CIS 4650: Checkpoint Two

Professor Fei Song

Andrew Trautrim, Andrew Truong

March 20, 2023

Table of Contents

What was done?

Overview

In this checkpoint a symbol table and type checker for the C- language was implemented. Specifically, the symbol table was used to maintain scope and to detect and report semantic errors such as mismatched types within expressions, undeclared identifiers, or redefinitions. For example, if a variable x was declared as an int, it would not be able to be re-assigned/redefined as a bool afterwards. Alongside that, it would also not be able to be redeclared as it already exists within the symbol table. The symbol table was implemented as a stack of hashmaps mapping variable names to their declarations, each hashmap representing a single scope. When entering a new scope, a new hashmap is pushed onto the stack, and when leaving a scope the last hashmap is popped off. When encountering a new declaration, it is added to the current scopes hashmap, and when referencing a variable, it is checked against all hashmaps within the stack. This ensures that any value within an expression is of the correct type and return types match the associated function declaration. Finally, in addition to error checking, the symbol table can be printed when given the “-s” flag in the input.

Design Process

The main challenge of this checkpoint was to implement the semantic analyser, which included the symbol table, type checking, and error reporting. Firstly, the syntax tree generated in checkpoint 1 was traversed in post-order, adding all declarations to the symbol table as it went along and checking for semantic errors. As mentioned before, in order to maintain scope, the

symbol table was implemented as a stack of hashmaps, this allowed for the scope to be easily maintained. Specifically, if a declaration was made, it was added to the hashmap at the top of the stack, and if a reference was made (e.g. a variable or a function call) it was checked against every hashmap within the stack; if any of these checks fail, an error is reported.

Moreover, the symbol table was checked through the syntax trees traversal to validate each declaration. Firstly, a variable could not be referenced if it did not already exist within the symbol table. Furthermore, a variable or function could not be declared multiple times within the same scope. However, a function can be declared twice if the first declaration is a prototype and the second declaration is its implementation.

Once the symbol table was implemented, it was mainly used for type checking, which was implemented for several scenarios. As mentioned before, the syntax tree was traversed in post order, this allowed the program to verify the types of subexpressions against the current expressions. For example, when checking an operation expression, the left hand side and right hand side were evaluated first and then their return types were compared against each other and with the current operation. The operators {+, -, *, /, >, >=, <, <=} required integer types, the operators {||, &&, ~} required boolean types, and the other operations {==, !=, =} only required that both sides be of the same type. Additional type checking was done as well, for example, return statement types were compared against the specified return type of the function, indexing expressions had to be integer types, and variables could not be declared as void types. Finally, if any of these checks failed, an error was reported with a location and an appropriate error message.

Reflection

Limitations

This checkpoint, when compared to checkpoint 1, led to fewer limitations overall. However, there were a few caveats encountered when implementing the symbol table. The symbol table acts as a stack when entering and exiting scopes, however, when checking for the existence of variables and functions, one has to check all entries within the stack. This led to the symbol table being implemented as an array with the functionality of a stack instead of using the built-in stack data type. Additionally, traversing the syntax tree in post-order does not allow for recursive calls without a prototype function declared beforehand. Furthermore, checking that function implementations were valid when a function prototype was declared beforehand led to some complications and reworking.

Potential Improvements

Fortunately, almost all semantic errors were successfully implemented. However, the error that could have been added if given enough time was verifying that all functions with a non-void return type ends with a valid return statement. This check was attempted but turned out to be much harder than we thought. Although, if given enough time I think that it could've been added as well. Moreover, checking that the arguments in a function call matches the definition of the function was not added. This was a relatively complex task that would have involved getting the expression types of all the arguments and comparing them against each parameter in

succession, which is a simple task but would require a reorganization of the code. Again, if given enough time, this check could've been added as well.

Contributions

Andrew Trautrim

- Symbol table/scope
- Tree traversal
- Type checking
- Error reporting
- Documentation

Andrew Truong

- Tree traversal
- Type checking
- Error Reporting
- Documentation