- 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 called thunks
  \[=\text{promise to eval expr later}\]
- Lazy eval can be added easily by mediating
  - proc applic to delay arg eval
  - expr eval by forcing eval only when needed
- Add new syntax so new evaluator, l-eval, can care args
  that are delayed or not
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

1. eval: (+ x x)  ; arg
   2
2. apply: (2 2)  ; foo
3. apply+: 4
```

printout

```
arg 2 foo 4
```
(define (foo x)
  (display 'foo)
  (+ x x))

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

(foo (bar 2))

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

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

1. eval \((\lambda x \text{foo}) \ (\text{bar} \ 2))\)

2. apply foo

3. apply +

4. printout: foo arg 2 arg 2 4
(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)

printout

```
foo arg 2 4
```
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

(output)
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)

printout

```
foo  arg  7  14
```
2. (define (initialized-list f n)
    (define (helper n lst)
      (if (= n 0) lst
       (helper (- n 1) (cons (f n) lst))))
    (helper n '())
    (define (accum)
      (let ((count 0))
        (lambda (x)
          (set! count (+ x count))
          count)))

; 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

1. helper 5 (f n)
   set! count (+ x count) 0 5
   lst (5)

2. helper 4
   (+ x count) 9
   (9 5)

3. helper 3
   (+ x count) 12
   (12 9 5)

4. helper 2
   (+ x count) 14
   (14 12 9 5)

5. helper 1
   (+ x count) 15
   (15 14 12 9 5)

printout

(15 14 12 9 5)
problem 2b: normal (lazy) order

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)

b. normal order (initialized-list's parameter f)

1. helper 5
2. helper 4
3. helper 3
4. helper 2
5. helper 1

(printout (1 2 3 4 5))
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)

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

first call to accum
printout a procedure (call it f) with local variable count; later calls to accum, print f + increment count

printout
(1 3 6 10 15)

0+1 0+1+2 0+1+2+3 ...
representing delayed objects: thunks

\[ \text{'thunk} \rightarrow \text{expr.} \rightarrow \text{env} \]

\[ \text{'evaluated} \rightarrow \text{<result>} \rightarrow \text{thunk} \]
(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 applie

delayed; re-evaluated each time needed

\( \text{thunk} \)

delayed; evaluated first time needed, value saved

\( \text{thunk-memo} \)

eg.

\( (\text{define} \ (\text{initialized-list} \ (f \text{ lazy}) \ n)) \)

\( ... \)