6.037 Lecture 7B

Scheme Variants
Normal Order
Lazy Evaluation
Streams

Edited by Mike Phillips & Ben Vandiver
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

Environment 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 a 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

=> 444
Normal Order Example

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

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

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
Normal order

```
noticed
inside foo
noticed
noticed
```

Think of as substituting values for variables in expressions
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?

m-apply – the original version

```
(define (m-apply 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)))))
```

Actual values
Actual values

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env)
  (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))))
```

Delayed expressions
Need to convert to actual values

Lazy Evaluation – l-eval

```
(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)))
```

• 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...

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

Need to convert to actual values

```
m-eval versus l-Eval

(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        (else (error "Unknown expression type -- EVAL" exp))))

(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))))

Actual vs. Delayed Values

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

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps) env)))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps) env))))

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: Noticed!
• (define d (+ b c)) d: 6 Noticed! Noticed!
• (define plus (identity +)) Plus: promise +
• (plus a b) => 6 Noticed!
• c => 3 Noticed!

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)
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 re-evaluation?

Memo-izing Promises

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

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)
• (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
Choose via Parameter Declarations

• Handle lazy and lazy-memo extensions in an upward-compatible fashion.
  
  (lambda (a (b lazy) c (d lazy-memo)) ...)

• "a", "c" are usual variables (evaluated before procedure application)

• "b" is lazy; it gets (re)-evaluated each time its value is actually needed

• "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed

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 1000000)) 100)

Create 100K elements

Generate only what you actually need...

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 a b) (cons a (lambda () b)))

Streams can be done in lazy eval

  (define (cons-stream a b) (cons a (lambda () b)))  // doesn’t work! (Why?)
  
  (define (cons-stream a b) (cons a (lambda (i) b)))

Streams – the lazy way

Beyond Scheme – designing language variants:

• Streams – an alternative programming style

  to infinity, and beyond...

Stream Object

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

  (cons-stream a b)
  
  (stream-car s) – Returns car of s
  
  (stream-cdr s) – Forces and returns value of cdr of s

• Example

  (define x (cons-stream 99 (/ 1 0)))
  
  (stream-car x) => 99
  
  (stream-cdr x) => error – divide by zero

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

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)))))

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)))))

One Possibility: Infinite Data Structures!

- Some very interesting behavior

(define ones (cons 1 ones))

(define (cons-stream 1 ones)) => 1

ones: 1 1 1 1 1 1 ....

add-streams (str-cdr ones)

add-streams ones

ints: 1 2 3 ...

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 ....

ints: 1 2 3 ...

add-streams ones

add-streams (str-cdr ones)

add-streams (str-cdr ints)
### Finding all the primes

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

Using a sieve

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

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

Finding all the primes

#### Interleave

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

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

Rationals

```scheme
(define (div-by-stream s n)
  (cons-stream (/ n (stream-car s))
               (div-by-stream (stream-cdr s) n)))
)
```

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

```scheme
(define rats (make-rats 1))
```

#### Power Series

- Approximate function by summation of infinite polynomial
- Great application for streams!

"<We’ll do this in recitation!>"