Scott Hansen

Lab 5

Crafting a Compiler:
Do exercises 8.1 and 8.3

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?

Hash Tables

Provided that the hash table is setup correctly (sufficiently large table, good hash function, appropriate collision-handling techniques), an insertion or retrieval from the table can take place in O(1) despite how many entries are in the table. The key is that hash tables must be implemented properly to take advantage of this. Aside from their complexity in relation to simple binary search trees, they are a very strong choice for symbol tables.

Binary Search Tree

On the other hand, binary search trees take advantage of the tree structure to provide the speed of a linked data structure for insertion with speed of binary tree search for lookup and retrieval. With random inputs, a tree can expect the performance of O(log n), where n represents the number of names in the tree.

In the real-world, this is not always true for symbol tables because programmers don't choose identifier names at random. Because of this, a tree could have large depths, causing performance during lookups to take closer to O(n) time. This can be avoided if one maintains a balanced tree, such as with red-black trees. Binary search trees do, though, have the advantage of being very simple. This along with malformed perceptions of realistic speed make them popular as symbol tables.

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.

Individual Table for Each Scope

In this approach, a whole new, unique table is created for every scope. Name scopes are opened and closed in a LIFO (last in first out) way and thus a stack can be used to organize such a search. In the event of a new scope being encountered, a new table is created as a child of the current one, and when a scope is closed, the current scope is kicked back up to the parent.

This approach, however, means that we may have to search in multiple parent scopes before a symbol is found. This approach means that we may have to check a number of intermediate scopes that don't have the symbol declared in order to finally reach an outer scope that does.

One Symbol Table

This approach involves entering all names in the same table. In the case of redeclaration in different scopes, the scope name and depth are used to ensure uniqueness. This approach avoids going through multiple scope tables in order to locate a name.

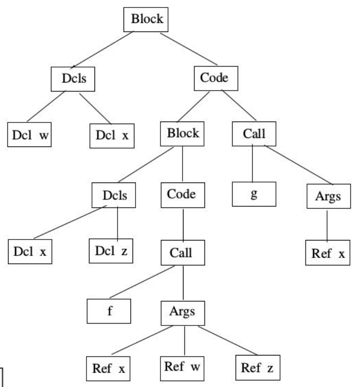
Sequence of actions

<u>Individual Tables</u>:
Generate first table
@Scope 0 table
add symbol to scope 0: Symbol 1, f, void func(float, float, float)
add symbol to scope 0: Symbol 2, g, void func(int)
encounter block, create new scope table as child of current scope
change to new scope
@Scope Table 1
add symbol to scope 1: Symbol 3, w, int
add symbol to scope 1: Symbol 4, x, float
encounter block, create new scope table as child of current scope
change to new scope
@Scope Table 2
add symbol to scope 2: Symbol 5, x, float
add symbol to scope 2: Symbol 6, z, float
check if f(x,w,z) is defined within current scope or parent scopes
encounter end of block, kick to parent scope
@Scope Table 1
check if g(x) is defined within current scope or parent scopes
encounter end of block, kick to parent scope
@Scope Table 0
EOP
<u>One Table</u>:
Create table and scope name 0, scope depth 0
add Symbol 1 to table @scope name 0, scope depth 0
add Symbol 2 to table @scope name 0, scope depth 0
encounter block, increase scope depth and create new scope name
add symbols 3 and 4 to table @scope name 1, scope depth 1
encounter block, increase scope depth and create new scope name
add symbols 5 and 6 to table @scope name 2, scope depth 2
check to see if function and symbols are defined in scope
end of block encountered, decrement scope depth and unique scope
check to see if function and symbols are defined in scope
end of block encountered, decrement scope depth and unique scope

EOP

```
import f(float, float, float)
import g(int)
{
  int w,x
  {
    float x,z
    f(x,w,z)
  }
  g(x)
}
```

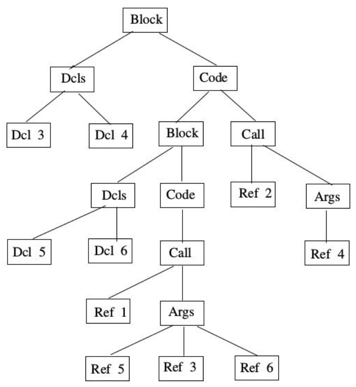
(a)



(b)

Symbol Number	Symbol Name	Attributes
1	f	void func(float,float,float)
2	g	void func(int)
3	w	int
4	x	int
5	x	float
6	z	float

(c)



(d)

Figure 8.1: Symbol table processing for a block-structured program