Examples:

1. `(car '(a b c))` ⇒ `a`
2. `(cdr '(a b c))` ⇒ `(b c)`
3. `(cdr '(a))` ⇒ `()`
4. `(car (cdr '(a b c)))` ⇒ `b`
5. `(cdr (cdr '(a b c)))` ⇒ `(c)`
6. `(car '((a b) (c d)))` ⇒ `(a b)`
7. `(cdr '((a b) (c d)))` ⇒ `((c d))`
8. `(cons 'a '())` ⇒ `(a)`
9. `(cons 'a '(b c))` ⇒ `(a b c)`
10. `(cons 'a (cons 'b (cons 'c '())))` ⇒ `(a b c)`
11. `(cons '(a b) '(c d))` ⇒ `((a b) c d)`
12. `(car (cons 'a '(b c)))` ⇒
13. `(cdr (cons 'a '(b c)))` ⇒ `(b c)`
14.
```
(cons (car '(a b c))
      (cdr '(d e f)))
```
⇒ `(a e f)`
15.
```
(cons (car '(a b c))
      (cdr '(a b c)))
```
⇒ `(a b c)`

16. Determine the values of the following expressions. Use your Scheme system to verify your answers.

a. `(cons 'car 'cdr)`
b. `(list 'this '(is silly))`
c. `(cons 'is '(this silly?))`
d. `(quote (+ 2 3))`
e. `(cons '+ '(2 3))`
f. `(car '(+ 2 3))`
g. `(cdr '(+ 2 3))`
h. `cons`
i. `(quote cons)`
j. `(quote (quote cons))`
k. `(car (quote (quote cons)))`
l. `(+ 2 3)`
m. `(+ '2 '3)`
n. `(+ (car '(2 3)) (car (cdr '(2 3))))`
o. `((car (list + - * /)) 2 3)`

Ans:

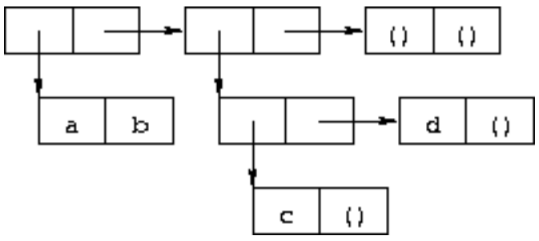
a. `(car . cdr)`
b. `(this (is silly))`
c. `(is this silly?)`
d. `(+ 2 3)`
e. `(+ 2 3)`
f. `+`
g. `(2 3)`
h. `#<procedure>`
i. `cons`
j. `'cons`
k. `quote`
l. `5`
m. `5`
n. `5`
o. `5`

17. `(car (car '((a b) (c d))))` yields `a`. Determine which compositions of `car` and `cdr` applied to `((a b) (c d))` yield `b`, `c`, and `d`.

```
(car (cdr (car '((a b) (c d))))) ⇒ b
(car (car (cdr '((a b) (c d))))) ⇒ c
(car (cdr (car (cdr '((a b) (c d)))))) ⇒ d
```

18.

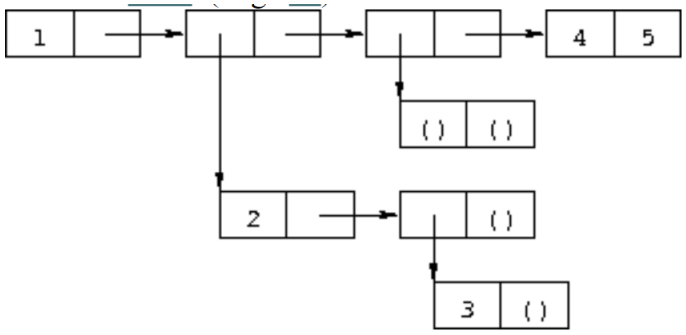
Write a Scheme expression that evaluates to the following internal list structure.



```
'((a . b) ((c) d) ())
```

19. Draw the internal list structure produced by the expression below.

```
(cons 1 (cons '(2 . ((3) . ())) (cons '(()) (cons 4 5))))
```



