CS 111 - Homework 3

Deadline

11:59 pm on **Friday, September 20, 2024**

Purpose

To introduce BSL Racket's `random` pseudo-random function, its `modulo` function, and two more functions from the family of `check-` expressions, to get more practice using the design recipe to write and test functions, and to use at least one of your functions as an argument in a `big-bang` expression to create an animation.

How to submit

You complete **Problems 1, 2, and 3** on the course Canvas site (short-answer questions on `random` and `modulo`, and related to `big-bang`), so that you can see if you are on the right track.

Then, you will submit your work for **Problems 4 onward** on the course Canvas site.

Submit your `111hw3.rkt` file-in-progress with your work on Problems 4 onward early and often!

Important notes - 10 points

- Remember that 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.
- Continue to follow the course style standards discussed in class and included on prior assignments.
- **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 the available 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-expressions portions of your functions. Typically you'll get at least half-credit for a correct signature, purpose, header, and examples/check-expressions, 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 - 8 points

Problem 1 is correctly answering the "HW 3 - Problem 1 - Short-answer questions on Racket's `random` function" on the course Canvas site.

Problem 2 - 10 points

Problem 2 is correctly answering the "HW 3 - Problem 2 - Short-answer questions on Racket's `modulo` function" on the course Canvas site.

Problem 3 - 8 points

Problem 3 is correctly answering the "HW 3 - Problem 3 - Short-answer questions related to `big-bang`" on the course Canvas site.

Homework File Setup for Problems 4-7

Complete the remainder of Homework 3's problems in a file named `111hw3.rkt` (that is, save your DrRacket Definitions window to a file named `111hw3.rkt`). Be sure to save frequently!

Start up DrRacket, if needed set the language to How To Design Programs - **Beginning Student** level, 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 3
; last modified: 2024-09-16
```

Problem 4 - trying out some more useful built-in functions - 12 points

The purpose of this problem is to try out some more functions that Racket provides.

You are NOT writing ANY function definitions in this problem -- you are **JUST** writing some **compound expressions** *trying out* these functions.

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

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

...followed by another blank line.

Now type the **compound expressions** specified below in the Racket Definitions window.

BEFORE 4 part a - quick intro to `random`

Consider: BSL Racket has a function named `random`, that can be described as follows:

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

4 part a

After a blank line, type the string expression:

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

...followed by another blank line.

JUST to TRY OUT `random`:

Write at least **three** compound expressions of your choice using `random`.

Run these several times, and observe that you do get **different** values when you run these, but they will **always** be in the **range** described above.

BEFORE 4 part b - quick intro to `check-random`

There's a `check-` expression that is very useful when testing functions whose function body uses `random`: `check-random` expects an expression to test and its expected value when `random` is involved, and provides the SAME pseudo-random number to BOTH expressions, so you can still have convenient testing!

4 part b

After a blank line, type the string expression:

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

...followed by another blank line.

JUST to TRY OUT `check-random`:

Type and run this expression in your `111hw3.rkt`:

```
(check-random (random 1000)
              (random (+ 2 998)))
```

Run this several times, and see that this test passes each time.

BEFORE 4 part c - quick intro to `modulo`

Consider: BSL Racket has a function named `modulo`, that can be described as follows:

```
; signature: modulo: number number -> number
; purpose: expects an integer dividend (numerator) and an
;      integer divisor (denominator),
;      and it returns the integer REMAINDER from the integer
;      division of that dividend and divisor
;
;      (that is,
;          (modulo 6 3) returns 0, because there is no integer
;              remainder from 6 / 3 --
;              that is, 6 / 3 is 2, with a remainder of 0
;          (modulo 7 4) returns 3, because there is an integer
;              remainder of 3 from 7 / 4 --
;              that is, 7 / 4 is 1, with a remainder of 3
;          (modulo 21 2) returns 1, because there is an integer
;              remainder of 1 from 21 / 2 --
;              that is, 21 / 2 is 10, with a remainder of 1)
```

This operation turns out to be surprisingly useful in computing!

- Consider: what is the biggest integer remainder when you divide any integer by 10?
  - ...that remainder has to be between 0 and 9, right?
- And what is the biggest integer remainder when you divide any integer by 50?
  - ...that remainder has to be between 0 and 49, right?

So, whenever you NEED an integer value to be guaranteed to be within a certain range, no bigger and no smaller, `modulo` can help make that possible. That is,

