Lab 5

Liam Healy

liam.healy1@marist.edu

April 20, 2019

1 Crafting a Compiler, Chapter 8, Exercise 8.1 page 336

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?

For each approach to implementing symbol tables, we examine the time needed to insert symbols, maintain scopes and in particular, retrieve symbols.

The **binary search tree** approach combines the efficiency of linked data structure insertion with the efficiency of binary search for retrieval. Thus, given random inputs, it is expected that an object can be inserted or found in O(log n) time, given that n is the number of objects in the table. This is not always the case though since a tree with n names could have a depth of n, resulting in the time taken to retrieve an object shooting up to O(n). Still, this approach does have the advantage of being widely known, especially for students at Marist who have used them in several courses. Most programmers' expectancy of average-case performance also make binary search trees a popular technique. We can also use the node system in a binary search tree to identify a stack of currently active scopes that declare the object.

Hash tables are the most common way to implement and manage symbol tables, primarily due to their very good performance. Even with large tables, a good hash function and appropriate collision-handling techniques can perform insertion or retrieval in constant time. Some languages, like Java, include hash table implementation in their core library. In terms of performance, the hash table method outranks most others, if not all others.

2 Crafting a Compiler, Chapter 8, 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.

In languages such as C and Java, scopes can be opened and closed using braces($\{, \}$). Nested scopes can be opened within other scopes, however within certain nested scopes, certain actions, such as declaring methods in C or Java, are prohibited. Others with Algol-like syntax, use reserved keywords such as 'begin' and 'end' to open and close scopes. For symbol tables, there are two common approaches to their implementation, the first involves creating an individual table for each scope, and the second involves just one table.

An individual table for each scope involves a scope stack of symbol tables, with one entry in the stack for each open scope. The innermost scope appears at the top of the stack. The next containing scope is second from the top, and then continues. If a new scope is opened, we create and push a new symbol table on the stack. If a scope is closed, the top symbol table is popped from the stack.

One symbol table is an organization in which all names in a compilation unitâĂŹs scopes are entered into the same table. If a name is declared in a different scope, then we use a scope name or depth to identify the name uniquely in the table. Needless to say, while using the same actions to open and close a scope, we would not need to look through several tables to retrieve a symbol in this case.

Both implementations for the program in figure 8.1 -

```
procedure buildSymbolTable( )
    call processNode(ASTroot)
end
procedure processNode(node)
    switch (kind(node))
        case Block
            call symtab.openScope( )
        case Dcl
            call symtab.enterSymbol(node.name, node.type)
        case Re f
            sym <- symtab.retrieveSymbol(node.name)
            if sym = null
            then call error( âĂİUndeclared symbol : âĂİ,sym)
    foreach c âĹĹ node.getChildren( ) do call processNode( c )
    if kind(node) = Block
    then
        call symtab.closeScope( )
end
```

enterSymbol(name, type), retrieveSymbol(name), declaredLocally(name) **An individual table for each scope**:

1. symtab.openScope() - first entry in the stack of scopes' symbol tables
2. symtab.enterSymbol(node.name, node.type) - enter a symbol in the first scope symbol table
3. symtab.retrieveSymbol(node.name) - retrieve the corresponding symbol
4. symtab.closeScope() - pop the corresponding symbol table from the stack

One symbol table:

1. symtab.openScope() - used to identify depth within the symbol table
2. symtab.enterSymbol(node.name, node.type) - enter a symbol in the symbol table
3. symtab.retrieveSymbol(node.name) - retrieve the corresponding symbol using either it's name or depth
4. symtab.closeScope() - used to identify depth within the symbol table