6.001 recitation 20    5/2/07

- lazy eval
- streams

Dr. Kimberle Koile
extending our evaluator: lazy evaluation

Key ideas:

- procedure args are not evaluated until needed
- represent delayed args as objects
- lazy eval can be added easily
- add new syntax
example a: applicative order

(define (foo x)
    (display 'foo)
    (+ x x))

(define (bar x)
    (display 'arg)
    (display x)
    x)

(foo (bar 2))

What is printed out? (via display and as a final return value)

a. applicative order:

    (foo (bar 2))  output
example b: normal (lazy) order

(define (foo x)
    (display 'foo)
    (+ x x))

(define (bar x)
    (display 'arg)
    (display x)
    x)

(foo (bar 2))

What is printed out? (via display and as a final return value)

a. *normal order (foo's parameter x is delayed):*

   (foo (bar 2)) output

printout
(define (foo x)
  (define (bar x)
    (display 'foo)
    (display 'arg)
    (+ x x))
  (display x)
  x)

(foo (bar 2))

What is printed out?  (via display and as a final return value)

c. normal order with memoization (foo's parameter is delayed and stored)
   
   (foo (bar 2))  output
   
   printout
problem 1a: applicative order

1. (define y 5)
   (define (foo x)
     (display 'foo)
     (+ x x))

   (define (baz x)
     (display 'arg)
     (set! y (+ y x))
     (display y)
     y)

   (foo (baz 2))

What is printed out? (via display and as a final return value)

a. applicative order
problem 1b: normal (lazy) order

1. (define y 5)
   (define (foo x)
     (display 'foo)
     (+ x x))
   (define (baz x)
     (display 'arg)
     (set! y (+ y x))
     (display y)
     y)

(foo (baz 2))

What is printed out? (via display and as a final return value)

b. normal order (foo's parameter x)
problem 1c: normal order with memoization

1. (define y 5)
   (define (foo x)
     (display 'foo)
     (+ x x))

   (define (baz x)
     (display 'arg)
     (set! y (+ y x))
     (display y)
     y)

   (foo (baz 2))

What is printed out? (via display and as a final return value)

c. normal order with memoization (foo's parameter x)
2. (define (initialized-list f n)
   (define (helper n lst)
     (if (= n 0) lst
       (helper (- n 1) (cons (f n) lst))))
   (helper n '()))

; example output:
(initialized-list (lambda(x) (* x x)) 5)
; value (1 4 9 16 25)

What is the value of the statement
  (initialized-list (accum) 5)

a. applicative order
2.  \( \text{define (initialized-list f n)} \)
    \( \text{(define (helper n lst)} \)
    \( \text{(if (= n 0) lst} \)
    \( \text{\quad (helper (- n 1) (cons (f n) lst)))))} \)
    \( \text{(helper n '())} \)

; example output:
(initialized-list (lambda(x) (* x x)) 5)
; value (1 4 9 16 25)

What is the value of the statement     (initialized-list (accum) 5)

b. normal order (initialized-list's parameter f)
representing delayed objects: thunks
thunks: delay-it, force-it (without memoization)

(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
            (actual-value (thunk-exp obj)
                          (thunk-env obj)))
        (else obj)))

(define (actual-value exp env)
  (force-it (l-eval exp env)))
thunks: memoizing implementation

(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
    (let ((result (actual-value (thunk-exp obj)
                                (thunk-env obj))))
      (set-car! obj 'evaluated-thunk)
      (set-car! (cdr obj) result)
      (set-cdr! (cdr obj) '())
      result))
    ((evaluated-thunk? obj) (thunk-value obj))
    (else obj)))
controlling argument evaluation: new syntax

\[
(\text{lambda} \ (a \ (b \ \text{lazy}) \ (c \ \text{lazy-memo})))
\]

- **eval before proc appli**
- **delayed; re-evaluated each time needed**
- **thunk**
- **delayed; evaluated first time needed, value saved**

\text{thunk-memo}