# CSE 391, Winter 2019
# Assignment 6: Advanced Shell Scripting
### Due Tuesday, May 21, 2019, 11:59 PM

This assignment focuses on Bash shell scripting at a more advanced level. Electronically turn in file `gradeit.sh` as described in this document. You will also want the support files from the Homework section of the course web site.

For this assignment, you will write a script `gradeit.sh` that "grades" solutions to a hypothetical homework assignment. You're given a set of students' solution files, each of which is supposed to be named `gettysburg.sh` (though some students used the wrong file name...), and your script will run each of them one at a time, examining the output to see if it matches the expected output. Your script will also examine the code to see whether it has enough comments. The output from a run of your program will look like this:

```
$ ./gradeit.sh 50
Retro Grade-It, 1970s version
Grading with a max score of 50

Processing bensonl ...
bensonl has correct output
bensonl has 3 lines with comments
bensonl has earned a score of 50 / 50

Processing dravir ...
dravir has incorrect output (1 lines do not match)
dravir has 0 lines with comments
dravir has earned a score of 38 / 50

Processing joeblow ...
joeblow has incorrect output (4 lines do not match)
joeblow has 5 lines with comments
joeblow has earned a score of 30 / 50

Processing oterod ...
oterod did not turn in the assignment
oterod has earned a score of 0 / 50
```

To get started, download the following support files from the course web site:

- **`students.tar.gz`** : This file contains a set of fake student solutions to examine. Extract this file with the `tar` command (see lecture 6 for information about how to decompress a `tar` file). It will create a set of student folders and files such as:

  ```
  students/bensonl/gettysburg.sh
  students/oterod/GETTYSburg.sh
  students/stepp/gettysburg.sh
  ```

  Each student's directory is supposed to contain a file representing that student's homework solution (though some students have not turned in the file properly, so it may not exist or may have the wrong name).

  Note that you are not targeting the specific students shown above; your code should not specifically mention names like `stepp` or `bensonl`. Your code should process *all* students in the `students` directory.

- **`expected.txt`** : This file contains the expected homework output. Download this file into your assignment folder. This is the text you'll compare to each student's output to see whether it is correct.

You must write a script `gradeit.sh` that gives each student a score on the assignment. Your program should accept the maximum score as a command-line argument. For example, to run it with a max score of 50 points, you would type:

```
$ ./gradeit.sh 50
```

If the user does not pass a value for the maximum points, your script should print the following error message and exit:

```
$ ./gradeit.sh
Usage: ./gradeit.sh MAXPOINTS
```

You do not need to check this argument for validity. If an argument is passed, you may assume it is a positive integer.

## Processing Each Student:

If your script is passed an argument, **it should assume that the current directory contains** 1) a **`students/`** directory whose contents resemble those in the sample files you have been given (containing one directory per student, each named with the student's id) and 2) **`expected.txt`**. Your script will examine each student in the `students/` folder as follows:

- Run the student's `gettysburg.sh` file, using the student's folder as the working directory. (You may want to capture the student's output into a file.) Note that the student script files may not have execute permission, so you should manually invoke `bash` to run them, such as: `bash ./gettysburg.sh`
- Compare the student's output against the expected output file using `diff`. Run `diff` so that it **ignores all white space** (explore the options: –w and –b). Count the number of lines that do not match. T**o simplify things**, we will consider any line of `diff` output containing a < **or** a > to count as 1 line of unmatched content. Produce output in this format:

```
bensonl has correct output
```

  or:

```
smith has incorrect output (8 lines do not match)
```

  For each unmatched line, deduct 5 points from the student's score. For example, if the `diff` output has 4 lines that have < or >, the student should lose 20 points. **If the student loses more points than there are points in the assignment, the student should receive 0 points.**
  *(Hint: You can look for each of < and > separately and total them, or search for `"[<>]"` to match either one.)*
- Check whether the student has sufficient code comments. Comments are worth 7 points on the assignment. A comment is defined as any line that contains a `#`. A student must have 3 or more lines of comments (if the file has an initial `#!/bin/bash` line, we will count that as a comment line, too). A student that has fewer than 3 lines of comments should lose 7 points from his/her assignment score, down to a minimum of 0. Produce output such as:

```
toddk4 has 5 lines with comments
```

- Output each student's score on the assignment. If a student `jones` has no differing lines of output and 5 lines of comments in his script, the score output for that student would be:

```
jones has earned a score of 50 / 50
```

  If student `davis` has 3 differing lines of output (-15) and 2 lines of comments (-7), the score output would be:

```
davis has earned a score of 28 / 50
```

If the student did not turn in the program or incorrectly named the file, the student gets 0 points on the assignment. (See the various `if` tests from lecture such as `-d`, `-e`, etc. to see whether files exist or have various properties.) If the user `lewis` did not turn in a proper `gettysburg.sh` file, you would output the following:

```
lewis did not turn in the assignment
```

See the course web site for several complete sample output runs of the program.

You may assume that no student's program tries to do anything evil to your computer, such as erasing all your files. You may also assume that students' code will not lock up and get stuck in any sort of infinite loop. It is okay if your `gradeit.sh` script creates temporary files while doing its work, though it would be best to remove them as you go along.

## Development Strategy (suggested):

- Make your script able to simply output the names of all of the students to be processed.
- Make your script able to simply run each student's `gettysburg.sh` program and show its output on the terminal.
- Some tasks in your script will involve running Unix commands and capturing their output with $( ). Since such commands run silently as their output is being captured, consider simply running the command first to see that the command is running properly and producing results that you expect.
- Remember to use `echo` statements to output partial results, computations, commands, etc. to verify them.

Your script should work if placed in any directory containing **`expected.txt`** and a **`students/`** directory. Do not hard code in a specific directory on your machine or your home directory on attu. **This is important for grading!!**

Each Linux/Unix box can be slightly different; for full credit, your commands must be able to work properly either on the CSE virtual machine image, `attu`, or on the CSE basement lab computers. For reference, our solution to this assignment is 60 lines long (40 "substantive" lines on the CSE 142 Indenter tool).