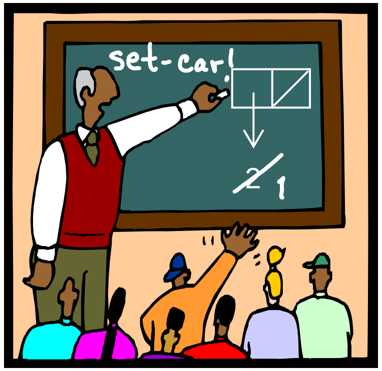
6.001 recitation 3/23/07

- from last time: set-car!, set-cdr!
- trees



Dr. Kimberle Koile

more set-car! and set-cdr! problems

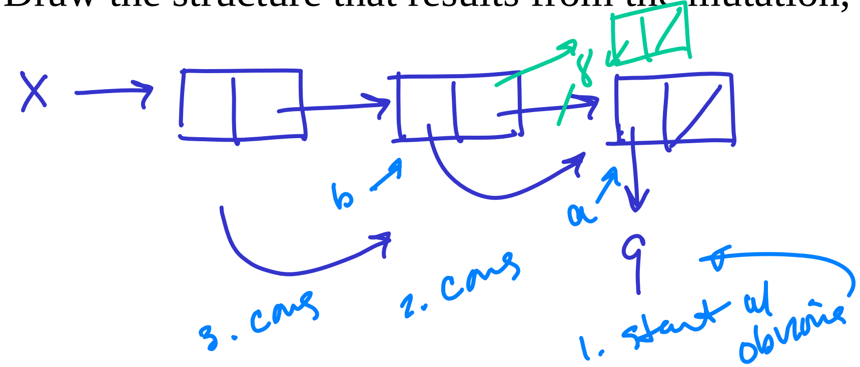
For the box & pointer diagram:

(a) Write what Scheme prints out for the structure (if it can)

(b) Write a Scheme expression that makes the structure(if an error, describe it)

(c) Draw the structure that results from the mutation, and its printed representation.

1.



a. x =>

(((9) 9) (9) 9)

- start w/ # elts

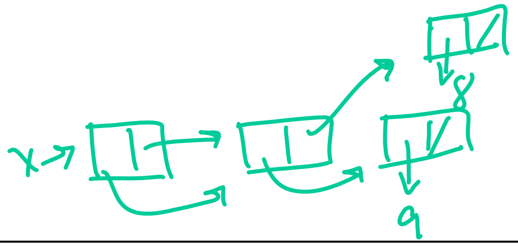
3 elts

b. Scheme expression:

```
(define x
      (let ((a '(9)))  1.
or let*   (let 4.((b (cons a a)))  2.
              (cons b b))))  3.

or (define x
    (let ((a (list 9 9 9)))
      (set-car! (car a) (cddr a))
      (set-car! a (cdr a))
   a))
```

can be anything

c. mutation: (set-cdr! (car x) '(8))



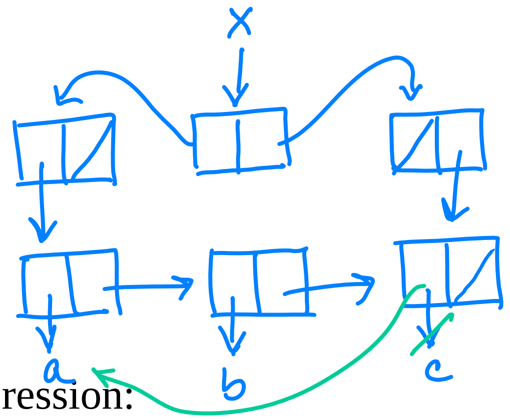
x => (((9) 8) (9) 8)

more set-car! and set-cdr! problems

* ptr can point to any part of box

For the box & pointer diagram:
(a) Write what Scheme prints out for the structure (if it can)
(b) Write a Scheme expression that makes the structure (if an error, describe it)
(c) Draw the structure that results from the mutation, and its printed representation.

