lazy vs. eager evaluation

```
(define (myif x y z) (if x y z))
(myif (null? '()) 0 (/ 3 0))
; error because of division by zero; this is eager evaluation and code stops
running here...

(delay (+ 3 5))
(define a (delay (+ 3 5)))
a
; lazy evaluation
```

1. `delay (+ 3 5))` creates a delayed computation, which is represented as a "promise". The expression inside `delay` is not immediately evaluated, but it is wrapped in a function-like object that can be evaluated later when the value is needed.
2. `(define a (delay (+ 3 5)))` assigns this delayed computation to the variable `a`. At this point, the expression `(delay (+ 3 5))` has not been evaluated, and `a` holds a promise.
3. When you print `a`, you see `<promise:a>`. This indicates that `a` is a promise object, and the actual computation (`(+ 3 5)`) has not been executed yet.

To force the evaluation and obtain the actual result, you can use the `force` procedure

```
(force(a))
; prints 8
```

tail recursion

```
(define naturals (letrec ((next (lambda (n) (cons n (delay (+ n 1)))))) (next 1)))
; "THATS ENOUGH SLICES" -kaitlyn
; building a very long list from 1 - n (natural numbers!)
naturals
; (1 . promise)
(cdr naturals)
; promise
(force (cdr naturals))
; (2 . promise)
; keep calling force and youll eventually get all the natural numbers
```

lazy squares

```
(define lazysquares (letrec ((next (lambda (n) (cons (* n n) (delay (next (+ n
1)))))))(next 1)))

lazysquares ; print 1
(force (cdr lazysquares)) ; print 4 . promise
```

lazy cubes

```
(define lazycubes (letrec ((next (lambda (n) (cons (* n (* n n)) (delay (next (+ n
1)))))))(next 1)))

(force (cdr lazycubes)) ; print 8
(force (cdr (force (cdr lazycubes)))) ; print 27
```

factorials

```
(define (myfac n)
     (if (= n 0)
         1
         (* n (myfac (- n 1)))))

(define (trfac n)
  (letrec ((helper (lambda (re m)
                (if (= m n) re
                    (helper (* re (+ m 1)) (+ m 1)) )))) (helper 1 0) )
  )

(myfac 5) ; prints 120
```

```
(define (trrev l)
  (letrec ((helper (lambda (l1 l2)
          (if (null? l1) l2
                  (helper (cdr l1) (cons (car l1) l2))) ))) (helper l '())))
```