CS 111 - Homework 5

Deadline

11:59 pm on **Friday, September 30, 2016**

How to submit

Each time you would like to submit your work:

- save your current Definitions window contents in a file with the name `111hw5.rkt`
- transfer/copy that file to a directory on `nrs-labs.humboldt.edu` (preferably in a folder/directory named `111hw5`)
- Now that your file is on nrs-labs.humboldt.edu, you need to log onto nrs-labs.humboldt.edu using `ssh`, so you can submit your file to me.
- WHILE you are logged onto nrs-labs:
  - IF you saved your file in a folder, use `cd` to change to the folder/directory where you saved it -- for example, if you saved it in the folder `111hw5`, then you would go to that directory by saying:

    ```
    cd 111hw5
    ```

  - use the `ls` command to make sure your `111hw5.rkt` file is really there:

    ```
    ls
    ```

  - type the command:

    ```
    ~st10/111submit
    ```

    ...and when asked, enter a homework number of `5`

    ...and when asked, enter `y`, you do want to submit all files with an appropriate suffix (I don't mind getting some extra files, as long as I also get `111hw5.rkt`; I'd rather receive too many files than too few due to typos.)

    ...make sure to carefully check the list of files submitted, and make SURE it lists `111hw5.rkt` as having been submitted! (The most common error is to try to run `~st10/111submit` while in a different directory than where your files are...)

Purpose

To practice thinking about and using lists, and to provide more practice using the design recipe to design and write functions, including functions involving `random` and lists.

Important notes

- Please note that only SOME, not all, of this homework's problems involve lists.
- NOTE: it is usually fine and often encouraged if you would like to write one or more **helper functions** to help you write a homework problem's required functions.
  - HOWEVER -- whenever you do so, EACH function you define SHOULD follow all of the design recipe steps: first write its signature, then its purpose statement, then its function header, then its tests/`check-` expressions, then replace its `...` body with an appropriate expression)

- Remember: Signatures and purpose statements are ONLY required for **function definitions** -- you do NOT write them for named constants or for non-function-definition compound expressions.
- You are expected to follow the Design Recipe for all **functions** that you design/define. So, each function is expected to include:
  - a signature comment, including a nicely-descriptive name of the function, the **types** of expressions it expects, and the **type** of expression it returns. This should be written as discussed in class (and you can find examples in the posted in-class examples). For example,

```
; signature: rect-area: number number -> number
```

  - a purpose statement comment, **describing** what the function expects and **describing** what it returns. For example,

```
; purpose: expects the length and width of a rectangle,
;          and returns the area of that rectangle
```

  - [following the design recipe, you will be writing the function header next; note that you **don't** need to write it twice. Follow the function header with a body of `...` at this stage, and replace that `...` with its body **later**, at the appropriate step in the design recipe.]
  - `check-expect` (or `check-within`, or other `check-` operation) expressions expressing the specific examples that you write **BEFORE** writing your function body. (These may be **placed** before or after your actual function, but you are expected to **create** these **BEFORE** writing the function body. I'll have no way of knowing if you really write these in the correct order, but note that I won't answer questions about your function body without seeing your examples written as `check-` expressions first...) For example,

```
(check-expect (rect-area 3 4)
              12)
(check-expect (rect-area 10 5)
              50)
```

  - How many `check-` expressions should you have? Remember, the basic rules of thumb are:

    * you need a test/`check-` expression for each "case" or category of data that may occur, AS WELL AS one for each "boundary" if intervals are involved, and you can always add more if you'd like!

    * if there is **only one** category of data, you should have **at least two** tests/`check-` expressions, for more-robust error-checking of your function.

    * **IF you have a function involving `random` in such a way that one or more of the needed `check-` expressions does not seem possible**, you should handle this as we will demonstrate for function `draw-images-randomly` in Week 6 Lecture 1 -- for each such needed test, put a string DESCRIBING what a specific example call should do, and follow it by that example call (so its result will appear under this string in the Interactions window when this is run).

      For example:

```
"====================================="
"I expect to see a scene with a red circle, a green star,"
"   and a purple star in random locations"
"====================================="
```

```
(draw-images-randomly
    (cons (circle 30 "solid" "red")
       (cons (star 40 "solid" "green")
          (cons (star 50 "solid" "purple")
             empty))))
```

* OR, you can put the failing `check-` expression, especially if looking at the not-matching actual and expected values nevertheless lets you tell if, except for the random part, your function really did work;

– [and, of course, your function definition itself!]

