Winter 2024
Victoria Sakhnini

Assignment: 03

Due:	Tuesday, February 6, 2024 9:00 pm
Coverage:	Slide 17 of Module 05
Language level:	Beginning Student
Files to submit:	`examples-a03.rkt`, `waterloo2-poker.rkt`, `collide.rkt`, `bonus-a03.rkt`

- **Make sure you read the OFFICIAL A03 post on Piazza** for the answers to frequently asked questions.
- Unless otherwise specified, you may only use Racket language features we have covered up to the coverage point above (Slide 17 of Module 05). As well, you have not seen recursion up to this point in the course, and it is not allowed for this assignment.
- For each function you are required to write, **you are also required to submit the design recipe.**
- The names of functions we tell you to write, and symbols and strings we specify must match the descriptions in the assignment questions exactly. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic test results will catch many, but not necessarily all of these types of errors.
- Policies from Assignment A02 carry forward, including:
  - More details about the design recipe grading are specified in the A03 Official Post on Piazza.

Here are the assignment questions you need to solve and submit.

1. **(20%)**: In this question you will perform step-by-step evaluations of Racket programs, as you did in assignment one. Please review the instructions on stepping in A01.

   To begin, visit this web page:

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

   When you are ready, complete the three required questions under the "Module 5a: Lists" category, using the semantics given in class for Beginning Student.

2. **(40%)**: You may be familiar with the card game *poker*. A poker hand is typically made up of five playing cards, and the most popular variants of poker use seven cards (where only five of the seven cards are used to make a hand). Three-card poker is also a popular game in some casinos.

For this question, to keep things simple, we will be using a *two-card poker* variant known as *Waterloo*^{2} *Poker*.

Waterloo^2 *Poker* uses a standard (North American) 52-card deck. Each `Card` has a `Rank` and a `Suit`, and to represent them in Racket, we have provided some user-defined types:

```
;; A Rank is one of: 2, 3, 4, 5, 6, 7, 8, 9, 10, 'J, 'Q, 'K, 'A

;; A Suit is one of: 'C, 'D, 'H, 'S

;; A Card is a: (cons Rank (cons Suit empty))
```

We have used short symbols to make your testing less onerous. For example, `'Q` is used to represent a *Queen*, and `'C` is used to represent *Clubs* so the following card: `(cons 'Q (cons 'C empty))` represents the playing card Queen of Clubs.

The 52-card deck of `Card`s is composed of every possible combination of a `Rank` and `Suit` ($13 \times 4 = 52$).

For now, we have defined a `Card` as a `list` with exactly two items. In Module 10 we will introduce structures, which can also be used to store multiple items. As we will see in that module, one advantage of using a structure is that you can give each field of the structure a meaningful name. However, we can achieve a similar effect with just lists by using a helper function to make your code easier to read:

```
;; suit: Card -> Suit
(define (suit c)
  (first (rest c)))
```

The `Rank` of a `Card` can be either a number or a symbol. Fortunately, each `Rank` also has a corresponding `ordinality`, which will make comparing `Rank`s easier. For number ranks, the `ordinality` is the same as the number. The symbols `{'J, 'Q, 'K, 'A}` have the corresponding ordinalities of `{11, 12, 13, 14}`, respectively. Note that in some variants of poker, the `'A` (Ace) can have an `ordinality` of either 1 or 14, but in *Waterloo*^{2} *Poker* the `'A` (Ace) always has an `ordinality` of 14 and there is no `Rank` with an `ordinality` of 1. The `ordinality` of a `Card` is simply the `ordinality` of the `Card`'s `Rank`.

A *Waterloo*^{2} *Poker* `Hand` is a list with two unique `Card`s:

```
;; A Hand is a: (cons Card (cons Card empty))
;; requires: the two cards are not identical
```

Important! A `Hand` is not a list of four items. It is a list of two `Card`s, and each `Card` is itself a list of length two.

This question has been designed to help you become comfortable with working with lists.

