CS 111 - Homework 2

Deadline

11:59 pm on **Friday, September 12, 2025**

Purpose

To practice reading and thinking about "translating" algebraic expressions into Racket syntax, to practice defining and using named constants and **`check-expect`** expressions, and to practice using the design recipe to define and test functions.

How to submit

You complete **Problems 1, 2, and 3** on the course Canvas site (short-answer questions on "translating" algebraic expressions into Racket, named constants, and on the design recipe), 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.

Each time you would like to submit your work so far:

- Save your current DrRacket Definitions window contents in a file with the name **`111hw2.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!
  - (Canvas DOES add your name and some numbers to your submitted file on the graders' end, including numbers on the end of the submitted file name if you submit more than once -- that's fine! Just please start out, on your end, with the file name **`111hw2.rkt`**.)
- Open https://canvas.humboldt.edu in a web browser, and click on the icon for the CS 111 course (or otherwise connect to the course Canvas site)
  - Click **Modules** on the left-hand-side of the page
  - Scroll down to the **Homeworks** section
  - Choose the **rest of Homework 2** link
  - Follow the Canvas instructions for uploading your Racket file **`111hw2.rkt`**.
- NOTE: If you have trouble submitting, please let me know.

Important notes - 10 points

- 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.
- Remember: A **signature** in Racket is written as a comment, and it includes the name of the function, the **Racket data types** of expressions it expects in the order they're to be expected, and the **Racket data type** of the expression it returns. This should be written as discussed in class. For example,

```
; signature: triple: number -> number

; signature: say-hi: string -> string

; signature: rect-area: number number -> number
```

- Remember: a **purpose statement** in Racket is written as 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 number, and returns triple that number

; purpose: expects a name, and returns a string containing
;     "Hi, " followed by their name

; purpose: expects a rectangle's length and width, and returns the
;     area of that rectangle
```

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

```
(define WIDTH 300)
```

- It is ANOTHER **course style standard** that parameters and function names are to be descriptive and written in all-lowercase -- for example,

```
(define (triple num)
    ...)
```

  - Also, do not use an *exact* data type name as a parameter name -- that is, parameter name `num`, above, is descriptive enough for function `triple`, but using a parameter name of `number` would violate class coding standards.

- Be sure to use the file name specified above (and note that Canvas does add your name and numbers to your file's name, and that's OK!).
- To make your BSL Racket code easier for the graders to grade, include the string expressions specified in each problem along with your answers for that problem.
- To make your BSL Racket code more readable, **put a blank line before *and* after:**
  - **each Racket comment or "block" of comments**
  - **each Racket multi-line expression**
  - Also, IF you would like, you can add an additional comment "border" before and/or after Racket comments.

    For example, something like:

    ```
    ;========
    ```

    or

    ```
    ;--------
    ```

- To make your BSL Racket code more readable, when you write a long compound expression:
  - **indent** the continued part to make it clearer it is still part of a long expression
  - **line-up** its arguments
- **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 - 5 points

Problem 1 is correctly answering the "HW 2 - Problem 1 - Short-answer questions on 'translating' algebraic expressions into Racket" on the course Canvas site.

Problem 2 - 5 points

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

Problem 3 - 8 points

Problem 3 is correctly answering the "HW 2 - Problem 3 - Short-answer questions on the design recipe" on the course Canvas site.

Problem 4 - practice defining named constants - 8 points

NOTE: you are NOT writing any new functions in this problem -- you will be defining several **named constants**.

Complete the remainder of Homework 2's problems in a file named **`111hw2.rkt`** (that is, save your DrRacket Definitions window to a file named **`111hw2.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 module by putting this expression at the beginning of your Definitions window:

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

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 2
; last modified: 2025-09-08
```

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

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

...also followed by another blank line.

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

(This is adapted from Homework 1 of J. Marshall's "The Way of the Program" course at Sarah Lawrence College.)

The Tasty-Waking coffee roasters sells whole-bean coffee at **$9.99** per pound plus the cost of shipping. Each order ships for **$1.29** per pound plus **$3.99** fixed cost for overhead.

