Lesson

Racket doesn't need much new machinery
under the hood to make local variables work.

The language already supports
everything we need to get the job done!

$$x = y + z$$

$$x = y + z$$

$$x = x + 5$$

```
x = y + z
```

```
x = x + 5
x = x + 10
```

```
x = y + z
```

```
x = x + 5
x = x + 10
x = x + 100
```

```
x = y + z
```

```
x = x + 5
x = x + 10
x = x + 100
```

```
x += 5
```

```
x = y + z
```

```
x = x + 5
x = x + 10
x = x + 100
```

```
x = x + 1
```

```
x += 5
```

```
x = y + z
```

```
x = x + 5
x = x + 10
x = x + 100
```

```
x = x + 1
x = x + 1
```

```
x += 5
```

```
x = y + z
```

```
x = x + 5
x = x + 10
x = x + 100
```

```
x = x + 1
x = x + 1
x = x + 1
```

```
x += 5
```

```
x = y + z
```

```
x = x + 5
x = x + 10
x = x + 100
```

```
x = x + 1
x = x + 1
x = x + 1
```

```
x += 5
```

```
x++
```

`x += 5`

and

`x++`

are

syntactic abstractions

`of x = y + z`

syntactic abstraction

"syntactic sugar"

When the compiler sees:

```
x += 42
```

... it rewrites it as:

```
x = x + 42
```

before proceeding.

When the compiler sees:

```
count++
```

... it rewrites it as:

```
count = count + 1
```

before proceeding.

In Racket,

cond

is a syntactic abstraction
of an if expression.

In Racket, this:

```
(define (lookup var env)
  ...)
```

is a syntactic abstraction of:

```
(define lookup
  (lambda (var env)
    ...))
```

Earlier, we learned about "currying" a function:

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

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

That means that

functions with > 1 argument

can be a syntactic abstraction of one-argument functions!

Local variables bind a value to a name.

Local variables bind a value to a name.

Function parameters do, too!

We defined a parameter:

```
(define positions-of-helper
  (lambda (s los position)
    ...))
```

and passed the value of the variable when we called the function:

```
(define positions-of
  (lambda (s los)
    (positions-of-helper s los 0)))
```

```
(define get-value
  (lambda (match)
    (if match
        (cdr match)
        (error ...))))

(get-value (assoc var env))
```

With a local variable in Python:

```
def lookup(var, env):
    match = assoc(var, env)
    if match:
        return match[1]
    else:
        raise ValueError(...)
```

With a local variable in Racket:

```
(define lookup
  (lambda (var env)
    (let ((match (assoc var env)))
      (if match
          (cdr match)
          (error ...)))))
```

```
def lookup(var, env):                    # Python
    match = assoc(var, env)
    if match:
        return match[1]
    else:
        raise ValueError(...)

(define lookup                           # Racket
  (lambda (var env)
    (let ((match (assoc var env)))
      (if match
          (cdr match)
          (error ...)))))
```

The syntax of Racket's let expression:

```
<let-expression> ::= (let <binding-list> <body>)

  <binding-list> ::= ()
                   | (<binding> . <binding-list>)

       <binding> ::= (<var> <exp>)

          <body> ::= <exp>
```

Lesson

Racket doesn't need much new machinery
under the hood to make local variables work.

The language already supports
everything we need to get the job done!