Reduce Problem

```
(define (reduce op lyst)
  (cond ((= (length lyst) 1) (car lyst))
        ((null? lyst) '())
        (else (reduce op (cons (op (car lyst) (cadr lyst))
                               (cddr lyst))))))
```

2.29

a)

```
(define (make-mobile left right)
  (list left right))

(define (make-branch length structure)
  (list length structure))

(define (left-branch mobile)
  (car mobile))

(define (right-branch mobile)
  (cadr mobile))

(define (structure branch)
  (cadr branch))
```

b)

```
(define (total-weight mobile)
  (if (pair? mobile)
      (+ (branch-weight (left-branch mobile))
         (branch-weight (right-branch mobile)))
      mobile))

(define (branch-weight branch)
  (if (weight? (structure branch))
      (structure branch)
      (total-weight (structure branch))))

(define weight? number?)
```

c)

```
(define (len branch)
  (car branch))

(define (torque branch)
  (* (len branch) (total-weight (structure branch))))

(define (branch-balanced? branch)
   (if (pair? (structure branch))
        (balanced? (structure branch))
        #t))

(define (balanced? mobile)
   (and (= (torque (left-branch mobile))
           (torque (right-branch mobile)))
         (branch-balanced? (left-branch mobile))
         (branch-balanced? (right-branch mobile))))
```

d)

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))

(define (left-branch mobile)
  (car mobile))

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

(define (structure branch)
  (cdr branch))

(define (len branch)
  (car branch))
```

We only have to change the simple constructors because this is where the abstraction lies. Instead of having cadr to get the first of the butfirst, we only need cdr now because it is a pair.

2.30

#1

```
(define (square-tree lyst)
  (cond ((= (length lyst) 0) '())
        ((list? (car lyst)) (cons (square-tree (car lyst))
                                  (square-tree (cdr lyst))))
        (else (cons (square (car lyst))
                    (square-tree (cdr lyst))))))

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

#2

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

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

With Map:

```
(define (square-tree lyst)
  (map (lambda (x)
         (if (list? x)
             (square-tree x)
             (square x)))
       lyst))

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

2.31

#1

```
(define (tree-map op lyst)
  (cond ((null? lyst) '())
        ((not (list? lyst)) (op lyst))
        (else (cons (tree-map op (car lyst))
                    (tree-map op (cdr lyst))))))

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

#2

```
(define (tree-map op lyst)
  (cond ((null? lyst) '())
        ((list? (car lyst)) (cons (tree-map op (car lyst))
                                  (tree-map op (cdr lyst))))
        (else (cons (op (car lyst)) (tree-map op (cdr lyst))))))
```

2.32

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

2.33

```
(define (map2 p sequence)
  (accumulate (lambda (x y) (cons (p x) y)) '() sequence))

(define (append2 seq1 seq2)
  (accumulate2 cons seq2 seq1))

(define (length2 sequence)
  (accumulate (lambda (x y) (+ 1 y)) 0 sequence))
```

2.35

```
define (count-leaves t)
  (accumulate2 + 0 (map (lambda (x)
                          (if (number? x)
                              1
                              (count-leaves x)))
                        t)))
```