Chapter 3

Introduction to Shells and Scripting

3.1 What is a Shell?

For our purposes, a shell will be considered as a command-line interface that allows the user to input commands. These commands can range from seeing what's in a directory on the hard drive to listing currently running processes on the computer. In chapter 9 you will even see what it takes to build your own shell as you are given the project of creating one yourself. For now we will introduce what kinds of tasks, both simple and complex, that you can use a shell for. An example akin to what most people think of when they think of a shell is the windows command prompt. When you first open a terminal window for your shell in unix, you will likely see something like this:

```
data 49 $
```

Right now, the shell is waiting on your input. If we provide a command to the shell at this point, it will happily carry out our task to the best of its ability or tell us it has no clue what we are doing. An example would be if I wanted to see the contents of my home directory. I would run the command by typing “ls” into the shell.

```
data 49 $ ls
CS177     CS250      CS252_old  CS390      jmp-0.51
CS180     CS251      CS307      Desktop    scratch
CS240     CS252      CS348      Documents  tmp-message
CS240_old CS252_TA   CS354      workspace  xinu-spring2013-x86
data 50 $
```

As you can see, all of the contents of the current directory, both individual files and directories, have been printed to the terminal window after running the command, and the shell is now waiting for another command. A fun activity you could try is to look up information on the internet about customizing the prompt for the shell you use. You can change the “$” into a “>” or remove the number telling what line of input this is. You can also in some shells even change the color of the prompt.

3.2 Simple Shell Commands

Now that we have an idea of what the shell should look like, we will cover some of the most useful commands you will come across while working in your shell. Firstly though, it is

important to note that the ls command that we just saw, and almost all other commands are actually just normal programs that you could write yourself if you were so inclined. You can also invoke scripts in the same way, we will cover scripts in more detail later, but keep in mind that most often when you invoke a command, you are just running a program or script that someone else wrote. As well, before continuing, know that if you ever want or need more information on a command, you can use the "man [command_name]" command which will print a large amount of information to your shell on the command name you provided.

3.2.1 Unix Command - ls

These commands, or scripts, can take arguments as well as what are conventionally termed as options. One of the most common options for the ls command is the -al option, which is actually a combination of two different options. This option lists all files including hidden files (whose names start with a ".") and timestamps for their creation, modify and access dates in a "long listing format". This also shows permissions on a file.

```
data 50 $ ls -al
drwxr-s--x  61 jennen jennen  4096 Feb  7 16:31 .
drwxr-sr-x 224 root   root 4096 Jan 20 11:40 ..
drwxr-xr-x   2 jennen jennen  4096 Oct 30 15:48 .fontconfig
drwx--S---   3 jennen jennen  4096 Jan 30  2011 .sunw
drwxr-sr-x   2 jennen jennen  4096 Aug  3  2010 .www
drwxr-sr-x   2 jennen jennen  4096 Nov 14  2010 .xine
drwxr-s--x   3 jennen jennen  4096 Jan 13 22:50 CS177
drwxr-s---   19 jennen jennen  4096 Feb 16  2012 CS180
drwxr-sr-x   15 jennen jennen  4096 May  1  2012 CS240
drwx--S---   15 jennen jennen  4096 Apr 25  2011 CS240_old
drwxr-s---   2 jennen jennen  4096 Dec 11  2011 CS250
drwxr-sr-x   4 jennen jennen  4096 Sep 27 21:24 CS252_TA
drwxr-sr-x   8 jennen jennen  4096 Apr  3  2012 CS252_old
drwxr-sr-x   2 jennen jennen  4096 Mar 24  2012 CS307
drwxr-sr-x   6 jennen jennen  4096 Dec  6  2012 CS348
drwxr-sr-x   7 jennen jennen  4096 Dec  3  2012 CS354
drwxr-sr-x   4 jennen jennen  4096 Jan 15 16:27 CS390
drwxr-sr-x   2 jennen jennen  4096 Apr  3  2012 Desktop
drwxr-s---   2 jennen jennen  4096 Oct 30 15:53 Documents
```

