Homework 3
Shells, Environment, and Scripting

1 Understanding your **PATH**

In a terminal, type `PATH=` (just hit enter after the equal sign, no space characters anywhere). Try to use the terminal like normal (try running `ls`). What happened?

Give an example of a command that used to work but now doesn't:

Most things don't work `mv, cp, rename, gcc, make, ....` I even gave you one example in the question itself (`ls`).

Can you still run this command with an empty `PATH`? How?

Lecture showed off the `which` command, but that may have flown by.

This is a great example of something that should have been managable to figure out on your own. The first google result for "ls command not found", is a stack exchange post where (admittedly the second response) says "To check that it is indeed a problem with your path, what's the result of `/bin/ls` ?" (The query "what does PATH do" is equally fruitful).

Another good way of problem-solving this would be to open a new terminal and look at the original contents (i.e. `echo $PATH`). The colon-separated list is a little cryptic, but exploring those directories would hopefully let you figure this out.

Give an example of a command that works the same even with an empty `PATH`. Why does this command still work?

Anything built-in to bash would still work. `cd` is the easy answer here (why doesn't it make sense for `cd` to be an extrenal command?). For a full list, you can type `help` and see a list of built-ins (incidentally, `help` itself is a built-in and would have been a good answer to this question).

A interesting last question to add would have been, *Why do you need to type `./a.out` to run your programs instead of simply `a.out`? What is the dot in `./` doing?*

2 Playing with the shell a bit: Special Variables

Bash has quite a few special variables that can be very useful when writing scripts or while working at the terminal.

What does the variable `$?` do? Give an example where this value is useful

`$?` holds the return value of the last command run. By convention, this is 0 if the command succeeded and nonzero if the command failed. It can be useful when writing scripts, especially if you want to have fairly complex things happen depending on success or failure (making it hard to just use `&&` or `||`).

Try writing a simple program and varying the return value from `main`. Run your program and look a the value of `echo $?`.

What does the variable `$1` do? Give an example where this value is useful

`$1` is the first argument to a function. Bash syntax looks a little different that some other languages you may have seen before. You use can use it something like this:

```
$ cat b.sh
#!/usr/bin/env bash

# The first argument passed to this script
echo $1

# Argument 0 is the script itself
echo $0

function iamafunction {
        echo "In iamafunction"
        # Notice, no argument list, just assume they're there
        echo "   Arg1 is ->$1<-"
        echo "   Arg2 is ->$2<-"
        echo "   Arg3 is ->$3<-"
}

# Can pass things through and in a different order, spaces separate arguments,
# no ()'s to call:
iamafunction one two three

# Can pass through variables too, note that $1 for the script is now $2 inside
# the called function and $3 is empty, so it just prints nothing
iamafunction firstarg $1

[-bash] Fri 07 Oct 19:39 [/tmp/bb]
$ ./b.sh arg1 arg2
arg1
./b.sh
In iamafunction
   Arg1 is ->one<-
   Arg2 is ->two<-
   Arg3 is ->three<-
In iamafunction
   Arg1 is ->firstarg<-
   Arg2 is ->arg1<-
   Arg3 is -><-
```

3 Basic Scripting

Recall from lecture that scripting is really just programming, only in a very high-level language. Interestingly, `sh` is probably one of the oldest languages in regular use today.

`make` is a good tool for build systems, but we can actually use some basic scripting to accomplish a lot of the same things. First, write a simple C program that prints "Hello World!". Write a shell script named `build.sh` that performs the following actions:

1. Compile your program
2. Runs your program
3. Verifies that your program outputs exactly the string "Hello World!"
   - There are good utilities that check the `diff`erence of two files. They could be helpful.
4. Prints the string "All tests passed." If the output is correct, or prints "Test failed. Expected output >>Hello World<<, got output >>{the program output}<<".

   I think this question would have been improved with a little more direction to help you get started, specifically, I would add:

   To get you started, here's a simple shell script. Try copying it and running it.

   ```
   #!/usr/bin/env bash

   echo "Hello, I am a shell script"

   # Anything after a # character is a comment

   # Shell scripts are like working in a terminal, only the script types the
   # commands automatically for you

   pwd
   ls

   # You won't break anything with shell scripts. The best way to write one is
   # to add some commands and run it until the script does what you want.
   ```

   Recall that your shell script began life a simple text file, it's up to you to let the operating system know that this is actually now an executable program. (← this is a pointer to `chmod +x` without actually saying it)

~~Copy the output of `cat build.sh` here:~~ Show that your script works as expected for both correct and buggy hello world programs. Copy the contents of your script below:

As for a solution, there are **many** ways of answering this question. This is true of scripting in general. Scripts are (usually) not meant to be "pretty" programs. There is no need to have the shortest or most elegant solution. That's a waste of everyone's time. They are the kind of tool where you simply find something that works and move on.

To that end, I've copied in several different solutions from several different students that show off the various ways this could be done on the next page.

```
Probably my favorite submission, as the citation history at the bottom shows
how this student identified the three challenges (command output -> variable,
conditionals in bash, and string comparison in bash), found examples for each,
and synthesized a result:

#!/bin/bash
#1. Compile the program
gcc main.c -o hello_world

#2. Run the program
output_val=$(./hello_world)
expected_val='Hello World!'

#3. Verify the program outputs 'Hello World!'
if [ "$output_val" == "$expected_val" ]
then
  echo "All tests passed"
else
  echo "Test failed. Expected output >>Hello World<<, got output >>$output_val<<"
fi

Sources:
http://www.cyberciti.biz/faq/unix-linux-bsd-appleosx-bash-assign-variable-command-output/
http://www.thegeekstuff.com/2010/06/bash-if-statement-examples/
http://stackoverflow.com/questions/4277665/how-do-i-compare-two-string-variables-in-an-if-statement-in-bash

--------------------------------------------------------------------------

#!/bin/bash
# Compile and make executable
gcc hello.c -o hello_exec

# Set the desired value in a temp variable
test="Hello World!"

# Store the output in a temp variable
output="$(./hello_exec)"

# Run diff against the desired string
diff <(echo $output) <(echo $test) &>/dev/null

# Check the exit code of diff
if [ $? = 0 ]; then
  echo "All tests passed."
else
  echo "Test failed. Expected output >>$test<<, got output>>$output<<"
fi

--------------------------------------------------------------------------

$ cat build.sh
g++ hello.cpp -o hello
"Hello World!" > correct
./hello > output
cmp --silent output correct\
  && echo "All tests passed"\
|| echo "Test Failed. Expected output >>Hello World!<<, got output >>$(cat output)<<"

--------------------------------------------------------------------------

gcc main.c -o main
./main > file1
echo -n "Hello World!" > file2
if diff -q file1 file2 > /dev/null; then
  echo "All tests passed."
else
  echo -n "Test failed. Expected output >>Hello World!<<, got output >>"
  cat file1
  echo "<<"
fi
```

4 Controlling your environment

In lecture, we added a directory to our `PATH` so that we could just type `hello` and the Hello World program would run. It would be annoying to update the `PATH` variable every time we open a new terminal. Fortuntately, we can do better.

Describe how you would set up your system to modify your `PATH` automatically every time you open a new terminal (what file would you change and what would you put in it?)

Any of the files that are read on startup would work, but the traditional one to use is `.bashrc`

You'll need to include `export` such any changes are propogated into subshells. Something like:

```
export PATH=$PATH: /bin
```

is quite common.

Roughly how long did you spend on this assignment? ______________________