20. The behavior of `(car (car (car '((a b) (c d)))))` is undefined because `(car '((a b) (c d)))` is `(a b)`, `(car '(a b))` is `a`, and `(car 'a)` is undefined. Determine all legal compositions of `car` and `cdr` applied to `((a b) (c d))`.

```
(car '((a b) (c d))) ⇒ (a b)
(car (car '((a b) (c d)))) ⇒ a
(cdr (car '((a b) (c d)))) ⇒ (b)
(car (cdr (car '((a b) (c d))))) ⇒ b
(cdr (cdr (car '((a b) (c d))))) ⇒ ()
(cdr '((a b) (c d))) ⇒ ((c d))
(car (cdr '((a b) (c d)))) ⇒ (c d)
(car (car (cdr '((a b) (c d))))) ⇒ c
(cdr (car (cdr '((a b) (c d))))) ⇒ (d)
(car (cdr (car (cdr '((a b) (c d)))))) ⇒ d
(cdr (cdr (car (cdr '((a b) (c d)))))) ⇒ ()
(cdr (cdr '((a b) (c d)))) ⇒ ()
```

21. Rewrite the following expressions to give unique names to each different `let`-bound variable so that none of the variables is shadowed. Verify that the value of your expression is the same as that of the original expression.

a.
```
(let ((x 'a) (y 'b))
  (list (let ((x 'c)) (cons x y))
```

```
          (let ((y 'd)) (cons x y))))
```

b.
```
(let ((x '(a b) c)))
   (cons (let ((x (cdr x)))
           (car x))
         (let ((x (car x)))
           (cons (let ((x (cdr x)))
                   (car x))
                 (cons (let ((x (car x)))
                         x)
                       (cdr x))))))
```

Ans:

a.
```
(let ((x0 'a) (y0 'b))
   (list (let ((x1 'c)) (cons x1 y0))
         (let ((y1 'd)) (cons x0 y1))))
```

b.
```
(let ((x0 '(a b) c)))
   (cons (let ((x1 (cdr x0)))
           (car x1))
         (let ((x2 (car x0)))
           (cons (let ((x3 (cdr x2)))
                   (car x3))
                 (cons (let ((x4 (car x2)))
                         x4)
                       (cdr x2))))))
```

22.

```
(define cadr
  (lambda (x)
    (car (cdr x)))) or

(define (cadr x)
  (car (cdr x)))

(define cddr
  (lambda (x)
    (cdr (cdr x))))

(cadr '(a b c)) ⇒ b
(cddr '(a b c)) ⇒ (c)
```

23.

Definition of `shorter`, which returns the shorter of its two list arguments, or the first if the two have the same length. Write `shorter` without using `length`. [*Hint*: Define a recursive helper,`shorter?`, and use it in place of the length comparison.]

```
(define shorter?
  (lambda (ls1 ls2)
    (and (not (null? ls2))
```

```
          (or (null? ls1)
              (shorter? (cdr ls1) (cdr ls2))))))
```

```
(define shorter
  (lambda (ls1 ls2)
    (if (shorter? ls2 ls1)
        ls2
        ls1)))
```

24.

Use `map` to define a procedure, `transpose`, that takes a list of pairs and returns a pair of lists as follows.

`(transpose '((a . 1) (b . 2) (c . 3)))` ⇒ `((a b c) 1 2 3)`

[*Hint*: `((a b c) 1 2 3)` is the same as `((a b c) . (1 2 3))`.]

```
(define transpose
  (lambda (ls)
    (cons (map car ls) (map cdr ls))))
```

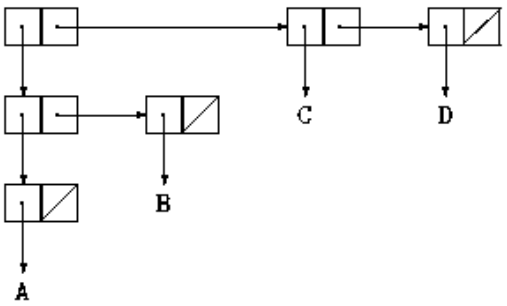
25.

```
(define colors '(red yellow ((orange) grey) ((blue) green)))

(car colors)

(cdr colors)

(cadr colors)

(caddr colors)

(cdddr colors)

(cdaddr colors)

(car (caaddr colors))

(cons 'cat '())

(cons '(cat mouse) '())

(define animals '(cat mouse))

animals

(cons 'bear animals)
```

