CS 7400 October 5-7, 2009
Dr. Wand Readings: EOPL3, Secs. 3.6–3.8

Lecture 4: Lexical Addressing

Key Concepts:

variable reference, declaration
scope
lexical scoping
lexical depth, lexical address
shadowing
contour diagram
local, non-local, global variables
scope, binding, and extent
static analysis
lexical address analysis
static environment
nameless environment

4.1 Scope, Binding, and Extent

Variables may appear in two different ways: as *references* or as *declarations*.

A variable reference is a use of the variable. For example, in

```
(f x y)
```

all the variables, `f`, `x`, and `y`, appear as references. However, in

```
(lambda (x) ...)
```

or

```
(let ((x ...)) ...)
```

the occurrence of `x` is a declaration: it introduces the variable as a name for some value.

Corresponding to each declaration of a variable `x` is a region of the program in which any use of the variable `x` refers to that declaration of `x`. This is called the *scope* of the declaration. Scoping allows the same name to be reused for different purposes in different parts of a program.

In Scheme, the scope of any declaration is determined by the *lexical scope rule*:

> *In* `(lambda (`x_1 `...` x_n`)` B`)` *or* `(let ((`x_1 E_1`) ... (`x_n E_n`))` B`)` *the scope of* $x_1, \ldots, x_n$ *is the body* B.

This is a *static* rule, meaning that one can compute the scopes without executing the program.

To find which declaration corresponds to a given use of a variable, we search *outwards* from the use until we find a declaration of the variable.

```
(let ((x 3)                Call this x1
      (y 4))               Call this y1
  (+ (let ((x              Call this x2
             (+ y 5)))
       (* x y))            Here x refers to x2
     x))                   Here x refers to x1
```

Lexical depth tells us how far up we have to travel. In `(* x y)`,

`x` is bound at lexical depth 0
`y` is bound at lexical depth 1

Lexical scopes are nested: each scope lies entirely within another scope.

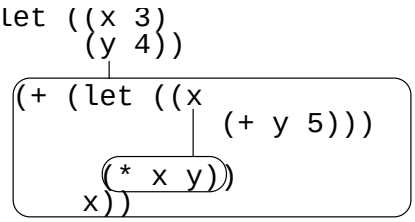
Scoping rules like this are called *lexical scoping* rules, and the variables declared in this way are called *lexical variables*.

Under lexical scoping, we can create a hole in a scope by redeclaring a variable. We say the inner declaration *shadows* the outer one. For instance, in the example above, the inner `x` shadows the outer one in the multiplication `(* x y)`.

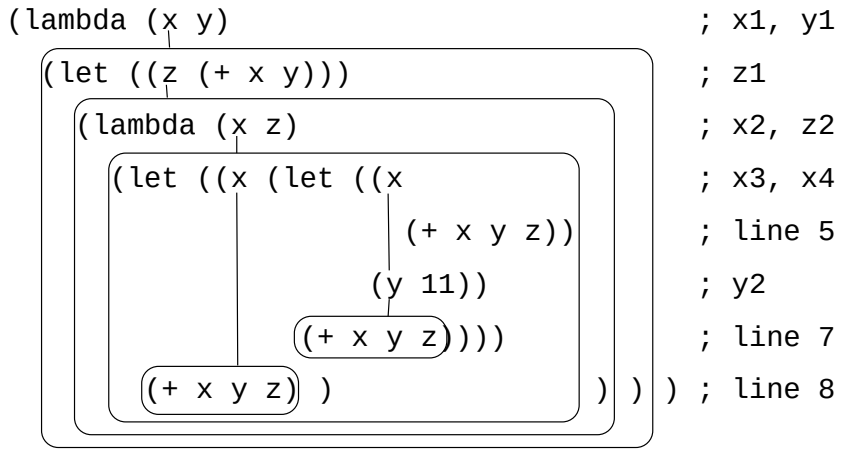
We can illustrate lexical scoping with a *contour diagram*.

