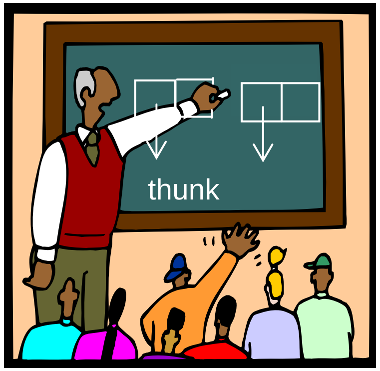
6.001 recitation 20 5/2/07

- 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 called thunks
  = promises to eval expr later
- lazy eval can be added easily by modifying
  - proc applic to delay arg eval
  - expr eval by forcing eval only when needed
- add new syntax so new evaluator, l-eval, can have 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: (<proc> 2) arg 2

2. apply foo: (+ 2 2) foo

3. apply +: 4 4

printout arg 2 foo 4

example b: normal (lazy) order

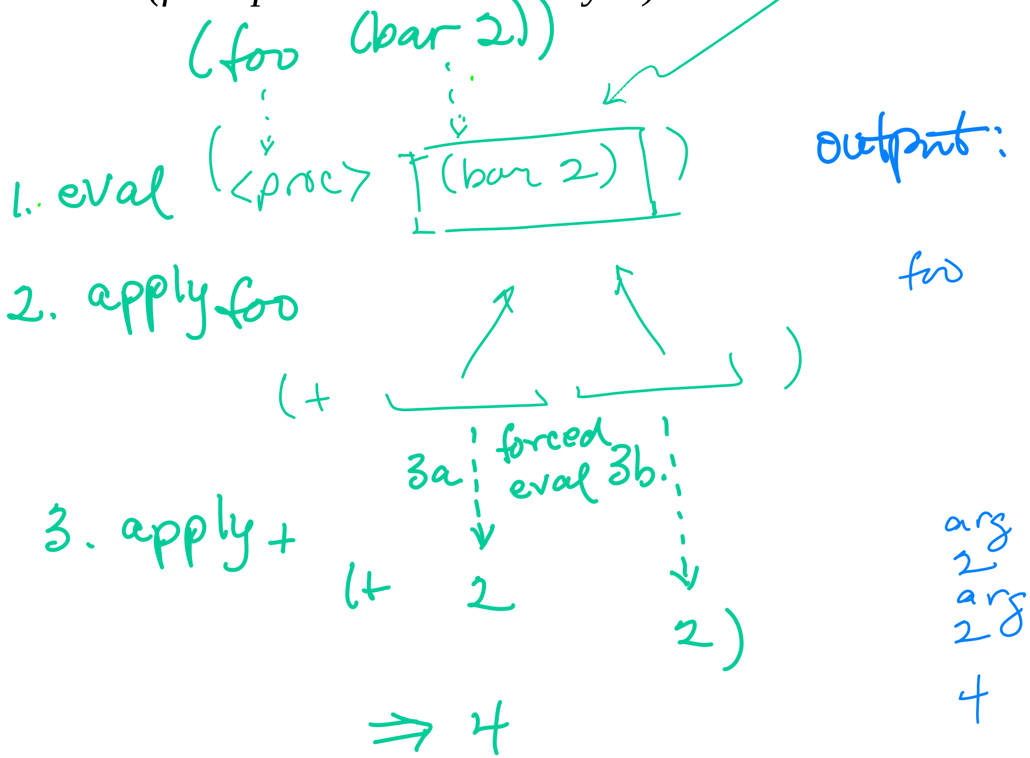
```
(define (foo x)
      (display 'foo)
      (+ x x))
```