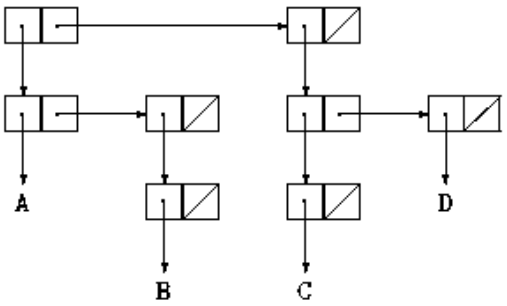
```
animals
 (car animals)
(cons '(bear lion) animals)
(append animals '(tiger giraffe))
animals
(append animals animals animals)
(define birds (list 'jay 'grackle 'eagle))
(caddr birds)
(list birds animals)
(append birds animals)
(list '(armadillo) birds)
(car (list '(armadillo) birds))
(define zoo (append animals birds))
(cadr zoo)
(car birds)
(reverse animals)
(car (reverse birds))
(eqv? 'a 'a)
 (+ (length animals) (length birds))
(= (+ (length animals) (length birds)) (length (append animals birds)))
```

Box Diagrams: Give the box diagrams for each of these:

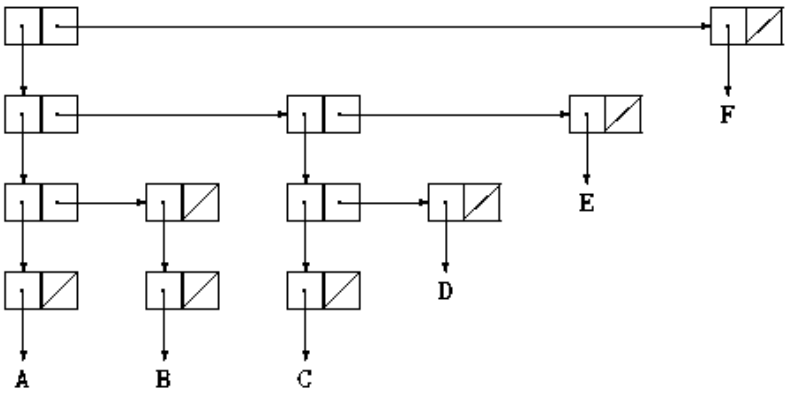
- `(((a) b) c d)`



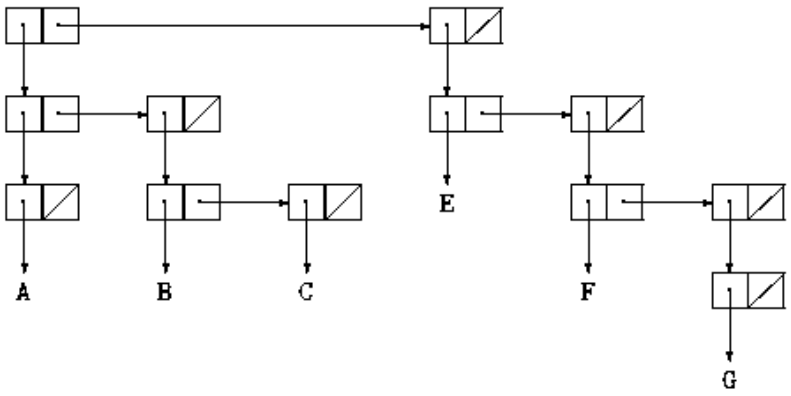
- `((a (b)) ((c) d))`



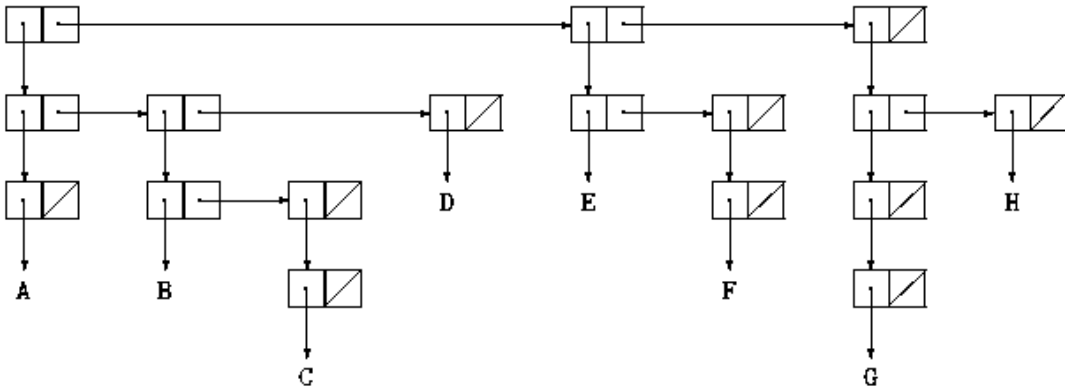
- `((((a) (b)) ((c) d) e) f)`



- `(((a) (b c)) (e (f (g))))`

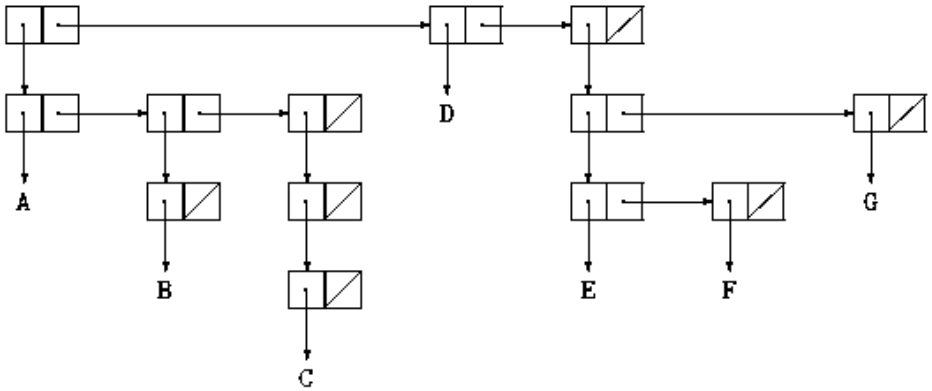


- `(((a) (b (c)) d) (e (f)) (((g)) h))`

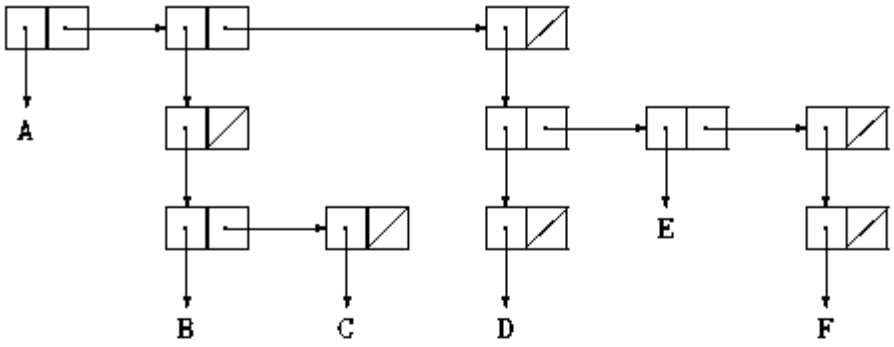


Box Diagrams: Give the list structure for each of these:

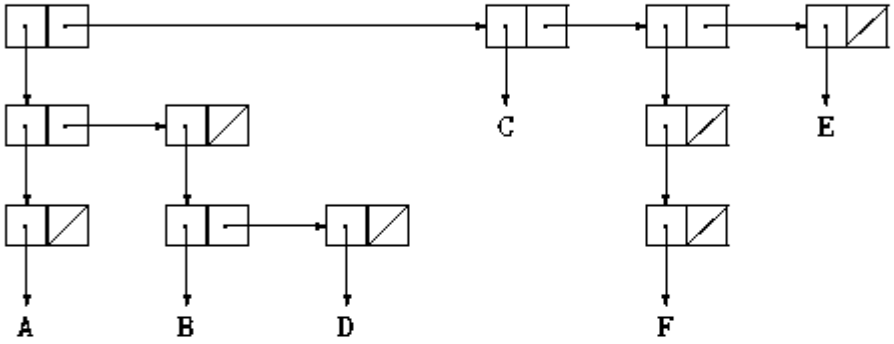
- `((a (b) ((c))) d ((e f) g))`



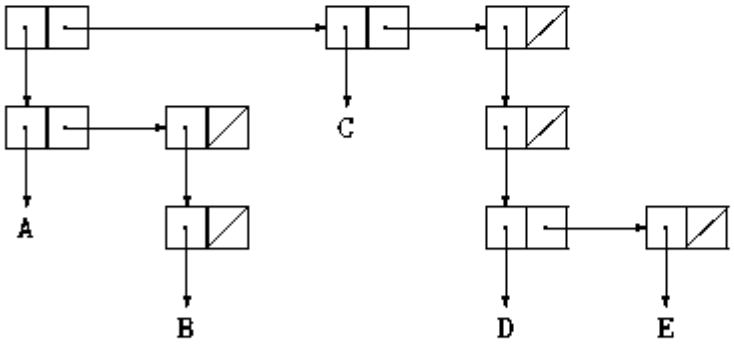
- (a ((b c)) ((d) e (f)))



