CSSE 304 Assignment 6

Objectives: You should learn

- To begin to use first-class procedures effectively.
- To think a bit about efficiency of programs.

Before doing A6, be sure to read EoPL-1 from page 22 to the end of chapter 1.

Details for these instructions are in the previous assignment
Individual assignment**.** Comments at beginning, before each problem, when you do anything non-obvious. Submit to server (test offline first). Mutation not allowed.

Assignments 6a (problems 1-7) and 6b (problems 8-12) are due on different days.

Reading Assignment: See the schedule page. Have you been keeping up with the reading?

The first four problems are different than problems on previous assignments, in that I want you to read the EoPL-1 material and do related problems without first discussing the material in class. Of course, you are allowed to get help from other students, from the TAs, and from me.

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

Some of the problems deal with ***currying***. http://en.wikipedia.org/wiki/Currying describes this as:

> In mathematics and computer science, currying (schönfinkeling) is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument (partial application). It was originated by Moses Schönfinkel and later worked out by Haskell Curry.

Optional, not required knowledge for this course: An interesting discussion of the advantages of currying (the language of discourse is Haskell, but I think you can still follow much of the discussion):
http://www.reddit.com/r/programming/comments/181y2a/what_is_the_advantage_of_currying/
Some simple examples of currying appear on pages 26 (last sentence) through 28 of EoPL-1. The first two turnin-problems are from that section, and I recommend that you also think about problem 1.3.6.

Reminder: EoPL-1 is the first edition of EoPL. An excerpt was handed out on the first day of class. It is also available on Moodle.

#1 (10 points) `curry2`. This is EoPL-1 Exercise 1.3.4, page 28. Examples are on that page.

curry2 : ((*SchemeObject* × *SchemeObject*) → *SchemeObject*) → (*SchemeObject* → (*SchemeObject* →*SchemeObject*))

#2 (10 points) EoPL-1 Exercise 1.3.5, page 28. Call your procedure `curried-compose` .
For example, `(((curried-compose car) cdr) '(a b c))` ➔ `b`

curried-compose : (*SchemeObject* → *SchemeObject*) → ((*SchemeObject* →*SchemeObject*) → (*SchemeObject* →*SchemeObject*))

#3 (10 points) `compose`. EoPL-1 Exercise 1.3.7, page 29. This one will most likely begin

```
(define compose
  (lambda list-of-functions        ; notice the lack of parentheses around the argument name.
```

compose : *listOf* (*SchemeObject* → *SchemeObject*) → (*SchemeObject* →*SchemeObject*)

Examples:

```
((compose list list) 'abc)           ➔  ((abc))
((compose car cdr cdr) '(a b c d))   ➔  c
```

#4 (10 points) Write the procedure `make-list-c` that is a curried version of `make-list`.
(Note that the original `make-list` is described in TSPL Exercise 2.8.3
.
make-list-c : *Integer* → (*SchemeObject* → *Listof(SchemeObject*))

Examples:

```
      ((make-list-c 3) 'xyz)         ➔   (xyz xyz xyz)
      (let ([triple (make-list-c 3)])
        (triple "cat"))               ➔   ("cat" "cat" "cat")
```

#5 (10 points) Write (`reverse-it lst`) that takes a single-level list `lst` and returns a new list that has the elements of `lst` in reverse order. The original list should not be changed. Can you do this in O(n) time? [this is not a requirement]. You probably cannot do it in O(N) time if you use `append`. Can you see why? Obviously, you should not use Scheme's `reverse` procedure in your implementation.

#6 (10 points) Write (`map-by-position fn-list arg-list`) where

- `fn-list` and `arg-list` have the same length,
- each element of the list `fn-list` is a procedure that can take one argument,
- and each element of the list `arg-list` has a type that is appropriate to be an argument of the corresponding element of `fn-list`.

