Checkpoint 2: Semantic Analysis
Project Report
CIS 4650* Compilers
March 23rd, 2022

Umer Ahmed
Numan Mir

Introduction

The objective of CIS*4650's course project is to build a compiler for the C- (C minus) language. Creating this compiler requires building several connected items: a scanner for lexical analysis, a parser for syntactic analysis, a semantic analyzer, and an assembly code translator/code generator. These items are being built gradually over a series of checkpoints. As of the date on this report, all items up to and including the semantic analyzer have been implemented.

For the first checkpoint of the course project, we built a scanner and parser to handle lexical and syntactic analysis for the C- language. This consisted of 1) a scanner that scans all tokens of a C- file and reports lexical errors, 2) a CUP program that defines C- grammar rules, enforces syntactic error recovery, and produces abstract syntax trees, and 3) a folder of classes defining tree nodes and a tree visitor program to display the aforementioned abstract syntax tree. In the second checkpoint, we implemented the semantic analyzer, which consisted of 1) building and displaying a symbol table that stores all relevant information about variables in a hash table, and 2) using the symbol table to type-check variables in all scopes, across a thorough set of testing conditions.

The rest of this report will describe the process of incrementally building the semantic analyzer component of our compiler project. Below, we have highlighted our design strategy, the course knowledge we reinforced, our testing methodology, assumptions, limitations, and potential improvements.

Design Process

We completed the requirements of the second checkpoint incrementally, starting off by building the symbol table. As instructed, we created a tree visitor pattern for semantic analysis

(SemanticAnalyzer.java), that we used to traverse the abstract syntax tree (produced by checkpoint one's parser) in post-order. With this method, we iterated through each scope of a C- program (.cm file extension) and captured all variable declarations, storing all relevant information in a HashMap representing the symbol table, and deleting a variable's data as its respective scope is exited. Simultaneously, we displayed the symbol table using indentation to represent scopes; in this way, we ensured that variables were being stored and deleted properly.

Once our symbol table was complete, we began to incorporate error-checking for redefined and undefined variables. At this stage, we also included error-recovery for variables declared with the type "void": we generated an appropriate error and processed their types as "int" so that the rest of the program works as expected.

Next, we executed type-checking. Firstly, we implemented and declared the "dtype" property of all expressions, to easily refer to and compare data types. We then prioritized the following major areas of a C- program: array indices, assignment-expressions, operator-expressions, function-calls and return-expressions, and test conditions for if- and while-expressions. To aid us in the development of this process, we created C- programs which consisted of simple scenarios in which a type-checking error should occur. Using our symbol table and the "dtype" property of expressions, we compared variables and functions to their declarations and generated corresponding semantic errors.

The final step of our design process was testing, developing edge cases, and refining our code accordingly. In our submission, we included C- programs: '2', '3', '4', '5' (all ending with .cm) which provide and describe various types of parsing errors to showcase the thoroughness of our error recovery implementation.

Course Knowledge Exercised

By completing the second checkpoint of our course project, we deepened our understanding of the components involved in the "front-end" of a compiler. With our implementation, we examined how semantic analysis is connected to the parsing process, and how semantic rules are derived from grammar rules. Additionally, with our semantic tree visitor program (SemanticAnalyzer.java), we learned how to navigate nodes of a tree in post-order. Working with abstract syntax trees enabled us to carry out a multi-pass approach for our semantic analysis, in which semantic rules are simplified. We were better able to understand why a single-pass analysis/syntax-directed translation would not be an ideal solution, even for a simplistic programming language like C-, as many details would need to be tracked which would result in overly-complicated code that is difficult to maintain and debug. Furthermore, through the type-checking portion of the assignment, we exercised our knowledge of the significance of symbol trees and how they are used by the semantic analyzer to ensure error-free code.

Assumptions, Limitations and Improvements

Assumptions:

- Relative operations with int types on both sides will be evaluated as an int type (e.g., "2 < 3" is an int). For instance, the test condition for "if (2 < 3)" is valid because test conditions need to be of type int and operator-expressions need to result in an int.
- Since we are using a HashMap for our symbol table, when we display variables at the end of a scope, they may sometimes be in an order different from the one they were declared in. We are assuming this behaviour is expected.
- For any block of code (code surrounded by curly braces) that is not a function, the symbol table output will be defaulted to "Entering a new block" (i.e., for if-expression blocks, while-expression blocks, and non-specified blocks).

- Due to the post-order design of our semantic analysis, we are assuming that our semantic errors should be printed after our symbol tree output.

Limitations and potential improvements:

- Our symbol table design prioritized space over speed. We used a HashMap containing pairs of a String key and an ArrayList of NodeTypes. We could have instead chosen to store our symbol table as a structure containing multiply-linked HashMaps. This would result in easier and faster lookups and deletions.
- Our compiler so far does not incorporate null-checking. For example, an integer may be declared, "int x;", and then immediately passed as an argument to a function expecting an integer, "funcName(x);" without errors. We may incorporate null-checking in a future iteration of our compiler.

Member Contributions

For most of our second checkpoint, we collaborated via Microsoft's Live Share extension for VS Code, so that both members could see each other's changes in real-time. The symbol table implementation was completed entirely over these live sessions. In the interest of time, we decided to split up the type-checking portion of the assignment as evenly as possible, for each of us to work on individually. Below you will find each member's responsibilities for type-checking and error-testing:

- Umer: type-checking and related error-testing of array indices, assignment-expressions, and operator-expressions
- Numan: type-checking and related error-testing of function-calls, return-expressions, if-expression test conditions, and while-expression test conditions