```racket
#lang lazy

(define (try a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0)) ; no error because b is never used

; (try 1 (/ 1 0)) ; error because b is used

(define (our-if clause t f)
  (if clause
      t
      f))

(our-if (= 0 0)(printf "true!\n")(printf "false!\n"))

;; In lazy evaluation, expressions aren't evaluated until we nee them.
;; Instead, we get a 'promise' that the value will exist

(define numbers (list 1 2 3 4 5))
numbers   ; promise
(first numbers) ; value
numbers

(define (trace)
  (printf "Operation!\n"))

(define squares (map (lambda (x)(trace)(* x x)) (list 1 2 3 4)))
```

```scheme
squares
(first squares)
(first squares)  ; No operation performed here; result is remembered

(define (add x y)
  (printf "Addition!\n")
  (+ x y))

(define (subtract x y)
  (printf "Subtraction!\n")
  (- x y))

(define (multiply x y)
  (printf "Multiply!\n")
  (* x y))

(define (fac n)
  (if (= n 1)
      1
      (multiply n (fac (subtract n 1)))))

(fac 5)

(define (tail-fac n)
  (letrec ((helper (lambda (x res)
                   (if (= x 1)
                       res
                       (helper (subtract x 1)
                               (multiply x res))))))
    (helper n 1)))
```

```

(tail-fac 5)

(define (three x)
  3)

(define (loop-forever)
  (loop-forever))

(three (loop-forever))

;; Defining potentially infinite streams of
data

(define (plus-1 n)
  (cons n (plus-1 (+ n 1))))

(define pos-ints (plus-1 0))

pos-ints
(first pos-ints)
(second pos-ints)


```