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

```
(foo (bar 2))
```

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

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



printout

foo	arg	2	arg	2	4

example c: normal with memoization

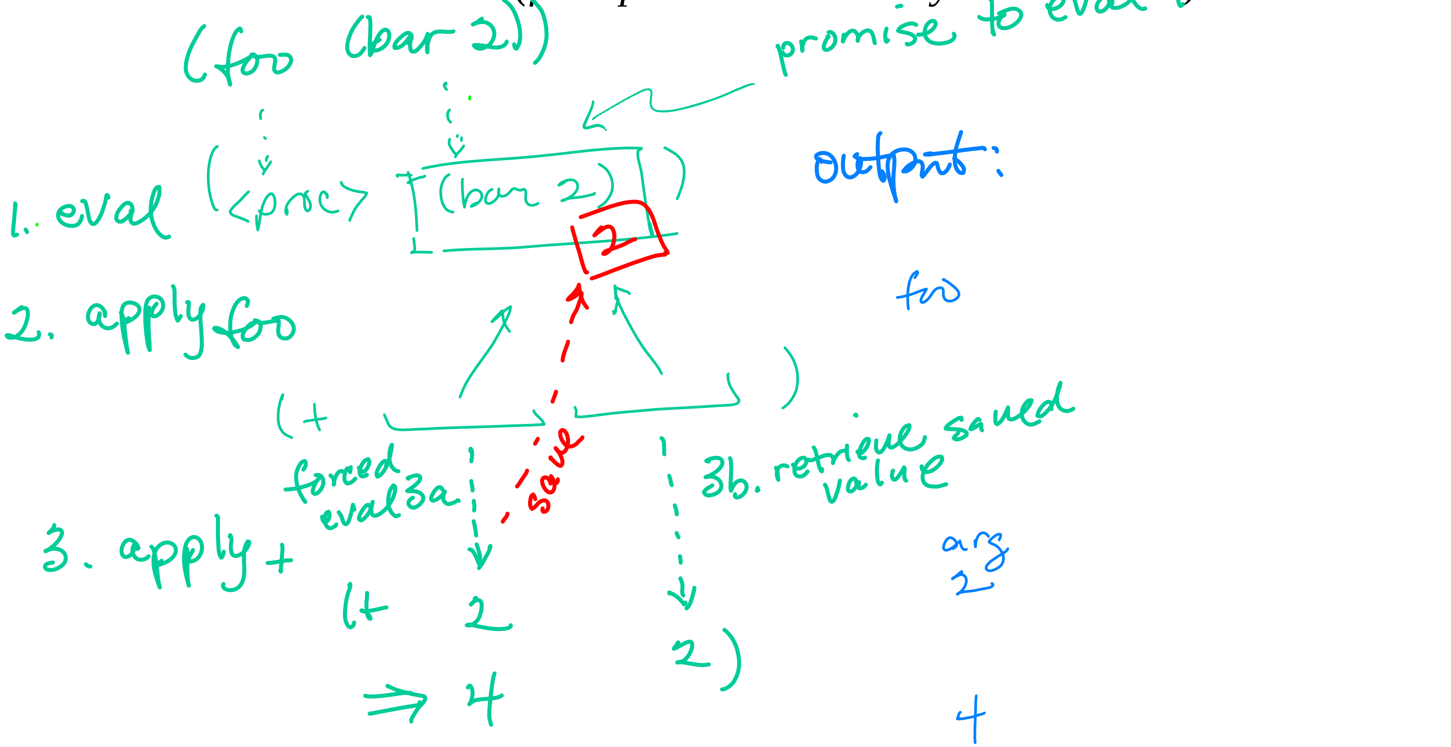
```
(define (foo x)
    (display 'foo)
    (+ x x))
```

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

```
(foo (bar 2))
```

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

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*

(foo (baz 2))

1. eval (<proc> 7)

2. apply foo (+ 7 7)

3. apply + ⇒ 14

⇒ 4

output:

arg
7
foo
14
4

printout: arg 7 foo 14 4

problem 1b: normal (lazy) order

1. (define y 5)

