# Lab Five

Alex Smith

alex.smith1@Marist.edu

March 27, 2019

## 1 Crafting a Compiler

### 1.1 Exercise 8.1

The two data structures most commonly used to implement symbol tables in production compilers are binary search trees and hash tables. What are the advantages and disadvantages of using each of these data structures for symbol tables?

Binary search trees are good to implement symbol tables due to their O(n) time complexity. Another advantage is it is easy to recognize the parent and sibling scopes of the current scope. A disadvantage is that binary search trees only have 2 children which can limit the number of scopes within a block.

Hash tables are used to implement symbol tables within a scope because a hash collision can notify the compiler that a variable was re-declared within the same scope. Hash tables also have an average time complexity of O(1) which can be useful for searching for variables within the current scope or scope that the current scope is contained within. A disadvantage of hash tables is if a large amount of collisions were encountered in a given program (such as variables attempting to be re-declared often), the speed of the program would be considerably slower.

### 1.2 Exercise 8.3

Describe two alternative approaches to handling multiple scopes in a symbol table, and list the actions required to open and close a scope for each alternative. Trace the sequence of actions that would be performed for each alternative during compilation of the program in Figure 8.1.

Approach 1: An individual symbol table for each scope
This approach makes multiple symbol tables, one for each scope, by putting tables on a stack and pushing them when the scope is created and popping them when when the scope is closed. For Figure 8.1, a starting symbol table would store the symbols f and g. When the left bracket is detected, a new symbol table will be pushed on the stack. This new symbol table will contain int w and int x. The next line is

a left bracket which causes a new symbol table to be pushed onto the stack. This symbol table will store float x and float z. The first right bracket will be detected, closing the scope and popping the symbol table off of the top of the stack. The next right bracket will cause the next scope to be ended and will pop the current symbol table off of the top of the stack. The last symbol table will remain until the end of the program.

Approach 2: One symbol table for the entire program
This approach places all symbols in the same table and uses a depth or scope name attribute to distinguish different scopes. When a new scope is opened, the depth counter increases and when the scope closes, the depth counter decreases. In Figure 8.1, f and g will have a depth of 0 because no scope has been opened. When the first left bracket is detected, the depth counter increases to 1. Int w and int x will have a depth of 1, indicating they are in a different scope from f and g. The next left bracket will be detected and will increase the depth counter to 2. Float x and float z will have a depth of 2 and when the next right bracket is detected, the depth counter will decrease back to 1. The following right bracket will decrease the depth counter back to 0 and the program will end.