Naming Issues: Example 1

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

Consider, in 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)
(define (foo x)
  (lambda () (display x)))
((foo 5))
(display 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 active bindings at the point of an expression
- **Allocation**: Setting aside space for an object
- **Lifetime**: The time when an object is in memory

Options

Scoping

- **Single Global Scope**
  Just one symbol table
- **Dynamic Scope**
  Stacks of scopes, depends on *run-time* behavior
- **Lexical Scope**
  Scope is based on the syntactical (lexical) structure of the code.

Allocation

- **Static Allocation**
  Allocation fixed at compile-time
- **Stack Allocation**
  Follows function calls
- **Heap Allocation**
  Done at run-time, as objects are created and destroyed

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.

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?
(Compiler writing would be easier!)

What is a scope?

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

```
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!
}
```


Nested Scopes

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 are also (e.g. `for`, `if`, etc.)

Lexical scoping follows the nesting of scopes in the actual source code (as it is parsed).
Dynamic scoping follows the nesting of scopes as the program is executed.

Declaration Order

In many languages, variables must be *declared* before they are used.
(Otherwise, the first use 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 evaulated.

Declaration Order and Mutual Recursion

Consider the following familar code:

```
void exp() { atom(); exptail(); }

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...

Dynamic vs. Lexical Scope

Dynamic Scope

- Bindings determined by *most recent declaration* (at run time)
- The same name can refer to many different bindings!
- Examples:

Lexical Scope

- Bindings determined from lexical structure at compile-time
- The same name always refers to the same binding.
- More common in "mature" languages
- Examples:

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?

Implementing Dynamic Scope

A *Central Reference Table* is used to implement dynamic scope.

This *global* object contains:

- A mapping of names to *stacks of values*.
  Declaring a new binding pushes onto the stack; exiting that binding's scope pops off the top of the stack.
- A stack of sets of names. Each set stores the names declared in some scope (so we know what bindings to pop!).

Example: Central Reference Tables with Lambdas

```
{
  new x := 0;
  new i := -1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    if (i > 0) { ret := p@0; }
    else {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

What gets printed by this (dynamically-scoped) SPL program?

Lexical Scope Tree

Name resolution in lexical scoping follows the *scope tree*:

- Every (nested) scope is a node in the tree.
- The root node is the global scope
- Nodes contain names defined in that scope.
- To determine active bindings, follow the tree up from the current scope until you see the name!

Example (program on previous slide):

Reminder: The class of functions

Recall that functions in a programming language can be:

- **Third class**: Never treated like variables
- **Second class**: Passed as parameters to other functions
- **First class**: Also returned from a function and assigned to a variable.

Implementing Lexical Scope

With *lexical scoping*, rules for binding get more complicated when functions have more flexibility.

- Third-class functions:
  Can use "static links" into the function call stack
- Second-class functions:
  Can use "dynamic links" into the function call stack
- First-class functions:
  Must use Frames

Lexical Scope with 1st-Class Functions

What happens here?

```
{
  new f := lambda x {
    new g := lambda y { ret := x * y; };
    ret := g;
  };

  new h := f@2;
  write h@3;
}
```

Where are the *non-local references* stored?

Frames

A *frame* is a data structure that represents the *referencing environment* of some part of a program.
It contains:

- A link to the *parent frame*.
  This will correspond to the *enclosing scope*, (or `null` for the global environment frame).
- A *symbol table* mapping names to values.
  (Notice: no stacks!)

Looking up a name means checking the current frame, and if the name is not there, *recursively* looking it up in the parent frame.

Function calls create new frames.

SPL Example for Frames

How would this program work using *lexical scoping*?

```
new x := 8;

new f := lambda n {
  write n + x;
};

{ new x := 10;
  write f@2;
}
```

- How do frames compare with activation records on the stack?
- Can we use frames for *dynamic* scoping?

Closures

How are functions represented as values (i.e., first-class)?
With a *closure*!

Recall that a closure is a function definition plus its referencing environment. In the frame model, we represent this as a pair of:

- The function definition (parameters and body)
- A link to the frame *where the function was defined*

Example with closures

Draw out the frames and closures in a Scheme program using our stacks:

```
(define (make-stack)
  (define stack '())
  (lambda (arg)
    (if (eq? arg 'pop)
        (let ((popped (car stack)))
          (set! stack (cdr stack))
          popped)
        (set! stack (cons arg stack)))))

(define s (make-stack))
(s 5)
(s 20)
(s 'pop)
```


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
- The motivation behind declare-before-use rules, and their effect on mutual recursion.
- Why some languages restrict functions to 3rd-class or 2nd-class
- What non-local references are, and what kind of headaches they create
- How memory for local variables is allocated when in lexical scoping with first-class functions
- Why first class functions *require* different allocation rules
- What is meant by closure, referencing environment, and frame.

Class outcomes

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 dynamic and lexical scoping.
- Draw the state of the Central Reference Table at any point in running a dynamically-scoped program
- Draw the tree of nested scopes for a lexically-scoped program.
- Trace the run of a lexically-scoped program.
- Draw the frames and closures in a program run using lexical or dynamic scoping