CS 111 - Homework 3

Deadline

11:59 pm on **Friday, September 16, 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 `111hw3.rkt`
  - Note: please use that *exact* name -- do not change the case, add blanks, etc. If I search for a file with that name in your submission, I want it to show up! 8-)
- transfer/copy that file to a directory on `nrs-labs.humboldt.edu` (preferably in a folder/directory named `111hw3`)
  - If you are in a campus lab, you can do so by copying them to a folder on the `U:` drive
  - If you are not in a campus lab, you can do so by using `sftp` (secure file transfer) and connecting to `nrs-labs.humboldt.edu`, and then transferring them.

    Graphical versions of `sftp` include WinSCP and Secure Shell Transfer Client for Windows, Cyberduck and Fugu for Mac OS X, and FileZilla that has versions for both Windows and Mac OS X. (Mac OS X and Linux also come with a command-line `sftp` -- an intro to this is included in the posted handout, "useful details - `sftp`, `ssh`, and `~st10/111submit`", available on the public course web site in the References section.)
  - Whichever version of `sftp` you are using, use a **host name** of `nrs-labs.humboldt.edu` if you are not in a campus lab (you can get away with just `nrs-labs` for the host name in a campus lab), your campus username, and when you are prompted for it, your campus password. IF you need to give a port number, give a port number of **22**.
- 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.
  - In BSS 313, the Windows computers have a graphical version of `ssh` called PuTTY. There is a graphical version of `ssh` for Mac OS X named Jelly FiSSH, (although Mac OS X and Linux also come with command-line `ssh`, and an intro to this is also included in the "useful details - sftp, ssh, and ~st10/111submit" handout.)
  - Again, whichever version of `ssh` you are using, use a **host name** of `nrs-labs.humboldt.edu` if you are not in a campus lab (you can get away with just `nrs-labs` for the host name in a campus lab), your campus username, and when you are prompted for it, your campus password. IF you need to give a port number, give a port number of **22**.
- 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 `111hw3`, then you would go to that directory by saying:

    ```
    cd 111hw3
    ```
  - use the `ls` command to make sure your `111hw3.rkt` file is really there:

    ```
    ls
    ```

  - type the command:

    `~st10/111submit`

    ...and when asked, enter a homework number of `3`

    ...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 `111hw3.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 `111hw3.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...)

- (we practiced the above during the Week 1 Lab, on Friday, August 26 -- ASK ME if this is still not clear, or if you have any problems with submission! If you get stuck, E-MAIL me your `111hw3.rkt` file as an e-mail attachment before the deadline, and then submit it as soon after that as you are able.)
  - I am happy to walk through submitting files with you -- just come by my office during office hours, or e-mail me to set up an appointment.

Purpose

To get more practice using the design recipe to write and test functions.

Important notes

- 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: after thinking about the function and the kind of data involved, 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.
- Remember: A **signature** in Racket is written in a comment, and it includes the 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: purple-star: number -> image
  ```

- Remember: a **purpose statement** in Racket is written in a comment, and it **describes** what the function expects and **describes** what the function returns (and if the function has side-effects, it also **describes** those side-effects). For example,

  ```
  ; purpose: expects a size in pixels, the distance between
  ;    points of a 5-pointed-star, and returns an image
  ;    of a solid purple star of that size
  ```

- Remember: it is a COURSE STYLE STANDARD that named constants are to be descriptive and written in all-uppercase -- for example,

  ```
  (define WIDTH 300)
  ```

- 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 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).

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 3`,
- 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 3
; last modified: 2016-09-09
;
```

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

```
;
; Problem 1
;
```

Note: you are **NOT** writing a function in this problem -- you are just writing two compound expressions.

Sometimes, when a function returns a number with a fractional part, especially a number such as the result of `(/ 1 3)`, it becomes difficult to give an expected value in a `check-expect` that is "exact enough" for a passing test. Yet, if the expected value is "close enough" to the actual value, we might like to be able to decide that that is good enough for our purposes.

That is why, along with several additional `check-` functions, BSL Racket also includes `check-within`, which 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
- a number that is 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 a quick example, what if you simply wanted to test whether BSL Racket's predefined `pi` was within .001 of `3.14159`? This `check-within` expression can do this, and show that it is:

```
; passing test

(check-within pi
              3.14159
              .001)
```

But if you want to know if its predefined `pi` is within .000001 of `3.14159`, this `check-within` expression can do this, and show that it is not:

```
; failing test

(check-within pi
              3.14159
              .000001)
```

Just to **practice** writing a few expressions using `check-within`:

- write a `check-within` expression that will result in a passing test for testing what the expression

  `(/ 2 3)`

  ...should be roughly equal to;
- write a `check-within` expression that will result in a passing test for testing what the expression

  `(* pi 10)`

  ...should be roughly equal to.

Problem 2 - HERE PLEASE - continue F16 conversion here!

Next, in your definitions window, after a blank line, type the comment lines:

```
;
; Problem 2
;
```

True story: my spouse likes to set every item that displays a temperature to display that temperature in Celsius. That prompts an idea: I decide I'd like a function that expects a temperature given in Fahrenheit, and returns the equivalent Celsius temperature. Write such a function `fahr->cels`, using the following design recipe steps:

(It is reasonable to search the web or an appropriate reference for the conversion formula for this -- consider it problem research.)

2 part a

Write an appropriate **signature** comment for this function.

2 part b

Write an appropriate **purpose statement** comment for this function.

2 part c

Write an appropriate **function header** for this function (putting `...`) for its body for now).

2 part d

Write at least 2 specific tests/`check-expect` or `check-within` expressions for this function.

2 part e

Only NOW should you replace the `...` in the function's body with an appropriate expression to complete this function. Run and test your function until you are satisfied that it passes its tests and works correctly.

