

CSSE 304 Assignment 7

Objectives: You should learn

- More about list processing.
- More about the use of `let`, `letrec`, `named let`, `map`, and `apply`.
- How to base recursive programs on recursive datatype definitions.

At the end of this document, there are several questions and answers from previous term's Piazza.

Details for these instructions are in the previous assignment, so not repeated here: Individual assignment. Comments at beginning, before each problem, and when you do anything non-obvious. Submit to server (test offline first). Your code must not mutate (unless a particular problem calls for it), read, or write anything. Assume arguments have correct form unless problem says otherwise.

Abbreviations for the textbooks:

EoPL	-	Essentials of Programming Languages, 3 rd Edition
TSPL	-	The Scheme Programming Language, 4 th Edition
EoPL-1	-	Essentials of Programming Languages, 1 st Edition (handout)

Reading Assignment: See the schedule page, and the Things to do page on Moodle. Have you been keeping up with the reading?

Problems to turn in: For many of these, you will want to write one or more helper procedures.

#1 (10 points) `(vector-append-list v ls)` returns a new vector with the elements of `ls` attached to the end of `v`. Do this without using `vector->list`, `list->vector`, or `append`.

For this problem only, you can and should use mutation: namely the `vector-set!` procedure. Note that `vector-set!` does not return a value.

Hint that I posted on Piazza for a previous term:

Try using this code, and writing the two recursive helper procedures. You are not *required* to use this code.

```
(define vector-append-list
  (lambda (vec ls)
    (let ([new-vector (make-vector (+ (vector-length vec) (length ls)))]
          (copy-from-vector new-vector vec 0)
          (copy-from-list new-vector ls (vector-length vec))
          new-vector)))
```

#2 (10 points) `(group-by-two ls)` takes a list `ls`. It returns a list of lists: the elements of `ls` in groups of two. If `ls` has an odd number of elements, the last sublist of the return value will have one element.

group-by-two: List \rightarrow ListOf(List)

```
> (group-by-two '())
()
> (group-by-two '(a))
((a))
> (group-by-two '(a b))
((a b))
> (group-by-two '(a b c))
((a b) (c))
> (group-by-two '(a b c d e f g))
((a b) (c d) (e f) (g))
> (group-by-two '(a b c d e f g h))
((a b) (c d) (e f) (g h))
```

#3 (20 points) `(group-by-n ls n)` takes a list `ls` and an integer `n` (you may assume that $n \geq 2$). Returns a list of lists: the elements of `ls` in groups of `n`. If `ls` has a number of elements that is not a multiple of `n`, the length of the last sublist of the return value will be less than `n`. **For full credit, your code must run in time $O(\text{length}(ls))$. In particular, this means that no recursive procedure in your code can call `(length ls)`.**

group-by-n: List \rightarrow ListOf(List)

```
> (group-by-n '() 3)
()
> (group-by-n '(a b c d e f g) 3)
((a b c) (d e f) (g))
> (group-by-n '(a b c d e f g) 4)
((a b c d) (e f g))
> (group-by-n '(a b c d e f g h) 4)
((a b c d) (e f g h))
> (group-by-n '(a b c d e f g h i j k l m n o) 7)
((a b c d e f g) (h i j k l m n) (o))
> (group-by-n '(a b c d e f g h) 17)
((a b c d e f g h))
> (group-by-n '(a b c d e f g h i j k l m n o p q r s t) 17)
((a b c d e f g h i j k l m n o p q) (r s t))
```

#4 (57 points) Consider the following syntax definition from page 9 of EoPL:

`<bintree> ::= <integer> | (<symbol> <bintree> <bintree>)`

Note that this representation is quite different than the BST representation in Assignment 6.

Write the following procedures:

- `(bt-leaf-sum T)` finds the sum of all of the numbers in the leaves of the bintree `T`.
- `(bt-inorder-list T)` creates a list of the symbols from the *interior* nodes of `T`, in the order that they would be visited by an inorder traversal of the binary tree.
- `(bt-max T)` returns the largest integer in the tree.
- `(bt-max-interior T)` takes a binary tree with at least one interior node, and returns (in $O(N)$ time, where N is the number of nodes) the symbol associated with an interior node whose subtree has a maximal leaf sum (at least as large as the sum from any other interior node in the tree). If multiple nodes in the tree have the same maximal leaf-sum, return the symbol associated with the leftmost (as it appears in the tree's printed representation) maximal node.

The `bt-max-interior` procedure is trickier than it looks at first!

- You may not use mutation.
- You may not traverse any subtree twice (such as by calling `bt-leaf-sum` on every interior node).
- You may not create any additional size- $\Omega(N)$ data structures that you then traverse to get the answer.
- Think about how to return enough info from each recursive call to solve this without doing another traversal.

Note: We will revisit this linear-time `bt-max-interior` problem several times during this course. If you do not get this version, the later versions will be harder for you, so you should do what it takes to get this one.

#5 (50 points)

These s-list procedures have a lot in common with the s-list [procedures](#) that we wrote during our Session 8 class. Recall the extended BNF grammar for s-lists:

`<s-list> ::= ({<s-expression>}*)`
`<s-expression> ::= <symbol> | <s-list>`

FOLLOW THE GRAMMAR!

