CSSE 304 Exam #1 Part 2 Sept 22, 2020 (day 12.5) (12:00)

The Exam2 assignment on the PLC server will close at 9:30. After a few minutes I will reopen it for students with accommodations and for those who start late due to choir practice.

The maximum score in the Moodle gradebook for this part of the exam id 62 points. The total possible for these problems adds up to 67. So it is possible to earn more than 100%, or to get full credit while failing a test-case or two.

Starting code and offline test cases will be available on Moodle (at the bottom of the page)

When you finish this exam, you must indicate that by submitting the Finished Exam 1 survey on Moodle.

Part 2, programming. You may use your notes, *Chez* Scheme, the three textbooks from the course, The *Chez* Scheme Users' Guide, any materials that I provided online for the course. You may do searches for built-in Scheme procedures, but not for the particular problems that you are solving. You may not use any other web or network resources, or programs written by other (past or present) students. You are allowed use any code that *you* have previously written. Assume that all of your procedures' input arguments have the correct types and values; your code does not need to check for illegal input data.

Mutation is not allowed unless I state otherwise for a particular problem.

Efficiency and elegance will not affect your score unless a particular problem statement says otherwise.
Be careful not to use so much time on one problem that you do not get to work on other problems.

C1. (10 points) `(alternating-reverse lol)` takes a list of lists `lol` as its argument. It returns a list of lists where every other inner list is reversed. The first inner list is reversed, the second one is not, etc. You are allowed to use Scheme's `reverse` procedure. **For full credit, your code must run in O(N +M) time, where N is the number of sublists, and M is the total of the lengths of all of the sublists.**

```
> (alternating-reverse '())
()
> (alternating-reverse '((a b c) (d e) (f g h) (i j)))
((c b a) (d e) (h g f) (i j))
> (alternating-reverse '((a b c) (d e) (f g h)))
((c b a) (d e) (h g f))
> (alternating-reverse '((a b) () (c d) (e) (f g) (h) (i j)))
((b a) () (d c) (e) (g f) (h) (j i))
```

C2. (9 points) `(member-n? sym n los)` takes a symbol, a positive integer, and a list of symbols `los`. Returns `#t` if `sym` occurs at least `n` times in `l0s`, and returns `#f` otherwise. This procedure should short-circuit if it finds `n` occurrences of `sym`; it should not continue traversing the list.

```
(member-n? 'a 1 '())              ➔ #f
(member-n? 'a 1 '(b))             ➔ #f
(member-n? 'a 1 '(a))             ➔ #t
(member-n? 'a 2 '(b a b a b))     ➔ #t
(member-n? 'a 1 '(b a b a b))     ➔ #t
(member-n? 'a 3 '(b a b a b))     ➔ #f
(member-n? 'b 3 '(b a b a b))     ➔ #t
```

C3. (8 points) `(opposites-attract ls)` takes a proper list as its argument. It returns a list of 2-lists. The first element of `ls` is paired with the last element, the second is paired with the next-to-last, …, the last is paired with the first.

```
(opposites-attract '(a b c d))   ➔ ((a d) (b c) (c b) (d a))
(opposites-attract '(a b c d e)) ➔ ((a e) (b d) (c c) (d b) (e a))
(opposites-attract '())          ➔ ()
(opposites-attract '(a))         ➔ ((a a))
```

C4. (9 points) In Assignment 3, we saw that a **relation** can be represented in Scheme by a list of 2-lists (lists of length 2). A relation said to be is ***symmetric*** if for every `(a b)` is in the relation, then `(b a)` is also in the relation. Write a Scheme procedure `(symmetric? rel)` that takes a relation `rel` and returns `#t` if `rel` is symmetric, but returns `#f` if `rel` is not symmetric. Of course, you may assume that `rel` actually is a relation; your code does not have to test for that.

