user or group using the `dscl` (directory service command line) tool. However, if you regularly need to add users, it can be advantageous to script the task.

The code listings here (which are also included in the Companion Files archive) show how to create a new user and a new group, including choosing unused user and group IDs.

Listing D-10   Script for adding a new user using dscl (adduser.sh)

```
#!/bin/sh


# Usage:
#
# adduser [−a] <USERNAME> <LONGNAME> <PRIMARY_GID> [ <HOME_DIRECTORY> [ <UID> ]]
#
# −a: Make the user an admin user.
# USERNAME: The OS X "short name", e.g. jdoe
# LONGNAME: The OS X "real name", e.g. "John Doe"
# PRIMARY_GID: The primary group ID.
# HOME_DIRECTORY: The user's home directory.  Leave blank to use /Users/username.
#                 The script attempts to create this directory if it does not
exist.A
# UID: The user ID for the new user.  Leave blank for the script to automatically
#      choose the first unused ID at or above MINUID (currently 501).


ADMIN="user"
if [ "$1" = "−a" ] ; then
    ADMIN="admin user"
    shift
```

```
 fi

USERNAME="$1"
LONGNAME="$2"
PRIMARY_GID="$3"
HOMEDIR="$4" # Optional
NEWUID="$5" # Optional

MINUID=501
DOMAIN="."

# Must have newline here.
IFS="
"

# /*!
#     @abstract Checks to see if a long name is reasonable.
#     @discussion Ideally, this should do more checks.
#  */
valid_username()
{
    local NAME="$1"

    if [ "$NAME" = "" ] ; then
        return 1;
    fi

    return 0;
}

# /*!
#     @abstract Checks to see if a long name is reasonable.
#     @discussion
#         Checking for non-empty strings is good enough for now,
#         but ideally, this should also check for duplicates.
```

```
#          The code doesn't do this because there's no good way
#          that doesn't involve a huge file and grep.
#  */
valid_longname()
{
    local NAME="$1"


    if [ "$NAME" = "" ] ; then
        return 1;
    fi
    return 0
}


# /*!
#      @abstract Checks to see if a (numeric) group ID is reasonable.
#  */
valid_gid()
{
    local NEWGID="$1"


    # Empty primary GID is illegal.
    if [ "$NEWGID" = "" ] ; then
        return 1;
    fi


    local NEWGIDSTR="$(printf "%d" "$NEWGID" 2> /dev/null)"


    if [ "$NEWGIDSTR" != "$NEWGID" ] ; then
        return 1;
    fi


    return 0;
}
```

```
# /*!
#     @abstract Checks to see if a (numeric) user ID is reasonable.
#  */
valid_uid()
{
    local NEWUID="$1"


    # Empty UID means "choose one for me"
    if [ "$NEWUID" = "" ] ; then
        return 0;
    fi


    local NEWUIDSTR="$(printf "%d" "$NEWUID" 2> /dev/null)"


    if [ "$NEWUIDSTR" != "$NEWUID" ] ; then
        return 1;
    fi


    return 0;
}


# /*!
#     @abstract Creates an associative pseudo-array for UID to username mapping.
#  */
initUIDMap()
{
    local SKIPUSER="$1"


    local USERS="$(dscl "$DOMAIN" -list /Users)"


    for i in $USERS ; do
        if [ "$i" != "$SKIPUSER" ] ; then
            eval "UID_$(dscl "$DOMAIN" -read /Users/"$i" UniqueID 2>/dev/null |
sed 's/UniqueID: //' | sed 's/-/MINUS/')=\"$i\""
        fi
```

```
    done
}


# /*!
#    @abstract Looks up a UID in the pseudo-array and maps it to a username
#  */
uidToName()