3.2.2 Unix Commands - mkdir, cd, pwd

Another common command you will probably use is the *"mkdir [desired_directory_name]"* command. This command takes an argument, in this case what you want the new directory to be named, as the second word in the command. After calling this command it will create a new directory with the desired name in the current directory. We can

then make this new directory the current directory by using the “cd [directory_name]” command. You can see the current directory path at any time by using the “pwd” command.

```
data 55 $ ls
CS252-Slides-New.pptx  lab1 grading  lab1-src  lab1-testing
lab1.tar
data 56 $ mkdir new_directory
data 57 $ ls
CS252-Slides-New.pptx  lab1-src         lab1.tar
lab1 grading           lab1-testing     new_directory
data 59 $ cd new_directory
data 60 $ pwd
/homes/jennen/CS252/new_directory
```

3.2.3 Unix Commands - cp, mv, rm

More commands you will find useful for file manipulation include cp, mv, and rm. Cp copies the contents of one file to another, mv will allow you to move or rename a file, and rm will remove/delete a file from the filesystem. The file extension or type of the file do not matter to these commands. You can see examples of these commands below.

```
data 56 $ ls
test.txt
data 57 $ cp test.txt test2.txt
data 58 $ ls
test.txt  test2.txt
data 59 $ ls
test.txt  test3.txt
data 60 $ rm test3.txt
data 61 $ ls
test.txt
```

3.2.3 Unix Commands - cat, grep, head, tail

Finally there are commands which can be used almost exclusively for text file manipulation such as grep, cat, head, and tail. Head and tail print the beginning and ending lines of a file respectively while cat prints the entire file. But grep is a different beast altogether, it allows you to search a file or files for a line containing a “regular expression” we will cover these in more detail later in the chapter, but suffice to say that for now the simplest regular expression is just a normal word.

```
data 58 $ cat seuss.txt
```

```
"Look at me!
Look at me!
Look at me NOW!
It is fun to have fun
But you have to know how."
data 59 $ grep "Look" seuss.txt
"Look at me!
Look at me!
Look at me NOW!
```

3.3 Shell Input and Output

Another important feature to know when using a shell is how to handle file I/O to the commands you are using. Standard input is input directly from the shell to a command that you typed in. Standard output is output from a command that is printed to the terminal, as is standard error. You can redirect input to a command from standard input to a file you specify by using the "<" symbol. You can redirect the output from a command from standard output to a file you specify by using the ">" symbol, however this will overwrite any information already in the file you specify. If you want to append, you will use the ">>" symbol. You can also redirect standard error, which contains any error messages encountered while a command is running whenever you redirect standard output by adding an ampersand at the end, i.e. ">&", ">>&". Our example will use the echo command which simply prints whatever we give it to output. To illustrate this, the first use of echo will not contain any I/O redirection.

```
data 52 $ echo hello2you
hello2you
data 53 $ echo hello2you > outfile
data 55 $ cat < outfile
hello2you
```

3.4 Pipes and Combining commands

Often you will find that you use the output from a command as input to another command, maybe you even used temporary files to hold the information between calling the commands. There is in fact a simpler and more efficient way of doing this. You can use the pipe symbol, "|", to immediately call a second command after you call the first and provide the output of the first as input to the second command. This allows you to create more complex commands using what you already know. Consider the following example. You need to find out if a few specific files are in a directory containing a very large number of files. You know the files you are looking for contain the word "command" in their name. you could simply use ls and then look through each entry yourself, but if there are hundreds of files this could take quite some

time. You could alternatively output the directory to a temp file and then use grep on that file. Or the best solution would be to simply pipe the output of ls into grep using the word command as the regular expression to search for.

```
data 53 $ ls > out
data 54 $ grep command < out
command.cc
command.h
command.o
data 55 $ ls | grep command
command.cc
command.h
command.o
```

3.5 Basic Scripting

Scripting is an extremely valuable skill that you will want to learn thoroughly. There are many powerful scripting languages including bash, python, lua, and many many more. We will not go extremely in-depth for the purposes of this book, there are many texts which cover scripting in much more detail if you are interested. However we will cover enough basic bash scripting that you will be able to write simple scripts such as searching directories for files containing a specific line or backing up files in a directory periodically. Scripting languages are interpreted instead of compiled, so the only thing you need to do after writing your script is to run it and see what happens. In most cases when compared to compiled high level languages like Java and C++ scripting has a lower development time but slower run time.

