This lab is designed to help you practice:

- defining structs
- writing programs which consume structs as arguments
- writing programs that consume structs and output structs
- working with NESTED structs
- writing interactive world programs

Please hand in all answers in a single file named "lab4.rkt". Do all work in pairs. **Include the names of both partners at the top of the file in a comment. You need only to submit one lab per pair.**

Review of define-struct

In class we showed that the single structure definition for the compound type snake:

```
(define-struct snake (name length color))
```

allows DrRacket to automatically define a constructor:

```
;; make-snake: String Number String -> Snake
;; (make-snake "fred" 21 "green") --> (make-snake "fred" 21 "green")
```

and (in this case) automatically defines three selectors:

```
;; snake-name: Snake -> String
;; snake-length: Snake -> Number
;; snake-color: Snake -> String
;;
;; (snake-name (make-snake "fred" 21 "green")) --> "fred"
;; (snake-length (make-snake "wilma" 19 "yellow")) --> 19
;; (snake-color (make-snake "betty" 14 "white")) --> "white"
```

and automatically defines one predicate:

```
;; snake?: ANYTHING -> boolean
;; (snake? "hello") -> false
;; (snake? (make-snake "betty" 14 "white")) --> true
```

The Design Recipe

The Design recipe (v2.0) states that whenever we need to create compound data, we should create a ***data definition***, ***examples***, and a ***template***. For example:

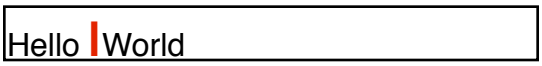
Question: *design a structure to represent CD's by artist, name, and price*

```
;; [Data Definition]
(define-struct cd (artist name price))
;; A CD is (make-cd String String Number)
;; interp of fields:
;; artist [String]: who created the music
;; name   [String]: title of the album,
;; price  [Number]: the sale price in US Dollars.
;;
;; [define-struct AUTOMATICALLY creates these functions]
;;   ...you do not have to write these out after Lab 3...
;; [CONSTRUCTOR]
;;   make-cd: String String Number --> Cd
;; [SELECTORS]
;;   cd-artist: Cd --> String
;;   cd-name  : Cd --> String
;;   cd-price : Cd --> Number
;; [PREDICATE]
;;    cd?: ANY --> Boolean
;;
;; [EXAMPLES: you can use these in your unit tests]
(define CD1 (make-cd "Game Theory" "Tinker to Evers to Chance" 19.95))
(define CD2 (make-cd "New Order" "Get Ready" 21.89))

;; [Function Template: functions that consume a CD]
#;
(define (cd-fun acd)
    (string-fn (cd-artist acd)) ; String
    (string-fn (cd-name acd))   ; String
    (number-fn (cd-price acd))  ; Number
 )
```

Problem 1: a Simple Text Editor Box

A common element of interactive programs is an input text editor, that is, a place to type text, edit it, and submit it for further computation. Common examples are the URL bar in your browser, or the Google or Bing search box. In this part we will develop the data definition and helper functions for a simple one-line text editor (and later, we'll use it in an interactive program). Consider this Real-World VIEW (as opposed to the MODEL we need to build computationally):



