CS 434 Meeting 2 — 2/7/06

Introduction

1. First phase of project will be assigned on Thursday.
2. Shall we schedule a C "lab" sometime.

Understanding Block Structure

1. First, recall the rules of nested block structure.
   - A scope is a subsection of a program's text typically corresponding to a procedure/function/method/class or a block of statements.
   - An identifier can have only one definition in a given scope.
   - An identifier can be used in a scope as long as it is either defined in the scope or in a containing scope. (Note: Sometimes forward references are not allowed.)
   - Any use of an identifier refers to the declarations of the identifier occurring in the smallest enclosing scope.
2. A very helpful way to understand such scope rules is to recognize that the nesting is just an alternate way to represent a tree structure:
   - The tree shown in Figure 1

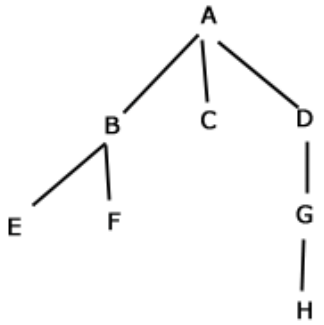

Figure 1: A simple tree structure

   can be encoded by the nested diagram shown in Figure 2, or by the textual nesting:

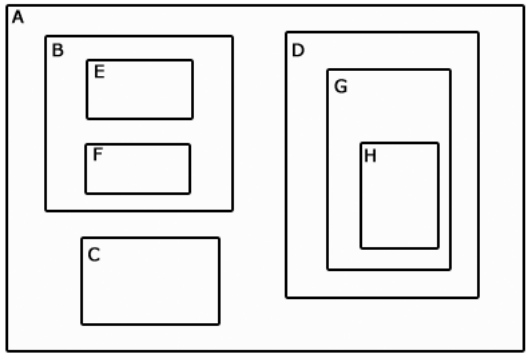

Figure 2: An alternate representation of a tree structure

```
begin A
        begin B
                begin E
                end E
                begin F
                end F
        end B
        begin C
        end C
        begin D
                begin G
                end G
                begin H
                end H
        end D
end A
```

3. For example, consider the interpretation of identifier references in the program whose skeleton is shown in Figure 5.
4. Within each node of this tree, we have written the names of the identifiers declared within the scope that corresponds to the tree node.

```
public class NestingExample {
    int x; int y;
        class One {
          Two y;  int z;

          class In1 {
              One y;
            }

          void m1() { int y;  y = 1;  x = y;  }
        }

     class Two {
        Three x;
        void m2()  x.m3();
        }

      class Three {
          Two x; One y;

          class In3 extends One {
              void mIn3() {
                  y.m2();
                  m1();
              }
          }

          void m3() { /* ... */ }
        }
}
```