`map-by-position` returns a list (in their original order) of the results of applying each function from `fn-list` to the corresponding value from `arg-list`. **You must use `map` to solve this problem; no explicit recursion is allowed.**

```
(map-by-position (list cadr - length (lambda(x)(- x 3)))
                '((1 2) -2 (3 4) 5))                        ➔ (2 2 2 2)
```

#7 (45 points) **Examples are in the test cases**. A Binary Search Tree (BST) datatype is defined on page 10 of EoPL. Note that, as the book describes, the grammar alone is not sufficient to completely describe what constitutes a BST. We also need the "smaller values in left subtree, larger in right subtree" property, which cannot be descibed using a context-free grammar. Write the following procedures. Remember, no mutation of lists is allowed ! If you cheat by representing a BST as a single-level sorted list or some other simpler structure, you will get 0 points, even if your code passes a lot of the test cases.

1. `(empty-BST)` takes no arguments and creates an empty tree, which is represented by an empty list.
2. `(empty-BST? obj)` takes a Scheme object `obj`. It returns `#t` if `obj` is an empty BST and `#f` otherwise.
3. `(BST-insert num bst)` returns a BST *result*. If `num` is already in `bst`, *result* is structurally equivalent to `bst`. If `num` is not already in `bst`, *result* adds `num` in its proper place relative to the other nodes in a tree whose shape is the same as the original. Like any BST insertion, this should have a worst-case running time that is O(height(`bst`)).
4. `(BST-inorder bst)` (can you do it in O(N) time?) produces an ordered list of the values in `bst`.
5. `(BST? obj)` returns `#t` if Scheme object `obj` is a BST and `#f` otherwise.
6. `BST-element`, `BST-left`, `BST-right`. Accessor procedures for the parts of a node.
7. `(BST-insert-nodes bst nums)`starts with tree `bst` and inserts each integer from the list `nums`, in the given order, returning the tree that includes all of the inserted nodes. The original tree is not changed, and this procedure does no mutation.
8. `(BST-contains? bst num)` determines, in time that is O(height(`bst`)), whether `num` is in `bst`.
9. `(BST-height bst)` returns the height of the BST `bst`. **Examples:**

```
    (BST-height '()) ➔ -1
    (BST-height '(3 () ())) ➔ 0
    (BST-height '(2 ()(6 (4 () ()) ()))) ➔ 2
```

#8 (10 points) Write `let->application` which takes a `let` expression (represented as a list) and returns the equivalent expression (also represented as a list) that represents an application of a procedure created by a `lambda` expression. Your solution should not change the *body* of the `let` expression. This procedure's output list replaces only the top-level `let` by an equivalent application of a `lambda` expression. You do not have to find and replace any non-top-level `let`s. You may assume that the `let` expression has the proper form; your procedure does not have to check for this. Furthermore, you may assume that the `let` expression is *not* a named `let`.

let->application : *SchemeCode* → *SchemeCode*

Example:

```
      (let->application '(let ((x 4) (y 3))
                          (let ((z 5))
                             (+ x (+ y z))))) ➔
     ((lambda (x y)
        (let ((z 5))
          (+ x (+ y z))))
      4 3)
```

#9 (10 points) Write `let*->let` which takes a let* expression (represented as a list) and returns the equivalent nested `let` expression. This procedure replaces only the **top-level** `let*` by an equivalent nested `let` expression. You may assume that the `let*` expression has the proper form.

let*->let: *SchemeCode* → *SchemeCode*

Example:

```
      (let*->let '(let* ([a 3] [b (+ a 4)]) b ))
          ➔
      (let ([a 3])
         (let ([b (+ a 4)])
            b))
```

#10 (20 points) `(qsort pred ls)` is a Scheme procedure that you will write whose arguments are

a predicate (total ordering) which takes two arguments `x` and `y`, and returns `#t` if `x` is "less than" `y`, `#f` otherwise.

a list whose items can be compared using this predicate.

`qsort` should produce the sorted list using a QuickSort algorithm (write your own; do not use Scheme's `sort` procedure).

