Lab 9: Bash scripting (Week of 4.4.16)

Checkoffs are due in the lab period. To get checked off, complete all tasks below and demonstrate your scripts to a TA.

Hello World

Scripting is synonymous with programming — it is programming — except it's generally considered more light weight. For the scripts we'll write, the commands will be stored in a file, but that file is **interpreted** by the bash shell. That means each line actually runs like a sequence of command lines executed in the order they appear in the file as if they were typed onto the command line.

You must indicate that you wish for your script to treated as such by designated which program should be interpreting each line of commands. For that, we use a `#!` symbol followed by the program. For this lab, that program is `bash`. Next, for the first program you write in any language, you begin with "Hello World," so let's take a look at that first.

```
#!/bin/bash

echo "Hello World"  # This is a comment
```

By convention, all bash scripts, or shell scripts, have a file type `.sh`, and we will use that convention in this lab. At the start of all bash scripts is the `#!` symbol and the path to the bash program to interpret the script, see Line 1. The `#!`, pronounced "shebang" tells the shell when reading this file, treat it as bash.

On line 3, there is the actual command to print to standard out, "Hello World", as well as a comment, anything following a "#". The `echo` command echoes back anything it is given to `stdout`, in this case, the string "Hello World."

We still need to execute your program, you have to make the file executable.

```
#> chmod +x helloworld.sh
```

Then you execute with `./` indicator

```
#> ./helloworld.sh
Hello World
```

And there you have it, "Hello World" from bash.

Output to stdout vs. stderr

A bash script is just like other command line tools we've seen so far with respect to output and input. The same standard file names and file descriptor numbers are automatically provided as well as the ability to redirect them.

When you put a command in a script that writes to standard out, like say a `head` or a `cat` command, then that output will also be part of the standard output of the script, just like `echo` was above. Similarly, if a command has an error and writes to standard error, that will also be a part of the script's standard error.

If you want to write to standard error directly in your script, your only option is the `echo` command to generate output, but `echo` only outputs to standard out. You must redirect the standard output of `echo` to `stderr` using a redirect like so:

```
echo "ERROR: Something bad happened"  1>&2  #<-- redirect file descriptor 1
to 2
```

Recall that the standard files have numeric file descriptors, 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`. When you redirect, you can choose based on the file descriptor number, and you can even redirect to another file descriptor using the `&` followed by the numeric descriptor.

Bash Variables

You can assign variables in bash just like in C, but you refer to variables using a $ symbol.

```
name=amit
echo "Your name is $name"
```

Note that spacing is important, the assignment must occur immediately following the equal sign. If there are spaces in the string value, use quotes.

```
name="Amit Asthana"
echo "Your name is $name"
```

Finally, variable replacement, that is substituting a variable for its value, occurs throughout a script, including in quotes, like above.

Typing: Strings vs. Arithmetic

One major difference between bash, which is a scripting language, and C, is that you don't need to identify the type of the data you're storing to a variable. Instead, you have to provide some context for how you want the data to be interpreted, otherwise, by default, it will be treated like strings and + is concatenation.

```
n=1
n=1+$n
echo $n   #<-- print "1+1" not 2!
```

To perform arithmetic operations, you use the `let` operations.

```
n=1
let n=1+$n #<-- use quotes if you need white space
echo $n   #<-- prints 2!
```

There is also no such things as floats in bash. Everything is a numeric integer.

Empty variables

Finally, as in most scripting languages, you don't need to declare your variables ahead of time. That means a variable can be referenced before it ever received a value. In those cases, bash treats the variable as empty, equivalent to the empty string.

```
echo $foo
```

Will echo nothing, since variable `foo` doesn't exist.

Environment Variables

In a bash script, and in bash shells in general, there are a number of environment variables available for convenience. Once such environment variable we've seen previously is the `PATH` variable:

```
echo $PATH
```

By convention all environment variables are upper case. Below are some really useful ones to know:

- `$USER` : the current user
- `$HOME` : the home directory
- `$PWD` : The current working directory
- `$SHELL` : The name of the shell you are currently running

For example, this program prints a helpful message

```
#!/bin/bash

echo "Hello $USER!"
echo "You are currently here: $PWD"
echo "But, your home directory here: $HOME"
```

Additionally, there are environment variables for arguments passed to the shell script. These are $0, $1, $2, etc, where the number refers to the command line argument. For example, consider the script, printargs

```
#!/bin/bash

echo "arg 0: $0"
echo "arg 1: $1"
echo "arg 2: $2"
echo "arg 3: $3"
echo "arg 4: $4"
```

If we were to run it:

```
arg#  0          1 2 3 4
      |          | | | |
      v          v v v v
#>./printargs x y z
arg 0: ./printargs
arg 1: x
arg 2: y
arg 3: z
arg 4:
```

Argument 0 always refers to the script, and each `$1` refers to the arguments to the script. The script prints the first 4 arguments, but there isn't a fourth argument. `$4` is treated as the empty string. To refer to all arguments, not including `$0`, use `$*`. `$#` is special variable set to the number of arguments.

```
#!/bin/bash

echo "There are $# number of args"
echo "Arguments _not_ including the name of the script: $*"
```

Conditional Control Flow

Like any reasonable programming environment, you want to do more than just print things out in an iterative fashion. We need a mechanism to change the program based on some condition, like if/else statements and looping.

