CS 116 Winter 2013

Assignment 04

Due Wednesday, February 13 at 10am

- The design recipe has changed with the inclusion of mutation. Read the Style Guide for Mutation in Scheme.
- Unless otherwise stated in the question, new state variables (beyond those defined in the interface file) may only be defined for testing purposes.
- You may use abstract list functions or recursion in your solutions, if needed.
- When a short and simple function is used only once, for example as a helper to an abstract list function, you must use lambda. For more complex helper functions, you may define them locally for readability.
- For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch, including contracts and purposes for local helper functions.
- 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.
- Follow the instructions in the style guide. Specifically, your solutions should be placed in files `a04qY.rkt`, where Y is a value from 1 to 4.
- Download the interface file from the course Web page.
- 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.

Coverage: Module 4
Language level: Advanced Student Scheme

The following structure is used Questions 1 and 2:

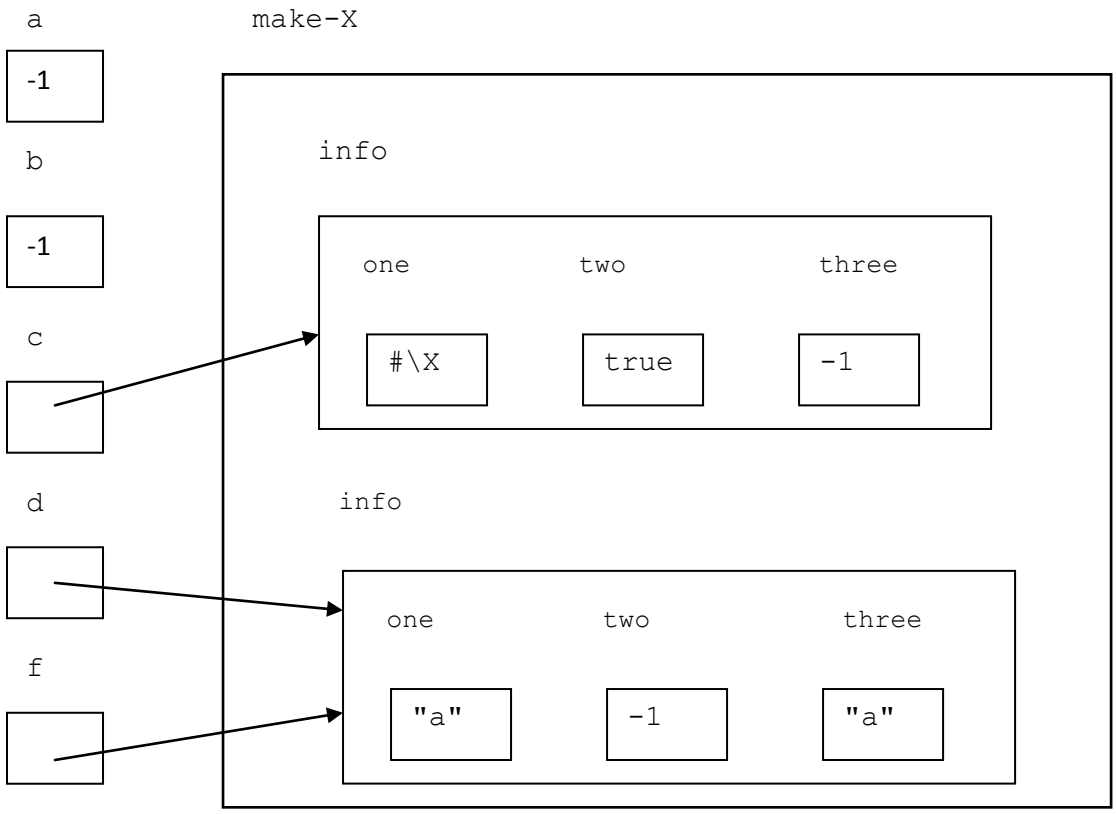
```
(define-struct info (one two three))
;; An info is a structure (make-info on tw th),
;; where on, tw, th can be values of any type.
```

1. In this question, you will complete two Scheme functions `part-A` and `part-B`, both of which consume no parameters and produce `(void)`. When following the design recipe for this question, you need only provide one example and one test for each of these two functions. The interface file defines several variables for this question (be sure to copy these definitions into your solution, and do not add new ones, except for testing purposes). You will write code to assign certain values to these variables.

