Problem 1

Consider the following data definition:

```racket
(define-struct node [value left right])
```

; A BinaryTree (BT) is either:
; - '()
; - (make-node Number BT BT)

A binary tree is like our original rumor mill, except it contains node structures instead of gossiping person structures.

a. The following figure depicts a binary tree, where the circles represent node structures and the lines below each circle represent the left and right binary trees for each node. (Nodes 4, 5, 6, and 7 have '()' left and right fields, which are not shown in the figure.)

![Binary Tree Diagram]

```
Using the data definition for BinaryTree, translate this figure into its Racket representation.
```

**Hint:** It's easiest to built the tree one level at a time, starting from the bottom. That is, define names for nodes 4, 5, 6, and 7 first, then use those names for defining 2 and 3, etc.
b. Design the function `sum-odds`. It consumes a binary tree and produces the sum of all the odd numbers in the tree.

You should use the existing function `odd?` in your solution.

*Follow the design recipe step by step.* For full credit, be sure to include a template `bt-temp`, signature/purpose statement/header, examples (you should have at least two, based on the data definition for `BT`), and, last but not least, the function body.
Problem 2

Consider the following data definitions:

(define-struct tweet [author text])
(define-struct image-tweet [author text image])
(define-struct retweet [author tweet])

;; A Tweet is one of:
;; - (make-tweet String String)
;; - (make-image-tweet String String Image)
;; - (make-retweet String Tweet)
;; Interpretation: A tweet is either a message (make-tweet), a message and
;; an image (make-image-tweet), or a retweet of another tweet
;; (make-retweet). All tweets have authors.

;; A Feed is a [List-of Tweet]
;; Interpretation: A list of the tweets in a user's feed.

a. Without using the list abstractions (like map, filter, etc.), design the
function count-tweets, which accepts a Feed and returns a Number
representing the total number of Tweets in the Feed. Note that a retweet
has more than one tweet in it. For example,

(check-expect
count-tweets
(list (make-tweet "foo" "bar")
  (make-retweet "baz"
    (make-retweet "blah"
      (make-image-tweet "arg" "grr"
        empty-image))))

4)
b. Redesign the function `count-tweets` *using* any appropriate list abstractions (`map`, `filter`, etc.) You may use any helper functions that you designed for the previous part.
Problem 3

Recall the intertwined definitions for our “more realistic rumor mill”:

;; A ListOfGossip is either
;; - '()
;; - (cons Gossip ListOfGossip)

(define-struct gossip [who nexts])

;; A Gossip is
;; (make-gossip Image ListOfGossip)

Using these definitions, design remove-person-from-gossip, which takes a person image and a gossip and returns a gossip where the given person is uninformed.
Problem 4

Given the now familiar data definition for a list of numbers:

```
;; A ListOfNumbers is either:
;; - ()
;; - (cons Number ListOfNumbers)
```

and the template we saw in Lecture 21 for a function that processes two lists of the same size in parallel:

```
(define (lons-temp l1 l2)
  (cond [(empty? l1) ...]
        [(cons? l1)
         (... (first l1) ...
              (first l2) ...
              (lons-temp (rest l1) (rest l2)) ...))
)
```

Design a function named `any=?` that consumes two lists of numbers (of the same length) and determines where any pair of numbers in corresponding positions (e.g., the first number in both lists, the second number in both lists, etc.) are equal. Follow the design recipe, but there’s no need to include a template (as it’s above). Your solution should include a signature, purpose statement, examples (in the form of at least three `check-expects`), and function body.
Problem 5

Complete the definition for the following function using the accumulator method:

;;; count-satisfying : [List-of Number], [Number -> Boolean] -> Number
;;; Count how many numbers in a list satisfy a given predicate.

(check-expect (count-satisfying (list 1 2 3 4 5 6) even?) 3)
(check-expect (count-satisfying (list 6 1 3 2 5 4) even?) 3)
(check-expect (count-satisfying (list -2 1 -1 2) positive?) 2)
(check-expect (count-satisfying (list -2 1 -1 2) zero?) 0)

(define (count-satisfying lon p?)