CS 111 - Homework 4

Deadline

11:59 pm on **Friday, September 26, 2025**

Purpose

To provide practice with:

- reading **`cond`** expressions
- thinking about the **number of tests needed** for functions involving different kinds of data
- considering how to write **`boolean`** expressions in Racket
- using the design recipe to design and write functions, including functions involving a **`cond`**/conditional branching expression and functions that use functions you have written earlier.

How to submit

You complete **Problems 1, 2, and 3** on the course Canvas site (short-answer questions on reading a **`cond`** expression, on the number of tests needed for functions, and on writing **`boolean`** expressions in Racket), so that you can see if you are on the right track.

Then, you will submit your work for **Problems 4 onward**, in your file **`111hw4.rkt`**, on the course Canvas site.

Submit your **`111hw4.rkt`** file-in-progress with your work on Problems 4 onward early and often!

Important notes - 8 points

- Please note that only *SOME*, **not** all, of this homework's functions contain a **`cond`** expression.
- Now that we have functions involving branching, NOTE that you frequently need **MORE than two** tests/`check-` expressions!
  - How many `check-` expressions should you have? The basic rules of thumb are:

    * for functions expecting data that can be grouped into "cases" or categories, you need at least one test/`check-` expression for each "case" or category of data that may occur -- but sometimes you need more:

    * for functions expecting **interval**-style data, you need at least one test/`check-` expression for each "case" or category of data that may occur, **AS WELL AS** one for each **"boundary"** between intervals, and you can always add more if you'd like!

    * for functions expecting **enumeration**-style data, you need at least one test/`check-` expression for each value within the enumeration, and **if** the function specifies a desired action for a value that is **not** part of that enumeration, you should include a test for **THAT** case as well.

    * for functions expecting data for which there is only one category, you should have **at least two** tests/`check-` expressions, for more-robust error-checking of your function.
  - You may include as many additional calls or tests of your function as you would like after its definition (and indeed, sometimes a problem will require that you do that).
- 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.

- **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 or a function from one of your previou assignments 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.

Problem 1 - 10 points

Problem 1 is correctly answering the "HW 4 - Problem 1 - Short-answer questions on reading **`cond`** expressions" on the course Canvas site.

Problem 2 - 5 points

Problem 2 is correctly answering the "HW 4 - Problem 2 - Short-answer questions on **number of tests needed** for functions" on the course Canvas site.

Problem 3 - 4 points

Problem 2 is correctly answering the "HW 4 - Problem 3 - Short-answer questions on translating **`boolean`** expressions into Racket" on the course Canvas site.