For example:

```
(qsort <= '(4 2 4 3 2 4 1 8 2 1 3 4)) ➔ (1 1 2 2 2 3 3 4 4 4 4 8)

(qsort (lambda (x y) (<= (abs (- x 10)) (abs (- y 10))))
       '(5 1 10 8 16 17 23 -1))
     ➔ (10 8 5 16 17 1 -1 23)
```

If you do not remember how QuickSort works, see http://en.wikipedia.org/wiki/Quicksort or Chapter 7 of the Weiss book used for CSSE230. There are quicksort algorithms that do fancy things when choosing the pivot in order to attempt to avoid the worst case. You do not need to do any of those things here; you can simply use the `car` of the list as the pivot. Since mutation is not allowed, your algorithm cannot do the sort in-place. Furthermore, you are not allowed to copy the list elements to a vector, then sort the vector and copy back to a list. All of your work should be done with lists.

#11 (15 points) Write a Scheme procedure `(sort-list-of-symbols los)` which takes a list of symbols and returns a list of the same symbols sorted as if they were strings. You will probably find the following procedures to be useful: `symbol->string`, `map`, `string<?`, `sort` (you can look them up in the Chez Scheme Users' Guide). Note that we have not covered specifics related to this problem, It is time for you to read some documentation and figure out how to use things.

sort-list-of-symbols: *ListOf*(*Symbol*) → *ListOf*(*Symbol*)

Example `(sort-list-of-symbols '(b c d g ab f b r m)) ➔ (ab b b c d f g m r)`

Previous questions and answers on Piazza

Map-by-position

I'm not sure I get what the procedure is supposed to do.
Is it supposed to apply each fn-list to each object in the arg-list? And does it do it in that order?
Or does it apply the corresponding function to the corresponding object?
I thought that's what it was, but I don't see which elements are corresponding since the example one gives functions that don't apply to the argument it's matched up with.

the instructors' answer,
where instructors collectively construct a single answer
It is your second case, corresponding.

Look more closely. The procedures and arguments in my example DO correspond. Just as in the example from the beginning of yesterday's class, **list** is not one of the procedures in the list.
edit · good answer0

Updated 5 months ago by Claude Anderson

followup discussions
I think I see my mistake now, is "list" not one of the functions? That would be my problem...
Reply to this followup discussion
● Resolved ○ Unresolved

Claude Anderson 5 months ago

list just makes a list of the functions

qsort: odd error 'incorrect argument count in call'

In the partition method I wrote for qsort, I've been getting an odd error that tells me there may be an incorrect argument count in call "(partition pred? (cdr ls) (list (car ls)))". It's the same line of code every time. But that method is defined right under qsort, as having 3 arguments "(define partition
(lambda (pred? ls pivot)" Oddly enough, if I cut the code out, load it, then put the code back in, it will load perfectly fine without errors and work correctly. However, I'm getting this error when uploading to the server now and have no idea how to fix it. The cases that fail on the server work fine locally, they just give that error on the server.

hw7
edit · good question0

the instructors' answer,
where instructors collectively construct a single answer
There is a link on the Schedule page (Day 1, resources column) called "What if the grading program gives zero points for something that works on your computer?" Follow it and read the document.

I believe that this document describes your problem and tells you how to fix it.
The built-in procedure that you are redefining is "partition".