   Complete the Scheme function `part-A`, which assigns values to `a`, `b`, `c`, `d`, `f` to match the following picture:

Assignment 04

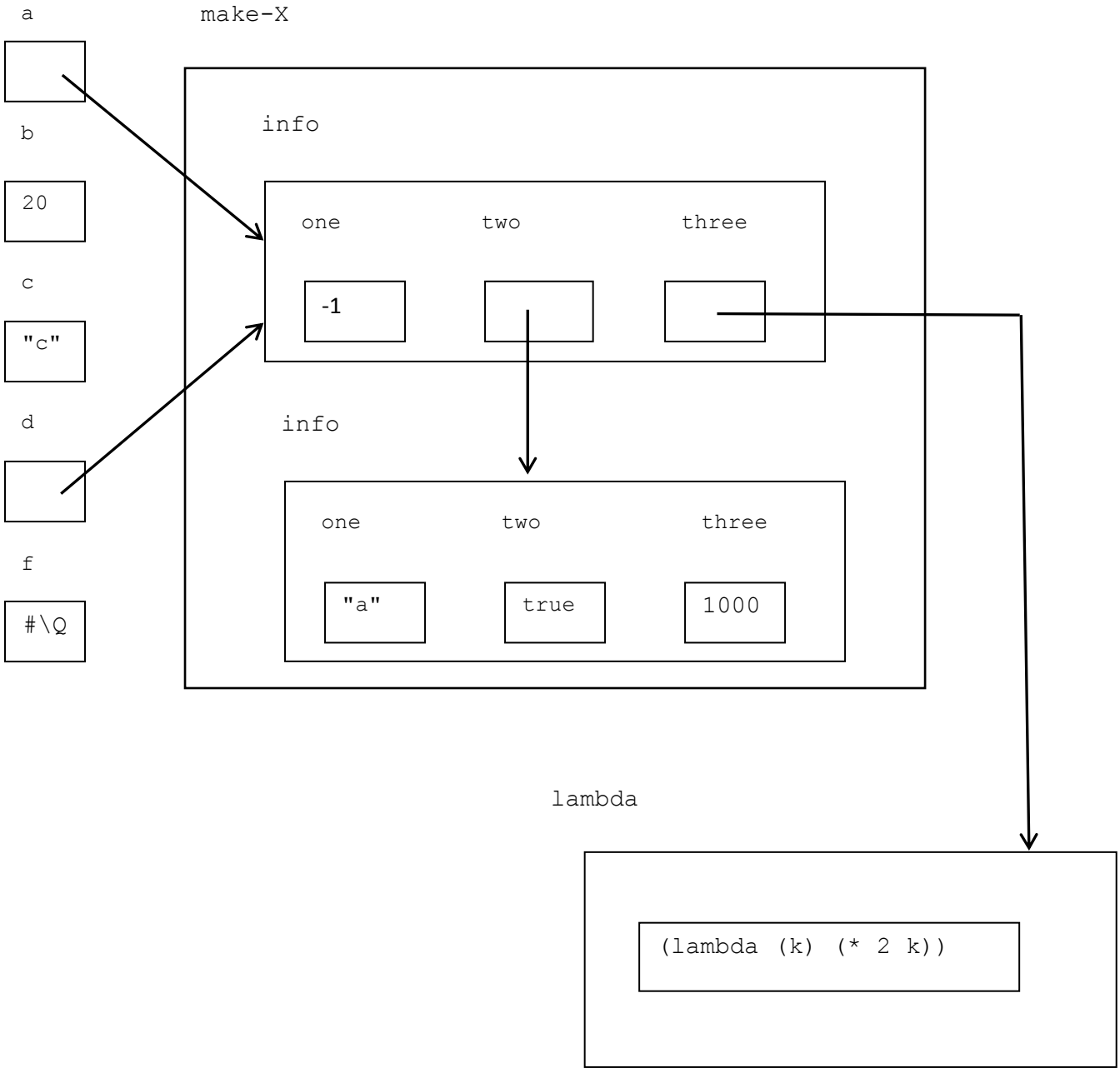
Due Wednesday, February 13 at 10am



Complete the Scheme function `part-B` which modifies the state variables from their values in `part-A` to the values indicated in the following diagram. You are **not** allowed to use any `make-info` calls in your solution to `part-B` or to declare any new variables (except for testing purposes). You must use the `info` field mutator functions to make any needed changes. Assume that `part-A` works (but do not call it in the body of `part-B`). We will test your solution to `part-B` using a correct version of `part-A`.

