Implementing Dynamic Scope

For dynamic scope, we need a stack of bindings for every name.

These are stored in a *Central Reference Table*.
This primarily consists of a mapping from a *name* to a *stack of values*.

The Central Reference Table may also contain a stack of sets, each containing identifiers that are in the current scope.
This tells us which values to pop when we come to an end-of-scope.

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?

Example: Central Reference Tables with Lambdas

The *i* in
`new g := lambda z { write i; };`
from the previous program could be:

- The *i* in scope when the function is actually called.
- The *i* in scope when *g* is passed as *p* to *f*
- The *i* in scope when *g* is defined

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.

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

Implementing Lexical Scope

What's tough about lexical scope?

Many older languages (C/C++, Fortran) avoid this by treating functions as third-class and prohibiting *nested functions*.

Then every name has local scope (to a function or block), or global scope.

The result is *compile-time name resolution* — fast code!

Lexical Scope with Nested Functions

What if we allow just things like this:

```
void f(int x) {
  void g(int y) {
    print(x+y);
  }
  if (x < 5) g(10);
  else f(x-1);
}

int main() { f(6); }
```

We can use *static links* to find bindings in the most recent enclosing function call.

Lexical Scope with 2nd-Class Functions

What if functions have full 2nd-class privileges?

```
(define (f a g)
  (define (h b) (display (+ a b)))
  (if (< a 5)
      (f (g a) h)
      (g a)))

(f 4 add1)
```

Bindings may be further down than most recent call.
We need *dynamic links* into the stack!

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);
}
```

There are some *very* non-local references here!
Where should we store local variables?

Class outcomes

You should know:

- What is meant by shallow/deep binding (roughly)
- Why some language restrict functions to 3rd-class or 2nd-class
- What static links are, and when they can and can't be used
- What non-local references are, and what kind of headaches they create

You should be able to:

- Draw the state of the Central Reference Table at any point in running a dynamically-scoped program
- Trace the run of a lexically-scoped program.