– You may include as many additional calls or tests of your function as you would like after its definition.

- You should use **blank lines** to separate your answers for the different parts of the homework problems. If you would like to add comments noting which part each answer is for, that is fine, too!
- **Because the design recipe is so important**, you will receive **significant** credit for the signature, purpose, header, and tests/check-expects portions of your functions. Typically you'll get at least half-credit for a correct signature, purpose, header, and examples/check-expects, even if your function body is not correct (and, you'll typically **lose at least half-credit** if you omit these or do them poorly, even if your function body is correct).

Problem 1

Start up DrRacket, (if necessary) setting the language to How To Design Programs - **Beginning Student** level, and adding the HTDP/2e versions of the image and universe teachpacks by putting these lines at the beginning of your Definitions window:

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

Put a blank line, and then type in:

- a comment-line containing your name,
- followed by a comment-line containing `CS 111 - HW 5`,
- followed by a comment-line giving the date you last modified this homework,
- followed by a comment-line with no other text in it --- for example:

```
; type in YOUR name
; CS 111 - HW 5
; last modified: 2016-09-23
;
```

Below this, after a blank line, now type the comment lines:

```
;
; Problem 1
;
```

The main purpose of this problem is to provide you with more practice writing expressions involving lists. **You are NOT writing any new functions for this problem!**

1 part a

- **Paste or type in** the comment containing the data definition for a Racket list. (It is OK if you do not also copy over the data definition for Anything...)
- **Paste or type in** the comment containing the TEMPLATE for a function that "walks through" a list
  - (note that BOTH of the above are conveniently located in `111lab05-reminders.rkt`, available from the CS 111 public course web site under "In-class Examples"...)
- Decide on a theme/topic, and define a named constant, with an appropriate, descriptive name, whose value is a list of at least FIVE things related to that theme/topic.
  - For this part, list abbreviations will **not** be accepted.

1 part b

Now, **USING your named constant list**:

- Write an expression whose value is the **first** thing in your named constant list.
- Write an expression whose value is the list of **all BUT the first thing** in your named constant list.
- Write an expression whose value is **JUST** the **SECOND** thing in your named constant list.
- Write an expression whose value is the list of **all BUT the first THREE things** in your named constant list.

Problem 2

Next, in your definitions window, type the comment lines:

```
;
;  Problem 2
;
```

Not all functions involving lists are necessarily recursive -- (usually they are recursive if you need to "visit" each list element or "walk through" the whole list).

Consider: what if, as part of a problem, you find yourself frequently wanting to grab JUST the fourth element in a given list? You might decide to write a helper function `get-fourth` that expects a list, and returns the fourth element in that list.

What should happen if some misinformed user calls `get-fourth` with a list of fewer than 4 elements? It is decided that, for this function's particular purposes, that this function should simply return `#false` if called with a list containing fewer than 4 elements. (Hint: remember that BSL Racket has a function, `length`, that expects a list and returns the number of elements in that list.)

Using the design recipe, design and write this function `get-fourth`.

Problem 3

Next, in your definitions window, type the comment lines:

```
;
;  Problem 3
;
```

This function does NOT involve lists at all. It is here to "play" with `random`, and to provide a function we'll use on a later problem.

3 part a

We will try out Racket's `random` function in Week 6 Lecture 1 -- it expects one number argument, an integer, and returns a pseudo-random number (which does happen to be an integer) in [0, given integer).

One could use this to create random colors! `make-color` expects a red value, green value, and blue value, each in [0, 256):

```
(make-color (random 256) (random 256) (random 256))
```

Write **two** expressions of type `image` that use the above expression for their color argument -- you should (quite likely) see two different-colored images resulting in the Interactions window (and they'll (quite likely) be different each time you click run.)

3 part b

I decide that I would like a little function to simply return a random-colored star outline image of a specified size (the distance between its points given in pixels).

Using the design recipe, design and write a function `random-star` that expects a desired distance between its points given in pixels, and returns a star outline image of that size and of a random color.

Because this uses `random`, remember to write your tests/examples as described in the **Important Notes** section above (as we will for the non-empty test for the to-be-posted Week 6 Lecture 1 example `draw-images-randomly`).

