Scott Crawford: Scott-Crawford

Sumanyu Gupta: sumanyugupta

<https://github.com/sumanyugupta/CS1632_Deliverable6>

CS 1632 – Deliverable 6: Testing Strategy for RPN++

**Review of Program Quality**

REPL mode: Functions properly, variables are not persisted across executions. Invalid commands result in errors and line is properly ignored.

File mode: Functions properly, multiple command line arguments are interpreted as paths and program executes a concatenated file of RPN commands. Variables persist in the span of one execution.

Keyword usage: Function properly, only three keywords are used and allow user to exit the program, declare variables, and print resulting output(s). Keywords follow proper syntax as specified in the requirements.

Error handling: Function properly, five error codes are used to mark any invalid input or execution statements in REPL or File mode. User sees error message immediately and can react accordingly.

User input for REPL: Functions properly, valid keywords and RPN commands are processed and results are displayed as output immediately.

Program style/Rubocop: 4 minor repeated errors that do not impact program functionality. Complexity and length of class/method considered low priorities.

**Areas of Concern**

After writing many unit tests and testing manually, we were unable to find any defects in our program.

**Testing Strategies Used**

To begin, we used the smoke testing strategy to confirm that the core functionality of our REPL and file modes was proper and could be tested further. After finding some major bugs in our code, we began whiteboarding a list of known issues alongside one/two word answers on how to fix them. Resolving the issues led to us performing exploratory testing for a few hours, where we learned more about our program’s numerous edge cases and how simple statements that we hadn’t thought of before resulted in critical bugs. For example, we discovered that our system did not read command line arguments as paths to files, instead as strings. Researching on Stack Overflow allowed us to use the Ruby File class to deal with file arguments. Another issue we kept stumbling upon is a random new line character being printed out after outputting results of RPN commands. We found out that this was an issue caused by the terminal emulator a member of the group was using. The Linux subsystem on Windows functions differently than the built-in command prompt, so we had to accept that the program output would always look slightly different on one machine. Since the requirements stated to think of all possible test cases, we also used a path-based testing approach to conclude the development of the program. For every requirement, we thought of all possible input values and the expected error or mathematical value the execution was supposed to return. This segment of testing was arduous as there were many methods and therefore many possibilities of inputs.

We spent about 30 minutes smoke testing because it did not take long to diagnose the bigger causes of our bugs and implement their patches. Exploratory testing took us between 1-2 hours to find more complex bugs and whiteboard their fixes. Finally, path-based testing and actually writing the unit tests took between 6-8 hours because of the sheer number of input possibilities we had to consider. We started with simple tests that we expected to pass and then added more and more complex RPN expressions in order to break our program. Once we had tested exhaustively and up to our satisfaction, we moved on to the next requirements. However, a large portion of the testing time was solely repeated work because often times, a fix to a bug would create a previously fixed bug to be reborn. Thus, we spent a lot of time performing regression tests as well.