March 20, 2023

CIS*4650 - Compilers Checkpoint 2 Documentation

Zeynep Erdogru - Dogu Gerger
Student IDs #1047085 - #1068684

Summary

- For the Compiler's Implementation project, Checkpoint Two involved two main tasks: Implementing the Symbol Table and performing Type Checking, which is a major task of Semantic Analysis. After completing Checkpoint One, our team was able to generate an abstract syntax tree that would detect and report any syntactic errors for valid inputs. However, for this checkpoint, we needed to traverse the abstract syntax tree in a post-order to detect and report any semantic errors such as mismatched types in expressions or undeclared/refined identifiers.
- To complete this checkpoint, we used the lectures and lecture slides as our guides. With the implementation of the Symbol Table, our compiler can differentiate between scopes within C programs, such as global and local scopes. The symbol table can also recognize symbols within different scopes and report them to the compiler. In addition, our compiler can now perform Type Checking of expressions in C programs. This means that it can ensure that variable assignments are valid, check function return types, verify that arrays are in the proper range, and check the validity of other operations. By performing these tasks, our compiler can detect and report any semantic errors that arise during the compilation process.

Incremental Design Process

- **Step 1 - Understand the Basics:** Our objective was to gain a basic understanding of the requirements and design for the project. This involved carefully reviewing the provided documentation and our code from Checkpoint 1, as we would be building upon it for this task. We also thoroughly studied all the relevant lecture slides and the textbook to ensure we had a clear understanding of the project's objectives and specifications.

After completing these preparatory steps, we were finally able to start the assignment with a solid foundation and a clear direction towards our goal.

- **Step 2 - The Symbol Table:** At the beginning of the assignment, one of the crucial decisions we had to make was regarding the implementation of the Symbol Table. After reviewing the lecture slides on Courselink, we agreed to adopt his recommended approach for the Symbol Table.
- We decided to use a Hashmap of ArrayList to facilitate the creation of new scopes, addition and deletion of symbols/nodes and scopes, and the retrieval of symbols and scopes. We created a new class called “NodeType” and using the Nodetype object we were able to define the scope (level), name and the type of declaration.
  Implementing this abstraction layer with the Symbol Table was important for the next step of the assignment, which involved implementing the type checking. By using this approach, we were able to make sure our implementation was modular and maintainable.
- **Step 3 - Type Checking:** The process of implementing type checking and semantic analysis involved multiple steps. We created methods for various declaration and expression classes that required type checking. Within these methods, we checked for semantic validity and any semantic errors that might arise. Some examples of type checking included verifying the existence of a referenced function in a function call, ensuring that an array was not assigned as an integer, and checking for other similar cases. Whenever appropriate, this component reported errors to System.err and called methods from the Symbol Table to enter or exit a scope. By following this approach, we were able to ensure that our type checking and semantic analysis was robust and able to detect any issues or errors in the program.

Possible Improvements

- The error messages can provide additional details about the error, beyond just the type of error and row number that is currently being reported.
- Our compiler may miss some errors that could be identified through semantic analysis.

Learnings

- Checkpoint Two enabled us to gain further knowledge about the intricacies of a compiler. In our first warm-up assignment, we familiarized ourselves with the workings of a scanner, while Checkpoint One delved into the integration of abstract syntax trees, parsers, and scanners to verify program structure. In this assignment, we expanded our knowledge by learning about the interdependence between scopes and type checking and how they complement each other.
- We were able to gain hands-on experience about the things we learned in class, such as how type checking uses type information to ensure that all constructs are valid under the type rules, static type checking performed at compilation time, and dynamic type checking performed at execution time. We also gained an understanding of how scopes are tracked within the compiler. This checkpoint provided us with valuable insights into the implementation of a compiler and enhanced our understanding of its various components.
- We also gained an essential insight into the significance of using version control when collaborating in groups. Throughout our work, there were occasions where we had to abandon branches due to misconceptions regarding a particular aspect of the assignment.

The use of GIT for version control played a vital role in keeping our development process coordinated and optimized.

Assumptions and Limitation

- In our implementation of this checkpoint, there are certain assumptions and limitations that need to be considered. One of these is that when a function has an array as a parameter, we have assumed that the array can only be of type int.
- If a variable is declared as void, it will result in a semantic error. We chose to handle void variables in this way as it would have required significant effort to handle them otherwise.

Team Member Contributions

Zeynep Erdogru

- Writing Documentation
- Implementing methods within SymbolTable.java
- Creating test files and performing testing
- Modifying Readme

Dogu Gerger

- Writing Documentation
- Implementing methods within SemanticAnalyzer.java
- Implementing functionality for -s and -a flags within main
- Adding additional functionality within the .CUP file

- Modifying makefile

Acknowledgements

- The implementation of this project was based on the starter code provided by Dr. Fei Song, which we built upon using our existing code from Checkpoint One. Following Dr. Fei Song's guidance from the lecture slides and the documentation provided in the starter code, we adhered to his recommendations for the implementation process. Our .CUP file contains grammars that are directly derived from the "C-Specification".