# CS 111 - Homework 4

## Deadline

**11:59 pm** on **Friday, September 23, 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 `111hw4.rkt`
- transfer/copy that file to a directory on `nrs-labs.humboldt.edu` (preferably in a folder/directory named `111hw4`)
- 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.
- 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 `111hw4`, then you would go to that directory by saying:

    ```
    cd 111hw4
    ```

  - use the `ls` command to make sure your `111hw4.rkt` file is really there:

    ```
    ls
    ```

  - type the command:

    ```
    ~st10/111submit
    ```

    ...and when asked, enter a homework number of `4`

    ...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 `111hw4.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 `111hw4.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...)

## Purpose

To provide practice using the design recipe to design and write functions, including functions that use functions you have written earlier, and functions involving a `cond`/conditional branching expression (including functions involving intervals and enumerations).

## Important notes

- Please note that only SOME, **not** all, of this homework's functions contain a `cond` expression.
- 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: first 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.
- You are expected to follow the Design Recipe for all **functions** that you design/define. So, each function is expected to include:
  - a signature comment, including a nicely-descriptive name of the function, the **types** of expressions it expects, and the **type** of expression it produces. This should be written as discussed in class (and you can find examples in the posted in-class examples). For example,

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

  - a purpose statement comment, **describing** what the function expects and **describing** what it returns. For example,

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

  - [following the design recipe, you will be writing the function header next; note that you **don't** need to write it twice. Follow the function header with a body of `...` at this stage, and replace that `...` with its body **later**, at the appropriate step in the design recipe.]
  - `check-expect` (or `check-within`, or other `check-` operation) expressions expressing the specific examples that you write **BEFORE** writing your function body. (These may be **placed** before or after your actual function, but you are expected to **create** these **BEFORE** writing the function body. I'll have no way of knowing if you really write these in the correct order, but note that I won't answer questions about your function body without seeing your examples written as `check-` expressions first...) For example,

    ```
    (check-expect (rect-area 3 4)
                  12)
    (check-expect (rect-area 10 5)
                  50)
    ```

  - How many `check-` expressions should you have? Remember, the basic rules of thumb are:

    * you need a test/`check-` expression for each "case" or category of data that may occur, AS WELL AS one for each "boundary" if intervals are involved, and you can always add more if you'd like!

    * if there is **only one** category of data, you should have **at least two** tests/`check-` expressions, for more-robust error-checking of your function.

  - [and, of course, your function definition itself!]
  - You may include as many additional calls or tests of your function as you would like after its definition.
- 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 tests/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 4`,
- 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 4
; last modified: 2016-09-16
;
```

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

```
;
; Problem 1
;
```

A store gives discounts to frequent shoppers based on their past level of purchases; they are either "silver" level, "gold" level, or "platinum" level. Silver level frequent shoppers receive a 12% discount, gold level frequent shoppers receive a 25% discount, and platinum level frequent shoppers receive a 33% discount. All other shoppers receive no discount.

## *1 part a*

Write an appropriate data definition comment for `FreqShopperLevel`, in the same style as the data definition comment for `ArrowKey` used in class examples.

## *1 part b*

Define appropriate, descriptive **named constants** for the **values** of the silver level frequent shoppers' discount amount, the gold level frequent shoppers' discount amount, and the platinum level frequent shoppers' discount amount, expressed as decimal fractions. (Hint: the values of these named constants should be of type number!)

## *1 part c*

Using the design recipe, and remembering that the function `string=?` can be used to compare two strings for equality, design a function `discount-amt` that expects a string representing the level of frequent shopper and produces the appropriate discount for that level written as a decimal fraction. (It should produce a discount of 0 if the string is not one of those noted above.) Make sure you appropriately use your named constants from Problem 1 part b! For this function, you need at least 4 well-chosen, appropriate specific tests/`check-` expressions.

## Problem 2

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

```
;
;  Problem 2
;
```

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

Consider your function `discount-amt` from Problem 1. You can use this function in *another* function to make its task easier.

Using the design recipe, design a function `total-wi-discount` that expects the total of a purchase before discount and a string representing the level of frequent shopper, and produces the appropriate discounted total for that purchase. For full credit, your solution must appropriately use `discount-amt`.

(Hint: you should **NOT** need a `cond` expression inside the `total-wi-discount` function, thanks to `discount-amt`.)

## Problem 3

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

```
;
;  Problem 3
;
```

Assume 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 - "parka"
(32, 50) - "coat"
[50, 60) - "jacket"
>= 60 - "no outerwear today"
```

Using the design recipe, design a function `outerwear-choice` that expects the predicted high temperature in Fahrenheit, 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 (hint: including boundary cases!).

