Compilers Checkpoint One Report

Kristan Samaroo

Indeep Farma

CIS4650

March 9, 2022

1 - Summary

Checkpoint One for the Compiler's Implementation project involved implementing the following:

- Scanner that generates a sequence of relevant tokens
- Parser that generates and shows an abstract syntax tree
- Error recovery, that reports errors and continues the parsing process

Given to us, was the “Specification for the C-Language” document which we derived information from to develop our scanner that contains the relevant tokens such as keywords like ‘else’, ‘if’, ‘int’, etc. As well as special symbols such as: ‘+’, ‘-’, ‘*’, ’/’, etc. The parser will analyze the syntax of the input files based on the grammar given in the “Specification for the C-Language” document, and then the compiler will print out an abstract syntax tree based on the input.

2 - Related Techniques and Design Process

To develop our Scanner and Parser for Checkpoint One, we decided to think things through and develop a plan of action - this involved using an incremental approach as we went.

Task One - Basic Understanding and Design

Our first goal was to develop a basic understanding of what was being asked of us to do for this checkpoint. This involved reading and reviewing the different documentation given to us, and reviewing our previous warm-up assignment (since it involved creating a parser). It also involved reviewing the sample starter code that was provided to us by Professor Song. Once we understood the basics and the working mechanisms of those files, we were able to finally start the assignment with some idea of how to work towards our end goal.

Task Two - Scanner

To start off the assignment, we had to make sure we were able to parse the input information properly. This meant that we must start with creating the actual scanner. Luckily, the tiny.flex file that was in the SampleParser provided by Professor Song had a wonderful foundation laid out for us to use. We cross-examined the SampleParser, and "Specification for the C-Language" document to create our own cm.flex file that contained all the symbols, and regular expressions needed to parse the input file.

Task Three - Parser

The process of implementing the parser and corresponding CUP files for this assignment required us to build grammars derived from the "Specification for the C-Language" document provided to us by Professor Song. We decided to follow the approach that Professor Song used by trying to create an interactive approach to the cm.cup file. So, the initial file starting was similar to tiny.cup.bare where it has defined essentially all token types but no grammar rules - we used this to make sure our scanner was working. Then we implemented a bare-bones grammar that corresponded to the tiny.cup.rules file. This produces no output, but makes sure that the grammar works. Next, we created what was essentially like tiny.cup.layered which uses the same grammar rules that were created, except this time our goal was to produce the abstract syntax tree. So we made sure to work on the ShowTreeVisitor.java file to be able to do that. Once those files were finished, and we ran the program, we were able to see a working Abstract Syntax Tree being printed by the parser.

3 - Lessons Learned

This checkpoint allowed us to learn a significant amount about the inner workings of a compiler. In the first warm up assignment we learned about how a scanner works, in this assignment we dove deep into how abstract syntax trees, parsers and scanners can be integrated together in order to verify program structure.

Another important lesson we learned was the importance of using version control while working in groups. There were often times when we had to abandon branches because our understanding of a component of the assignment was incorrect. Without the use of GIT for version control, we would not have been able to keep our development process as organized and efficient.

4 - Assumptions and Limitations

While working on the development of our project, we made a couple assumptions. We assumed that code we created would be run against a valid “C Minus” file. We assume that the only keywords that will be used throughout the file are: “if”, “else”, “while”, “int” and “return”.

Some limitations of our project include only being able to parse a small array of keywords for C (as mentioned above). Also, the possibility of our regular expressions not recognizing/ matching certain words.

5 - Possible Improvements

Some ways in which our code could be improved include the following:

- Improved/More thorough error handling. Error handling could be implemented to catch a much wider array of different types of syntactical errors.

- Further simplification of grammars. Our grammars are based solely off of the "Specification for the C-Language" document. It is possible to create simpler grammars from this.

6 - Team Member Contributions

Kristan Samaroo

- Writing CFGs representing "Specification for the C-Language" syntactical rules
- Writing abstract syntax tree classes
- Modifying Makefile/ adding targets for ease of execution
- Writing Documentation

Indeep Farma

- Creating JFlex Scanner regular expressions
- Implementing methods within ShowTreeVisitor.java
- Creating Test Files
- Writing Documentation

7 - Acknowledgements

The code for this project was based off of Dr. Fei Song's starter code provided to us for the assignment. We followed his recommendation for the implementation process from the lecture slides and the documentation he provided in the starter code. The grammars defined in our CUP file are based directly off of the "Specification for the C-Language"