Examples:

```
(symmetric? '())                                ➔ #t
(symmetric? '((a a)))                           ➔ #t
(symmetric? '((a b) (c b) (b a) (c c) (b c)))   ➔ #t
(symmetric? '((a b) (c b) (b d) (c c) (b c)))   ➔ #f
```

C5. (8 points) In Assignment 3, we saw that a **matrix** can be represented in Scheme by a nonempty list of nonempty lists of numbers, where all of the sublists have the same length. A matrix is **square** if the number of sublists (rows) is the same as the length of each sublist. A square matrix is **lower triangular** if all entries above the main diagonal are zero. Write a procedure `(lower-triangular? m)` that returns `#t` if square matrix `m` is lower triangular, and `#f` otherwise. You may assume that m actually is a square matrix.

Examples:

```
(lower-triangular? '((2)))                          ➔ #t
(lower-triangular? '((2 1) (0 3)))                  ➔ #f
(lower-triangular? '((2 0) (1 3)))                  ➔ #t
(lower-triangular? '((2 0 0) (3 4 0) (5 6 8)))      ➔ #t
(lower-triangular? '((2 0 0) (3 4 1) (5 6 8)))      ➔ #f
```

C6. (10 points for part a) (a) I have provided (in the starting code file) my solution code for the Binary Search Tree functions from Assignment 7. Each of them takes a BST as an argument; some return a new BST. None of them modify the tree that they are given as input. Suppose we want to have BST's be persistent objects, using the "Scheme procedure as object" approach that we used for stacks, ArrayLists, and slist-leaf-iterators. A call to `(make-BST)` produces such an object that represents an empty tree. The example code below (which is also one of the test cases for this problem) illustrates the various methods and what they do. You should not have to write a lot of code, since all of the hard work for the functionality is done in the given functions, which you may freely use. **You are allowed to use mutation for both parts of this problem.**

Examples

```
(let ([t (make-BST)]
      [t2 (make-BST)])
  (t 'insert 10)
  (t 'insert 5)
  (t 'insert 4)
  (t 'insert 7)
  (t 'insert 20)
  (t 'insert 3)
  (t 'insert 15)
  (t 'insert 18)
  (t 'insert 6)
  (t2 'insert 200)
  (t2 'insert 100)
  (list (t 'inorder) (t 'preorder) (t 'height)
    (t2 'inorder) (t 'contains? 15) (t 'contains? 50)))
```

Your code should begin

```
(define make-BST
  (lambda ()
```

```
(let ([t (make-BST)])
  (t 'insert 10)
  (t 'insert 5)
  (t 'insert 4)
  (t 'insert 7)
  (t 'insert 20)
  (t 'insert 3)
  (t 'insert 15)
  (t 'insert 18)
  (t 'insert 6)
  (t 'remove 5)
  (list (t 'inorder) (t 'preorder) (t 'height)
      (t 'contains? 15) (t 'contains? 50)))
```

Output from the examples should be: `((3 4 5 6 7 10 15 18 20) (10 5 4 3 7 6 20 15 18) 3 (100 200) #t #f)`
and `((3 4 6 7 10 15 18 20) (10 6 4 3 7 20 15 18) 3 #t #f)`

(13 points for part b) (b). Add a `remove` method to the BST class. If `t` is a BST that was created by `make-BST`,

1. if n is not in `t`, then `(t 'remove n)` replaces `t`'s BST by a BST that is `equal?` to `t`'s BST.
2. if `n` is in `t`, then `(t 'remove n)` replaces `t`'s BST by a BST that has all of `t`'s elements except `n`. The running time for this must be O(`(t 'height)`); use the standard BST deletion algorithm that you learned in CSSE 230. You can look up the algorithm if you wish. Producing the new tree with `n` deleted can be done without mutation (and the code may be easiest to write without mutatiob), but that is not a requirement of this problem**.**

   A few reminders about the deletion algorithm:
   If the node that contains `n` is a leaf or if it only has one child, then removing that node is very simple. If the node containing n has two children, the node that actually gets removed is either the inorder predecessor or inorder successor of `n`'s node (programmer's choice). For this exam, so that you should always get the same trees that I get, you must remove the inorder successor node.
   You are welcome to write multiple helper procedures. I did.
   Suggestion: Write `BST-remove`, with an interface that is similar to `BST-insert`. Test it in the non-object-oriented situation; when it is working, call it from the implementation of the `'remove` method.

When you finish this exam, you must indicate that by submitting the Finished Exam 1 survey on Moodle.