let* is like ML's let: environment includes previous bindings
(let* [(x (+ x 3))
         (y (+ x 2))
         (z (+ y -5))])

(let* [(x (+ x 3))
         (y (+ x 2))
         (z (+ y -5))])

(letrec uses an environment where all bindings in scope
* you get #<undefined> if you use a variable before it's defined
  where as always function bodies not used until called
(bindings still evaluated in order)
(define (triple x)
  [(x (+ x 3))
   (y (+ x 2))
   (w (+ x 7))]
  (f -9)])

(define (mod2 x)
  [(even? (lambda (x) (if (zero? x) #t (odd? (- - x 1)))))
   (odd? (lambda (x) (if (zero? x) #f (even? (- - x 1)))))
   (if (even? x) 0)])

(define (bad-letrec-example x)
  [(y z) (letrec [(y z) (lambda (x) (if (zero? x) #t (odd? (- - x 1))^))]
    [(z 13)]
    (if x y z))])

; you can use define locally (in some positions)
; the same as letrec when binding local variables
(define (mod2_b x)
  [(define even? (lambda (x) (if (zero? x) #t (odd? (- - x 1)))))
   (define odd? (lambda (x) (if (zero? x) #f (even? (- - x 1)))))
   (if (even? x) 0)])

; at the top-level (*)
; same letrec-like rules: can have forward references, but
; definitions still evaluate in order and cannot be repeated
; (*) we are not actually at top-level -- we are in a module called lecl3.rkt
(define (f x) (+ x (* x y))) ; forward reference okay here
(define y 3)
(define z (+ y 4)); backward reference okay
(define w (+ v 4)); not okay (get an error)
(define v 5)
(define f 17); not okay: f already defined in this module

; what is c, d, and e?
(define b 3)
(define g (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define d (g 4))
(define e c)

; the truth about cons: it just makes a pair
(check-expect 1 (cons 1 2))
; how do these procedures differ?

(define accumulate-1
  (let ([acc 0])
    (lambda (x)
      (begin
        (set! acc (+ acc x))
        acc)))))

(define accumulate-2
  (lambda (x)
    (let ([acc 0])
      (begin
        (set! acc (+ acc x))
        acc)))))

; this distinction really matters even without mutation...
; this is a silly addition function that purposely runs slowly for
; demonstration purposes

(define (slow-add x y)
  (letrec ([slow-id (lambda (y z)
                       (if (= 0 z)
                           y
                           (slow-id y (- z 1))))]
            (+ (slow-id x 500000000) y))))

; multiplies x and result of y-thunk, calling y-thunk x times
(define (my-mult x y-thunk) ; assumes x is >= 0
  (cond [(= x 0) 0]
        [(= x 1) (y-thunk)]
        [#t (+ (y-thunk) (my-mult (- x 1) y-thunk))])))

; these calls: great for 0, okay for 1, bad for > 1
; (my-mult 0 (lambda () (slow-add 3 4)))
; (my-mult 1 (lambda () (slow-add 3 4)))
; (my-mult 2 (lambda () (slow-add 3 4)))

; these calls: okay for all
; (my-mult 0 (let ([x (slow-add 3 4)]) (lambda () x)))
; (my-mult 1 (let ([x (slow-add 3 4)]) (lambda () x)))
; (my-mult 2 (let ([x (slow-add 3 4)]) (lambda () x)))