Goals

- Free and bound variables
- Let construct
- Lexical scope

```
(define (f x)
  (cond
    [(> (big-calculation x) 15)
     (g (big-calculation x))]
    [else (h (big-calculation x))]))
```

or

```
(define (f x)
  (let ((y (big-calculation x)))
    (cond
      [(> y 15) (g y)]
      [else (h y)])))
```

Let introduces local bindings

```
(let ((name1 expression1)
  ...
      (nameN expressionN))
  body)
```

Each of the names is bound in body, but none of the names are bound in the expressions.

Some examples:

```
(define (example1 x)
  (let ((a (* x 2))
        (b (* x 3)))
    (let ((c (+ a b)))
      (if (> c 100)
          (+ c a)
          (+ c b)))))
```

(example1 10) = 80
(example1 30) = 10

```
(let((a 5))
  (let ((a 6))
    a))
```

=
6

```
(let ((a 5))
  (+ (let ((a 6))
        (+ a a))
      a))
```

=
17

Rule1

```
(let ((name1 expression1)
       ...
       (nameN expressionN))
  body)
```

=

```
(let ((name2 expression2)
       ...
       (nameN expressionN))
  body1)
```

where body1 is the expression resulting from substituting the *value* of expression1 into body for all the variables of name1.

Rule2
(let () body) = body

Rule1 is broken. Using it:

```
(let ((a 5))
  (+ (let ((a 6))
        (+ a a))
      a))
```

=

```
(let ()
  (+ (let ((a 6))
        (+ 5 5))
      5))
```

```
(let ()
  (+ (let ((a 6))
        (+ 5 5))
       5))
```

=

```
(+ (let ((a 6))
       (+ 5 5))
    5)
```

```
(+ (let ((a 6))
       (+ 5 5))
    5)
```

=

```
(+ (let ()
       (+ 5 5))
    5)
```

```
(+ (let ()
       (+ 5 5))
    5)
```

=

```
(+ (+ 5 5)
    5)
```

```
(+ (+ 5 5)
    5)
```

=

```
(+ 10 5)
```

=
15

It was supposed to be 17
The problem is that the a in the (+ a a) was supposed to be 6, but we replaced it with 5.

Revised rule1:

```
(let ((name1 expression1)
      ...
      (nameN expressionN))
  body)

=

(let ((name2 expression2)
      ...
      (nameN expressionN))
  body1)
```

where body1 is the expression resulting from substituting the *value* of expression1 into body for all the *free* variables of name1.

This means our first step should have been

```
(let ((a 5))
  (+ (let ((a 6))
        (+ a a))
      a))

=

(let ()
  (+ (let ((a 6))
        (+ a a))
      5))
```

In this example the body of the outer let is:

```
(+ (let ((a 6))
      (+ a a))
    a)
```

The **a***s* are called bound. They are bound by the let.
The *a* is called free. It is not bound *in this expression*.
It is bound if we consider the previous program.
But by a different let.

Define also binds variables. (define x 10)
(+ x 5)
=
15

Function arguments are binding too.

```
(define (f x)
   (+ x 10))
(f 10)

=
20
```

```
exp = data
    |  (operator exp exp)
    |  (cond
          [exp exp]
          [else exp])
       (let ((name exp))
          exp)
```

where data is some set of primitive scheme data (strings, numbers, booleand, etc.) and operator is some set of binary operators on that data (+, =, and, string-append, ...)

let* is similar to let.

```
(let((a 5))
  (let ((a 10) (b (+ 5 a)))
    b))

=
10

(let((a 5))
  (let* ((a 10) (b (+ 5 a)))
    b))

=
15
```

In let* nameX is bound in expressionX+1 and up as well as in the body.

letrec is similar to let, except that each name is bound in every expression.

This isn't useful without local function definitions.