State Variables and the Environment Model

The substitution model has taught us that "equals can always be substituted for equals" -- also refereed to sometimes as referential transparency.

**EXCEPTION 1.** We used a procedure `random` to generate random numbers, where `(random n)` evaluated to a random integer between 0 and `n-1`.

```
(define (law-of-identity proc arg)
  (= (proc arg) (proc arg)))
;Value: law-of-identity

(law-of-identity square 5)
;Value: #t

(law-of-identity random 2)
???
```

**EXCEPTION 2.** (Hypothetical) What if we want to implement something with a fixed "remembered state" that changes over time?

```
(define A (make-bank-account 100))
;Value: A

(A 'balance)
;Value: 100

((A 'deposit) 20)
;Value: 120

((A 'deposit) 20)
;Value: 140

((A 'withdraw) 40000)
;Value: (insufficient funds)

((A 'withdraw) 50)
;Value: 90
```

**EXCEPTION 2.5** Quotation:

```
(quote (fact 5))  //=? (quote 120)
```

The Environment Model: A New Genesis

In the beginning God created a GLOBAL ENVIRONMENT: in it He put Primitive Procedures, and Names with which to refer to them.

When you define things, you get to associate names with them too, i.e.,

```
(define pi 3.14159)
```

Since expressions evaluate to something relative to an environment, if evaluation is taking place in the global environment, that is where name `pi` is put.

**Lambda Expressions Evaluate to Procedures**

Modern science, interdisciplinary research, and electron microscopy lets us look at the structure of a procedure: a procedure packages together

a pointer to an environment
a list of formal parameters
a procedure body.

For example, defining `square` at the top level,

```
the pointer is to the global environment
the parameter list is `(x)`
the procedure body is `(\+ \* x x)`
```

Now, to evaluate `(square (+ 5 5))`, the elements of the list are evaluated separately in the global environment:
square evaluates to a procedure
(+ 5 5) evaluates to 10.

The procedure is then applied to the arguments: a new environment frame is created where x is bound to 10, and the body of the procedure is evaluated in the new environment.

Duality: eval and apply

Bank accounts:

(define (make-bank-account balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        '(insufficient funds)))))

(define A (make-bank-account 100))
(define B (make-bank-account 500))

(B 300)

Two new things are going on here:

The use of set! - which rebinds a name-value pair in the environment

The use of (begin e1 e2 ... en) -- which evaluates e1 through en in sequence, and evaluates to whatever en evaluated to. Then why the other arguments? They may cause side-effects (printing, set!, etc.).

Objects and State

We can now define procedures that create objects, and use Scheme in an idiom that is that of object-oriented programming. An object has an internal state. We communicate with the object by sending it messages. In response, the object returns values, and may mutate its internal state.

(define (make-bank-account balance)
  (lambda (m)
    (cond ((eq? m 'balance) balance)
          ((eq? m 'deposit) (lambda (amount)
                                  (begin
                                    (set! balance (+ balance amount))
                                    balance))
          ((eq? m 'withdraw) (lambda (amount)
                                   (begin
                                     (set! balance (- balance amount))
                                     balance))
          (else '(bad message)))))
  ;Value: make-bank-account

(define A (make-bank-account 100))
;Value: A

(A 'balance)
;Value: 100

((A 'deposit) 50)
;Value: 150

((A 'withdraw) 20)
;Value: 130