CS 111 - Homework 5

Deadline

11:59 pm on **Friday, October 3, 2025**

Purpose

To provide more practice thinking about and using lists in addition to more practice with using the design recipe to design and write functions.

How to submit

You complete **Problem 1 and Problem 2** on the course Canvas site (short-answer questions on recognizing kinds of data for which branching is useful, and on lists), so that you can see if you are on the right track.

Then, you will submit **Problems 3 onward**, in your file **`111hw5.rkt`**, on the course Canvas site.

Submit your **`111hw5.rkt`** file-in-progress with your work on Problems 3 onward early and often!

Important notes - 6 points

- Please note that only *SOME*, **not** all, of this homework's problems involve lists.
- Remember that for functions involving lists whose lengths might vary -- as well as for functions involving itemization-style data -- you need to include **at least one test** for **each item in its itemized data definition** (and **sometimes more,** depending on the particular function's purpose)
  - So, given the Data Definition for a list, *IF* your function can accept a list of *any* size, make sure one of your tests has an **`empty`** list as its argument!
  - (You do ***NOT*** need a test for an `empty` list if that function expects a ***fixed***-size list, however.)
- Remember that a scanned copy of the "graphic design recipe helper" is posted in several places -- for example, one link to it is on the Canvas site home page -- whenever you would like a reminder of the steps you are required to follow in developing your functions.
- **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 IS EXPECTED TO follow **ALL** of the design recipe steps!
- **NOTE:** it is also fine and **encouraged** to define and use named constants when you notice there is some "set" value you are reusing!
- Signature and purpose statement comments are **ONLY** required for **functions that you have written and defined yourself** – you do **not** write them for named constants, or for functions that are already built into the Racket environment or provided modules.
  - That said, if you **copy** one of the in-class functions for use in your homework, DO also **copy** its signature, purpose, and `check-` expressions/tests as well as the function definition.
- **The design recipe is important!** You will receive **substantial** credit for the signature, purpose, header, and examples/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).

- Please let me know if you have any questions or concerns about the above requirements.

Problem 1 - 7 points

Problem 1 is correctly answering the "HW 5 - Problem 1 - Short-answer questions on recognizing kinds of data for which branching is useful" on the course Canvas site.

Problem 2 - 9 points

Problem 2 is correctly answering the "HW 5 - Problem 2 - Short-answer questions on lists" on the course Canvas site.

Homework File Setup for Problems 3 onward

Complete the remainder of Homework 5's problems in a file named **`111hw5.rkt`** (that is, save your DrRacket Definitions window to a file named **`111hw5.rkt`**). Be sure to save frequently!

Start up DrRacket, if needed set the language to How To Design Programs - **Beginning Student** *OR* **Beginning Student with List Abbreviations** level (your choice!), and add the HTDP/2e image **and** universe modules by putting these expressions at the beginning of your Definitions window:

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

Put a blank line, followed by these comments, adding in your name, and follow these with another blank line:

```
;  your name here
; CS 111 - HW 5
; last modified: 2025-09-29
```

Problem 3 - *helper* function - for a random-colored image - 8 points

Below this -- after the blank line -- type this string expression:

```
"=== Problem 3 ==="
```

...also followed by another blank line.

This function does NOT involve lists at all. It does define a "*helper* function" to be used in a later problem.

reminder: `random` function

- Recall BSL Racket's **`random`** function:

```
; signature: random: number -> number
; purpose: expects an integer number, and returns a pseudo-random number
;     that is also an integer within the range [0, given-integer)
;     (Note that it MIGHT return 0, but it will NOT return given-integer)
```

  – So, for example, **`(random 10)`**

    ...will return either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

- Also recall that BSL Racket has a `check-` function **`check-random`** that you can use for testing functions that happen to involve calls to function **`random`**.

reminder: `make-color` function

- Recall BSL Racket's **`make-color`** function:

`make-color` expects 3 arguments, each an integer in [0, 255], representing red, green, and blue values, respectively, and returns a `color` (actually a new data type!) made up of those red, green, and blue values.

- And **`make-color`**'s result can be used as the color argument for 2htdp/image functions like `circle`, `square`, etc.!

  For example:

  ```
  (star 100 "solid" (make-color 0 128 128))
  ```

Your task for Problem 3:

For some of the later problems, we want a function that expects a number, and returns a **randomly**-colored image whose size is somehow based on that number.

You (as the programmer/designer) get to **choose** what that image is -- it could be a star, a circle, a square, a triangle, image-text, some combination of images, etc. (It is **NOT** an argument the caller specifies.)