Homework File Setup for Problems 4-10

Complete the remainder of Homework 4's problems in a file named **`111hw4.rkt`** (that is, save your DrRacket Definitions window to a file named **`111hw4.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 4
```

`; last modified: 2025-09-22`

Problem 4 - function `get-freq-disc` - 13 points

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

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

...also followed by another blank line.

A store gives discounts to frequent shoppers based on their past level of purchases; they are either **"bronze"** level, **"silver"** level, or **"gold"** level. Bronze level frequent shoppers receive a **10%** discount, silver level frequent shoppers receive a **15%** discount, and gold level frequent shoppers receive a **20%** discount. All other shoppers receive **no** discount.

4 part a

After a blank line, type the string expression:

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

...followed by another blank line.

For this part, define some **named constants** for the *numeric* values of the discounts frequent shoppers can receive.

Define appropriate, descriptive **named constants** for the *numeric* **values** of the bronze level frequent shoppers' discount amount, the silver level frequent shoppers' discount amount, and the gold level frequent shoppers' discount amount.

Since Racket does not recognize percents written with %, use decimal values (for example, use **`0.15`** for 15%).

4 part b

After a blank line, type the string expression:

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

...followed by another blank line.

Using the design recipe, design a function **`get-freq-disc`** that expects a string representing a level of frequent shopper, and returns the appropriate discount for that level written as a decimal fraction. It should return a discount of **`0`** if the string is not one of those noted above.

(For example, `(get-freq-disc "silver")` should return `0.15`)

For this function, choose an appropriate number of **`check-`** expressions that test **all** the branches in your **`cond`** expression.

Hints:

- This function involves **enumeration** data, a list of distinct shopper levels.
- Make sure you appropriately use your named constants from Problem 4 part a!
- Remember that the function **`string=?`** is used to compare two strings to see if they are equal.

Optional additions:

- You are required to use the frequent shopper levels and discounts given, BUT you may add *additional* frequent shopper levels and discounts if you would like.

- (If you do so, make sure you include these additional levels in this function's purpose statement, and include additional test(s) for those additional frequent-shopper levels.)

Problem 5 - function `total-incl-disc` - 10 points

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

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

...followed by another blank line.

One important theme in this course is writing functions that work **together** to solve a problem.

Consider the scene function we created in-class during in Week 3 - Lecture 2, and how its function body called the image function we created in-class during Week 2 - Lecture 2 (and pasted into the Week 3 Lecture 2 `.rkt` file).

Now consider your function `get-freq-disc` from Problem 4. You can call this function in *another* function to make that other function's task easier.

Using the design recipe, design a function **`total-incl-disc`** that expects a string representing the level of frequent shopper *and* the total of a purchase *before* discount, and returns the calculated discounted total for that purchase after applying the appropriate discount, as provided by the `get-freq-disc` function.

(For example, `(total-incl-disc "silver" 10)` should return `8.5`)

For full credit, your solution **must** appropriately **call `get-freq-disc`** in its function body expression.

Hints:

- you will **NOT** need a **`cond`** expression inside the **`total-incl-disc`** function, thanks to **`get-freq-disc`**.
- In your tests/**`check-`** expressions, be sure to **vary** the example purchase totals as well as the example frequent shopper levels.
- If you need it, remember that **`check-within`** can sometimes be helpful for testing a function that returns a number with a fractional part.

Problem 6 - function `sugg-outerwear` - 10 points

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

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

...followed by another blank line.

It can be convenient for people to write intervals of numbers using something called **interval notation** (also discussed in the **course text** in **Section 4.4**) -- you give the first number and then the last number of the interval,

- putting an opening **SQUARE bracket**, [, before the first number if it is considered **IN** the interval,

  and putting an opening **PARENTHESIS**, (, if it is **NOT**, and
- putting a closing **SQUARE bracket**,], after the last number if it is considered **IN** the interval,

  and putting a closing **PARENTHESIS**,), if it is **NOT**.

SO,

- **`[0, 100]`** means all of the numbers from 0 to 100, INCLUDING 0 and INCLUDING 100

- **`(5, 8)`** means all of the numbers between 5 to 8, NOT including 5 and NOT including 8
- **`[0, 10)`** means all of the numbers from 0 up to but NOT including 10, INCLUDING 0 and NOT including 10
- **`(90, 100]`** means all of the number greater than 90 up to 100, NOT including 90 and INCLUDING 100

This is convenient, because it is a compact and (hopefully!) clearer way to say whether the end-points of an interval are considered part of that interval or not! **Note that this is a popular math notation, NOT Racket syntax!**

Suppose that a client wants a function that will recommend what outerwear to wear on a given day, given the predicted high temperature in Fahrenheit degrees, based on the following:

