Compiler Implementation Project - Checkpoint One Report

Nathan Brommersma, Ahmad Sawan, Jacob McKenna

March 3rd, 2025

CIS*4650 - Dr. Fei Song

1. Introduction

This report documents our implementation of a compiler for the C- language, focusing on the first checkpoint requirements: the scanner, parser, and error recovery mechanisms. Our team used JFlex for scanner generation and CUP for parser generation, following the specifications provided for the C- language.

2. Implementation Overview

For our implementation we have completed the scanner, parser, and error handling as per the checkpoint description. After determining how our program would be designed (Documentation section 3.1), we implemented in steps starting with the parser.

The integration between JFlex and CUP was set up in the sample parser, which was helpful for us to integrate the same way. We used “java_cup.runtime.Symbol” in our cm.flex file and created each token to be compatible with our CUP. As we implemented our cup file first we stated each terminal and non-terminal, based on the tokens in the C-Minus Specification document and the suggested rules for our language. It took us the longest to implement the syntactic rules, as the learning curve for CUP syntax was difficult for us. Additionally we had to add our own files for each of the non-terminals in the “absyn” folder. Overall, our implementation was very similar to

that of the C1-Package file, which provided good guide rails to how our code should be implemented.

3. Design Process and Testing

3.1 Design Process

The design process started with us reviewing the checkpoint one description and grading scheme, helping us to create our program requirements. After getting our requirements for each 3 parts of the assignment (scanner, parser, error handling), we focused on learning more about the technologies needed for this assignment such as CUP and JFlex. We found that the warm-up assignment had made us knowledgeable about JFlex already. We designed our CUP file based off of the provided tiny.cup file, with our error handling function at the top and our semantic rules at the bottom. For design patterns, we had learned in lecture how to leverage the Visitor pattern, which we set up in the source code to easily display our abstract syntax tree once it was created.

3.2 Testing

Our general testing strategy was to use the provided files (C1-Package) in order to test our source code throughout the whole development stage. All tests were performed on the Linux server at linux.socs.uoguelph.ca.

1. **Scanner:** We tested our scanner with the tiny.cup.bare file, which enabled us to see how our scanner was breaking down each token from the input file. We tested our scanner with the C-Minus source codes: fac.cm and gcd.cm. Once all tokens were showing up as expected from all files, we moved on to the next stage of testing.
2. **Parser:** We tested the parser by checking the created abstract syntax trees for the following programs: fac.cm, booltest.cm and gcd.cm. We checked each created abstract

syntax tree manually to ensure that our parser was working correctly before moving on to error handling.

3. **Error Handling:** Error Handling was tested manually by creating our own test files [12345].cm. For the first test file we re-used the booltest.cm file, since it did not have any semantic or lexical errors in it. For the 2.cm, 3.cm and 4.cm files we tested each error we implemented one at a time, and validated the reported error messages in our output to make sure the error handling was working correctly.

4. Lessons Learned and Limitations

4.1 Lessons Learned

1. The biggest lesson learned from implementation is getting comfortable with the CUP syntax in our cm.cup file. It took us the longest time total to write the parser for this checkpoint, as there was a big knowledge gap we had to overcome. For example, the precedence rules were especially challenging for our group to get correct. To overcome this we had to study the provided C1-Package files and review the CUP documentation provided.
2. A technical lesson we learned was how semantic error handling needs to be implemented. At first we thought that we just called the report error function and set the parser valid variable to false, however that would not lead to a graceful recovery. We needed to ensure that the syntax tree would still be generated properly in the case of an error, so more steps needed to be taken. For example, in cm.cup we handle variable declaration, so we have to create a new VarDeclExp to recover gracefully.

4.2 Limitations

1. A limitation in our current syntax error handling is that our error messages are general, and do not go into much detail other than location and the general type of error. For example, a type mismatch error in our 3.cm file (int x = true;) will be reported as a variable declaration error. The error is a variable declaration error, but it would be helpful to add that the problem is with the type assignment.
2. As per the assignment instructions, we did not implement every possible type of error in our error handling. This means that if someone were to write an error that there is not a rule for, our parser will not be able to recover gracefully.

5. Team Member Contributions

We split the work for the assignment, providing peer programming for each other to get all source code working.

Ahmad Sawan: Worked on all parts of the source code including the scanner, parser and error recovery. Also split the work on the documentation document.

Nathan Brommersma: Worked on all parts of the source code including the scanner, parser and error recovery. Also split the work on the documentation document.

Jacob McKenna: Worked on all parts of the source code including the scanner, parser and error recovery. Also split the work on the documentation document.

5. Conclusion

In conclusion, we were able to successfully combine the scanner, parser and error handling to implement part of the front-end analysis portion of a compiler. We have learned valuable lessons on core technologies to use in compiler programming, such as JFlex, CUP, and the visitor design pattern.