CS 111 - Homework 6

Deadline

11:59 pm on **Friday, October 11, 2024**

Purpose

To provide a bit more practice with file input and output and with lists (including using the design recipe to write functions that "walk through" a list).

How to submit

Submit your `111hw6.rkt` file ***and also*** any `.txt` files you create for *testing* your functions for these problems on the course Canvas site. (Remember, submit early and often!)

(I think it is OK if you do not submit copies of `.txt` files that your `111hw6.rkt` *creates* -- we should "get" copies of those when we run your `111hw6.rkt`.)

Important notes

- Please note that only *SOME*, **not** all, of this homework's problems involve lists, and only *SOME*, not all, involve file input and/or file output.
- 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 on the public course web site, at the end of the "References" section, if 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.

Homework File Setup

Complete these problems in a file named `111hw6.rkt` (that is, save your DrRacket Definitions window to a file named `111hw6.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 **AND batch-io** modules by putting these expressions at the beginning of your Definitions window:

```
(require 2htdp/image)
(require 2htdp/universe)
(require 2htdp/batch-io)
```

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 6
; last modified: 2024-10-07
```

Problem 1 - function `emphasize-list`

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

```
"=== Problem 1 ==="
```

...followed by another blank line.

Now, we want to consider lists whose contents are ONLY strings!

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. (And if called with an empty list, it should simply return an empty list.) That is, the expression:

```
(emphasize-list (cons "Hey" (cons "Oh my!" (cons "mooo" empty))))
```

...should return:

```
(cons "Hey!!" (cons "Oh my!!!" (cons "mooo!!" empty)))
```

Problem 2 - a little more file output and file input practice

For this problem, you are **not** writing any new functions -- you are writing several compound expressions.

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

```
"=== Problem 2 ==="
```

...followed by another blank line. (And double-check that you put: **`(require 2htdp/batch-io)`** ...in your `111hw6.rkt` file as noted in the **Homework File Setup** section!)

2 part a

After a blank line, type the string expression:

```
"--- 2 part a ---"
```

...followed by another blank line.

Remember: in the Week 6 Lab Exercise, you tried out module `2htdp/batch-io`'s **`write-file`** function. (And its signature and purpose are included in the Week 6 Lab Exercise.)

Write an expression, **using `write-file`**, whose side-effects should be:

- to create a file whose **whose name BEGINS with YOUR last name, and has the suffix (ENDS with)** `.txt`
- that contains **at least 5 lines** worth of words or phrases of your choice,
- making sure that **at least two lines** contain a phrase of **more** than one word each
  - (Remember: `write-file` only expects two `string` arguments -- the name of the file to write to, and the string to write there -- but you can use `"\n"` within a string to say that you want a newline character.)

2 part b

After a blank line, type the string expression:

```
"--- 2 part b ---"
```

...followed by another blank line.

Remember: in the Week 6 Lab Exercise, you tried out module `2htdp/batch-io`'s **`read-lines`** function. (And its signature and purpose are included in the Week 6 Lab Exercise.)

FIRST: Write an expression, using **`read-lines`**, whose value will be a **list** of strings, such that each string is **one line** from the file you wrote in Problem 2 part a.

Hey -- Problem 1's `emphasize-list` function expects a list of strings! So:

SECOND: **ALSO write an expression** in which you call Problem 1's `emphasize-list` function with that `read-lines` expression, reading from the file you wrote in Problem 2 part a, as its argument.

2 part c

After a blank line, type the string expression:

