LAZY EVALUATOR AND ANALYZING EVALUATOR 12

COMPUTER SCIENCE 61AS

Warm-up

1. What are applicative order and normal order? Which one is used in Lazy Evaluator?

   Solution: In applicative order, all the arguments to procedures are evaluated right away when the procedure is applied. In normal order, procedure arguments are NOT evaluated until the actual values of the arguments are needed. Scheme uses applicative order, but Lazy Evaluator uses normal order.

2. What is a thunk? What are the two procedures that are used to create and evaluate a thunk?

   Solution: A thing that remembers an expression and an environment so that it can be evaluated later, whenever it is needed. We create a thunk by delay-it and evaluate the code in the thunk by force-it.

Lazy Evaluator

1. There are two versions of the lazy evaluator – one in which thunks are memoized, and the other without memoization (memoization of thunks is analogous to the memoization of streams in lab 11). Which procedures need to be different in the memoizing and the non-memoizing versions of the lazy evaluator?

   Solution: force-it only – it can remember its value the first time it is evaluated, and then every other time it just returns that value.

2. When should an expression be delayed in the lazy evaluator? (In other words, when should delay-it be called?) Hint: What's the difference between applicative and normal order?)

Solution: When we see a procedure invocation, we delay the argument values.

3. What should an expression be forced in the lazy evaluator, and why?

Solution: a. The procedure in a procedure application has to be forced – we need to know if it is primitive or compound, and we need to apply it.
b. The arguments to a primitive procedure – since we use underlying Scheme's apply, which doesn't know about thunks.
c. Certain arguments to special forms, such as the test condition in an if – for similar reasons as part b.
d. When something is being printed to the screen (you shouldn't print a thunk).

Lazy Evaluator VS Metacircular Evaluator

1. For each of the following, say whether lazy evaluator would evaluate and print an answer faster than the regular metacircular evaluator.

(a)
```
(define x 4)
(define (foo a b)
        (set! x (+ a 2))
        b)
(foo (* 6 9) (* 8 3))
```

Faster Not faster

Solution: Not Faster

(b)
```
((lambda (x y) (+ x y)) (* 4 4) (* 9 9))
```

Faster Not faster

Solution: Not Faster

(c)
```
((lambda (x y) (* 2 x)) (* 4 4) (* 9 9))
```

Faster Not faster

Solution: Faster

2. Here's the way we represent (non-memoized) thunks:

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

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

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

(define (thunk? obj)
        (tagged-list? obj 'thunk))
```

What would be the result of the following pieces of code in the metacircular evaluator, and in the lazy evaluator? Assume second is a primitive procedure that returns its second argument.

(a)
```
(define (f x) 42)
(f (/ 4 0))
```

MCE:__________ Lazy evaluator:__________

Solution: MCE: Error, Lazy: 42

(b)
```
(list 'thunk 1 2)
```

MCE:__________ Lazy evaluator:__________

Solution: MCE: (thunk 1 2), Lazy: 1

(c)
```
(list 'thunk '(+ 2 3) ())
```

MCE:__________ Lazy evaluator:__________

Solution: MCE: (thunk (+ 2 3) ()), Lazy: Error

(d)
```
(define x 10)
(begin (second (set! x (- x 1)) (set! x (- x 1)))
       x)
```

MCE:__________ Lazy evaluator:__________

Solution: MCE: 8, Lazy: 8

(e)
```
(define (my-second x y) y)
(define x 10)
(begin (my-second (set! x (- x 1)) (set! x (- x 1)))
      x)
```

MCE:__________ Lazy evaluator:__________

Solution: MCE: 8, Lazy: 10

(f)
```
(define x 10)
(begin (second x
               (my-second (set! x (- x 1))
                          (set! x (- x 1))))
       x)
```

MCE:__________ Lazy evaluator:__________

Solution: MCE: 8, Lazy: 9

Analyzing Evaluator

1. What is the benefit of analyzing evaluator and how is it achieved?

Solution: Analyzing evaluator achieves more efficiency compared to regular metacircular evaluator by doing the expression only once. In the evaluation process, the evaluator finds those parts which depend only on exp and not on env, and do those only once. It is analogous to memoization, but instead of memorizing the actual answer, it's memorizing the process.

2. What does the procedure analyze take as an argument? What does it return?

Solution: It takes only the expression, and returns a new procedure, the execution procedure, that encapsulates the work to be done in executing the analyzed expression.

3. Which expressions will be evaluated faster in Analyzing Evaluator?

```
a. 5
b. (* 2 3)
c. (map (lambda (x) (* x 3)) (list 1))
d. (define (factorial n)
     (if (= n 1)
         1
         (* (factorial (- n 1)) n)))
   (factorial 4)
e. (map (lambda (x) (+ x 100)) (list 1 2 3 4 5 6 7 8 9 10 11))
f. (accumulate cons nil (list ’hi ’nice ’to ’meet ’you))
```

Solution: Answer: d, e.
a: symbol, b: primitive, c: lambda is called only once, f: cons is a primitive.