Sort-list-of-symbols: Clarification

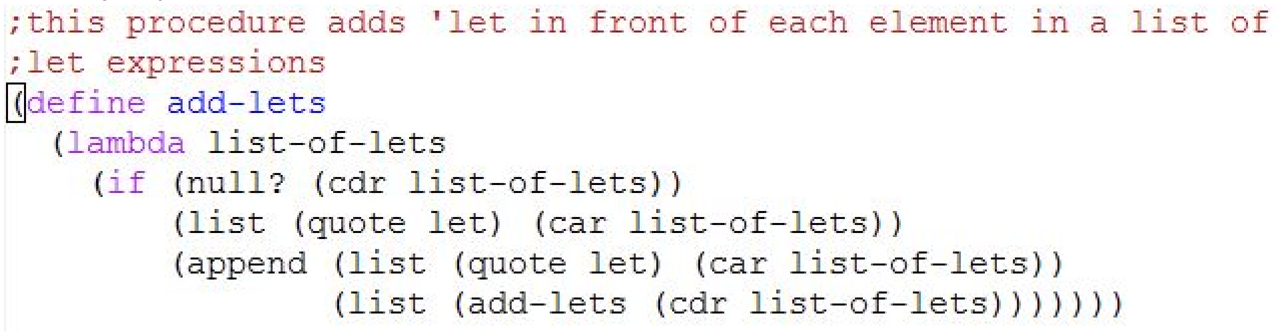
So to clarify, we aren't allowed to use any sort of recursion or looping for this, or are we allowed to use recursion with map?

the instructors' answer,
where instructors collectively construct a single answer
You may use map. The function that you are mapping over the list can (I think must) be one that you write, but it cannot be call itself or call another procedure that calls it.

Let->let*: getting let to not combine

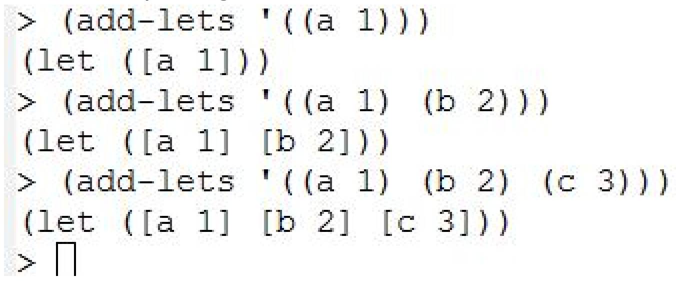
Im working on the let*->let problem and i made a helper function that take in the chunk of the passed list that holds the various let expressions (i.e '((a 1) (b 2) (c 3))) and is supposed to put a let in form of each expression (i.e. create '(let (a 1) (let (b 2) (let (c 3))))) but no matter what combination of cons/list/append I use scheme/emacs is combining my let expressions into one (even though each recursive call i use an individual (quote let)). is there a way to circumvent this?

this is my helper function code:



```
;this procedure adds 'let in front of each element in a list of
;let expressions
(define add-lets
  (lambda list-of-lets
    (if (null? (cdr list-of-lets))
        (list (quote let) (car list-of-lets))
        (append (list (quote let) (car list-of-lets))
                (list (add-lets (cdr list-of-lets)))))))
```

and the outputs I get



```
> (add-lets '((a 1)))
(let ([a 1]))
> (add-lets '((a 1) (b 2)))
(let ([a 1] [b 2]))
> (add-lets '((a 1) (b 2) (c 3)))
(let ([a 1] [b 2] [c 3]))
> 
```

the instructors' answer,

where instructors collectively construct a single answer

I believe I see what's going wrong for you here. The initial issue is your function add-lists is using a lambda with what's termed "var-args". If you don't enclose list-of-lets in parentheses, then the function will 1) accept any number of arguments, and 2) collect all its arguments, themselves as a list, in list-of-lets.

To take a small example:

```
> (define simple-example

    (lambda x

      (list x x x)))

> (simple-example 'foo)

'((foo) (foo) (foo))

> (simple-example 'foo 'bar 'baz)

'((foo bar baz) (foo bar baz) (foo bar baz))

>
```

These so-called 'var-args' functions can be useful. For instance

```
> (+ 1 2)

3

> (+ 1 2 3)
```

```
6
```

We just saw another use above, with list. As a fun exercise, think about how to implement `list` as a var-args function.

The upshot, I think, is that if you add a set of parens around the function argument, Scheme should instead treat this as an ordinary, one-argument function and I believe will give the behavior you're intending.

BST problem:

So does scheme have INT_MAX and INT_MIN?

