CS 1632 DELIVERABLE 6: RPN CALCULATOR

https://github.com/mho8/1632_D6

Lucas Brennan -lbrennan26 (github)

Michael Okonski – mho8 (github)

Professor William Laboon

April 18th, 2018

Program Structure

After determining whether to run in file or REPL mode, our program works by executing one line of RPN code at a time. The input string, be it from the user or some file, is split by whitespace into an array of elements. Each element is then analyzed, and a string is generated to represent the types of elements in the array. The elements are assigned numbers based on the following conditions:

> 1 => Integer
>
> 2 => Variable
>
> 3 => Operator
>
> 4 => Keyword
>
> 5 => Unknown Keyword
>
> 0 => Not a Token

Our program will then match the generated string against a variety of REGEX options and perform the necessary operation. If no REGEX is matched, then the program assumes that the pattern of tokens is not one it can calculate, and it will inform the user that it was unable to process that line (and quit in file mode).

Overall Quality

After thorough testing, all aspects of our program appear to behave as intended. The following subsystems and classes work together to provide complete functionality:

GREEN: RPN =>

> This is our main file, which calls all other code and initializes the variables hash, as well as runs file/REPL mode depending on the flag determined by args_checker

GREEN: ARGS_CHECKER =>

> Runs file mode if any arguments are given, and REPL mode if none are given.

GREEN: FILE_MODE =>

> Runs each line of one or multiple files through the program and exits upon completion or an error.

GREEN: REPL_MODE =>

> Runs each line entered by the user through the program and exits only upon the keyword QUIT.

GREEN: MAIN_PROGRAM =>

> A group of files and classes that handle the actual execution of each RPN calculation. Thoroughly tested with no known bugs.

Areas of Concern

After testing there are currently no known defects to our program. Because of the nature of the program in matching patterns with REGEX, we believe that any edge case errors we have not thought of will be caught by our program. This is because it was much easier to think of all acceptable cases and handle those, than think of all unacceptable cases. Using this strategy, our program will simply inform the user it was unable to parse a line if all acceptable cases are tested and none match the current line.

Our program does have a single Rubocop error: an assignment branch condition size of 17.49, exceeding the recommended 15. This error was left in the program because all the variables passed into that specific method definition were necessary for the execution of the program. Additionally, 17.49 only exceeds the cap of 15 by 16.6%, which should have a relatively low impact on performance and traceability.

Testing Strategy

A majority of our testing was done at the unit level. Each class for our program was written with unit tests in mind, and methods were written to be as simple as possible to allow for unit testing to encapsulate a majority of our code. This strategy is what allowed us to have over 99% code coverage in our test suite.

Exploratory testing was useful in discovering different cases we would need to accommodate. For each new case, new methods were written which would then be unit tested, to maintain a virtually flawless code coverage.