And, **you** get to choose how the given number affects its size. (It could **BE** its size, as for a circle's radius, or it might be used in a computation to result in part of an image's size, for example. You *could* use **`modulo`** with it if you'd like, also.)

FOR THE REST OF THIS HOMEWORK, when you see *your-chosen-img*, replace it with the name of the kind of image **YOU** chose here!

Using the design recipe, design such a function. Name this function **`random-`***your-chosen-img*, so I will recognize it as being your Problem 3 function.

REMEMBER: use **`check-random`** to write your tests/`check-` expressions for this function!

Problem 4 - several special-list helper functions

Next, in your definitions window, after a blank line, type this string expression:

```
"=== Problem 4 ==="
```

...followed by another blank line.

What if you wanted to represent information about an image that is to be part of a scene with a **4-item "image-state list"**, such that:

- the first item is the image's size (or is used to determine its size),
- the second item is the image's center's x-coordinate in the scene,
- the third item is the image's center's y-coordinate in the scene,
- and the fourth item is the image's current direction, either `"up"`, `"down"`, `"left"`, or `"right"`

That is, such a list could be written as:

```
(cons 50 (cons 10 (cons 20 (cons "left" empty))))
```

or as

```
(list 50 10 20 "left")
```

We'll give a data definition for this:

```
; DATA DEFINITION
; an ImageStateList is a list such that:
;     its first item is a number, a size in pixels
```

```
;      its second item is a number, an x-coordinate
;      its third item is a number, a y-coordinate
;      its fourth item is a string, "up", "down", "left", or "right"
```

Type or paste this data definition into your file.

Now, (in terms of class style standards), you can use **`ImageStateList`** as a type in function signatures for this homework.

In this problem, we'll create some convenient functions for use with such image-state lists.

4 part a - 8 points

After a blank line, type the string expression:

`"--- 4 part a ---"`

...followed by another blank line.

Using the design recipe, design a function **`get-size`** that expects an `ImageStateList`, and returns the size value in that image-state list.

For example:
`(get-size (list 20 3 6 "up"))` returns **`20`**.

4 part b - 8 points

After a blank line, type the string expression:

`"--- 4 part b ---"`

...followed by another blank line.

Using the design recipe, design a function **`get-x`** that expects an `ImageStateList`, and returns the x-coordinate in that image-state list.

For example:
`(get-x (list 20 3 6 "up"))` returns **`3`**.

4 part c - 8 points

After a blank line, type the string expression:

`"--- 4 part c ---"`

...followed by another blank line.

Using the design recipe, design a function **`get-y`** that expects an `ImageStateList`, and returns the y-coordinate in that image-state list.

For example:
`(get-y (list 20 3 6 "up"))` returns **`6`**.

4 part d - 8 points

After a blank line, type the string expression:

`"--- 4 part d ---"`

...followed by another blank line.

Using the design recipe, design a function **`get-direc`** that expects an `ImageStateList`, and returns the direction string in that image-state list.

For example:
`(get-direc (list 20 3 6 "up"))` returns **`"up"`**.

Problem 5 - function `draw-img-state-scene` - 9 points

Next, in your definitions window, after a blank line, type this string expression:

`"=== Problem 5 ==="`

...followed by another blank line.

Consider Problem 3's function **`random`**-*your-chosen-img*, and Problem 4's functions **`get-size`**, **`get-x`**, **`get-y`**, and **`get-direc`**, that expect image-state lists.

Using the design recipe, design a function **`draw-img-state-scene`** that expects an `ImageStateList`, and returns a scene of a randomly-colored image whose size, x-coordinate, and y-coordinate are those given in the image-state list. (This particular function does **NOT** happen to use the direction from this list.)

You are expected to use Problem 3's function (**`random`**-*your-chosen-img*) in **`draw-img-state-scene`**.

(For *this* function, define **named constants** for the scene width, scene height, and backdrop.)

For example, **`(draw-img-state-scene (list 20 30 80 "right"))`** returns a scene with a randomly-colored *your-chosen-img*, whose size is determined by 20, centered at (30, 80).

IMPORTANT: use **`check-random`** to write your tests/`check-` expressions for this function!

Problem 5 - OPTIONAL variation

Write your **`draw-image-state-scene`** to expect a **list** containing **TWO** image-state lists (instead of expecting an image-state list), and to return a scene a scene of TWO randomly-colored images whose size, x-coordinate, and y-coordinate are those given in the list's TWO image-state lists. The other problem requirements still apply to this optional variation.

Problem 6 - direction-related helper functions

Next, in your definitions window, after a blank line, type this string expression:

`"=== Problem 6 ==="`

...followed by another blank line.