If/Elif/Else Statements

The format of an if statement is like so:

```
if cmd
then
    cmd
elif cmd
then
    cmd
else
    cmd
fi        #<-- close off the if statement with fi
```

The `if cmd` part proceeds if the `cmd` succeeds, that is, it doesn't have exit with failure. In general, the `cmd` used in if blocks and others are the `[` command (see `man [`) which checks conditions.

With the `[` you can do simple comparators and other operations. Here is an example script that checks if the script was run from the home directory:

```
#!/bin/bash

echo "Hello $USER"

if [ $HOME == $PWD ]
then
    echo "Good, you're in your home directory: $HOME"
else
    echo "What are you doing away from home?!?"

    cd $HOME

    echo "Now you are in your home directory: $PWD"
fi
```

There are two things of note here. First, the `[ $HOME =` $PWD] is a command that succeeds only when `$HOME` is the present working directory. Also, when you change directories within a script, you are not changing the directory of the shell you ran the script from, just the present working directory of the script itself.

Comparators

Conditions in bash exist within brackets, `[ ]`, but the `[` is actually a command that returns true if the conditions were met. For string comparators, you can use `= and !` but for numerics, you need to use options to the `[` command. See below for the varied usages.

```
#!/bin/bash

#string
var=amit
if [ $1 = $var ] ; then echo "string $1 equals $var" ; fi
if [ $1 == $var ] ; then echo "string $1 equals $var" ; fi
if [ $1 != $var ] ; then echo "string $1 does not equals $var" ; fi
if [ -z $1 ] ; then echo "string $1 is empty!"; fi
if [ -n $1 ] ; then echo "string $1 is not empty!"; fi

#numeric
a=1
if [ $a -eq $1 ] ; then echo "number $1 equals $a" ; fi
if [ $a -ne $1 ] ; then echo "number $1 does not equal $a" ; fi
if [ $a -gt $1 ] ; then echo "$a is greater than $1" ; fi
if [ $a -lt $1 ] ; then echo "$a is less than $1" ; fi

#file/dir properties
if [ -d $1 ] ; then echo "$1 exists and is a directory!" ; fi
if [ -e $1 ] ; then echo "$1 exists!" ; fi
if [ -f $1 ] ; then echo "$1 exists and is not a directory!" ; fi
```

```
if [ -r $1 ] ; then echo "$1 exists and is readable!" ; fi
if [ -s $1 ] ; then echo "$1 exists and has size greater than zero!" ; fi
if [ -w $1 ] ; then echo "$1 exists and is writable!" ; fi
if [ -x $1 ] ; then echo "$1 exists and is executable!" ; fi

# NOTE: Since everything's on the same line, you need ;s between
# the if and the then and between the then and the fi.
```

Additionally, you can use the standard set of conditional operators:

```
if [ $condition1 ] || [ $condition2 ] ; do echo "either condition1 or
condition2 are true"; done
if [ $condition1 ] && [ $condition2 ] ; do echo "condition1 and condition2
are true"; done
if [ ! $condition1 ] ; do echo "condtion1 is not true"; done
```

Task 1

Write a script, `getsize.sh`, which takes a path as an argument and prints out the size of the file/dir at that path. Your script **must do error checking**. Here is some sample output:

```
#>./getsize.sh
ERROR: Require file
#>./getsize.sh adadf
ERROR: File adadf does not exist
#>./getsize.sh empty.txt
0
#>./getsize.sh larger.txt
5000
```

You should be able to use a `cut`, `ls`, and/or `wc` to get the information you need. All errors should be written to `stderr` such that:

```
#> ./getsize badfile > /dev/null
ERROR: File badfile does not exist
```

You may also find it useful to use the `tr` command. Consult the man page for more details, but `tr` is used for translation, substituting one string for another. The *squeeze* option `-s`, in particular, could be useful to get rid of extra whitespace so that your `cut` fields are more consistent.

Conditional Control Flow

Like in C, there are two standard loop structures: iterate until a condition is met (while loops) and iterate for each item (for loops).

While Loops

The syntax of a while loop is as follows:

```
while cmd
do
   #commands
done
```

Here is a program that counts from 1 to 10, printing the output

```
#!/bin/bash

i=1
while [ $i -le 10 ]
do
    echo $i
    let i++ # you can use incrementers in lets
            # and, you don't need to use the $ here
done
```

For loops

A for loop has a similar structure, but unlike for loops in C, for loops in bash iterate for each item provided:

```
for var in str1 str2 .. strn
do
   # commands
   # $var is available for references, set to str1 -> strn in each loop
done
```

Here's a loop that iterates through all arguments, printing the number of that argument:

```
#!/bin/bash

i=1
for arg in $*
do
    echo "arg $i: $arg"
    let i++
done
```

Task 2

Create a script called `getallsizes.sh`, which takes in any number of files on the command line and prints their sizes as follows. Here's some sample usage:

```
#> ./getallsizes.sh empty.txt isbigger larger.txt
empty.txt 0
isbigger.sh 204
larger.txt 5000
#> ./getallsizes.sh empty.txt BADFILE larger.txt
empty.txt 0
getallsizes.sh: ERROR: File BADFILE does not exist
larger.txt 5000
```

```
#> ./getallsizes.sh empty.txt BADFILE larger.txt 2>/dev/null
empty.txt 0
larger.txt 5000
```

(HINT) Check out the man page for `echo` to print without a trailing new line so you can align the file name with their size.

