**Piazza posts related to this assignment from previous terms:**  
A7 #1 vector-append-list

Here is an example that has some things in common with vector-append-list.  It makes a new vector that is the reverse of it's argument.  Ths procedure is not directly useful for problem #1, but it may help you to better understand the use if vectors.

(define vector-reverse ; return a vector that is the reverse of v  
   (lambda (v)  
      (let\* ([v-len (vector-length v)]  
             [result (make-vector v-len #f)])  
         (let loop ([ i 0])  
            (if (< i v-len) ; one-armed if, has no "else" part.  
                 (begin (vector-set! result (- v-len i 1)  
                        (vector-ref v i))  
                        (loop (+ i 1))))  
            result))))

(vector-reverse '#(2 3 4 5))

Are we required to use vector-set! for our solution? As it's not too difficult to have a solution making use of the vector procedure instead.



[**Claude Anderson**](https://piazza.com/class/is9cjqgxyh31b?cid=47) [5 months ago](https://piazza.com/class/is9cjqgxyh31b?cid=47)

No, you are not required to use vector-set!

# Most Scheme mutation procedures (such as vector-set!) do not return a value.

The subject line says it all.

# A7 Bintree internal sum

Is it allowed if we return the sum for each internal node and just find the max of those sums? It only adds another O(n) to find the max.

**the instructors' answer,**

*where instructors collectively construct a single answer*

But it bypasses what I wanted you to figure out how to do, so no.

# group-by-two and group-by-n: Don't be afraid to apply reverse in your code

I talked with a student today who was reluctant to use reverse, thinking that I have discouraged its use.  I think it may be easiest and most efficient to write the group-by procedures using cons, and then reversing the list once you have all of it.  Calling reverse once is O(N).  Appending each element to the end of a list (one at a time) is O(N^2).

# Seeing the procedure that compose makes

I'm writing make-c...r, and am having a hard time debugging since tracing the function just gives #<procedure>, so I can't really see what's going wrong with the composing.

Is there some way to see the function that compose makes or if not other ways to debug it?

Edit: right now I'm at the point where I'm composing a list of cars and cdrs (ex: '(car cdr cdr)) , and would be helped if I could see what the result of composing them looked like

**the students' answer,**

*where students collectively construct a single answer*

You may be able to trace compose itself.

**the instructors' answer,**

*where instructors collectively construct a single answer*

[Actions](https://piazza.com/class/jl863803n0a6tl?cid=55)

trace-lambda may be what you need

(trace-lambda trace-name (args) bodies), where trace-name is a name that you make up so that the tracing can identify which procedure is being traced.

There are also trace-let, trace-letrec, and trace-define.

# Problem 6 assignment 7

When the problem says "no explicit recursion" does that mean that you cannot use any recursion at all or a certain type? if so what does explicit mean?

**the students' answer,** It means that you should not write a recursion but you can use build-in procedure/syntax.

# Assignment 7b

Is it required for the s-list procedures to follow the exact format that we covered in class (i.e. named let and the three cases)? I didn't use name lets, which I don't think is a problem. However I'm "following the grammar" in a slightly different way.

**The instructors' answer,**

No.  It is not required.

One of the reasons I used named let in the in-class examples is because I thought that many students would not yet be comfortable with named let as they came into todays class and I hoped to increase their comfort level.

# BST- max- interior

I can't even think of where to start for this problem. Can anyone lend out a pointer?

**the students' answer,**

*where students collectively construct a single answer*

[Actions](https://piazza.com/class/k37lfj6dtp3c4?cid=27)

I don't know if this will help, but conceptually these are the cases:

1. the current node is a list, with two numbers as the children, so you return the sum of the two numbers

2. the current node is a list, with two lists as the children, so you return the max between the recursive call on the left child, the recursive call on the right child, and the sum of those values

3. the current node is a list, with one list and one number as children, so you return the max between the recursive call on the list child and the sum of that recursive call with the number child

traverse the tree in this way and you should only go through it once

also consider making a helper function to do this, so you can make the return type keep track of both the symbol and the sum associated with it (since you have to return the symbol of the interior node, not the sum)

**the instructors' answer,**

*where instructors collectively construct a single answer*

[Actions](https://piazza.com/class/k37lfj6dtp3c4?cid=27)

Each recursive call needs to return enough info so that the same calculation can be done at that node’s parent in the tree, without ever visiting the lower node again ( in other words, a post-order traversal of the tree).

