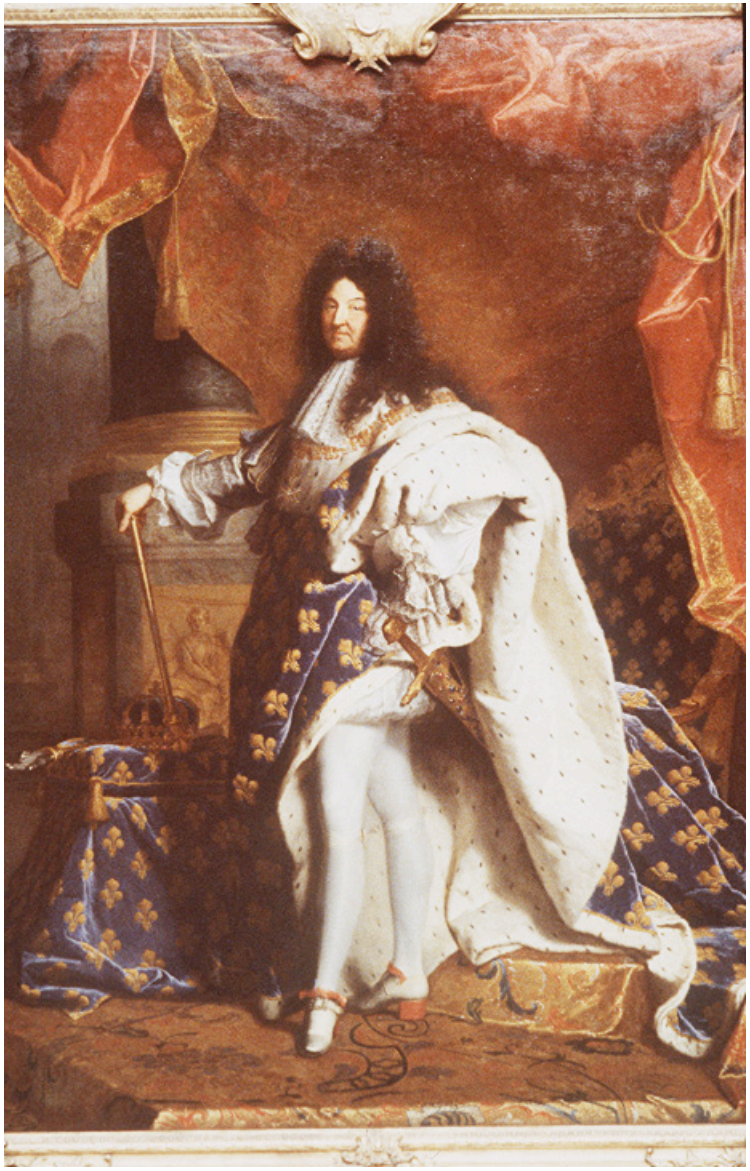
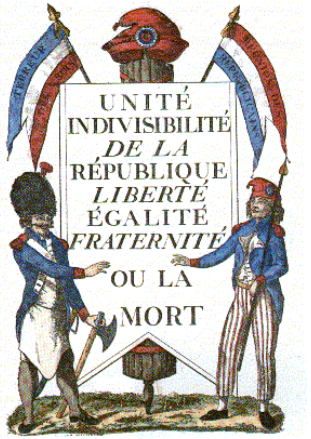


Louis XIV

L'Etat, c'est moi: State and the Environment Model

CS21b: Structure and Interpretation of Computer Programs
Spring Term, 2013



What's Wrong with the Substitution Model?

