Assignment 3, Due October 4

1 Summary

This assignment gives you practice with writing shell scripts. Shell scripting is also known as `bash` programming. Your shell is `bash`, and when you write a shell script you are writing a `bash` script, which means that you are really writing a program in the *`bash` programming language*. You may not have thought about it in this way before, but that is precisely what you have been doing. `bash` is not just a command line interface to the UNIX kernel; it is a programming language.

The reason that this seemingly semantic different matters now is that soon you will begin writing programs in another programming language, *Perl*, and it is a good time to realize that this will not be a giant step but a sequence of small steps towards mastering Perl. By learning how to write `bash` programs, you are also learning how to write programs in general.

2 Some Useful `bash` Commands

We have been calling your programs *shell scripts*. A script is a program, make no bones about it. Scripts are programs written in a *scripting language*, which is a special kind of programming language. All scripting languages are programming languages, but not vice versa. The distinction will be explained in a later lecture. I will call `bash` both a programming language and a scripting language.

When you type a command such as `who`, you are really telling `bash` to run the `who` command. Similarly, when you type the command `date`, you are telling `bash` to run the `date` command. These are simple commands for `bash` because all it has to do is find the file that contains the command and execute that file. But `bash` has other, more complex instructions, which we also call ***statements***.

One such instruction is the if-instruction, whose simplest form is

```
if <test_expression>
then
    <sequence of statements>
fi
```

where `<test_expression>` is a statement that is interpreted to be true or false, and `<sequence of statements>` is any sequence of statements. (The *if-statement* is more general than this; we are starting simple.) The test statement has the form

```
test <expression>
```

where the `<expression>` is one one many different possible forms.

Example

```
if test $# -ne 2
then
    echo "usage: $0 arg1 arg2"
    exit
fi
echo "You typed two arguments: $1 and $2"
```

When `bash` executes this statement, it first evaluates the `test` statement

```
test $# -ne 2
```

This statement compares `$#` to 2. If the test evaluates to true, which means that the number of command parameters (`$#`) is not equal to 2 (`-ne 2`), then the statements between `then` and `fi` are executed. If the test is not true, `bash` skips over these statements and executes what comes after them, which is the second `echo` command. The -ne is a "not equal" operator. There are also five other operators: `-eq`, `-lt`, `-gt`, `-le`, `-ge`, meaning equal, less-than, greater-than, less-than-or-equal-to, and greater-than-or-equal-to.

The `bash` programming language has several statements that are known as *looping statements*. A looping statement is one that makes it possible to repeat a sequence of statements one or more times. For example, `bash` has a looping statement called a *while-statement*, whose form (*syntax*) is

```
while <expression>
do
    <list-of-statements>
done
```

in which *`<expression>`* is usually the `test` command, but could be any other statement that is evaluated as being true or false, and *`<list-of-statements>`* is any sequence of statements (including more looping statements.) The following *snippet*[1] of a script shows one example of a while-statement:

```
# make sure there is a command line argument:
if test $# -lt 1
then
    echo "usage: $0 <positive_integer>"
    exit
fi
# copy the command line value into the variable number
let number=$1

# loop while $number > 0
while test $number -gt 0
do
    echo -n "."           # print one dot
    let number=$number-1  # decrease number by 1
done
echo                      # echo nothing but a newline
```

The above script will test whether the user entered a command line argument. If not it prints a usage message and exits. Otherwise it copies the value that the user typed into `number`. `number` is just the name of a storage cell. When you write

```
let number=$1
```

you are giving `bash` the instruction to create a storage cell, name it `number`, and copy `$1` into it. There are rules for what names are valid. We skip them for now. Assume that all words formed with letters are valid.

After copying the value into `number`, the script executes the while statement test. If the value stored in `number`, which is written as `$number`, is greater than 0, the statements between `do` and `done` are executed,

[1] *A snippet is just a little piece. People call small amounts of code, code snippets.*

and the test is re-evaluated. This goes on and on until the test becomes false, meaning `$number` equals 0. That is when the loop exits.