Problem 4

Next, in your definitions window, type the comment lines:

```
;
;  Problem 4
;
```

4 part a

We want to work with some lists of strings. So, develop a data definition comment for a `StringList`, (in the style of `NumberList`), and then ALSO develop a comment containing a TEMPLATE for a function "walking through" a `StringList`.

- (note that the `NumberList` data definition, and the template for a function "walking through" a `NumberList`, are conveniently located in `111lab05-reminders.rkt`, available from the CS 111 public course web site under "In-class Examples"...)

4 part b

Following the design recipe, design and write a function `emphasize-list` that expects a list of strings, and it returns a new list of strings in which `!!` has been added to the end of every string in the given list of strings. That is, if this function has as its argument a list of strings containing `"Hey"`, `"Oh my!"`, and `"mooo"`, then it would return a new list of strings containing `"Hey!!"`, `"Oh my!!!"`, and `"mooo!!"`. (And, if called with the empty list as its argument, this function should simply return an empty list.)

Problem 5

NOTE: Problems 5 and 6 are involving star outlines. IF YOU WISH, you may instead SUBSTITUTE another non-circle shape's outline of your choice.

Next, in your definitions window, type the comment lines:

```
;
;  Problem 5
;
```

5 part a

We want to work with some lists of numbers. So, paste or type in the data definition comment for a `NumberList`, and then paste or type in the comment containing the TEMPLATE for a function that "walks through" a `NumberList`.

- (note that BOTH of the above are conveniently located in `111lab05-reminders.rkt`, available from the CS 111 public course web site under "In-class Examples"...)

5 part b

Consider a list of numbers, in which each number represents the size of a star (the distance between its points in pixels).

Using the design recipe, develop a function `many-stars`, which expects a list of numbers representing star sizes, and returns a scene containing star outlines with those sizes but of random colors, each placed in the center of the scene.

You are expected to appropriately use Problem 3 part b's `random-star` function in your `many-stars` function.

Because this uses `random`, remember to write your non-empty-list tests/examples as described in the **Important Notes** section above (as we will for the non-empty test for the to-be-posted Week 6 Lecture 1 example `draw-images-randomly`). (You can write a proper `check-expect` for your `empty` list example call!)

5 part c

Define a named constant `EX-STAR-SIZE-LIST` which is a list containing at least 6 different star sizes, each of which is considerably smaller than whatever `WIDTH` and `HEIGHT` constants you are using for this homework's scenes.

Also copy over the function `add1-list` from the Week 5 Lab posted examples (be sure to include its signature, purpose, and tests, too).

Finally, write an expression calling `add1-list` with `EX-STAR-SIZE-LIST` as an argument.

5 part d

Write a `big-bang` expression whose initial world value is your `EX-STAR-SIZE-LIST`, whose `on-tick` expression uses `add1-list` and whose `to-draw` expression uses your function `many-stars`.

What do you see when you run this? Describe what you see in a comment after your `big-bang` expression.

Problem 6

Next, in your definitions window, type the comment lines:

```
;
;  Problem 6
;
```

Consider: I'd like to be able to push the up-arrow key to add a new star size to the beginning of my world's list of star sizes, and I'd like to be able to push the down-arrow key to remove the first star sizes from my worlds' list of star sizes (if the list isn't already empty, of course).

OPTIONALLY, if you'd like any **other** keystrokes to do something to your list of star sizes, you may add code to do this, also.

6 part a

Using the design recipe, develop a keystroke handler `update-star-size-list` that expects a list of star sizes and a string representing a keystroke, and

- if that keystroke is `"up"` for the up-arrow, it returns a new list of star sizes with the value 20 added as the new **first** element to those star sizes already there, and
- if that keystroke is `"down"` for the down-arrow, **IF** the star sizes list is not empty, it returns a new list of star sizes with the first radius removed (otherwise it returns the `empty` list)
- (if you decided to add any additional keystroke actions, you'll implement those)
- (for any other keystroke, it returns the given star sizes list, unchanged).

HINT: a `cond` branch's result-expression CAN itself be another `cond` expression...! But if you don't care for that, you can always write a helper function to help avoid that...