3.5.1 Variable declaration and terminal output

The first script we will write is the iconic Hello World script. This will show how to create a variable and print text to the console. However before beginning your script first type the following command into your terminal “which bash”.
This will print a line something like "/usr/local/bin/bash" to your console which we may need in your script. If there is no bash interpreter installed on your machine, you will need to get a copy but in most cases your unix machine will already have a bash interpreter installed. You can find information specific to your unix flavor online in many locations for installing a bash interpreter. The full script we will be writing looks like this:

```
1. #!/bin/bash
2. #declare STRING variable
3. STRING="Hello World"
4. #print variable on a screen
```

```
   5. echo $STRING
```

We will first run this script and then look at what each line is doing. Create a file called "hello_world.sh" using your favorite text editor and put the above script into the document without the line numbers. Then save the file and use the command "chmod +x hello_world.sh" in the directory you saved the file in, and then type "./hello_world.sh" to execute our script. If the script does not run then replace the first line with the file path that was returned by your "which bash" command. You are strongly encouraged to write the scripts yourself.

- Line 1 line of any script we will write will be the location of the interpreter for our script, in this case, the location returned by our "which bash" command. When we execute our script this location is used to find the program that will read our script called an interpreter.
- Lines 2 and 4 are comments designated by the # symbol
- Line 3 is the declaration of our variable, keep in mind we could just have easily named this variable HELLO or GOODBYE and that there is no type declaration as there would be in C or Java indicating what type of variable we are creating. We simply choose a name for our variable and assign it a value.
- Line 5 uses the echo command which is simply a shell command that can be used at any time in the shell. Try this now by typing "echo stuff" into your terminal to see the results. The $ symbol prior to the STRING variable is a symbol used to reference our variable and if we had left off this symbol the output of our program would have been "STRING" instead of "Hello World".

3.5.2 Conditional, if-else statements

Next we will cover conditional statements. Here forward we have included a sample file name at the beginning of the line numbers, but truly it doesn't matter what you name your files.

```
if_else.sh
   1. #!/bin/bash
   2. directory="./Scripting"
   3. #if statement to check if directory exists
   4. if[ -d $directory ]; then
   5.     echo "Directory exists"
   6. else
   7.     echo "Directory does not exist"
   8. fi
```

- Line 1 contains our interpreter file path
- Line 2 contains a variable declaration to a fictional directory

- Line 4 begins our if statement with a conditional in brackets followed by a semi-colon and the "then" keyword which must follow if statements. It is important to note that the brackets must not be touching the reference to our variable.
- Line 5 and 7 are essentially print statements to the console
- Line 6 contains the else statement, nothing special about it
- Line 8 contains the "fi" keyword which closes the innermost un-closed if statement, all if statements must be closed with this keyword.

3.5.3 Passing command line arguments

```
passing_arguments.sh
   1. #!/bin/bash
   2. echo $1 $2 $3
```

Run this script by typing "./passing_arguments.sh Why Hello There!" into your terminal.

- Line 2 contains the $ reference symbol from our introduction to variables but with three numbers. These numbers (and more if necessary) contain the arguments passed to the script in order as you can see from running the code.

3.5.4 File I/O review

```
file_io.sh
   1. #!/bin/bash
   2. echo Hello there people! > outfile.txt
   3. cat < outfile.txt
```

- Line 2 contains the ">" symbol which redirects all output from a command to the file followed by the symbol.
- Line 3 contains the "<" symbol which provides input to the command from the file followed by the symbol.

