1. (10 points) Consider the two trees given below. Assume that each tree is constructed from bnodes and that empty nodes are not shown.

(a) (3 points) Divide the node names (e.g., 40, 29, etc.) in TreeA into three categories (note that a node may appear in more than one category):

- Root: 40
- Internal nodes: 29 and 46
- Leaves: 11, 35, 52, and 75

(b) (1 points) Which of the two trees above is a binary tree? Answer either TreeA, TreeB, or both.

Both TreeA and TreeB are binary trees.

(c) (3 points) At least one of the trees above is a binary search tree (BST). Which of them is a BST? Answer either TreeA, TreeB or both. Explain your answer. If one of them is not a BST, state (in English, not code) how the tree could be modified to make it a BST or change the contents of the nodes in the pictures.

TreeB is a binary search tree.

TreeA is not a binary search tree. The binary search tree property is violated in TreeA because the node with key 52 is the left child of the node with key 46.

(d) (3 points) Insert the following nodes into whichever tree is a binary search tree (TreeA or TreeB) using the insertBST function: 28, 45, and 66, in that order. Draw the tree after the three new nodes have been inserted. You can draw the new nodes directly on one of the trees above, or you can redraw the binary search tree below.

Key 28 is inserted into TreeB as the right child of the node with key 11. Key 45 is inserted into TreeB as the left child of the node with key 52. Key 66 is inserted into TreeB as the left child of node 75.
2. Consider the `define-struct`, `define`, and `make-btnode` statements on the left below. The data definition for a binary tree is given on the right.

```
(define-struct btnode (key left right)) ;; A binary tree (BT) is either:
  ;;
(define a (make-btnode 11 null null)) ;; 1. null, or
(define c (make-btnode 13 null null)) ;; 2. it’s a
(define f (make-btnode 74 null null)) ;; (make-btnode n lft rght),
(define i (make-btnode 92 null null)) ;; where n is a number and
(define b (make-btnode 14 a c)) ;; lft and rght are BTs.
(define h (make-btnode 101 f i))
(define d (make-btnode 42 b h))
(define Tree-A d)
```

(a) Draw the tree formed by the sequence of define statements above using circles to represent each node and lines to represent connections between nodes and their left and right btnodes. Be sure to label each node with its defined name on the left side of the node and show the key (i.e., the number returned by calling btnode-key on a BT) inside each node. Show null subtrees as we did in class or using a connection to the word null.

![Tree-A Diagram]

(b) Divide the node names (e.g., h, d, g, etc.) in Tree-A into three categories:

- **Root**...................... Tree-a or d
- **Internal nodes**......... b  h
- **Leaves**..................... a  c  f  i

(c) Is the tree you drew for part (a) a binary search tree? **Answer yes or no and explain your answer.** If your answer is no, be explicit about which node pairs violate the rules for a binary search tree.

*No. the binary search tree rules are violated by placing 13 (c) as the right child of 14 (b) and by placing 92 (i) as the right child of 101 (h).*
3. (10 points) Consider the define-struct given below:

```
(define-struct pos (x y))
```

(a) (3 points) Define pos struct variables named \texttt{p1}, \texttt{p2}, and \texttt{p3} with any numbers you choose for the x,y fields in each pos.

```
(define p1 (make-pos 1 2))
(define p2 (make-pos 2 3))
(define p3 (make-pos 4 5))
```

(b) (2 points) Define a list of pos structs called \texttt{pos-list}, that contains each of the three pos structs you defined in part (a).

```
(define pos-list (list p1 p2 p3))
```

(c) (5 points) Design the function \texttt{translate}. It consumes and produces a list of pos structs. For each pos with fields x and y in the input list, the output list should contain a pos with field x unchanged and the value in field y incremented by 1. Write contract, input, and purpose statements; write 1 check-expect statement, using your list from part (b); and then define the function.

```
; Contract: (translate plisty) -- list of pos structs
; Input: plisty is a list of pos structs
; Output: a list of pos structs

(check-expect (translate poslist)
 (list (make-pos 1 3)
      (make-pos 2 2)
      (make-pos 2 4)))
```

