Assignment: 4+5

Due:	Tuesday, October 18, 2016 9:00pm
Language level:	Beginning Student
Allowed recursion:	(Pure) Structural recursion
Files to submit:	`lists.rkt`, `curling.rkt` `eights.rkt`, `pie.rkt`
Warmup exercises:	HtDP 8.7.2, 9.1.1 (but use box-and-pointer diagrams), 9.1.2, 9.5.3, 10.1.4, 10.1.5, 11.2.1, 11.2.2, 11.4.3, 11.5.1, 11.5.2, 11.5.3
Practise exercises:	HtDP 8.7.3, 9.5.4, 9.5.6, 9.5.7, 10.1.6, 10.1.8, 10.2.4, 10.2.6, 10.2.9, 11.4.5, 11.4.7

- Policies from Assignment 3 carry forward.
- 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. Submissions that do not follow this restriction will be heavily penalized.
- In addition, you must include a data definition for all user defined types. You do not need to include templates in your solutions unless specifically required by the question.
- Test data for all questions will always meet the stated assumptions for consumed values.
- You may use the **cond** special form. You are **not** allowed to use **if** in any of your solutions.
- **It is very important that the function names match ours.** You must use the basic tests to be sure. In most cases, solutions that do not pass the basic tests will not receive any correctness marks. 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.
- Any string or symbol constant values must exactly match the descriptions in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic tests results will catch many, but not necessarily all of these types of errors.
- Since each file you submit will contain more than one function, it is very important that the code runs. If your code does not run then none of the functions can be tested for correctness.
- Do not send any code files by email to your instructors or tutors. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
- You may use examples from the problem description in your own solutions.
- For all questions you may use the functions from any other part of the same question as helper functions.
- Of the built-in list functions, you may only use *cons*, *first*, *rest*, *empty?*, *cons?*, *length*, *string->list*, *list->string*, and *member?*. You may not use list abbreviations.

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. (a) Write a function *count-ints* which consumes a list of numbers, and produces the number of integers in that list. The empty list contains 0 integers (because it contains 0 numbers of any sort). For example (*count-ints* (*cons* 1 (*cons* 2.1 (*cons* 0 *empty*))) ⇒ 2. You may find the Racket function *integer?* useful for this question.

   (b) Write a function *string-replace* which consumes a string, a target character, and a replacement character. The function produces a new string, which is identical to the consumed string with all occurrences of the target character (if any) replaced with the replacement character. For example, (*string-replace* "word" #\o #\y) ⇒ "wyrd". Note: you may not use any built-in string functions other than *string->list* and *list->string*.

   (c) Write a function *perfect-squares?* which consumes a list of integers, and produces *true* if all of the numbers are *perfect squares*, and *false* if one or more are not. A perfect square is the square of a rational number. Since this question deals only with integers, a perfect square is an integer where the square root is also an integer. E.g. 4 is a perfect square, as its square root is 2. 5 and -4 are not, as $\sqrt{5}$ is irrational, and $\sqrt{-4}$ is either not allowed or imaginary (depending on your math background).

   The empty list contains no numbers, so (*perfect-squares? empty*) ⇒ *true*.

   (d) Write a function *keep-divisible* which consumes a list of integers, and a non-zero integer *n*, and produces a new list that contains the numbers from the consumed list that are divisible by *n* (meaning the remainder when dividing by *n* is zero). The elements of the produced list must be in the same order as they were in the consumed list. For example, (*keep-divisible* (*cons* 1 (*cons* 6 (*cons* 2 (*cons* 3 *empty*)))) 3) ⇒ (*cons* 6 (*cons* 3 *empty*))

   Submit your code for this question in a file named `lists.rkt`

3. Curling is a sport where two teams slide stones toward a target (the "house") with the objective of having the closest stone to the center of the target (the "button"). After each team has thrown eight stones, the winning team is awarded one point for each stone they have within the house that is closer than **all** of the opposing team's stones. A stone is only counted if it is within the "house". Stones that end up out of bounds are immediately removed from play.

In this question you will be scoring by slightly different rules than regulation Curling, so you must not look online for the rules. In our version, the house has a radius of exactly 183cm, and all distances are measured from the center of the stone. A stone is considered "in the house" if it is strictly less than 183cm from the center. If both teams' closest stone is the same distance from the button (e.g. both teams have a stone 10cm away) then each team has 0 stones closer than the other team, so neither team is awarded points.

(a) Write the function *count-less-than* that consumes a list of numbers and another number, and produces the number of elements in the list that are strictly less than the second parameter.

For example: (*count-less-than* (*cons* 5 (*cons* 2 (*cons* 3 (*cons* 1 (*cons* 4 *empty*))))) 4) ⇒ 3 (because 1, 2, and 3 are less than 4, while 4 and 5 are not).

(b) Write the function *end-points* that consumes two **sorted** lists of numbers, where each list represents the distances for all of each team's stones to the "button" (in cm). If the first team wins, the function produces the number of points they won (e.g. 2 if team 1 wins 2 points), if the second team wins, the function produces the number of points they won, as a negative number (e.g. -3 if team 2 wins 3 points) and in the event of a tie, the function produces 0.

