In the **REGULAR** version
Where do arguments get evaluated?

A. In mc-eval  
B. In mc-apply  
C. In both  
D. In neither  
E. ???

---

**Changes to mc-eval for the lazy evaluator**

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

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

---

**The lazy mc-apply**

- **DO** evaluate the arguments
- **Don't** evaluate the arguments

**The RANGE of mc-eval includes Thunk ADTs**

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

**mc-eval may return a Thunk ADT**

**User-defined procedure:**  Don't evaluate the arguments

---

**THE RULES**

The lazy mc-eval might return a Thunk ADT, we should force these:

- Before you print something returned by mc-eval  
- Before you pass arguments to a primitive procedure – if is LIKE a primitive procedure the predicate shouldn't be a Thunk ADT.

We should CREATE Thunk ADTs (delay stuff)

- Before you pass arguments to a compound procedure
Tracing a call that returns a Thunk ADT from mc-eval

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

STk> (mc-eval '((lambda (x) x) (+ 2 3)) g-env)

This was: mc-eval

Replace a call to mc-eval to avoid printing a Thunk ADT

(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))))

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

actual-value

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

Should we add a Thunk? check to mc-eval?

A. No – not necessary
B. No handled by another case
C. ?

Should we add a Thunk ADT check to mc-eval?

A. Yes
B. No handled by another case
C. ?

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

The image contains code snippets and explanations related to Scheme programming, specifically focusing on the mc-eval function and its behavior with respect to Thunk ADTs. The text discusses the evaluation process, the need for actual values, and the implications of returning Thunk ADTs from mc-eval. Examples are given to illustrate how the code works in practice.
Example of why we call actual-value

STk> (load "lazy.scm")
ok
STk> (define g-env (setup-environment))
g-env
STk> (mc-eval
  '((((lambda (x) x)
      ((lambda (y) y)
        (+ 2 3)))
    g-env)
  a.(thunk ((lambda (y) y) (+ 2 3))
  b.(thunk ((lambda (x) x) (+ 2 3))
  c.(thunk ((lambda (x) x) ((lambda (y) y) (+ 2 3)))
  d. 5  e. ??

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

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

When I get forced: evaluate the exp in this environment

Why you need to evaluate Thunk ADTs in their original environment

STk> (define (crazy arg)
  (let ((x 3))
    (+ x arg arg)))
STk> (define x 4)
STk> (crazy (+ x 1))

WRONG way: without old environment

(crazy (+ x 1))
(crazy (+ x 1))
(crazy 5)
(* x arg arg)
(* 3 5 5)
(* 3 (+ 3 1)(+ 3 1))

Does the third element point to:
A. Global
B. E1
C. E2
D. None
E. ??

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 for predicates

Run (query) and tell it some facts

STk> (load "query.scm")
ok
STk> (query)
;;; Query input:
(assert! (colleen likes cookies))
Assertion added to data base.

;;; Query input:
Some facts I told the query system

(assert! (colleen likes cookies))
(assert! (hamilton likes cookies))
(assert! (stephanie likes oreos))
(assert! (kevin likes pizza))
(assert! (eric likes pizza))
(assert! (phill likes everything))

We can ask the query system questions

;;; Query input:
(?who likes pizza)

We can get multiple answers!!!!!!

The query system “filters” out facts that don’t match

;;; Query input:
(?who likes pizza)

Don’t keep this one!

The query system “filters” out facts that don’t match

;;; Query input:
(?who likes pizza)

Keep this one!

We can ask the query system questions

;;; Query input:
(?who likes pizza)

The variable name doesn’t matter

;;; Query results:
(kevin likes pizza)
(eric likes pizza)

Filtering allows us to get multiple things back!

(colleen likes cookies)
(hamilton likes cookies)
(stephanie likes oreos)
(eric likes pizza)
(phill likes everything)
(kevin likes pizza)
Write a query that matches ALL assertions that we've added!
(colleen likes cookies)
(hamilton likes cookies)
(stephanie likes oreos)
(eric likes pizza)
(phill likes everything)
(kevin likes pizza)
A. Not possible  D. Need 3 variables
B. Need 1 variable  E. Stuck
C. Need 2 variables

What can a query return
(colleen likes cookies)
(hamilton likes cookies)
(stephanie likes oreos)
(eric likes pizza)
(phill likes everything)
(kevin likes pizza)

How many results?
A. 0 B. 1 C. 2 D. 3-6 E. ??

Query input:
(?who likes elephants)

Do these match?
(assert! (colleen likes ice cream))
(assert! (colleen likes cookies))

Query input:
(colleen likes ?what)
Query results:
A. Only cookies  B. Only ice cream  C. Both  D. Neither  E. ??

We need to think about pairs

?? Query input:
(colleen likes ?what)
(colleen .(likes .(?what .())))

We need to think about pairs

(colleen likes cookies)
(colleen .(likes .(cookies.())))

We need to think about pairs

(colleen likes ice cream)
(colleen .(likes .(ice .(cream .()))))

We need to think about pairs

?? Query input:
(colleen likes . ?what)
(colleen .(likes . ?what))

Equivalent

(matches!)

We need to think about pairs

(colleen likes ice cream)
(colleen .(likes .(ice .(cream .()))))
**Facts with variables: rules**

We can add things WITH variables to the “facts”

```lisp
(assert! (rule (car ?a (?a . ?b))))
```

```lisp
;;; Query input:
(car ?x (5 6 7))
(car ?x (5 . (6 . (7 . ()))))
```

```lisp
;;; Query results:
(car 5 (5 6 7))
```

**We can return things with variables**

```lisp
;;; Query input:
(car 1 ?y)
```

```lisp
;;; Query results:
(car 1 (1 . ?b))
(car 1 (1 . (?b-27))
```

**Rules can have Bodies**

```lisp
(assert! (rule (awesome ?x)
(?x likes cookies)))
```

```lisp
;;; Query input:
(awesome ?y)
```

```lisp
?x = ?y ; figured out
(awesome ?y)
(?x likes cookies)
(stephanie likes oreos)
```

**Write 2nd**

```lisp
;;; Query input:
(2nd ?x (4 5 6))
```

```lisp
;;; Query results:
(2nd 5 (4 5 6))
```

```lisp
;;; Query input:
(2nd 3 ?x)
```

```lisp
;;; Query results:
(2nd 3 (?a-29 3 . (?c-29))
```