CIS*4650 Project:
Milestone 1

Ben Carlson 1044277 carlsonb@uoguelph.ca

Conor Roberts 1056167 crober35@uoguelph.ca

March 2022

Summary (TLDR)

This checkpoint/milestone involved the creation of a scanner, parser and abstract syntax tree for the C- language. The primary deliverables for this milestone are as follows:

1. Create a scanner that generates a sequence of relevant tokens
2. Create a parser, utilizing CUP that generates and shows an abstract syntax tree
3. Create robust error recovery, that will report errors and allow the parser to recover

Many principles, including the production rules and scanner regex within cminus.cup and cminus.flex respectively were created from the C- specs file provided.

Our program will take a .cm file, scan it into a relevant stream of tokens via JFlex and then parse it into an AST utilizing CUP.

Features of C- Language
Reserved keywords:

1. else, if, while
2. types: void, int, bool
3. operators: +, -, /, *
4. logical operands: $\|, \&\&, != , ==, <, >, <=, >=, =$

Techniques utilized
When creating the program, we attempted to follow best practices of OOP, abiding to SOLID. Each stage of the development process was done incrementally, optimistically we planned to test each module separately, however, this changed as the scope of the project became more visible to us. Our development process followed 5 stages with one final debug at the end.

Basic Understanding/Design
Following the warm up assignment, we had a working understanding of JFlex and how to utilize it to our advantage. This was only the tip of the iceberg for this milestone, where we had to read docs/experiment with CUP and the provided files to further our understanding. This was done by comparing specs for the tiny language to the c minus specs and then translating rules accordingly.

Scanner
After getting a base understanding of the file structure and everything related, we started by tackling the scanner portion of the project, considering we already had a solid understanding of JFlex. This required a little bit of preliminary setup in the cminus.cup file so that we could return the correct output tokens as opposed to our previously created token class.

Absyn
Once the scanner was built, we split into two. The parser and absyn classes could be built without needing each other. Building absyn was primarily done by referencing the lecture slides material. The goal was to build the most in depth, but maintainable class structure since we are only a team of 2 creating this project.

Parser
We first laid out a skeleton for the CUP rules by translating the specification file provided. Then, we were able to fill in the body of each CUP rule with our class definitions based on our understanding of the content to be matched. Finally, we added error checking on to as many common error cases as we could so that users would be informed of syntax issues.

Debugging
We grouped up again and pair programmed the remaining aspects of this milestone as well as debugging the entire program start-to-finish. This program was hard to debug in separate modules, due to the output of Scanner.java and its relation to cup. We ended up clearing all of our errors and warnings before moving onto testing our provided and custom files.

Learning Experiences

- Learned about different modules of a compiler and their interaction
- learned how compilers verify program structure and syntax
- Importance of using version control (git) for increased efficiency and organization

Assumptions
Required software

1. JFlex
2. Java/JDK
3. CUP/JCUP

Limitations
Supported files

1. .cm cminus files

Improvements

1. Grammar Optimization
2. More detailed error reporting

Contributions

Ben

1. Created AST
2. Created Tree visitor functions
3. Wrote Scanner code
4. Refactored cup rules
5. created documentation

Conor

1. Implemented CUP rules
2. Debugged CUP rules
3. Extended JFlex Rules
4. Implemented logic and embedded code in CUP

Acknowledgements
This project was heavily based on/inspired by resources provided by Dr. Fei Song's compilers content. We were provided a rough idea of the implementation process as well as a portion of starter code. The grammars created in CUP are heavily based off the the C- specs. The absyn class structure is heavily based off of lecture content.