Examples:

(*end-points* (*cons* 15 (*cons* 25 (*cons* 35 *empty*))) (*cons* 35 (*cons* 40 (*cons* 50 (*cons* 60 *empty*))))) ⇒ 2 (since 15 and 25 are less than 35)
(*end-points* (*cons* 188 (*cons* 190 (*cons* 200 *empty*))) (*cons* 180 (*cons* 185 *empty*))) ⇒ −1 (since 180 and 185 are less than 188, but 185 is not within 183 so it does not count)

Note: Although each team throws eight stones, stones that end up out of bounds are removed from play, so the lists may not always contain 8 values (and one or both could be empty if all stones ended up out of bounds).

Submit your code for this question in a file named `curling.rkt`

4. **This is the start of the Assignment 5 material**

In the card game "crazy eights" players take turns playing a card in the center, with the goal of being the first player without any cards in their hand. Players must play a card that matches the suit of the center card, with two exceptions. First, a player may instead play a card that matches the value of the center card. Second, eights are "wild", so a player may always play an eight. When a player plays an eight, they say the name of a suit. The eight is treated as having this suit, even if it does not. For example, if a player plays the eight of spades, and says "hearts", then the next card played must be a "hearts" card (or another eight).

Instead of playing a card, the player may draw a card. If they can play the new card they may do so, otherwise this is the end of their turn.

When a player runs out of cards, they have won the hand. The winning player receives points based on what cards the other players are holding.

For this question we will represent cards using the `Card` type:

```
(define-struct card (suit value))
;; A Card is a (make-card Sym Nat)
;; requires: suit is one of 'hearts, 'diamonds, 'clubs, or 'spades
;;           value between 1 and 13, using 11 for Jack, 12 for Queen, and 13 for King.
```

Note: We are representing these cards with a computer, so it is easy enough to actually change the eight of spades into an eight of hearts when it is played! For that reason, in the following functions you may assume that the current suit is equal to the suit of the current card in the center, even if the center card is an eight.

(a) Write a function *card=?* which consumes two `Card` structures and produces *true* if they represent the same playing card, and *false* otherwise. You must not use *equal?* in your solution. For example, (*card=?* (*make-card* 'hearts 3) (*make-card* 'hearts 3)) produces *true*.

(b) Write a function *crazy-count* which consumes a list of `Card` structures and the current center `Card`, and produces the number of `Card` structures in the list that can legally be played in the center.

(c) Write a function *crazy-dumb* which consumes a list of `Card` structures and the current center `Card`, and produces a `Card` that can be played in the center. In the event that multiple `Card` structures can be played, it produces the first playable `Card` (this is often a dumb move). If there is no `Card` that can be played, the function produces *false* to indicate that the player cannot play a card.

(d) Write a function *crazy-smart* which behaves as above, but only plays an 8 if there is no other choice. That is, it produces the first `Card` it can play that is not an eight. If there is no such `Card`, then it produces the first eight. If there are no eights either, it produces *false*.

(e) Write a function *crazy-points* which consumes a list of `Card` structures and produces the number of points it would be worth if another player had those cards at the end of the game. Each card is worth points equal to its value, with the face cards (Jack, Queen, and King) worth 10. As an exception to this rule, eights are worth 50 points instead of 8.

Submit your code for this question in a file named `eights.rkt`

5. You are writing software for a new start-up bakery, πr^2 (pronounced "pie are squared"). This innovative bakery is disrupting the pie industry by making square pies (hence the name).

   Pies are priced as follows:

   - 'small is $6
   - 'medium is $9
   - 'large is $15

   A customer also selects one or more fillings. There is no extra charge for additional fillings (since you still get the same amount of pie) but there are certain "deluxe" fillings that cost extra. Deluxe filling adds $2 to the price of small and medium pies, and $5 to the price of large pies. This price is applied even if there are also regular fillings in the pie, and is only applied once even if there are several deluxe fillings.

   The deluxe fillings are 'blackberry, 'raspberry, and 'durian. Any other symbol is considered a regular filling.

   A Pie will be represented as a structure with two fields. The size of the pie (a symbol), and a list of one or more fillings (symbols). Duplicate fillings are allowed, and indicate that the customer wants extra of this filling (again, this has no effect on the price).

   Example: A medium pie that is 2/3 strawberry, 1/3 rhubarb would be represented as
   (**define** *sample-pie* (*make-pie* 'medium (*cons* 'strawberry (*cons* 'strawberry (*cons* 'rhubarb *empty*)))))

   (a) Below is a structure definition for *pie*. Complete the data definition for *Pie* and write the template
   (*define-struct pie* (*size fillings*))
   ;; A Pie is ...

   (b) Write a function *pie-cost* that consumes a *Pie* (yum) and produces the cost of that pie (see pricing information above).

   (c) Write a function *order-cost* that consumes a (*listof Pie*) and produces the total cost of the list of pies.

   (d) A customer might have certain food allergies. Write a predicate *ok-to-order?* that consumes a (*listof Pie*) and a filling, and produces *true* if **none** of the pies in the list used that topping, and *false* otherwise. This lets you know if a problematic pie has been prepared since the last thorough cleaning.

   Note: You must not use (*make-pie* ...) in any of your function definitions, only in your tests and examples! See the "Allowed Recursion" restriction in the instructions for why.

   Submit your code for this question in a file named `pie.rkt`

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.