Homework #1: Intro

Out: Thursday, March 18, Due: April 05 , 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.
General note: do not duplicate code! If there is an expression that is used in multiple places, then you should use `let`.

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. Consult the Style Guide, and if something is unclear, ask questions on the course's forum.

Questions

1. Define the following functions, all of which consume five characters and return a string.

   1.1. The function *append5* – consumes 5 characters and returns the concatenation of them. For example, `(append5 #\a #\b #\c #\d #\e)` would return `"abcde"`. Here is another example, written in a form of a test that you can use:

   ```
   (test (append5 #\e #\d #\c #\b #\a) => "edcba")
   ```

   1.2. The function *permute3* – consumes 3 characters and returns a list of strings the concatenation of them in any possible ordering (you may assume all of them are distinct). **Important:** make sure to declare the most precise type for the returned value of your function. For example, if you know you are using a list of two naturals, then, `(List Natural Natural)` is more precise than `(Listof Any)`.

   Here is an example, written in a form of a test that you can use.

   ```
   (test (permute3 #\a #\b #\c) =>
             '("abc" "acb" "bac" "bca" "cab" "cba"))
   ```

2. Reminder:

   Remember that lists are defined inductively as either:

   2.1. An empty list — `null`
   2.2. 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 `count-3lists` that consumes a list of lists (where the type of the elements in the inner lists may be any type) and returns the number of inner lists (within the wrapping list) that contain exactly 3 elements.

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

```
(test (count-3lists '((1 3 4) () (1 2 3)) ("tt"
"Three" 7) (2 4 6 8) (1 2 3))) => 3)
```

b. Define a function `count-3lists-tail` that works the same as `count-3lists`, but now you need to use tail-recursion.

   You can use the same example (only here the name of the function is `count-3lists-tail`):

```
(test (count-3lists-tail '((1 3 4) () (1 2 3)) ("tt"
"Three" 7) (2 4 6 8) (1 2 3))) => 3)
```

c. Write an additional function `count-3listsRec` that is similar to the above `count-3lists`, however counts the number of lists of length 3 recursively (i.e., on all levels of nesting).

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

```
(test (count-3listsRec '((1 3 4) () (1 2 3)) ("tt"
"Three" 7) (2 4 6 8) (1 2 3))) => 4)
```

3. In this question we will implement a keyed-stack data structure. In this data structure you will need to define a new type called `KeyStack`. Each element in the stack 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.

3.1. Implement the empty stack `EmptyKS` – this should be a variant of the data type (constructor).
3.2. Implement the push operation `Push` – this too should be a variant of the data type. The push operation should take as input a symbol (key), a string (value), and an existing keyed-stack and return an extended key-stack in the natural way – see examples below.
3.3. Implement the search operation `search-stack` – the search operation should take as input a symbol (key) and a keyed-stack 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 stack, 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).
3.4. Implement the pop operation `pop-stack` – the pop operation should take as input a keyed-stack and return the keyed-stack without its first (keyed) value – see examples below. If the original stack was empty, 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).

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

```
(test (EmptyKS) => (EmptyKS))

(test (Push 'b "B" (Push 'a "A" (EmptyKS))) =>

      (Push 'b "B" (Push 'a "A" (EmptyKS))))

(test (Push 'a "AAA" (Push 'b "B" (Push 'a "A"  (EmptyKS))))
=> (Push 'a "AAA" (Push 'b "B" (Push 'a "A" (EmptyKS)))))

(test (search-stack 'a (Push 'a "AAA" (Push 'b "B" (Push 'a
"A" (EmptyKS))))) => "AAA")

(test (search-stack 'c (Push 'a "AAA" (Push 'b "B" (Push 'a
"A" (EmptyKS))))) => #f)

(test (pop-stack (Push 'a "AAA" (Push 'b "B" (Push 'a "A"
(EmptyKS))))) => (Push 'b "B" (Push 'a "A" (EmptyKS))))



(test (pop-stack (EmptyKS)) => #f)
```

4. In this question you are given full code together with tests for the presented functions. All you are required to do is to add the appropriate comments for each of the functions. These comments should describe what the function takes as input, what it outputs, what its purpose is, and how it operates. Do not forget to also add your personal remarks on the process in which you

personally came to realize the above. You should copy the following code into your .rkt file, and add the comment therein.

```
(: is-odd? : Natural -> Boolean)
;; << Add your comments here>>
;; << Add your comments here>>
(define (is-odd? x)
  (if (zero? x)
      false
      (is-even? (- x 1))))

(: is-even? : Natural -> Boolean)
;; << Add your comments here>>
;; << Add your comments here>>
(define (is-even? x)
  (if (zero? x)
      true
      (is-odd? (- x 1))))

;; tests --- is-odd?/is-even?
(test (not (is-odd? 12)))
(test (is-even? 12))
(test (not (is-odd? 0)))
(test (is-even? 0))
(test (is-odd? 1))
(test (not (is-even? 1)))

(: every? : (All (A) (A -> Boolean) (Listof A) -> Boolean))
;; See explanation about the All syntax at the end of the file…
;; << Add your comments here>>
;; << Add your comments here>>
 (define (every? pred lst)
  (or (null? lst)
      (and (pred (first lst))
           (every? pred (rest lst)))))


;; An example for the usefulness of this polymorphic function
(: all-even? :   (Listof Natural) -> Boolean)
;; << Add your comments here>>
;; << Add your comments here>>
 (define (all-even? lst)
  (every? is-even? lst))


;; tests
(test (all-even? null))
(test (all-even? (list 0)))
```

```
(test (all-even? (list 2 4 6 8)))
(test (not (all-even? (list 1 3 5 7))))
(test (not (all-even? (list 1))))
(test (not (all-even? (list 2 4 1 6))))


(: every2? : (All (A B) (A -> Boolean) (B -> Boolean) (Listof A) (Listof B) ->
Boolean))
;; << Add your comments here>>
;; << Add your comments here>>
 (define (every2? pred1 pred2  lst1 lst2)
  (or (null? lst1) ;; both lists assumed to be of same length
     (and (pred1 (first lst1))
          (pred2 (first lst2))
          (every2? pred1 pred2 (rest lst1) (rest lst2)))))
```

`(All (a ...) t)`
`(All (a ... a ooo) t)`
is a parameterization of type `t`, with type variables `v ...`. If `t` is a function type constructed with infix `->`, the outer pair of parentheses around the function type may be omitted.

Examples:

```
> (: list-length : (All (A) (Listof A) -> Natural))
> (define (list-length lst)
    (if (null? lst)
        0
        (add1 (list-length (cdr lst)))))
> (list-length (list 1 2 3))
- : Integer [more precisely: Nonnegative-Integer]
3
```

In the above example, all elements in the list argument must be of the same (polymorphic) type.