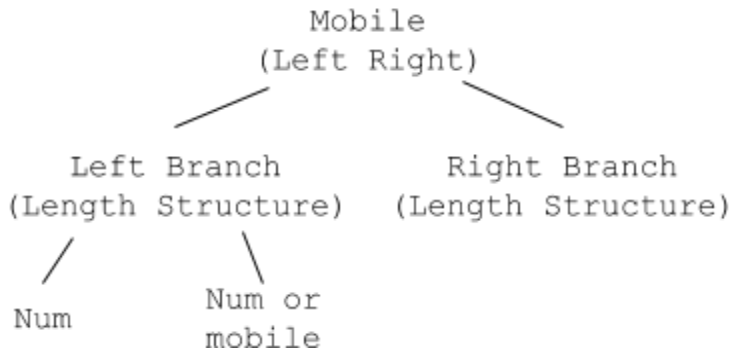
Hannah Zhang

2.29, 2.30, 2.31, 2.32, 2.33, 2.35

(2.33 need to review)

; prompt
; comments
; answers

2.29



a. Write the corresponding selectors left-branch and right-branch, and branch-length and branch-structure.

```
; get the input right so abstraction happens
(define (left-branch mobile)
  (car mobile))

(define (right-branch mobile)
  (car (cdr mobile)))

(define (branch-length branch)
  (car branch))

(define (branch-structure branch)
  (car (cdr branch)))
```

b. Using your selectors, define a procedure total-weight that returns the total weight of a mobile.

```
; the thinking is similar to that of deep-reverse
; if it is empty → add 0
; if it is a number → add the number
; if it is a list → do recursion until it becomes a number
```

```
; important note that weight is all the structures added
together (does not include length)
(define (total-weight mobile)
  (cond ((empty? mobile) 0)
        ((number? mobile) mobile)
        (else (+ (total-weight (branch-structure (left-branch
mobile)))
                 (total-weight (branch-structure (right-branch
mobile)))))))
```

c. Design a predicate that tests whether a binary mobile is balanced.

```
;I was thinking to test if torque is true
```

- **If it is a weight, return true**
- If true, test if there are submobiles
  - If there are no submobiles, return true (structure must be the same since torque is true)
  - If the structure of the submobiles are balanced, true
  - If not, return false
- If torque is not true, return false

```
(define (torque branch)
  (* (branch-length branch) (total-weight (branch-structure
branch))))

(define (balanced? mobile)
  (if (= (torque (left-branch mobile)) (torque (right-branch
mobile)))
      (cond ((number? mobile) #t)
            ((and (balanced? (branch-structure (left-branch
mobile)))
                  (balanced? (branch-structure (right-branch
mobile)))) #t)
            (else #f))
      #f))
```

d. How much do you need to change your programs to convert to the new representation?

- If the change the representation to cons instead of list, we do not need to change a lot
- All we need to change is the abstraction part and the other functions stay the same

2.30

Define a procedure square-tree analogous to the square-list procedure

```
; the thinking is very similar to deep-reverse
; if the list is empty, return empty
; if the list doesn't have a sublist, square the number directly
; if the list has a sublist, call the function on car and cdr and cons it together
```

- This gets rid of sublists, even if there are multiple

Directly

```
(define (square-tree lyst)
  (cond ((empty? lyst) null)
        ((not (list? lyst)) (square lyst))
        (else (cons (square-tree (car lyst)) (square-tree (cdr lyst))))))
```

Map & Recursion

```
; map takes a procedure and a list
; define a helper function that is the same as the direct version
; use helper function in map
(define (helper lyst)
  (cond ((empty? lyst) null)
        ((not (list? lyst)) (square lyst))
        (else (cons (helper (car lyst)) (helper (cdr lyst))))))

(define (mst lyst2)
  (map helper lyst2))

; without a helper function using lambda
; note the common pattern to 2.35
(define (mst lyst)
  (map (lambda (x) (cond ((empty? x) '())
                         ((number? x) (square x))
                         (else (mst x)))) lyst))
```

