Functional and Logical Programming

Exercise 2 – Lists and Recursion

General Guidelines:

Submission deadline is **Wednesday, October 12, 23:55**
Submit your answers as a single RKT file named ex2-YourID.rkt,
for example: ex2-012345678.rkt
Post the RKT file in the submission page in course website.
Do NOT pack the file as an archive (no ZIP/RAR/anything).
Do not submit any additional file, or use any other file format.

- No late submission will be accepted! (Submission page will automatically close)
- You should work on your exercise by yourself. Misconducts will be punished harshly.
- Place a comment with your ID at the top your code file.
- Unless specifically noted, you may assume that the input is always correct
  (Your functions should not check for input parameters validity).
- You must **never** change the interface of the functions!
- Do not use the built-in **map** function in this exercise.

Part 1 – List Manipulation (30 points: 15 + 15)

You are not allowed to use the built-in **map** function in this part

A. The function **(cars lists)** receives a list of lists named **lists** (that is, **lists** is a list whose elements are lists themselves). The function returns a list in which only the first element from each one of the lists in **lists** appears, according to the order in **lists**.

   Examples:
   (cars '((1 2 3) (4 5 6) (7 8 9))) ; returns (1 4 7)
   (cars '((a b c) (d e f) (g h i))) ; returns (a d g)
   (cars '(((1) (2) (3)) ((4) (5) (6)) ((7) (8) (9)))) ; returns ((1) (4) (7))

   Write an implementation for the function **cars**.

B. The function **(cdrs lists)** receives a list of lists named **lists** (that is, **lists** is a list whose elements are **lists** themselves, just like the previous part). The function returns a list in which only the remaining of the lists, without the first elements from each one of the lists in **lists** appears, according to the order in **lists**.

   Examples:
   (cdrs '((1 2 3) (4 5 6) (7 8 9))) ; returns ((2 3) (5 6) (8 9))
   (cdrs '((a b c) (d e f) (g h i))) ; returns ((b c) (e f) (h i))
   (cdrs '(((1) (2) (3)) ((4) (5) (6)) ((7) (8) (9)))) ; returns (((2) (3)) ((5) (6)) ((8) (9)))

   Write an implementation for the function **cdrs**.

Part 2 – Sorting (40 points: 25+15) +10 optional bonus

Do not use the built-in (quicksort lst pred).
Important note on edge cases when sorting:

- Duplicate elements are not interesting for this part, you do not have to handle this case at all and we will not test for it. (so a list such as (1 2 4 4 5 6) is not a valid input, (1 2 3 4 5 6) is a valid input, it is also sorted, but still a valid input)

A. Implement **(quick-sort lst)**

A reminder on how quicksort works:

1. Pick an element from the list (called pivot)
2. Reorder the list so that all elements smaller than pivot, and the pivot itself are on the left, larger are on the right
3. Rinse and repeat on left and right sides

When quick-sorting, the pivot selection is crucial, as incorrect selection can lead to worst case execution running time (think about the case of when the list is already sorted)

For part A, we will not implement any special logic for pivot selection, pick the leftmost element as a pivot to simplify the implementation.

When implementing, think about how to split the lists and then rejoin them while not losing the pivot.

Guidelines:

- Implement a nested function called `(pivot lst)` that returns the pivot element to work with.
- Implement a nested function called `(split lst pivot)` that will receive a list and the pivot element and will return a list of two lists where numbers smaller than or equal to the pivot are on the first list and numbers bigger than the pivot are on the second list.
  Example:
  if lst is (2 3 1 4 5 6)
  (split lst 4) will return the list ((2 3 1 4) (5 6))
  [order of elements in result lists is not important]
- Implement the main sort logic, that uses the above two functions to reclusively quick-sort the list. Think of when the recursion ends, think on what to send to the recursive function each time and how to build the final input after the recursion ends.

B. Now that you have a working, basic quicksort, we will implement smarter pivot selection. The function name for this section should be **(quick-sort-with-pivot lst)**

Duplicate the code from part A.

Optimize your solution by writing a smarter `(pivot lst)` function instead of the old one.
The `(pivot lst)` function will return the pivot element to work with when quicksorting.

I. Design pivot so it chooses the middle index of lst (10 points)
There can be more than one correct answer, all answers will be accepted.
(think about even list length, which one is the middle index?)

II. **Bonus:** offer a different optimization (up to 10 points)
To get any points for this bonus, aside from implementation, you must explain (in a comment) what your optimization does. Your explanation should use formal language.

Important submission instructions:
If you do the bonus, duplicate the code and call it `(quick-sort-bonus lst)`, do not forget to explain as instructed above.
Regard the bonus:
Any solution will be reasonable and valid as long as you can explain why.
Write a short explanation, try to use formal terms.
If you submit a solution that picks the last element (it is a different solution, but not really an optimization) it will not receive any points.

Part 3 is on the next page

Part 3 – Tail Recursion (30 points, 10+10+5+5)

All the functions in this part must be written as tail recursions. If you fail doing so, you will lose a large portion of your grade.

A. Write the function `(numOfBitsOn number)`
The function receives a number and returns the number of bits that would have been required to be "on" in order to represent the input number in binary base.
For example, the number 5 is represented as 101 in binary and therefore requires two bits to be "on".

Example:
(numOfBitsOn 5) will return 2 because 5 in binary is 101
(numOfBitsOn 101) will return 4 because 101 in binary is 1100101

B. Write the function `(findSqrt n delta)`
The function returns the estimated square root of n.
delta defines how 'good' does the estimate have to be
Use binary search to find the square root, using the following algorithm:
1. Start with a guess = n/4, min = 0, max = n/2.
2. If ($|guess^2 – n| < delta$), then return guess (it is a good enough estimation)
3. If ($guess^2 – n < 0$), then: (the guess is too low)
   Try again with guess = (guess + max) / 2, min = guess, max remains the same
4. Else: (the guess is too high)
   Try again with guess = (guess + min) / 2, min remains the same, max = guess

- Use the function `(abs x)` to get the absolute value of `x`.
- You may assume that `n` is a positive integer.

Remember: this should be implemented as a tail recursion.

Example:
(findSqrt 16 0.2) will return 4
(findSqrt 80 1) will return 8 29/32 (almost 9)
(findSqrt 80 0.2) will return 8 121/128 (closer to 9)
(findSqrt 80 0.00000000002) 8 1038237944827/1099511627776 (we can call it 9, but it cannot really return 9!)
(findSqrt 81 0.00000000002) 9 9/35184372088832 (this is how it looks from the other side, this happens because computers do not deal well with floating point operations, you can read about it here: http://bit.ly/1qlDn43)

In part 1 you implemented **cars** and **cdrs**. Write their implementation as tail recursion

C. Implement **cars-tail** as tail recursion version of **cars** from part 1.

D. Implement **cdrs-tail** as tail recursion version of **cdrs** from part 1.

* if you already implemented them as tail originally, paste the same implementation in the '-tail' version, you must have the functions present with these names when submitting or they will not be graded!