CS 135
Winter 2014
Troy Vasiga

Assignment: 3

Due:	Tuesday, January 28, 2014 9:00 pm
Language level:	Beginning Student
Files to submit:	`lines.rkt`, `cards.rkt`, `square.rkt`
Warmup exercises:	HtDP 8.2.1, 8.2.2, 8.6.2, 8.6.3, 6.1.1
Practise exercises:	HtDP 8.2.3, 8.2.4, 8.3.1, 8.3.2

Here are the assignment questions you need to submit.

1. In this question, you will perform step-by-step evaluations of Scheme programs, by applying substitution rules until you either arrive at a final value or cannot continue. You will use an online evaluation tool that we have created for this purpose.

   To begin, visit this web page:

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

   (Note that the use of `https` is significant; the system won't work if you omit the `s`. This link is also in the course web page's table of contents.) You will need to authenticate using your Quest/WatIAm ID and password. Once you're logged in, you should try the questions in the "Warm-up questions" category under "CS 135 Assignment 3", in order to get used to the system. Note the "Show instructions" link at the bottom of each problem. Read the instructions before attempting a question!

   When you're ready, complete the six stepping problems in the "Assignment questions" category, using the semantics given in class for Beginning Student Scheme. You can re-enter a step as many times as necessary until you get it right, so keep going until you completely finish every question. All you have to do is complete the questions online—we'll be recording your answers as you go, and there is no file to submit! The public tests for this assignment will tell you whether or not we have a record of your completion of the stepper problems. **You are not done a question until you see the message** `Question complete!` You should see this once you have arrived at a final value and clicked on "simplest form".

   You should **not** use DrRacket's Stepper to help you with this question, for a couple of 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. There will definitely be step-by-step evaluation questions on at least the first midterm.

2. Recall from high school geometry that two points, (x_1, y_1) and (x_2, y_2), define a line in a plane. One standard equation for a line is $y = mx + b$ where m is the slope, defined by

$$m = (y_1 - y_2)/(x_1 - x_2)$$

and b is the y-intercept (i.e., the point where the line crosses the y-axis).

This form of the equation of a line does not work well for vertical lines, however (since the slope is undefined, and there is no unique y-intercept). If the line is vertical, there is a unique x-intercept which does define the vertical line.

Place your solutions to the following questions in `lines.rkt`.

(a) Write a structure definition and an accompanying data definition for a *Line*. A *Line* should have two fields, *slope* and *intercept* (in that order). Note the capitalization rules in Module 04 concerning how to define a *Line*.

(b) Write a function *two-points→Line* which consumes two distinct points (as *Posn* structures) and produces the corresponding *Line* which goes through these two points. Specifically, you should produce a *Line* structure with the slope set to the computed slope between the two points, and the intercept to be the y-intercept. Should your *Line* be vertical, you should set the slope of the line to be 'undefined and set the intercept to be the intercept on the x-axis.

(c) Write a function *perp-Line* that consumes a *Posn* representing a point and a *Line* in that order. Your function should produce a *Line* which goes through the given point and is perpendicular to the given *Line*. Again, your consumed or produced *Line* may be vertical, and should be treated the same way as in part (b).

As a reminder, a perpendicular line has slope which is the negative-reciprocal to the original line (i.e., the negative-reciprocal of 2 is $-\frac{1}{2}$). The remaining part is to figure out the new intercept, which involves computing the value b if the produced line is not vertical, or determing the x-intercept if the line is vertical.

3. A *Card* is defined by its *suit* and its *rank*.

The *suit* will be one of 'clubs, 'diamonds, 'hearts, or 'spades. The *rank* is an integer in the range 1 to 13, inclusive.

Place your solution to the following questions in the file `cards.rkt`. Note that your symbols must match the expected symbols *exactly*: a sample file with the symbol names has been made available to ensure you do not make any typographical errors in the symbol names used in this question.

(a) Write a structure definition and an accompanying data definition for a *Card*. Use the field names as described above (with *suit* being the first field). As in the previous question, use the capitalization rules outlined in Module 04.

(b) Write a function *better-Card* which consumes two cards, and produces the *Card* which is the better of the two.

The better card is either the card with the better *suit* (the suits are increasing from 'clubs (worst), 'diamonds (second-worst), 'hearts (second-best), or 'spades (best)) or, if the cards have the same *suit*, then the best card is the card with the largest/highest *rank*. If the two consumed *Card*s are the same (i.e., same *suit* and *rank*), return either one.

(c) Write the function *hand-value* which consumes three distinct *Card*s, and produces a symbol indicating the best hand-value of the given three *Card*s (where the arrangement/order of the three cards can be altered). The hand-values are described in decreasing order (from best to worst) below:

- 'straight-flush: all three cards are the same suit and their ranks are three consecutive integers;
- 'flush: all three cards are the same suit;
- 'straight: the ranks of the three cards are three consecutive integers;
- 'three-of-a-kind: the ranks of the three cards are the same;
- 'pair: the ranks of two of the cards are the same
- 'high-card: none of the previous outcomes are satisfied.

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 file *square.rkt*, write the function *four-square?*, which consumes four distinct *Line*s and produces *true* if there are exactly four distinct points of intersection between the four *Lines*, and these four points of intersection are the corners of a square, and *false* otherwise. (You can assume that all slopes and intercepts are rational numbers, with the exception that vertical lines have an 'undefined slope.) (You should also add the definition of *Line* to your file for testing (i.e., use the same definition of a *Line* as in Question 2)).

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

The textbook uses graphical examples early through the use of the `draw.ss` teachpack. We have avoided this material in early lectures, because it is not purely functional (the "state" of the canvas is changed by drawing functions through what are generally called "side effects") and it is hard to talk about examples and tests. We will treat similar material in the proper context (later in the course) and the next edition of the textbook will use more functional teachpacks, such as `image.ss` (for drawing images) and `world.ss`.

The `world.ss` teachpack extends the image manipulation functions of `image.ss` to allow simple animations. It provides functions to initialize a canvas and schedule repeated updates at the

"tick" of a clock. The updates occur through the use of an update function that you write which consumes the current "world" and produces the next world, one tick later. The description of the world can be as simple as one number, but typically it involves a structure containing information about the world, and function applications to draw the picture corresponding to that information. You can read the documentation through DrRacket's Help Desk. We invite you to explore the world of functional animation. The teachpack also provides a function that allows you to program responses to graphical updates through keyboard events. Consider what you would need on top of this to implement the basics of some of your favourite games or interactive applications. What sort of computations do you need to describe, and what Scheme vocabulary are you lacking in order to express those computations?

Notice that in using `world.ss`, you are providing a function as an argument in order to get the updates done. Strictly speaking, this violates the rules of Beginning Student, but this is a hint as to the power of Scheme that will be revealed in upcoming weeks.

The authors of *How to Design Programs* have a follow-on book called *How to Design Worlds* which explores the `world.ss` teachpack in depth. If you are interested, you can find it at `http://world.cs.brown.edu/`.