Nick Sallinger - nicksallinger

Wei-Hao Chen – yoshino0705

CS 1632 - DELIVERABLE 6: Testing Strategy for RPN++

https://github.com/yoshino0705/CS1632_Deliverable_6

Overall Quality

Subsystem	Status	Notes
REPL Mode	Green	Test plan passed with no defects. Able to enter REPL mode by not specifying command line argument.
File Mode	Green	Test plan passed with no defects. Able to read command line file and execute.
Parse Commands	Green	Test plan passed with no defects. Reads in multiple files specified on command line and executes in that order. QUIT command in file will stop execution of all files afterwards.
Keyword Recognition	Green	Test plan passed with no defects. System recognizes case-insensitive Quit, Print, and Let commands and executes them.
Error handling	Green	Test plan passed with no defects. System raises and notifies user of error in file or REPL argument.
Utilities	Green	Test plan passed with no defects. Helper functions work correctly and aid in parsing and verifying commands

Areas of Concern

One of our area of concern is performance. The requirements do not have a strong performance benchmark for real-time execution. To meet this requirement, we were able to use faster data structures, such as hashes, where an array would have been sufficient.

Another concern is with non ascii characters as inputs:

SUMMARY: Entering non ascii characters would cause errors

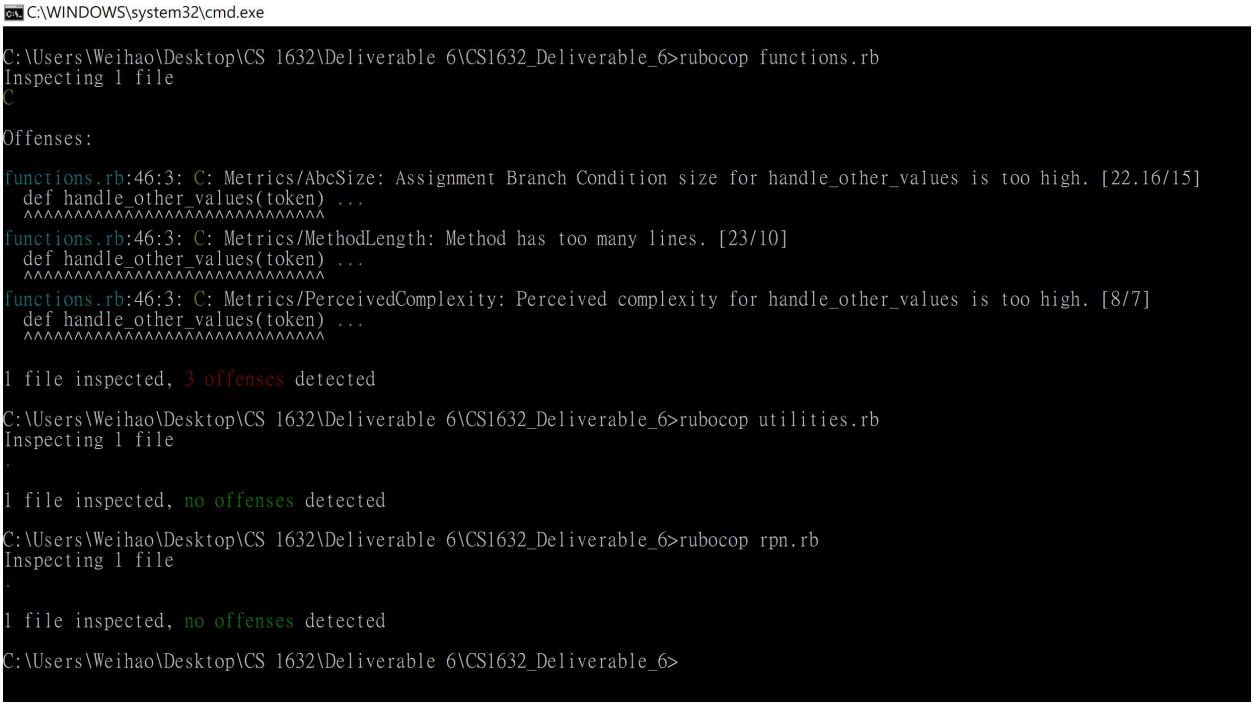
DESCRIPTION: When entering a utf-8 character as variable name, errors pop out.

REPRODUCTION STEPS:
1. start the program via "ruby rpn.rb"
2. when prompted, enter: let □ 5

EXPECTED BEHAVIOR: Outputs "Line 1: Invalid variable name"

OBSERVED BEHAVIOR: ArgumentError occurs

Rubocop Outputs



rpn.rb: no offenses

utilities.rb: no offenses

functions.rb: 3 offenses

46:3: C: Metrics/AbcSize

46:3: C: Metrics/MethodLength

46:3: C: Metrics/PreceivedComplexity

Testing Strategies

We began testing the system by using exploratory testing. This was able to show us areas that we wanted to focus more attention on, and possible edge cases that we could test further with unit testing. The first unit tests we wrote were to test the basic functionality of the program, this included the system's ability to parse command line arguments, file inputs, keywords and parsing. The first unit test we wrote were to test the LET keyword. We tested if valid syntax, invalid syntax, and case sensitivity. We wrote the syntax tests because it effected large parts of the execution of the system, and case sensitivity was an explicit requirement. Tests for the execute_keyword (token, keyword) function tested the systems ability to parse, recognize, and execute arguments. For this we tested valid and invalid keywords, valid and invalid variable names, and whether the argument was entered in REPL mode or from a file. These tested the systems response to bad input from the file or users, and verified an error was displayed at the correct parts of execution. Test were also written for the handle_other_values function, that dealt with variable values and arithmetic. During these tests we were able to test that proper errors were shown in the correct situations. We also tested our helper function in utilities.rb. These functioned helped verify the validity of arguments before they were parsed in functions.rb. Tests we wrote for this class were not to verify any requirements were met, but to verify that they worked correctly for our needs, as the rest of the system depended on them.

Most of our time was spent writing unit tests. We were able to verify many of the requirements throughout the system with unit testing. This was because the system relied on text input, so it was easy to simulate these inputs, rather than a GUI. We were able to spend a good amount of time coming up with edge cases to test as well. Many of the edge cases lead to error codes being thrown, so through testing we were able to verify that they happened at the right time.

Using exploratory testing, we were able to quickly check functionality and verify many requirements. After writing a new part of the system, we were able to use exploratory testing to quickly enter the program and poke around to see if things were working. Exploratory testing was used from the beginning of the project until the end because it offered us a very quick way of verifying the work we did. Though exploratory testing helped us greatly, we never relied on it totally for verification, especially for the subsystems that performed many calculations, because we could not verify what was going on behind the scenes. We were able to verify simple requirements without writing unit tests, such as REPL lines beginning with "> ", but for the functionality that a large part of the system depended on we always wrote unit tests, rather than relying on exploratory testing.