CS116 Winter 2013

Assignment 03
Due 10 am on Wednesday, February 6

Instructions:

- This assignment is testing your knowledge of material from Module 3. For full marks, each question must use accumulative recursion in a non-trivial way. Some questions may require more than one accumulator.
- For this and all subsequent assignments, you are expected to use the design recipe for all functions that you write including contract and purpose for local helper functions.
- Do **not** use abstract list functions. For 'simple and short' functions that are one time use only, use `lambda`. You may use helper functions that are defined locally.
- Do not use `reverse` or built in sorting functions unless otherwise stated.
- Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include reference to the parameter names of your functions.
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources. **Do not** email or share any of your code with fellow students.
- Do not send any code files by email to your instructors or tutors. It will not be accepted by course staff as an assignment submission. Course staff will not debug code emailed to them.
- Test data for all questions will always meet the stated assumptions for consumed values.
- Read the course web page for more information on assignment policies and how to organize and submit your work.
- Download the interface file from the course Web page.
- Follow the instructions in the style guide. Specifically, your solutions should be placed in files `a03qY.rkt`, where `Y` is a value from 1 to 4.
- For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe, including the definition of constants and helper functions where appropriate.
- Read each question carefully for restrictions and before posting questions on piazza.

Language level: Intermediate Student with lambda.
Coverage: Module 3

Structure definition for Question 1:

```
(define-struct trans (action amount))
;; A trans is a structure (make-trans a amt),
;; that represents a transaction for a bank account, where
;; a is a symbol, either 'deposit or 'withdraw. A deposit is a
;; payment made into the account. A withdrawal is a payment
;; made out of the account.
;; amt is a positive number representing the amount added
;; or removed from the account.
```

Structure definition for Question 3:

```
(define-struct student (id cav passed failed))
;; A student is a structure (make-student id cav p f)
;; id is a four digit natural number (student id number)
;; cav is a number between 0 and 100 (student's cumulative average)
;; p is a non-negative number (number of credits student passed)
;; f is a non-negative number (number of credits student failed)
```

1. Use accumulative recursion to write a function named `update-balance` that consumes a list of transactions that took place during the month (`lot`), and a starting balance at the beginning of the month (`start-bal`). The function produces the balance of a bank account after processing all of the transactions in `lot`. The balance may drop below zero at any point. You may assume that deposits and withdrawals will have no more than two decimal places and thus the balance produced by `update-balance` will have no more than two decimal places.
   For example:

```
(update-balance
   (list (make-trans 'withdraw 604.34)
         (make-trans 'deposit 300)) 0) => -304.34
(update-balance
   (list (make-trans 'withdraw 20.99)(make-trans 'deposit 60)
         (make-trans 'deposit 900)(make-trans 'withdraw 55.5)
         (make-trans 'deposit 100)(make-trans 'deposit 40))50)
=> 1073.51
```

2. Use accumulative recursion to write a function named `repeat`, that consumes a string, `str` and produces a string that has each character in the string repeated twice. For example `(repeat "Nice.")` produces `"NNiiccee.."`. If the consumed string is empty, `repeat` produces the empty string. You may use the list operation `reverse` for a more efficient solution to this question, but are not required to do so.

3. This question uses the student structure. Write a function named `class-avg-range` that consumes a list of students in a class, `class-list`. The function produces a list of size two, where the first element is the class average for cumulative averages across all students. The second element in the produced list is the range between the highest and lowest outliers from the class average. You may assume that the `class-list` is non-empty.
   To receive full marks for this question you must traverse the list only once. Therefore, gather all of the information you need with only one pass through the list. Abstract list functions are not permitted.
   For example:

```
(class-avg-range
  (list (make-student 1112 78.5 5 1)))
   => (list 78.5 0)
(class-avg-range
  (list (make-student 3128 61.6 10 4)(make-student 3123 55.3 8 5)
        (make-student 4166 82.4 13 1)(make-student 3146 89.4 14 0)
        (make-student 3242 50.5 8 6) (make-student 4936 60.1 9 5)))
   => (list 66.55 38.9)
```

4. The common card game **Go Fish** has an adapted childrens version known as **Go-Fish-for-Fun**☺. In this game each player tries to have as many cards in a set as possible (up to 4). When one player reaches four cards of the same number the game ends. However, the winner is the player that gets the highest score. Scores are calculated as follows: Cards which occur at least three or four times in a hand count towards the score. Cards which appear zero, once or twice in a hand do not count towards the score. Card values begin at 2 and end at 14. Each card is given its value multiplied by the number of occurrences of the card in the player's hand. For example a player with the cards (2 2 2 5 8 8 10 10 10 10 14) receives 46 points. This is because 2 occurs 3 times, and 10 occurs 4 times, so the total is 2*3+4*10=46.
   A player with the cards (2 3 5 7 7 7 9 11 12 12 12 13) receives 57 points.

Use accumulative recursion to complete the scheme function `go-fishff-points`, that consumes a list of numbers between 2 and 14 inclusive (called `card-list`), and produces a natural number that is the total points for `card-list`. You may assume the `card-list` will always be in ascending order.

For example,

- `(go-fishff-points empty) =>  0`
- `(go-fishff-points (list 3 3 4 7 7 8 8 12 13)) => 0`
- `(go-fishff-points (list 4 6 6 7 7 9 9 9 9 12 14)) => 36`