Finally, include at least 2 example calls of your function (such that you will see their results) after your function definition.

Problem 3

Next, in your definitions window, after a blank line, type the comment lines:

```
;
; Problem 3
;
```

I decide I'd like a function `name-badge` that expects a name and a color, and returns an image of a "name badge" for that name with that "background" color: a shape (of your choice of shape) of the given color with the given name (of your choice of font size, but using black letters) in its middle. Somehow make the width of your "badge" be based on the name's length.

(hint: remember that `string-length` expects a string and returns how many characters are in that string.)

3 part a

Write an appropriate **signature** comment for this function.

3 part b

Write an appropriate **purpose statement** comment for this function.

3 part c

Write an appropriate **function header** for this function (putting `... )` for its body for now).

3 part d

Write at least 2 specific tests/`check-expect` or `check-within` expressions for this function.

3 part e

Only NOW should you replace the `...` in the function's body with an appropriate expression to complete this function.

Run and test your function until you are satisfied that it passes its tests and works correctly.

Finally, include at least 2 example calls of your function (such that you will see their results) after your function definition.

Problem 4

We've mentioned that BSL Racket includes a `make-color` function. `make-color` expects three

arguments, each an integer number in [0, 255], representing its red, green, and blue values, respectively, and returns a color 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 has no transparency).

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.

Design a function `color-dot` that expects desired red, green, and blue values, and returns a solid circle image whose color has those red, green, and blue values. (You can pick a reasonable constant size for this circle.)

OPTIONAL: if you'd like, you can also design this to expect a transparency value, also.

4 part a

Write an appropriate **signature** comment for this function.

4 part b

Write an appropriate **purpose statement** comment for this function.

4 part c

Write an appropriate **function header** for this function (putting `...)` for its body for now).

4 part d

Write at least 2 specific tests/`check-expect` or `check-within` expressions for this function.

4 part e

Only NOW should you replace the `...` in the function's body with an appropriate expression to complete this function.

Run and test your function until you are satisfied that it passes its tests and works correctly.

Finally, include at least 2 example calls of your function (such that you will see their results) after your function definition.

Problem 5

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

```
;
;  Problem 5
;
```

5 part a

Remember that `place-image` expects a scene expression for its 4th argument. It can be convenient to use a named constant for this scene, especially if you will be using it in a number of `place-image` expressions and if it might itself include several unchanging elements.

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 `WIDTH`, a scene width of your choice (except it should not be bigger than 1300

pixels).

- define a named constant `HEIGHT`, a scene height of your choice (except it should not be bigger than 400 pixels).
- define a named constant `BACKDROP`, an unchanging, constant scene that will serve as a backdrop scene in some future problems.
  - its size should be `WIDTH` by `HEIGHT` pixels
  - it should include at least one visible unchanging image of your choice (and you may certainly include additional visible unchanging images if you wish).
- (you may certainly include additional named constants, also, if you wish!)

5 part b

Consider a "world" of type number -- starting from a given initial number, in which that number changes as a ticker ticks, and in which the number determines what is shown where (or how) in a scene.

For this world, `big-bang`'s `to-draw` clause will need a scene-drawing function that expects a number, and produces a scene based on that number. Don't get too far ahead of yourself, here! Just decide what image(s) is/are going to be ADDED to your `BACKDROP` scene from **5 part a** based on a number value.

Do something that is at least slightly different than the posted class examples, and at least slightly different than what you did for the Week 3 Lab Exercise.

(For example, an image could be placed in the `BACKDROP` whose size depends on that number -- and/or an image's x-coordinate within the `BACKDROP` could be determined by that number -- and/or an image's y-coordinate within the `BACKDROP` could be determined by that number -- and/or an image's color could be determined by that number -- and/or some combination, letting it determine **both** the x- and y-coordinate, for example.)

Write an appropriate **signature** comment for this function.

5 part c

Write an appropriate **purpose statement** comment for this function.

5 part d

Write an appropriate **function header** for this function (putting `...)` for its body for now).

5 part e

Write at least two specific examples/tests/`check-expect` expressions for this function, placed either before or after your not-yet-completed function definition, whichever you prefer.

5 part f

Finally, replace the `...` in this function's body with an appropriate expression to complete it. For full credit, make sure that you use your `BACKDROP` scene from **5 part a** appropriately within this expression.

Run and test your function until you are satisfied that it passes its tests and works correctly.

Also include at least 2 example calls of your function (such that you will see their results) after your function definition.

5 part g

What change do you want to happen to your world number as the world ticker ticks? Will it increase by 1? decrease by 1? Increase and/or decrease by varying amounts?

Decide how your world number will change on each ticker tick. If your change can be done by a built-in function such as `add1` or `sub1`, that is fine. Otherwise, using the design recipe, develop this function, that expects a number and produces a number.

- (if you develop your own function here, REMEMBER to do all the steps you have been doing for the other functions in this homework -- first write its signature, then its purpose statement, then its function header, then its tests/`check-expect` expressions, then replace its `...` body with an appropriate expression)

Then write a `big-bang` expression consisting of an initial number, and a `to-draw` clause with the name of your scene-drawing function from **5 parts b-f**, and an `on-tick` clause with the name of your (new or existing) number-changing function -- something like:

`(big-bang` *initial-number-you-choose*

`(to-draw` *name-of-your-scene-drawing-function*`)`

`(on-tick` *name-of-your-number-changing-function*`))`

(By the way, you can also change the speed of the ticker in this `on-tick` clause if you would like to -- give it an optional 2nd expression, a number that is a `ticker-rate-expression`, and the ticker will tick every `ticker-rate-expression` seconds.)

Now you should see something changing in your scene, and now you are done with this problem.