Requirements Testing Exercise
With Test Automation

PART I:

1. Study the Postfix Calculator Functional Specification.
2. From the Specification, derive a set of Testing Requirements

   NOTE: we want to be sure that you can do Part II, so after you have created a few testing requirements, go on to Part II and complete the tasks listed there for those few testing requirements. You can then return to Part I and complete the task, in class and afterward.

PART II:

3. The rest of this exercise needs to be performed in Unix/Linux Environment.
4. Download the Calculator executable from Canvas.
   make sure it has executable permission, and is on your path so that you can execute it. This program implements the requirements in the specification you studied in Part I.
5. Create and automate test cases that account for all of your testing requirements using a Unix shell script. Your script should execute each of your test cases, in turn, and for each test case it should do the following.
   a. Print a message saying which test case is being executed.
   b. Save the output of the test run to a file specific to that test case.
   c. Compare the saved output of the test run to a previously saved output that contains the proper output of the test case.
   d. Report cases in which the current saved output and previously saved output differ; these are "regression errors".
6. Make sure that it is clear, in your test script, which testing requirements are covered by each test, and make sure that it is clear, in your testing requirements document, which test cases in the test script cover those requirements.

DELIVERABLES:

Organize your materials, including your testing requirements, test script, any input files used, and all output files used, in a directory. Use informative file names, but also include a "README" file that describes the contents and structure of the directory. Also include a file containing the output of your test script when you execute it. Make sure that this output includes at least one example where a difference in test results is detected. Zip that directory (on Linux, "tar" and "gzip" do the job).

By the end of Thursday, April 6th, submit the directory on Canvas

NOTES AND SHELL SCRIPTING HINTS:

- Your script needs to have "execute" permission. Use *chmod u+x SCRIPTNAME* to set the permission.
- Your PATH must be set to include the location of the script, or you can specify the script's location using a complete path name.
- To print messages from a script, you can use the "echo" command; for example, *echo "testing"* prints out the message "testing"
- Any text following the "#" character is considered a comment; you can use that to include traceability information in your script.
- To cause a program to read its input from a file, instead of from the keyboard, one approach is to redirect the program's input. For example, *calc.exe < FILENAME* causes calc.exe to read its input from the file "FILENAME".
- An alternative to creating numerous input files is to embed, in your script, code that creates an input file. For example, you can use the command *echo "2 1 + = q" > infile* in your test script to create a file containing that sequence of characters, then have the script issue the command to run calc.exe, redirecting input from that file. This lets you reuse the same file for each test, and allows you to see the inputs being used directly in the test script, instead of having to look inside lots of little files.
- To cause a program to write its output to a file, redirect the output. For example, *calc.exe < FILENAME > OUTFILENAME* causes calc.exe to read its input from the file "FILENAME" and write its output to the file "OUTFILENAME".
- Two files' contents can be compared using the Unix "cmp" command or the Unix "diff" command.
- If you just use the "cmp" or "diff" command, then, if output files differ from prior output, the results of the command (the list of differences) is displayed. This can get ugly. An alternative, with "cmp", is to use something like the following: *cmp –s outfile prioroutfile || echo "test failed"*. In this command, the –s flag suppresses output of the differences. However, if a difference is detected, the cmp command sets a return value of "1" in an environment variable. As it happens, the "||" causes that variable's value to be checked, and if its value is 1, the echo command is executed. Now you just see messages output on tests that do fail, without all sorts of other ugly output. There is probably a way to do this with the "diff" command too.