

6.184 Lecture 5

Scheme Variants

Normal Order

Lazy Evaluation

Streams

Edited by Ben Vandiver <benmv@alum.mit.edu>

Original Material by Eric Grimson & Duane Boning

Further Variations on a Scheme

Beyond Scheme – more language variants

Lazy evaluation

- Complete conversion – normal order evaluator
- Selective Laziness: Streams

Punchline: Small edits to the interpreter give us a *new programming language*

Evaluation model

Rules of evaluation:

- If expression is self-evaluating (e.g. a number), just return value
- If expression is a name, look up value associated with that name in environment
- If expression is a lambda, create procedure and return
- If expression is special form (e.g. if) follow specific rules for evaluating subexpressions
- If expression is a compound expression
 - Evaluate subexpressions in any order
 - If first subexpression is primitive (or built-in) procedure, just apply it to values of other subexpressions
 - If first subexpression is compound procedure (created by lambda), evaluate the body of the procedure in a new environment, which extends the environment of the procedure with a new frame in which the procedure's parameters are bound to the supplied arguments

Alternative models for computation

- Applicative Order (aka Eager evaluation):
 - evaluate all arguments, then apply operator
- Normal Order (aka Lazy evaluation):
 - go ahead and apply operator with unevaluated argument subexpressions
 - evaluate a subexpression only when value is *needed*
 - to print
 - by primitive procedure (that is, primitive procedures are "*strict*" in their arguments)
 - to test (if predicate)
 - to apply (operator)

Making Order of Evaluation Visible

- (define (notice x)
 (display “noticed”)
 x)
- (+ (notice 52) (notice (+ 4 4)))
noticed
noticed
=> 60

Applicative Order Example

```
(define (foo x)
  (display "inside foo")
  (+ x x))
```



```
(foo (notice 222))
=> (notice 222)
=> 222
```

```
=> (begin (display "inside foo")
          (+ 222 222))
```

We first evaluated argument, then substituted value into the body of the procedure

```
noticed
inside foo
```

```
=> 444
```

Normal Order Example

```
(define (foo x)
  (display "inside foo")
  (+ x x))
```

```
(foo (notice 222))
```

```
=> (begin (display "inside foo")
          (+ (notice 222)
             (notice 222)))
```

From body
of foo

As if we substituted the *unevaluated expression* in the body of the procedure

```
inside foo
noticed
noticed
```

```
=> 444
```

Applicative Order vs. Normal Order

```
(define (foo x)
  (display "inside foo")
  (+ x x))

(foo (notice 222))
```

Applicative order

noticed
inside foo

Think of as substituting values for variables in expressions

Normal order

inside foo
noticed
noticed

Think of as expanding expressions until only involve primitive operations and data structures

Normal order (lazy evaluation) versus applicative order

- How can we change our evaluator to use normal order?
 - Create “promises” – expressions whose evaluation has been delayed
 - Change the evaluator to force evaluation only when needed
- Why is normal order useful?
 - What kinds of computations does it make easier?

Mapply – the original version

```
(define (mapply procedure arguments)
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else (error "Unknown procedure" procedure))))
```

The diagram highlights the flow of actual values in the `mapply` function. Two red boxes, each labeled "Actual values", are connected to the `arguments` parameter in the function definition. The first box points to the `arguments` parameter in the `(apply-primitive-procedure procedure arguments)` branch. The second box points to the `arguments` parameter in the `(extend-environment (procedure-parameters procedure) arguments (procedure-environment procedure))` branch.

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```

Need to convert
to actual values

Delayed
expressions

Need to create
delayed version
of arguments
that will lead to
values

Delayed
Expressions

Lazy Evaluation – `l-eval`

- Most of the work is in `l-apply`; need to call it with:
 - actual value for the operator
 - just expressions for the operands
 - the environment...

```
(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((application? exp)
         (l-apply (actual-value (operator exp) env)
                   (operands exp)
                   env))
        (else (error "Unknown expression" exp))))
```

Meval versus L-Eval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((cond? exp) (meval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                  (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

```
(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((cond? exp)
         (meval (cond->if exp) env))
        ((application? exp)
         (l-apply (actual-value (operator exp) env)
                  (operands exp) env))
        (else (error "Unknown expression" exp))))
```

Actual vs. Delayed Values

```
(define (actual-value exp env)
  (force-it (1-eval exp env)))
```