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

As far as I know, the size of integers is limited only by the amount of memory that is available to Scheme. For example, use the definition of factorial from the day05 folder in Live-in-class. I tried computing the factorial of 20,00, and it worked just fine. The answer ts more than 77,000 digits long. None of the assigned problems require knowing these max and min numbers. Just use the numbers in the tree (perhaps with a post-order traversal).

Questions for assignment 6

I am having trouble finding the questions for assignment 6a. The homework sheet for assignment 6 says the question is exercise 1.3.4 on page 28 in EoPL, but all I can find on page 28 are exercises 1.xx and above, but none in the format of 1.x.x. Furthermore, none of the questions mention currying. Is there something I'm not understanding?

the students' answer,
where students collectively construct a single answer
Actions

You'll find the problems in EoPL-1, which is the handout/packet we received in class that had four book pages copied onto each page.

bt-inorder-list Can I use filter-in?

Can I just make a list of in-order traversal of the whole tree then filter in all the symbols? Or I need to check if the value is a symbol when I make the list?

In this problem, does it mean all interior nodes symbols and all leaves integers? If so it would be easier not to use filter-in.

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

- Flag as Inappropriate

Best: When you traverse the tree to make the list, don't include the leaf nodes. Then you don't have to remove them.

Quicksort Algorithm: Does the runtime have to be strictly N(logN) best case?

The implementation I have in mind behaves like Java Quicksort, but has a longer run time than best-case N(log N).

hw6
edit·good question0

Updated 2 months ago by

Eric Tu

the instructors' answer,
where instructors collectively construct a single answer
I can't think of any reasonable approach that would not be N log N in the best case. If it is not N log N, is it really quicksort?

Problem 7 - assumptions about data in BST

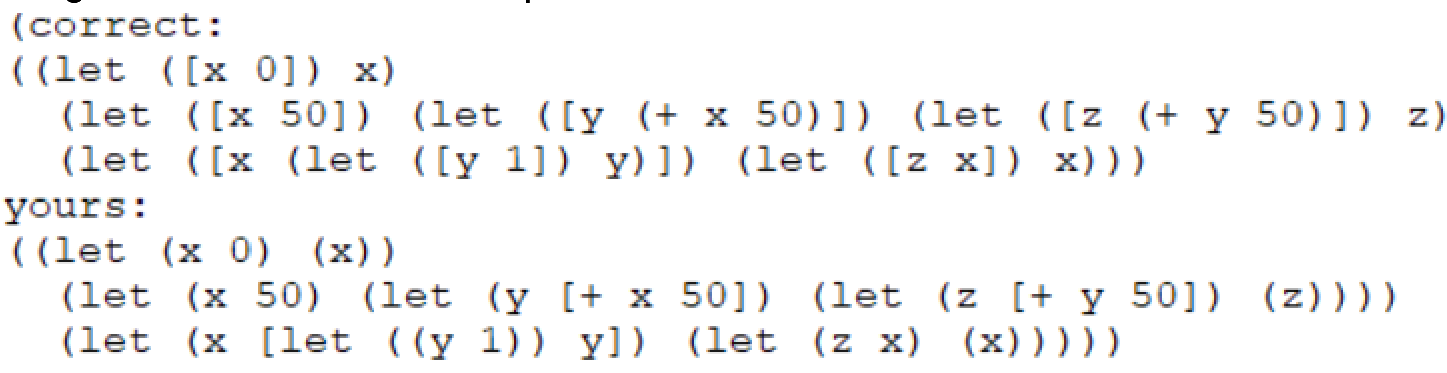
For problem 7 on homework 6, if we try to insert a new value that already exists in the BST, should we just not insert it? If not, which side should it be inserted on?

the instructors' answer:
It is said int the problem that *if num is already in bst, result is structurally equivalent to bst*, so just don't insert it.