A while-statement is usually called a *while loop* because if we visualize the sequence of executed statements as being connected by an imaginary thread, then this thread loops around and around the lines of the script.

`bash` also has a *for-loop* and an *until-loop.* I will not discuss the until-loop here. The for-loop is very different from the while-loop. It has two forms. One form (again the proper term is *syntax*) is

```
for <variable> in <argument-list>
do
    <list-of-statements>
done
```

and the other is

```
for <variable>
do
    <list-of-statements>
done
```

The *<variable>* can be any valid variable name (words starting with letters and containing letters, digits, and the underscore character.) The *<argument-list>* can be any sequence of words, including words that look like numbers. Examples are

```
for number in 1 2 3 4 5 6 7 8 9 10
for name in John Jacob Judy Jocelyn
for file in ‘ls .‘
```

As you can see, this can be very powerful. As with the while-loop, the list of statements is any list of statements, but the intention is that the variable plays a role in this list. For example, the script

```
let sum=0
for number in 1 2 3 4 5 6 7 8 9 10
do
    let square=$number*$number
    let sum=$sum+$number
    echo The square of $number is $square
done
echo The sum of the numbers is $sum.
```

displays ten lines showing the squares of the first ten positive integers and then displays their sum. Notice how the sum is calculated.

The second form of the for-loop does not need an argument list. It automatically assigns to the variable the successive words from the command line arguments of the script when it is run:

```
let count=1
for name
do
    echo $count: $name
    let count=$count+1
done
```

The if-statement tests whether the output of the pipe is greater than zero. If it is, then the name is a user who is logged in at least once.

3 What Makes a Script Good?

This assignment consists of three exercises in writing relatively simple shell scripts. The objectives when writing any script are

clarity	the script should be easy to understand by someone with a basic knowledge of UNIX and must be well-documented;
efficiency	the script should use the least resources possible; and
simplicity	the script should be as simple as possible.

An example will demonstrate. Suppose we needed a script that would count the number of lines in a file named `molecule` containing the word 'ATOM' anywhere on the line. The following script would achieve this:

```
#!/bin/bash
grep ' ATOM ' molecule >| atomcount
wc -l atomcount >| answer
rm atomcount
cat answer
rm answer
```

but it is very inefficient because

- it needlessly creates files and then removes them,
- it is hard to understand because the reader spends more time reading it,
- the user may not be familiar with certain operators such as `>|`, and
- it is not as simple as it could be.

A simple, well-documented, and efficient solution is

```
#!/bin/bash
# Displays how many lines in the file molecule contain ATOM as a complete word
# Written by Stewart Weiss on September 12, 2018
# Usage: The file name is hard-coded into the program. A more general solution
#        would let the user enter the file name and check if it exists before
#        doing anything with it. The file molecule MUST be in the current working
#        directory and must be readable by whoever runs this script.
grep -c ' ATOM ' molecule    # The -c option to grep counts matching lines
```

It has comments to explain what it does and it achieves it with a single command that can be looked up easily.

4 Tasks

1. Write a bash script named `triangletype` that expects three ***positive whole numbers*** on the command line, in increasing order. In other words, it should be called like this

   ```
   $ triangletype  3  12  15
   ```

The script checks first of all that there are three numbers and that they are all positive, and increasing in value. If either of these conditions is false, it displays a usage message and exits. Suppose we call the entered numbers a, b, and c, in that order. If the user has entered valid numbers, the script determines whether :

- the numbers are not the sides of a triangle ($a + b \leq c$)
- the numbers are the sides of an obtuse triangle ($a + b > c$ *and* $a^2 + b^2 < c^2$)
- the numbers are the sides of a right triangle, or ($a + b > c$ *and* $a^2 + b^2 = c^2$)
- the numbers are the sides of an acute triangle ($a + b > c$ *and* $a^2 + b^2 > c^2$)

