Compilers Checkpoint Two Report

Kristan Samaroo

Indeep Farma

CIS4650

March 23, 2022

1 - Summary

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

- Symbol Table
- Type Checking (Major task of semantic analysis)

After finishing up Checkpoint One, we were able to generate an abstract syntax tree with valid input, otherwise detect and report syntactic errors. The next steps to our compiler were to now detect and report semantic errors by traversing the abstract syntax tree we generate in a post-order. Examples of semantic errors are things such as mismatched type in expressions and undeclared/refined identifiers. We used the lectures and lecture slides as our guides for completing Checkpoint 2.

During Checkpoint 2 the compiler can now differentiate between scopes within C programs, i.e. global scope vs local scope, and is able to create a symbol table for the program. It can also report semantic errors as they arise. The symbol table is also able to recognize symbols within different scopes and tell the compiler about it. The compiler can also perform type checking of expressions in C programs, where it can ensure that assignments of variables are valid, check function return types, arrays are in proper range and other operations are valid.

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, similar to how we went about for Checkpoint One.

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 code from Checkpoint One (since we will be building on top of it). Once we finished watching all the lectures and reading the lecture slides, we were able to finally start the assignment with some idea of how to work towards our end goal.

Task Two - Symbol Table

To start off the assignment, we had to decide on how to implement the symbol table. After watching the lectures by Professor Song we decided that we would follow his recommendation for the Symbol Table. To allow for easily creating new scopes, adding new symbols, deleting symbols, deleting scopes, retrieving symbols and scopes we would create an ArrayList of Hashmaps. Each Hashmap would define a scope, with the contents of the Hashmap being symbols in the scopes. By having an ArrayList of Hashmaps we are able to easily keep track of scopes and where we are within the program. Creating a layer of abstraction with the Symbol Table was needed for the next step of the assignment, which was implementing type checking.

Task Three - Type Checking

The process of implementing type checking/ the semantic analysis involved a couple different steps. We implemented methods for the different declaration or expression classes that required it. Within these methods we checked for semantic validity along with semantic errors if they came up. Some examples of this include: checking if a function call is making reference to a function that exists, trying to assign an array as an integer, and much more. This component reported errors to System.err when appropriate and called methods from the Symbol Table when actions like entering a scope or existing a scope were required.

3 - Lessons Learned

This checkpoint 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. In this assignment, we learned about how scopes and type checking go hand in hand to help each other, as well as how scopes are kept track of within the compiler.

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

Some assumptions and limitations from our implementation of this checkpoint 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.

5 - Possible Improvements

Some ways in which our code 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.
- Error messages could include more information about the error. Currently we only report row, along with the type of error.
- 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.

6 - Team Member Contributions

Kristan Samaroo

- Adding additional functionality within the .CUP file
- Implementing methods within SemanticAnalyzer.java
- Implementing functionality for -s and -a flags within main.
- Modifying Makefile/ adding targets for ease of execution
- Writing Documentation

Indeep Farma

- Implementing methods within SymbolTable.java
- Creating Test Files, and testing the 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 and built upon our existing code from CheckPoint One. 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"