The substitution model has taught us that "equals can always be substituted for equals" -- also referred 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)
???
```

What's Wrong with the Substitution Model?

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)
```

(see Bertrand Russell, "On Denoting" [1905])

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.,

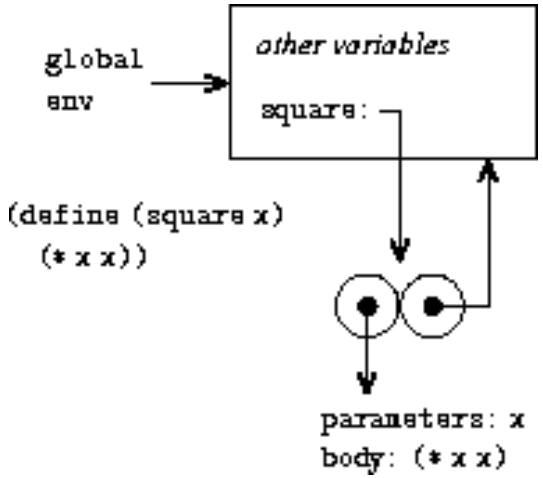
```
(define pi 3.14159)
```

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-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "insufficient funds")))

(define W1 (make-withdraw 100))
(define W2 (make-withdraw 500))

(W1 300)
```

Two new things are going on here:

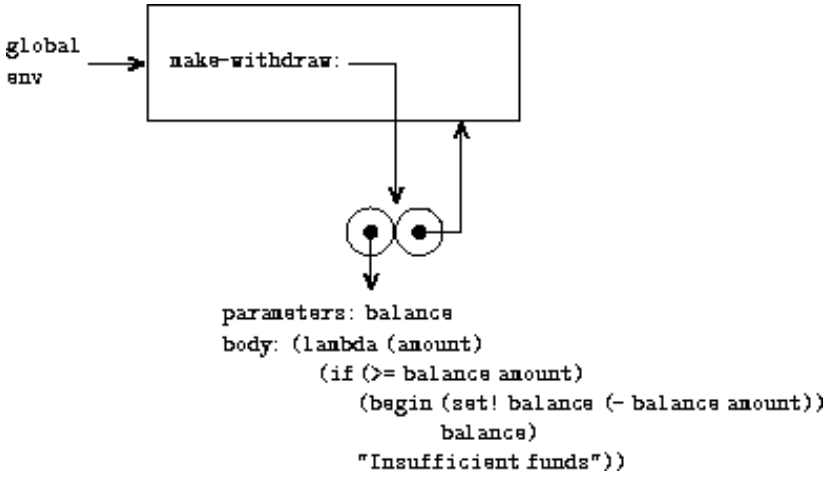
The use of **`set!`** -- which rebinds a *name-value* pair in the environment

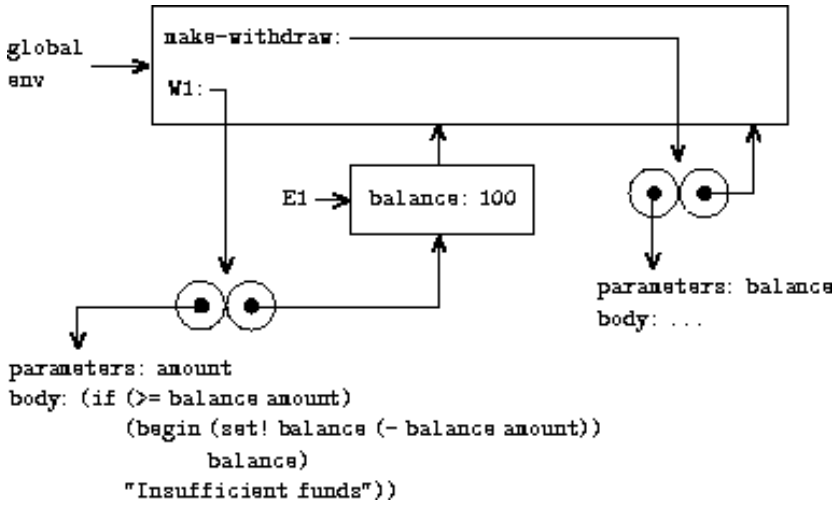
The use of **`(begin`** ***`e1 e2 ... en`***`)` -- which evaluates ***`e1`*** through ***`en`*** in sequence, and evaluates to whatever ***`en`*** evaluated to. Then why the other arguments? They may cause side-effects (printing, **`set!`**, etc.).

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "insufficient funds")))

(define W1 (make-withdraw 100))
(define W2 (make-withdraw 500))

(W2 300)
```





Review of the environment model

Every **expression** is evaluated in the context of an **environment**,
which supplies **values** to the **free variables (names)** in the expression.

