Compilers Checkpoint Three Report

Kristan Samaroo

Indeep Farma

CIS4650

April 3, 2022

1 - Summary

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

- AssemblyGenerator class for converting/ emitting syntactically and semantically correct C code into Assembly.
- -c command line argument for compiling C code.

Upon completion of this checkpoint, our compiler can generate assembly code. In addition to the functionality we implemented in the previous checkpoints including the generation of an abstract syntax tree and semantic analysis of the given program.

2 - Compiler Construction Process

2.1 - Understanding and Development Process Setup

This journey of creating a compiler began with us deciding to use GIT for version control/ incremental development. This decision hugely contributed to our success during the past checkpoints as it allowed us to work on feature branches without the risk of breaking our previously working code. In addition it made collaboration between multiple developers very seamless.

For most of the checkpoints we completed, the hardest part was always understanding and building a correct mental model of what had to be done. To begin each checkpoint, we took the time to read through the assignment documents together, outline components we would each like to work on and start our design from there. This allowed us to work together well and help each other if either one of us were struggling.

2.2 - Abstract Syntax Tree

Checkpoint One involved implementing regular expressions within our .flex file along with converting the grammar rules outlined in the “Specification for the C-Language” document into the grammar rules defined in our .cup file. The .flex file matches words within the .cm files in order to be fed into our parser. The .cup file/ parser creates the syntax tree we use for much of the further processing done throughout the other checkpoints.

After ensuring that our grammar rules worked as expected, we moved onto creating our ShowTreeVisitor class which creates a representation for the AST that can be logged to the console. The process of creating 'visit' methods within ShowTreeVisitor taught us a lot about the way that a compiler processes different types of expressions and declarations and helped us gain an understanding of what there was to do in the future checkpoints.

2.3 - Semantic Analyzer and Symbol Table

We started this checkpoint by referencing our ShowTreeVisitor class in order to know which expressions and declarations we would have to address while semantically analysing the syntax tree we constructed in the previous checkpoint. As we implemented our 'visit' methods, we also completed our SymbolTable class with methods as needed. This involved us figuring out when scopes had to be entered/ excited or destroyed, etc.

During the process of implementing methods within our SemanticAnalyzer class, we had to think of different types of semantic errors that could be made for each given type of expression or declaration. This was probably the hardest part of this assignment.

2.4 - Assembly Code Generation (Checkpoint Three)

2.4.1 - Basic Understanding and Design

We started this checkpoint much like the other ones by creating an outline of tasks we had to accomplish. There were a few major things we outlined including: creating emit methods, visit methods for identifying corresponding opcodes, -c command line flags, and additional error handling.

2.4.2 - Error Handling and Flag Checking

This part of the checkpoint involved making some changes to our syntactic and semantic analysis in order to improve our error handling. This was necessary in order to correctly generate assembly. Adding the -c flag was a fairly straightforward process, as we had done something similar in the previous checkpoint.

2.4.3 - Assembly Generation

The process of generating assembly code involved us identifying different opcodes and how they should be generated based on the different their associated C expressions. We started this process by generating assembly for declarations and then moved on to generating assembly code for expressions, functions, loops, etc. as outlined in the lecture slides by the professor.

4 - Compiler Retrospective

4.1 - Overall Design

To provide a high level overview of the design of our compiler, I can start from our main file. We first create our parser and lexer given a filename. This generates code based on our cm.cup file which I will address later on. We then create some booleans that represent flags provided via command line args. After this the parsing begins. Looking at our cm.cup file, we define a handful of grammars based on the “Specification for the C-Language" document provided. Within our “program” grammar, we conduct the semantic analysis (then logging the corresponding symbol table if the “-s” flag is provided). After this, under the conditions that the “-c” flag is provided and there are no syntactic or semantic errors, then we can start the Assembly generation process. The ShowTreeVisitor, SemanticAnalyzer, and AssemblyGenerator classes follow similar steps in which they have multiple overridden methods for each given Variable or Declaration object. Within each of these methods, they either output part of the tree (ShowTreeVisitor), check for errors (Semantic Analyzer) or generate assembly code (Assembly Generator).

After this process has completed, our compiler will then output a success message or a failed method to the console along with the various reasons why.

4.2 - Lessons Learned

This project allowed us to continue learning a significant amount about the inner workings of a compiler. In the first warm up assignment we learned about how a scanner works, and in Checkpoint One we dove deep into how abstract syntax trees, parsers and scanners can be integrated together in order to verify program structure. Checkpoint Two taught us how semantic analysis allows compilers to catch semantic errors. We learned about the process of how a compiler goes upon determining if variable types are mismatched, methods are called incorrectly, symbol tables are used to handle scopes, and much more. Checkpoint Three taught us how assembly code gets generated after a compiler has verified that there are no semantic or syntactical errors.

Another important lesson we learned was the importance of using version control while working in groups. Without the use of GIT for version control, we would not have been able to keep our development process as organized and efficient.

4.3 - Assumptions and Limitations

Some assumptions and limitations from our implementation of our project include:

- If a function has an array as a parameter, we assume that they can only be of type int.
- Variables declared as void will raise a semantic error. This was the easiest way of dealing with them as it would involve significant effort in order to consider void variables.
- We assumed that only multi-line comments are within the file. Currently, single line comments produce an error.

- Upon generating assembly, the file to be compiled gets overwritten after compilation. Which is not the expected behaviour.

4.4 - Possible Improvements

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

- There are some other errors that could be picked up in our semantic analysis that our compiler may have missed.
- We were unsure what semantic analysis could be done (if any) for our integer expressions, so we decided to skip semantic analysis on integer expressions.
- There are some visit methods that we could improve upon. Such as our AssignExp method that has slightly vague comments.

8 - Team Member Contributions

Kristan Samaroo

- Implementing flag checking and error handling
- Modifying Makefile/ adding targets for ease of execution
- Writing Documentation

Indeep Farma

- Implementing methods/ helper functions for AssemblyGenerator.java
- Creating Test Files, and testing the files
- Writing Documentation

9 - Acknowledgements

The code for this project was based off of Dr. Fei Song's starter code provided to us for the assignment and built upon our existing code from the previous two checkpoints. 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"