State

This is taken from Structure and Interpretation, Section 3.1

We say that an object has *state* if its behavior can change to reflect its, or its system's, history.

The function (define f (lambda (x y) (+ x y)))
does not have state. No matter what happens around it,
(f 2 3) will always be 5.

Here is a function with state:

```
(define simple (lambda (initial)
                 (let ([value initial])
                   (lambda (x)
                     (begin
                       (set! value (+ value x))
                       value)))))

(define A (simple 100))
```

If we evaluate

(A 10)

it returns 110.

However, if we evaluate

(A 10)

again, it returns 120.

A's behavior changes to reflect the history.

Most Scheme functions do not create objects with state; only set! does this.

So what?

We both gain and lose when we start working with state.

Here are some properties of functions that don't have state:

- They always return the same values for the same arguments.
- They can be evaluated by simple substitution (remember that let-expressions are just applications of lambdas). Environments are not needed to explain stateless functions.
- Two stateless functions can be regarded as equal if they return the same values for the same arguments.
- No stateless function depends on any other function unless it calls that function. Stateless functions are independent of each other and may be evaluated individually.
- All in all, stateless functions are easy to understand and easy to write correctly.

For example

```
(define f
        (let ([value 100])
                (lambda (x)
                        (+ value x))))
```

We can evaluate (f 10) by replacing x by 10 and value by 100 in the expression
(+ value x)

Stateless functions can always be evaluated by substitutions like this.

All of this goes out the window when we introduce state.

Consider the following example:

```
(define F (let ([value 100])
      (lambda (i)
            (cond
                  [(= i 0) (lambda (x)
                                    (+ value x))]
                  [(= i 1) (lambda (x)
                                    (begin (set! value (+ value x))
                                          value))]))))
```

If we then define f as

(define f (F 0))

f seems to be the same as (lambda (x) (+ x 100)

(f 10) returns 110 no matter how many times we evaluate it.

However, if we then define g as

(define g (F 1))

and evaluate (g 10)

this changes the behavior of f. Now

(f 10) evaluates to 120.

To evaluate one function with state, we need to take into consideration all functions that might share that state.

Pure functional program is stateless. Some people find that a more natural way to program.

Functions with state also have some advantages, that might compensate for their added complexity:

- The world has state. Objects in the world change over time. To the extent that we want to model this, state is useful. This is why object-oriented programming is so popular.
- Without state there is no random, or even pseudo-random, behavior; everything is predictable.

- If we try to model dynamic things with stateless functions, the model become very complex. For example, we could represent a bank account by three functions: deposit, withdraw and balance. Each time we made a deposit or withdrawal we would need to install new functions for these activities. We change the functions instead of changing their state, because that is all stateless functions can do. On the other hand, if the three functions share and mutate an environment we get a much more self-contained representation of the bank account. To some extent functions with state are complex because the world is complex and they provide a better way to model this complexity than state-free functions.