A ***frame*** is a ***list*** of bindings of names to values, with a ***pointer*** to another frame.

The ***global environment*** is a frame where the pointer is set to '().

An ***environment*** is a ***list*** of frames (where the list structure is given by the chain of pointers).

** *To evaluate a name:* look it up in the environment (frame by frame).

** *To evaluate a combination* `(e_0 e_1 ... e_n)`: evaluate each of its parts (the `e_i`), producing values `v_i`. If the function `v_0` is built-in, just do it. Otherwise, `v_0` is a user-defined procedure, so (1) make a frame and attach it where the procedure says, (2) bind the formal parameters of the procedure to the `v_i` and put the bindings in the frame, (3) evaluate the body of the procedure in the environment beginning with that frame.

** *To evaluate a lambda-expression:* build a procedure, with formal parameters and body, and attach its pointer to the current environment.

** *To evaluate a definition:* put the name "where you are," evaluate the expression, connect the pointer.

** *To evaluate* `set!`: same thing, only evaluate "where you are," and reset the pointer by "finding" the variable (which is in the current frame, or one linked to it).

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
```

Monitoring the use of a function

```
(define (make-monitored fun)
  (let ((count 0))
    (lambda (x)
       (if (eq? x 'how-many-calls?)
           count
           (begin (set! count (1+ count))
                  (fun x))))))
;Value: make-monitored

(define mmsquare (make-monitored square))
;Value: mmsquare

(mmsquare 5)
;Value: 25

(mmsquare 7)
;Value: 49

(mmsquare 'how-many-calls?)
;Value: 2
```

Memoized functions

```
(define (assoc x lst)
  (if (null? lst)
      '()
      (if (eq? x (caar lst))
          (cadar lst)
          (assoc x (cdr lst)))))
;Value: assoc

(assoc 1 '((2 3) (4 5) (1 2) (8 9) (1 5) (2 8)))
;Value: 2

(define (memoize f)
  (let ((table '()))
    (lambda (x)
      (let ((check-table (assoc x table)))
        (if (null? check-table)
            (let ((new (f x)))
              (newline)
              (write (list 'memoizing x))
              (set! table (cons (list x new) table))
              new)
            (begin (newline)
                   (write (list 'found x))
                   check-table))))))
;Value: memoize
```

```
(define (fib n)
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
;Value: fib

(define mfib (memoize fib))
;Value: mfib

(mfib 4)
(memoizing 4)
;Value: 3

(mfib 5)
(memoizing 5)
;Value: 5

(mfib 5)
(found 5)
;Value: 5

(mfib 10)
(memoizing 10)
;Value: 55
```

What's going wrong?

```
(define (fibfun f)
  (lambda (n) (if (< n 2) n (+ (f (- n 1)) (f (- n 2))))))
;Value: fibfun
```

Or equivalently...

```
(define fibfun (lambda (f) (lambda (n) (if (< n 2) n (+ (f (- n 1)) (f (- n 2)))))))
;Value: fibfun

(define (memoize fun)
  (let ((table '()))
    (define (f x)
      (let ((check-table (assoc x table)))
        (if (null? check-table)
            (let ((new ((fun f) x)))
              (set! table (cons (list x new) table))
              new)
            check-table)))
    f))
;Value: memoize

(define mfib (memoize fibfun))
;Value: mfib
```

Notice that

```
(fun f)= (fibfun f)
       = (lambda (n) (if (< n 2) n (+ (f (- n 1)) (f (- n 2)))))

