Homework: Optional Assignment 1

Due-date: Mar 31 at 11:59pm
Submit online on Canvas

Homework must be individual's original work. Collaborations and of any form with any students or other faculty members are not allowed. If you have any questions and/or concerns, post them on Piazza and/or ask 342 instructor or TAs.

This is an optional assignment. If you do this assignment, the points you get for this assignment will be considered as extra-credits, and can contribute at most 8 *points to your overall grade.*

Learning Outcomes

- Knowledge and application of Functional Programming
- Ability to understand grammar specification
- Ability to design software following requirement specifications (scheduling problems)

Questions

1. You are the keymaster for the Ames Badminton Club (ABC). Different groups send requests to you that they would like to play at certain times for certain duration at your club. That is, each request has a start-time and finish-time (based on them, you can infer the duration of the request as well).

   Due to this time-limit it may not be possible service all the requests as the ABC has only one Badminton court. Your objective is to serve (pun intended) as many requests as possible.

   You are tasked with writing a program which will generate schedules for requests based on the following strategy. You will order the services following one of the three choices: request r_i appears before r_j in the order if

   (a) start-time of r_i is earlier than start time of r_j;

   (b) finish-time of r_i is earlier than the finish time of r_j;

   (c) duration of r_i is shorter than the duration of r_j.

   We will call the above *strategy-choice*.

   After you have the order, you will select the requests to create a schedule by the following process:

   Step1: select the first element in the order (let us refer to it as r_0) and add it to your schedule.

   Step2: select the first r_i from the order that does not overlap with the additions you have already made to the schedule, add the r_i to your schedule.

   Step3: repeat the above step till there is no request to select.

Implement the above strategy in Racket and include a function `getreqlst` such that it takes as input

- a list of requests, where each request is a list of pair of elements corresponding to start-time and end-time of the request;
- a strategy-choice, which can be either `'st` (earliest start-time), `'ft` (earliest finish-time) and `'sh` (shortest duration)

and produces an output as a list of requests in the schedule.

For instance, for the following

```
(define requests
  '((0 8) (8 11)
    (1 3) (4 13) (5 6)
    (3 10) (7 9) (10 12)))

> (getreqlst requests 'st)
'((0 8) (8 11))
> (getreqlst requests 'ft)
'((1 3) (5 6) (7 9) (10 12))
> (getreqlst requests 'sh)
'((5 6) (1 3) (7 9) (10 12))
```

The example run of steps for `(getreqlst requests 'st)` is as follows:

- The sorted order as per strategy-choice `'st` is as follows: `'((0 8) (1 3) (3 10) (4 13) (5 6) (7 9) (8 11) (10 12))`
- The first request selected is `(0 8)`.
- The second request that does not overlap with the already selected requests is `(8 11)`.
- No more selection is possible.

(*You can use the built-in sort function in Racket for this assignment.*)

2. We have already learned about lambda calculus. We will use the following grammar for representing lambda expressions in Racket:

```
Expr -> Var | (lambda Var Expr) | (Expr Expr)
Var  -> Symbol
```

In the above, consider `Var` to be any symbol. For instance, the following are valid lambda expressions in Racket as per the above grammar.

```
(define e1 'x)  ;; just a variable

(define e2 '(lambda x x)) ;; identity function

(define e3 '(lambda f (lambda x (f x)))) ;; lambda abstraction

(define e4 '(  (  (lambda f (lambda x (f x)))
                  g)
               z)
)  ;; two lambda applications
```

You will write a function `lsem` in Racket, which takes as input a lambda expression, and β-reduces the expression repeatedly until it cannot be β-reduced any further. The input to your function will be always valid lambda expressions (as per the above grammar). You do not have to implement α-conversion and manage possible non-termination in β-reductions due to presence of self-replicating expressions.

The following presents expected results of your implemented functions.

```
> (lsem '((lambda x x) y))
'y

> (lsem '(((lambda f (lambda x (f x))) g) z))
'(g z)

> (lsem '( (lambda n (lambda f (lambda x (f ((n f) x)))))
           (lambda g (lambda y y))
         )
  )
'(lambda f (lambda x (f x)))
```

Programming Rules

- You are required to submit one file hwopt1-⟨net-id⟩.rkt[1]. The file must start with the following.

```
#lang racket
(require "tests.rkt")
(provide (all-defined-out))
```

In the above, the tests.rkt will be used as an input file for our test programs. For instance, it may contain the following tests.

[1] Your netid is your email-id and please remove the angle brackets, they are there to delimit the variable net-id.

```
#lang racket
(provide (all-defined-out))

(define requests
  '((0 8) (8 11)
    (1 3) (4 13)
    (5 6) (3 10)
    (7 9) (10 12)))
```

- You are **only allowed** to use functions, if-then-else, cond, list operations (**you can use the built-in sort function**), operations on numbers. No imperative-style constructs, such as begin-end or explicitly variable assignments, such as get/set are allowed. If you do not follow the guidelines, your submission will not be graded. If you are in doubt that you are using some construct that may violate rules, please contact instructor/TA (post on Piazza).
- You are expected to test your code extensively. If your implementation fails any assessment test, then points will be deducted. Almost correct is equivalent to incorrect solution for which partial credits, if any, will depend only on the number of assessment tests it successfully passes.
- The focus is on correct implementation. In this assignment, we will not assess the efficiency; however, avoid re-computations and use tail-recursion, if possible.

Postscript. It is not necessary to read this for successfully completing the assignment. This section is intended to provide with some additional context/knowledge.

The first problem is an example of classic job-scheduling problem and the strategies you are implementing are called *greedy strategies*. For job scheduling, one of the discussed strategies is guaranteed to give you the optimal solution, i.e., applying that strategy you are guaranteed that you will be able to schedule maximum number of jobs (requests in your case). You will learn more about such greedy strategies, and this and other job scheduling problems when you learn about algorithms.