CIS 4650 Checkpoint Two Documentation

Implementation

For our implementation, we first have a class “SymbolEntry” which makes up an entry of the symbol table. Our types are encoded as an int so we have a helper function “getTypeAsString” to get the string value of our type. And we have a custom toString method to print out the relevant info for each entry. We then have the class “SymbolTable” in which SymbolEntry’s are stored using a stack of hash maps. We keep track of the currentScope with methods for “enterScope” and “exitScope”. The exitScope method also removes symbol table entries accordingly on exit. Most importantly we have a method “insert” to put an entry in the table. We also have a “lookup” method to find a specific entry and a printTable method to print the entire table. We then have a class “SemanticAnalyzer”, which uses the visitor design pattern to populate the symbol table as it parses through the syntax tree. We have an “analyze” method to start the process of the visitor. While populating the syntax tree, we check for various semantic issues along the way and produce errors accordingly. We also have an “AnalyzerPrinter” class which is responsible for spitting our AST and symbol table to either the terminal or a file. The printer is passed into our other classes, and its methods are also used in the visit functions along the way to print out the relevant info. We create a separate printer for our AST and for our Semantic Analysis. Lastly, in main we have two helper functions, one for checking for command line arguments and another for producing the output files with the correct extension.

Techniques and Design Process

We have tiered data structures: SymbolEntry->SymbolTable->SemanticAnalyzer. SemanticAnalyzer is made of symbol table(s) (in our case only one, but for the future many), and a SymbolTable is made of SymbolEntry's. Our AnalyzerPrinter class abstracts away our IO and allows us to seamlessly flush our output to the terminal or a file.

Our analyzer traverses the tree in post order. Before doing the type checking at a node, it will travel to the end of the tree first to get the type data before attempting the typecheck. SymbolEntry keeps track of the scope, dimension, type, and parameters of a declaration. Our symbol table populates as our analyzer traverses the tree and also keeps track of the current scope. When the analyzer leaves a scope, all the symbols in that scope will also be deleted.

Lessons learned

- Need to have a symbol table per scope
    - Helps tracking within different scopes such as nested blocks and function scopes
    - Allows for a push/pop functionality when entering or leaving a scope, removing the need for individual removing or adding of entries
    - Allows for proper handling of shadowing (local vars hiding global)
- Difficulty in type checking and complex with various expressions and var (we used a generic function for expression type checking that helped)
- Need global flags for things such as checking if main exists and is last function and also ensuring return types match function declarations

Assumptions

- Assume cm file is correct syntactically
- Don’t enter both types of command line arguments at once, only one or the other.

Limitations

- No implicit conversions between int and bool
- Does not detect unreachable code (code after return)
- Some errors are difficult to recover from (hard to identify missing semicolons or if the statement is supposed to go to the next line)

Improvements

- More advanced type inference
- Warnings for unused variables or functions
- Use multiple symbol tables per scope
- Have a better way of accessing the type of a node (storing the name of the type in the node instead of inferencing it from an int)
- Implement boolean algebra

Contributions of each member

- Jeffery Wang
    - Documentation
    - Symbol Table Functionality

    - Semantic Analysis Functionality
    - Refactoring and Polishing
    - Final Testing and cleaning up
- Jubair Ali
    - Documentation
    - Implementing the AnalyzerPrinter class
    - Symbol Table Functionality
    - Semantic Analysis Functionality
    - Refactoring and Polishing
    - Final Testing and cleaning up
- Justin Wegner
    - Documentation
    - Implementing Command Line Arguments
    - AnalyzerPrinter Refinements
    - Refactoring and Polishing