(mfib 1000)
;Value:
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

A trace, with a hint

```
(+ (begin (newline) (write 'first) 5)
   (begin (newline) (write 'second) 7))
second
first
;Value: 12
```

```
(define (memoize fun)
  (let ((table '()))
    (define (f x)
      (newline) (write (list 'f x))
      (let ((check-table (assoc x table)))
        (if (null? check-table)
            (let ((new ((fun f) x)))
              (newline) (write (list 'memoizing x))
              (set! table (cons (list x new) table))
              new)
            (begin (newline)
                   (write (list 'found (list 'f x)))
                   check-table))))
    f))
;Value: memoize

(define (fibfun f)
  (lambda (n) (if (< n 2) n (+ (f (- n 1))
                               (f (- n 2))))))
;Value: fibfun

(define mfib (memoize fibfun))
;Value: mfib
```

```
(mfib 10)
(f 10)
(f 8)
(f 6)
(f 4)
(f 2)
(f 0)
(memoizing 0)
(f 1)
(memoizing 1)
(memoizing 2)
(f 3)
(f 1)
(found (f 1))
(f 2)
(found (f 2))
(memoizing 3)
(memoizing 4)
(f 5)
(f 3)
(found (f 3))
(f 4)
(found (f 4))
(memoizing 5)
(memoizing 6)
(f 7)
(f 5)
(found (f 5))
(f 6)
(found (f 6))
(memoizing 7)
(memoizing 8)
(f 9)
(f 7)
(found (f 7))
(f 8)
(found (f 8))
(memoizing 9)
(memoizing 10)
;Value: 55
```

Environments and Naming

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
;Value: sqrt

(define (sqrt x)
  (define (sqrt-iter guess)
    (define (good-enough?)
      (< (abs (- (square guess) x)) 0.001))
    (define (improve) (average guess (/ x guess)))
    (if (good-enough?)
        guess
        (sqrt-iter (improve))))
  (sqrt-iter 1.0))
;Value: sqrt
```

How do these work in the environment model?

Mutable list-structures

`set-car!  set-cdr!`

These functions allow us to build destructive list operations and circular data structures. (Convince yourself that you cannot build a circular list structure without them!)

```
(define (last-pair L)
  (if (null? (cdr L))
      L
      (last-pair (cdr L))))
;Value: last-pair

(define (append! L1 L2)
  (if (null? L1)
      L2
      (set-cdr! (last-pair L1) L2)))
;Value: append!

(define first '(a b c d))
;Value: first

(define second '(p q r s t))
;Value: second

(append first second)
;Value: (a b c d p q r s t)

first
;Value: (a b c d)

(append! first second)
;No value

first
;Value: (a b c d p q r s t)
```

Evaluation Order: Applicative versus Normal Order

When evaluating a combination **`(f x y z w),`** in what order does stuff get evaluated? Different computations evolve as a consequence---here's an old, but suggestive example:

```
(define (new-if p x y) (if p x y))
;Value: new-if

(define (loop) (loop))
;Value: loop

(new-if #t 0 (loop))
??
```

In *normal order evaluation,* we delay as much as possible the evaluation of subexpressions---we are forced to evaluate to **print** (in the read-eval-print loop), to **evaluate predicates** in a conditional, and to **evaluate the function** in a combination....

```
(define x 0)
;Value: x

(define (id n)
  (set! x n)
  n)
;Value: id

(define (inc a) (1+ a))
;Value: inc

(define y (inc (id 3)))
;Value: y
```

Applicative order: *evaluate definitions (up to lambda), and in a procedure call, evaluate the arguments*

```
x
;Value: 3

y
;Value: 4

x
;Value: 3
```

Normal order: *don't compute anything until you have to and procrastinate as much as possible, except for testing, printing, and evaluating functions. Observe that* `(define x 0)` *causes* `x` *to be bound to a "thunk" (blob), in this case just* `0`*, that remains unevaluated...*

```
x
;Value: 0

y
;Value: 4

x
;Value: 3
```

[There is no similar difference when we do not have state in the language, i.e., it is "purely functional"...]