CIS*4650 W24 Checkpoint 2 Project Report

Salman Azhar (1058981)

Overview:

In this report, I will be going over the process of how I finished this checkpoint, going through the preparation, then the symbol table formulation, and finally the type checking. Following this, I will provide some reflections on my process.

Implementation Process:

Related techniques: Inheritance, visitor pattern, hashtables and function overloading

Preparation:

In order to get started on this project, I went through the following steps:

1. Read the checkpoint 2 description posted on Courselink and went over the Lecture 8 lecture slides for the hints on how to get started.
2. Ensured that my code from checkpoint 1 ran smoothly.
3. Looked up what hastables are and how they are implemented in Java
    a. Learned about collisions, and the ways to address them.
    b. Took a look at the Hashmap<String, ArrayList <NodeType>> and NodeType class provided in lecture 8 slides and how they would be useful for creating a Symbol table.

Symbol Table.

This is the process I followed to implement the symbol table:

1. Looked through the lecture 8 “Type Checking” slides to see the method we are expected to follow to get started.
2. Created a NodeType.java class to contain relevant information about each symbol.
3. Created SymbolTable.java to handle all operations directly related to the SymbolTable data structure itself.
    a. Initializes and creates the SymbolTable with a Hashmap<String, ArrayList <NodeType>>.
    b. Enter scope: symbol table increments its scoping level and prints the line notifying that a new scope has been entered, and prints the scope name.
    c. Lookup functions: Checks if a variable has been declared before, and if it was in the same scope as the current one
    d. Inserts a declaration into the symbol table by first looking up for insert and gives an error if it’s a redeclaration.
    e. Exit scope: Prints the declarations of the scope when exiting and prints the name of the scope that has been exited.
4. Created SemanticAnalyzer.java to handle the visiting of each node. Here I overloaded the void analyze() function to basically handle every kind of object in Absyn. It would enter and exit scope whenever it came across a compound expression, and added the function, simple and array declarations to the symbol table whenever it encountered one, and print an error if it was a redeclaration.

Type-Checking:

The Type-checking turned out to be the most difficult portion of this checkpoint to get a mental grasp on. However, upon understanding how to approach it, it became significantly easier. In particular, the role of “dtype” in the type checking process. Here is the process I followed to implement type checking:

1. Added Dec dtype; to the absyn file, ensuring its inheritance by the children
2. In SemanticAnalyzer.java, I tried to trace how and when we would need to know the type of a certain node. E.g OpExp would have to have a type of its operads, since we need to check their types.
3. Implemented the functions getType() isBool() and isInteger(), which all take dec dtype as an argument to make the code for type checking a little cleaner.
4. Had to take extra steps to ensure that the assignment would type check correctly. This was needed to ensure that array declarations with simple vars can only be assigned to other array declarations with simpleVars.
5. CallExp also had to have a similar edge case when calling a function with an array declaration as a parameter required to pass a simpleVar which referred to an array declaration.

Reflection:

Assumptions:

My primary assumptions for this checkpoint were all that was specifically listed in the lecture 8 slides, and information from Professor Song in lectures and the discussion board. Also, this program assumes that the AST for the -a flag will output the code to a filename with the same name but a .ast extension, and the -s flag will output a symbol table to a file with the same name as the input file but with the .sym extension. Although the -s does generate an ast, it does not print it to any file.

Limitations:

For the limitations, I think my code handles all the expected functionality. Although this was not an expected functionality, my code does not report an error if the user reached the end of a function without returning something. In most programming languages, this would be an error, however since it was not required for this checkpoint, I left it out. Also, Professor Song mentioned in the discussion board that we do not need to evaluate the main function first, so I left that out as well.

Possible Improvements:

An area where I feel I could have improved my code with regards to better coding practices was to use the visitor class already laid out in the Absyn package. However, I simply created a new semantic analyzer class and overloaded one function on it. Using the visitor pattern, I would not have to redirect the nodes from their parent classes such as redirecting every possible exp from the exp handler. This wasn't too much added work but my code was not as clean as it could have been.