{
    local CHECKUID="$1"


    local CHECKUID_ENCODED="$(echo "$CHECKUID" | sed 's/-/MINUS/')"



    eval echo '$UID_'$CHECKUID_ENCODED
}


# /*!
#    @abstract Finds the next unused UID.
#  */
assignUID()
{
    initUIDMap


    # An error here means somebody screwed up MINUID.
    local POS=$MINUID


    while true ; do
        # echo "Trying $POS" 1>&2
        local TEMPNAME="$(uidToName $POS)"
        if [ "$TEMPNAME" = "" ] ; then
            echo $POS
            return;
        fi
        POS="$(expr $POS '+' 1)"
```

```
    done

}


# /*!
#     @abstract Returns success if no other user has the chosen UID.
#  */
uid_not_conflicting()
{
    local NEWUID="$1"
    local NEWUSER="$2"

    initUIDMap "$NEWUSER"

    local TEMPNAME="$(uidToName "$NEWUID")"

    if [ "$TEMPNAME" != "" ] ; then
        return 1;
    fi

    return 0
}



while ! valid_username "$USERNAME" ; do
    printf "Enter username: "
    read USERNAME
done


while ! valid_uid  "$NEWUID" ; do
    printf "Invalid UID specified.  Enter desired UID: "
    read NEWUID
done


while ! valid_gid  "$PRIMARY_GID" ; do
    printf "Invalid group ID specified.  Enter desired GID: "
```

```
    read PRIMARY_GID
done


while ! valid_longname "$LONGNAME" ; do
    printf "Invalid long name specified.  Enter desired long name: "
    read LONGNAME
done



# Test code
### echo "UID Conflict check:"
### uid_not_conflicting "501" "dg" # Test this first or else.
### echo "$? should be 0"
### uid_not_conflicting "501" "Schlomo"
### echo "$? should be 1"


### echo "First free UID is $(assignUID)"

dscl $DOMAIN -read /Users/"$USERNAME" > /dev/null 2>&1
if [ $? = 0 ] ; then
    echo "Failed.  A user with that name already exists.." 1>&2
    exit -1
fi


dscl $DOMAIN -create /Users/"$USERNAME"


if [ $? != 0 ] ; then
    echo "Failed.  User could not be created." 1>&2
    exit -1
fi


dscl $DOMAIN -create /Users/"$USERNAME" UserShell /bin/bash
dscl $DOMAIN -create /Users/"$USERNAME" RealName "$LONGNAME"
if [ "$NEWUID" = "" ] ; then
```

```
    NEWUID="$(assignUID)"
 fi
 dscl $DOMAIN -create /Users/"$USERNAME" UniqueID $NEWUID


 while ! uid_not_conflicting "$NEWUID" "$USERNAME"; do
     echo "A user with ID $NEWUID exists already.  Assigning a new UID." 1>&2
     OLDUID="$NEWUID"
     NEWUID="$(assignUID)"
     dscl $DOMAIN -change /Users/"$USERNAME" UniqueID "$OLDUID" "$NEWUID"
 done


 dscl $DOMAIN -create /Users/"$USERNAME" PrimaryGroupID $PRIMARY_GID


 if [ "$HOMEDIR" = "" ] ; then
     dscl $DOMAIN -create /Users/"$USERNAME" NFSHomeDirectory /Users/"$USERNAME"
     if [ ! -d "/Users/$USERNAME" ] ; then
         mkdir "/Users/$USERNAME"
     fi
 else
     dscl $DOMAIN -create /Users/"$USERNAME" NFSHomeDirectory "$HOMEDIR";
 fi


 dscl $DOMAIN -passwd /Users/"$USERNAME" "*"
 # passwd "$USERNAME"


 UUID="$(/usr/bin/uuidgen)"
 dscl $DOMAIN -create /Users/"$USERNAME" GeneratedUID "$UUID"


 if [ "$ADMIN" = "admin user" ] ; then
     dscl $DOMAIN -append /Groups/admin GroupMembership "$USERNAME"
     dscl $DOMAIN -append /Groups/admin GroupMembers "$UUID"
 fi


 echo "Added $ADMIN $USERNAME with ID $NEWUID and UID $UUID.  Please remember to
 set a password for the user."
```