1. Develop a *data definition* for a STEB (Simple Text Editor Box). Remember that a data definition consists of the define-struct needed to model the definition and the interpretation of each field. Write the names and signatures of the constructor, selectors, and predicate in a comment (we won't require this after this lab). Provide three examples this time (normally we only require one). Hint: think carefully about how you want to represent a STEB. You need to "compound" together at least the text typed so far and the location of the cursor.
2. Develop a function `move-cursor-left` that consumes a STEB and produces a new STEB with the cursor one character to the left, unless the cursor is already at the start, in which case do nothing. Be sure, as always, to develop appropriate unit tests.
3. Develop `move-cursor-right` that moves the cursor one character to the right. Consider what the possible cases are here and develop appropriate unit tests.
4. Develop the function `delete-char` that deletes the character to the left of the cursor. Consider what to do if the last character is deleted. Consider the case where there is nothing to the left of the cursor. Consider what happens to the cursor location.
5. Develop the function `insert-char` that consumes a STEB and a Letter (single character), and inserts that letter to the left of the cursor. Again, consider the cases, developing tests for each. Again, consider if more than one element of your STEB compound structure should change, given your interpretation in your original data definition. Make sure to define a new data types that you need (e.g., Letter)
6. Write a function `steb->image` that consumes a STEB, and produces an image similar to the one above. *As always, follow the full Design Recipe or lose credit for the problem. We won't keep repeating this but expect you to know it implicitly.* Hint: You might find it easier to wish-list an auxiliary function that produces a picture of the string with the colored cursor (here, the vertical bar "I") inserted at the correct spot. You can then overlay the produced image onto the background in the appropriate location. Make sure to define appropriate constants.
7. Consider how key events effect the state of a STEB. Write a function `steb-handle-key` that consumes an STEB and a KeyEvent, and produces a new STEB.
   - Pressing the left arrow should move the STEB's cursor to the left.
   - Pressing the right arrow should move the STEB's cursor to the right.
   - Pressing delete should delete the character to the left of the cursor
   - Pressing a letter key should insert the character to the right of the cursor

8. At this point you should be able to interact with a STEB by copying and pasting the following code

```
(define (test-steb asteb)
  (big-bang asteb
            (on-draw steb->image)
            (on-key  steb-handle-key)
            (check-with ...))
```

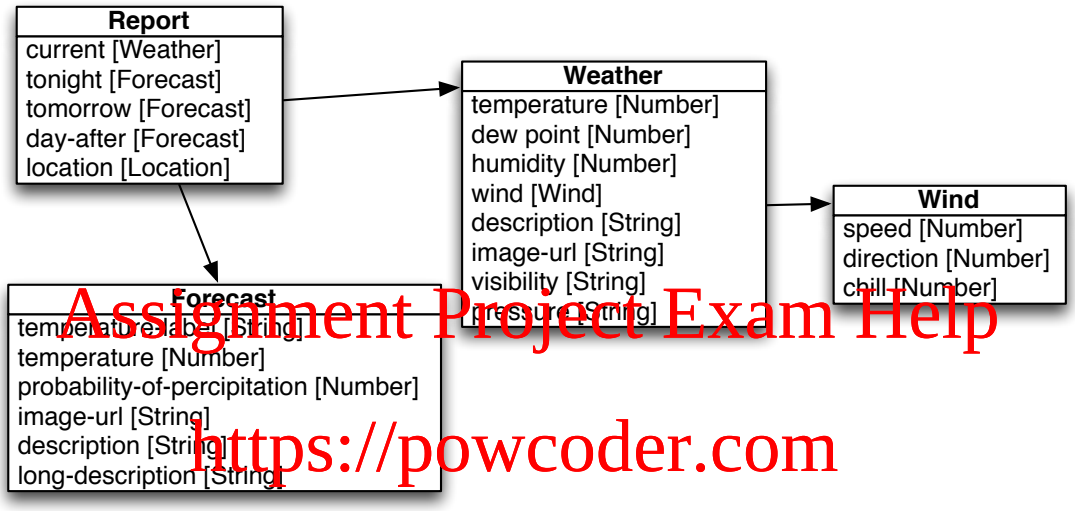
Here the world includes only the STEB, and nothing else. Your STEB is controlled by the keyboard, and not the mouse or the clock, so there are no mouse or clock handlers (you could add mouse support, of course...)

The `(check-with ...)` clause may be new. `Check-with` consumes the name of a predicate that consumes one argument of any type and returns true if and only if that argument is a proper world. Given that the world is a STEB, you should be able to provide such a function name...

Problem 2

Most real-world information requires fairly large structures to represent that information as data that can be used for computation. Let's consider weather data. A weather report for a city might consist of the current weather, and forecast information for the next evening, tomorrow, and the day after tomorrow. In particular, consider the data definitions for Report, Weather, Wind, and Forecast in Lab 3 rubric.

Sometimes programmers summarize structure data definitions with a picture like the following (Often called a UML diagram). This picture shows you that a Report struct contains a Weather struct and three Forecasts (ignoring Location at the moment), and the Weather struct contains a Wind struct. The other fields are atomic types we already know about.