3.5.5 For and while loops

```
loops.sh
   1. #!/bin/bash
   2. # for loop
   3. for f in $( ls /var/ ); do
   4.     echo $f
   5. done
   6.
   7.
```

```
8. COUNT = 6
9. # while loop
10.   while [ $COUNT -gt 0 ]; do
11.       echo Count: $COUNT
12.       let COUNT=COUNT-1
13.   done
```

- Line 3 contains a for loop declaration where a variable f is created that is filled in with each item in the array returned by ls /var/ on each iteration of the for loop.
- Line 4 prints the name of the file name currently stored in f on each iteration of the for loop.
- Line 5 has the done keyword signaling the end of the body of the for loop, this must always be included as if it were a closing brace for a for loop in the C programming language.
- Line 9 contains a while loop declaration where the count variable is used with the -gt (greater than) option to compare it to 0.
- Line 11 uses the new let keyword to perform arithmetic on the count variable.
- Line 12 contains the done keyword which works as described above for the for loop.

3.5.6 String Manipulation

string_manipulation.sh

```
1. #!/bin/bash
2. foo="Hello"
3. foo="$foo World"
4. echo $foo
5. a="hello"
6. b="world"
7. c=$a$b
8. echo $c
9. echo ${#foo}
```

This script is fairly self-explanatory. It shows two fairly similar ways to concatenate strings, one is no better than the other and it is up to personal preference which you use. The last line shows how to get the number of characters in a string.

3.5.7 Common Environment Variables

environment_variables.sh

```
1. #!/bin/bash
2. echo $HOME
```

```
3. echo $PATH
4. echo $USER
```

This script is a simple one, it illustrates the use of "environment variables". You can see all the environment variables currently available by using the command printenv. These are variables that are persistent throughout the terminal shell you are using and are available to programs and scripts run in the shell.

3.6 Complex Scripting

Now its time to solidify our new found bash scripting knowledge by writing our very own script! Our script will be a simple program to backup our home directory once every three minutes into a backup folder and name the backup according to the time it was created. The code for this script will not be provided, this is an exercise for you to practice! Make sure to make a solid attempt at the script. The script has been broken up in parts or steps to make it easy to complete in increments.

Backup Script Part 1

We will begin by remembering to identify the interpreter using #!/bin/bash. Then we will need to make sure we are working in the home directory so we can get to the files we want to backup. Remember to use the cd command to change directories, and you can use the "~" symbol in place of a directory path to specify your home directory.

Backup Script Part 2

Next we need to make our "backup" directory! We can use the mkdir command for this. After that we will start the actual process of backing up everything once every three minutes. To this we will need to repeat an operation, that means we will need to use a never-ending loop. We can use a while loop for this with the condition being any expression that will always evaluate to true. Note in general, we would want to provide some way for the user to tell the script to stop running, but in our case we will just stop it ourselves by sending the interrupt signal to the terminal, otherwise known as typing into the terminal while a process is running "Ctrl+C", this will end the backup script for us.

Backup Script Part 3

Now, inside our loop we need to make a copy of our home directory first off. However, we don't want to use up too much memory when we do this because we could have a lot of files in our home directory. So we will use the "tar" function to compress all our files into a .tar file. We can tell it to tar all our files by using the "." symbol as the filename argument to tar. However we don't want it to include our backup folder! To prevent this we can use the --exclude=backup argument for the tar command. The exclude argument should go first, followed by the ".", which is then followed finally by what we want to name our file. We wanted to name our file with the current date and time. We will do that by using the date command. We will supply this for your to save

time, your command will look something like this: $(date +%Y-%m-%d-%H-%M-%S). That will create a string with the current year-month-day-hour-minute-second as the string respectively.

Backup Script Part 3

Still inside our loop now, we have made our backup successfully in the last part. Now we just need to wait for three minutes and we are finished with our first script! To do that, all we have to do is use the sleep command, which tells the process to wait for the supplied amount of time. Note that the sleep command will take its argument in milliseconds, so you will have to convert three minutes into the number of milliseconds you want the script to wait. After that you are all finished!

