# Checkpoint 2 Report

By: Jessica, Justin, and Arvind

## What Has Been Done For This Checkpoint

**Jessica** worked on building the **symbol table**. The symbol table uses a stack of hashmaps. Each hashmap correlates to a scope that holds a set of Variable Name string keys where each one leads to an ArrayList of NodeType variables. Each node in the ArrayList is defined in NodeType.java where each one holds the level, variable name, data type (using simple variables such as int, bool, and void), and scope that it's in. With this stack functionality, the function blocks have their entry (pushing the hashmap scope onto the stack) and exit (popping the hashmap scope off of the stack) dealt with. Errors of undefined or redefined variables in insert(), lookup(), and delete() functions are detected and reported to the terminal for examination.

**Justin** worked on building the **type checker** functions**,** updating Checkpoint 1's error recovery, the semantic analyzer's visit functions, helper type checking functions within the semantic analyzer, redefining of appropriate absyn files, debugging of the semantic analyzer, printing of the symbol table, handling of the symbol table and the handling of the linked hash tables.

**Arvind** worked on the **command line flags "-a" and "-s"** and debugging code. Command line flags determine the type of output that will be printed to a file. The main method (**Main.java**) checks for the **-a** flag to enable printing of the **Abstract Syntax Tree (AST)** and the **-s** flag for outputting the symbol table. When the **-a** flag is used, a human-readable **AST** is generated by invoking a visitor (**ShowTreeVisitor**) on the parsed syntax tree, then it writes the tree to a file in the **tests** folder with the **.abs** extension. Similarly, if the **-s** flag is used, the **SemanticAnalyzer** processes the **AST** to create the symbol table, which is then output to the **tests** folder with the **.sym** extension. Furthermore, debugging was performed to address issues in the code. Several of the errors that were encountered were due to type mismatches, which complicated the process of retrieving the types of variables. This made it difficult for the semantic analysis to correctly identify valid code constructs, causing errors to be displayed when there were not any.

**Related Techniques and Design Process**

We used a stack to hold each scope's respective symbol table/hashmaps the stack gets a new symbol table/hash table pushed to it each time we enter a new scope (this occurs within the visit function for compoundExp as we always enter a new scope when a compound expression is encountered) and pop the stack of tables everytime we reach the end of the compoundExp visit function. We approached the implementation of the symbol table this way because it seemed to be the easiest way to keep track of declarations when dealing with multiple scopes. Originally we were tracking the scope of declarations using levels and strings (held the name of the scope) but found that when dealing with simple compound expression like if/while statements this method fails (if as a scope name fails us because we could have multiple ifs and this will cause unnecessary errors), after consulting the lecture slides yet again it was decided that we would use the linked symbol table method. This method made it very easy to perform the declaration checking process as we only keep tables that matter on the stack (with the popping of symbol tables for lower level scopes we do not acknowledge lower level declarations and thus only the important declarations remain) and with this all we simply need to do to check declarations is loop through each table and search for the desired declaration. The tables stored within the stack use a string as a key that represents the name associated with the declaration. The keys are connected to variables of the type ArrayList<NodeType>, originally it was just a NodeType but we found this made it hard to deal with function prototypes and function declarations so it was reverted back to ArrayList<NodeType>. We built three functions for the system to interact with the symbol tables. The first is the insert() function that inserts a new node into the key’s node list and reports errors of redefined variables. The second is the lookup() function which looks for a node list by a specific key and returns the respective node list or reports errors if looking up undefined variables. The third is the delete() function which deletes a key and its node list and reports errors of undefined variables. The symbol table was designed update/output during the traversal of the AST we found this to be the best way as our linked symbol tables would also be popped during the traversal so a post traversal would be impossible, the output was done using the indent function along with simple System.out.println calls. The symbol table will only output declarations. With the visit functions, we defined them in a way where they can simply depend on each other, similar to the showTree version we have them connected in the same way but now we have the appropriate checks and table operations within the visit functions that need it, for example, FunctionDec’s visit we handle the things needed for functionDec and instead of doing all the checks for the body we simply call the body.accept function to let it handle it. Along with the visit functions we also implemented some helper functions to aid in the type checking, these functions include typeChecker(), wasDefined(), funcWasDefined() and most importantly getExpType(). getExpType was crucial to the successful running of our checkpoint 2 functionality, initially, we used the function defined for the exp class getType(), however we found that a lot of the time when we were dealing with a subclass of Exp (like varExp ) it would default the getType() to the original definition and give us the wrong type, to fix this we eventually found that it was important to determine what instance of Exp we were dealing with, type cast the exp variable and then utilize the proper type yielding techniques to return a value that represents the type.

**Error checking**

Most of the error checking was solely based on type checking. Generally everytime we encounter a variable we check it against the current important declarations and throw an error if the types do not match or if the variable was never defined. Another kind of error we would watch out for are re-declarations of existing variables within the same scope.

**Lessons Gained In the Implementation Process**

With the **symbol table**, we learned that there were a lot of ways that the table could have been implemented when using a stack or a class instead of just a hashmap. However, in the end, we chose to stick to using a hashmap as it follows the 2nd checkpoint's implementation tips and rubric. With the **type checker** we learned that it's not supposed to be a single simple function but a collection of functions that can work in tandem to help in the type checking process, like how instances of Exp, Var and Dec will have different ways to type check.

**Assumptions, Limitations, and Improvements**

For assumptions, it is assumed that parsing and the generation of the **Abstract Syntax Tree** are completed before semantic analysis, making sure that Checkpoint One outputs a valid tree that Checkpoint Two can process. Furthermore, the code will be tested on the **SOCs Linux servers.**

For limitations, we found that we just could not get the symbol table to print in the specific order of lower scope declarations first and then outer scope declarations.

We also had trouble creating tests that involved using array indices (**g[2] = 5**). We will have to fix this issue before we start Checkpoint Three.

For improvements, we would like to implement more test files and error recovery code to ensure that our code is rigid enough to identify and recover from more complex errors.