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 y in this Java program? Could we print y instead of x in the last line?

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

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 exactly 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.

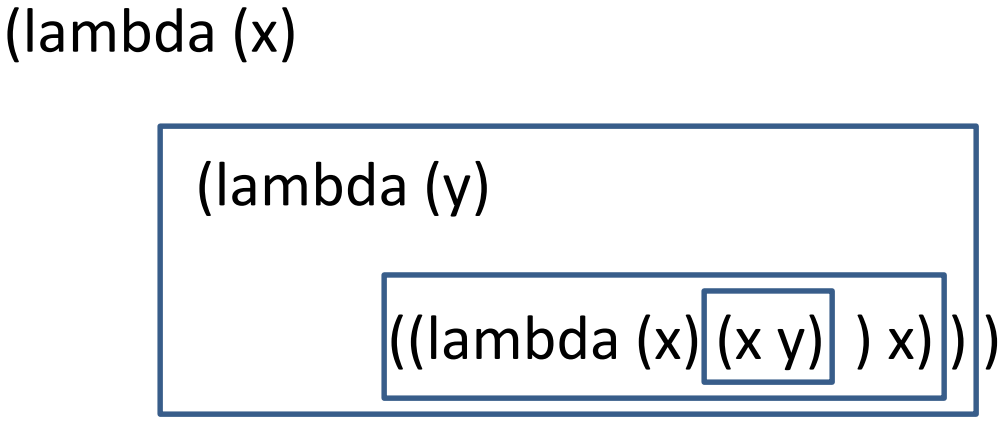
It is more accurate to say that the scope of a variable declared in a let-expression or lambda-expression is the body of that expression *unless that variable also occurs bound in the body*.

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.

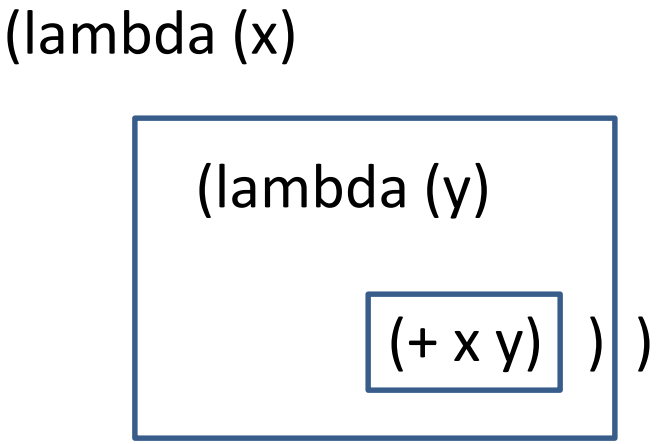
Contour diagrams draw the boundaries of the regions in which variable declarations are in effect:



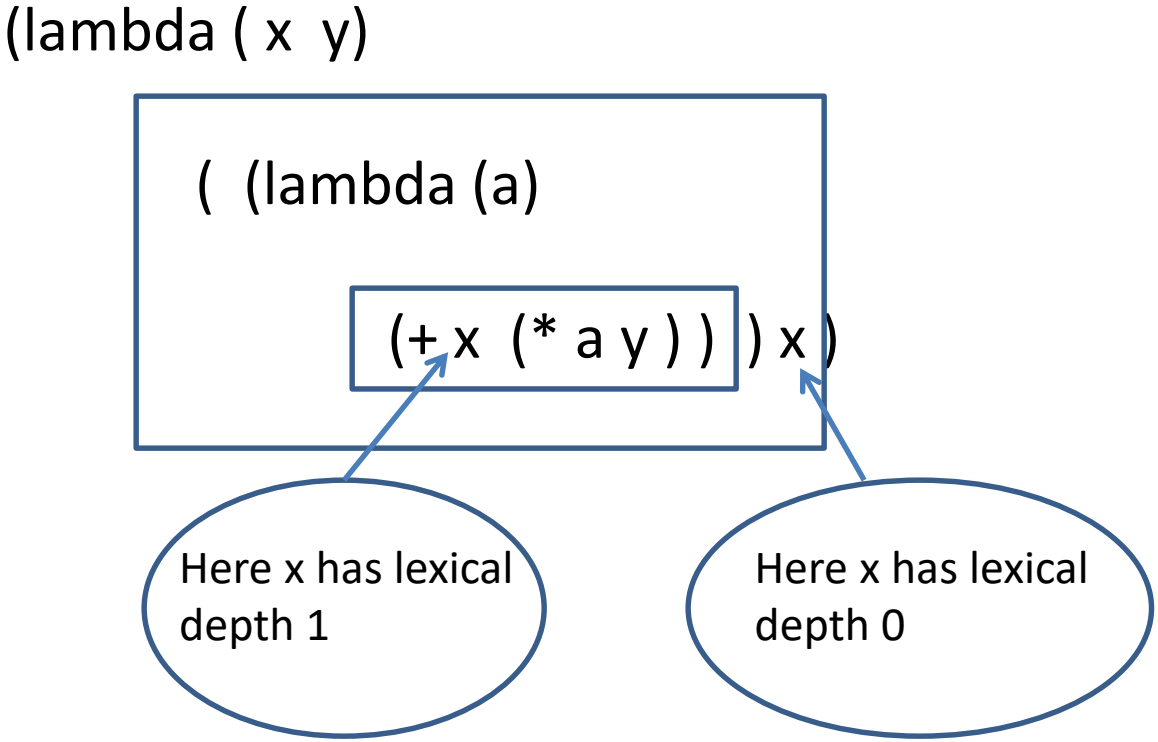
The body of a let or lambda expression determines a contour. Each variable refers to the innermost declaration outside its contour.

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



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

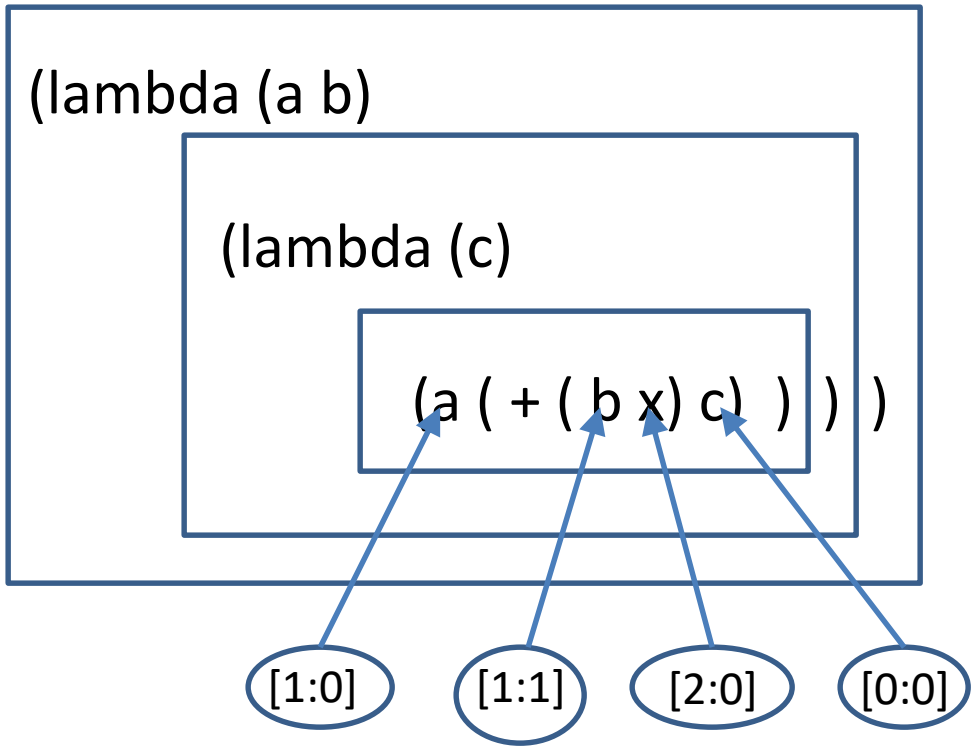
The *lexical address* of a variable reference consist of a pair:

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.