Consider the following example:

```
(define ace-of-spades (cons 'A (cons 'S empty)))
(define ace-of-diamonds (cons 'A (cons 'D empty)))

(define good-hand (cons ace-of-spades (cons ace-of-diamonds empty)))
```

The value of `good-hand` is:

```
(cons (cons 'A (cons 'S empty)) (cons (cons 'A (cons 'D empty)) empty))
```

You should review this example carefully to make sure you understand a `Hand`.

In *Waterloo*^{2} *Poker*, we need to determine the `strength` of a `Hand`, which will require some new terminology. In *Waterloo*^{2} *Poker*:

- A *pair* is when both `Card`s have the same `Rank`.
  example: `good-hand`
- A *flush* is when both `Card`s have the same `Suit`.
  example:

  ```
  (define 3h (cons 3 (cons 'H empty)))
  (define Jh (cons 'J (cons 'H empty)))
  (cons 3h (cons Jh empty)) ; this is a flush
  ```

- A *straight* is when the `ordinality` of the `Card`s have a difference of exactly one.
  example:

  ```
  (define Js (cons 'J (cons 'S empty)))
  (define Qd (cons 'Q (cons 'D empty)))
  (cons Js (cons Qd empty)) ; this is a straight
  ```

- A *straight flush* is when the `Hand` is both a *straight* and a *flush*.
  example:

```
(define 10h (cons 10 (cons 'H empty)))
(cons Jh (cons 10h empty)) ; this is a straight flush
```

In *Waterloo*^{2} *Poker*, each `strength` is assigned a numerical value:

Description	Strength
straight flush	4
pair	3
straight	2
flush	1
none of the above	0

Note that the strengths in *Waterloo*^{2} *Poker* are different than traditional 5-card poker, because the probability distributions are different.

When comparing two hands, the best `Hand` is the one with the larger `strength`. However, if two `Hand`s have the same `strength`, then the ordinalities of the `Hand`s must be compared.

Because each `Hand` has two `Cards`, it has a high `ordinality` (corresponding to the `Card` with the largest `ordinality`) and a low `ordinality`. Note that for a *pair*, the high and low ordinalities have the same value.

If two `Hand`s have the same `strength`, then the best `Hand` is the `Hand` with the largest high `ordinality`. If they have the same `strength` and the same high `ordinality`, then the best `Hand` is the `Hand` with the largest low `ordinality`. Two `Hand`s are equivalent if and only if they have the same `strength`, same high `ordinality` and the same low `ordinality`.

As a reminder, you may use the function you define in part (a) as a helper function for the function you define in part (b), *etc*.

(a) Write a function `ordinality` which consumes a `Card` c and produces the ordinality of c.

(b) Write a function `strength` which consumes a `Hand` h and produces the strength of h as a number according to the table above (*i.e.,* one of: $0, 1, 2, 3, 4$).

(c) Write a predicate function `hand<?` which consumes two `Hand`s $h1$ and $h2$ and produces `true` if $h2$ is a better `Hand` than $h1$, and `false` otherwise.

(d) Write a function `winner` which consumes two `Hand`s $h1$ and $h2$ and produces `'hand1` if $h1$ is a better `Hand` than $h2$ or `'hand2` if $h2$ is a better `Hand` than $h1$, and `'tie` if both `Hand`s are equivalent.

Place your functions for this question in a file `waterloo2-poker.rkt`.

3. **(40%):** When playing a video game like Super Mario Bros. or Legend of Zelda, your character may run into objects or get shot by projectiles. These events are known as collisions. It is very expensive, computationally, to calculate exactly when and where two objects collide.

So, we cheat. We place game objects, called assets, into simple bounding shapes, such as a sphere, and compute whether the bounding shapes collide—a much easier/faster process.