; Function definition:
(define (translate plisty)
 (local
  [(define (inner-acc poslist posacc)
      (cond
       ; base case: return accumulator
       [(empty? poslist) posacc]
       ; recursive case: call accumulator function with the next
       ; translated pos on the right.
       [else
        (inner-acc
         (rest poslist)
         (append posacc (list
                         (make-pos
                          (pos-x (first poslist))
                          (add1 (pos-y (first poslist)))))))]))
  (inner-acc plisty '()))
)
4. (10 points) Consider the function `average-list`, given below (assume the function works correctly and finds the average of all numbers in `listy`):

```scheme
;; Contract: (average-list listy) -> number
;; Input: listy = list of numbers
;; Purpose: Find the average of a list of numbers.
(define (average-list listy)
  (local
    [(define len (length listy))]
    (cond
      [(= len 0) 0]
      [else
       (/ (apply + listy) len)])))
```

Rewrite `average-list` below so that it uses iteration (i.e., a `while` loop and mutation) instead of a higher-order function or recursion.

```scheme
(define (average-list listy)
  (local
    [(define len (length listy))]
    (define *pos* 0)
    (define *sum* 0)
    (while (< *pos* len)
      (set! *sum* (+ *sum* (list-ref listy *pos*)))
      (set! *pos* (add1 *pos*)))
    (/ *sum* len)))
```
5. (5 points) A *left rotation* of an ordered collection of items \(a_0, a_1, \ldots, a_n\) produces the collection \(a_1, a_2, \ldots, a_n, a_0\), where all items are shifted to the left and \(a_0\) is put on the right end of the collection. Write a function `vleft!` that consumes a non-empty vector called `vec` and rotates the elements in positions 1...(sub1 (vector-length `vec`)) to the left, changing the value at the last position in the vector to hold the item originally in position 0 of the vector.

The contract, input, purpose, pre-function tests, and a helper function (`swap!`) are given for you.

```
;; Contract: (swap! vecty p1 p2) -> vector of any valid type
;; Input: vecty = vector of any valid type
;; p1,p2 = numbers in the range 0...((vector-length vecty) - 1)
;; Pre-function tests:
(check-expect (swap! (vector 1 2 3 4) 0 2) (vector 3 2 1 4))
(check-expect (swap! (vector 5 6) 0 1) (vector 6 5))

(define (swap! vecty p1 p2)
  (local
    [(define temp (vector-ref vecty p1))
     (vector-set! vecty p1 (vector-ref vecty p2))
     (vector-set! vecty p2 temp)
     vecty))

;; Contract: (vleft! vec) -> non-empty vector of any valid type
;; Input: vec = non-empty vector of any valid type
;; Purpose: Rotate the elements of vec to the left, moving
;; item originally on the left to the right end of vec.
;; Pre-function tests:
(check-expect (vleft! (vector 1 2 3 4)) (vector 2 3 4 1))
(check-expect (vleft! (vector 4 3 2 1)) (vector 3 2 1 4))

;; Function definition of vleft!
(define (vleft! vecty)
  (local
    [(; save item in position 0
      (define frst (vector-ref vecty 0))
    ;; save length of vector because it is used several times
      (define len (vector-length vecty))
    ;; make a mutable variable *i* to keep track of position in vector
      (define *i* 1)]
    (while (< *i* len)
      (vector-set! vecty (sub1 *i*) (vector-ref vecty *i*))
      (set! *i* (add1 *i*))
      (vector-set! vecty (- len 1) frst))
    vecty))
```
6. (5 points) Consider the Racket code shown below. To the right of the code, draw a picture of the window after the user presses the WISH-KEY while the program is running. Indicate colors as best you can.

```
(define WIDTH 400)
(define HEIGHT 400)
(define WISH-TEXT "HAPPY HOLIDAYS!!")
(define MODE "solid")
(define BCOLOR "green")
(define TYPE 36)
(define TYPE-COLOR "red")
(define WISH-KEY "w")
(define wish ") ;; INITIAL WORLD STATE IS EMPTY STRING

(define (render w)
  (place-image
   (text w TYPE TYPE-COLOR)
   (/ WIDTH 2)
   (/ HEIGHT 2)
   (rectangle WIDTH HEIGHT MODE BCOLOR)))

(define (elf-key w k)
  (cond
    [(key=? k WISH-KEY)
      (string-append w WISH-TEXT)]
    [else w]))

(define (main)
  (big-bang
   wish
   (on-key elf-key)
   (on-draw render)))
```

Picture is 400 x 400 pixel green box with the text "HAPPY HOLIDAYS!!" centered on the scene in red letters.