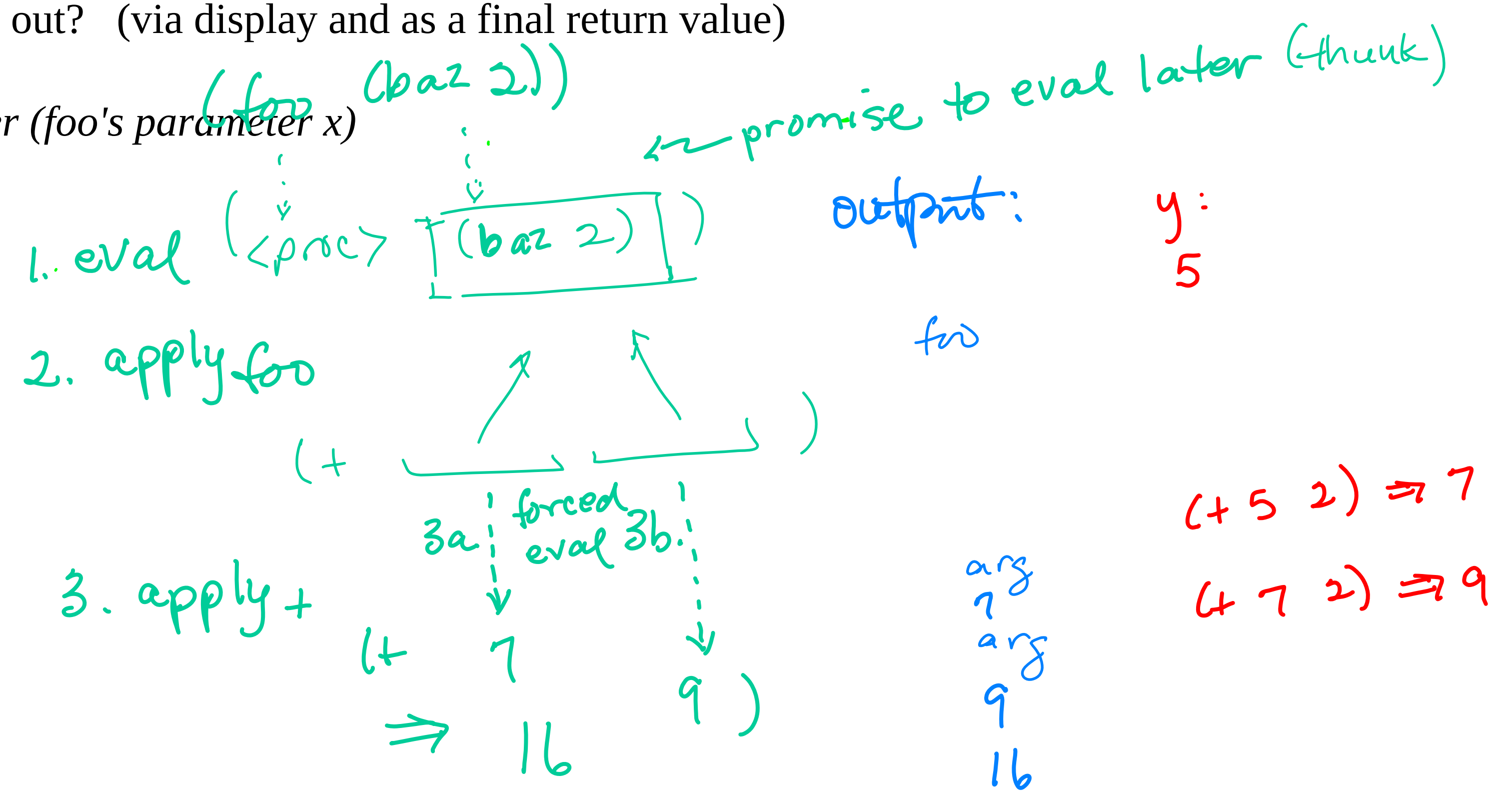
```
(define (foo x)
  (display 'foo)
  (+ x x))

(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)*



printout | foo arg 7 arg 9 16

problem 1c: normal order with memoization

1. (define y 5)

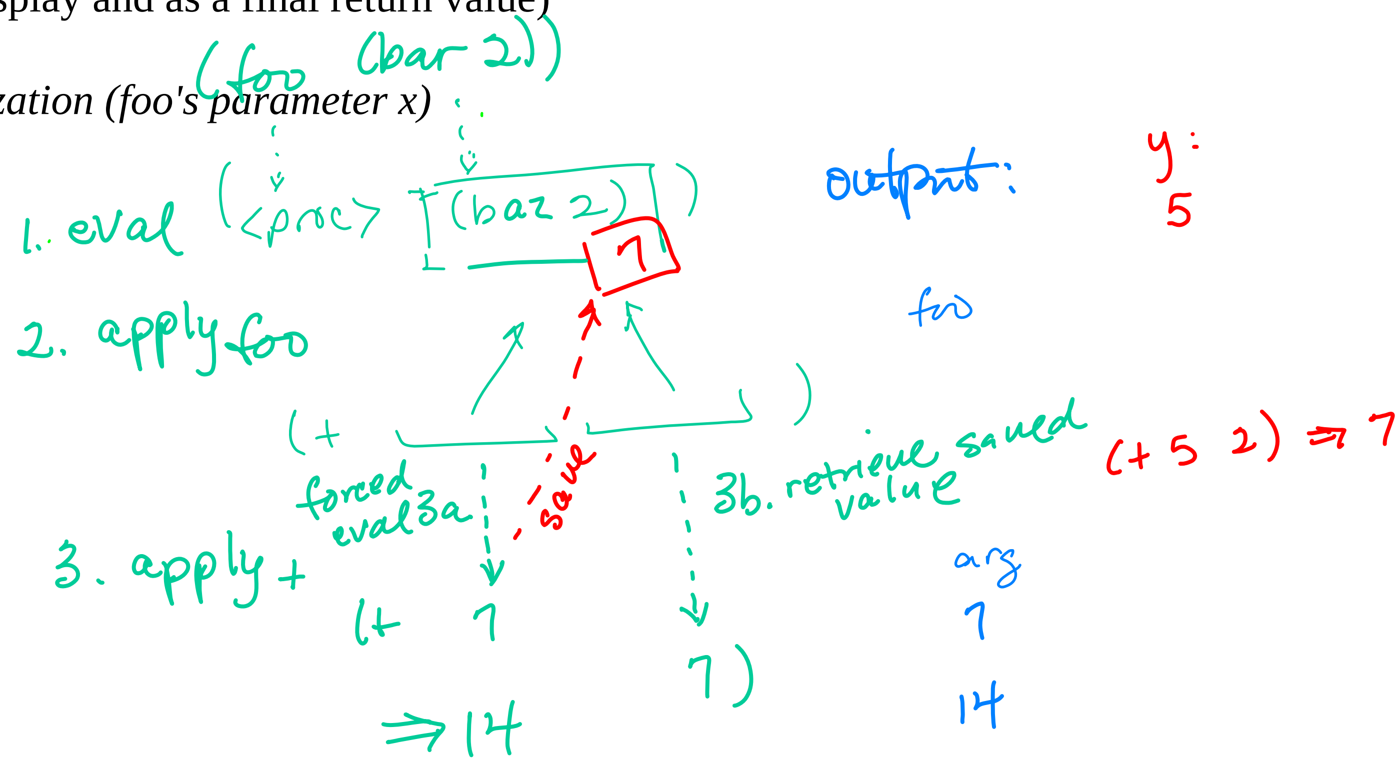
```
(define (foo x)
  (display 'foo)
  (+ x x))
```

