Rob Colleran: rjc59

Testing Strategy for RPN++

**Quality Review**

File processing – All file processing meets the requirements and has been tested

REPL mode – All features of REPL mode meet the requirements and all have been tested

QUIT, LET, PRINT commands – These commands are all supported according to the requirements and all have been tested

Postfix expression evaluation – All core functionality for evaluating postfix expressions is there and unit tests show that it works.

Output – All output has been tested to be according to the requirements

Error handling – All errors given in the requirements and more are implemented and tested. There is one minor case where an error doesn’t match the given sample output, but this is covered by the requirement stating that, when handling multiple errors, we can choose to display any one of them.

**Issues and Enhancements**

When entering a command such as “a 3 +” where a is an uninitialized variable, the error message “Line n: Unknown keyword a” is displayed, as opposed to the error message “Line n: Variable a is not initialized”, which would match the sample output. However, these are both valid errors in this case (as a is neither an initialized variable nor a valid keyword), and the requirements state that: *In cases where multiple errors on a line would be appropriate, you should print out only one error. It is up to you to decide which one*. Therefore, this doesn’t seem to count as a defect.

No defects were found via unit tests and exploratory testing. The program took 240 ms (measured with Measure-Command) to run on a file with 240 lines of assignment and addition commands, and this is acceptable performance.

In addition to the error messages specified in the requirements, additional messages for catching file not found, division by 0, invalid variable name, and invalid token errors were implemented.

**Testing Strategy**

I wrote mostly unit tests to verify that the methods implementing the core functionality worked correctly and handled errors. I did this because catching errors in my code at a lower level would make developing the more user-facing aspects easier. I knew that most errors could be detected during the evaluation of an RPN expression, by lower level methods that had already been tested. Therefore, while coding the user input/output aspects, I just had to rescue from any errors I knew could be raised by those methods that had already been implemented. Unit testing accounted for 85% of total testing time and effort.

After user input and output was implemented, exploratory testing was used to verify that everything worked according to the given requirements and sample output. This helped catch a few errors that were difficult to catch with unit testing. For example, when testing a command like “3 4” my program was outputting “Line n: 1 elements in stack after evaluation” instead of “Line n: 2 elements in stack after evaluation”. This was because my evaluation function was always popping the last value off the stack when it completed to yield the result, which wasn’t a technically incorrect behavior and therefore wasn’t detected as wrong by any unit test. To give correct output, I moved the checking of the number of elements left on the stack to the end of the evaluation function. This way, it would raise an exception rather than bothering to pop the stack and yield a result if more than one element was left on the stack at this stage. Exploratory testing accounted for 10% of total testing time and effort.

Lastly, basic performance testing with Measure-Command was used to verify that the program was reasonably performant. It consistently took only a few hundred milliseconds to run files with hundreds of lines, so this seemed reasonable and no code was changed at this stage. Performance testing accounted for 5% of total testing time and effort.