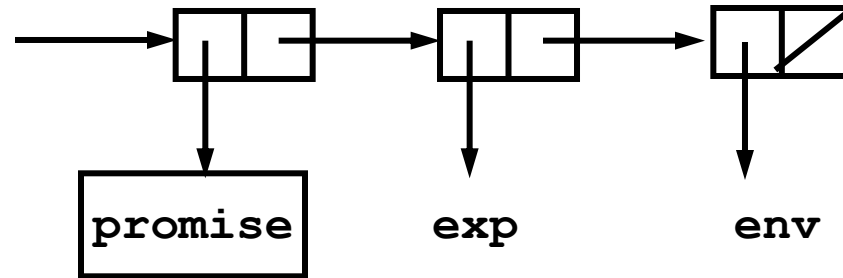
```
(define (list-of-arg-values exps env) Used when applying a
  (if (no-operands? exps) '() primitive procedure
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                                env))))
```

```
(define (list-of-delayed-args exps env) Used when applying a
  (if (no-operands? exps) compound procedure
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

Representing Promises

- *Abstractly* – a "promise" to return a value when later needed ("forced")

- *Concretely* – our representation:



- Book calls it a *thunk*, which means procedure with no arguments.
- Structure looks very similar.

Promises – delay-it and force-it

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

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

```
(define (actual-value exp env)
  (force-it (l-eval exp env)))
```

Lazy Evaluation – other changes needed


- Example – need actual predicate value in conditional if...

```
(define (l-eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))
```

- Example – don't need actual value in assignment...

```
(define (l-eval-assignment exp env)
  (set-variable-value!
   (assignment-variable exp)
   (l-eval (assignment-value exp) env)
   env)
  'ok)
```

Examples

- (define identity (lambda (x) x)) **identity: <proc>**
- (define a (notice 3)) **a: promise 3 Noticed!**
- (define b (identity (notice 3))) **b: promise (notice 3)**
- (define c b) **c:** 
- (define d (+ b c)) **d: 6 Noticed! Noticed!**
- (define plus (identity +)) **plus: promise +**
- (plus a b) **=> 6 Noticed!**
- c **=> 3 Noticed!**

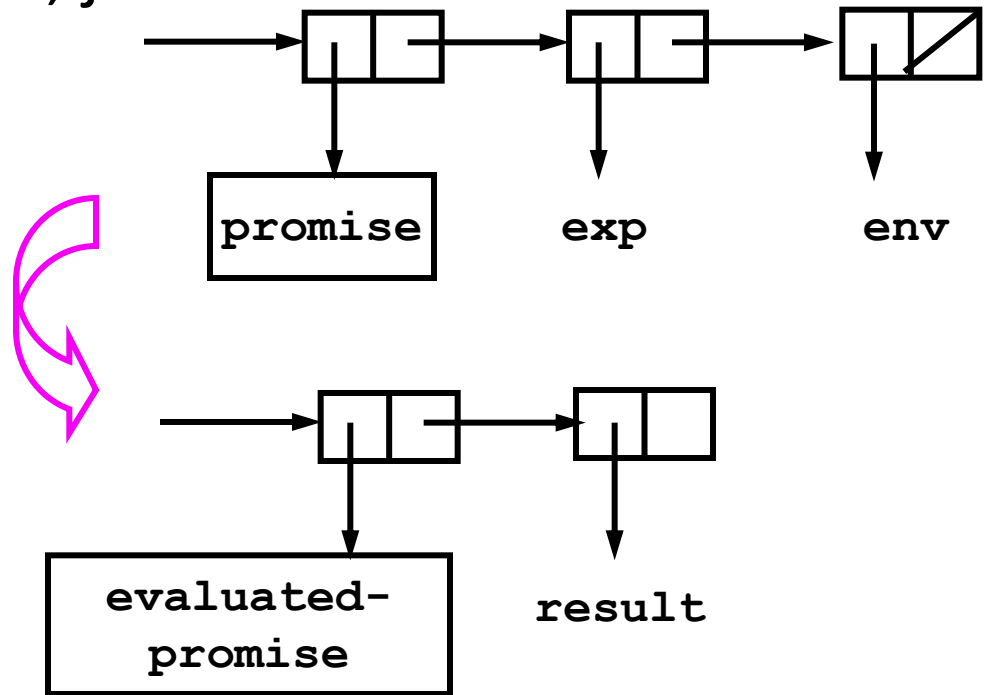
Memo-izing evaluation

- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In usual (applicative) evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we've obtained them, and avoid cost of reevaluation?

Memo-izing Promises

- *Idea*: once promise **exp** has been evaluated, remember it
- If value is needed again, just return it rather than recompute

- *Concretely* – mutate a promise into an evaluated-promise



Why mutate? –
because other names or
data structures may
point to this promise!


Promises – Memoizing Implementation

