CIS*4650 ~ Checkpoint 2 Project Report

Isabella McIvor (1101334) , Huda Nadeem (11439431), Zuya Abro (1109843)

What has been done for this checkpoint:

For checkpoint 2, the main additions were implementing symbol table and type/error checking functionalities. Adding on from checkpoint 1, the program now detects and reports semantic errors, this includes type mismatches in expressions and handling undeclared/redefined identifiers. Once an error is detected, the program continues and will display the error in the terminal. We created 5 different test files: 1.cm displayed correctly, 2.cm checks when a int variable is assigned non integer to catch this error, 3.cm checked is the array index is processed accordingly to its type, 4.cm checks if there is no main function declared, and 5.cm tests multiple type errors. The -a flag was implemented to display the abstract syntax tree in a separate file filename .abs. The -s flag was implemented to create a new file that displays an abstract tree similar to -a, but also creates another file that displays the symbol table .sym. [make clean] command can clean up all files that were generated, .abs and .sym files.

Design process:

1. **Preparation**

The first step for this checkpoint was to create a SemanticAnalyzer.java file along with a NodeType.java. These were starter files to start implementing node structure and adding a private HashMap<String, ArrayList<NodeType>> table;. Then we updated the absyn folder accordingly to accommodate the structure, adding public Dec dtype; to Exp.java . After this we viewed the CM.java to understand how the output would display and how we would cater to the separate outputs.

2. **Symbol Table**

When creating the semantic analyzer class, the first step was to create visit methods for every absyn class. Next, we initialized the hash table and added helper methods to lookup, insert and delete in the hash table. Within the visit methods, we first focused on building the symbol table before considering any type checking. In every visit method for a Dec class, we added that declaration to the hashtable along with the integer value representing its scope. Then, to produce the proper output for the symbol table, we added code to print when the program was entering or leaving a scope. This code was found in the FunctionDec, IfExp and WhileExp class visit methods where new scopes would be created. Before exiting each scope, we used a leaveScope() helper method to print the function and variable declarations within that scope and delete them from the table after leaving the scope.

3. **Type checking**

We approached the type checking by going through each visit method for the classes inherited from the Exp class and looking at which possible type errors could occur from visiting this node. We started with the visit methods that would represent the “leaves” of the tree because they did not have children in which their types depended on. These included SimpleVar, ArrayVar, IntExp, and BoolExp. We used a lastVisited variable to keep track of the type of the last visited node and an isArr variable to keep track of whether each last visited node was equivalent to a simple or array expression. In these leaf node visit methods, we assigned the lastVisited variable to the type of the node to be passed to their parent nodes. This allowed parent nodes to be able to check the type of a whole expression rather than simply a variable. We also checked the variable expressions that had already been declared by checking if they existed in the symbol table. The order of the rest of the type checking code was to implement type checking

for a visit method of a node and then move on to their parent node next. Each visit method had specific things to check for. For the If and While expressions we checked that the conditional expressions were of boolean or integer type. For the OpExps, we checked that both of the operands were the same and that they were of boolean type or integer type depending on the type of operation.

Summarizing any lessons gained in the implementation process

- ➢ We learned about using version control software in a group project and how to resolve merge conflicts.
- ➢ We learned about semantic analysis, and how to recover after detecting semantic errors
- ➢ We learned about how to display a scope structure accordingly

Discussing any assumptions:

- ➢ All global variables and function declarations are declared right before leaving the global scope
- ➢ The -s flag means the tree should still be saved to .abs in addition to the symbol table saved to .sym
- ➢ No flag means that both tree and symbol table should be printed
- ➢ “int input(void)” and “void output(int)” are predefined functions that do not show in symbol table
- ➢ “if” condition must be of boolean type and “while” condition must be of boolean or integer type
- ➢ When -a is used, semantic analysis is not used

Limitations:

- ➢ While we have designed 5 specific test cases, this may not cover all possible errors. In the future, we can create more tests to thoroughly check edge cases.
- ➢ Additionally, we did not verify if the callExp parameters matched the corresponding functionExp.

Possible improvements:

- ➢ Instead of using a number of different variables to keep track of different statuses of each node visited (such as type or whether it is an array), we could have found a cleaner way to pass this information to visit methods.

Contribution/Assessment:

Ella:

- ➢ Added -s flag to CM.java
- ➢ Contributed to SemanticAnalyzer.java
- ➢ Contributed to documentation
- ➢ Performed testing

Huda:

- ➢ Added -a flag to CM.java
- ➢ Contributed to SemanticAnalyzer.java & documentation
- ➢ Worked on displaying output properly according to scope
- ➢ Collaborated on test files (12345.cm)

Zuya

- ➢ Collaborated on documentation
- ➢ Added error handling for missing main function in SemanticAnalyzer.java
- ➢ Collaborated on test files (12345.cm)