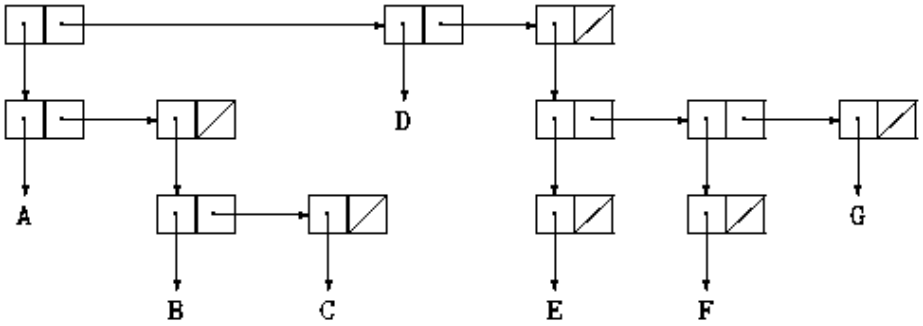
- (((a) (b d)) c ((f)) e)



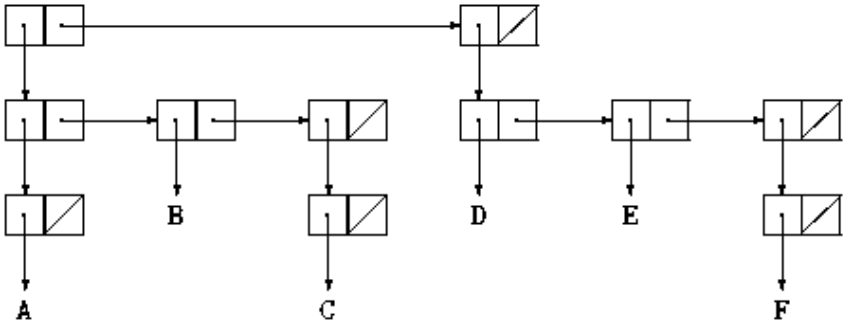
- ((a (b)) c ((d e)))



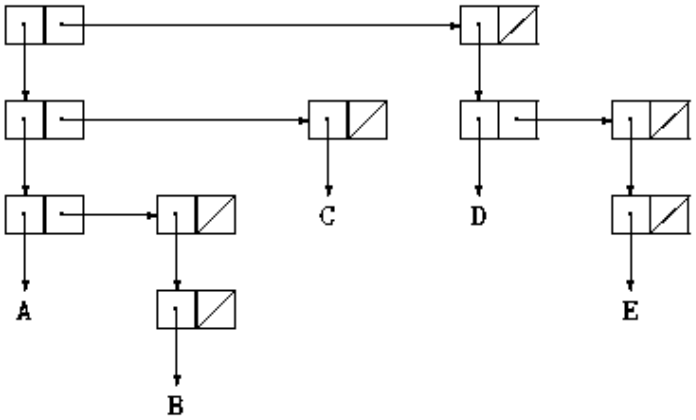
- ((a (b c)) d ((e) (f) g))



- (((a) b (c)) (d e (f)))



- (((a (b)) c) (d (e)))



27. Write a function that counts the number of symbols at the top level of a list `l`. Example: `(countsymbols '(a b 7 (peas) #f carrots))` = `3`

```
(define (countsymbols l)
  (if (null? l)
      0
      (+ (if (symbol? (car l)) 1 0)
         (countsymbols (cdr l)) ) ) )
```