In this question, a point will be represented by a three element list of numbers as `(cons x (cons y (cons z empty)))`. A sphere will be represented by a two element list, where the first element is a point representing the center of the sphere —represented by a three element list, and the second element is the sphere radius which must be greater than 0.

```
;; A Point is a (cons Num (cons Num (cons Num empty)))

;; A Sphere is a (cons Point (cons Num empty))
;; requires: the second element in the list (representing the radius)
   must be > 0
```

For example,

```
(define p1  (cons 0 (cons 1 (cons 2 empty))))
(define s1 (cons p1 (cons 5 empty)))
```

s1 represents a sphere with radius 5 centered at $(0,1,2)$.

(a) Write a function called `distance-between-points` that consumes two points and produces the distance between them.

The distance between (x_1,y_1,z_1) and (x_2,y_2,z_2) is given by:
$d = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}$.

(b) Write a function called `point-in-sphere?` that consumes a point and a sphere and produces true if the point is inside the sphere and false otherwise. Recall that the equation of a sphere is: $(x-c_x)^2+(y-c_y)^2+(z-c_z)^2 = r^2$ where (c_x,c_y,c_z) is the position of the sphere's center, r is the sphere's radius, and (x,y,z) is the point.

Note: a point (x,y,z) is in the sphere if $(x-c_x)^2+(y-c_y)^2+(z-c_z)^2 \leq r^2$.

(c) Write a function called `collide?` that consumes two spheres and produces true if the spheres collide and false otherwise. Two spheres collide if there is a point inside the first sphere that is also inside the second sphere.

Place your functions for this question in a file `collide.rkt`.

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

Assignments will sometimes have additional questions that you may submit for bonus marks.

4. **5% Bonus:** A box is specified by a four element list as: `(cons xmin (cons xmax (cons ymin (cons ymax empty))))`. Write a function called `overlap-area` that consumes two

boxes and produces the size of the overlapping region. If the boxes do not overlap, then 0 should be produced. A negative value should never be produced.

Note: for all inputs you may assume that $xmin < xmax$ and $ymin < ymax$. Place your solution in `bonus-a03.rkt`.

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

The textbook upon which CS135 is based has graphical examples. We've tended to avoid them because they aren't purely functional and are hard to test. But they are excellent enhancement material!

Start by adding `(require 2htdp/image)` at the top of an empty Racket program.

You can make a solid blue circle with the function application `(circle 25 'solid 'blue)`. The arguments are the radius (in pixels), the "mode" (one of `'solid`, `'outline`, or an integer between 0 and 255 to indicate the transparency), and the colour (e.g. `'blue`, `'black`, and `'purple`; or see the documentation for a much longer list). There are similar functions available to create lines, ellipses, text, and polygons.

Multiple shapes can be combined with the `overlay` or `underlay` functions and their variants. For example, the following program will create the image on the right.

```
(require 2htdp/image)

(underlay
 (rectangle 80 80 'solid 'mediumseagreen)
 (polygon (list (make-posn 0 0)
                (make-posn 50 0)
                (make-posn 0 50)
                (make-posn 50 50))
          'outline
          (make-pen 'darkslategray 10 'solid 'round 'round)))
```

Image manipulation can be extended to simple animations by using `(require 2htdp/universe.rkt)`. 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, which you write and which consumes an integer representing the current tick and produces an image of the universe at that time.

You can read the related documentation through DrRacket's Help Desk. We invite you to explore the universe of functional animation.

A very simple example:

```
(require 2htdp/image)
(require 2htdp/universe)

(define max-size 100)

;; (update tick) updates the image for each tick of the animation's clock.
(define (update tick)
  (underlay
```

```
  (empty-scene (* 2 max-size) (* 2 max-size))
  (circle (modulo tick max-size) 'solid 'red)))

(animate update)
```

You might also look for techniques that allow 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 Racket vocabulary are you lacking in order to express those computations?

Notice that `animate` is consuming a function as an argument in order to get the updates done. This violates the rules of Beginning Student, but hints at the power of Racket that will be revealed in the upcoming weeks.

The `robot-gui` that is provided uses more advanced techniques than described above. Take a look, if you want, but be fore-warned that it's different.