```
#lang racket
(let ((a 1) (b 2))
  (+ a b))
;  ==> 3

(define a 10)
(define c 30)
(let ((a 1) (b 2))
  (+ a b c))
;  ==> 33

(define (silly1 z)
  (let ((x 5))
    (+ x z)))

; this one won't work!
(define (silly2 z)
  (let ((x 5) (answer (+ x z)))
    answer))

(define (silly2-fixed z)
  (let* ((x 5) (answer (+ x z)))
    answer))

(define (silly3 z)
  (let* ((x (if (> z 0) z 4)) (y (+ x 1)))
    (if (> x y) (* 2 x) (* y y))))

(define (silly4)
  (let ((x 1))
    (+
     (let ((x 2)) (+ x 1))
     (let ((y (+ x 2))) (+ y 1)))))

; Notice that the inner function define IS nested inside the outer define, but
; the body of the outer function (count-up 1 x) is NOT nested inside the inner d
efine.
; This is contrary to let/let*/letrec, where the body of the function is nested
; inside the let/let*/letrec.
(define (count-up-from-one-nested-define end)
  (define (count-up start end)
    (if (= start end)
        (cons start '())
        (cons start (count-up (+ 1 start) end))))
  (count-up 1 end))

(define (count-up-from-one-better end)
  (define (count-up start)
    (if (= start end)
        (cons start '())
        (cons start (count-up (+ 1 end)))))
  (count-up 1))
```

```
; max and repeated computation

(define (bad-max lst)
  (cond
    ((null? (cdr lst))
     (car lst))
    ((> (car lst) (bad-max (cdr lst)))
     (car lst))
    (#t
     (bad-max (cdr lst)))))

(define (good-max lst)
  (cond
    ((null? (cdr lst))
     (car lst))
    (#t
     (let ((max-of-cdr (good-max (cdr lst))))
       (if (> (car lst) max-of-cdr)
           (car lst)
           max-of-cdr)))))

; mutation

; Recall that sort-pair takes a pair and returns
; an equivalent pair so that car > cdr.

(define (sort-pair pair)
  (if (> (car pair) (cdr pair))
      pair
      (cons (cdr pair) (car pair))))

(define x '(4 . 3))
(define y (sort-pair x))
; Somehow mutate (car x) to hold 5
(define z (car y))

(define (sort-pair-alt pair)
  (if (> (car pair) (cdr pair))
      (cons (car pair) (cdr pair))
      (cons (cdr pair) (car pair))))

(define (my-append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))))

(define lst1 '(2 4))
(define lst2 '(5 3 0))
(define lst3 (append lst1 lst2))
```