3.7 Regular Expressions

Regular expressions are used in pattern matching in order to find certain strings or combinations of letters and numbers in documents or string objects. It is likely you have already worked with regular expressions before, whether you realize it or not. Here we will look specifically at using egrep with regular expressions. The simplest regular expression is one that just looks for a specific word or character as demonstrated first. We will use a text file called regex_test.txt with the contents shown below for all the examples unless otherwise noted:

```
Patty Farnsworth
Phone Number:
765-899-4756

4lph4num3r1c

Hello there students
@regular
```


For each of the examples we will use egrep to facilitate our examples, however regular expressions are used in many other applications as well. The formatting will be the command we put into the terminal, followed by the output, and then a brief explanation of what we are doing with that expression. The first argument supplied will be our regular expression and the second argument will be our text file that we want to parse for a line that meets certain criteria.

egrep Patty regex_test.txt
Output: Patty Farnsworth
This command will return the line containing the name Patty from the document. Note that this is case-sensitive and that using "patty" instead of "Patty" will not return the line we want.

egrep [PH] regex_test.txt

Output: Patty Farnsworth
Phone Number: 765-899-4756
Hello there students @regular
When multiple symbols are placed inside of brackets, as in this case, you are saying find lines containing "P or H".

egrep [0-9] regex_test.txt
Output: Phone Number: 765-899-4756
4lph4num3r1c
You can search for all the numbers 0,1,2,3,4,5,6,7,8,9 by using a '-' symbol between two numbers 0 and 9. Similarly you can search for alpha characters by using a-z or A-Z for capital letters.

egrep [0-9][a-z] regex_test.txt
Output:4lph4num3r1c
It is important to see that when the brackets are used, it will only search for a single character. So in this case we are looking for any line that contains the pattern of any number immediately followed by any lower case letter.

egrep [0-9]+ regex_test.txt
Output:Phone Number: 765-899-4756
Here we see the + operator. This operator says "one or more" of the preceding character so in this case it will match any line that has 1 or more numbers consecutively.

egrep . regex_test.txt*
Output:Patty Farnsworth
Phone Number: 765-899-4756
4lph4num3r1c
Hello there students @regular
This introduces a new symbol and operator, the "." symbol which matches anything and the "*" operator which matches 0, 1 or more of the preceding character. So this regular expression will literally match every line with at least one character of any type in the document. It is important to note that this will not return the empty lines since they do not contain a character.

egrep [0-9][a-z]+[0-9] regex_test.txt
Output: 4lph4num3r1c
In this command we notice that multiple regular expressions can be combined to find complex patterns in a document. For example, in this expression we are asking for a pattern that matches a number following by at least 1 or more lowercase letters followed finally by another number.

egrep [#$\+%@] regex_test.txt
Output: Hello there students @regular

In this command we are searching for lines containing specific special characters. Note we have escaped the + symbol to prevent it from being used as a matching operator in regex and is instead treated as just the normal "+" character so that we can search the file for it.

3.7.1 Backreferences

A final important piece of syntax for regular expressions to learn is the backreference. The backreference allows you to match specific types of characters for a certain number of repetitions. For example you can use [0-9][0-9]* to match 1 or more decimal as shown previously. However, sometimes we want to reference repeated instances of the same character. If we do not know what the character will be exactly, it can be more difficult than simply saying “match for ‘9999’ in this string”. This is when we will use a backreference. When parts of an expression appear in parentheses they can be referred to later using backreferences. We use \1 to refer to the first backreference, \2 to the second, and so on.

An example of this would be if we wanted to match two or more of the same decimals in a row. We don’t care which decimal, just that there are at least two or more of a kind. ([0-9]) will match on the first character. Then we can append a \1 to match on the next character, then add the extended regex character + to indicate 1 or more matches. It will then look like this: “([0-9])\1+”. This would match strings like ‘1111’ or ‘33333’. Explicitly, this says “match on any single decimal, then one or more of what we just matched”. As a further example, if we wanted to match on strings such as ‘121212’ or ‘232323’, we could use the expression: “([0-9])([0-9])\1\2+”.

Chapter Summary Questions

1. What command would you use to see the contents of a directory?
2. How do you provide the output of one command to another as input with a unix command?
3. How do you access the arguments passed into a script?
4. Create an if-else statement containing a while loop in bash, it doesn’t matter what they actually do.
5. Write a regular expression to match both of these strings, do not use “.*”: Aaab123 and 123Baaa.