Exactly one of these conditions must be true. The script should display a message stating which is true. The message should be at most one line long and should include the numbers and indicate what is true about them. For example:

```
$ triangletype 3 4 5
3, 4, and 5 are the sides of a right triangle
```

2. Write a `bash` script named `realname` that accepts any number of words on the command line and for each word, if this word is the username of a user on our network, it displays this username and the full name of the user who is assigned this username (as shown in the network password file which is displayed with the `ypcat passwd` command.) The format of each output line should be

```
username: real-name
```

If there is no such name in the network password file, the script should display nothing. I should be able to run a command such as

```
$ realname sweiss r2d2 tbw
sweiss: Stewart Weiss
tbw: Tom Walter
```

Notice that there is no line for `r2d2`, because `r2d2` does not have an account on our system. The real name of each user is the ***fifth field*** in the output of `ypcat passwd`. The fields are separated by colons (":"). Your program must extrapolate the fifth field efficiently. (Hint: `awk`.) It must also error check the input and exit with a usage statement if the usage is incorrect.

Create a directory named `your-username-hwk3` using the `mkdir` command, where you replace "your-username" by your actual username. For example, I would create `sweiss-hwk3`. Put both scripts into the directory.

5 Grading Rubric

This homework is graded on a 100 point scale. Each script is worth the same number of points. Each script will be graded primarily on its correctness. This means that it does exactly what the assignment states it must do, in detail. Correctness is worth 70% of the grade. Then it is graded on its clarity, simplicity, and efficiency, as described above. Good comments are worth 15%; good design another 5%, and simplicity and efficiency the remaining 5%. Naming all files and directories correctly is 5%.

6 Submitting the Solution

This assignment is due by the end of the day (i.e. 11:59PM, EST) on Monday, October 4. (I give a grace period of six hours after that, so it is okay to submit it by 6:00 AM of the following day.)

There is a directory in the CSci Department network whose full path name is `/data/biocs/b/student.accounts/cs132/hwks` You must put it in that directory.

To submit your project, you must follow the instructions below exactly! Do not deviate from these instructions.

To be precise:

1. Login using `ssh` to `eniac.cs.hunter.cuny.edu` with your valid username and password, and then `ssh` into any `cslab` host. Do not forget this step. You will not be able to run the `submithwk` command on `eniac`.

2. Make sure that you see the `your-username-hwk3` directory that you created in the assignment by typing `ls` and verifying that it appears in the output list. If you do not see it, then either you are in the wrong working directory or you misplaced it somehow.

3. Run the command

   ```
   zip -r your-username-hwk3.zip your-username-hwk3
   ```

   This will create the file `your-username-hwk3.zip.` The `zip` command is a special command that compresses the files in the directory and creates a new file that can later be extracted by the `unzip` command. So it will create a "zip file" named `your-username-hwk3.zip` containing your `your-username-hwk3` directory and the three files it contains. For example, I would run

   ```
   zip -r sweiss-hwk3.zip sweiss-hwk2
   ```

4. Run the command

   ```
   /data/biocs/b/student.accounts/cs132/bin/submithwk 3  your-username-hwk3.zip
   ```

   Do exactly this. Do not mistype it. The command will create a copy of the file `your-username-hwk3.zip` in the directory

   `/data/biocs/b/student.accounts/cs132/hwks/hwk3`

   It will be named `hwk3_`*`username`*, where *`username`* is your username on the network. You will not be able to read this file, nor will anyone else except for me. If you decide to make any changes and resubmit, just do all the steps again and it will replace the old file with the new one. I will be able to unzip the file, extracting whatever files you created. Do not try to put your file into this directory in any other way - you will be unable to do this.

Although these instructions may seem complicated, they simplify the way you submit your work and the way I can retrieve it. If you make mistakes, just start over. If things don't seem to work out, post a question on Piazza with the details included.