Contour diagram for our simple example:

```
(let ((x 3)
      (y 4))
  (+ (let ((x
            (+ y 5)))
       (* x y))
     x))
```



A more complicated example



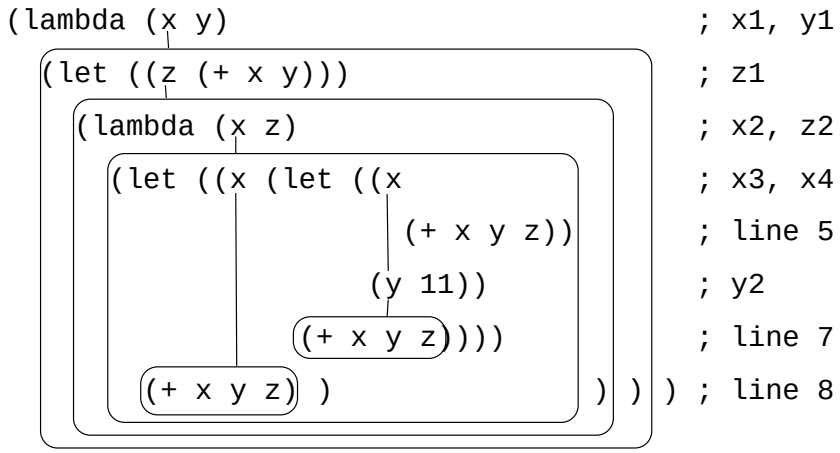
Here there are three occurrences of the expression `(+ x y z)`, on lines 5, 7, and 8

At line 5, we are within the scope of the declarations

`(x z)` at depth 0
`(z)` at depth 1
`(x y)` at depth 2

So at line 5,

`x` is bound at depth 0 and position 0, and refers to `x2`
`y` is bound at depth 2 and position 1, and refers to `y1`
`z` is bound at depth 0 and position 1, and refers to `z2`

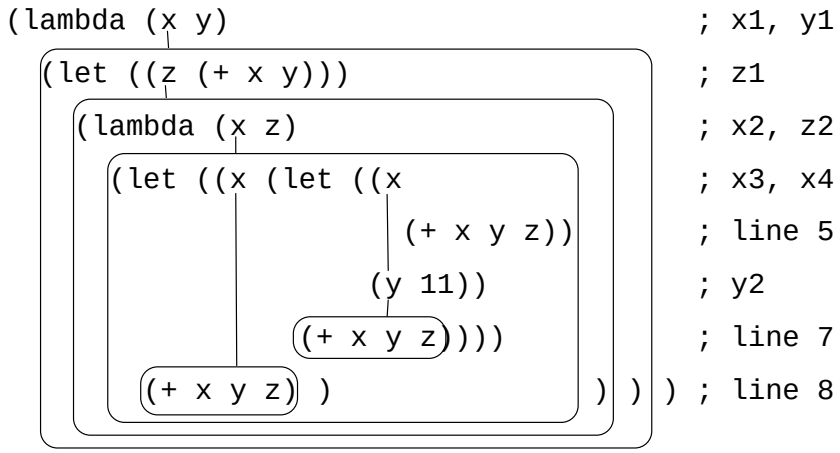


At line 7, we are within the scope of the declarations

`(x y)`	at depth 0
`(x z)`	at depth 1
`(z)`	at depth 2
`(x y)`	at depth 3

So at line 7,

`x`	is bound at depth 0 and position 0, and refers to `x4`
`y`	is bound at depth 0 and position 1, and refers to `y2`
`z`	is bound at depth 1 and position 1, and refers to `z2`



At line 8, we are within the scope of the declarations

`(x)`	at depth 0
`(x z)`	at depth 1
`(z)`	at depth 2
`(x y)`	at depth 3

So at line 8,

`x`	is bound at depth 0 and position 0, and refers to `x3`
`y`	is bound at depth 3 and position 1, and refers to `y1`
`z`	is bound at depth 1 and position 1, and refers to `z2`

The combination of lexical depth and position is called a *lexical address*.

We can classify the variable uses in an expression by where they are bound:

- *Local* variables are those bound by the immediately enclosing binder (`let`, `letrec`, or `lambda`).
- *Non-local* variables are those bound somewhere else in the expression.
- *Free* or *global* variables are those that are not bound in the expression.

For example, in the last line of

```
(let ((x 5) (y 7))
  (let ((a 6) (b 7))
    (lambda (u v)
      (...))))
```

`u` and `v` are local, `a`, `b`, `x`, and `y` are non-local, and any other variables are free (global).

The *binding* of a variable is the value associated with it. Bindings are created by `extend-env`, so you can look at the specification to see how the binding is created. Formal parameters are bound when the procedure is applied:

A variable declared by a `proc` is bound when the procedure is applied.

```
(apply-procedure (procedure x e ρ) v)
 = (value-of e [x=v]ρ)
```

A `let`-variable is bound to the value of its right-hand side.

```
(value-of (let-exp x e1 e2) ρ) =
  (let ((v (value-of e1 ρ)))
    (value-of e2  [x=v]ρ)))
```

A variable declared by a `letrec` is bound using its right-hand side as well.

```
(value-of (letrec-exp p x e1 e2) ρ)
= (value-of e2 (extend-env-recursively p x e1 ρ))
```

The *extent* of a binding is the *time interval* during which the binding is maintained.