Making a list out of InOrder traversal

I can't seem to figure out a way to format the result for this function into a singular list. I have it all "in order," but not in a list. Is there a simple way to do this?
the instructors' answer,
Use a combination of append and list. I don't think I can say much more without entirely giving it away.
Followup: Is it possible to do it without append for efficiency?
the instructors' answer: Probably possible, but not at all easy.

Assignment 6b: How to add square bracket?



```
(correct:
((let ([x 0]) x)
  (let ([x 50]) (let ([y (+ x 50)]) (let ([z (+ y 50)]) z)
  (let ([x (let ([y 1]) y)]) (let ([z x]) x)))
yours:
((let (x 0) (x))
  (let (x 50) (let (y [+ x 50]) (let (z [+ y 50]) (z))))
  (let (x [let ((y 1)) y]) (let (z x) (x)))))
```

I tried to add brackets to the result but somehow some brackets appear but some disappeared.

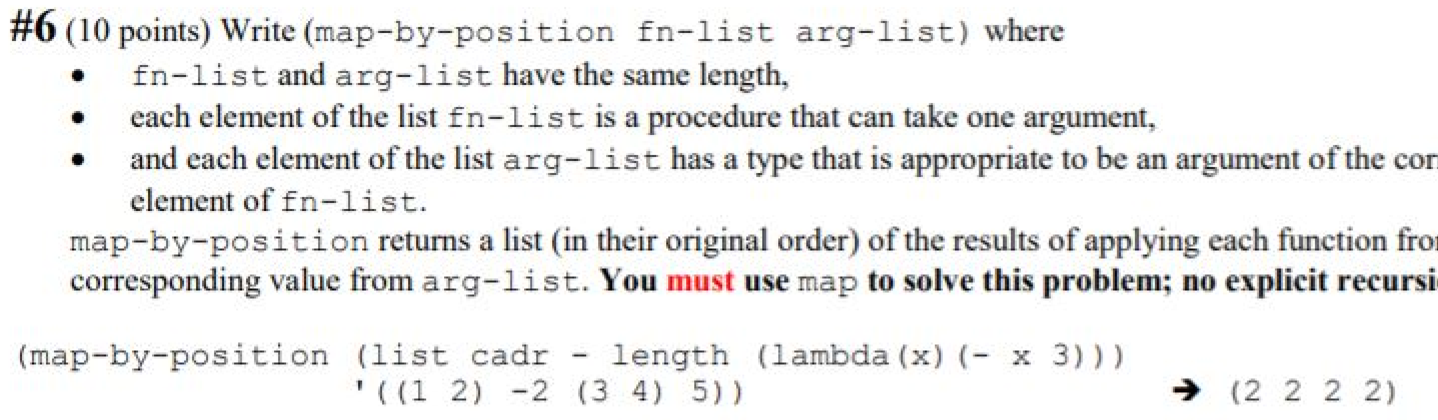
the students' answer:
You don't need to add the square brackets by yourself. If your list structure is correct, the parenthesis will be turned into square brackets when needed.

Reply: Got it. Do you know what kind of list structure can cause a square bracket?

Reply: If you look at your answer, you have square brackets, but they are at the wrong places. This is because your list structure is incorrect and the tests are turning parentheses into brackets where they are not supposed to be turned. The correct list structure should look like the "correct" version above your answer.

#6 on Assignment 6a

I'm a little confused by this question. The question says that fn-list and arg-list should have the same length but the example case shows it having 5 procedures in the fn-list and 4 arguments in the arg-
list