```
(modulo my-function-parameter DESIRED-LIMIT)
```

...is guaranteed to be an integer between 0 and (`DESIRED-LIMIT` - 1), no matter what argument expression a caller might try to use.

4 part c

After a blank line, type the string expression:

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

...followed by another blank line.

JUST to TRY OUT `modulo`:

Write at least **three** compound expressions of your choice using `modulo`.

Run these, and make sure you understand their results; notice that, no matter HOW large a first argument you give to `modulo`, the result is NEVER more than (its second argument - 1)!

BEFORE 4 part d - quick intro to `check-within`

Finally, we will practice with another member of the `check-` function family that we have briefly mentioned earlier.

Sometimes, when a function returns a number that's not an integer, especially a number such as the result of `(/ 1 3)`, it becomes difficult to write an exact expected value in a `check-expect` expression – after all, 1/3 is an infinite decimal, and 0.333 won't exactly match that. It would be better in this case to have the expected value be "close enough" to the actual value.

`check-within` lets us test to see if the expected answer and the calculated answer are "close enough" to sufficiently test our functions.

The `check-within` function expects **THREE** arguments:

- the expression being tested, which in `check-within`'s case should be of type `number`
- an **approximate** expected value for that expression being tested
- a number that represents the largest that the **difference** between the expression and the expected value can be and still consider this to be a passing test. (In math, this maximum difference is also called a **delta**.)

As an example, to test whether Racket's built-in constant `pi` is within 0.001 of our usual estimate of pi, `3.14159` -- the `check-within` expression would look like this:

```
(check-within  pi         ; expression being tested
               3.14159    ; APPROXIMATE expected value
               .001)      ; how close we consider to be "close enough"
```

And, to test whether Racket's built-in constant `e` is within 0.0001 of our usual estimate of e, `2.71828` -- the `check-within` expression would look like this:

```
(check-within  e          ; expression being tested
               2.71828    ; APPROXIMATE expected value
               .0001)      ; how close we consider to be "close enough"
```

(Yes, `pi` and `e` are Racket system-provided named constants – I wish they were written in all-uppercase, but they are not. We, however, WILL write our self-defined constants in upper case characters!)

4 part d

After a blank line, type the string expression:

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

...followed by another blank line.

JUST to TRY OUT `check-within`:

- Write a `check-within` expression that results in a **passing** test for testing what the expression

  `(/ 3 13)`

  ...**should** be **approximately** equal to. Use an estimate that is within `0.01` of the actual value.
- Write a `check-within` expression that will result in a **passing** test for testing what the expression

  `(* pi 100)`

  ...should be **approximately** equal to.

Problem 5 - choose a numeric function - 13 points

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

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

...followed by another blank line.

CHOOSE ONE of the FOLLOWING OPTIONS for this problem, functions involving numbers where you might find `check-within` or `check-random` to be useful for their tests! (You can choose to do all of them for the practice, but I will only grade one of them, in the interests of time... `8-)`).

For full credit, be sure to include **ALL** of the design recipe steps.

5 option 1 - function `fahr->cels`

The Celsius temperature scale, used by most of the rest of the world, is different from the Fahrenheit temperature scale frequently used in the U.S.. Using the design recipe, design a function `fahr->cels` that expects a temperature given in **Fahrenheit** and returns the equivalent **Celsius** temperature.

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

Note:

- It is reasonable to search the web or an appropriate reference for the conversion formula for this -- consider this as part of thinking about the problem, as part of the first step of the design recipe.

5 option 2 - function `random-in-range`

Someone asked during class whether BSL Racket had a random function that let you specify a desired range of values to choose from. It does not, but you could design such a function and use the provided `random` function to help to build it.

Using the design recipe, design a function `random-in-range` that expects a beginning integer and an ending integer representing a range, the beginning integer assumed to be strictly less than the ending integer, and returns a pseudo-random integer in the range [beginning integer, ending integer).

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

5 option 3 - function dist-btwn

You can compute the distance between two points (x_1, y_1) and (x_2, y_2) using the formula:

$$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Using the design recipe, design a function `dist-btwn` that expects the x and y coordinates for one point and the x and y coordinates for another point, and returns the distance between those two points.

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

Note: you might find `check-within` useful for its tests!

5 option 4 - function redwood-bst

In computing carbon sequestration, **bst**, stem biomass, can be computed using a tree's diameter at breast height, **dbh**, in centimeters, using the formula:

$$bst = e^{a + (b * \ln(dbh))}$$

...where `a` and `b` are constants determined by analyzing sets of measurements of a given tree type.

For redwoods,

