CS 434 Meeting 3 — 2/9/06

Introduction

1. Phase 1 + Intermediate form handouts will be available online (and outside my door?) by tomorrow.
2. Pick a language and a partner by tomorrow! Get started!
3. Brief tutorial on C programming shortly after class in TCL 104.

Outline

- Review symbol table organization
- Discuss contents of declaration descriptors
  - Sneak in need for type descriptors
- Structure of semantic processing code
- Handling targeted method invocations
- Semantic processing subphase ordering
- Handling inheritance
  - as a symbol table issue
  - as a type checking issue

Block Structure & Symbol Table Organization (cont.)

1. Last time we discussed how to organize a symbol table that would handle the scoping rules associated with block-structured programming languages.

   (a) We distinguished between four types of entries/structures that are used to manage the symbol table:

   Identifier descriptors -
   - One per identifier.
   - Created by scanner, used by later phases.
   - Accessed through hash table managed by the scanner.
   - Abstract syntax tree produced by the parser will use pointers to identifier descriptors to represent identifiers referenced in the program text.

   Declaration descriptors -
   - One per declaration.
   - Created during semantic processing, used in later phases.
   - Accessed through stacks of bindings associated with identifier descriptors and through pointers inserted in the abstract syntax tree during semantic processing.
   - After semantic processing the abstract syntax tree will use pointers to declaration descriptors to represent identifiers referenced in the program text.
   - Contain information about declared items like type, number of parameters, etc.

   Binding descriptors
   - One per activation of declaration (multiple activations can be created through inheritance).
   - Created during semantic processing, irrelevant to later phases
   - Contain pointers to the associated declarations
   - Contain pointers to lists of other bindings in scope
   - Contain pointers to stacks of hidden bindings of ids

   Scope descriptor
   - One per scope (classes and methods in Woolite) per pass...
   - Only used during semantic processing
   - Hold pointers to surrounding scopes and list of bindings in each scope

   (b) We sketched out the process followed to create symbol table entries and associate them with the appropriate nodes of the abstract syntax tree.

   i. 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.

ii. The semantic processor creates a new scope descriptor each time it begins processing the subtree of a class or method in the abstract syntax tree. It keeps these descriptors in a stack.

iii. The semantic processor creates a declaration descriptor each time it encounters a declaration.

iv. When the semantic processor wants to make an identifier accessible within a scope, it creates a binding for that declaration and:

- pushes it onto the stack of bindings for that identifier,
- adds it to the list of bindings for the current scope.

v. When the semantic processor encounters a use of an identifier in the tree, it will place a pointer to the current declaration descriptor for that identifier (i.e. the one pointed to by the binding on the top of the identifier's stack of bindings) in that tree node.

vi. When the semantic processor completes the processing of a scope, it pops the binding stack for each identifier for which there is a binding in the scope, then it pops the scope descriptor itself off a stack of scopes.

Declaration Descriptor Contents

1. Declaration descriptors are really the most important element of the symbol table. All the other elements are designed to provide an efficient way to find the correct declaration descriptors.

2. There are a few common elements we will want to include in all declaration descriptors:

   - We will want each declaration descriptor to include a pointer to the associated identifier descriptor (so that we can easily access the string of character used as the identifier for compile-time error message and symbolic debugging).
   - We will want each declaration descriptor to include the nesting depth at which the declaration occurred. When we generate code, we will depend on this information.

3. Beyond this short list of common elements, the contents will vary from one type of declaration to another:

   Variables and formals
   - For each variable we will want to include a description of its type. As a result, shortly, we will describe yet another type of descriptor, the *type descriptor*.
   - For code generation, we will eventually need information about the offset to the variable within the heap object or method activation record that contains it. On the WC34000, all data objects require one word, so if we simply count variable declarations and store the current count in each variable's descriptor, this will be sufficient. Note, however, that for instance variables, this count must include all variable's in super classes.