```
(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

problem 2a: applicative order

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)))
```

f

; example output:

(initialized-list (lambda(x) (* x x)) 5)

; value (1 4 9 16 25)

call the return value of accum 5 times

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

a. *applicative order*

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

```
(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)

b. *normal order (initialized-list's parameter f)* n = x

		n	lst
1.	helper	5	((accum) 5)
2.	helper	4	((accum) 4)((accum) 5))
3.	helper	3	((accum) 3)((accum) 4) ((accum) 5))
4.	helper	2	((accum) 2)((accum) 3) ... ((accum) 5))
5.	helper	1	((accum) 1) ((accum) 2) ... ((accum) 5))

(accum) is not evaluated until return value is needed

each call to accum reinitializes count to 0

printout: (1 2 3 4 5)

problem 2~~b~~c: normal (lazy) order + memoization

2. (define (initialized-list f n)
```
(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)))
```

f (annotation on the lambda)

; 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~~c. *normal order + memoization (initialized-list's parameter f)* n = x

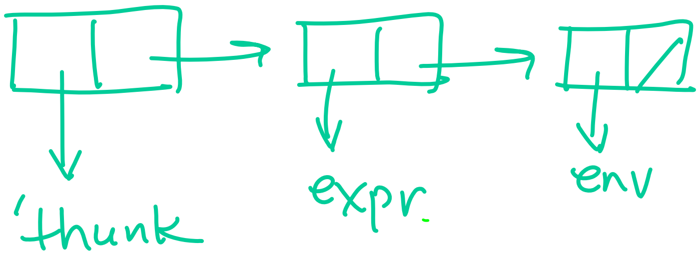
first call to accum passes a procedure (call it f) with local variable count; later calls to accum, find f + increment count

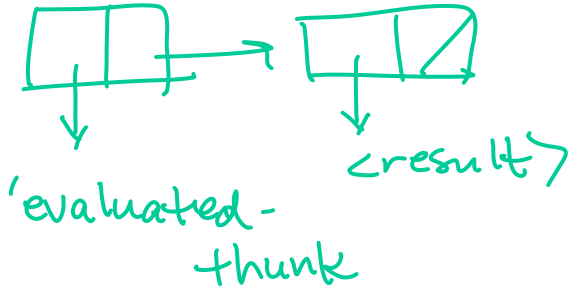
			lst
			()
1.	helper	5	((accum) 5)
2.	helper	4	((accum) 4)((accum) 5))
3.	helper	3	((accum) 3)((accum) 4) ((accum) 5))
4.	helper	2	((accum) 2)((accum) 3) ... ((accum) 5))
5.	helper	1	((accum) 1) ((accum) 2) ... ((accum) 5))

printout: (1 3 6 10 15)

0+1 0+1+2 0+1+2+3 ...

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

(lambda (a (b lazy) (c lazy_memo))

a → eval before proc applic

(b lazy) → delayed; re-evaluated each time needed

thunk

(c lazy_memo) → delayed; evaluated first time needed, value saved

thunk-memo

e.g.

```
(define (initialized-list (f lazy) n)
      ...)
```