Class 14: Nested Scopes and Declaration Order

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011
int x = 10;

int foo(int y) {
    x = y+5;
    print(x);
}

int main() {
    int x = 8;
    foo(9);
    print(x);
}

What happens in a dynamic vs. lexically scoped language?
Another dynamic/lexical example

```c
int width = 10;
char justification = 'L';

void print(string s) {
    int space = width - length(s);
    if (justification == 'L') print(s);
    for (int i=0; i<space; ++i) print(' ');           
    if (justification == 'R') print(s);
}
```

Suppose we want a function `foo` that prints a series of names, using the existing `print` function, all right-justified to 20 characters width. How would we write this in a dynamic vs. a lexically scoped language?
Another dynamic/lexical example

In a dynamically scoped language, we could just write

```c
void foo(string names[], int n) {
    int width = 20; char justification = 'R';
    for(int i=0; i<n; ++i)
        print(names[i]);
}
```

In a lexically-scoped language, we would have to change the global values of `width` and `justification`, and then (to be nice) change them back before returning.
Another dynamic/lexical example

In a dynamically scoped language, we could just write

```c
void foo(string names[], int n) {
    int width = 20; char justification = 'R';
    for(int i=0; i<n; ++i)
        print(names[i]);
}
```

In a lexically-scoped language, we would have to change the *global* values of `width` and `justification`, and then (to be nice) change them back before returning.

What would the effect of *nested function calls* be on the above strategies?
Nested scopes

Certain language structures create a *new scope*. For example:

```c
int temp = 5;

// Sorts a two-element array.
void twosort(int A[]) {
    if (A[0] > A[1]) {
        int temp = A[0];
        A[0] = A[1];
        A[1] = temp;
    }
}

int main() {
    int arr = {2, 1};
twosort(arr);
cout << temp; // Prints 5, even with dynamic scoping!
}
```
In C++, nested scopes are made using curly braces (\{ and \}). The scope resolution operator :: allows jumping between scopes manually.

In most languages, function bodies are a nested scope. Often, control structure blocks also form nested scopes (e.g. for, if, etc.)

*Lexical scoping* creates a tree structure with the nested scopes. Every name that is *visible* within some scope is either defined *locally* within that scope, or is defined *above* somewhere on the path from the root.
Nested Functions

With nested functions, we have to consider scope and allocation rules.

```c
void f(int a, int b) {
    int g(int c) {
        return a + c;
    }
    if (a == 0) return;
    print(g(g(b)));
    f(a−1,b+1);
}
```

What integers are printed from the call \( f(5,5) \)?
Nested Functions

With nested functions, we have to consider scope and allocation rules.

```c
void f(int a, int b) {
    int g(int c) {
        return a + c;
    }

    if (a == 0) return;

    print(g(g(b)));
    f(a-1, b+1);
}
```

What integers are printed from the call `f(5, 5)`?

15, 14, 13, 12, 11
Declaration Order

In many languages, variables must be *declared* before they are used. (Otherwise, the first use of a variable constitutes a declaration.)

In C/C++, the scope of a name starts at its declaration and goes to the end of the scope. Every name must be declared before its first use, because names are *resolved* as they are encountered.

C++ and Java make an exception for names in *class scope*. Scheme doesn’t resolve names until they are evaluated.
Consider the following familiar code:

```c
void exp() { atom(); exp tail(); }

void atom() {
    switch(peek()) {
        case LP: match(LP); exp(); match(RP); break;
        // ...
    }
}
```

Mutual recursion in C/C++ requires *forward declarations*, i.e., function prototypes.

These wouldn’t be needed within a class definition or in Scheme. C# and Pascal solve the problem in a different way…
Class outcomes

You should know:

- Relative advantages of dynamic and lexical scoping.
- The motivation behind declare-before-use rules, and their effect on mutual recursion.

You should be able to:

- Draw the tree of nested scopes for a lexically-scoped program.
- Trace a program with nested function calls using lexical scoping.