2.



a. x =>

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

b. Scheme expression:

```
(define x
    (let ((w '(a b c)))
        (cons (list w)
              (cons '() (cddr w)))))
```

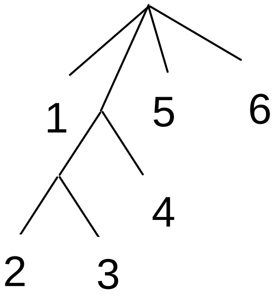
c. mutation: (set-cdr! (cddr x) (caaar x))

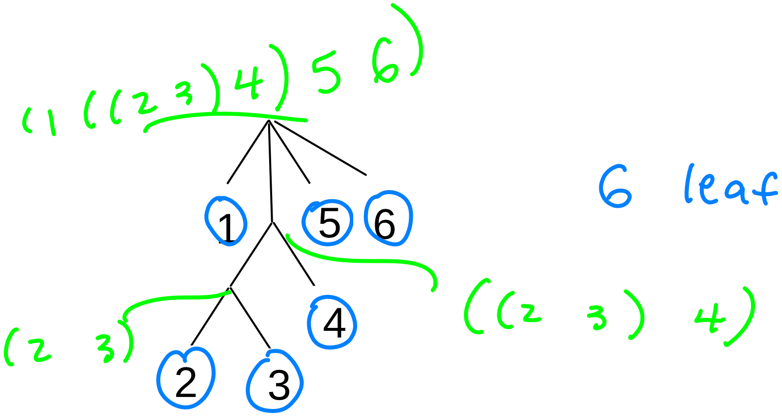
x =>

(((a b a)) () a)

trees

- **A tree is a nested list; each node is a list of the children of that node**
- **A child is either another tree or a leaf node**
  - A child that is a tree is called a *subtree*
  - A leaf node is anything that is not a pair (i.e., a symbol or a self-evaluating value).