```
a = -2.0336
b = 2.2592
```

Using the design recipe, design a function `redwood-bst` that expects a redwood tree's dbh and returns its bst.

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

Note:

- in Racket, you can compute `ln(x)` using `(log x)`
- in Racket, you can compute the value from raising `e` to some value using the `exp` function -- that is, you can compute e^x using `(exp x)`

Problem 6 - choose an image function - 13 points

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

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

...followed by another blank line.

CHOOSE ONE of the FOLLOWING OPTIONS for this problem, functions involving images. (You can choose to do both for the practice, but I will only grade one of them, in the interests of time... `8-)`).

For full credit, be sure to include **ALL** of the design recipe steps.

6 option 1 - function `name-badge`

Consider a function `name-badge` that expects a desired name, desired letter color, and desired background color, and returns an image of a "name-badge" for that name displayed in letters of the given letter color in the middle of a "badge" with the given background color.

Using the design recipe, design this function `name-badge`. For full credit, be sure to include **ALL** of the design recipe steps.

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

Notes:

- YOU get to decide on the **shape** of this name-badge image (the user does **not** specify it, *you* will choose it as part of **designing** this function).
- YOU also get to decide on the font-size you want to use for the given name's letters on your resulting name-badge. (It would be good to make your chosen font size a named constant!)
- For this function, make sure that your tests/`check-expect` expressions involve names of **different** lengths, with **different** letter colors and **different** background colors.
- An interesting feature to try here, if you would like: within your function, determine the width of your "name-badge" image based on the given name's length.
  - (HINT: there is more than one way to do this -- consider how a function like `image-width` or `string-length` could help with this, for example.)

BEFORE 6 option 2 - quick intro to `make-color`

FUN FACT: BSL Racket includes a `make-color` function.

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

(Optionally, it can take a 4th argument, another integer in [0, 255], giving the transparency of that color (0 is completely transparent, 255 is completely opaque).

But - to play with this is a little inconvenient, because to "see" what the returned color looks like, you have to use the resulting color in some image function.

For example, you can run:

```
(make-color 100 200 150)
```

...but it doesn't return what the color looks like; it returns `(make-color 100 200 150 255)`.

But, you **can** see the resulting color if you run something like:

```
(square 30 "solid" (make-color 100 200 150))
```

6 option 2 - function `color-play`

Using the design recipe, design a function `color-play` that expects desired red, green, and blue values, and returns a solid image (your choice of shape!) whose color has those red, green, and blue values. For full credit, be sure to include **ALL** of the design recipe steps.

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

Note:

- YOU get to decide the returned image's shape (the user does **not** specify it, *you* will choose it as part of **designing** this function).
- YOU get to pick a reasonable constant size for your chosen shape (the user does **not** specify this, *you* will choose this as part of **designing** this function).

OPTIONAL: IF you would like, you can design this function to **also** expect a transparency value (that is, you can design it to expect 4 arguments instead of 3 arguments). No extra credit, it's just something you can try. If you choose to do so, you should make sure your signature reflects this and purpose statement explains this!

Optional EXPERIMENT to try: What happens if you call your function `color-play` with three arguments of `(random 256)`?

Problem 7 - a function that expects a `number`, returns a `scene` - 26 points

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

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

...followed by another blank line.

7 part a - 6 points

After a blank line, type the string expression:

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

...followed by another blank line.

You are going to be creating some scenes in later parts of this problem, so for this part, define the following named constants:

- define a named constant `SC-WIDTH`, a scene width of your choice (but not bigger than 600 pixels).
- define a named constant `SC-HEIGHT`, a scene height of your choice (but not bigger than 400 pixels).
- define a named constant `BACKDROP`, an unchanging, constant scene that will serve as a backdrop scene in some expressions.
  - its size should be `SC-WIDTH` by `SC-HEIGHT` pixels
  - it can be empty or not, your choice!
- (you may certainly include **additional** named constants, also, if you wish!)

7 part b - 12 points

After a blank line, type the string expression:

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

...followed by another blank line.

Consider -- you are now going to write a function that creates a scene that is somehow based on some single given number (as is our scene function from Week 3 Lecture 2, and as is the Week 3 Lab Exercise's `number-scene` function).

Using the design recipe, design a function `draw-hw3-scene` function that expects a number, and returns a scene in which one or more images have been placed into your `BACKDROP` in such a way that something in

that scene is based on that given number. For full credit, be sure to include **ALL** of the design recipe steps.

For example:

- image(s)' **size**(s) might somehow be based on that number
- image(s)' **x-coordinate** and/or **y-coordinate** might somehow be based on that number
- image(s) **color**(s) might somehow be based on that number
- **etc**.! (And **combinations** of these are fine, as long as that combination is based on a **SINGLE** number.)

ALSO: in ADDITION to your function's tests, write at least 2 compound expressions of your choice using your function (so that you will see their results) after your function definition.

Note:

- Do something that is at least slightly different than the posted class examples/lab exercises.

7 part c - 8 points

After a blank line, type the string expression:

`"--- 7 part c ---"`

...followed by another blank line.

Write a `big-bang` expression with

- a first argument that is a number of your choice
- a second argument that is `(on-tick add1)`
- a third argument that is `(to-draw draw-hw3-scene)`

Now you should see an animation involving something changing, and now you are done with this problem.

OPTIONAL:

- If you would like, you may use a function other than `add1` for `big-bang`'s `on-tick` expression.
- You can also change the **speed** of the ticker in the `on-tick` clause by giving `on-tick` an optional *second* argument: a number that gives the number of seconds per tick.
  - The default value is `(/ 1 28)` or 1/28 of a second, which means it ticks 28 times per second.
  - To go **slower**, put a number greater than 0.036.
  - To go **faster**, put a number less than 0.035 (but greater than 0).
  - For example: `(on-tick add1 0.2)`

    ...will tick more slowly, only 5 ticks per second (about 0.2 seconds per tick)