Figure 3: A class definition skeleton illustrating nested declarations

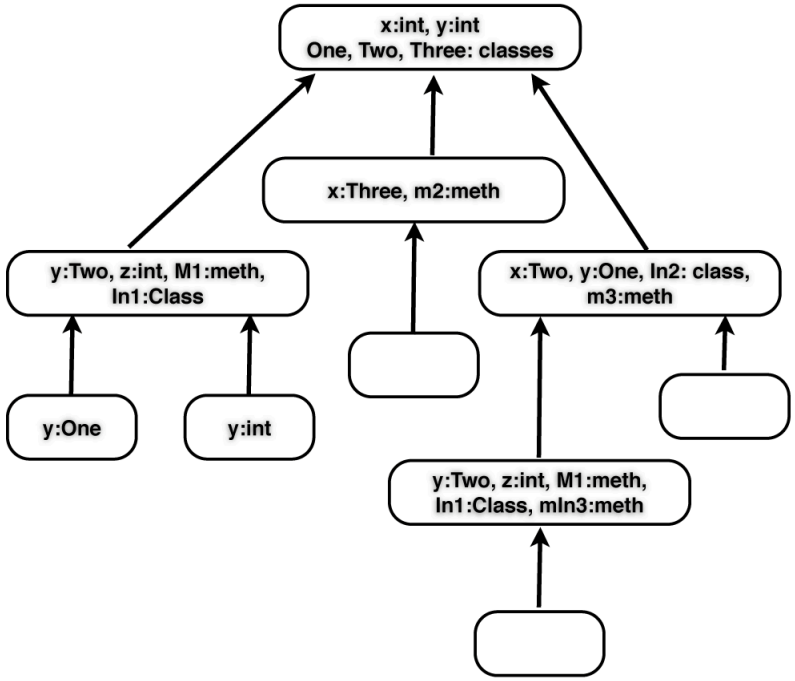

Figure 4: Tree of scopes corresponding to Figure 3

5. If an identifier is referenced within a given scope in the program, the correct definition can be found by sequentially searching the scopes on the path in this tree from the node for the scope to the root of the tree.

6. Note the interesting contents of the scope for the class `In3`. Because this class extends the class `One`. It includes new bindings for the names y, z, m1, and In1. These names are not redeclared. Rather, new bindings to existing declarations are created and added to the scope.

   - To appreciate the difference between adding bindings and new declarations, consider what happens if the method `m1` is invoked from within `In3`. In particular, when that method references `x`, it must use the binding for `x` in the scope of the declaration of `m1` (i.e. the declaration of `x` in the outer class) rather than the binding of `x` within `In3`.

Symbol Tables vs. Symbol Table Organization

1. The organization of most compiler symbol tables is fairly complex as a result of the need to support the scope rules associated with block structure. The standard approach to explaining symbol table organization, however, adds additional complexity by failing to properly distinguish the role of the scanner in building the symbol table from that of the semantic analysis routines in completing it.

2. Many compiler texts describe the symbol table as a dictionary, typically implemented using a hash table or a search tree.

   - In order to handle block structure, such texts proposed maintaining one hash table for every node in the scope tree, and searching them sequentially.

3. In my (somewhat odd) view, the symbol table is just a collection of records/structures in which the attributes of identifiers are stored. The hash table (or whatever else is used) is simply a mechanism that enables the scanner to associate symbol table entries with the character string form of identifiers it processes.

4. Once the program has been transformed into a syntax tree, the semantic analyzer has direct access to symbol table entries through pointers stored in the tree. The hash table used by the scanner is not needed. Thus, it seems wrong to me to talk about the hash table as if it were the symbol table.

5. The key to this approach is to start by thinking about the abstract entities that need to be represented in the symbol table.

   - Obviously, there are identifiers. We would like to have a single object/structure that represents all occurrences of a given identifier. That is, within the syntax tree, we would like all occurrences of a given identifier to be represented by pointers to the same object/structure. We will call these structures *identifier descriptors*.

     This is easy to accomplish using a simple hash table maintained by the lexical analyzer.

   - Next, there are declarations. Again, we want to have a single structure that describes each declaration found in the program. We will call these structures *declaration descriptors*. If this is all we worry about (i.e. we don't worry about which declaration goes with which reference to a given identifier), this is quite easy too. When the semantic processing routines in our compiler encounter a declaration, they just need to allocate and initialize a new structure describing the declaration.

   - Finally, there are associations or bindings between identifiers and declarations. The identifiers and declarations don't change as we enter and leave scopes, but the bindings do. If we know which bindings are active in a given scope, we can figure out which declaration goes with a given identifier at any points.

     Therefore, the third type of thing we should explicitly represent as a structure or object in our compiler is a binding between an identifier and a declarations. If we do this, then we will have a convenient way to keep track of things like the set of all bindings made in a given scope or the stack of bindings associated with a given identifier.

6. The syntax tree produced by the syntactic analyzer represents each identifier occurrence by a pointer to the corresponding identifier descriptor. Unfortunately, when processing the syntax tree to do code generation or type checking, it would be much more useful if each identifier in the program was represented in the tree by a pointer to the appropriate declaration descriptor.

   As a result, the first step in semantic processing will be to process the syntax tree creating the required declaration descriptors and replacing/augmenting the pointers to identifier descriptors in the tree by pointers to the appropriate declaration descriptors. We will call these processes *declaration processing* and *identifier reference resolution*.

