Assignment: 4

Due:	Tuesday, February 2, 2016 9:00pm
Language level:	Beginning Student
Allowed recursion:	(Pure) Structural recursion
Files to submit:	`int-list.rkt`, `bridge.rkt`, and `bridge-bonus.rkt`
Warmup exercises:	HtDP 8.7.2, 9.1.1 (but use box-and-pointer diagrams), 9.1.2, 9.5.3
Practise exercises:	HtDP 8.7.3, 9.5.4, 9.5.6, 9.5.7

- You should note a new heading at the top of the assignment: "Allowed recursion". In this case the heading restricts you to pure structural recursion, i.e., recursion that follows the data definition of the data it consumes. See slide 05-38.
- Provide your **own examples** that are distinct from the examples given in the question description of each function.
- Of the built-in list functions, you may only use *cons*, *first*, *rest*, *empty?*, *cons?*, *length*, and *member?*. You cannot use other functions such as *list*, including in examples and tests.
- Your solutions must be entirely your own work.
- For this and all subsequent assignments, you must include the design recipe for all functions, including helper functions, as discussed in class. In addition, you must include a data definition for all user-defined types. Unless otherwise stated, if *X* is a known type then you may assume that (*listof X*) is also a known type. You do not need to provide template functions unless explicitly stated.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
- You must use the **cond** special form, and are not allowed to use **if** in any of your solutions.
- It is very important that the function names and number of parameters match ours. You must use the basic tests to be sure. The names of the functions must be written exactly. The names of the parameters are up to you, but should be meaningful. The order and meaning of the parameters are carefully specified in each problem.

Here are the assignment questions you need to submit.

1. Perform the assignment 4 questions using the online evaluation "Stepping Problems" tool linked to the course web page and available at

   https://www.student.cs.uwaterloo.ca/~cs135/stepping.

   The instructions are the same as those in assignment 3; check there for more information if necessary. Reminder: You should not use DrRacket's Stepper to help you with this question. First, DrRacket's evaluation rules are slightly different from the ones presented in class, and you must use the ones presented in class. Second, when writing an exam you will not have the Stepper to help you. Third, you can re-enter steps as many times as necessary to get them correct, so you might as well maximize the educational benefit.

2. This problem involves lists of integers. It is not necessary to include a data definition or a template for this question. In the file `int-list.rkt`, write the following functions:

   (a) Write a function *elements-more-than* that consumes a (*listof Int*) and an *Int* in that order. The function will produce a list of integers from the original list that are strictly greater than the integer given. The order of the integers in the produced list should match the order of the integers in the list consumed.

   (*elements-more-than* (*cons* 1 (*cons* 2 (*cons* 3 *empty*))) 2) ⇒ (*cons* 3 *empty*)
   (*elements-more-than* (*cons* 5 (*cons* −1 (*cons* 3 *empty*))) 0) ⇒ (*cons* 5 (*cons* 3 *empty*))

   (b) An arithmetic sequence is a sequence of numbers where the difference between consecutive terms is a constant. For example:

   - 1, 2, 3, 4 is an arithmetic sequence since the difference is 1 each time
   - 5, 5, 5 is an arithmetic sequence since the difference is 0 each time
   - 4, 2, 0, -2, -4 is an arithmetic sequence since the difference is -2 each time

   Write a function *arithmetic-sequence?* that consumes a (*listof Int*) and produces *true* if the list is an arithmetic sequence and *false* otherwise. If the list consumed is *empty* or has only one or two elements, the function should produce *true*.

   (c) Write a function *digits->integer* that consumes a (*listof Int*) and produces the integer represented by those digits, where the least significant digit is at the beginning of the list and the most significant digit is at the end of the list. If any of the values in the list are not single digit, non-negative integers, then the function should produce 'error. If the list consumed is *empty*, then the function should produce 0. For example

   (*digits->integer* (*cons* 4 (*cons* 5 (*cons* 6 *empty*)))) ⇒ 654
   (*digits->integer* (*cons* 0 (*cons* 1 (*cons* 2 (*cons* 0 *empty*))))) ⇒ 210
   (*digits->integer* (*cons* −7 (*cons* −8 (*cons* −9 *empty*)))) ⇒ 'error

3. Bridge is a card game that is played using a standard deck of 52 playing cards. Each card has a value and a suit. The values are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. The suits are clubs, diamonds, hearts and spades. For this question you will be using lists to represent the cards held by a person who is playing Bridge.

   This question uses the following data definitions:

   ```
   ;; A BridgeHand is one of:
   ;; ⋆ empty
   ;; ⋆ (cons bhe BridgeHand), where bhe is a BridgeHandElement


   ;; A BridgeHandElement is one of:
   ;; ⋆ a Str
   ;; ⋆ a Sym, one of 'Ace, 'King, 'Queen, or 'Jack
   ;; ⋆ a Nat in the range 2 to 10 inclusive
   ```

   The *BridgeHand* and the *BridgeHandElement* follow these rules:

   - the *String* elements have the format "X Start" or "X End", where X is one of "Clubs", "Diamonds", "Hearts" or "Spades",
   - the *BridgeHandElement* values that appear between "X Start" and "X End" represent the cards from suit X. There will be at least one other *BridgeHandElement* between these strings.
   - "X Start" will always appear before a matching "X End" and there will be a maximum of one pair of these strings for each of the suits in a deck of cards.
   - there will not be an "Y Start" between "X Start" and "X End"
   - there will not be any values between "X End" and "Y Start"

   Note that these rules are included for completeness and to help you design test data.

