CS 252: Systems Programming
Fall 2019
Project 1

Prof. Turkstra

Due Monday, September 2 11:58PM

Goals

A shell is a program that interprets and executes the commands that a user inputs. There are various types of shells. In this lab we will be learning bash scripting. The basic idea of bash scripting is to write a reusable script that performs a task—often involving multiple commands. For example, a user could write test scripts to test the output of a program based on various inputs, or could write a script to backup a directory whenever a change is detected.

BASH Tutorial

The first step for this lab will be to learn the basics of bash scripting. With that in mind, please work through the tutorial here before continuing with the project. Please anticipate spending around 1 hour working these examples. Working through this tutorial will save you a significant amount of time on the project as the examples will be applicable to the project.

REGEX Tutorial

Another important topic to cover is regular expressions (also known as "regex"), which will be vital to completing the first half of the project. This introduction to regex should take roughly 20 minutes and will introduce all concepts necessary to complete the first half of the project. Please see this page for the regular expression tutorial.

Startup

Log in to data.cs.purdue.edu and run the following commands:

```
$ mkdir cs252
$ cd cs252
$ mkdir lab1
$ cd lab1
$ tar -zxvf ~cs252/labfiles/lab1.tar.gz
```

Part 1: Basic BASH (25 Points)

Script Name:	hailstone
Purpose:	to calculate and display a Hailstone sequence.
Parameters:	two, the starting number, and number of elements to generate
Return Value:	zero on success, one if invalid arguments are given.

“Hailstone sequences” are one of mathematics’ unsolved problems in that it has yet to be proved that every starting value in a Hailstone sequence will eventually degenerate to the sequence 4, 2, 1, 4, 2, 1, ... In other words, does there exist a sequence that never settles to a repeating cycle? A Hailstone sequence is defined as follows:

Starting with any positive integer n, form a sequence in the following manner:

- If n is even, divide it by 2 to give $n' = n/2$
- If n is odd, multiply it by 3 and add 1 to give $n' = 3 * n + 1$ Take n' as the new starting number and repeat.

For example, $n = 5$ results in the following sequence:
5, 16, 8, 4, 2, 1, 4, 2, 1, ...

$n = 11$ produces the following:
11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

These are called “Hailstone sequences” because they go up and down in a similar manner to hailstones in a cloud before they fall to Earth.

Your task for this part is simple. You are going to write a BASH script that generates a Hailstone sequence when provided with a starting value, n, and a limit, k. The limit represents how many values in the sequence should be generated.

You should check that the number of parameters is correct, but do not worry about the user providing invalid starting or limit values.

The following are a few sample runs of our solution. Your output should be identical to ours. **We use the UNIX diff utility to grade.** If we have to manually inspect your output you will automatically lose **2 points**.

```
$ hailstone
Usage: hailstone <start> <limit>
$ echo $?
1
$ hailstone 5
Usage: hailstone <start> <limit>
$ hailstone 5 6
5 16 8 4 2 1
$ echo $?
0
$
```

Part 2: Looping with non-integer values (25 points)

Script Name:	compile
Purpose:	to compile and test a series of C source files.
Parameters:	none.
Return Value:	zero on success, one on error.

Assuming that you executed the tar command at the beginning of this lab, you should have the following files in your lab1 directory:

`src1.c, src2.c, src3.c, input1.data, input2.data, input3.data`

The script `compile` should attempt to compile all files beginning with the name "`src`" and ending with ".c". It should then print a message indicating whether the compilation was successful or not.

If compilation was successful, you should then run the generated executable—`a.out`–sequentially on all input files beginning with "`input`" and ending with "`data`".

This means that you should have two nested loops.

Here is some pseudo-code:

```
For each src*.c file in current directory:
    Compile file using gcc -Wall -std=c99 filename
    Check gcc's return value - 0 = success, nonzero = failure (HINT: Remember Bash's "special variables")
    Display appropriate message (see output below)
    If gcc succeeded
        for each input*.data file in current directory
            Display appropriate message (see output below)
            Execute generated a.out binary on each input file
                Eg, a.out < input_file_name
            Check return value of a.out, display appropriate message
```

Here is a sample run of our solution (please note that when we grade your program there will be other files in the directory and many more src and input files). Note that some of the output (output in **bold**) is generated by running a.out (and sometimes gcc) and should not be explicitly generated by your script:

```
$ compile bob
Usage: compile
$ echo $?
1
$ compile
C program: src1.c
Successfully compiled!
Input file: input1.data
The prime numbers between 3 and 5 are: 3
Run successful.
Input file: input2.data
The prime numbers between 3 and 10 are: 3 5 7
Run successful.
Input file: input3.data
Error...lower limit is larger than upper limit
Run failed on input3.data.

C program: src2.c
src2.c: In function ‘main’:
src2.c:57:9: error: expected declaration or statement at end of input
         return 0;
         ^~~~~~
Compilation of src2.c failed!

C program: src3.c
Successfully compiled!
Input file: input1.data
The prime numbers between 3 and 5 are: 3
Run successful.
Input file: input2.data
The prime numbers between 3 and 10 are: 3 5 7
Run successful.
Input file: input3.data
Error...lower limit is larger than upper limit
Run failed on input3.data.

$
```

Part 3: UNIX Commands (25 points)

Script Name:	finduser
Purpose:	to display user information.
Parameters:	none.
Return Value:	zero if match is found, one otherwise.