6 part b

Write a `big-bang` expression whose initial world value is your `EX-STAR-SIZE-LIST`, whose `on-tick` expression uses `add1-list`, whose `to-draw` expression uses your function `many-stars`, and whose `on-key` expression uses your function `update-star-size-list`.

BONUS PROBLEM 1 (up to 10 points)

If you choose to try this bonus problem, in your definitions window, type the comment lines:

```
;
;  Bonus Problem 1
;
```

Consider the concept of a dot product of two vectors:

Vector-a = (a1 a2 ... aN)

Vector-b = (b1 b2 ... bN)

The dot-product of Vector-a and Vector-b is, then,

(a1 * b1) + (a2 * b2) + ... + (aN * bN)

And what, really, is a vector but a list of numbers?

Using the design recipe, design a function `dot-product` that expects two vectors of the same size, expressed as two lists of numbers of the same length, and returns the dot product of those two vectors.

NOTE #1: assume that the dot product of two empty vectors is `0`.

NOTE #2: for our purposes in this bonus problem, you may ASSUME the two argument lists indeed are the same length.

BONUS PROBLEM 2 (up to 20 points)

If you choose to try this bonus problem, in your definitions window, type the comment lines:

```
;
;  Bonus Problem 2
;
```

Bonus 2 part a

Decide on a list of some type that you'd like to use as a world. It can be as simple as a list of numbers (representing anything BUT star sizes -- measures for other non-circle shapes are fair game!), or a list of colors, or a list of images, or a list of strings, etc.

EITHER:

- IF you are not using a list of numbers or strings, write a data definition comment for a list containing just the type you have chosen, and then also a comment containing a TEMPLATE for a function doing something with a whole instance of such a list.
- IF you are using a list of numbers or strings, write a comment saying, in this world, what the numbers or strings in those lists will be representing.

THEN, define a named constant whose value is an example of a list of the type you have chosen.

Bonus 2 part b

What will happen to your world list as the clock ticks? Will nothing happen, or will something get bigger (like the star sizes in the list of star sizes in preceding problems), or will something get smaller, or will something be added to the list, or will something be removed from the list, or will something random happen? It is up to you, and you may choose.

Use the design recipe to design the function whose name you will use in `big-bang`'s `on-tick` expression to have the chosen effect on your world on each clock tick. (Remember, it must expect a list of whatever type you decided in Bonus 2 part a, and return a list of whatever type you decided in Bonus 2 part a.)

Bonus 2 part c

How should your world be shown in a scene at each clock tick? Anything but a list of images being put in random locations or a list of star sizes being used to draw concentric star outlines is fair game, although variations on either of those is acceptable. (A list of images could be placed in some way other than randomly -- they could be placed across the bottom of a scene, for example. Or, a list of star sizes could be used to create stars that aren't all in the center of a scene.) You could place different shapes based on numeric measures, or place shapes or text-images whose colors are those from a list of colors, or text-strings based on a list of strings, etc.

Use the design recipe to design the function whose name you will use in `big-bang`'s `to-draw` expression to draw a scene based on your world on each clock tick. (Remember, it must expect a list of whatever type you decided in Bonus 2 part a, and return a `scene`.)

Bonus 2 part d

What keystrokes may be used to modify your world? Decide on at least 3 keys that can be typed to modify your world, and any 3 keys are fair game (yes, even "up" and "down" along with at least one other key). And, for each of those keys, decide on how your world should be changed as a result. Will you add something to the beginning of the world-list when a certain key is typed? Will you remove the first element from the

world-list? Will you change each element in the world-list in some way? As long as your at-least-3 keys each do something at least a little different, and each do something noticeable to your world, that is acceptable. (Any other key should leave the world-list unchanged.)

Then, use the design recipe to design the function whose name you will use in `big-bang`'s `on-key` expression to change your world when one of your specified keys is typed. (Remember, it must expect a list of whatever type you decided in Bonus 2 part a and a string representing a keystroke, and it must return a list of whatever type your decided in Bonus 2 part a.)

Bonus 2 part e

Write a `big-bang` expression whose initial world value is your constant from Bonus 2 part a, whose `on-tick` expression uses your function from Bonus 2 part b, whose `to-draw` expression uses your function from Bonus 2 part c, and whose `on-key` expression uses your function from Bonus 2 part d.

(Do you want to include additional `big-bang` clauses? If so, feel free to do so!)