Homework #1: Intro

Out: Thursday, November 10, Due: November 24 , 23:55

Administrative

The purpose of this homework is to familiarize you with the various tools that will be used throughout the course, and to get a feeling of basic programming in Racket (pl). In this particular homework, you will generally be graded on contracts (declarations) and purpose statements, other comments, style, test quality, etc. Correctness will play a very small role here, since everyone is expected to be able to solve these questions.
The first thing you will need to do is to download and install Racket and then the course plugin. You might want to consult *How to Design Programs* and the Class Notes before writing your code.
For this problem set, you are required to set the language level to the course's language, by beginning you file with `#lang pl`.
This homework is for individual work and submission.
The code for all the following questions should appear in a single .rkt file named <your ID>_1.rkt (e.g., 333333333_1.rkt for a student whose ID number is 333333333).
Do not submit multiple files. Do not compress your file.

Integrity: Please do not cheat. You may consult your friend regarding the solution for the assignment. However, you must do the actual programming and commenting on your own!! This includes roommates, marital couples, best friends, etc… We will be very strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Comments: Submitted code for each question should include **at least two lines of comments with your personal description of the solution**, the function and its type. In addition, **you should comment on the process of solving this question** – what the main difficulties were, how you overcame them, how much time you invested in solving it, did you need to consult others. **A solution without proper comments may be graded 0.**

Tests: For each question, you should have enough test cases for complete coverage (DrRacket indicates covered expressions with colors for covered and uncovered source code, unless your code is completely covered). See below on the way to create tests. Note that covering the whole code is not sufficient and you need have diversity in your tests and to cover all end-cases.

Important: Your tests should cover your whole code; otherwise the server will heavily penalize your submission. You should not have any uncovered expressions after you hit "Run" — it should stay at the same color, indicating complete coverage. Furthermore, the server will run its own tests over your code, which means that you will not be able to submit code that does not work. Reminder: this means that most of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests.

The language and how to form tests: In this homework (and in all future homework) you should be working in the "Module" language and use the appropriate language using a `#lang` line. You should also click the "Show Details" button in the language selection dialog, and check the "Syntactic test suite coverage" option to see parts of your code that are not covered by tests: after you click "run", parts of the code that were covered will be colored in green, parts that were not covered will be colored in red, and if you have complete coverage, then the colors will stay the same. Note that you can also set the default language that is inserted into new programs to `#lang pl`, to make things more convenient. There are some variants for the `pl` language for various purposes — in particular, `#lang pl untyped` will ignore all type declarations, and will essentially run your code in an untyped Racket.
The language for this homework is:

```
#lang pl
```

As will also be shown in class, this language has a new special form for tests: `test`. It can be used to test that an expression is true, that an expression evaluates to some given value, or that an expression raises an error with some expected text message. For example, the three kinds of tests are used in this example:

```
(define (safe-length list)
  (cond [(null? list) 0]
        [(pair? list) (add1 (safe-length (cdr list)))]
        [else (error 'safe-length "bad value: ~s" list)]))
(test (zero? (safe-length null)))
(test (safe-length '(1 2 3)) => 3)
(test (safe-length "three") =    error> "bad value")
```

In case of an *expected* error, the string specifies a pattern to match against the error message. (Most text stands for itself, "`?`" matches a single character and "`*`" matches any sequence of characters.)
Note that the `=error>` facility checks only errors **that *your* code throws**, not

Racket errors. For example, the following test **will not** succeed (because it is an error thrown by Racket):

```
(test (/ 4 0) =error> "division by zero")
```

Reminder: code quality will be graded. Write clean and tidy code.

Questions

1. Reminder:

   Remember that lists are defined inductively as either:

   1.1. An empty list — `null`
   1.2. A `cons` pair (sometimes called a "cons cell") of *any* head value and a list as its tail — `(cons` x y`)`
   A "`Listof` T" would be similar, except that it will use the type T (which needs to be defined by you, say, Symbol, Natural, or Number) instead of "any".

   a. With this in mind, define a recursive function `open-list` that consumes a list of lists (where the type of the elements in the inner lists in a Number) and returns a list contains all the elements of the inner lists concatenated in the same order.