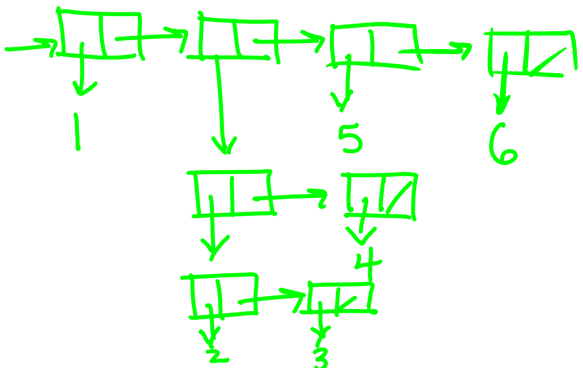




6 leaf nodes

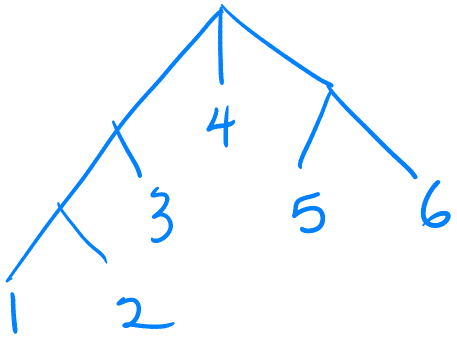
length? 4

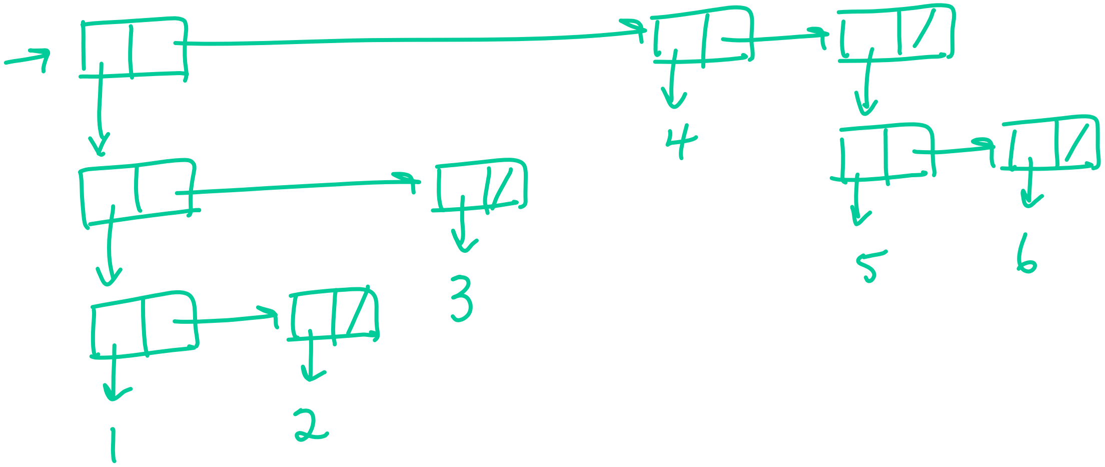
leaves? 6



tree representation

1. Draw a box-and-pointer structure for the following tree. How does the interpreter print this structure?



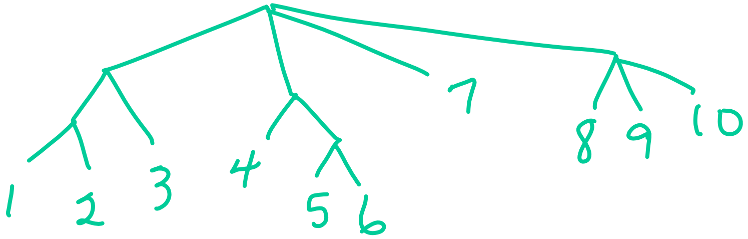


box&pointer

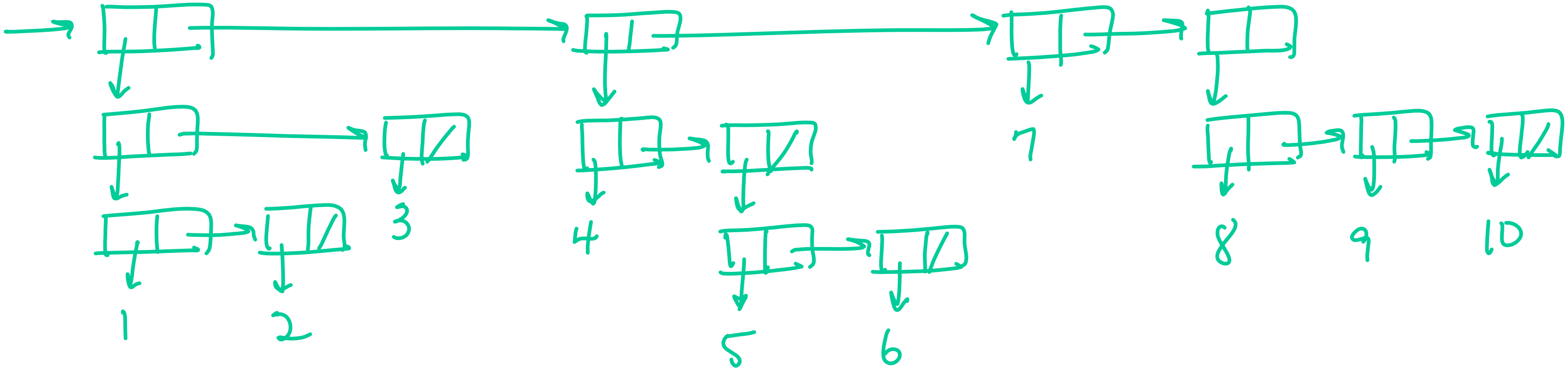
printed representation $(\underbrace{((1\ \ 2)\ \ 3)}\ \ \underbrace{4}\ \ \underbrace{(5\ \ 6)})$

tree representation

2a. Draw the interpretation of this list as a tree structure: $(\underbrace{((1\ 2)\ 3)}\ \underbrace{(4\ (5\ 6)}\ \underbrace{7}\ \underbrace{(8\ 9\ 10)})$)



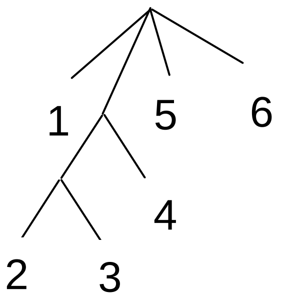
2b. Draw the box-and-pointer diagram.

