CPSC 1070
Lab Project: bash shell scripts
Nov. 11 & 13

Introduction

This is a two day exercise to begin your familiarization with scripting within the Linux bash shell. You are to complete as many of the tasks on the last page as you are able to before checking out with your lab TA on Wednesday.

The interface that you type into and interact with when you are using Linux at the command line is called the *shell*. There are several shell versions, the most well known being the *bash shell*, the *korn shell*, and the *c shell*. The standard School of Computing Linux distribution uses the *bash shell*. One of the powerful aspects of using Linux at the command line is that you can define short programs, called *shell scripts*, that you can create quickly to do complex or tedious tasks by typing a single command. Shell scripts allow you to combine multiple shell operations using loops, and conditionals. All of the operations available to you under the bash shell, including programs you have written yourself can be used to build these scripts.

The link on the lab schedule page for Nov. 11 will take you to a bash shell scripting cheatsheet that you can use to help you find your way around during these labs. You can see by the cheatsheet that there is much, much more to shell scripting than will be covered in this lab.

Tasks and Concepts for Monday

Make a working directory

Download the zip file `scripting.zip` linked to the lab schedule for Nov. 11 into your top level directory, and unzip it. This will create a directory named `scripting`. Move into this directory. If you execute an `ls` command, it should display:

```
message.txt morefiles/  somefiles/  text.txt
```

This set of directories and files will be used for this lab and the subsequent ones, giving you a playground for trying shell scripts, without worrying about damaging any of your other files. If you mess things up you can just go back up to your main directory, delete the `scripting` directory, and unzip it again:

```
$ rm –r scripting
$ unzip scripting.zip
```

Basic shell scripting concepts

A shell script is simply a text file, containing a sequence of shell commands that are intended to be executed in order. Shell commands are the ones you are already used to like `cd`, `ls`, `mv`, `cat`, plus several additional ones like `for` and `if` that you can use to form a program. In addition, the names of any executable programs you have can be treated just like other Linux commands. Let's say that your shell script is named `script`, then it can be run by the command

```
$ bash script
```

or what is usually done is that the script is made executable via the command

```
$ chmod a+x script
```

and then the shell script can be run as if it were a compiled program like this

```
$ script
```

or you might have to run it like this

```
$ ./script
```

if you have not included the current directory `.` in your `PATH`.

Here is a simple "hello world" example to demonstrate the basic structure of a shell script:

```
#!/bin/bash
echo "Hello World"
```

The first line tells whatever shell you are running that this shell is written in the bash shell scripting language, and that it should be run under the bash shell. The `echo` Linux command prints whatever its argument is. So, if you have saved the above script with the name `hello`, then you can make it executable, and run it like this:

```
$ chmod a+x hello
$ hello
Hello World
```

Please create this script in a text editor, make it executable, and run it, so you get the feel for how this works.

A first useful example

Here is an example script that actually does something useful:

```
#!/bin/bash
i=0
for file in *.txt
do
  let i=$i+1
done
echo $i
```

The new commands and keywords that we have not yet seen are bolded. Note that all spaces and lack of spaces in the above example are important. For example, the line `i=0` should have no spaces, while the line `for file in *.txt` needs to have the spaces as shown. This is because the bash shell uses spaces to separate commands from their arguments and arguments from each other.

The line `i=0` creates a variable named `i` and sets it to 0. Then in the code, `i` stands for the variable, and `$i` stands for the variable's contents. There should be no spaces in the statement `i=0`, as the = denotes a special form of bash command that assigns the value on the right of the = to the variable on the left.

The `for` line initiates a for loop with a Python like syntax. `*.txt` is a wildcard that will evaluate to the names of all of the files in the current directory that have the suffix `.txt`. the full statement `for textfile in *.txt` means to loop once for each name in the list of filenames returned by `*.txt`, assigning each subsequent file name to a variable named `textfile` on each iteration. The body of the loop is bracketed by `do` and `done`.

The statement `let` is a command to evaluate an arithmetic expression, so `let i=$i+1` takes the value of `$i`, adds 1 to it, and stores the result back in `i`. The same expression without the `let` treats the expression as a string expression, so if `i` contained `0`, `i=$i+1` would store the string `0+1` in `i`, instead of the arithmetic sum.

You can probably see that this program counts the number of `.txt` files in the current directory, and prints out the total.

Create this shell script in the `scripting` directory, name it `counttexts`, make it executable, and run it . Since there are 2 `.txt` files in the `scripting` directory, the result should look like this

```
$ countexts
2
```

Now, `cd` into the `morefiles` directory and run the program again, like this:

```
$ ../counttexts
1
```

What is wrong? There are no `.txt` files in the `morefiles` directory, but the program prints 1. The problem is that the wildcard `*.txt` returns `*.txt` if it finds no matching files, so the for loop will execute once, adding 1 to `i`.

We could fix this by using an if statement so that `*.txt` will no longer be counted, like this

```
#!/bin/bash
i=0
for file in *.txt
do
  if [ $file != "*.txt" ]
  then
    let i=$i+1
  fi
done
echo $i
```

The statement `if [ ]` denotes an if statement, with its test inside the brackets, the statements `then` and `fi` bracket the statements that should be executed only if the test is true.

Again, note that since the shell uses spaces to separate arguments to a command, the spaces in the above example are important. The shell command is `if`. Its arguments are `[`, `$file`, `!=`, "`*.txt`", and `]`.

Add this correction to your `counttexts` script, and test it to make sure it works correctly.

Useful reading on if and looping statements

Here are links to two excellent tutorials on if and looping statements in bash scripts:

https://ryanstutorials.net/bash-scripting-tutorial/bash-if-statements.php
https://ryanstutorials.net/bash-scripting-tutorial/bash-loops.php

Please read through these tutorials to be ready for lab on Wednesday.

Tasks for Wednesday

On Wednesday you should complete as many of the following tasks as possible before leaving the lab. Demonstrate to the TA the scripts that you have working before leaving lab.

Task 1

Write a script that will convert all of the `.jpg` files in the `somefiles/images/` directory into `.bmp` files.

Remember that you can use the *imagemagick* program `convert` that we learned about in the October 21 lab to convert from `.jpg` to `.bmp`. Recall that if you have the image file `homer.jpg` and you want to convert it into a `.bmp` file, you will need to execute the command

`convert homer.jpg homer.bmp`.

In order to do this in a script, you will need to know how to extract a filename prefix from its suffix. If you have the filename `homer.jpg` in a variable named `imgfile`, then the syntax `${imgfile%.jpg}` will return the name `homer` with the suffix `.jpg` removed.

Task 2

Write a script that will find all of the executable files in the subdirectories of the directory `scripting/somefiles/source/` and print their names. Recall from the reading that the test for a file being a directory is of the form `[ -d $filename ]`, and the test for a file being executable is of the form `[ -x $filename ]` The printout should be:

```
subdirectory: big23
   big2
   big2-1print
   big2-1print.py
   big2.py
   big3
   big3.py
subdirectory: biggest
subdirectory: bigloop
   bigloop
   bigloop.py
subdirectory: broccoli
   broccoli
```

Task 3

Write a script that will find all of the `.o` files and executable files in each of the subdirectories of the directory `scripting/somefiles/source/` and delete them. After deleting these files it should make a zip archive of each of these subdirectories. Recall that to zip a directory you use the command

`zip -r <directoryname> <directoryname>`