      For example, written in a form of a test that you can use:

      ```
      ((test (open-list '((1 2 3) (2 3 3 4) (9 2 -1) (233 11 90))) => '(1 2 3 2 3 3 4 9 2 -1 233 11 90))
      ```

   b. Define a function `min&max` that consumes a list of lists (where the type of the elements in the inner lists in a Number) and returns a list containing the minimum and the maximum of the values in the inner lists.
      - Here you should use the Racket built-in min/max functions that consumes numbers as parameters and returns the maximum/minimum value between them.
        - Note that the built-in min/max functions returns the minimum/maximum value in the greater family between the given numbers, following are some examples:
          - (max 9 3.3) returns 9.0 (and not 9) since Number is contains Reals (it contains both floating numbers and reals).

- (min 0.75 1/2) returns 0.5 since Numbers contains both floating numbers and exact ratios.
- (max 1 1/2) returns 1 but as an Exact rational.

- Hint: you may use the default values of the minimum/maximum values as +inf.0 / -inf.0 which refers to $\pm\infty$.

For example, written in a form of a test that you can use:

```
(test (min&max '((1 2 3) (2 3 3 4) (9 2 -1) (233 11
90))) => '(-1.0 233.0))
```

c. To solve the "problem" in part "b" you may want to use the built in Racket apply, and min/max.
Write a function `min&max_apply` that does exactly what you did in part "b" but using apply function.
For example, written in a form of a test that you can use:

```
(test (min&max_apply '((1 2 3) (2 3 3 4) (9 2 -1) (233
11 90))) => '(-1 233))
```

2. In this question we will implement a simple Table data structure. In this data structure you will need to define a new type called `Table`. Each element in the table will be keyed (indexed) with a symbol. In the following the operations that you are required to implement are detailed below, together with some guidance.
   2.1. Implement the empty table `EmptyTbl` – this should be a variant of the data type (constructor).
   2.2. Implement the add operation `Add` – this too should be a variant of the data type. The add operation should take as input a symbol (key), a string (value), and an existing table and return an extended table in the natural way – see examples below.
   2.3. Implement the search operation `search-table` – the search operation should take as input a symbol (key) and a table and return the first (LIFO, last in first out) value that is keyed accordingly – see examples below. If the key does not appear in the original table, it should return a `#f` value (make sure the returned type of the function supports this; use the strictest type possible for the returned type).
   2.4. Implement the remove item operation `remove-item` – the remove item operation should take as input a table and a symbol (key) and return a new table contains the items of the original table except of the item to be deleted without the (first (LIFO) keyed value) – see examples below. If the original table was empty, it should return an empty table value.

For example, written in a form of a test that you can use:

```
(test (EmptyTbl) => (EmptyTbl))

(test (Add 'b "B" (Add 'a "A" (EmptyTbl))) =>

      (Add 'b "B" (Add 'a "A" (EmptyTbl))))

(test (Add 'a "aa" (Add 'b "B" (Add 'a "A" (EmptyTbl)))) =>

      (Add 'a "aa" (Add 'b "B" (Add 'a "A" (EmptyTbl)))))

(test (search-table 'c (Add 'a "AAA" (Add 'b "B" (Add 'a "A"
(EmptyTbl)))))

=> #f)

(test (search-table 'a (Add 'a "AAA" (Add 'b "B" (Add 'a "A"
(EmptyTbl)))))

=> "AAA")

(test (remove-item (Add 'a "AAA" (Add 'b "B" (Add 'a "A"
(EmptyTbl)))) 'a)

=> (Add 'b "B" (Add 'a "A" (EmptyTbl))))

(test (remove-item (Add 'a "AAA" (Add 'b "B" (Add 'a "A"
(EmptyTbl)))) 'b)

=> (Add 'a "AAA" (Add 'a "A" (EmptyTbl))))
```

Enjoy,

Good Luck!