   Methods
   - We will want to store the method's return type
   - We will want to keep a pointer to a list of the declaration descriptors for the method's formals (this will be needed to type check invocations of the method (among other things)).
   - We will need a count of the number of local variable so that we can tell how much memory to allocate for an frame/activation record when the method is invoked.
   - For code generation, we will build a table of pointers to the code for each of the methods associated with a class. Therefore, like variables, we will want to associate an offset with each method. Again, as with variables, since all data values occupy 1 word on the WC34000, a simple count of this method's position within the list of methods associated with the class will do the job. Note that this count will have to include methods defined in superclasses, but exclude methods that simply override existing methods (since they will not get a new slot in the table but instead reuse the old slot).

   Classes
   - If the class is a subclass, we will want to know its superclass.
   - To allocate objects of a class, we will have to keep track of the total space required for variables. Again, on the WC34000 all variables will take one word. Therefore, the class declaration

descriptor is a good place to keep a count of how many variables we have processed.

- We will also want to keep the count of the number of distinct method names associated with the class.

4. As mentioned, we now need to design yet another descriptor used to hold information about the type of variables, expressions, etc. in Woolite.

   - Luckily, there aren't too many choices. The only types are `int`, declared classes, and arrays of the above. A type descriptor can therefore simply be a pair including
     - a pointer to the base type (a class declaration descriptor), and
     - a count of the number of dimensions (where 0 means the type is not an array at all).

Semantic Processing Subphase Ordering

1. The fact that Woolite supports arbitrarily deep nesting of class definitions and allows forward references to declarations (except in extends clauses) makes it necessary to make several partial sub-passes over the syntax tree and to think very carefully about what processing to perform during each pass.

2. For example, consider the simple program shown in Figure 1. Even though this program only includes three declarations at the top level, it is enough to show that several "obvious" strategies for processing the declarations won't work.

   - If you try processing the declarations in order, you get into trouble immediately. When you go to process the declaration of the variable x, you discover that it is supposed to be of type `Forward`, but you haven't processed the declaration of `Forward` yet. You will want to put a pointer to the declaration descriptor for `Forward` into the declaration descriptor for `x`, but the descriptor for `Forward` won't even exist yet.

```
class Program {
      Forward x;

      class Forward {
            int counter;

            void relay() {
                  x.action();
             }

             void action() {
                 counter = counter + 1;
             }

             void init() {
                 counter = 0;
             }

             int get() {
                 return counter;
              }
        }

        int main() {
            x = new Forward;
            x.relay();
            return x.get();
        }
}
```

Figure 1: Coping with forward references

- If you try to process the declaration of `Forward` before `x`, however, you will just run into a different problem. Forward references x in its relay method. If there is no declaration descriptor or binding for x, how can you resolve this reference to x?
- If you try the last possibility, processing the method main first, you again encounter references to x before you have created its declaration descriptor.

3. In addition to these (relatively simple) issues raised by forward references, the ability to invoke a method on an object leads to a form of vertical forward reference through which it appears to become necessary to process inner scopes before outer scopes.

   - Consider the program in Figure 2.

4. To deal with this, your semantic processing phase will make two passes over the syntax tree.

   - During the first pass, you will completely ignore method bodies.
   - This pass will have two goals:
     (a) to build (partial) declaration descriptors for each class, instance variable, and method declaration in the program.
     (b) to construct the hash table that maps class/method-name pairs to method declaration descriptors
   - During this first pass, you will construct scopes and place lots of bindings, but you will discard all this information as your traversal exits scopes.
   - During the second pass, you will again traverse the tree, this time processing method bodies.
   - While you will have to rebuild all the bindings and scopes you built in the first pass, you won't have to revisits the parts of the syntax tree examined during the first pass (class and method headers, and instance variable declarations). Instead, you can access the declaration descriptors you constructed from these elements of the tree from other declaration descriptors and simply create new bindings to refer to them.

```
class FourDeep {

    class A {
        int meth() {
            B x;
            x = new B;
            return x.meth().meth();
        }
     }


    class B {

        class C {
            int meth() {
                return 4;
            }
         }

        C meth () {
            return new C;
        }
    }

     int main() {
        A y;

        y = new A;
        return y.meth();
     }
}
```

Figure 2: Accessing methods from inner scopes