(a) `(slist-map proc slist)` applies `proc` to each element of `slist`.

slist-map: *procedure* \times *Slist* \rightarrow *NestedListOfThingsThatAreInTheRangeOfProcedure*

```
(slist-map symbol? '((a (()) b) c () (d e)))  $\rightarrow$  ((#t (()) #t) #t () (#t #t))
(slist-map (lambda (x)
  (let ([s (symbol $\rightarrow$ string x)])
    (string $\rightarrow$ symbol (string-append s s))))
  '((b (c) d) e ((a)) () e))  $\rightarrow$  ((bb (cc) dd) ee ((aa)) () ee)
```

(b) (`slist-reverse slist`) reverses `slist` and all of its sublists.

slist-reverse: *Slist* → *Slist*

```
(slist-reverse '(a (b c) ( ) (d (e f)))) → (((f e) d) ( ) (c b) a)
```

(c) (`slist-paren-count slist`) counts the number of parentheses required to produce the printed representation of `slist`. You must do this by traversing the structure, not by having Scheme give you a string representation of the list and counting parenthesis characters. You can get this count by looking at `cars` and `cdrs` of `slist`).

slist-paren-count: *Slist* → *Integer*

```
(slist-paren-count '()) → 2
(slist-paren-count '(a (b c) d)) → 4
(slist-paren-count '(a (b) (c ( ) ((d)))))) → 12
```

Note: s-lists are always *proper* lists.

(d) (`slist-depth slist`) finds the maximum nesting-level of parentheses in the printed representation of `slist`. You must do this by traversing the structure, and *not* by having Scheme give you a string representation of the list and counting the maximum nesting of parenthesis characters.

slist-depth: *Slist* → *Integer*

```
(slist-depth '()) → 1
(slist-depth '(a b c)) → 1
(slist-depth '(a (b c) d)) → 2
(slist-depth '(a (b (c)) (a b))) → 3
(slist-depth '(((a) (( ) ) b) (c d) e)) → 4
```

(e) (`slist-symbols-at-depth slist d`) returns a list of the symbols from `slist` whose depth is the positive integer `d`. They should appear in the same order in the return list as in the original s-list. This one has the basic pattern of the other s-list procedures, but when writing the solution, I found it easier to use a slight variation on that pattern.

slist-symbols-at-depth: *Slist* × *PositiveInteger* → *ListOf(Symbol)*

```
(slist-symbols-at-depth '(a (b c) d) 2) → (b c)
(slist-symbols-at-depth '(a (b c) d) 1) → (a d)
(slist-symbols-at-depth '(a (b c) d) 3) → ()
```

#6 (10 points) (`path-to slist sym`) produces a list of `cars` and `cdrs` that (when read left-to-right) take us to the position of the leftmost occurrence of `sym` in the s-list `slist`. Notice that the returned list contains the symbols `'car` and `'cdr`, not the `car` and `cdr` procedures. Return `#f` if `sym` is not in `slist`. Only traverse as much of `slist` as is necessary to find `sym` if it is there.

```
> (path-to '(a b) 'a)
(car)
> (path-to '(c a b) 'a)
(cdr car)
> (path-to '(c ( ) ((a b))) 'a)
(cdr cdr car car car)
> (path-to '(((d (f ((b a)) g))) 'a)
(car cdr car cdr car car cdr car)
> (path-to '(((d (f ((b a)) g))) 'c)
#f
```

#7 (25 points) Predefined Scheme procedures like `cadr` and `cdadr` are compositions of up to four `cars` and `cdrs`. You are to write a generalization called `make-c...r`, which does the composition of any number of `cars` and `cdrs`. It takes one argument, a string of a's and d's, which are used like the a's and d's in the names of the pre-defined `c...r` functions. For example, `(make-c...r "addddd")` is equivalent to `(compose car cdr cdr cdr cdr)`.

```
> (define caddddr (make-c...r "addddd"))
> (caddddr '(a (b) (c) (d) (e) (f)))
(e)
> ((make-c...r "") '(a b c))
(a b c)
> ((make-c...r "a") '(a b c))
a
> ((make-c...r "ddaddd") '(a b c ((d e f g) h i j)))
(i j)
> ((make-c...r "addddddddddd") '(a b c d e f g h i j k l m))
l
```

I have provided the code for `compose`. We will discuss this solution in class soon. For now you can just use it here. For full credit, you should write `make-c...r` in a functional style that only applies (calls)

- built-in procedures (including, `car`, `cdr`, `map`, and `apply`, of course),
- anonymous procedures, and
- `compose` (which I am giving you, so it does not count as a self-written recursive procedure)

in your definition of `make-c...r`. More precisely, you may not write and call your own recursive procedures

```
(define compose
  (case-lambda
    [(()) (lambda (x) x)]
    [(first . rest)
     (let ([composed-rest (apply compose rest)])
       (lambda (x) (first (composed-rest x))))])
```

Hint: My solution uses the built-in procedures `map`, `apply`, `string->list`, `list->string`, `string->symbol`, and `eval`; also the character constants `#\c` and `#\r`. You are not required to use these, but you may find them helpful. I do not assume that you are already familiar with all of them; *The Scheme Programming Language* contains info on them; my intention is that you demonstrate an ability to look up and use new procedures. My solution calls `map` multiple times.

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. This 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 5 months ago

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

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 today's 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

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

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*

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, the way I currently have make-c...r written, the list of functions to apply is proper (ex. (car 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?

the instructors' answer,*where instructors collectively construct a single answer*

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*

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.