The description above includes three constant values. Define three appropriate, descriptive named constants for these values. (This is so, when prices change, all you need to do is change the constant definitions.)

Then write one or more expressions (simple or compound, your choice!) such that you **use** each of your three new named constants at least once, so you can check that they have been successfully defined.

Problem 5 - completing your quilt-square - 12 points

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

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

...followed by another blank line.

Yet again, you are not writing new functions for this problem, either.

Remember the image you created for Homework 1, Problem 7? This will be called your quilt-square image in this problem.

5 part a

After a blank line, type the string expression:

`"--- 5 part a ---"`

...followed by another blank line.

FIRST: Copy its compound expression(s) here, **OR** those for a NEW quilt-square image if you prefer, as long as it meets the Homework 1 - Problem 7 requirements:

- the final resulting image somehow should be the result of **at least THREE different** image or image-related operations from the 2htdp/image module. (More is fine!)
- the final resulting image should be an expression of type image that is **precisely** 200 pixels wide and 200 pixels high
- IF you would like to use `define` to give names to some expressions to make your task easier, that is allowed and encouraged (but not required)
- As long as you meet the above requirements, your quilt-square image may be as simple or as complex as you would like.

THEN, after your quilt-square image expression(s):

- write a Racket **`define`** expression to define a **named constant** ...
  - ...whose **name** is an all-uppercase version of *your* name (or some part of your name, your choice) and
  - ...whose **value** is your *entire finished* quilt-square image.
- write the **simple expression** consisting of your newly-defined named constant for your quilt-square image -- when you now click Run, you'll see that the value of this named constant is your quilt-square image.

5 part b

After a blank line, type the string expression:

`"--- 5 part b ---"`

...followed by another blank line.

You're now going to **test** that your quilt-square image from **Problem 5 part a** is exactly 200 pixels wide and 200 pixels high.

Recall that operation **`image-width`** expects an image and returns its width in pixels, and that operation **`image-height`** expects an image and returns its height in pixels.

Write TWO **`check-expect`** expressions:

- First, write a **`check-expect`** expression,

  whose FIRST argument is an expression using `image-`**`width`** and the named constant for your finished quilt-square image from **Problem 5 part a**, and whose SECOND argument is 200.

- SO, this test will **pass** if your named constant quilt-square image is exactly 200 pixels wide.

- Second, write a **`check-expect`** expression,

  whose FIRST argument is an expression using `image-`**`height`** and the named constant for your finished quilt-square image from **Problem 5 part a**,

  and whose SECOND argument is 200.

  - SO, this test will **pass** if your named constant quilt-square image is exactly 200 pixels in height.

Run these to test your quilt-square image!

If your quilt-square image doesn't pass both tests, then you need to revisit **Problem 5 part a** and alter your quilt-square image expression(s) so that it passes the tests.

Problem 6 - function `avg-trio` - 10 points

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

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

...followed by another blank line.

NOW, you are defining a new function!

You are working in a project where, frequently, sets of three measurements are to be averaged.

So: Assume you would like a function to average each such trio of measurements.

Follow the design recipe steps to write a function **`avg-trio`**. (I'm specifying the function name to make it easier to test when the assignments are graded.)

6 part a

Think about these questions:

- **How many** and **what data type** of expression(s) should `avg-trio` expect, to be able to return such an average?
- What **data type** should `avg-trio` return?

Now write an appropriate **signature** comment for `avg-trio`.

6 part b

Write an appropriate **purpose statement** comment for `avg-trio`.

6 part c

Write a **function header** for `avg-trio`, giving good descriptive names for its parameters, and for now put `...` for its function body.

6 part d

Write at least two **`check-expect`** (or **`check-within`**) expressions with two example expressions USING/TESTING your function-in-progress `avg-trio`, with two DIFFERENT trios of measurements.

Interestingly, you may actually place these either before or after your not-yet-completed function definition, whichever you prefer.