6 part a - 8 points

After a blank line, type the string expression:

`"--- 6 part a ---"`

...followed by another blank line.

Using the design recipe, design a function **`is-direction`** that expects a **string**, and returns **`#true`** if that string is **`"up"`**, **`"down"`**, **`"left"`**, or **`"right"`**, and returns **`#false`** otherwise.

For example:

`(is-direction "up")` returns **`#true`**.

6 part b - 8 points

After a blank line, type the string expression:

`"--- 6 part b ---"`

...followed by another blank line.

Using the design recipe, design a function **`set-direc`** that expects an **`ImageStateList`** and a new desired direction -- expected to be **`"up"`**, **`"down"`**, **`"left"`**, or **`"right"`** -- and returns a image-state list with the same size, x-coordinate, and y-coordinate as in the given list, but with the new given direction as its direction. (BUT, if given a string that is **NOT** one of those directions, it just returns the image-state list **unchanged**.)

For example:

`(set-direc (list 20 3 6 "up") "left")` returns `(list 20 3 6 "left")`

`(set-direc (list 20 3 6 "up") "moo")` returns `(list 20 3 6 "up")`.

6 part b - OPTIONAL variation

If you did Problem 5's optional variation:

In **addition** to function `set-direc`, ALSO use the design recipe to write a function **`set-both`** that expects a list containing TWO image-state lists and a new desired direction, and returns a list of two image-state lists each with the same size, x-coordinate, and y-coordinate as in the given list, but each with the new given direction as its direction. It should appropriately use `set-direc`.

Problem 7 - function `move-center`, followed by a `big-bang`!

Next, in your definitions window, after a blank line, type this string expression:

`"=== Problem 7 ==="`

...followed by another blank line.

7 part a - 9 points

After a blank line, type the string expression:

`"--- 7 part a ---"`

...followed by another blank line.

Decide how much an image should move at a time. Declare this as a named constant, **`MOVE-AMT`**.

Using the design recipe, design a function **`move-center`** that expects an **`ImageStateList`**, and returns a new image state list whose x-coordinate and/or y-coordinate have been changed based on its direction:

- if its direction is **`"up"`**, JUST the new image-state list's **y**-coordinate should be **reduced** by **`MOVE-AMT`** (with the **same** size, x-coordinate, and direction)
- if its direction is **`"down"`**, the new image-state list's **y**-coordinate should be **increased** by **`MOVE-AMT`** (with the **same** size, x-coordinate, and direction)

- if its direction is **`"left"`**, the new image-state list's **x**-coordinate should be **reduced** by **`MOVE-AMT`** (with the **same** size, y-coordinate, and direction)
- if its direction is **`"right"`**, the new image-state list's **x**-coordinate should be **increased** by **`MOVE-AMT`** (with the **same** size, x-coordinate, and direction)

For example, if **`MOVE-AMT`** is defined to be 2, then:

`(move-center (list 70 10 20 "right"))` should return `(list 70 12 20 "right")`

`(move-center (list 70 10 20 "down"))` should return `(list 70 10 22 "down")`

7 part a - OPTIONAL variation 1

Design `move-center` so that, if the **new** coordinate value will be **outside** the bounds of the scene, it is instead set to be still in the scene, and with its direction changed to the opposite of its current value.

For example, if **`MOVE-AMT`** is defined to be 2. the scene's width is 200, and the scene's height is 100, then:

`(move-center (list 40 199 20 "right"))` should return `(list 40 200 20 "left")`

`(move-center (list 40 10 99 "down"))` should return `(list 40 10 100 "up")`

`(move-center (list 40 1 20 "left"))` should return `(list 40 0 20 "right")`

`(move-center (list 40 10 1 "up"))` should return `(list 40 10 0 "down")`

7 part a - OPTIONAL variation 2

If you did Problem 5's optional variation:

In **addition** to function `move-center`, ALSO use the design recipe to write a function **`move-both`** that expects a list containing TWO image-state lists and returns a new list of TWO image-state-lists, each of which has had its x-coordinate and/or y-coordinate changed based on its direction. It should appropriately use `move-center`.

7 part b - 4 points

After a blank line, type the string expression:

`"--- 7 part b ---"`

...followed by another blank line.

Now write a **`big-bang`** expression that:

- has an image-state list of your choice as its first argument
- has a **`to-draw`** clause using **`draw-img-state-scene`**
- has an **`on-tick`** clause using **`move-center`** and (optionally) how fast the ticker should tick
  - (use `move-both` here if you did the optional variations for Problem 5/6b/7a-option2)
- has an **`on-key`** clause using **`set-direc`**
  - (use `set-both` here if you did the optional variations for Problem 5/6b/7a-option2)