;; Procedures for operating on trees
;; by Holly Yanco, retrofit by Mark Sherman September 26, 2013
(define nil '())
;; define a couple of trees for testing
(define tree1 (cons (list 1 2) (list 3 4)))
(define tree2 (list (list 1 2)
(list 3 (list 4 5 6))
(list 7 8)))

;; procedure to count the leaves of a tree
(define (count-leaves tree)
  (cond ((null? tree) 0)
    ((not (pair? tree)) 1)
    (else (+ (count-leaves (car tree))
             (count-leaves (cdr tree))))))

(count-leaves tree1)
; Value: 4
(count-leaves tree2)
; Value: 8

;; fringe returns a list of the leaves of the tree
(define (fringe tree)
  (cond ((null? tree) nil)
    ((not (pair? tree)) (list tree))
    (else (append (fringe (car tree))
                  (fringe (cdr tree))))))

(fringe tree1)
; Value: (1 2 3 4)
(fringe tree2)
; Value: (1 2 3 4 5 6 7 8)

;; sum-fringe returns the sum of the leaves of a tree
(define (sum-fringe tree)
  (cond ((null? tree) 0)
    ((number? tree) tree)
    (else (+ (sum-fringe (car tree))
             (sum-fringe (cdr tree))))))

(sum-fringe tree1)
; Value: 10
(sum-fringe tree2)
; Value: 36

;; accumulate-tree completely captures a general pattern in tree accumulation
(define (accumulate-tree tree term combiner null-value)
  (cond ((null? tree) null-value)
    ((not (pair? tree)) (term tree))
    (else (combiner (accumulate-tree (car tree) term combiner null-value)
                    (accumulate-tree (cdr tree) term combiner null-value)))))

;; We could use this on-the-fly to work like sum-fringe
(accumulate-tree tree1 (lambda (x) x) + 0)
; Value: 10
(accumulate-tree tree2 (lambda (x) x) + 0)
; Value: 36

;; We can also use accumulate-tree to make count-leaves
;; (this replaces the previous definition of count-leaves)
(define (sum-fringe tree)
  (accumulate-tree tree
                   (lambda (x) x)
                   +
                   0))

(count-leaves tree1)
; Value: 4
(count-leaves tree2)
; Value: 8

;; use accumulate-tree to make count-leaves
;; (replaces previous definition of count-leaves)
(define (count-leaves tree)
  (accumulate-tree tree
                   (lambda (x) 1)
                   +
                   0))

(fringe tree1)
; Value: (1 2 3 4)
(fringe tree2)
; Value: (1 2 3 4 5 6 7 8)
;;; map-tree applies a procedure to each leaf of the tree
define (map-tree tree op)
  (cond ((null? tree) nil)
          ((number? tree) (op tree))
          (else (cons (map-tree (car tree) op)
                       (map-tree (cdr tree) op))))

;;; write increment-tree using map-tree
(define (increment-tree tree)
  (map-tree tree inc))

;;; write scale-tree using map-tree (multiplies each leaf by a factor)
define (scale-tree tree factor)
  (map-tree tree (lambda (x) (* x factor))))

;;; copy-tree makes a new copy of the tree
define (copy-tree tree)
  (cond ((null? tree) nil)
          ((not (pair? tree)) tree)
          (else (cons (copy-tree (car tree))
                       (copy-tree (cdr tree))))))

;;; procedure to count the maximum depth of the tree
define (max-depth tree)
  (cond ((null? tree) 0)
          ((not (pair? tree)) 1)
          (else (max (max-depth (car tree))
                     (max-depth (cdr tree))))))

;;; flattens one layer of a tree
;;; Takes a list, returns a new list that contains all
;;; the items of the input list (which may contain sub-lists)
define (flatten tree)
  (cond ((null? tree) nil)
          ((not (pair? tree)) (list tree))
          (else (append (list (car tree))
                        (flatten (cdr tree)))))

;;; Just for fun, let's scale a tree and get the fringe
(fringe (scale-tree tree2 10))
; Value: (10 20 30 40 50 60 70 80)