Listing D-11   Script for adding a new group using dscl (addgroup.sh)

```sh
#!/bin/sh


# Usage:
#
# addgroup <GROUPNAME> <LONGNAME> [<GID> ]
#
# GROUPNAME: The OS X "short name", e.g. admin
# LONGNAME: The OS X "real name", e.g. "Administrators"
# GID: The group ID for the new group.  Leave blank for the script to automatically
#       choose the first unused ID at or above MINGID (currently 501).
#


GROUPNAME="$1"
LONGNAME="$2"
NEWGID="$3" # Optional


MINGID=501


DOMAIN="."


# Must have newline here.
IFS="
"


ADDGROUP="./addgroup.sh"


if [ -f "/usr/local/bin/addgroup" ] ; then
    ADDGROUP="/usr/local/bin/addgroup"
fi
```

```
# /*!
#     @abstract Checks to see if a group long name is reasonable.
#     @discussion
#         Checking for non-empty strings is good enough for now,
#         but ideally, this should also check for duplicates.
#         The code doesn't do this because there's no good way
#         that doesn't involve a huge file and grep.
# */
valid_longname()
{
    local NAME="$1"

    if [ "$NAME" = "" ] ; then
        return 1;
    fi

    return 0;
}


# /*!
#     @abstract Checks to see if a group name is reasonable.
#     @discussion Ideally, this should do more checks.
# */
valid_groupname()
{
    local NAME="$1"

    if [ "$NAME" = "" ] ; then
        return 1;
    fi
    return 0
}



# /*!
```

```
#      @abstract Checks to see if a (numeric) group ID is reasonable.
#  */
valid_gid()
{
    local NEWGID="$1"


    # Empty primary GID means "choose one for me"
    if [ "$NEWGID" = "" ] ; then
        return 0;
    fi


    local NEWGIDSTR="$(printf "%d" "$NEWGID" 2> /dev/null)"


    if [ "$NEWGIDSTR" != "$NEWGID" ] ; then
        return 1;
    fi


    return 0;
}



# /*!
#      @abstract Creates an associative pseudo-array for GID to username mapping.
#  */
initGIDMap()
{
    local SKIPGROUP="$1"


    # GROUPS is BASH reserved word
    local ALLGROUPS="$(dscl "$DOMAIN" -list /Groups)"


    for i in $ALLGROUPS ; do
        if [ "$i" != "$SKIPGROUP" ] ; then
            eval "GID_$(dscl "$DOMAIN" -read /Groups/"$i" PrimaryGroupID 2>/dev/null
 | sed 's/PrimaryGroupID: //' | sed 's/-/MINUS/')=\"$i\""
```

```
        fi

    done

}



# /*!

#     @abstract Looks up a GID in the pseudo-array and maps it to a group name

#  */

gidToName()

{

    local CHECKGID="$1"


    local CHECKGID_ENCODED="$(echo "$CHECKGID" | sed 's/-/MINUS/')"



    eval echo '$GID_'$CHECKGID_ENCODED

}


# /*!

#     @abstract Finds the next unused UID.

#  */

assignGID()

{

    initGIDMap


    # An error here means somebody screwed up MINGID.

    local POS=$MINGID


    while true ; do

        # echo "Trying $POS" 1>&2

        local TEMPNAME="$(gidToName $POS)"

        if [ "$TEMPNAME" = "" ] ; then

            echo $POS

            return;

        fi
```

```
        POS="$(expr $POS '+' 1)"
    done
}


# /*!
#     @abstract Returns success if no other group has the chosen GID.
#  */
gid_not_conflicting()
{
    local NEWGID="$1"
    local NEWGROUP="$2"

    initGIDMap "$NEWGROUP"

    local TEMPNAME="$(gidToName "$NEWGID")"

    if [ "$TEMPNAME" != "" ] ; then
        return 1;
    fi

    return 0
}



while ! valid_groupname "$GROUPNAME" ; do
    printf "Enter group name: "
    read GROUPNAME
done

while ! valid_gid  "$NEWGID" ; do
    printf "Invalid or no group ID specified.  Enter desired GID: "
    read NEWGID
done
```

```
while ! valid_longname "$LONGNAME" ; do

    printf "Invalid long name specified.  Enter desired long name: "

    read LONGNAME

done



# Test code

# echo "GID Conflict check:"

# gid_not_conflicting "80" "admin" # Test this first or else.

# echo "$? should be 0"

# gid_not_conflicting "80" "Schlomo"

# echo "$? should be 1"


echo "First free GID is $(assignGID)"


dscl $DOMAIN —read /Groups/"$GROUPNAME" > /dev/null 2>&1

if [ $? = 0 ] ; then

    echo "Failed.  A group with that name already exists.." 1>&2

    exit —1

fi


dscl $DOMAIN —create /Groups/"$GROUPNAME"


if [ $? != 0 ] ; then

    echo "Failed.  Group could not be created." 1>&2

    exit —1

fi


dscl $DOMAIN —create /Groups/"$GROUPNAME" RealName "$LONGNAME"

if [ "$NEWGID" = "" ] ; then

    NEWGID="$(assignGID)"

fi

dscl $DOMAIN —create /Groups/"$GROUPNAME" PrimaryGroupID $NEWGID


while ! gid_not_conflicting "$NEWGID" "$GROUPNAME"; do
```

```
    echo "A user with ID $NEWGID exists already.  Assigning a new GID." 1>&2

    OLDGID="$NEWGID"

    NEWGID="$(assignGID)"

    dscl $DOMAIN -change /Groups/"$GROUPNAME" PrimaryGroupID "$OLDGID" "$NEWGID"
done


UUID="$(/usr/bin/uuidgen)"

dscl $DOMAIN -create /Groups/"$GROUPNAME" GeneratedUID "$UUID";


# Legacy UNIX group password

dscl $DOMAIN -create /Groups/"$GROUPNAME" Password "*"


echo "Added $GROUPNAME with ID $NEWGID and UUID $UUID.  Please remember to set a
password for the user."
```