For testing purposes, we have two versions of a live, real-time weather data service. One uses cached data for a few cities so that you can write tests, and the other gives access to the actual real-time data feed. We will assume you are using the canned version, and will turn in that version, but once your code is working feel free to use the live version of your weather app.

Attached to this lab are three files: weather-service-offline.rkt, weather-service-online.rkt, and cache.json (that contains the canned/cached data). Place these three files in the same directory as your lab3.rkt file, and place the line `(require "weather-service-offline.rkt")` at the top (replace with the online version when you want to play with the live service).

The interface provided is the function `get-weather`:

```
;; a LocationString is an address recognizable to Google's location service,
;;                  e.g. "Orlando, FL"
;; get-weather : LocationString -> [Report or String]
;; consumes: a location string
;; produces: a real-time (or cached) weather report for the given location or
;;           a string containing an error message
```

The cache includes reports for Newark, DE; Orlando, FL; Chicago, IL; Washington, DC; and Los Angeles, CA. For example:

```
(define NEWARK-REPORT   (get-weather "Newark, DE"))
(define NEWARK-CURRENT  (report-current NEWARK-REPORT))
(define NEWARK-TOMORROW (report-tomorrow NEWARK-REPORT))
```

We'll use this data source to write a small interactive application that can be used to help plan last-minute, skiing get-away weekends.

1. Develop a function `forecast->recommendation-image` that consumes a day-time Forecast and a String that is the time period of the Forecast (i.e., "Tomorrow", "Day After"). The function should produce an image that indicates whether the forecast is good for skiing weather. To be acceptable skiing weather, the temperate needs to be above 0 degrees and below 40 degrees. In addition, the chance of precipitation must be below 10%, unless the precipitation will be snow, in which case any level of participation is acceptable. The image should clearly indicate the time period of the forecast, whether the forecast is for good skiing weather, and, if the forecast is not favorable, a brief description of why. For example, for `NEWARK-TOMORROW`, your function could produce something like the following:

   Hint, write several helper functions to break this problem into more manageable chucks.

2. Develop a function `report->recommendation-image` that consumes a Report and produces an image that clearly indicates whether tomorrow is good for skiing and whether the day-after is good for skiing. Hint, combine the outputs from calling `forecast->recommendation-image` with the appropriate forecasts.

3. OK, finally we can complete the app. When designing models (worlds), we need to consider what elements will change due to events (e.g., clock, keyboard, mouse). Here, only the keyboard will effect our model; the weather reports do not change so quickly that we need to update it every 1/24 of a second. So there are TWO things stuck together in our world, a STEB and the current weather Report:


```
;; a WeatherWorld (WW) is a struct
(define-struct ww (steb report))
;; a WW is (make-ww STEB Report)
;; [Interpretation]
;;   steb   [STEB]  : a SimpleTextEditorBox holding the current location
;;   report [Report]: the current report for our location
;; CONSTRUCTOR
;;   make-ww : STEB Report --> WeatherWorld
;; SELECTORS
;;   ww-steb : WeatherWorld --> STEB
;;   ww-report: WeatherWorld --> Report
;; PREDICATE
;;   ww? : ANY --> Boolean
#;
(define (ww-fun aww)
  ... (steb-fun (ww-steb aww))     ; fn on a STEB
  ... (report-fun (ww-report aww)) ; fn on a Report
  )
```


   We can now develop a render function to draw the ***view***, and a key-handler to handle the key ***controller*** events, and wrap them into a main application big-bang function to create and update our ***model*** world.

   Develop `ww-render` that consumes a WeatherWorld and produces an Image containing both the summary image from render-weather and the image from render-steb (on top or bottom, your choice). Use your existing functions, this is a simple one!

4. Develop `ww-handle-key` that follows the signature for a big-bang key handler. Consider that we DO NOT want to copy all the work we did writing a key-handler for the STEB, but only want to indicate

when our app should update the weather report (because the user has entered another location, perhaps). We will choose the enter/return key as the only event that will make us update the WeatherWorld by calling `get-weather`.

5. Create a main function to run your app. Make sure to include a `check-with` clause like we did for Problem 1. When you think it is working, feel free to require the online version of the library to get the real-time weather (for US locations).