Applicative vs. Normal Order

QUESTIONS

1. Above, applicative order was more efficient. Define a procedure where normal order is more efficient.

   Anything where not evaluating the arguments will save time works. Most trivially,
   
   (define (f x) 3) ;; a function that always returns 3

   When you call (f (fib 10000)), applicative order would choke, but normal order would just hapilily drop (fib 10000) and just return 3.

2. Evaluate this expression using both applicative and normal order: (square (random x)). Will you get the same result from both? Why or why not?

   Unless you’re lucky, the result will be quite different. Expanding to normal order, you have (* (random x) (random x)), and the two separate calls to random will probably return different values.

3. Consider a magical function count that takes in no arguments, and each time it is invoked, it returns 1 more than it did before, starting with 1. Therefore, (+ (count) (count)) will return 3. Evaluate (square (square (count))) with both applicative and normal order; explain your result.

   For applicative order, (count) is only called once – returns 1 – and is squared twice. So you have (square (square 1)), which evaluates to 1.

   For normal order, (count) is called FOUR times:

   (* (square (count)) (square (count))) =>
   (* (* (count) (count)) (* (count) (count))) =>
   (* (* 1 2) (* 3 4)) =>
   24

The Lazy Way Out

QUESTIONS: What is printed at each line?

1. > (define x (+ 2 3))
   > x => 5
   > (define y ((lambda (a) a) (* 3 4)))
   > y => 12
   > (define z ((lambda (b) (+ b 10)) y))
   > z => 22

2. > (define count 0)
   > (define (foo x y) (x y))
   > (define z (foo (lambda (a) (set! count a) (* a a))
       (begin (set! count (+ 1 count)) count)))
   > count => 0
   > z => infinite loop

3. > (define count 0)
> (define (incr!) (set! count (+ count 1)))
> (define (foo x)
>   (let ((y (begin (incr!) count)))
>     (if (<= count 1)
>       (foo y)
>       x)))
> (foo 10) => infinite loop

**Nondeterministic and Indecisive**

**QUESTIONS**

1. Suppose we type the following into the amb evaluator:
   > (* 2 (if (amb #t #f #t)
   >         (amb 3 4)
   >         5))
   
   What are all possible answers we can get?

6, 8, 10

2. Write a function an-atom-of that dispenses the atomic elements of a deep list (not including empty lists). For example,
   > (an-atom-of '((a) ((b (c))))) => a
   > try-again => b

   (define (an-atom-of ls)
     (cond ((null? ls) (amb))
           ((atom? ls) ls)
           (else (amb (an-atom-of (car ls))
                      (an-atom-of (cdr ls))))))

3. Use an-atom-of to write deep-member?.

   (define (deep-member? X ls)
     (let ((maybe-x (an-atom-of ls)))
       (require (equal? x maybe-x))
       #t))

4. Fill in the blanks:
   > (define (choose-member L R)
   >   (cond ((null? R) (amb))
   >         ((= (car L) (car R)) (car L))
   >         (else (amb (choose-member L (cdr R))
   >                    (choose-member (cdr L) R))))
   > (choose-member '(1 2 3) '(4 2 3))
   3
   > try-again
   2
   > try-again
   2