Compiler Implementation Project – Checkpoint 2 Report

Nathan Brommersma, Ahmad Sawan, Jacob McKenna

March 17, 2025

CIS*4650 – Dr. Fei Song

1. Introduction

This report outlines the documents on the implementation of our compiler implementation project, for checkpoint 2. Checkpoint 2 source code additions included adding a symbol table, semantic analyzer, and type checking. The report covers the following topics: implementation, design process, testing, lessons learned, limitations, and team contributions. At the end of this report the reader should have a good understanding of the design and challenges associated with the checkpoint.

2. Implementation Overview

For the implementation in checkpoint 2 we added a symbol table and a semantic analyzer to our compiler implementation project. We combined these two parts to create a working type checker that checks for a variety of potential typing errors in provided source code. Each variable and function are put into this symbol table, stored in stack and HashMap data structures. This allows us to keep track of the scope of each symbol, which was necessary for a language like C-Minus. From there we had to build the semantic analyzer, which we found to be the most time-consuming part of implementation. The semantic analyzer (found in SemanticAnalyzer.java) uses a visitor design pattern to perform a post order traversal of the tree. We implemented our type checking in this class, to catch a variety of variable re-declaration and type checking errors. Overall, our implementation went smoothly, and we were able to build all necessary parts of checkpoint 2.

3. Design Process and Testing

3.1 Design Process

Our design process started the same as the last checkpoint, we evaluated the checkpoint 2 description and marking scheme to determine our program requirements. From there we designed a symbol table to use a stack for each level, and each level of the stack has a HashMap. At the time we figured this was the best design for the symbol table, but looking back we see there could have been a better solution (4.1 Lessons Learned). After designing the symbol table, we moved on to the semantic analyzer. The semantic analyzer uses the visitor design pattern, which we use to make our post order traversal of the tree as simple as possible. This is in our "SemanticAnalyzer" class, to keep all the related methods together. Once the semantic analyzer was created, we were able to perform type checking with our class.

3.2 Testing

Our testing strategy involved using the simple C-Minus files provided such as "gcd.cm" and "fac.cm" to test controlled files with expected output. All testing was performed on the linux.socs.uoguelph.ca Linux server for a controlled environment.

1. **Symbol Table**: The symbol table was tested by manually comparing the output files [12345].cm with the printed symbol table. We were able to follow the program and look at how the symbol table should be output, and then we made the necessary changes if anything was incorrect. This method was especially useful when getting the symbol table levels to work properly.
2. **Semantic Analyzer**: The semantic analyzer was tested in conjunction with the type checker, since the post order traversal type checking was useful to determine if the

semantic analyzer was working correctly. We ran files with obvious type errors, and evaluated if the compilation would stop upon reaching these errors.

3. **Type Checking**: For our type checking, we tested various [12345].cm files that we created to showcase the different types of type checking we added. In those files, we utilized the provided example code snippets from the checkpoint 2 marking scheme document for a head start on what to be testing.

4. Lessons Learned and Limitations

4.1 Lessons Learned

1. A lesson we learned during checkpoint 2 was when building our symbol table. Our current symbol table implementation uses a stack that contains a HashMap in every level of the stack, and we use that to store the variables. In hindsight we think there is a more elegant solution, specifically if we were to have a single HashMap, and then store each symbol with a level value. This would allow us to save space, since we would no longer need multiple stacks and HashMap's for every level. Instead, our lookup would just check the level of each symbol. The lesson we learned here is a technical one and would improve our implementation if we ever require a similar data structure.
2. Another small lesson we learned was when we were building the semantic analyzer. For us this was a difficult part of the checkpoint, particularly the part where we needed to add function parameters to the correct scope. We were constantly adding to the wrong level of scope by one level, after a lot of looking, we realized we needed to do "level+1". We learned that it was already on the lecture slides along with other important hints. As we move forward to checkpoint 3, we have learned to make better use of the provided hints.

4.2 Limitations

1. One limitation is that our functions must be defined prior to calling, since our implementation does not support function prototyping. This means that if you were to call a function that is not written until further down in the code, it might not type check properly. It is worth noting that we have not tested recursive functions, but it could also affect the functionality of these. This could be addressed by adding function prototypes to the CUP file, which would then make this functionality possible.

5. Team Member Contributions

All three of our team members evenly distributed the work amongst all of us. With this method we were able to work effectively and efficiently to complete checkpoint 2.

Ahmad Sawan: Worked on the source code with the others using peer programming, including the symbol table, semantic analyzer, and type checking.

Nathan Brommersma: Worked on the source code with the others using peer programming, including the symbol table, semantic analyzer, and type checking.

Jacob McKenna: Worked on the source code with the others using peer programming, including the symbol table, semantic analyzer, and type checking.

6. Conclusion

In checkpoint 2 we were able to leverage what we have learned in the course to properly implement the symbol table, semantic analyzer, and type checking for our compiler implementation project. While doing this we have learned how to use post order traversal for type checking, and how to handle symbols in differing scopes. This has resulted in good progress in our compiler implementation project that we are proud of.