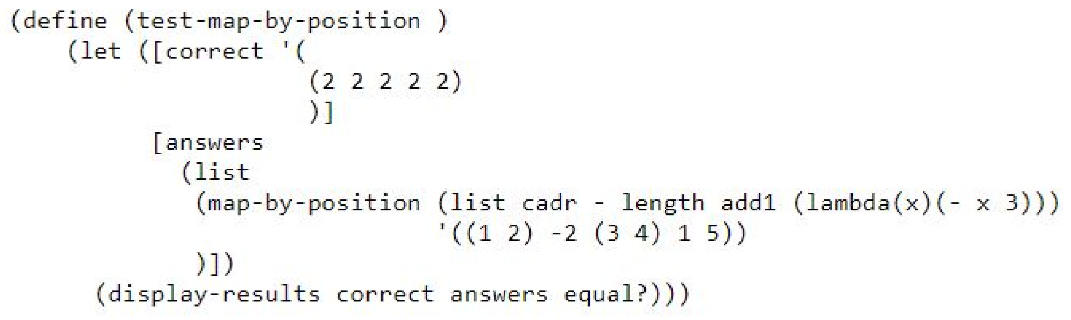
#6 (10 points) Write (`map-by-position fn-list arg-list`) where

- `fn-list` and `arg-list` have the same length,
- each element of the list `fn-list` is a procedure that can take one argument,
- and each element of the list `arg-list` has a type that is appropriate to be an argument of the cor
  element of `fn-list`.

`map-by-position` returns a list (in their original order) of the results of applying each function fro corresponding value from `arg-list`. **You must use `map` to solve this problem; no explicit recursi**

```
(map-by-position (list cadr - length (lambda(x)(- x 3)))
                 '((1 2) -2 (3 4) 5))                        ➔ (2 2 2 2)
```

In the example case, it appears to have 6 procedures in the fn-list and only 5 arguments in the arg-list.



```
(define (test-map-by-position )
    (let ([correct '(
                     (2 2 2 2 2)
                     )]
          [answers
            (list
             (map-by-position (list cadr - length add1 (lambda(x)(- x 3)))
                              '((1 2) -2 (3 4) 1 5))
             )])
      (display-results correct answers equal?)))
```

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

Look closely. I believe that in the example case, there are 4 functions and 4 arguments.
This is all explained near the end of the "map and apply" video.

Time limit on quicksort?

Is there a time limit on quicksort?

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

It's a very generous time limit, 100 ms. The average run of a test case on student code so far is about 0.35 ms.

#6 on Assignment 6a Recursion

#6 says that we have to use map and no explicit recursion. Does this mean no recursion at all in the solution?

the students' answer,
where students collectively construct a single answer

I believe so. If you are using recursion you might be making it a little too complicated in your head. The way that I got started on this problem was to recall what I did for matrix-inverse. The important thing is, map can take more than two arguments.

~ An instructor (
Claude Anderson
) endorsed this answer ~

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

Yes. All of the things you would normally do recursively should be done by using **map** instead. Thus none of your helper procedures can call themselves.

qsort should not use list-ref

This is not a requirement, since it wasn't in the specification, but you should try to make your code as efficient as you reasonably can.

A student who was in my office hours used list-ref in his code. Since list-ref is Θ(N) there is no way that qsort can be N log N even in the best case if you use list-ref. When doing the partition phase of qsort, use car, cdr, and cons, each of which is constant time.

Letrec restriction

One thing that I've been struggling to grasp well is why, when using letrec, should the exprs be evaluated prior to the vars? For example, why is this operation illegal?

```
(letrec (       [x (+ y 1)]

                                                  [y 1]    )

x)
```

I think you may be confusing the behavior of `letrec` with `let*`. The behavior of `letrec` can be a bit confusing. In order, it (1) bind all the variables to some `undefined` value, (2) evaluates the expression of each variable (in an unspecified order), (3) assigns each variable to the result, then (4) evaluates the body. The expression you currently have is trying to evaluate `(+ y 1)` while `y` is `undefined`. Notice that the following expression won't work for the same reason:

```
(letrec ([y 1] [x (+ y 1)])

  x)
```

The behavior is documented a bit more in the docs.

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

The order of evaluation of those initializing expressions in letrec, like in let, is unspecified. There is a letrec*.
My suggestion: Only use letrec to define and name local **procedures**, not just any local variables. Then the lambdas will prevent the bodies from getting executed sooner than yo want.
In the interpreter project, we will assume that letrec in our interpreted language is only used in this restricted way.