ECE251: Programming Languages & Translators Fall 2010

Lecture 15 — October 13, 2010

Patrick Lam *version 1*

We expect compilers to keep track of *names*, or symbols. The compiler has to remember which name corresponds to which address in memory at which point in time. Chapter 3.3 in the textbook discusses these issues in greater detail. In this lecture, we'll talk about design choices behind the symbol table, and the applicable scoping rules.

Scoping

In Lab 1, we had one global mapping of variable names to values; I suggested that your `Interp` class keep track of this store. Real programming languages allow reuse of names:

```
class X {
  final int i;
  public X(int i) {
    this.i = i;               // assign field to arg
  }

  class Y {                   // inner class
    int i;                    // also defines a field i
    public int getLocalI() { int i = 4; return i; }
    public int getInnerI() { return i; }
    public int getOuterI() { return X.this.i; }
  }
}
```

Obviously our simple solution from Lab 1 doesn't work here. We will need a *symbol table*. (Note that you can't even check that all names are defined in the parsing stage; defined-ness is not a context-free property.) But let's first talk about the design choices behind the symbol table.

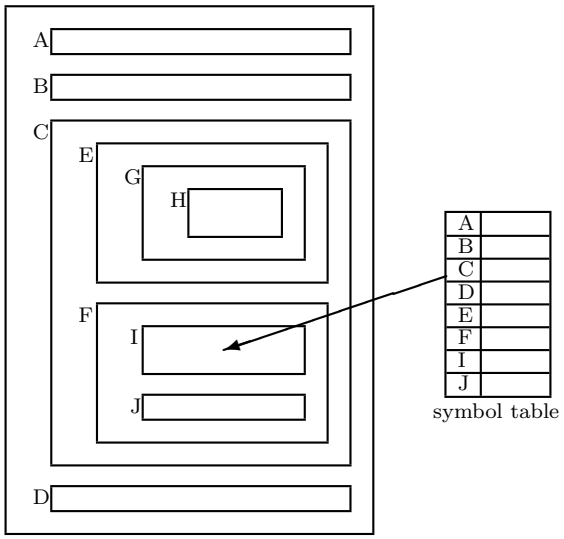
We will explore two different kinds of scoping. The difference is, when a name is not defined in the current scope, which scope should we search next? In *dynamic scoping*, we search the callers to a procedure (considered insane in most contexts, these days), while *static scoping* searches the lexically-enclosing scopes. Java, and most sane languages, use static scoping.

Static Scoping

Most modern programming languages use the closest declaration of a variable, searching outwards from the use point. Static scoping makes it possible to tell, at compile time, the location that each variable refers to. We will discuss implementation strategies for static scoping later.

Nested procedures and classes. Object-oriented programs contain classes, which contain fields and methods. Methods contain local variables. So, as a first approximation, given a variable `v`, we search the method for a declaration of local variable `v`; if it doesn't exist, we then look for field `v`. If neither exist, then the compiler issues an error.

Java allows nesting of classes ("inner classes"); other languages also allow nested procedures. Because classes may be arbitrarily nested, we need to use symbol tables to store information about nested variable names and to match up variables with their definitions.



Holes in scopes. The above examples show instances where a variable declaration in some inner scope *hides* a variable in an outer scope. We can also call this a *hole* in the scope of the variable. For instance, the constructor for `X` above cannot read the field `i` without explicitly asking for it. Java, however, allows you to access hidden variables by fully identifying them.

Static variables. Some languages also have a notion that is inverse to holes in scopes. In C, these are known as *static variables*. While the name goes out of scope, the storage location for the variable continues to persist between calls.

```
int counter() {
  static int i = 0;
  return ++i;
}
```

In the above example, you can't access `i` between calls to `counter`, but it keeps its value.

Declaration order. Where are you allowed to make a declaration? In modern languages (e.g. Java, modern C, C++), you can declare a variable anywhere you can write a statement, and the declaration only takes effect after it occurs. But, you can also require that all declarations precede all code, or allow a use of a variable to implicitly define that variable.

Here's a surprising side-effect of Java's declaration order semantics. What does this print?

```
public class Y {
    static int x;
    public static void main(String[] argv) {
        x = 5;
        System.out.println(x);
        int x = 2;
        System.out.println(x);
    }
}
```

Declarations vs definitions. If you want to require that everything is declared before it is used, how can you write mutually-recursive definitions?

C allows you to declare something before defining it, thus enabling recursive definitions. For instance, consider the following C code, which declares first and defines later.

```
void list_tail (follow_set fs);   // declaration
void list (follow_set fs) {
  switch (input_token) {
    case id: match(id); list_tail(fs);
  }
}
void list_tail (follow_set fs) {  // definition
  // more code...
}
```

Nested blocks. We can set up micro-scopes even within methods, which is helpful when writing code.

```
{
    int temp = x;
    x = y;
    y = temp;
}
```

Note that Java won't let you hide already-existing local names this way.

Dynamic Scoping

Some programming languages use dynamic scoping. The only one you are likely to encounter is shell scripts: you can set variables, and they get passed along to callees. Perl may use dynamic scoping (at the programmer's option), and so did early versions of LISP. TEX is one other example of a dynamically-scoped language.

It is hard to statically detect errors with dynamic scoping. There are some advantages to dynamic scoping; you can think of it as a generalization of global variables.

Static versus dynamic scoping example. Consider the following code:

```
n: integer;
procedure first
  n := 1

procedure second
  n: integer
  first()

n := 2
if read_integer() > 0
  second()
else
  first()
write_integer(n)
```

In a dynamically-scoped language, depending on the input, procedure `first` writes to either the global variable `n` or the local variable `n` declared in `second`. This is hard to understand and also hard to compile into efficient code.