7. To produce an efficient compiler, we want to avoid algorithms that require either time or space that is more than linear in the size of the program whenever possible. This implies that we need way to find the correct declaration descriptor for each identifier in our syntax tree in constant time!

8. A way we can arrange for such constant time processing is to ensure that when we encounter a reference to an identifier in the tree, the identifier descriptor for the identifier already points to a binding descriptor that in turn points to the the correct (i.e. current) declaration of that identifier.

    - This means we will have to do some (hopefully small) amount of work every time the "correct" declaration associated with any identifer changes.
    - Fortunately, the "correct" declaration/identifer associations only change when we enter and leave scopes.
        - Note: "Enter" and "leave" here refer to compile time traversal of the syntax tree not to run-time calls and returns.

9. As the semantic processing routines traverse the syntax tree, the bindings associated with a given identifier behave in a stack-like manner.

    - If a declaration of X is included in the block which we are begining to traverse, a binding between the identifier and the declaration found in that scope become the current binding for the identifier. That is, a binding descriptor referring to the new declaration gets pushed on top of the stack.
    - When a block that contains a declaration of X is exited, the binding that had been 'current' for X before the block was entered should become current again. That is, the newest binding on the identifier's stack is popped.

10. The stack of bindings for a given identifier can be kept as a linked list with the head pointer stored in the associated identifier descriptor.

11. If such stacks are maintained, it is easy to replace/augment each identifier descriptor pointer encountered while traversing the syntax tree.

    - When an identifier descriptor is found in the syntax tree, simply follow the stack head pointer stored in that identifier descriptor to find the top of the binding stack. Access the current declaration descriptor through the binding and store a pointer to that declaration descriptor in the syntax tree.
    - If a use of an identifier is encountered at a point where the binding stack is empty, the use should be reported as a reference to an undefined name.

12. It is easy to push the necessary binding descriptor onto the appropriate identifier descriptor's stack for each identifier declared within a scope.

    - The subtree of the AST representing the declaration will contain a pointer to the identifier descriptor.

13. Semantic processing of the end of a scope requires removal of all declaration descriptors on the current scope's list from the stacks of declaration descriptors attached to the identifier descriptors involved.

    - To make this efficient one can keep a list of all the bindings that were created in the current block.
    - Note: One actually needs to keep a stack of such lists (one list for each open scope). So, the semantic processing routines will need to push an empty list onto this stack of lists when a new scope is entered. We will call this the *open scope stack*.
    - We also need to make each binding or declaration descriptor hold a pointer back to its identifier descriptor.

14. To summarize the process suggested above (and to make it more algorithm-like:

    (a) The scanner creates a new identifier descriptor each time it sees an identifier it has not previously seen. It uses a hash table to keep track of the descriptors it has already created.

    (b) The semantic processor creates a declaration descriptor each time it encounters a declaration.

    (c) The semantic processor maintains a stack of binding descriptors for each identifier.
        - Each identifier's stack is pointed to by the identifier's descriptor.
        - The stack contains a binding descriptor for each declaration of the identifier associated with a scope that is currently "open" (i.e. that the semantic processing routines have started to work on but not yet completely finished).
        - The entry for the innermost scope is kept at the top of the stack. The outermost scope is at the bottom of the stack.

    (d) The semantic processor maintains a list of the bindings encountered in each open scope. We will call these lists *scope binding lists*. These lists are then organized in a stack call the *open scope stack* with the scope declaration list for the innermost scope at the top of the stack.

    (e) To process a scope (i.e. class, procedure, main program, etc.) the semantic processor
        - Pushes an empty binding list on the open scope stack.

- For each declaration, creates a declaration descriptor, pushes a binding referring to this declaration onto the identifier's binding stack, and adds it to the topmost scope binding list on the open scope stack.
- Scans the contents (statement, expressions, etc.) of the block replacing each pointer to an identifier descriptor by a pointer to the declaration descriptor that is currently pointed to by the binding on the top of that identifier descriptor's active binding stack.
- Closes the scope by popping a binding descriptor from the stack of every identifier descriptor on the topmost scope binding list and then pops this list off the open scope stack.