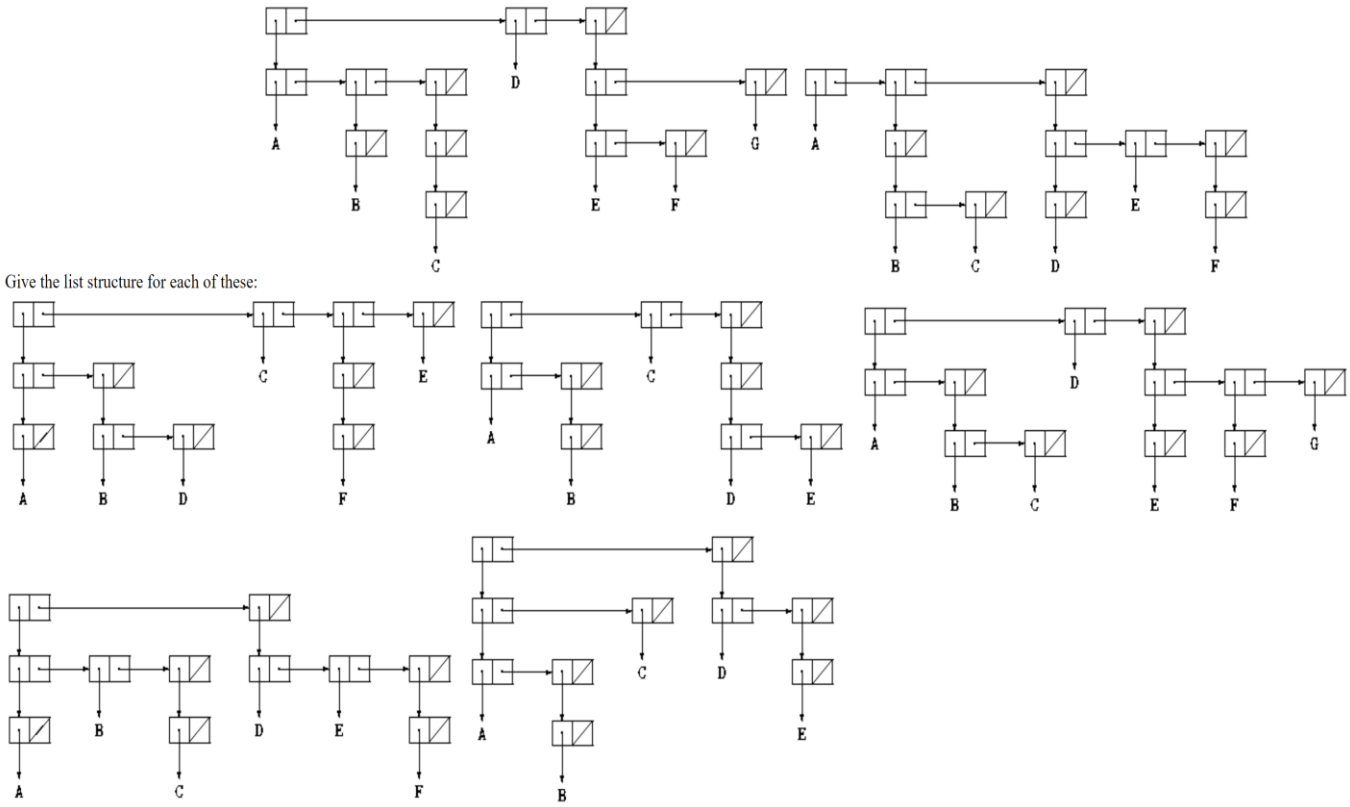
28.

Give the box diagrams for each of these:

```
(((a) b) c d)

((a (b)) ((c) d))

((((a) (b)) ((c) d) e) f)

(((a) (b c)) (e (f (g))))

(((a) (b (c)) d) (e (f)) (((g)) h))
```

29.

Box Diagrams:



Give the list structure for each of these:

30.

Given as input a list of numbers, add up the squares of the numbers.

```
(define (sumsq l)
  (if (null? l)
      0
      (+ (expt (car l) 2)
         (sumsq (cdr l)) ) ) )
```

31. Given as input a list structure (tree) containing some numbers (and possibly other things), write a function to add up the cubes of all the numbers.

```
(define (cubes x)
  (if (pair? x)
      (+ (cubes (car x))
         (cubes (cdr x)))
      (if (number? x)
          (expt x 3)
          0) ) )
```

32. Write a function that will return the sum of the even numbers in a list structure (tree) that may contain things that are not numbers.

```
(define (evensum x)
  (if (pair? x)
      (+ (evensum (car x))
         (evensum (cdr x)))
      (if (and (number? x) (even? x))
          x
          0) ) )
```

33. Write a function remove-k that removes the kth element from a given list. (remove-k '(0 1 2 3 4 5) 4) ---> (0 1 2 3 5)

```
(define (remove-k lst k)
(cond ((= k 0) (cdr lst))
        (else (cons (car lst) (remove-k (cdr lst) (- k 1))))))
```

34. Define a procedure list-4 that takes in 4 elements and outputs a list equivalent to one created by calling list.

```
(define (list-4 e1 e2 e3 e4) (cons e1 (cons e2 (cons e3 (cons e4 '()))))
```

35. Define a procedure list? that takes in something and returns #t if it's a list, #f otherwise.

```
(define (list? ls) (or (null? ls) (and (pair? ls) (list? (cdr ls)))))
```