Assignment 04

Due Wednesday, February 13 at 10am



2. Write a Scheme function `update-info` that consumes an `info` structure (called `data`, in which all field values are numbers), and a nonnegative number (called `upper-bound`). The function produces the number of fields in `data` which are negative or are strictly greater than `upper-bound`. In addition, all fields of `data` which are negative are replaced with 0, and all fields of `data` that are greater than `upper-bound` are replaced with `upper-bound`. Use field mutation operations to update `data` – you may not create any new `info` structures in `update-info`. For example,

```
> (define d (make-info 11 9 -7))
> (update-info d 10)
2
> d
(make-info 10 9 0)
```

Assignment 04
Due Wednesday, February 13 at 10am

The next two questions are based on the game Rock-Paper-Scissors (called *RPS*, for simplicity). The following is a description of the basic game:

- *RPS* is played by two players.
- A match is made up of a series of rounds.
- In a single round, both players simultaneously choose one of the following finger gestures: rock, paper, scissors. If the players make the same choice, it is a tie. Otherwise, the winner is chosen according to the following rules:
  - Rock beats scissors,
  - Scissors beats paper, and
  - Paper beats rock.
- One point is awarded to a player for a win, 0.5 points for a tie, and 0 points for a loss.

The structure `RPS-player` maintains, for each player, the player's name, the number of points in the current match, and the length of their current winning streak (the number of rounds *in a row* that they have won outright) in the current match.

```
(define-struct RPS-player (name points current-streak))
;; A RPS-player is a structure (make-RPS-player n p c) where
;; * n is a nonempty string, for the player's name
;; * p is a nonnegative number, for the number of points the player
;;   has in the current match
;; * c is a nat, for the number of rounds in a row that this player has
;;   won in the current match
```

The structure RPS-match maintains, for each match, two player structures (`player1` and `player2`), and the number of rounds in this match so far.

```
(define-struct RPS-match (player1 player2 num-rounds))
;; A RPS-match is a structure (make-RPS-match p1 p2 n) where
;; * p1 and p2 are of type RPS-player, corresponding to
;;   player 1 and player 2,
;; * n is a nat, for the number of rounds completed in this match.
```

The following state variables are maintained for our *RPS* environment –

- `current-match`, of type `RPS-match`, corresponding to the currently active match,
- `past-matches`, of type `(listof RPS-match)`, which contains all completed matches (with the most recently completed match at the front) with at least one round played, and
- `highest-rate`, a value between 0 and 100, corresponding to the highest *points rate* for any player in a *completed* match. A player's rate in a match is defined as

$$rate = 100 \cdot \frac{points\ in\ the\ match}{rounds\ in\ the\ match}$$

3. Write a function called `new-players` that consumes two nonempty strings (`name1` and `name2`, names for `player1` and `player2` respectively, in a new match), and produces `(void)`. The function performs the following steps:
   - If `current-match` has more than 0 completed rounds,
     - Add it to the front of `past-matches`,
     - Update `highest-rate`, if necessary.
   - Create a new `RPS-match` object for the new players (creating new `RPS-player` values, setting their points and current streaks to 0), and set this as the new value of `current-match`.

For example, if `current-match` is
`(make-RPS-match (make-RPS-player "A" 4 2) (make-RPS-player "T" 1 0) 5)`,
`past-matches` is
`(list (make-RPS-match (make-RPS-player "L" 1 1)(make-RPS-player "C" 3 0) 4))`,
and `highest-rate` is 75,
then calling `(new-players "M" "R")`, sets `current-match` to
`(make-RPS-match (make-RPS-player "M" 0 0) (make-RPS-player "R" 0 0) 0)`,
`past-matches` is updated to

```
(list (make-RPS-match (make-RPS-player "A" 4 2) (make-RPS-player "T" 1 0) 5)
      (make-RPS-match (make-RPS-player "L" 1 1)(make-RPS-player "C" 3 0) 4))
```

and `highest-rate` is set to 80.

Note:

- Your solution must create a new `RPS-match` and `RPS-player` structures. Do not mutate the fields of `current-match`.
- Remember that the order in which you perform actions is very important as state variables are changing.

4. Write a function called `update` that consumes two legal *RPS* choices, `play1` and `play2` (i.e. one of `'rock`, `'paper`, `'scissors`), and produces `(void)`. The function mutates the number of rounds in `current-match` by one, and then updates the number of points for each player based on the rules of the game. Also, it updates the length of the `current-streak` for each player (increase the winner's streak by 1, and set the loser's to 0). In the case of a tie, both streaks are set to 0.

   For example, if `current-match` is
   `(make-RPS-match (make-RPS-player "M" 4 2) (make-RPS-player "A" 1 0) 5)`,
   then `(update 'rock 'paper)`, mutates `current-match` to
   `(make-RPS-match (make-RPS-player "M" 4 0) (make-RPS-player "A" 2 1) 6)`.
   Calling `(update 'scissors 'scissors)`, then mutates `current-match` to
   `(make-RPS-match (make-RPS-player "M" 4.5 0)(make-RPS-player "A" 2.5 0) 7)`.

   Notes:

   - The parameters `play1` and `play2` correspond to the choices made by `player1` and `player2` respectively.
   - Do not create any new structures in the body of `update`. All the updates must be made by mutating the fields of `current-match`.
   - No state variables other than `current-match` are modified in `update`.