2.31

Abstract your answer to 2.30 to produce a procedure tree-map

```
; this abstraction makes it easier to change the proc to not be
limited to just square
(define (square-tree tree) (tree-map square tree))
(define (tree-map proc lyst)
  (cond ((empty? lyst) '())
        ((not (list? lyst)) (proc lyst))
        (else (cons (tree-map proc (car lyst)) (tree-map proc
(cdr lyst))))))
```

2.32

Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works

```
; how subsets works
```

- The set of all subsets excluding the first number
  - (subsets (cdr s))
- The set of all subsets excluding the first number, with the first number re-inserted into each subset
  - (cons (car s) subset) for each member of subset

```
; the part to fill in is the second point, adding the first of
the list to each member of subset
```

- (cons (car s) rest)
- Must use lambda
- The input to x is each member

```
(define (subsets s)
  (if (null? s)
      (list null)
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda (x) (cons (car s) x))
rest)))))
```

2.33

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations

```
; map, applies p to every element in sequence
; in lambda, x is each element (changes each time) and y is the
rest of the sequence after x is taken
; therefore we must apply p to every element (which is x)
```

```
(define (map1 p sequence)
  (accumulate1 (lambda (x y) (p x)) null sequence))
```

Or

```
; use both x and y through cons
; this adds the element with the procedure applied to the list,
and then moves on to the next element
(define (map1 p sequence)
  (accumulate1 (lambda (x y) (cons (p x) y)) null sequence))

; append, combines the two lists given into one list
; in this case op → cons, id → seq2, sequence → seq1
(define (append1 seq1 seq2)
  (accumulate1 cons seq2 seq1))
; how this works: end condition is second sequence and keep on
cons each member of the first list to the second sequence
```

Trace:

```
(append1 '(1 2 3) '(4 5 6))
> (cons 1 (accumulate1 cons (4 5 6) (2 3)))
> > (cons 1 (cons 2 (accumulate1 cons (4 5 6) (3))))
> > > (cons 1 (cons 2 (cons 3 (accumulate1 cons (4 5 6) ()))))
> > > > (cons 1 (cons 2 (cons 3 (4 5 6))))
> > > (cons 1 (cons 2 (3 4 5 6)))
> > (cons 1 (2 3 4 5 6))
> (1 2 3 4 5 6)

; length, find how many elements in a list
; it creates a new instance for each element in the list
; then simplifies by adding one for each element/instance
(define (length1 sequence)
  (accumulate1 (lambda (x y) (+ y 1)) 0 sequence))
```

Trace:

```
(length1 '(1 2 (3 4)))
```

- Let lam mean (lambda (x y) (+ y 1))

```
> (lam 1 (accumulate1 lam 0 (2 (3 4))))
> > (lam 1 (lam 2 (accumulate1 lam 0 (3 4))))
> > > (lam 1 (lam 2 (lam (3 4) (accumulate1 lam 0 ()))))
> > > > (lam 1 (lam 2 (lam (3 4) 0)))
> > > (lam 1 (lam 2 1))
> > (lam 1 2)
```

```
> 3
```

2.35

```
Redefine count-leaves as an accumulation
; count the number of actual numbers (not list) in a list
; take into account sublists

; op should be +
; initial should be 0
; the map should test if there are sublists, if so do recursion,
if not add 1
(define (count-leaves t)
  (accumulate1 + 0 (map helper t)))

(define (helper x)
  (cond ((empty? x) 0)
        ((not (list? x)) 1)
        (else (+ (helper (car x)) (helper (cdr x))))))
OR
; all in one function using lambda
; the key part is instead of calling (helper (car x)) and
(helper (cdr x)), we just call (count-leaves x) in the else
clause
; this gets rid of one sublist and runs everything over again
(define (count-leaves3 t)
  (accumulate1 + 0 (map (lambda (x) (cond ((empty? x) 0)
                                          ((not (list? x)) 1)
                                          (else (+
(count-leaves3 x)))))
                        t)))
```