CIS 130 - Homework #3

Due:

Thursday, February 18th, 11:59 pm

Purpose:

More practice writing functions using the design recipe and writing and using named constants, some of these with the purpose of creating a simple animation.

How to submit:

When you are done with the following problems:

- save your resulting Definitions window contents in a file with the suffix `.ss` or `.scm`
- transfer/save that file to a directory on **nrs-labs**
- use **ssh** to connect to **nrs-labs**
- cd to the directory/folder where you saved it (`cd 130hw3` for example)
- use `~st10/130submit` to submit it
- make sure that `~st10/130submit` shows that it submitted your homework `.ss` or `.scm` file
- (ASK ME if this is not clear, or if you have any problems with submission!)

Important notes:

- Each student should work individually on this assignment.
- Remember, you are expected to follow the Design Recipe for all functions that you write. (But that's for functions, not for when you give a name a value, such as when you define a named constant.) Functions that do not include the expected design recipe parts will not receive full credit.
- Here are some beginning style guidelines for our Scheme code -- we will add to these as the semester continues:
    - You should choose **descriptive** names for identifiers (parameters, function names, named constants, those names that the programmer chooses).
    - Putting a blank line before and after function definitions makes them easier to read
    - The body of a function should never be on the same line as its header -- the body should always start on the next line after the function header.
    - The body of the function should always be indented under the function header.
    - Use hard returns rather than letting lines of code wrap to the next line.

Homework problems:

Problem 0

For this assignment, you should be using the **Beginning Student** language in DrScheme, with the **universe.ss** and **fabric-teachpack.scm** teachpacks installed.

In the definitions window, type in a comment-line containing your name, followed by a comment-line

containing CIS 130 - HW 3, followed by a comment-line with no other text in it --- that is,

```
; type in YOUR name
; CIS 130 - HW 3
;
```

Problem 1

Below what you typed in Problem 0 above, type the comment lines:

```
; Problem 1
;
```

(Adapted from a problem by Karen O'Loughlin, Ankeny High School)

To warm up: imagine a class in which the final semester grade is determined as follows:

- The average of all homework scores determines 45% of the final semester grade.
- The average of all quizzes' scores determines 20% of the final semester grade.
- The final exam score determines 35% of the final semester grade.

1 part a

Define named constants for the amount of the final semester grade determined by the homework average, for the amount of the final semester grade determined by the quiz score average, and for the amount of the final semester grade determined by the final exam score.

1 part b

Now, using the design recipe, design a function that expects a homework average, a quiz score average, and a final exam score, and it produces the final semester grade for someone with those scores. For full credit, make appropriate use of your named constants from part a in your function.

Provide at least two examples for this function.

Problem 2

Skip a line, and write a comment noting that what follows are your expressions for:

```
; Problem 2
;
```

In the "Elaborating DrScheme scenes" handout -- available from the public course web page along with the Week 4 lab exercise, as well as along with this homework handout -- you will find a description of a make-color function, that expects three numbers, a red-value between 0 and 255, a green-value between 0 and 255, and a blue-value between 0 and 255, and produces a result of type color that is that "mix" of those red-green-blue -- or RGB -- values.

It turns out that the fabric-teachpack.scm and universe.ss teachpack functions that expect a color string can also accept the result of a make-color expression. (The fabric-teachpack.scm functions did not used to work with them -- thus the comment in the "Elaborating DrScheme scenes" handout -- but they seem to now, in my experiments!) That is, these work:

```
(create-solid-fabric (make-color 255 0 0) 100 100)

(circle 40 "solid" (make-color 130 40 50))
```

From now on, then, when a function expects or produces a color, you can consider either a string containing a color's name or the result of the make-color expression to be of type `color` (that is, the contract for `make-color` would be:

```
; contract: make-color: number number number -> color
```

2 part a

To practice with this function, write at least three expressions (different from the two above) that:

- result in images, and
- use `make-color` with different combinations of red, green, and blue values than in the two above examples

2 part b

Now, consider: if you had a function that expects a time-counter value, and produces a color that is different for different time-counter values as a result, you would need to make sure that you always use `make-color` with red, green, and blue values that are strictly between 0 and 255.

(rhetorical question/hint:) If you called the `modulo` function with a time-counter parameter as its first expression, what could you use as the second expression to ensure that the resulting value would always be between 0 and 255?

Using the design recipe, design a function `get-color` that expects a time-counter value, and produces a color whose green-value is determined by the result of using the `modulo` function with that time-counter value and with that second expression you need to make sure the result is between 0 and 255. You can use whatever values you would like for the not-changing red-value and blue-value.

I'll provide the check-expects for this function: if you have defined `MY-RED` and `MY-BLUE` to be your desired red and blue values, then the following tests should pass, if your function is working properly:

```
(check-expect (get-color 57)
              (make-color MY-RED 57 MY-BLUE))
(check-expect (get-color 1000)
              (make-color MY-RED 232 MY-BLUE))
```

Problem 3

Skip a line, and write a comment noting that what follows are your expressions for:

```
; Problem 3
;
```

Now, imagine that, given a time-counter value, you would like a solid circle whose radius is the time-counter value modulo some maximum radius that you choose, and whose color is the color that `get-color` returns for that time-counter value.

Define a named constant for your desired maximum radius for this circle, and then, using the design recipe, design a function `get-circle` that expects a time-counter value, and produces just such a circle image. To receive full credit, your function should appropriately use `get-color`.

Provide at least two examples for this function, at least one of which has a time-counter value larger than 300.

Problem 4

Skip a line, and write a comment noting that what follows are your expressions for:

```
; Problem 4
;
```

4 part a

Define named constants for your desired scene `WIDTH` and `HEIGHT` for an animation -- **but** for this particular homework, make sure that they are **not** the same value; make sure that the difference between them is at least 50 pixels. For the rest of this homework, use these `WIDTH` and `HEIGHT` named constants to represent the scene's width and height in expressions you write.

4 part b

Define a named constant `BACKDROP` that has as its value a desired scene with at least four visible images "placed" within it. (You can put whatever you'd like -- circles, rectangles, images from the web, etc. Remember, you can use the "Elaborating DrScheme scenes" handout for ideas.)

After this definition, put the now-simple-expression `BACKDROP` in your Definitions window, so your backdrop will appear in the Interactions window when the Definitions window contents are Run.

Problem 5

Using the design recipe, design a function `create-circle-scene` that expects a time-counter value, and produces a scene that:

- places the image that results from calling Problem 3's `get-circle` function into your backdrop from Problem 4,
- ...such that the time-counter helps to determine either the x-coordinate, the y-coordinate, or both where it will be placed in the scene,
- ...using `modulo` so that, no matter what the time counter is, the image will always be placed *within* the scene (at least part of the circle should always be visible in the scene, no matter how big the time counter value happens to be)

You may have additional moving or changing images within your created scene if you would like, as long as the above requirements are also met.

In writing your check-expects for this function, include at least 2 check-expects, and make sure that at least one of your check-expects is for a time-counter value larger than both the `WIDTH` and `HEIGHT` of your `BACKDROP`.

Then, write an `animate` expression that uses your `create-circle-scene` function to create an animation.