Justin Canton, 1000017910
Samprit Raihan, 998138830

Compilers and Interpreters CSC467 Lab 3
AST Construction and Semantic Checking Documentation

The implementation of features involved in this lab were broken down into a few tasks:

1. AST construction
2. AST tear-down
3. AST printing
4. Semantic checking

Approach to AST structure:

Based on the context free grammar given to us in the parser.y we decided to recursively allocate nodes of our abstract syntax tree (AST) as we broke down the incoming tokens into the respective terminal symbols, and built the tree bottom up. i.e the children nodes were allocated first, and then attached to their parent, and so on. This was done using the ast_allocate() function. We used the enumerations provided in ast.h to signify each type of node, and had case statements to specifically allocate that kind of node.

The AST tear-down process was also done in a bottom up fashion using the ast_free() function, which freed up the memory associated with each child, and then moved up to the parent and so on. Again, enumerations were used to access each type of node and free it's data members.

The ast printing involved a top down approach, but with the added constraint of sometimes looking ahead to children nodes to infer the datatype of the parent node, and print it accurately, and in the case of an invalid statement with type mismatch, print out an error message saying that said was invalid. This approach also used enumerations to access the type of node, and based on that, take specific actions, which sometimes involved burrowing down to the leaf nodes, and sometimes looking at the symbol table to infer the type of the parent.

The approach to creating a symbol table involved creating a struct that contained all the vital information related to a symbol node. Then, for each scope in our program, we generated a specific symbol table, represented by a linked list of symbols. The symbol tables pertaining to each scope was in turn connected with each other at the head of linked list.

Based on all of the above, the semantic checking boiled down to a brute force approach, done by the function semantic_check(). This function looked at the AST, and for each specific kind of node and it's children, to check that the datatype propagated bottom up were always consistent. In the case of a type mismatch, it reported semantic error.

The challenges that we faced included understanding the multi step recursion that happens while traversing the AST using the different methods of traversal, and trying to look ahead depth first into the children for information that has significant implications on determining validity of the program, and doing this all the while avoiding segmentation faults, and putting the appropriate null checks and error checks in the right place.

The novelty of the approaches taken lies in the fact that a large portion of it used recursion, which provides a better intuition of how to systematically traverse ASTs as opposed to iterative alternatives.