```
(define (evaluated-promise? obj)
  (tagged-list? obj 'evaluated-promise))

(define (promise-value evaluated-promise)
  (cadr evaluated-promise))

(define (force-it obj)
  (cond ((promise? obj)
        (let ((result (actual-value (promise-exp obj)
                                     (promise-env obj))))
          (set-car! obj 'evaluated-promise)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          result))
        ((evaluated-promise? obj) (promise-value obj))
        (else obj)))
```

Examples - Memoized

- (define identity (lambda (x) x)) **identity: <proc>**
- (define a (notice 3)) **a: promise 3 Noticed!**
- (define b (identity (notice 3))) **b: promise (notice 3)**
- (define c b) **c:** 
- (define d (+ b c)) **d: 6 Noticed! *CHANGE***
- (define plus (identity +)) **plus: promise +**
- (plus a b) **=> 6 *CHANGE***
- c **=> 3 *CHANGE***

Summary of lazy evaluation

- This completes changes to evaluator
 - Apply takes a set of expressions for arguments and an environment
 - Forces evaluation of arguments for primitive procedure application
 - Else defers evaluation and unwinds computation further
 - Need to pass in environment since don't know when it will be needed
 - Need to force evaluation on branching operations (e.g. if)
 - Otherwise small number of changes make big change in behavior of language

Laziness and Language Design

- We have a dilemma with lazy evaluation
 - Advantage: only do work when value actually needed
 - Disadvantages
 - not sure when expression will be evaluated; can be very big issue in a language with side effects
 - may evaluate same expression more than once
- Memoization doesn't fully resolve our dilemma
 - Advantage: Evaluate expression at most once
 - Disadvantage: What if we *want* evaluation on each use?
- Alternative approach: **Selective Laziness**

Streams – the lazy way

Beyond Scheme – designing language variants:

- Streams – an alternative programming style



to infinity, and beyond...

Decoupling computation from description

- Can separate order of events in computer from apparent order of events in procedure description

```
(list-ref
```

```
  (filter (lambda (x) (prime? x))
```

```
    (enumerate-interval 1 10000000))
```

```
  100)
```

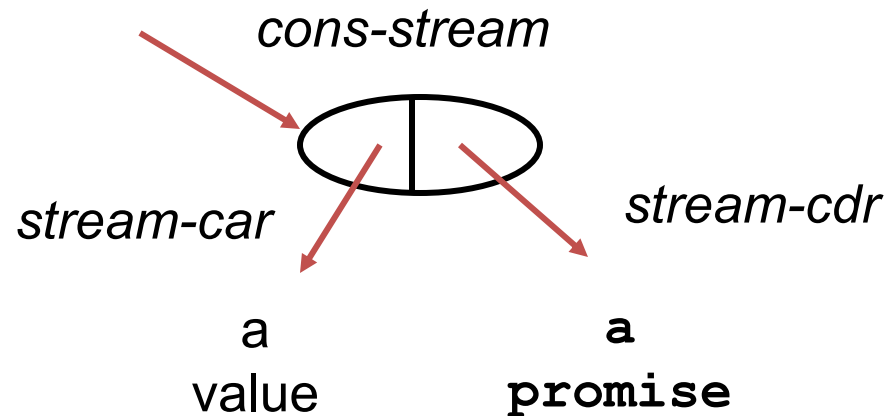
Creates 100K
elements

Creates 1M
elements

Generate only what you actually need...

Stream Object

- A pair-like object, except the cdr part is *lazy* (not evaluated until needed):



- Example

