CSSE 304 Exam #1 Part 2 Dec21, 2020 (day 13.5)

Unless I announce otherwise, the Exam2 assignment on the PLC server will close at 9:00 After a few minutes I will reopen it so students with accommodations can have the extra time that they are given..

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.

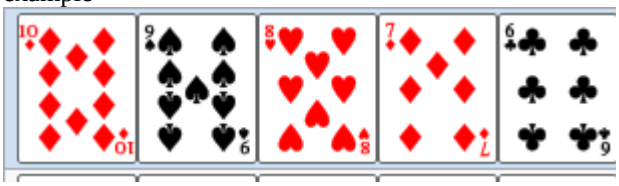
Starting code and offline test cases will be available on Moodle (in the Exams section)

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. You may 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. (15 points) `(straight? lon)` In the card game Poker, a *straight* consists of five cards whose numbers are consecutive, for example



For this problem, I abstract the "cards" to be positive integers, and I allow the "hand" to be a list of any positive length. I ignore the notion of suits. Suppose that k is the lowest number in an n-card hand. In order to be a straight, the hand must contain (in some order) the numbers k, k+1, …, k+n-1. For 10 points, pass all of the testcases (without customizing the code for those cases). For the full 15 points, your code's running time must be O(n log n), where n is the number of cards in the hand. [Hint: call `list-sort`]. For 5 bonus points (20 total), also do not use `if` or `cond` in your code.

```
(straight? '(4 2 3 1))            ➔ #t
(straight? '(4 2 3 1 6))          ➔ #f
(straight? '(3 1 2 4 3 1 5))      ➔ #f
(straight? '(13 11 9 10 12))      ➔ #t
(straight? '(1))                  ➔ #t
```

C2. (10 points) `(snlist-flatten slist)` using `snlist-recur`. In the starting code file, you will find my version of snlist-recur. Based on my `snlist-recur` definition or yours, use it to define `snlist-flatten`. As in the A9 `snlist-recur` problems, there can be no explicit recursion in the procedures that you pass as arguments to `snlist-recur`. In particular, these procedures may not call `snlist-flatten`, `snlist-recur`, or any recursive procedure that you write.

```
(flatten '(() (a ((b ((c)) (d e (f (g)))))))      ➔ (a b c d e f)
(flatten '(f () ((b d (((d) e))) () r)))          ➔ (f b d d e r)
```

C3. (12 points) `(bt-max-interior bt)` Same as the procedure in A7, except that if there is tie, the rightmost tied interior node wins. By rightmost, I mean the one that would appear latest in an inorder traversal of `bt`. As before, you may not traverse any subtree twice (such as by calling `bt-sum` or `reverse` at every interior node, and you may not use mutation (you will earn zero points if you do either of those). You will lose some points if you customize your code to the given test cases (for example, putting into your code a specific number that is more negative than any of the negative numbers in the test cases).

```
(bt-max-interior '(a 2 (c (d -1 -2) (e -1 -2))))           ➔ e
(bt-max-interior '(a 0 (b 7 (c (d -2 -1) (e -3 -4)))))     ➔ d
(bt-max-interior '(a (b (c (d -2 -1) (e -3 -4)) 7) 0))     ➔ a
 bt-max-interior '(a 0 (b 2 3)))                           ➔ b
(bt-max-interior '(a (b 2 3) 0))                           ➔ a
(bt-max-interior '(a 2 (b (c -1 -1) (d -1 -1))))           ➔ d
(bt-max-interior '(a (b (c -1 -1) (d -1 -1)) 2))           ➔ a
(bt-max-interior '(a (b -2 (d -1 -1)) (c (e -1 -1) -2)))   ➔ e
```

C4. (12 points) In the homework, you wrote the code-transformation procedures `let->application` and `let*->let`. Here you are to write `named-let->letrec`. It takes a syntactically correct named `let` expression and translates it into the corresponding `letrec` expression. As in the homework problems, you only need to translate the top-level expression; you do not need to look inside at any sub-expressions for additional named `let`s to translate. Your procedure's return value does not have to be formatted just like mine, but it does have to be `equal?` to mine.

```
> (named-let->letrec
    '(let fact ([n 5] [acc 1])
       (if (zero? n) acc (fact (sub1 n) (* n acc)))))
(letrec ([fact (lambda (n acc)
                 (if (zero? n) acc (fact (sub1 n) (* n acc))))])
  (fact 5 1))
> (named-let->letrec
    '(let loop ([start 5])
      (display start)
      (set! start (sub1 start))
      (if (>= start 0)
         (loop start))))
(letrec ([loop (lambda (start)
                 (display start)
                 (set! start (sub1 start))
                 (if (>= start 0) (loop start)))])
  (loop 5))
```

C5. (13 points) In an in-class live demo, we wrote part of the code for `make-array-list` which creates an "object" whose behavior is similar to a Java `ArrayList` object. I am providing that code for you. **It uses mutation, and so can your code.** There were two parts that we did not do in class. Suppose `a` is an `array-list` object:

(a) Add a given item at a specified position. `(a 'add 5 3)` adds the item 5 to `a` at position 3. Your code does not have to check that the position is valid.

(b) Remove the item from a specified position. `(a 'remove 4)` removes and returns the item from position 4 of `a`. Your code does not have to check that the position is valid. Note that in the starting code, vector positions that are not part of the array-list are filled by #f. This should be true after a call to `remove`.

```
> (let ([al (make-array-list)])
  (al 'add 5)  ; add at end
  (al 'add 4)
  (al 'add 3)
  (al 'add 2)
  (al 'add 7 1); add at index 1
  (al 'show))
((size 5) (capacity 6) (v #(5 7 4 3 2 #f)))
> (let ([al (make-array-list)])
  (al 'add 5)  ; add at end
  (al 'add 4)
  (al 'add 3)
  (al 'add 2)
  (al 'add 7 1); add at index 1
  (al 'add 6 2)
  (al 'add 8 3)
  (al  'show))
((size 7) (capacity 12) (v #(5 7 6 8 4 3 2 #f #f #f #f #f)))
> (let ([al (make-array-list)])
  (al 'add 5)  ; add at end
  (al 'add 4)
  (al 'add 3)
  (al 'add 2)
  (al 'add 7 1); add at index 1
  (al 'add 10 3)
  (al 'remove 2)
  (al 'show))
((size 5) (capacity 6) (v #(5 7 10 3 2 #f)))
> (let ([al (make-array-list)])
  (al 'add 5)  ; add at end
  (al 'add 4)
  (al 'add 3)
  (al 'add 2)
  (al 'add 7 1); add at index 1
  (al 'add 10 3)
  (al 'remove 2)
  (al 'remove 3)
  (al 'show))
((size 4) (capacity 6) (v #(5 7 10 2 #f #f)))
```