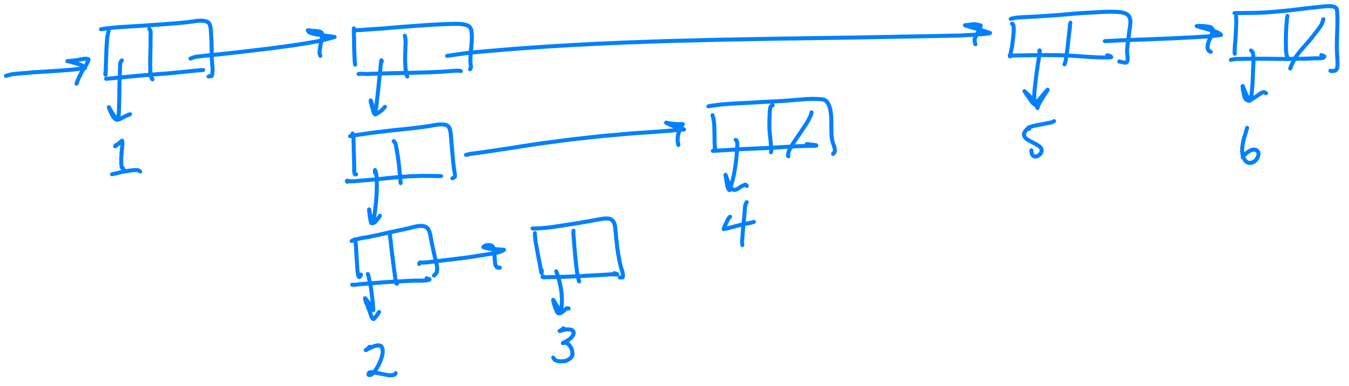


counting leaves

```
(define (countleaves tree)
    (cond ((null? tree) 0)
              ((leaf? tree) 1)
              (else (+ (countleaves (car tree))
                         (countleaves (cdr tree))))))


(define (leaf? x)
    (not (pair? x)))
```

(1 ((2 3) 4) 5 6)

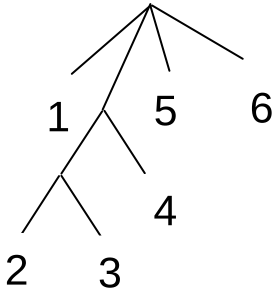




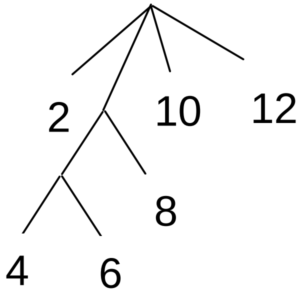
doubling a tree: version 1

```
(define (countleaves tree)
   (cond ((null? tree) 0)
         ((leaf? tree) 1)
         (else (+ (countleaves (car tree))
                  (countleaves (cdr tree))))))

(define (leaf? x)
   (not (pair? x)))
```

```
(define (double-tree tree)
   (cond ((null? tree) ‘())
         ((leaf? tree)  (* 2 tree)  )

         (else (  cons  (  double-tree  (car tree))
                       (  double-tree  (cdr tree))))))
```



=>



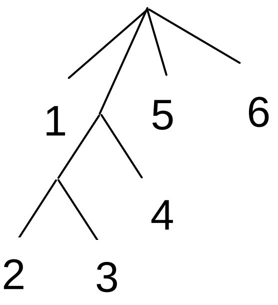
(1 ((2 3) 4) 5 6)

(2 ((4 6) 8) 10 12)

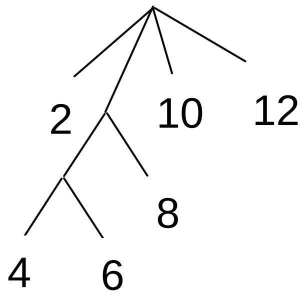
doubling a tree: version 2, map

```
v. 1 (define (double-tree tree)
    (cond ((null? tree) ‘())  (* 2 tree)
          ((leaf? tree)    double-tree        )

          (else (  cons    (  double-tree      (car tree))

                   (                          (cdr tree))))))
```

```
v. 2 (define (double-tree tree)
       (if  (leaf? tree)
            (  (* 2 tree)  )
            (  (map double-tree tree)  )))
```



=>



(1 ((2 3) 4) 5 6)

(2 ((4 6) 8) 10 12)

doubling a tree: version 3, map-tree

```
v. 1 (define (double-tree tree)
        (cond ((null? tree) '())
              ((leaf? tree)     (* 2 tree)     )
              (else (   cons  (double-tree    (car tree))
                       (    double-tree      (cdr tree))))))
```

```
v. 2 (define (double-tree tree)
        (if  (leaf? tree)
             (     * 2 tree      )
             (     map  double-tree tree       )))
```

```
(define (map-tree proc tree)
   (if  (leaf? tree)
        (proc tree)
        ~~(map-tree proc  tree )))~~
        (map (λ (tree) (map-tree proc tree))
              tree)
```

or
(cons (map-tree proc (car tree))
(map-tree proc (cdr tree)))

why need λ?
b/c arg mismatch:
proc passed to map
takes 1 arg, not 2

```
(define (double x)
   (*  2  x))
```

```
(define (double-tree tree)
   (map-tree double tree)
)
```