2.17

```
(define (last-pair lyst)
  (if (equal? (length lyst) 1)
      lyst
      (last-pair (cdr lyst))))
```

2.18

#1

```
(define (reverse1 lyst)
  (define (reverse-helper newlyst iter-lyst)
    (if (null? iter-lyst)
        newlyst
        (reverse-helper (append (list (car iter-lyst)) newlyst) (cdr
iter-lyst))))
  (reverse-helper '() lyst))
```

#2

```
(define (reverse2 lyst)
  (define (reverse2-helper newlyst lyyst)
    (if (null? lyyst)
        newlyst
        (reverse2-helper (cons (car lyyst) newlyst) (cdr lyyst))))
  (reverse2-helper '() lyst))
```

2.20

```
(define (same-parity x . y)
  (if (odd? x)
      (cons x (filter1 odd? y))
      (cons x (filter1 even? y))))

(define (filter1 pred lyst)
  (if (null? lyst)
      '()
      (if (pred (car lyst))
          (cons (car lyst) (filter1 pred (cdr lyst)))
          (filter1 pred (cdr lyst)))))
```

2.21

```
(define (square-list items)
  (if (null? items)
      '()
      (cons (square (car items)) (square-list (cdr items)))))

(define (square-list2 items)
  (map square items))

(define (square x)
  (* x x))
```

2.22

#1

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

This gives us the answer in the reverse answer because cons adds it’s first element in front of the second element. In this case, as you go down the list, every element is being added in front of everything else. In other words, cons works from right to left rather than from left to right so the list turns out backwards.

#2

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

This also doesn’t work because cons’s first argument shouldn’t be a list unless you are making a pair. In this case, our arguments are flipped so the actual values are in the right order but they are made into pairs.

2.23

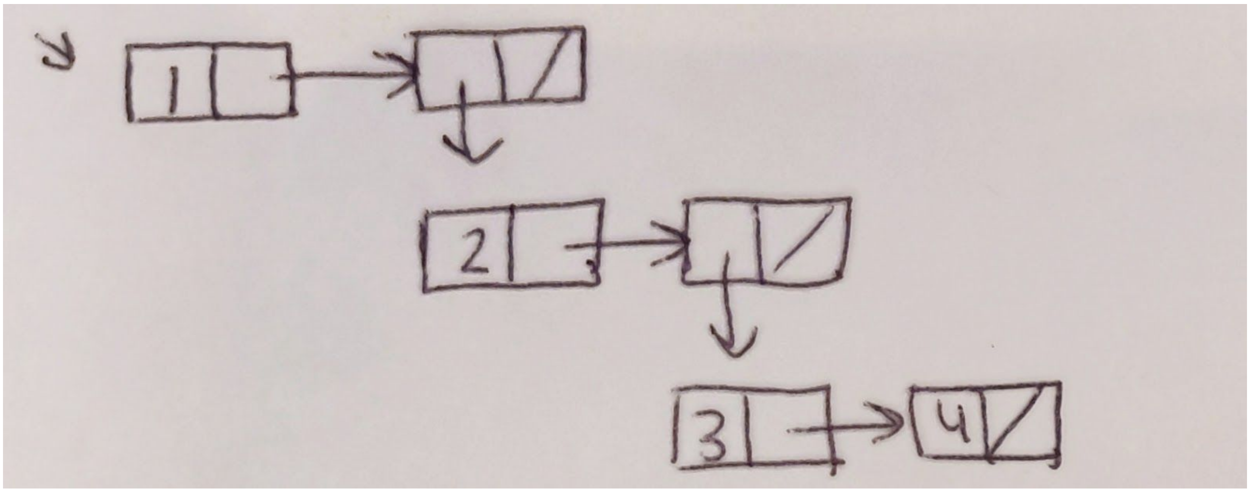
```
(define (for-each1 proc lyst)
  (if (null? lyst)
      #t
      (begin (proc (car lyst)) (for-each1 proc (cdr lyst)))))
```

2.24

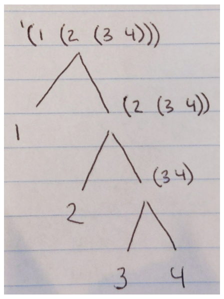
a)

‘(1 (2 (3 4)))

b)



c)



2.25

1) **cadaddr**
2) **caar**
3) **cadadadadadr**

2.26

(append x y): `(1 2 3 4 5 6)`

(cons x y): `((1 2 3) 4 5 6)`

(list x y): `((1 2 3) (4 5 6))`

2.27

Not Structured the best but it works :)

```
(define (deep-reverse lyst)
  (define (deep-helper newlyst lyyst)
    (if (null? lyyst)
        newlyst
        (deep-helper (cons (if (list? (car lyyst))
                               (deep-reverse (car lyyst))
                               (car lyyst))
                           newlyst)
                     (cdr lyyst))))
  (deep-helper '() lyst))
```

AVERAGE PROBLEM

```
(define (average x y . a)
  (define (helper sum lyst)
    (if (null? lyst)
        (* 1.0 (/ sum (+ 2 (length a))))
        (helper (+ sum (car lyst)) (cdr lyst))))
  (helper (+ x y) a))
```

SCVAL PROBLEM

```
(define (add n)
  (define (helper sum count)
    (if (= sum n)
        sum
        (if (> sum n)
            #f
            (helper (+ sum count) (+ count 1)))))
  (helper 0 1))

(define (multiply x)
  (* x (+ x 1) (+ x 2)))

(define (getnums x)
  (define (helper n lyst)
    (if (> n x)
        lyst
        (helper (+ n 1) (if (add (multiply n))
                            (cons n lyst)
                           lyst))))
  (helper 1 '()))

(define (make-lyst)
  (define (helper newlyst oldlyst)
    (if (null? oldlyst)
        newlyst
        (helper (cons (list (getn (multiply (car oldlyst))) (car
oldlyst) (multiply (car oldlyst))) newlyst) (cdr oldlyst))))
  (helper '() (getnums 636)))

(define (getn n)
  (define (helper sum count)
    (if (= sum n)
        (- count 1)
        (helper (+ sum count) (+ count 1))))
  (helper 0 1))
```