Review Analyzing Evaluator

• What procedures look like
• What the output of `analyze` was
• Fact: The body of a `lambda` gets analyzed!
• We can give names to analyzed `lambdas`

What gets returned by `mc-eval`? (not analyzing `eval`)

STK> (mc-eval '(lambda (x) (* x x)) '(((a) 3)))

A. `(procedure (x) ((* x x)) (((a) 3)))`
B. `(procedure (x) (* x x) (((a) 3)))`
C. `(lambda (x) ((* x x)) (((a) 3)))`
D. `(lambda (x) (* x x) (((a) 3)))`
E. Other (or no clue!)

REVIEW What is a procedure?

STK> (mc-eval '(lambda (x) (* x x)) '(((a) 3)))

The body of a lambda gets analyzed!

STK> (mc-eval '(lambda (x) (λ(env) 3)) '(((a) 3)))

List representing expression → analyze → STK Scheme expression

Global
a: 3

Procedure (x) `((λ(env) 3)) (((a) 3)))`

Global
a: 3

Procedure (x) `((a) . (3))`
(analyze '(if #t 3 4))
(define (analyze exp)
  (cond ((self-evaluating? exp)
          (analyze-self-evaluating exp))
        ((if? exp) (analyze-if exp))...
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
         (cproc (analyze (if-consequent exp)))
         (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))

The body of a lambda gets analyzed!
STk> (mc-eval '(define f
                (lambda (x)
                  '((if #t 3 4))
                )
          '((a) 3)))
(f 1) ((a) (3))

Understanding Lazy [eval/smurf]
- Thunk ADT
  - A Thunk List – not a real scheme thunk
- Storing the environment
  - If we’re going to delay the evaluation of arguments we need to keep track of the environment where they should be evaluated.
- Delay arguments to user-defined procedures!
  - Not to primitive procedures
Thunk ADT (not a STk REAL 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))
```

exp is (A) list representing an expression or (B) REAL Scheme?

What would lazy [eval/smurf] do?

```
STk> (define (square x) (* x x))
STk> (square (+ 2 3))
```

```
(square (+ 2 3))
(square (+ 2 3))
(square 5)
(* (+ 2 3) (+ 2 3))
(* 5 5)
25
```

A. Applicative Order  B. Normal Order

STk> (define x 3)
STk> (define (square x) (* x x))
STk> (square (+ 2 x))

```
(square (+ 2 x))
(square (+ 2 x))
(square 5)
(* (+ 2 x) (+ 2 x))
(* 5 5)
25
```

Applicative Order  Normal Order

If we’re going to delay-it we need to keep track of the environment!

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

Before we call square: figure out arguments!

```
When I get forced: evaluate the exp in this environment
```

How (regular) mc-eval evaluated args?!

```
(define (mc-eval exp env)
  (cond ...
      ((application? exp)
       (mc-apply (mc-eval (operator exp) env)
                 (list-of-values (operands exp) env))))
```

```
We're going to change the range of mc-eval, so we'll have to change this too.
```

How lazy [smurf] mc-eval evaluates args?!

```
(define (mc-eval exp env)
  (cond ...
      ((application? exp)
       (mc-apply mc-eval (operator exp) env)
                 (list-of-values (operands exp) env)))
```

```
We're going to change the range of mc-eval, so we'll have to change this too.
```
Regular mc-apply

(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)))

This is the lazy version – how many changes? A. 1 B. 2 C. 3 D. 4

(define (mc-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)))))

The lazy mc-apply

DO evaluate the arguments

Don’t evaluate the arguments

Primitive procedure

Compound (User defined) Procedure

In the lazy version
Where do arguments get delayed?
A. In mc-eval
B. In mc-apply
C. In both
D. In neither
E. ???

In the REGULAR version
Where do arguments get evaluated?
A. In mc-eval
B. In mc-apply
C. In both
D. In neither
E. ???

In the lazy version
Where do arguments get evaluated?
A. In mc-eval
B. In mc-apply
C. In both
D. In neither
E. ???
How many of these are different in Normal vs. Applicative order?
(invent one example that isn’t and one that is)

<table>
<thead>
<tr>
<th>STk&gt; 3</th>
<th>STk&gt; (set! x 4)</th>
<th>STk&gt; (define x 3)</th>
<th>STk&gt; (if #t 3 4)</th>
<th>STk&gt; x</th>
<th>STk&gt; (lambda (x) x)</th>
<th>STk&gt; (begin 2 3)</th>
<th>A. 0 B. 1-2 C. 3-5 D. 6-8 E.??</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>okay</td>
<td>(define x 3)</td>
<td>(if #t 3 4)</td>
<td>x</td>
<td>(lambda (x) x)</td>
<td>(begin 2 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A problem with delaying stuff

<table>
<thead>
<tr>
<th>STk&gt; (load &quot;lazy.scm&quot;)</th>
<th>okay</th>
<th>STk&gt; (define g-env (setup-environment))</th>
<th>g-env</th>
<th>STk&gt; (mc-eval '((lambda (x) (+ 1 x)) (+ 2 3)) g-env)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>okay</td>
<td></td>
<td></td>
<td>(thunk (+ 2 3) g-env)</td>
</tr>
</tbody>
</table>

User-defined procedure: Don’t evaluate the arguments

What can be returned by the lazy mc-eval function

What can be ret
A. Values
B. Lists
C. Thunk ADTs
D. All of the above
E. None of the above

What happens here?

<table>
<thead>
<tr>
<th>STk&gt; (load &quot;lazy.scm&quot;)</th>
<th>okay</th>
<th>STk&gt; (define g-env (setup-environment))</th>
<th>g-env</th>
<th>STk&gt; (mc-eval '((lambda (x) (+ 1 x)) (+ 2 3)) g-env)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>okay</td>
<td></td>
<td></td>
<td>(thunk (+ 1 5) g-env)</td>
</tr>
</tbody>
</table>

What is returned?
a. (thunk (+ 1 5) g-env)
b. (thunk (+ 1 (+ 2 3)) g-env)  
c. 6  
d. Something else  
e. ???
SOLUTION

If the driver loop needs to print it – make sure you haven’t been TOO lazy.

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output))))

This was: mc-eval

mc-eval might return a delayed argument from a compound procedure

Example of why we call actual-value

STk> (load "lazy.scm")
okay
STk> (define g-env (setup-environment))
g-env
STk> (mc-eval
      '(((lambda (x) x)
          ((lambda (y) y)
            (+ 2 3))
        )
      )
      g-env)
(thunk ((lambda (y) y) (+ 2 3))

A. (mc-eval
    B. (actual-value

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

(define (force-it obj)
  (if (thunk? obj)
    ?? (thunk-exp obj)
    (thunk-env obj))
    obj))

What happens if we pass a Thunk ADT as exp?

A. errors
B. application
C. lambda
D. self-eval.
E.??

if’s need actual values!

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

mc-eval sometimes returns Thunk ADTs

Summary & Additional Notes

• Thunk ADTs could also be memoized
• We delayed arguments to compound procedures
  – Compound procedures are defined by the user
• We didn’t delay arguments to primitive procedures
• We made sure we had the actual value to print it
• Ifs needed REAL values