ECE251: Programming Languages & Translators Fall 2010

Lecture 16 — October 14, 2010

Patrick Lam *version 1*

We just saw the theory of operation behind symbol tables. Now we'll see how to build them. This knowledge will enable you to do the second part of the final project.

Building Symbol Tables[1]

The symbol table maps identifiers to meanings, e.g.

`i`	local	`int`
`done`	local	`boolean`
`insert`	method	...
`List`	class	...

Interpreters use symbol tables to store values for variables. Compilers use symbol tables to store type and (possibly) address information for variables. A symbol table enables compilers to:

- detect undeclared variables and multiply-defined variables;
- find the appropriate definition for a variable reference; and
- compute how much space to allocate for variables.

We will assume:

- static scoping; and
- names are declared before use, at most once per scope.

Symbol tables constitute an abstract datatype. The concept of ADTs should be familiar to you from ECE250. We define the basic operations:

- insert new name in current scope (`void insertNewName(Var v)`);
- look up name in local scope only (`Var findLocal(String s)`);
- look up name in all enclosing scopes (`Var find(String s)`);
- create and enter a new scope (`SymbolTable createScope()`); and
- leave a scope and destroy variables in that scope.

[1] `http://www.d.umn.edu/~rmaclin/cs5641/Notes/L15_SymbolTable.pdf`

A List of Tables

At any point, we have a current `SymbolTable` object, which keeps a parent `SymbolTable` as internal state, as well as the bindings defined in the current scope up to the current program point, perhaps in a Java `HashMap` mapping `String`s for variable names to `Var`s containing type information, etc.

```
class X {
  final int i;
  class Y {
    int i;
    public int getLocalI(int x) { int i = 4; return i; }
  }
}
```

We can graphically represent the symbol table at `return i` as follows:

Operations. The first two operations, `insertNewName` and `findLocal`, simply need to delegate to the current `HashMap`.

To implement `find`, we recursively search each enclosing symbol table for the requested symbol by following `parent` pointers. We might return `null` if a symbol turns out to be undefined.

To implement `createScope()`, we instantiate a new `SymbolTable` object and set its parent pointer to the current scope.

Leaving a scope simply requires moving the current scope pointer up to its parent.

Using the Operations. Here's what we do.

- Upon scope entry: call `createScope` and set the current scope to what it returns
- At a declaration of `x`: use `findLocal(x)` and report a multiply-defined error if true, otherwise `insertNewName`.
- At a use of `x`: use `find` and identify the proper definition.
- Upon scope exit: set the current scope to the parent.

Other implementations

We don't have to use a list of tables. We could also use a table of lists, or a cactus stack.