```
(define x (cons-stream 99 (/ 1 0)))
```

```
(stream-car x) => 99
```

```
(stream-cdr x) => error - divide by zero
```

Stream-cdr applies the promise wrapped around (/ 1 0), resulting in an error

Implementing Streams

- Stream is a data structure with the following contract:
 - `(cons-stream a b)` – cons together *a* with promise to compute *b*
 - `(stream-car s)` – Returns car of *s*
 - `(stream-cdr s)` – Forces and returns value of cdr of *s*
- Implement in regular evaluator with a little syntactic sugar
 - `(define (cons-stream->cons exp)`
 `(list 'cons (second exp) (list 'lambda '() (third exp))))`
 - In meval, add to cond:
 `((cons-stream? exp) (meval (cons-stream->cons exp) env))`
 - And the following regular definitions (inside meval!)
 - `(define stream-car car)`
 - `(define (stream-cdr s) ((cdr s)))`
- Streams can be done in lazy eval
 - `(define (cons-stream a b) (cons a b))` ← doesn't work! (Why?)

Ints-starting-with

- (define (ints-starting-with i)
 (cons-stream i (ints-starting-with (+ i 1))))

Delayed!

- Recursive procedure with no base case!
 - Why does it work?

Stream-ref

- (define (stream-ref s i)
 (if (= i 0)
 (stream-car s)
 (stream-ref (stream-cdr s) (- i 1))))
- Like list-ref, but cdr's down stream, forcing

Stream-filter

```
(define (stream-filter pred str)
  (if (pred (stream-car str))
      (cons-stream (stream-car str)
                   (stream-filter pred
                                   (stream-cdr str)))
      (stream-filter pred
                    (stream-cdr str))))
```

Decoupling Order of Evaluation

```
(define (stream-filter pred str)
  (if (pred (stream-car str))
      (cons-stream (stream-car str)
                   (stream-filter pred
                                   (stream-cdr str)))
      (stream-filter pred
                    (stream-cdr str))))

(stream-ref
 (stream-filter (lambda (x) (prime? x))
               (ints-starting-with 2))
 4)
```

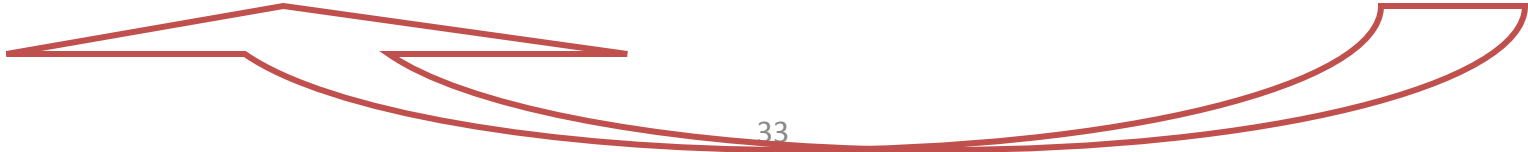
Decoupling Order of Evaluation

```
(stream-ref  
  (stream-filter (lambda (x) (prime? x))  
    (ints-starting-with 2))
```

4)

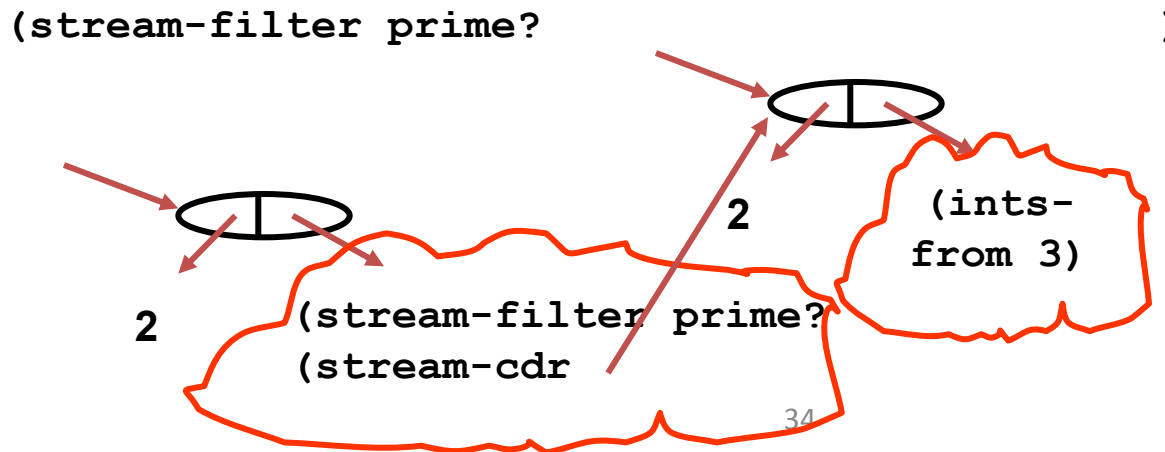
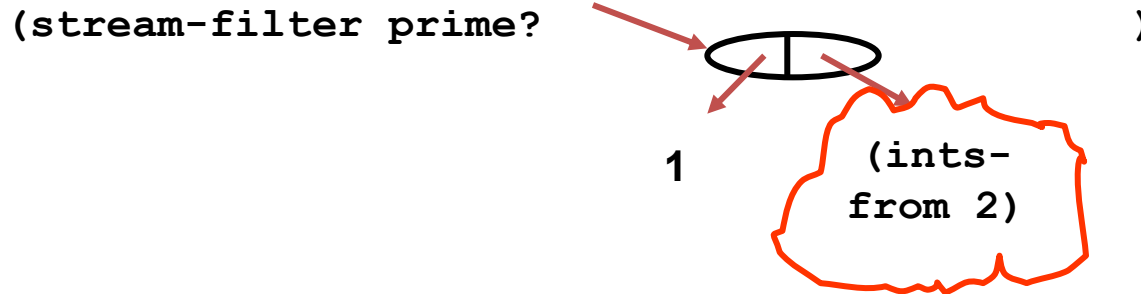
Creates 1 element,
plus a promise

Creates 1 element,
plus a promise



Decoupling Order of Evaluation