Sub Shells

While it is simple to have a shell script execute a set of commands and just write to the output, often there are situations where you wish to store the output of some command or parse the output of some command as a variable name. To do that you use a subshell.

The idea behind a subshell is that you can store the result of the computation, as outputted to `stdout` to a variable:

```
local_files=$(ls)
permissions=$(ls -l | cut -d " " -f 1)
```

The variables `local_files` and `permissions` now have the results of the computation.

Task 3

Write a script, `isbiggerthan.sh`, which takes a path and a size and determines if the file or directory is bigger (or equal to) the given size. Here is the usage:

```
isbiggerthan.sh size path
```

And here is some sample output

```
#> ./isbiggerthan.sh
isbiggerthan.sh: ERROR: Require path and size
#> ./isbiggerthan.sh 1
isbiggerthan.sh: ERROR: Require path and size
#> ./isbiggerthan.sh 0 empty.txt
yes
#> ./isbiggerthan.sh 2 empty.txt
no
#> ./isbiggerthan.sh -1 empty.txt
isbiggerthan.sh: ERROR: Require a positive number for -1
#> ./isbiggerthan.sh ad empty.txt
isbiggerthan.sh: ERROR: Require a number for ad
#> ./isbiggerthan.sh 0 doesnotexist
isbiggerthan.sh: ERROR: File doesnotexist does not exist
#> ./isbiggerthan.sh 3000 larger.txt
yes
#> ./isbiggerthan.sh 10000 larger.txt
no
```

Note, checking if a variable is a number is non-trivial. Here is some simple code to do that:

```
if [ "$var" -eq "$var" ] 2> /dev/null # check for a number pipe error to
/dev/null
then
    echo "it's a number"
else
    echo "it's *not* a number"
```

Exit Status

Every program that runs in the shell, and every program generally, when it completes returns an exit code. For example, in the bash if statements, the condition that is true is a successful exit status.

On Unix, a successful exit status is 0, and all other values are different errors. There is an environment variable $? to test an exit status. Here you can see exit statuses from the command line

```
#> [ 1 -eq 2 ]
#> echo $?
1
#> [ 1 -eq 1 ]
#> echo $?
0
#> cat empty.txt
#> echo $?
0
#> cat doesnotexist
cat: doesnotexist: No such file or directory
#> echo $?
1
```

In a bash script, by default, the script exits with the same exit status as the prior exit status. However, you can set the exit status explicitly in cases of errors using the `exit` command:

```
exit 0  #exit success
exit 1  #exit error
```

Task 4

Update your `isbiggerthan.sh` script to exit with different status codes dependent on if the file is bigger than the size or if there is an error. For example:

- exit 0 : if the file is bigger (or equal) to the size
- exit 1 : if the file is not bigger (or equal) to the size
- exit 2 : if not enough arguments
- exit 3 : did not receive a number for size
- exit 4 : received a negative number for size
- exit 5 : file does not exist

Once complete, now create a new script `isbiggerthanall.sh` which takes the following arguments:

```
isbiggerthanall.sh size path [path [...]]
```

which outputs all the files provided that are bigger than the size. Your script **must call** your `isbiggerthan.sh` script and check the exit status to determine the result and output.

Here is some sample output

```
#> ./isbiggerthanall.sh 1
isbiggerthanall.sh: ERROR: Require a size and at least one file
#> ./isbiggerthanall.sh 1 empty.txt
#> ./isbiggerthanall.sh 0 empty.txt
empty.txt
#> ./isbiggerthanall.sh 0 larger.txt
larger.txt
#> ./isbiggerthanall.sh 6000 larger.txt empty.txt
#> ./isbiggerthanall.sh 0 larger.txt empty.txt
larger.txt
empty.txt
#> ./isbiggerthanall.sh 0 *
empty.txt
getallsizes.sh
getsize.sh
isbigger.sh
isbiggerthan.sh
isbiggerthanall.sh
larger.txt
#> ls *.sh  | xargs ./isbiggerthanall.sh 100
getallsizes.sh
getsize.sh
isbigger.sh
isbiggerthan.sh
isbiggerthanall.sh
#> ls *.sh  | xargs ./isbiggerthanall.sh 300
isbiggerthan.sh
isbiggerthanall.sh
```

(On Your Own) Google and learn about case statements for bash, and use a case statement to check the exit condition.