In our little language (as in Scheme), all bindings have *semi-infinite* extent, meaning that once a variable gets bound, that binding must be maintained indefinitely (at least potentially). This is because the binding might be hidden inside a closure that is returned.

In languages with semi-infinite extent, the garbage collector collects bindings when they are no longer reachable. This is only determinable at run-time, so we say that this is a *dynamic* property.

If we didn't allow `proc`'s to appear as the body (or value) of a `let`, then the let-bindings would expire at the end of the evaluation of the `let` body. This is called *dynamic* extent, and it is a *static* property. Because the extent is a static property, you can predict exactly when a binding can be discarded. ("Dynamic" sometimes means "during the evaluation of an expression" and sometimes means "not calculable in advance." Sorry– I didn't make this up.)

4.2 Lexical Address Analysis

We've seen how we can use interpreters to model the run-time behavior of programs. We now introduce a new theme to the course: how to use the same technology to *analyze* or *predict* the behavior of programs without running them.

We will do three kinds of predictions:

1. First, we will do a *lexical address analysis*. This analysis will predict, at analysis time, where in the environment each identifier will be found at run time. We will see that the analyzer looks like an interpreter, except that instead of passing around an environment, we pass around a *static environment*, which associates with each identifier whatever we know statically about it.

   As a result of the lexical address analysis, we will be able to translate our programs into a variable-free form, in which every variable reference is replaced by an address in the environment, and we can replace our interpreter by one that uses these addresses instead of identifiers.

2. In Lecture 6 we will do *type analysis*. Most languages divide up their expressed values into *types*. The type of a value determines what operations are appropriate on that value. We will analyze our programs to ensure that no operation is ever performed on inappropriate data.

   This analysis will enable us to reject programs that are "dangerous," that is, those programs that *might* perform an inappropriate operation.

3. Then in lecture 7, we will use the same kind of analysis to establish abstraction boundaries in programs and reject those programs that violate the abstraction boundaries.

4.2.1 Eliminating Variable Names

Execution of the scoping algorithm may then be viewed as a journey outward from a variable reference to its matching declaration

We could, therefore, get rid of variable names entirely, and replace variables with their lexical depths.

For example, we could replace the Scheme expression

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

by something like:

```
(nameless-lambda
  ((nameless-lambda
     (#1 #0))
   #0))
```

Each `nameless-lambda` declares a new anonymous variable, and each variable reference is replaced by its lexical depth; this number uniquely identifies the declaration to which it refers. These numbers are called *lexical addresses* or *deBruijn indices*.

This way of recording the information is useful because the lexical address *predicts* just where in the environment any particular variable will be found.

Consider the expression

```
let x = e1
in let y = e2
   in -(x,y)
```

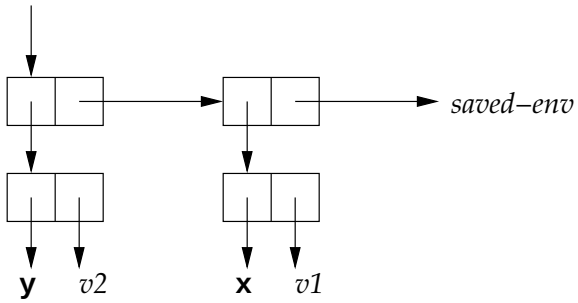
in our language. In the difference expression, the lexical depths of `y` and `x` are 0 and 1, respectively.

Now assume that the values of e_1 and e_2, in the appropriate environments, are v_1 and v_2. Then the value of this expression is

```
(value-of
  <<let x = e1
    in let y = e2
       in -(x,y)>>
  ρ)
=
(value-of
  <<let y = e2
    in -(x,y)>>
    [x=v1]ρ)
=
(value-of
  <<-(x,y)>>
  [y=v2][x=v1]ρ)
```

so that when the difference expression is evaluated, `y` is at depth 0 and `x` is at depth 1, just as predicted by their lexical depths.

If we use a data-structure representation of environments like we did for LET, then the environment will look like



so that the values of `x` and `y` will be found by taking either 1 cdr or 0 cdrs in the environment, regardless of the values v_1 and v_2.

