Class 13: Name, Scope, Lifetime

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Homework Review

1. Generate parse tree for the program

```
x := 5*3 + 4;
x > 10 & x/2 < 10;
```

2. Write the AST for that program.
3. Decorate the AST with the type of each node.

Naming Issues: Example 1

We need to know what thing a *name* refers to in our programs.

Consider, in Perl:

```perl
$x=1;
sub foo() { $x = 5; }
sub bar() { local $x = 2; foo(); print $x,"\n"; }
bar();
```

What gets printed for x?

Naming Issues: Example 2

We need to know what thing a *name* refers to in our programs.

Consider, in Scheme:

```
(define x 1)
(let ((x 2))
  (display (eval 'x)))
```

What gets printed for *x*?

Naming Issues: Example 3

We need to know what thing a *name* refers to in our programs.

Consider, in C++:

```
char* foo() {
  char s[20];
  cin >> s;
  return s;
}

int bar (char* x) { cout << x << endl; }

int main() { bar(foo()); }
```

What gets printed for *x*?

Basic terminology

- **Name**: A reference to something in our program
- **Binding**: An attachment of a *value* to a *name*
- **Scope**: The part of code where a *binding* is active
- **Referencing Environment**: The set of bindings around an expression

- **Allocation**: Setting aside space for an object
- **Lifetime**: The time when an object is in memory

Key Question

At a given point in the execution of our program,
what thing does a name refer to?

- We need to know this as *programmers*.
- We really need to know this as *compiler developers*.

Static Allocation

The storage for some objects can be fixed at compile-time.
Then our program can access them *really quickly*!

Examples:

- Global variables
- Literals (e.g. `"a string"`)
- *Everything* in Fortran 77?

Stack Allocation

The run-time stack is usually used for function calls.
Includes local variables, arguments, and returned values.

Example: What does the stack look like for this C program?

```
int g(int x) { return x*x; }

int f(int y) {
  int x = 3 + g(y);
  return x;
}

int main() {
  int n = 5;
  f(n);
}
```

Heap Allocation

The heap refers to a pile of memory that can be taken as needed.
It is typically used for *run-time memory allocation*.

This is the *slowest* kind of allocation because it happens at run-time.
More common in dynamic languages.

Compilers/interpreters providing *garbage collection* make life easier with lots of heap-allocated storage.
Otherwise the segfault monsters will come...

Single Global Scope

Why not just have every instance of a name bind to the same object?
This will make the compiler-writer's job easy!

Dynamic vs. Lexical Scope

Perl's `local` variables have *dynamic scope*.
The binding is determined by the most recent declaration *at run time*.

Most other languages (and `my` variables in Perl) have *lexical scope*.

The binding is determined *statically* (at compile time) as the closest *lexically* nested scope where that name is declared.

(Note: this is actually the hardest to implement!)

Dynamic vs. Lexical Example

```
int x = 10;

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

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

How does the behavior differ between a dynamic or lexically scoped language?

Class outcomes

You should know:

- The meaning of terms like *binding* and *scope*
- The trade-offs involved in storage allocation
- The trade-offs involved in scoping rules

You should be able to:

- Show how variables are allocated in C++, Java, and Scheme.
- Draw out activation records on a run-time stack.
- Determine the run-time bindings in a program using static, dynamic, and lexical scoping.