Project (part 1): Symbol table module
Due: 8:00 AM, April 4, 2011.

1 Overview

The first part of the compiler involves constructing a symbol table module. A symbol table module is used during the compilation process to hold information about the entities that a program declares as well as the attributes that the program associates with the entities. Examples of different kinds of entities are types, variables, constants and methods. Examples of attributes of an entity are its types, arguments, scope and others.

2 Goal

The primary goal of this project is to develop a set of C++ (or Java) classes that will be used to store and access information about the entities of programs. Note that much of the symbol table module will evolve as the rest of the compiler develops. We are, therefore, primarily interested in developing only the basic infrastructure needed to implement the symbol table module in this project. In the later parts of the project, you will be extending and modifying the symbol table module in order to include additional information. Also, you will be combining this module with the parser, semantic analyzer, and the code generator components for semantic analysis and code generation. It is, thus, important that you use modular programming techniques for writing this module since you will be changing the module extensively.

3 Name and scope

Your symbol table module will implement two kinds of mechanisms: a table for storing information about names declared in a specific scope, and a mechanism for maintaining scopes and resolving the usages of names. The following example will illustrate the two. Let the following program define a class, called `Dummy`:

```
class Dummy {
  int vis;                    // Line 1
  char val;                   // Line 2
  int f(int arg) {            // Line 3
     int val;                 // Line 4
     return (arg + vis + val);  // Line 5
  }
}
```

The class declares three entities: `vis`, `val` and `f`. Entities `vis` and `val` are instance variables, whereas `f` is a method. Note that the program associates various attributes with the names. For instance, `vis` is a variable of type integer, whereas `f` is a method that takes an argument of type `integer` and returns a value of type `integer`. In this program, declarations of `Dummy` and `f` define unique scopes. (Note that the scope defined by `f` is nested within the scope defined by `Dummy`).

There are two kinds of information that your symbol table module must maintain: Information about entities defined within a scope and the visibilities of entities in different scopes. The first involves storing and maintaining information about all entities defined within a scope. For instance, `vis`, `val` and `f` are declared within the scope defined by `Dummy`. Hence, information regarding these entities must be maintained. Similarly, information regarding the entities (`val` and `arg`) of `f` also needs to be maintained. This information is usually maintained by constructing a symbol table for storing information about entities that a program entity introduces. For instance, a symbol table for `Dummy` will store information about entities of `Dummy`, whereas a symbol table for `f` will store information about `f`'s entities.

The second component of the symbol table module involves maintaining the visibility of entities. This information is used to determine the correct definition of an entity given its name. The visibility of various entities of a program are determined by the scope rules of a programming language. In this project, it will be determined through static scoping. For instance, in line 5 of the above example, symbols `arg`, `vis` and `val` are used. The scope rules of a programming language determine what these symbols mean and how their definitions can be found. In this case, the definition for `vis` comes from the scope defined by `Dummy`, whereas the definition for `val` comes from the scope defined by `f`. Note that the declaration of `val` in `f` overrides the declaration of `val` in `Dummy`. The symbol table package needs to not only maintain information about variable program entities but also their scopes.

4 Symbol Table

Write a symbol table class for storing entities of a program. The entities are defined by an identifier (a string) and a set of attributes. For this phase, assume that each entity has a single attribute which is denoted by a class `EntityAttribute`. You can declare `EntityAttribute` as following:

```
class EntityAttribute {
public:
   EntityAttribute() {;}
   ~EntityAttribute() {;}
};
```

Note that you will need to change `EntityAttribute` in the later stages of the compiler. Currently it acts as a place holder.

The symbol table class can be implemented through a container data structure such as hash table, balanced binary tree, or one of the containers such as `map` or `multimap` of STL. Though `map` or `multi-map` may not be as efficient (check their worst case complexities) as a hash table based implementation[1], I suggest that you choose `map` or `multimap` for implementing your symbol table.
Define the following set of methods for your symbol table class:

- Constructor and destructor;
- Method for adding an entity to a symbol table;
- Method for deleting an entity from a symbol table; and
- Search for an entity given its identifier.

[1]Hash tables are the most common means of implementing symbol tables in production compilers and other system components. With a large enough table, a good hash function, and appropriate collision-handling techniques, searching can be done in essentially constant time.

5 Scope resolution

Implement the mechanism for defining and maintaining scopes of names. This is done by storing names within each scope in separate symbol tables. Since accessing the attributes of a name may span over many symbol tables, your scope resolution mechanism must implement any data structures needed to maintain nested scopes and visibility of symbols.

I will suggest one approach for implementing the scope resolution component. The scopes defined by a program can be viewed as a tree, called *scope tree*, with the outermost scope as the root node, the next inner scopes as its children nodes and so on. (You may want to check my notes on scopes from 140A.) The entities accessible from any given scope are only those that are defined in this scope or in one of its ancestor in the scope tree. Searching a name within a scope, therefore, requires searching the symbol table defined in the current node and searching incrementally all nodes in the path from the current node to the root node. Since your implementation does not need to keep track of the entire scope tree but only the branch that connects the current scope to the root, the scoping rules can be implemented by a `stack`[2], called *scope stack*, that holds the symbol tables for different scopes.

Entering a new scope involves creating a new symbol table for the scope and pushing it on the top of the scope stack, whereas exiting an scope involves popping the top symbol table off the stack. The top element of the stack will, thus, contain the symbol table for the innermost scope, the next element will store the scope surrounding the innermost scope and so on. Searching a name will involve traversing the stack from the top to bottom, looking for the first occurrence of the identifier in the symbol tables in the stack.

Define the following set of operations for the scope resolution mechanism

- Create a new scope: create a new symbol table.
- Enter a scope: create a scope and make it current.
- Exit a scope: discard current scope and make the previous scope current.
- Search for a name: search for a name in the scope stack.

You can use the `stack` container defined in STL for implementing the scope resolution mechanism.

6 Submission

Submit the following as part of the project:

- Source files for the various classes.
- A driver file: write a driver file that will exercise the methods of the various classes of your symbol table.
- A makefile to build your program.

[2]Note that this stack is slightly different from the traditional traditional stacks in that you will need to be able to search within the stack.