CS 135
Winter 2014
Troy Vasiga

Assignment: 4

Due:	Tuesday, February 11, 2014 9:00pm
Language level:	Beginning Student
Allowed recursion:	Structural recursion
Files to submit:	`int-list.rkt`, `line-list.rkt`, and `area.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

The overall instructions for this assignment are the same as previous assignments except you will note a new addition at the top: "Allowed recursion" restricts you to structural recursion – recursion that follows the data definition of the data it consumes. Provide your own examples that are distinct from the given examples in the design recipe of each function.

For this and future assignments, it is a good idea to create some constants (i.e., lists) for testing and examples across multiple functions. This will reduce the amount of typing you need to do and reading that you and others need to do.

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 Assignment 3; check there for more information, if necessary. Reminder: You should not use DrRacket's Stepper to help you with this question, for a few reasons. First, as mentioned in class, DrRacket's evaluation rules are slightly different from the ones presented in class; you are to use the ones presented in class. Second, in an exam situation, of course, 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. Of the built-in list functions, you may only use *cons*, *first*, *rest*, *empty?*, and *cons?*.

   In the file `int-list.rkt`, write the following functions:

   (a) Write a function *sum-fav* that consumes a (*listof Int*) and an *Int favourite* and produces the sum of all elements in the list which are greater than or equal to *favourite*.
   (*sum-fav* (*cons* 4 (*cons* −5 (*cons* 6 *empty*))) 4) ⇒ 10
   (*sum-fav* (*cons* −7 (*cons* 8 (*cons* −9 *empty*))) 9) ⇒ 0

(b) Write a function *reciprocate* that consumes a (*listof Int*) and produces a list containing the reciprocals of each integer in the list (in the same order as numbers appeared in the consumed list). The reciprocal is defined for non-zero integers x as $\frac{1}{x}$. If the reciprocal is undefined, instead of placing a number in the produced list, place 'undefined.

(*reciprocate empty*) ⇒ *empty*
(*reciprocate* (*cons* 0 (*cons* 1 (*cons* 2 *empty*))))
⇒ (*cons* 'undefined (*cons* 1 (*cons* 0.5 *empty*)))

(c) Write a predicate *ascending-or-descending?* that consumes a (*listof Int*) that has all distinct elements and a symbol that is either 'ascending or 'descending. As *ascending-or-descending?* is a predicate, it produces a Boolean. If the list is empty or has one element, the function produces *true*. If the function consumes 'ascending and the list is sorted in ascending order, the function produces *true*. If the function consumes 'descending and the list is sorted in descending order, the function produces *true*. The function produces *false* otherwise. Note that a list of integer is ascending if for every pair of adjacent elements in the list, the second is larger than the first; in case of descending list, the second is smaller than the first.

(*ascending-or-descending?* (*cons* 1 (*cons* 2 (*cons* 3 *empty*))) 'ascending) ⇒ *true*
(*ascending-or-descending?* (*cons* −7 (*cons* −8 (*cons* −9 *empty*))) 'descending)
⇒ *true*

3. This problem involves lists of lines. Of the built-in list functions, you may only use *cons*, *first*, *rest*, *empty?*, and *cons?*. Use the same definition of a line as in Assignment 3, which was:

> A *Line* should have two fields, *slope* and *intercept* (in that order), where *slope* is the slope of the line (or 'undefined if the slope is not defined), and *intercept* is the y-intercept (unless the slope is undefined, in which case *intercept* is the x-intercept).

To aid with the following parts, consider the following definitions:
(**define** *line1* (*make-line* 1 0))
(**define** *line2* (*make-line* 0 3))
(**define** *line3* (*make-line* −1/2 −2))
(**define** *line4* (*make-line* 'undefined −4))
(**define** *line-list* (*cons line1* (*cons line2* (*cons line3* (*cons line4 empty*)))))

In the file `line-list.rkt`, write the following:

(a) Place the given data definition and structure definition of a *Line* in your file. Use those two definitions to write the template for a function which consumes a (*listof Line*).

(b) Write the function *negate-slope* which consumes a (*listof Line*) and produces the same list, with all the slopes negated (unless the slope was 'undefined, in which case, it remains 'undefined). For example:

(*negate-slope line-list*)
⇒ (*cons* (*make-line* −1 0) (*cons* (*make-line* 0 3) (*cons* (*make-line* 0.5 −2) (*cons* (*make-line* 'undefined −4) *empty*))))

(c) Write the function *positive-line* which consumes a (*listof Line*) and produces the list of those lines which have either a positive slope or a positive (x or y) intercept. For example:

(*positive-line line-list*) ⇒ (*cons* (*make-line* 1 0) (*cons* (*make-line* 0 3) *empty*))

(d) Write the function *through-point* which consumes a (*listof Line*) and a point (represented as a *Posn*) and produces a list of those lines which pass through the point. Note that you may assume that all slopes, intercepts and point coordinates are exact values (or, in the case of the slope values, 'undefined is also valid). For example:

(*through-point line-list* (*make-posn* −4 3))
⇒ (*cons* (*make-line* 0 3) (*cons* (*make-line* 'undefined −4) *empty*))

(e) You want to know which consecutive pair of lines in a list of lines will intersect. Write a function *parallel-non-intersect-lines* which consumes a (*listof Line*) which has at least two elements. You should produce a list of Boolean values, with *true* indicating that a pair of lines are parallel and non-intersecting. You should produce one Boolean value for each consecutive pair of lines. For example:

(*parallel-non-intersect-lines* (*cons line1* (*cons line1* (*cons* (*make-line* 1 4) (*cons line2 empty*)))))
⇒ (*cons false* (*cons true* (*cons false empty*)))

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**:

   Lists can contain structures, like *Posn*s, as well as numbers and other atomic types.

   One useful thing one might want to do with a list of *Posn*s is to treat the *Posn*s as successive coordinates of the vertices of a polygon.

   Write a function *area-of-polygon* which consumes a non-empty list of *Posn*s, and produces the area of the polygon with those *Posn*s as successive vertices. You may assume the given vertices produce a polygon which is non-self-intersecting.

   `http://mathworld.wolfram.com/PolygonArea.html` has information about how to calculate the area of a polygon, given the coordinates of its vertices. Note that in some situations your code should produce a negative area. Place your solution to this question in a file called `area.rkt`.

Hint: you will probably need to write the helper function *copy-first-to-end*, which consumes a non-empty list, and produces the same list, with its first element copied to the end of the list. For example:

(*copy-first-to-end* (*cons* 3 (*cons* 6 (*cons* 2 *empty*))))
⇒ (*cons* 3 (*cons* 6 (*cons* 2 (*cons* 3 *empty*))))

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

Scheme 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 in Math 135). Writing such code is also a useful exercise, so let's pretend that Scheme 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 Scheme 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.