Project 2: Semantics Analyzer

Stephen (Blake) Sanders, Jared Wermager, Austin Johns

Professor Nasser Tadayon

CST-405

*Abstract Syntax Tree*

Our approach to creating the abstract syntax tree (AST) was to traverse the parse tree using a depth first search which would create the abstract syntax tree. By traversing the syntax tree, with different logic from the classes, the input is processed in a different way than the parser would. This new way of processing the input to the parser creates the information that will be placed into the syntax tree by removing some of the nodes of the parse tree. This tree exemplifies the different scopes of the code input. A scope in this context is defined as a name binding, usually from declaring a function. A scope can contain different variables and declarations. A global scope contains variables that can be claimed at any scope of the code structure. However, if a new scope is created (ex. int main()), any variables included in this scope cannot be used outside of the scope. Additionally, if another scope is created inside of int main(), the variables declared inside this scope can only be used in the scope, along with the variables defined in int main(). This process can be repeated with as many scopes as the code developer would like to create. These different scopes can be modeled in a tree.

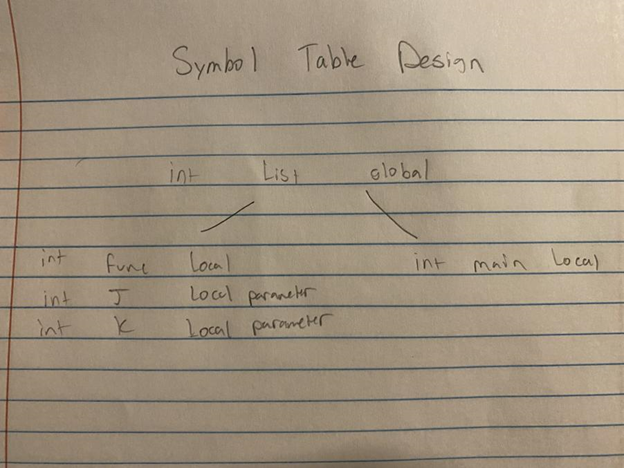
*Processing the Parse Tree*

In project 1, the parse tree was completed to display the processing of the tokens from the lexical analyzer. In this project, all information from the semantic analyzer is processed and displayed in a tree. This is to communicate to the user a simplified tree to help exemplify the scope of the code. The program creates a tree to show how the scope of each variable exists, and how the scopes affect the processing of information by the program.

Screenshot of the tree:

*Symbol Tables*

Initial design:



Final Design:

*Semantic Checks*

The purpose of the semantic check is to ensure that there are no errors in the syntax of the code. The rules come from the gcu-pl file, and the semantic check is designed to detect any errors in the code. The way that this check is done is

*Lexical/Syntax Error Reporting*

If a semantic error is detected by the semantic check, the user is alerted by an error output. The error is detected by the semantic check, and the program communicates this error with the output to the console, “*Error: There was a syntax error*.” The user has been notified and has the opportunity to detect the error and resolve it.

*Main Purpose*

The purpose of this application is to process the rules of the grammar even further. In order to successfully process input into a compiler, the semantics analyzer must first understand the scopes of the code inputted. For the successful processing of input to the compiler, the semantics processor analyzes the rules of the compiler to ensure that the syntax is correct.

*Implementation Approach*

Class created for symbol tables, logic for creating them are in the class. This is how tables are created and separated by scope.

*Screenshots*

Initial work:



