Lexical Binding

There are two ways a variable can be used in a program:

- As a declaration
- As a "reference" or use of the variable

Scheme has two kinds of variable "declarations" -- the bindings of a let-expression and the parameters of a lambda-expression. The *scope* of a declaration is the portion of the expression or program to which that declaration applies. Like Java and C, but unlike classic Lisp, Scheme uses *lexical binding (*sometimes called *static binding),* which means that the scope of a variable is determined by the textual layout of the program.

Every language has its own scoping rules. For example, what is the scope of variable j in this Java program?

```
public static void main(String[] args) {
        int i;
        i = 1;
        while (i < 10) {
                int j;
                j = i;
                System.out.write(i);
                i += 1;
        }
        System.out.write(j);
}
```

In Scheme it is tempting to say that the scope of a variable declared in the bindings of a let-expression is the body of the expression, but this isn't the case. For example

```
(let ([x 5]) (* ((lambda (x) (+ x 3) ) 7)  x ) )
```

the scope of the [x 5] declaration is only the second operand of the *-expression.

It is more accurate to say that the scope of a variable declared in a let-expression or lambda-expression is the body of that expression *unless that variable also occurs bound in the body*.

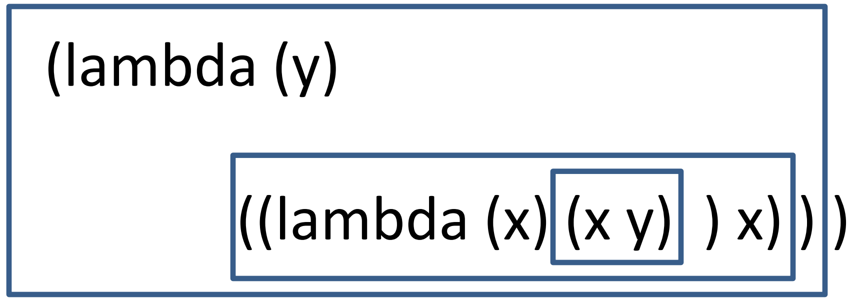
If the variable occurs bound in the body, we say that the inner binding *shadows* the outer binding.

To determine the appropriate binding to which a bound variable refers:

- Start at the reference (usage of the variable).
- Search the enclosing regions starting with the innermost and working outward, looking for a declaration of the variable.
- The first declaration you find is the appropriate binding.
- If you don't find such a binding the variable is free.

Contour diagrams draw the boundaries of the regions in which variable declarations are in effect:

```
(lambda (x)
```



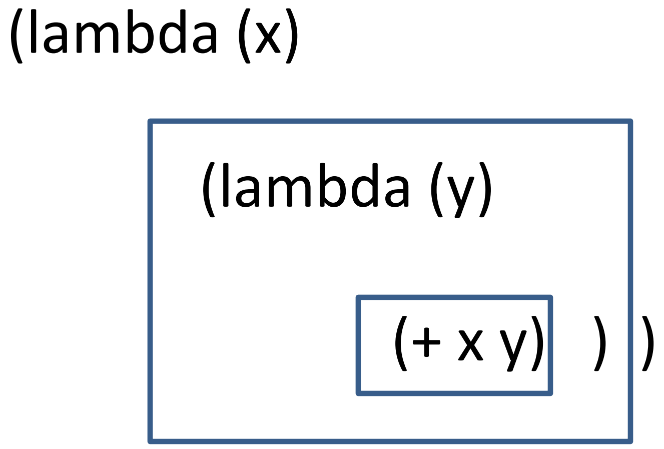
```
      (lambda (y)

            ((lambda (x) (x y) ) x)) )
```

The body of a let or lambda expression determines a contour. Each variable refers to the innermost declaration outside its contour.

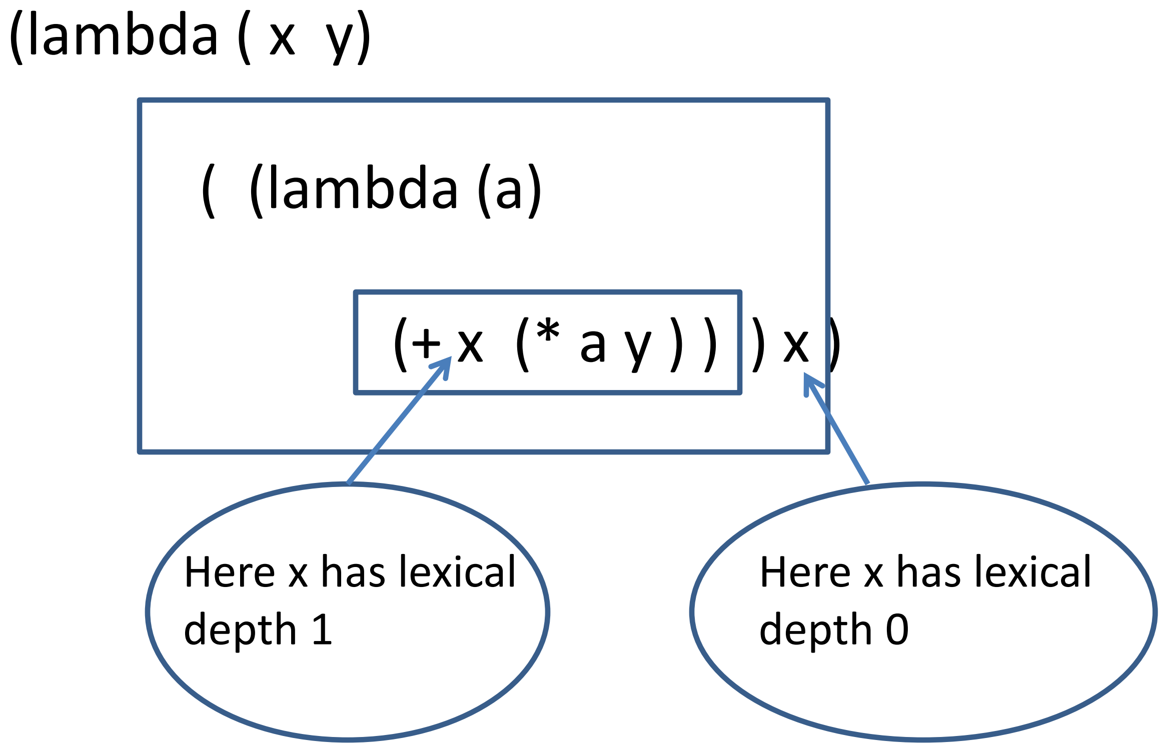
The *lexical depth* of a variable reference is 1 less than the number of contours crossed between the reference and the declaration it refers to.

For example



```
        (lambda (x)
            (lambda (y)
                   (+ x y)   )  )
```

In the (+ x y) portion of this expression x has lexical depth 1, while y has lexical depth 0.



```
(lambda ( x  y)
    (  (lambda (a)
           (+ x  (* a y ) ) ) x )
```

Here x has lexical depth 1

Here x has lexical depth 0

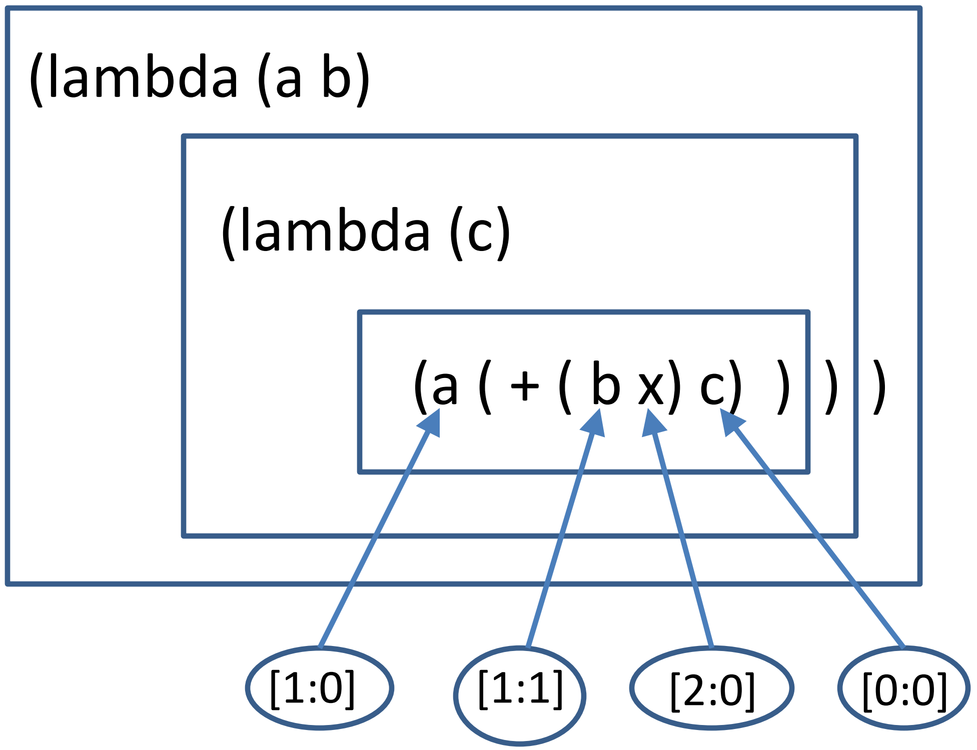
The *lexical address* of a variable reference consist of a pair:

a) The lexical depth of the reference
b) The 0-based position of the variable in its declaration.

We might write this as [depth:position]

For example, consider the expression

(let ([x 3] [y 4])



We could use lexical addresses to completely replace variable names:

```
(let ([3] [4])
          (lambda 2
                    (lambda 1
                              ([1:0] ( + ( [1:1] [2:0]) [0:2) ) ) )
```

The lexical address is essentially a pointer to where the variable can be found on the runtime stack.

Dynamic Binding

What is the value of the following expression?

```
(let ([y 3])
        (let ([f (lambda (x) (+ x y) ) ])
                (let ([y 17])
                        (f 2) ) ) )
```

The real question, of course, is What is the value of y in the body of f when we call (f 2)?

a) Scheme, Java and C use *static binding,* also called *lexical binding.* They connect the reference of y to the nearest surrounding declaration of y, which in this case is [y 3].
b) Early Lisp used *dynamic binding*, which connects a reference of y to the most recent declaration of y, which in this case is [y 17].

We have talked before about how to evaluate lambda expressions and applications (function calls) in Scheme, which is lexically scoped -- a lambda expression evaluates to a closure, which is a pair containing the environment at the time the lambda is evaluated (the surrounding environment) and the parameters and body of the lambda expression. When we apply this closure to argument expressions we evaluate the arguments in the current environment, make a new environment that extends the closure's environment with the new bindings, and evaluate the closure's body within this new environment.

In Java and C, which are also lexically scoped but without lambda expressions, the environments are much more static. At the time a program is compiled the compiler can keep track of the environments and link each variable reference to its declaration and its location on the runtime stack.

Think back to this example:

```
(let ([y 3])
        (let ([f (lambda (x) (+ x y) ) ])
                (let ([y 17])
                        (f 2) ) ) )
```

In dynamic scoping the binding of y that applies in the application (f 2) is [y 17], and the whole expression returns 19.

How would you evaluate lambdas and applications in a dynamically scoped language?

a) There is no need for closures; they maintain the lexical environment, which dynamic binding does not use.
b) The value of a lambda expression is just its parameters and body.
c) To apply a procedure to a list of arguments, we extend the *current* environment with the bindings of the parameters to their argument values and evaluate the body in this environment.

Why do (or did) people use dynamic binding?

- It was easy to implement. Indeed, dynamic binding was understood several years before static binding.
- It made sense to some people; the function (lambda (x) (+ x y) should use whatever the latest version of y.

Why do we now use lexical binding?

- Most languages we use today are derived from Algol-60, which used lexical binding.
- Compilers can use lexical addresses, known at compile time, for all variable references.
- Code from lexically-bound languages is easier to verify.
- It makes more sense to most people.