```
(stream-filter prime? (ints-from 1))
```



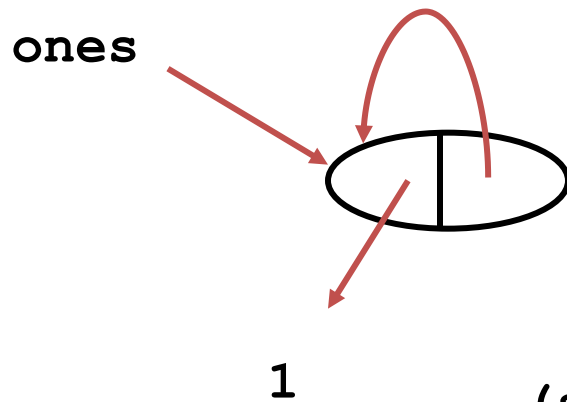
One Possibility: Infinite Data Structures!

- Some very interesting behavior

```
(define ones (cons 1 ones))
```

```
(define ones (cons-stream 1 ones))
```

```
(stream-car (stream-cdr ones)) => 1
```



The infinite stream of 1's!

ones: 1 1 1 1 1 1

```
(stream-ref ones 1) → 1
```

```
(stream-ref ones 1000) → 1
```

```
(stream-ref35 ones 10000000) → 1
```

Finite list procs turn into infinite stream procs

```
(define (add-streams s1 s2)
  (cons-stream
    (+ (stream-car s1) (stream-car s2))
    (add-streams (stream-cdr s1)
                  (stream-cdr s2))))
```

```
(define ints
  (cons-stream 1 (add-streams ones ints)))
```

ones: 1 1 1 1 1 1

ints: 1 2 3 . . .

add-streams ones
ints

add-streams (str-cdr ones)
(str-cdr ints)

Finding all the primes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Using a sieve

```
(define (sieve str)
  (cons-stream
    (stream-car str)
    (sieve (stream-filter
            (lambda (x)
              (not (divisible? x (stream-car str))))
            (stream-cdr str)))))
```

```
(define primes
  (sieve (stream-cdr ints)))
```

```
(2 sieve (filter ints 2) )
```

```
(2 3 sieve (filter
  (sieve (filter ints 2)
  3))
```

Interleave

Produce a stream that has all the elements of two input streams:

```
(define (interleave s1 s2)
  (cons-stream (stream-car s1)
               (interleave s2 (stream-cdr s1))))
```

Rationals

1/1	1/2	1/3	1/4	1/5	...
2/1	2/2	2/3	2/4	2/5	...
3/1	3/2	3/3	3/4	3/5	...
4/1	4/2	4/3	4/4	4/5	...
5/1	5/2	5/3	5/4	5/5	...
...

```
(define (div-by-stream s n)
  (cons-stream (/ n (stream-car s)) (div-by-stream (stream-cdr s) n)))
(define (make-rats n)
  (cons-stream n
    (interleave (div-by-streams (stream-cdr ints) n)
      (make-rats (+ n 1)))))
(define rats (make-rats 1))
```

Power Series

- Approximate function by summation of infinite polynomial
- Great application for streams!
<We'll do this in recitation!>