(**NOTE:** when **fractional** values are involved, `check-expect` sometimes has problems, because exact equality of these values can be hard to test! Feel free to use **`check-within`** here instead of `check-expect` if you'd like; it has a third argument, the maximum difference allowed between the actual and expected values, and considers the test to be passed if the difference between the first and second arguments is *within* that difference.)

Also, **AFTER** your not-yet-completed function definition, write **at least one example call of `avg-trio`**, so you can **see** its actual result in the Interactions window when this is run.

6 part e

Consider: how did you figure out the expected result for each of your tests? How could you do that in Racket? Where do your parameters fit in that?

Go back UP to 6 part c, and NOW **replace** the `...` in `avg-trio`'s function body with an expression, using the parameters, that will give the result desired.

Run and test your function `avg-trio`, and debug as needed, until you are satisfied that it passes its tests and works correctly.

Problem 7 - function `tri-area` - 10 points

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

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

...followed by another blank line.

Consider: given a triangle's base and height, how can you figure out that triangle's area?

Follow the design recipe steps to write a function **`tri-area`**. (I'm specifying the function name to make it easier to test when the assignments are graded.)

7 part a

Think about these questions:

- **How many** and **what data type** of expression(s) should `tri-area` expect, to be able to return such an area?
- What **data type** should `tri-area` return?

Now write an appropriate **signature** comment for `tri-area`.

7 part b

Write an appropriate **purpose statement** comment for `tri-area`.

7 part c

Write a **function header** for `tri-area`, giving good descriptive names for its parameters, and for now put `...` for its function body.

7 part d

Write at least 2 specific tests/**`check-expect`** or **`check-within`** expressions for `tri-area`.

(Since this function can result in fractional values, feel free to use `check-within` here instead of `check-expect` if you'd like.)

Also, **AFTER** your not-yet-completed function definition, write **at least one example call of `tri-area`**, so you can **see** its actual result in the Interactions window when this is run.

7 part e

Consider: how did you figure out the expected result for each of your tests? How could you do that in Racket? Where do your parameters fit in that?

Go back UP to 7 part c, and NOW **replace** the `...` in `tri-area`'s function body with an expression, using the parameters, that will give the result desired.

Run and test your function `tri-area`, and debug as needed, until you are satisfied that it passes its tests and works correctly.

Problem 8 - function `ask-how-doing` - 10 points

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

`"=== Problem 8 ==="`

...followed by another blank line.

Consider a function that **asks** a given person how they are doing. Given a person's name, it returns
`How are you doing,` ...that person's name... `?`

(**Optionally**, you may **change** the exact wording of the question, as long as it **includes** the person's name and **ends** with **at least one** question mark.)

You are now going to follow the design recipe steps to write this function **`ask-how-doing`**.

8 part a

Think about these questions:

- **How many** and **what data type** of expression(s) should `ask-how-doing` expect, to be able to return such a question?
- What **data type** should `ask-how-doing` return?

Now write an appropriate **signature** comment for `ask-how-doing`.

8 part b

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

8 part c

Write a **function header** for this function, giving a good descriptive name for its parameter, and for now put `...` for its function body.

8 part d

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

Also, **AFTER** your not-yet-completed function definition, write **at least one example call of this function**, so you can **see** its actual result in the Interactions window when this is run.

8 part e

Consider: how did you figure out the expected result for each of your tests? How could you do that in Racket? Where do your parameters fit in that?

Go back UP to 8 part c, and NOW **replace** the `...` in `ask-how-doing`'s function body with an expression, using the parameters, that will give the result desired.

Run and test your function `ask-how-doing`, and debug as needed, until you are satisfied that it passes its tests and works correctly.

Problem 9 - a function that returns an image - 10 points

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

`"=== Problem 9 ==="`

...followed by another blank line.

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

simpler option 9-1 - make-captioned-img

Suppose someone decides it would be useful to be able to take a desired image and desired caption wording, and create a new image that includes the desired image with the desired caption wording centered beneath it (as PART of the resulting image), with the desired caption wording shown in **30-pixel-high** black letters.

You are now going to follow the design recipe steps to write this function **`make-captioned-img`**.