Write a script that asks the user to enter part of a username to search for in the system's passwd file. Once the search string is entered, you should use **grep** along with **wc**, **cut**, and **head** to generate output identical to the following.

Recall how you can use the pipe | to feed output from one command into the input of another. Finally, you can obtain the requisite output to parse by executing `ypcat passwd`

```
$ finduser bob
Usage: finduser
$ finduser
Welcome to User Finder!
Please enter part or all of a username to search for: guest
I found 250 matches!
You might want to be more specific.
The first of these matches is:
Username: guest105
Name: Temporary Account
Home Directory: /homes/guest105
Shell: /usr/local/etc/disabled
Search complete on Wed Aug 22 16:07:38 EDT 2019
$ finduser
Welcome to User Finder!
Please enter part or all of a username to search for: turk
Match found!
Username: turkstra
Name: Jeffrey Alan Turkstra
Home Directory: /homes/turkstra
Shell: /usr/local/bin/bash
Search complete on Wed Aug 22 16:07:48 EDT 2019
$ finduser
Welcome to User Finder!
Please enter part or all of a username to search for: Turkstra
Sorry, I could not find that username.
$
```

Part 4: File I/O (25 points)

Script Name:	matrixmul
Purpose:	multiply two matrices
Parameters:	two, first matrix input filename and second matrix input filename
Return Value:	zero on success, one on error.

First, read about arrays in BASH.

Your job is to write a script that multiplies two matrices. Each matrix is provided in an input file. You may assume that the files contain well-formed matrices. That is, each row has the same number of columns and the values are all integers. Each value is whitespace separated with each row on its own line. The matrices may be any size.

You should read each file into an array and determine whether the matrices can actually be multiplied (the number of columns in the first matrix must match the number of rows in the second). If not, emit an error per below. If so, multiply them and output the result.

When finished, your script's output should match the following exactly:

```
$ matrixmul
Usage: matrixmul <matrix1> <matrix2>
$ echo $?
1
$ matrixmul m2
Usage: matrixmul <matrix1> <matrix2>
$ matrixmul noread m2
Error: noread is not readable!
$ matrixmul m2 noread
Error: noread is not readable!
$ cat m1
4 3 2
1 9 5
$ cat m2
5 4 9 5
3 8 7 6
10 11 5 80
$ matrixmul m1 m2
49 62 67 198
82 131 97 459
$ echo $?
0
$ matrixmul m2 m1
Error: incompatible matrix dimensions!
$
```

Part 5: File I/O and Regular Expressions (25 points)

Script Name:	verify
Purpose:	to display validation information regarding data..
Parameters:	one, input filename.
Return Value:	zero on success, one for improper arguments, two if filename is not readable.

Write a script that reads in the specified file and determines the validity of each field in the following manner:

Field 1 - Date of Birth
MM/DD/YYYY

- **MM** must be in the range 1-12 and may have a leading 0 when it is 1-9.
- **DD** must be in the range 1-31 and may have a leading 0 when it is 1-9.
- **YYYY** may take on the following formats:
  - 19YY where YY is any integer digit
  - 20YY where Y is any integer digit
  - YY where Y is any integer digit
  - Note that there must always be 2 or 4 digits.

Valid Examples: **01/02/1999**, **1/2/1999**, **1/2/99**, **11/13/2010**, **11/13/10**

Invalid Examples: **0/1/22**, **1/0/22**, **00/1/22**, **1/00/22**, **13/01/1999**, **1/2/1**, **1/2/1876**

Field 2 - Name
Last, First M.
Last - must begin with a capital letter and be followed by one or more lowercase letters.
First - must begin with a capital letter and be followed by one or more lower case letters.
M. - must be a single capital letter followed by a period (.).
This part (including the preceding space) is optional.

Valid Examples: **Davis, Mike**; **Leppla, David A.**

Invalid Examples: **D, Mike**; **L, David**; **smith, Mike**; **Jo, Fr a.**

Smith, Joe
↑ with an extra space on the end

You may assume that each line in this file will contain exactly one address and name and that there will be a dash (-) separating the two fields.

Sample Output:

```
$ verify
Usage: verify <filename>
$ verify noread
Error: noread is not readable!
$ verify birthdays
Line 1 is valid.
Line 2 has an invalid name.
Line 3 has an invalid birth date.
Line 4 has an invalid name and birth date.
$
```

You should test your script thoroughly. The sample input file provided does not come anywhere near testing all of the corner cases—that's your job. We will test using files with *many* more data.

Part 6: Turn in Your Files

1. Log in to data.cs.purdue.edu
2. Type:

   ```
   $ cd ~/cs252
   $ turnin -c cs252 -p lab1 lab1
   ```

   Important: previous submissions are overwritten with the new one. Your last submission will determine your grade.

You can verify your submission by running:

```
$ turnin -v -c cs252 -p lab1
```

Important: do not forget the -v flag, otherwise your submission will be replaced with an empty one