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 \((e_0 \ e_1 \ldots \ e_n)\): 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...

** To evaluate set! ...

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)
```
(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)))
              (set! table (cons (list x new) table))
              new)
            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

(trace assoc)
;No value

(mfib 0)
[Entering #[compound-procedure 1 assoc]
  Args: 0
    ()]
[()
   <= #[compound-procedure 1 assoc]
  Args: 0
    ()]
;Value: 0
(mfib 1)
[Entering #[compound-procedure 1 assoc]
 Args: 1
 ((0 0))]
[Entering #[compound-procedure 1 assoc]
 Args: 1
 ()]
[()]
  <= #[compound-procedure 1 assoc]
 Args: 1
 ((0 0))]
;Value: 1

(mfib 2)
[Entering #[compound-procedure 1 assoc]
 Args: 2
 ((1 1) (0 0))]
[Entering #[compound-procedure 1 assoc]
 Args: 2
 ((0 0))]
[Entering #[compound-procedure 1 assoc]
 Args: 2
 ((0 0))]
[Entering #[compound-procedure 1 assoc]
 Args: 2
 ()]
[()]
  <= #[compound-procedure 1 assoc]
 Args: 2
 ()]
[()]
  <= #[compound-procedure 1 assoc]
 Args: 2
 ((0 0))]
[()]
  <= #[compound-procedure 1 assoc]
 ;Value: 1

(mfib 2)
[Entering #[compound-procedure 1 assoc]
 Args: 2
 ((3 2) (2 1) (1 1) (0 0))]
[Entering #[compound-procedure 1 assoc]
 Args: 2
 ((2 1) (1 1) (0 0))]
[1]
  <= #[compound-procedure 1 assoc]
Question: what is getting memoized? Notice that in recursively defined procedures, the recursive call is not to the memoized version...

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

(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 m fib (memoize fibfun))
;Value: m fib

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

(m fib 1000)
;Value:
43466557686937456435688527675040625802564660517371780402481729089536555417
949051890403879892007925169295225930803226347752096896232397332247116164
2996440906533187938298969649928516003704476137795166849228875

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)

Observe that expressions evaluate to something, but don't change the world. Commands change the world, but don't (necessarily) evaluate to something.

(define (list->cycle L)
  (if (null? L)
      L
      (set-cdr! (last-pair L) L)))
;Value: list->cycle

(list->cycle second)
;No value

(car ((repeat 400 cdr) second))
;Value: p

Evaluation Order: Applicative versus Normal Order

(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
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...

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