Same thing works for procedure bodies. Consider

```
let a = 5
in proc (x) -(x,a)
```

In the body of the procedure, x is at lexical depth 0 and a is at lexical depth 1.
The value of this expression is

```
(value-of
  <<let a = 5
    in proc (x) -(x,a)>>
  ρ)
=
(value-of
  <<proc (x) -(x,a)>>
  (extend-env a 5 ρ))
=
(procedure x <<-(x,a)>> [a=5]ρ)
```

The body of this procedure can only be evaluated by `apply-procedure`, say

```
(apply-procedure
  (procedure x <<-(x,a)>> [a=5]ρ)
  7)
=
(value-of
  <<-(x,a)>>
  [x=7][a=5]ρ)
```

So again every variable is found in the environment at the place predicted by its lexical depth. If we had multiple arguments, then we'd have to keep track of the position, too, but that's also static.

4.2.2 Implementing Lexical Addressing

We now implement the lexical-address analysis we sketched above. We write a procedure `translation-of-program` that takes a program and removes all the identifiers from the declarations, and replaces every variable reference by its lexical depth.

For example, the program

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

is translated to

```
#(struct:a-program
  #(struct:nameless-let-exp
    #(struct:lit-exp 37)
    #(struct:nameless-proc-exp
      #(struct:nameless-let-exp
        #(struct:diff-exp
           #(struct:nameless-var-exp 0)
           #(struct:nameless-var-exp 1))
        #(struct:diff-exp
           #(struct:nameless-var-exp 2)
           #(struct:nameless-var-exp 1))))))
```

Will then write a new version of `value-of-program` that will find the value of such a nameless program, without putting identifiers in the environment.

Implementation is at `interps/lecture04/lexaddr-lang`

4.2.3 The Translator

We are writing a translator, so we need to know the source language and the target language. The target language will have things like `nameless-var-exp` and `nameless-let-exp` that were not in the source language, and it will lose the things in the source language that these constructs replace, like `var-exp` and `let-exp`.

Add to the SLLGEN grammar:

```
(expression ("%lexref" number) nameless-var-exp)
(expression
  ("%let" expression "in" expression)
  nameless-let-exp)
(expression
  ("%lexproc" expression)
  nameless-proc-exp)
```

We use names starting with `%` for these new constructs because that is normally the comment character in our language.

Our translator will reject any program that has one of these new nameless constructs (`nameless-var-exp`, `nameless-let-exp`, or `nameless-proc-exp`), and our interpreter will reject any program that has one of the old nameful constructs (`var-exp`, `let-exp`, or `proc-exp`) that are supposed to be replaced.

To calculate the lexical address of any variable reference, we need to know the scopes in which it is enclosed. This is a *context* argument.

So `translation-of-expression` will take two arguments: an expression and a *static environment*. The static environment will be a list of identifiers, representing the scopes within which the current expression lies. The variable declared in the innermost scope will be the first element of the list.

For example, when we translate the last line of the example above, the static environment should be

```
(z y x)
```

So looking up an identifier in the static environment means finding its position in the static environment, which gives a lexical address: looking up `x` will give 2, looking up `y` will give 1, and looking up `z` will give 0.

Entering a new scope will mean adding a new element to the static environment. We introduce a procedure `extend-senv` to do this.

[Puzzle: how would this change if we had multiple arguments or multiple declarations?]

Since the static environment is just a list of identifiers, these procedures are easy to implement:

senv = (list-of symbol)
lexaddr = N

empty-senv : () -> senv
```
(define empty-senv
  (lambda ()
    '()))
```

extend-senv : sym * senv -> senv
```
(define extend-senv
  (lambda (x senv)
    (cons x senv)))
```

apply-senv : senv * sym -> lexaddr
```
(define apply-senv
  (lambda (senv x)
    (cond
      ((null? senv)
       (eopl:error 'translation-of
         "unbound variable in code: ~s" x))
      ((eqv? x (car senv))
       0)
      (else
        (+ 1 (apply-senv (cdr senv) x))))))
```

For the translator, we have two procedures, `translation-of`, which handles expressions, and `translation-of-program`, which handles programs.

We are trying to translate an expression `e` which is sitting inside the declarations represented by `senv`. To do this, we recursively copy the tree, except that

1. Every `var-exp` is replaced by a `nameless-var-exp` with the right lexical address, which we compute by calling `apply-senv`.

2. Every `proc-exp` is replaced by a `nameless-proc-exp`, with the body translated with respect to the new scope, represented by the static environment `(extend-senv` x `senv)`.