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 ffl/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 328).

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 subroutine later
XVAL1=$(ord "$RAWVAL2") # more on this subroutine later
YVAL0=$(ord "$RAWVAL3") # more on this subroutine later
YVAL1=$(ord "$RAWVAL4") # more on this subroutine 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. (See the `perl` manual page for more information about Perl.)

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` subroutine.

```
YVAL1=$(echo $RAWVAL4 | perl -e '$/ = undef; my $val = <STDIN>; print ord($val);')
```

Finding Ordinal Rank Using AWK

The second 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. (See the `awk` manual page and How AWK-ward (page 122) for more information about AWK.) 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. It 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 328), 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 E-1   An Integer to Octal Conversion subroutine

```
# 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 a string containing a list of all of the possible ASCII characters except NULL (character 0) in order. This subroutine is called only once at program initialization; the shell version of this code is very slow as it is, and calling this subroutine each time you try to find the ordinal rank of a character would make this code completely unusable.

```
# Initializer for the scary shell ord subroutine.
ord_init()
{
    I=1
    ORDSTRING=""
    while [ $I -lt 256 ] ; do
        # local HEX=$(inttohex $I);
        local OCT=$(inttooct $I);
        # The following should work with GNU sed, but
        # OS X's sed doesn't support \x.
        # local CH=$(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 the `sed` implementation in OS X. 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 subroutine shown in Listing E-1 (page 329).

When this subroutine 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.

> ⚠️ **Warning:** Beginning in OS X v10.5, the `sed` command requires that its input strings contain only valid character sequences in the character set specified by your locale settings. The default character set is UTF-8.
>
> The raw streams of bytes used in this subroutine are not guaranteed to be a valid UTF-8 text sequence. As a result, with the default locale settings, this subroutine produces errors whenever it encounters most characters with values greater than 127 (high ASCII characters).
>
> To disable these `sed` constraints, your script *must* override the standard locale. To do this, add the following line near the top of the script:
>
> ```
> export LANG="C"
> ```
>
> This sets the locale to "C", a locale in which no multibyte character sequences exist and each character is treated as a raw byte for comparison purposes (sorting is in raw numeric order, and so on).
>
> See the `locale` manual page for more information about locales.

```
ord()
{
    local CH="$1"
    local STRING=""
    local OCCOPY=$ORDSTRING
    local COUNT=0;
```

```
    # Some shells can't handle NULL characters,
    # so this code gets an empty argument.
    if [ "x$CH" = "x" ] ; then
        echo 0
        return
    fi


    # 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 [ $CONT = 1 ]; do
        # Copy the string so we know if we've stopped finding
        # nonmatching characters.
        OCTEMP="$OCCOPY"


        # echo "CH WAS $CH"
        # echo "ORDSTRING: $ORDSTRING"


        # 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` subroutine 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 OS X Developer Library website.

```
#!/bin/sh


ITERATIONS=1000
SCALE=6


# Prevent sed from caring about high ASCII characters not
# being valid UTF-8 sequences
export LANG="C"


# Set FAST to "slow", "medium", or "fast".  This controls
```

```
# which ord() subroutine 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 subroutine.
ord_init()
{
    I=1
    ORDSTRING=""
    while [ $I -lt 256 ] ; do
    # local HEX=$(inttohex $I);
    local OCT=$(inttooct $I);
    # The following should work with GNU sed, but
    # OS X's sed doesn't support \x.
    # local CH=$(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;


    # Some shells can't handle NULL characters,
    # so this code gets an empty argument.
    if [ "x$CH" = "x" ] ; then
        echo 0
        return
    fi

    # 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 [ $CONT = 1 ]; do
    # Copy the string so we know if we've stopped finding
    # nonmatching characters.
    OCTEMP="$OCCOPY"

    # echo "CH WAS $CH"
    # echo "ORDSTRING: $ORDSTRING"
```

```
    # 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 subroutine, 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 subroutine."
    ord_init