## Problem 4

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

```
;
;  Problem 4
;
```

(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, develop a function `workout-amt` that determines the number of hours of exercise required to counter the excess fat from eating pizza. `workout-amt` expects a number that represents daily pizza consumption, in slices, and produces 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 |
| 1 to 4 slices | 1 hour |
| >4 slices | 1 hour + (1/2 hour per slice above 4) |

For this function, you need at least 5 well-chosen, appropriate specific tests/`check-expects`.

## Problem 5

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

```
;
;  Problem 5
;
```

The remaining problems were adapted from a problem suggested by James Logan Mayfield on the plt-edu mailing list.

Consider a virtual pet world in which the world's type is number: in this case, the number representing the world represents that pet's happiness.

Find or build ONE OR MORE images to represent your virtual pet (you can create them from image operations, or paste them in, your choice). Define a named constant for each of these virtual pet's image(s).

(You can have just a single pet image, or you could have several pet images, representing more happy or less happy versions of your pet...)

How can you show your pet's current happiness? You need to somehow visibly include an image version of the current world number along with your pet image.

(For example: you could use `number->string` to convert the current number representing the world into a string, and then use the `text` operation to convert that string into an image of that text using a desired font size and color. Then you could use `above` to combine your desired pet image and that text image into a single image, representing your pet and its current happiness.)

Now -- using the design recipe, design a `draw-pet-scene` function that expects a world number, here representing your pet's current happiness, and produces a scene containing (the appropriate version of) your pet's image and a text image showing the value of your pet's current happiness.

(For this function, note that it is up to you whether you need a `cond` expression or not -- if you have a single image for your pet, you likely will not, but if you have more than one image, you likely will.)

## Problem 6

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

```
;
;  Problem 6
;
```

This little helper-function does not need a `cond` expression.

You know that `big-bang` needs a function name to give to its `on-tick` clause, to determine how the world should change at each ticker tick. For our virtual pet's world, its happiness should **decrease** with each ticker tick -- you get to choose how much (any value MORE THAN 1). Then, using the design recipe, design a function `sub-happiness` that expects a world number, which stands for your pet's current happiness,

and produces the new decreased happiness that should result at the next ticker tick.

## Problem 7

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

```
;
;  Problem 7
;
```

What can make your virtual pet happy, then?

Feeding it makes it happy! Say that typing the key f -- represented as the string `"f"` -- should increase its happiness by 10.

Petting it makes it happy, too -- say that typing the key p -- represented as the string `"p"` -- should increase its happiness by 5.

But teasing it makes it less happy -- say the typing the key t -- represented as the string `"t"` -- should decrease its happiness by 3.

(**IF** you would like, you may **ADD** additional keys that affect your pet's happiness.)

Typing any other key should result in the pet's happiness staying the same (not changing).

Using the design recipe, design a function `interact-wi-pet` that expects the pet's current happiness and a string (representing a keystroke), and produces the resulting new value of the pet's happiness that should result from that keystroke.

Then write a `big-bang` expression consisting of an initial pet happiness value, a `to-draw` clause with with `draw-pet-scene`, an `on-tick` clause with `sub-happiness`, and an `on-key` clause with `interact-wi-pet` -- something like:

```
(big-bang initial-pet-happiness-number-you-choose
          (to-draw draw-pet-scene)
          (on-tick sub-happiness 1)
          (on-key interact-wi-pet))
```

(Note that the `on-tick` clause above is being asked to call `sub-happiness` about once per second.)

Now you should see your pet's happiness decreasing as this runs, and that typing keys causes its happiness to increase or decrease or not change, depending on what you type -- when you have checked out that this works, then you are done with this problem.

## BONUS PROBLEM (up to 10 points)

Read in the DrRacket documentation about `big-bang`'s `on-mouse` clause, 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. Include a `big-bang` call that also includes an `on-mouse` clause with your resulting function.