(HINT: The **`text`** function expects a string (as one of its arguments) and returns an image version of that string.)

more-open option 9-2 - simple-humboldt-meme

Suppose someone decides it would be useful to be able to take a desired image and desired meme wording, and create a simple meme image from those featuring, on a dark green (or, ah, `"darkgreen"`) rectangle, the desired image above the desired meme wording, with the desired wording shown in **30-pixel-high** white letters.

You are now going to follow the design recipe steps to write this function **`simple-humboldt-meme`**.

(Note 1: I am expecting that you will choose a constant width and constant height for the rectangle in this case. But -- as an additional optional challenge -- computing a pleasing width and height for the rectangle based on the given image and the given text is an interesting challenge, although NOT required...!)

(Note 2: I found `overlay/align`, `string-upcase`, and `text/font` to be not-strictly-necessary-but-pleasing in my, um, third version of this function.)

9 part a

Think about these questions:

- **How many** and **what data type** of expression(s) should your function expect, to be able to return such an image?
- What **data type** should your function return?

Now write an appropriate **signature** comment for your function.

9 part b

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

9 part c

Write a **function header** for this function, giving good descriptive names for its parameters, and for now put `...` for its function body.

9 part d

Write at least 2 specific tests/`check-expect` expressions for your function.

Also, **AFTER** your not-yet-completed function definition, write **at least one example call of your function**, so you can **see** its actual result in the Interactions window when this is run.

9 part e

Consider: how did you figure out the expected result for each of your tests? How could you do that in Racket? Where do your parameters fit in that?

Go back UP to 9 part c, and NOW **replace** the `...` in your function's body with an expression, using the parameters, that will give the result desired.

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

Problem 10 - function `order-total` - 12 points

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

`"=== Problem 10 ==="`

...followed by another blank line.

(This is adapted from Homework 1 of J. Marshall's "The Way of the Program" course at Sarah Lawrence College.)

Recall, from Problem 4 part a: The Tasty-Waking coffee roasters sells whole-bean coffee at **$9.99** per pound plus the cost of shipping. Each order ships for **$1.29** per pound plus **$3.99** fixed cost for overhead.

(For example, if someone buys 1 pound of whole-bean coffee, the total price is $15.27. If someone buys 2 pounds of whole-bean coffee, the total price is $26.55. If someone buys 5 pounds, the total is $60.39.)

Remember: you defined a named constants for important constants in the world of Tasty-Waking coffee roasters in Problem 4 part a.

10 part a

Consider a function named **`order-total`** whose purpose is to calculate the total cost for a coffee order.

Think about these questions:

- **How many** and **what data type** of expression(s) should `order-total` expect **from the user**, to be able to return such a total cost?
- What **data type** should `order-total` return?

Now write an appropriate **signature** comment for `order-total`.

HINT: **don't** include named constants in the signature! The three named constants you defined in Problem 4 part b do NOT need to be arguments to this function! Consider – a customer doesn't need to tell you the prices of your product and shipping costs; you already know that because you work there and are already aware of those prices. The value(s) the function should expect is/are the value(s) a customer *does* need to give when placing an order. So what *does* a customer need to tell the store when ordering coffee beans?

10 part b

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

10 part c

Write a **function header** for this function, giving a good descriptive name for its parameter(s), and for now put `...` for its function body.

(**HINT**: named constants do *not* belong in a function header, either – your function header should have parameter identifiers *only* for what the signature and purpose say is expected by the function.)

10 part d

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

(Since this function can result in fractional values, feel free to use `check-within` here instead of `check-expect` if you'd like.)

Also, **AFTER** your not-yet-completed function definition, write **at least one example call of your function**, so you can **see** its actual result in the Interactions window when this is run.

10 part e

Consider: how did you figure out the expected result for each of your tests? How could you do that in Racket? Where do your parameter(s) fit in that? For this function (and for full credit), where do Problem 3 part b's named constants fit in that?

Go back UP to 10 part c, and NOW **replace** the `...` in your function's body with an expression, using the parameter(s) and Problem 4 part b's named constants, that will give the result desired.

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