    # Test our ord subroutine
    echo "Testing ord subroutine"
```

```
    ORDOFA=$(ord "a")

    # That better be 97.

    if [ "$ORDOFA" != "97" ] ; then

        echo "Shell ord subroutine 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"
```

Historical Footnotes and Arcana

This appendix contains historical footnotes extracted from elsewhere in the document to improve readability. They appear in this appendix because although they may be of some interest, they are not critical to a general understanding of the subject.

Historical String Parsing

In some early Bourne-compatible shells, the second statement below does not do what you might initially suspect:

```
STRING1="This is a test"
STRING2=$STRING1
```

Most modern Bourne shells parse the right side of the assignment statement first (including any splitting on spaces), *then* expand the variable `$STRING1`, thus copying the complete value of `STRING1` into `STRING2`.

Note: This pre-splitting behavior is specific to the right side of assignment statements. All other statements are split *after* variables are expanded.

Some older shells, however, may do the space splitting after expanding the variable. Such shells interpret the second statement as though you had typed the following:

```
STRING2=This is a test
```

as a two-part statement: an assignment statement (`FIRST_ARGUMENT=This`) followed by a command (`is`) with two arguments (`a` and `test`).

Because there is no semicolon between the assignment and the command, the shell treats this assignment statement as an attempt to modify the environment passed to the `is` command (a technique described in Overriding Environment Variables for Child Processes (Bourne Shell) (page 30)). This is clearly not what you intended to do.

For maximum compatibility, you should always write such assignment statements like this:

```
STRING1="This is a test"
STRING2="$STRING1"
```

In any Bourne shell, this is interpreted correctly as:

```
STRING2="This is a test"
```

> **Compatibility Note:** This behavior was first introduced by `zsh` because this was a common programmer mistake that caused errors.
>
> When run as `/bin/sh`, some early versions of `zsh` emulate the previous Bourne shell behavior for compatibility. Thus, in a script that starts with `#!/bin/sh`, the statement may fail if `sh` is really `zsh`.
>
> Current versions of `zsh` obey the modern splitting rules even when run as `/bin/sh`.

Similarly, in modern shells, quotation marks and other special characters are parsed before expansion. Thus, quotation marks inside a variable do not affect the splitting behavior. For example:

```
FOO="\"this is\" a test"
ls $FOO
```

is equivalent to:

```
ls \"this
ls is\"
ls a
ls test
```

In older Bourne shells, however, this may be misinterpreted as:

```
ls "this is"
ls a
ls test
```

In general, it is not worth the effort to support shells with this broken splitting behavior, and it is unlikely that you will encounter them; the modern splitting behavior has been common since the mid-1990s.

Document Revision History

This table describes the changes to *Shell Scripting Primer* .

Date	Notes
2014-03-10	Updated to reflect the removal of support for the environment.plist file in OS X v10.8.
2013-08-08	Enhanced the Shell Script Security chapter.
2012-07-23	Added note about TextEdit in OS X v10.7.3.
2012-03-14	Incorporated Command Line Primer, fixed broken link, and fixed typographical errors.
2011-07-27	Fixed typos in CSH getopt example.
2011-06-21	Added more security information and reworded description of the OS X (Mach) console.
2011-01-11	Added information about alias permanence.
2010-10-01	Fixed links that pointed to the wrong section after reorganizing content. Fixed description of enclosing a single quote inside single quotes. Added description of Perl's rename command.
2010-09-01	Added "Starting Points" scripts for creating users and groups.
2010-06-18	Added chapter on shell script security.
2010-06-21	Added an AWK example and improved wording in a few spots.
2009-11-17	Made minor typographical fixes.
2009-10-19	Restructured document for easier access. Added an example for the nc (netcat) utility.

Date	Notes
2009-08-25	Added note about creating plain text files in TextEdit.
2009-08-11	Added chapter describing how to get to a shell prompt and pointing to Command Line Primer. Added an appendix of examples.
2009-07-23	Added content about line endings.
2009-06-01	Added information about using regular expressions in control statements.
2009-04-08	Added a forward link in the awk section. Added a few minor cross-platform porting notes. Added a CSH compatibility note about numeric comparisons.
2009-03-04	Added AppleScript/osascript section. Added portability notes for head and tail commands.
2009-01-06	Added index.
2008-11-19	Clarified text about C shell limitations, quoting arguments. Added additional cross-platform compatibility information.
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 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.

Date	Notes
2006-06-28	Fixed a number of typographical errors.
2006-05-23	First version.

Index