Project #: \_2\_

Semester: Spring ‘22

Names: \_Peter Gocon, Joshua Grieve, & Jordan Hopkins\_

I. Requirements: Restate the problem specification and any detailed requirements in your own words.

We were instructed to build a parser for the CMinus language, particularly, a Recursive-Descent lookahead 1 parser. We also had to provide the EBNFs we used, alongside the first and follow sets.

II. Design: How did you attack the problem? What choices did you make in your design, and why? Show class diagrams for more complex designs.

First, we redid the CMinusScanner to use states, as it was supposed to, then we added the bear-bone essentials, which includes a new exception and the CMinusParser class. We then completed the EBNF (and first and follow sets), then after we finally got them right, we started to construct the parser, making a new class and a new parse function according to what we had whenever we needed it (starting with Program, then moving to the Delcarations, Param, etc.). After we wrote all the classes and parses, we debugged our program and fixed the bugs that we found, using the file we used to test the CMinusScanner. Afterwards, we wrote up some more inputs, found more bugs, until we were out of ideas for weird but correct syntaxes that might not work. After that, we made some test cases to explicitly have syntax errors in them.

III. Security Analysis: State the potential security vulnerabilities of your design. How could these vulnerabilities be exploited by an adversary? What would be the impact if the vulnerability was exploited?

Since we wrote it in Java, there are no vulnerabilities. An error in tokens or syntax will just lead to the parser being unable to finish, but any potential data that could have been lost will have been garbage collected.

IV. Implementation: Outline any interesting implementation details in your solution.

We essentially had interface functions to work with the CMinusScanner to check for error tokens, so those checks were all done in one spot.

V. Testing: Explain how you tested your program, enumerating the tests if possible. Explain why your test set was sufficient to believe that the software is working properly, i.e., what were the range of errors for which you were testing.

We started off using the file we made for Project 1, and once that one finished correctly, we came up with two more test files that tested to more unique aspects of the CMinus language. Those ones help us catch a couple more errors. We also looked in debug mode to make sure the AST was forming correctly, since the output didn’t show that or parenthesis. Afterwards, we gave it some explicitly wrong inputs to see if it gave us the correct error messages.

VI. Summary/Conclusion: Present your results. Did it work properly? Are there any limitations? NOTE: If it is an analysis-type project, this section may be significantly longer than for a simple implementation-type project.

It worked properly for all of the test cases we tried, both for catching syntax errors and on correct strings. The only limitation is that it can only do the CMinus language, which is a very limited language in terms of things that it can do.

VII. Lessons Learned: List any lessons learned. For example, what might you have done differently if you were going to solve this problem again?

Recursive descent parser is more difficult than it first appears. We would have probably been more careful to ensure that we are consuming the right tokens in the right places and not consuming them in the wrong places, as well as having the switch cases for them be more explicit in what they are checking for.