What does it mean by maximal leaf sum?

for a subtree like such:

(ar (as (at 5 74)

(au -26 0))

(av (aw 51 -56)

(ax 39 70)))

why is ar the one we want instead of ax?

If a node whose both subtrees are not numbers, what do I do? According to the previous test case, I should sum up both results from the subtrees. However, if I do that, almost all answers become "aa". If that means to compare the left and right subtrees and pass the greater one on, it contradicts the test case above. I am very on edge of this.

**the students' answer,**

*where students collectively construct a single answer*

[Actions](https://piazza.com/class/ke8ite9gsc64w2?cid=72)

It kind of means both. You compare the left and right subtrees, as well as the node you're currently on, and pass on the greatest of those. Take the av aw ax tree. The sum at aw = 51 + (-56) = -4. The sum at ax = 39 + 70 = 109. The sum at av = aw + ax = (-4) + 109 = 105. Since 109 is the greatest of these, you would pass on ax.

# Compose Arguments for make-c...r

The way the provided compose function is formatted, it either expects the null list or an improper list as the arguments. However, they way I currently have make-c...r written, the list of functions to apply is proper (ex. (car cdr cdr) instead of (car . (cdr cdr)). This naturally causes issues. I know the rest of my code works, as I have gotten it working with code I had for compose which uses a proper list. Is there a way to convert a proper list to an improper one, or otherwise format my list so it will work?

**the instructors' answer,**

*where instructors collectively construct a single answer*

*The way the provided compose function is formatted, it either expects the null list or an improper list as the arguments.*

That is not true.  It does not expect its arguments to be in any kind of list.  We can call it like this: (compose) or (compose car car cdr).  When we CALL compose, Scheme takes the given arguments and puts them into a list.  That list will never be improper.

The video called "compose and case-lambda" attempts to explain all of this.

# cases for BT tests

For the binarytree tests do we need to worry about inputs that aren't binary trees, or can we assume that all inputs have the right form?

**he instructors' answer,**

*where instructors collectively construct a single answer*

[Actions](https://piazza.com/class/ke8ite9gsc64w2?cid=77)

you can assume that they have the correct form.

# Binary Tree Sum Recursion

In question 4 of assignment 7 it says that we can't call bt-leaf-sum on every interior node because it would make us traverse the same subtree multiple times. How would recursively calling bt-leaf-sum down the tree cause subtrees to be traversed multiple times and if we aren't calling it recursively then how are we supposed to code bt-leaf-sum?

**the instructors' answer,**

*where instructors collectively construct a single answer*

[Actions](https://piazza.com/class/ke8ite9gsc64w2?cid=78)

Calling bt-leaf-sum once on the whole tree does not traverse any subtree multiple times.  Calling bt-leaf-sum at every interior node makes it traverse the lower subtrees once for each of those calls, thus multiple times.

I hope this helps.

# Are we allowed make an wrapper object to solve bt-max-interior

I created a wrapper object using the stuff we learned in class today (i.e representing objects as procedures) and I was wondering if I was allowed to use this solution for the assignment. This solution has mutation in it, but I was unclear if when the problem says no mutation if that applies to data structures we create. This solution was the most efficient I can think of since you visit each node only once and I couldn't really think of another way to do this procedure without some sort of wrapper(object storing the local max. sum and symbol)

**the instructors' answer,**

*where instructors collectively construct a single answer*

The answer is no.  A main point of this problem is that you can do efficient programming without mutation, even for some complicated problems.

If you use mutation in any form, you will receive 0 credit for that part of the problem.