IDC Herzliya 11/11/2014

Functional and Logical Programming

Exercise 3 – Lambda, Arguments, Factories

General Guidelines:

Submission deadline is **Wednesday, November 19, 23:55**
Submit your answers as a single RKT file named ex3-YourID.rkt,
for example: ex3-012345678.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!

Part 1 – Quicksort again (40 points)

Write the function `(quick-sort pred lst)`
`lst` is the list of numbers to be sorted
`pred` is the predicate by which the list is ordered, the signature of this predicate is:

```
(lambda (x y) …)
```

Usage examples:

- `(quick-sort < lst)` will sort ascending (small to large)
- `(quick-sort > lst)` will sort descending (large to small)
- `(quick-sort (lambda (x y) (< (car x) (car y))) lst)` will sort a list with inner lists according to the first element of the inner list, ascending.

  We assume the input will always match the predicate logic.

Details:

We expect you to re-implement it and not submit the same implementation from the previous exercise.

- **You are expected to use (`filter`). Not using it will cause point reduction**
- Pivot selection wise, you may choose any way you want, you will not be graded for pivot optimization in this exercise.
- You are graded for the correctness of the sort (you are implementing quicksort, just a reminder)
- The edge case of identical numbers is still not important. (i.e their location after the sort)
- Double check that you are set to Pretty Big Custom settings as instructed at the beginning of the course
- Also be sure to call the function (quick-sort) with the dash, because there exists a (quicksort) in Racket, we are not using it.

Part 2 – Factories (46 points)

For this part, the length of the input list is always even!

A. Write the function `(do2add lst)`
   `lst` is a list with numbers

   The function will add numbers in pairs
   Example: `(do2add '(1 2 3 4 5 6))` -> (3 7 11)

B. Write the function `(do2F F lst)`
   `lst` is a list with numbers

   The function will apply `F` on each two consecutive elements
   `F` signature is `(lambda (x y) ..)`
   Example:
   `(do2F (lambda (x y) (- x y)) '(1 2 3 4 5 6))` -> (-1 -1 -1)

C. Write the function `(makeDo2F F)` that gets `F` as a parameter and returns a function with the parameter `lst` that applies `F` on each two consecutive elements of `lst`
   Example:
```
(let ((f (makeDo2F *)))
  (equal? (f '(2 3 4 5)) '(6 20))
```
   Will return #t

D. Using the function `(makeDo2F F)` written in C, write the following functions:

   I. `do2addFactory` – as in A.
   II. `do2mult` – accepts a list of numbers, multiplies two consecutive elements.
   III. `do2eq?` – accepts a list of symbols, checks if consecutive elements are equal.
   IV. `do2eq1st` – accepts a list of lists, checks if the first element of the inner lists is equals for each two lists.

   Examples:
```
> (do2addFactory '(1 2 3 4 5 6))
(3 7 11)
> (do2mult '(1 2 3 4 5 6))
(2 12 30)
> (do2eq? '(a b b b c d))
(#f #t #f)
> (do2eq1st '((2 3) (2 4) (3 4) (4 5) (6 7) (6 5)))
(#t #f #t)
```

(Part 3 is at the next page)

<u>Part 3 – Multiple arguments</u> (14 points)

Write the function `(makeDo2FM F)`
It will do the same as Part 2.C, but support multiple arguments instead of a list.
A minimum of 2 arguments is required, and the number of arguments will always be even.

Guidelines:

- Do not use `(do2F F lst)`, you should build a new lambda for this.
- Think about what lambda the factory should return
- For example, we can implement `do2mult` with this factory and then use it:
  First define the new do2mult
  `(define do2multM (makeDo2FM *))`
  Then use it
  `(do2multM 1 2 3 4 5 6)` and it will return: (2 12 30)