```
<= 32 - "down jacket"
(32, 48] - "sweater"
(48, 65] - "sweatshirt"
> 65 - "no outerwear today"
```

Using the design recipe, design a function **`sugg-outerwear`** that expects the predicted high temperature in Fahrenheit for the day, and returns the recommended outerwear for that day. For full credit, make sure that you include at least the minimum required tests for this data.

Hints:

- This function involves **interval** data, ranges of temperatures for each kind of outerwear. So, remember to include **boundary** cases in its set of tests/**`check-`** expects!
- Interval notation is for *people*, to make it more convenient for people to communicate interval information. It is **not** Racket syntax. (That is, Racket will not understand something like `(48, 65]`.)

  You have to translate those intervals into appropriate **`boolean`** expressions that Racket can understand.

Problem 7 - function `workout-hrs` - 10 points

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

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

...followed by another blank line.

(Adapted from Keith Cooper's section of Rice University's COMP 210, Spring 2002)

Conditionals (and Pizza Economics)

This problem considers an important consequence of increased pizza consumption –the need for additional exercise.

Using the design recipe, design a function **`workout-hrs`** that determines the number of hours of exercise required to counter the excess fat from eating pizza. **`workout-hrs`** expects a number that represents daily pizza consumption, in slices, and returns a number, in hours, that represents the amount of exercise time that you need.

For a daily intake of :	You need to work out for :
`0` slices	1/2 hour
`(0, 3]` slices	1 hour
`> 3` slices	1 hour + (1/2 hour per slice above 3)

For example, for 5 slices, the workout hours would be 2 hours, because that's 1 hour plus 1/2 hour plus 1/2 hour (because it is 1/2 hour more for each slice above 3).

For this function, you need **at least 4** well-chosen, appropriate specific tests/**`check-`** expressions.

Hints:

- Writing additional example tests can help in working out what you want for the case of more more-than-three pizza slices.
- Imagining trying to explain to someone - real or fictional - how you are figuring out the desired amount of workout hours for each of those tests can also be helpful.

Problem 8 - function `change-happiness` - 10 points

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

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

...followed by another blank line.

The remaining problems were adapted from a problem suggested by James Logan Mayfield on the `plt-edu` mailing list, a mailing list for Racket enthusiasts, (and are similar to some of the suggested exercises in Section I, Chapter 3 of the course text, but **NOT** exactly the same!).

Imagine that there is a virtual/computer pet. Imagine, also, that this virtual pet can have their current level of happiness measured by a number. (Think of it as "happiness points", like hit points or experience points in a role-playing game).

A virtual pet's companion can change their pet's happiness by various actions --

- FEEDING them makes the virtual pet happier, and **increases** their happiness by 10.
- SKRITCHING them makes them happier, and **increases** their happiness by 5.
- PLAYING with them makes them happier, and **increases** their happiness by 3.
- TEASING them makes them LESS happy, and **decreases** their happiness by 2.

NOW, say that:

- FEEDING is represented by the string **`"f"`**
- SKRITCHING is represented by the string **`"s"`**
- PLAYING is represented by the string **`"p"`**
- TEASING is represented by the string **`"t"`**

Using the design recipe, design a function **`change-happiness`** that expects a virtual pet's current happiness level and a string representing a companion's interaction with that virtual pet, and returns the resulting happiness level for that pet following that interaction. It should return the virtual pet's happiness level **unchanged** if given an interaction string **other** than **`"f"`**, **`"s"`**, **`"p"`**, or **`"t"`**.

(For example, `(change-happiness 100 "f")` means the companion fed that pet, and so returns a new happiness level of `110`.)

Optional additions:

- You are required to use the pet actions and happiness changes given, BUT you may add *additional* actions that affect a pet's happiness level if you would like.
  - (If you do so, make sure you include these additional possible actions in this function's purpose

statement, and include additional test(s) for those additional actions.)

Problem 9 - named constant(s) for a virtual pet - 5 points

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

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

...followed by another blank line.

What does a virtual pet look like? **You** now get to decide this, and **define at least one named constant** declaring something related to your virtual pet's appearance.

- It can be **JUST one named constant** of type **`image`** whose value is what your virtual pet looks like.
- It can **also be more intricate** IF you would like -- for example,
  - a collection of virtual pet images, happier and less happy
  - one or more named constants used by an optional function that creates your virtual pet image based on its current happiness level

Problem 10 - function `draw-pet-world` - 10 points

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

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

...followed by another blank line.

Now, using the design recipe, design a function **`draw-pet-world`** that expects a virtual pet's happiness level, and returns a **`scene`** that **displays at LEAST** the following:

- an **image** of your virtual pet
- the pet's **current happiness level**
  - That is, you need to somehow visibly include an image version of the given happiness along with your pet image.
  - Hint: how can you take a number, and get a string-version of that number?
  - Hint: how can you take a string, and get an image-version of that string?

NOTE: for this function **`draw-pet-world`**, it is **YOUR CHOICE** whether you need a **`cond`** expression or not.

- If you have a **single** image for your pet, displayed in the same way regardless of happiness level, then you likely will **not** need a **`cond`** expression.
- If you have **more** than one pet image, or want how your pet is shown in a scene to depend on its current happiness level, then a **`cond`** expression *might* be useful.

Problem 11 - demo your virtual pet world - 5 points

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

`"=== Problem 11 ==="`

...followed by another blank line.

Write a **`big-bang`** expression with at least:

- an **initial** pet happiness value
- a **`to-draw`** clause using your **`draw-pet-world`** function
- an **`on-key`** clause using your **`change-happiness`** function

That is, something like:

```
(big-bang  initial-pet-happiness-level-you-choose
           (to-draw draw-pet-world)
           (on-key change-happiness))
```

This **`big-bang`** expression should have the effect of displaying a scene of your pet and its current happiness in its World window, and when you type keystrokes that affect your pet's happiness, the scene displayed should show a different happiness level as a result!

Optional additions:

Optionally, you may add **additional `big-bang`** clauses (using the design recipe to design any new additional functions to use as the arguments to these clauses):

- an **`on-tick`** clause called with a function that changes your pet's happiness level each time it is called (could be as simple as `add1` or `sub1`, could be something else you design), and specifying the speed of `big-bang`'s ticker ticking (for example, you would put `1` to tick about once per second)
- a **`stop-when`** clause called with a function that returns **`#true`** if your pet's happiness reaches some critical level (could be as simple as `zero?`, could be something you design)
- or you can read about another **`big-bang`** clause and give it a try -- for example, you could read in the DrRacket documentation about the `on-mouse` clause in `big-bang`, and, using the design recipe, design a function that can be used with `on-mouse` to cause mouse actions of your choice to affect your virtual pet's happiness level.
  - Then, write a `big-bang` expression that includes an `on-mouse` clause using your resulting function.