15. To understand how this all works together, consider the program shown in Figure 5.

```
class Program {
        int W;
        int X;
        class A {
                int W;
                int Y;
                int Z;

                void B() { int X ; . . . }
                void C() { int X; int Y ; . . . }
                . . .
        } // end of A

        void D() { int Z ; . . . }
}
```

Figure 5: A class definition skeleton illustrating nested declarations

- The diagram in figure 6 shows the state of the symbol table while processing the body of the method C in the program whose skeleton is shown in figure 5. (The symbol table nodes for method and class names have been omitted from this diagram but would be present in the actual symbol table.)

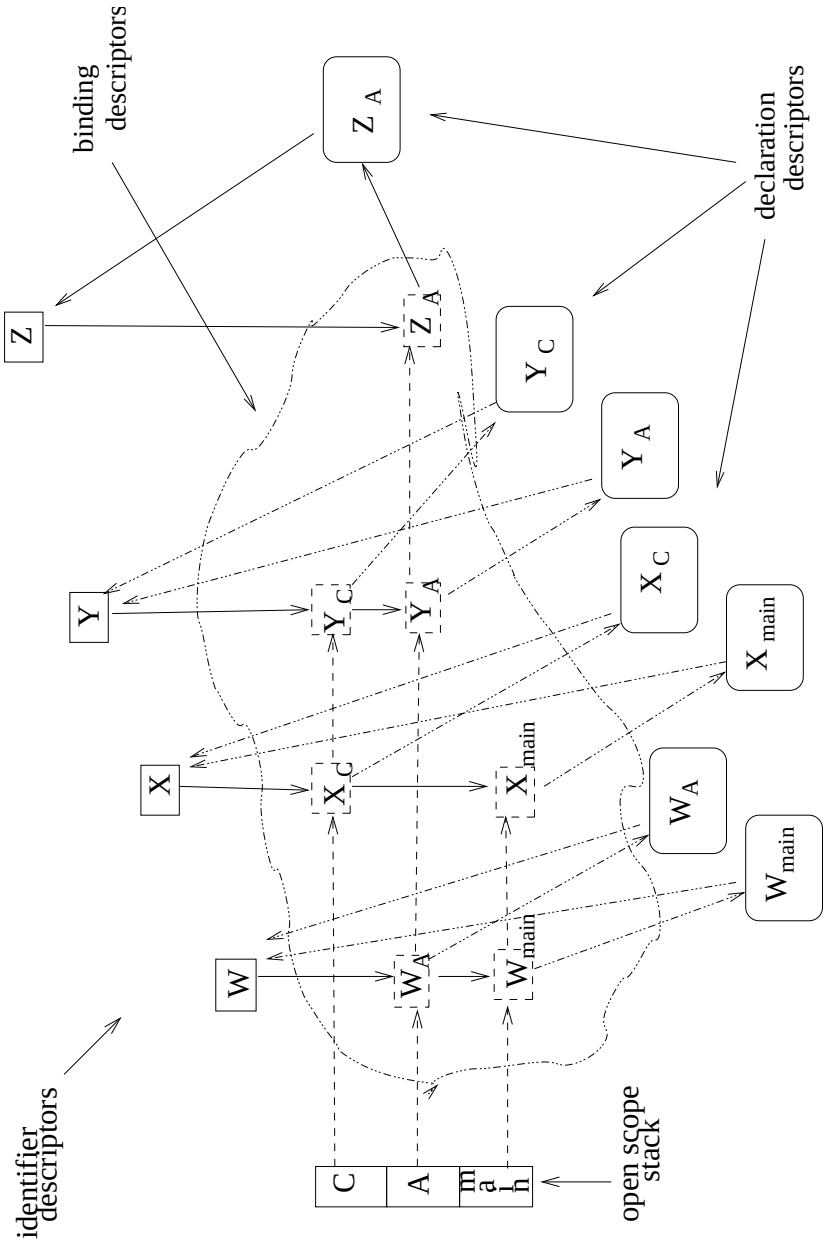

Figure 6: Symbol Table Organization