3. Every `let-exp` is replaced by a `nameless-let-exp`.

   The two subexpressions are each translated in an appropriate scope.

translation-of : expression * static-environment
-> nameless-expression

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp

      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))

      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))

      (let-exp (var rhs body)
        (nameless-let-exp
          (translation-of rhs senv)
          (translation-of body
            (extend-senv var senv))))

      (const-exp (num) (const-exp num))

      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))

      ... other cases of source language are similar

      (else (eopl:error 'translation-of
              "Illegal source expression ~s" exp))
      )))
```

The procedure `translation-of-program` simply runs `translation-of` in a suitable initial static environment.

translation-of-program : `program -> nameless-program`

```
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (e)
        (a-program
          (translation-of e (init-senv)))))))
```

init-senv : `() -> static-environment`

```
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

4.2.4 The nameless interpreter

Our interpreter takes advantage of the predictions of the lexical-address analyzer to avoid explicitly searching for variables at run time.

Since there are no more identifiers in our programs, we won't be able to put identifiers in our environments, but since we know exactly where to look in each environment, we don't need them!

Our top-level procedure will be `run`:

```
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string)))))
```

Instead of having full-fledged environments, we will have nameless environments, with the following interface:

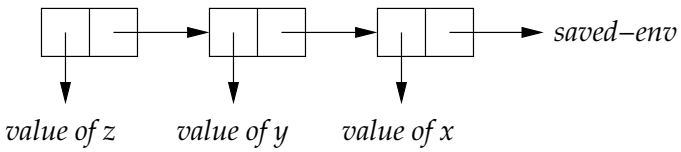
nameless-environment?	: `val -> bool`
empty-nameless-env	: `() -> nameless-env`
empty-nameless-env?	: `nameless-env -> bool`
extend-nameless-env	: `expval * nameless-env -> nameless-env`
apply-nameless-env	: `nameless-env * lexaddr -> expval`

We can implement a nameless environment as a list of expressed values, so that `apply-nameless-env` is simply a call to `list-ref`.

For example, at the last line of our example

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

the nameless environment will look like



nameless-environment? : scheme-value -> bool
```
(define nameless-environment? (list-of expval?))
```

empty-nameless-env : () -> nameless-env
```
(define empty-nameless-env
  (lambda ()
    '()))
```

empty-nameless-env? : nameless-env -> bool
```
(define empty-nameless-env? null?)
```

extend-nameless-env : expval * nameless-env -> nameless-env
```
(define extend-nameless-env cons)
```

apply-nameless-env : nameless-env * lexaddr -> expval
```
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

Having changed the environment interface, we need to look at all the code that depends on that interface. There are only two things in our interpreter that use environments: procedures and `value-of`.

The revised specification for procedures is just the old one with the variable name removed.

(apply-procedure (procedure e ρ) v)
= (value-of e (extend-nameless-env v ρ))

We can implement this by defining

procedure : nameless-expression * nameless-env -> proc
```
(define-datatype proc proc?
  (procedure
    (body expression?)
    (nameless-env nameless-environment?)))
```

apply-procedure : proc * expval -> expval
```
(define apply-procedure
  (lambda (proc1 v)
    (cases proc proc1
      (procedure (e saved-nameless-env)
        (value-of e
          (extend-nameless-env v saved-nameless-env))))))
```

`value-of` will be like before, except

- we omit the cases for `var-exp`, `let-exp`, and `proc-exp`
- replace them with new cases for `nameless-var-exp`, `nameless-let-exp`, and `nameless-proc-exp`

A `nameless-var-exp` gets looked up in the environment.

A `nameless-let-exp` evaluates its right-hand side e_1, and then evalutes its body e_2 in an environment extended by the value of the right-hand side. This is just what an ordinary `let` does, but without the identifiers.

A `nameless-proc-exp` produces a `nameless-proc`, which is then applied by `apply-procedure`.

value-of : nameless-expression * nameless-environment
-> expval

```
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num)    ...as before...)
      (diff-exp (e1 e2)   ...as before...)
      (zero?-exp (e1)     ...as before...)
      (if-exp (e1 e2 e3)  ...as before...)
      (call-exp (e1 e2)   ...as before...)

      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (e1 e2)
        (let ((val (value-of e1 nameless-env)))
          (value-of e2
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (e)
        (proc-val
          (procedure e nameless-env)))

      (else
        (eopl:error 'value-of
          "Illegal expression in translated code: ~s" exp))

      )))
```

Last, here's the new `value-of-program`:

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (e)
        (value-of-expression e (init-nameless-env))))))
```

And we're done.