```
"--- 2 part c ---"
```

...followed by another blank line.

Remember: in the Week 6 Lab Exercise, you tried out module `2htdp/batch-io`'s **`read-words`** function. (And its signature and purpose are included in the Week 6 Lab Exercise.)

FIRST: Write an expression, using **`read-words`**, whose value will be a **list** of strings, such that each string is **one "word"** from the file you wrote in Problem 2 part a.

Again, Problem 1's `emphasize-list` function expects a list of strings -- so:

SECOND: ALSO write an expression in which you call Problem 1's `emphasize-list` function with that `read-words` expression, reading from the file you wrote in Problem 2 part a, as its argument.

Problem 3 - function emphasize-from-file

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

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

...followed by another blank line.

Following the design recipe, design and write a function `emphasize-from-file` that expects a file name, has the side effect of reading from that file, and returns a list of strings, each of which is a line from that file followed by `!!`. For full credit, use `emphasize-list` appropriately in `emphasize-from-file`'s body.

Hint:

- The body of this function should be quite short!

Problem 4 - a function that writes to a file

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

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

...followed by another blank line.

CHOOSE ONE of the FOLLOWING OPTIONS for this problem. (You can choose to do more than one for the practice, but only one of them will be graded, in the interests of time... `8-)`).

option 4-1 - function ask-to-file

If you chose this option, after a blank line, type the string expression:

`"--- Problem 4 - option 1 ---"`

...followed by another blank line.

Consider function `ask-how-doing`, from Homework 2 - Problem 8. Copy its signature, purpose, tests/`check-` expressions and definition into `111hw6.rkt`.

Then, following the design recipe, design and write a function `ask-to-file` that expects a desired file name and a person's name, has the side-effect of trying to write to that file name `How are you doing,` ...that person's name... `?`, and returns the name of the file written. For full credit, use `ask-how-doing` appropriately in `ask-to-file`'s function body.

Hint:

- The body of this function should be quite short!

option 4-2 - function doubt-to-file

If you chose this option, after a blank line, type the string expression:

`"--- Problem 4 - option 2 ---"`

...followed by another blank line.

Consider function `doubt-it`, from Week 3 - Lecture 1. Copy its signature, purpose, tests/`check-` expressions and definition into `111hw6.rkt`.

Then, following the design recipe, design and write a function `doubt-to-file` that expects a desired file name, a desired exclamation, and a phrase to doubt, has the side-effect of trying to write to that file name that exclamation in all-uppercase followed by !, then a blank, then the phrase to doubt followed by ??, then a blank, then the phrase to doubt followed by ??, and returns the name of the file written. For full credit, use `doubt-it` appropriately in `doubt-to-file`'s function body.

Hint:

- The body of this function should be quite short!

option 4-3 - function emphasize-to-file

If you chose this option, after a blank line, type the string expression:

`"--- Problem 4 - option 3 ---"`

...followed by another blank line.

Consider function `string-list-smush`, from Week 6 - Lecture 2. Copy its signature, purpose, tests/`check-` expressions and definition into `111hw6.rkt`.

Following the design recipe, design and write a function `emphasize-to-file` that expects a file name and a list of strings, has the side effect of trying to write to that file name each of the strings in that list followed by `!!` on its own line, and returns the name of the file written. For full credit, use `emphasize-list` and `string-list-smush` appropriately in `emphasize-to-file`'s body.

Hint:

- The body of this function should be quite short!

Problem 5 - another recursive function

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. (You can choose to do more than one for the practice, but only one of them will be graded, in the interests of time... `8-)`).

option 5-1 - function words-to-images

If you chose this option, after a blank line, type the string expression:

`"--- Problem 5 - option 1 ---"`

...followed by another blank line.

Using the design recipe, write a function `words-to-images` that expects a list of words, and returns a list of image-versions of those words. (You may choose the font size and color for the image-versions of the words -- they may be always the same, or you may creatively use `random`, your choice!)

option 5-2 - functions images-to-scene

If you chose this option, after a blank line, type the string expression:

`"--- Problem 5 - option 2 ---"`

...followed by another blank line.

Using the design recipe, write a function `images-to-scene` that expects a list of images, and returns a scene in which those images have been placed at random locations within that scene.

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

Now write a `big-bang` expression that:

- has a list of images as its first argument
- has a `to-draw` clause using `images-to-scene`
- has an `on-tick` clause using `rest` and (optionally) how fast the ticker should tick
- has a `stop-when` clause using `empty?`

option 5-3 - function `many`-your-chosen-img-`scene`

If you chose this option, after a blank line, type the string expression:

```
"--- Problem 5 - option 3 ---"
```

...followed by another blank line.

Consider your function `random-`*your-chosen-img* from Homework 5 - Problem 3, that expects a number, and returns a **randomly**-colored image whose size is somehow based on that number. Copy that function's signature, purpose, `check-` expressions, and function definition into your `111hw6.rkt` file.

Now consider a list of numbers in which each number represents the size of the kind of image you chose for your function `random-`*your-chosen-img*.

Using the design recipe, develop a function `many-`*your-chosen-img*`-scene`, which expects a list of numbers representing image sizes, and returns a **scene** containing instances of that kind of image with **those** sizes but of random colors, each placed in the center of the scene.

(**Optional** variation: you can place them **randomly** in the scene, also, IF you prefer, instead of having them all in the center.)

You should use your `random-`*your-chosen-img* function in your `many-`*your-chosen-img*`-scene` function.

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

Copy the signature, purpose, `check-` expressions, and function definition for Week 6 Lecture 1's function `add1-list` into your `111hw6.rkt`.

Now write a `big-bang` expression that:

- has a list of positive numbers as its first argument
- has a `to-draw` clause using `many-`*your-chosen-img*`-scene`
- has an `on-tick` clause using `add1-list` and (optionally) how fast the ticker should tick

option 5-4 - function dot-product

If you chose this option, after a blank line, type the string expression:

```
"--- Problem 5 - option 4 ---"
```

...followed by another blank line.

Consider the concept of a **dot product** of two vectors:

Vector-a = (a1 a2 ... aN) where each "a" value is a `number`

Vector-b = (b1 b2 ... bN) where each "b" value is a `number`

The dot-product of Vector-a and Vector-b is, then, is calculated as a `number` to be:

(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.