   You have been provided with a starter file, `bridge.rkt`, that contains the data definitions for *BridgeHand* and *BridgeHandElement*. Using this file, complete the following:

   (a) Write a template based on the *BridgeHand* data definition. Your template should reflect the data definitions of both *BridgeHand* and *BridgeHandElement*. Note that the template does not need to reflect any of the extra information about a *BridgeHand* such as the range of the *Nat* values or the extra rules.

   (b) Write a function called *count-points* that consumes a *BridgeHand* and produces the number of points that the *BridgeHand* is worth. Points for a hand are based only on the total of the high cards in your hand. The points assigned to the high cards are as follows:

   - 4 points for an 'Ace
   - 3 points for an 'King
   - 2 points for an 'Queen
   - 1 point for an 'Jack

For example, if *count-points* consumes the *BridgeHand*

```
(cons "Clubs Start" (cons 8 (cons 'King (cons 'Ace (cons "Clubs End"
  (cons "Hearts Start" (cons 4 (cons "Hearts End"
    (cons "Diamonds Start" (cons 4 (cons 'Jack (cons "Diamonds End" empty))))))))))))
```

the function produces 8, because the 'King is worth 3 points, the 'Ace is worth 4 points and the 'Jack is worth 1 point.

This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the public test results after making a submission.

---

4. **5% Bonus**: In the card game Bridge, you calculate points for the high cards in your hand, but you also calculate distribution points. Distribution points are assigned as follows:

   - 3 points for a void suit, i.e. no cards in a suit
   - 2 points for a singleton, i.e. only one card in a suit
   - 1 point for a doubleton, i.e. only two cards in a suit

   So for example if your hand had 3 clubs, 2 diamonds, 8 hearts, and no spades, the distribution points would be 4 (a doubleton plus a void). Write a function called *count-distribution* that consumes a *BridgeHand* and produces the number of distribution points that the *BridgeHand* is worth. Note that a void in a suit X will be represented by the fact that there are no *String* values of the form "X Start" and "X End" in the list.

   A good solution should not have too much duplication in the subexpressions of your code or in any helper functions that you write.

   Place your solution to this question in a file called `bridge-bonus.rkt`.

This concludes the list of questions for which you need to submit solutions. Don't forget to check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Racket supports unbounded integers; if you wish to compute 2^{10000}, just type (*expt* 2 10000) into the REPL and see what happens. The standard integer data type in most other computer languages can only hold integers up to a certain fixed size. This is based on the fact that, at the hardware level, modern computers manipulate information in 32-bit or 64-bit chunks. If you want to do extended-precision arithmetic in these languages, you have to use a special data type for that purpose, which often involves installing an external library.

You might think that this is of use only to a handful of mathematicians, but in fact computation with large numbers is at the heart of modern cryptography (as you will learn if you take Math 135). Writing such code is also a useful exercise, so let's pretend that Racket cannot handle integers bigger than 100 or so, and use lists of small integers to represent larger integers. This is, after all, basically what we do when we compute by hand: the integer 65,536 is simply a list of five digits (with a comma added just for human readability; we'll ignore that in our representation).

For reasons which will become clear when you start writing functions, we will represent a number by a list of its digits starting from the one's position, or the rightmost digit, and proceeding left. So 65,536 will be represented by the list containing 6, 3, 5, 5, 6, in that order. The empty list will represent 0, and we will enforce the rule that the last item of a list must not be 0 (because we don't

generally put leading zeroes on our integers). (You might want to write out a data definition for an extended-precision integer, or EPI, at this point.)

With this representation, and the ability to write Racket functions which process lists, we can create functions that perform extended-precision arithmetic. For a warm-up, try the function *long-add-without-carry*, which consumes two EPIs and produces one EPI representing their sum, but without doing any carrying. The result of adding the lists representing 134 and 25 would be the list representing 159, but the result of the lists representing 134 and 97 would be the list 11, 12, 1, which is what you get when you add the lists 4, 3, 1 and 7, 9. That result is not very useful, which is why you should proceed to write *long-add*, which handles carries properly to get, in this example, the result 1, 3, 2 representing the integer 231. (You can use the warmup function or not, as you wish.)

Then write *long-mult*, which implements the multiplication algorithm that you learned in grade school. You can see that you can proceed as far as you wish. What about subtraction? You need to figure out a representation for negative numbers, and probably rewrite your earlier functions to deal with it. What about integer division, with separate quotient and remainder functions? What about rational numbers? You should probably stop before you start thinking about numbers like 3.141592653589…

Though the basic idea and motivation for this challenge goes back decades, we are indebted to Professor Philip Klein of Brown University for providing the structure here.