Lab 3

This lab will 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 graphical programs with complex, structured state

Relevant sections of text: *HtDP/2e Chapters 5 and 6.*

Remember:

- no "magic numbers"! Define constants and use the constant names instead.
- **follow the design pattern**: in particular, for each function definition, give a signature, purpose statement, unit tests, and finally the definition itself.
- **use precise data definitions**: this lab is focused on `structs`, which require very thorough documentation, reviewed below.

1. Review of Structure Definitions

The following example shows all the components of a complete structure definition. Do this every time you define a `struct`.

```
(define-struct entry [name phone email])
#|
An Entry is a (make-entry String String String)
interpretation
  String name : a person's name
  String phone : the person's phone number
  String email : person's email address
Constructor:
  make-entry: String String String -> Entry
Selectors:
  entry-name : Entry -> String
  entry-phone : Entry -> String
  entry-email : Entry -> String
Type predicate:
  entry? : Any -> Boolean
|#

;; Examples:
(define SIEGEL (make-entry "Siegel" "302-123-4566" "siegel@udel.edu"))
(define JOE (make-entry "Smith" "800-123-5683" "joe@joe.com"))
#|
Template for function consuming an Entry:
(define (entry-fun an-entry)
 ... (entry-name an-entry) ...
```

```
 ... (entry-phone an-entry) ...
 ... (entry-email an-entry) ...)
|#
```

2. Using structures for batch programming

Problem 1. Define a type *Date* for representing *dates*. This should be a structure with three components, in this order: day, month, and year. All three components are Natural Numbers: a day is in the range 1..31; a month is in 1..12; a year is any natural number. Make sure to give all components of the complete definition, including the documentation (see example above). You must do this every time you define a `struct`, even if we don't remind you.

Problem 2. Design a function `date->string` which converts a *Date* to a *String* of the form "*month*/*day*/*year*".

Problem 3. The State of Delaware's information system maintains a record for each taxpayer. Define a structured type *Person* for this purpose. It contains the following fields: first name, last name, birthday, city of residence, state of residence. For the birthday, use the *Date* type you defined in Problem 1.

Problem 4. Design a function `update-name` to change a person's name in the information system. This function should consume a *Person* and two *String*s (first name and last name) and return a *Person* which is identical to the given one except that the first and last name fields have been replaced with the new names.

Problem 5. Now we want three functions which are all similar. All three functions consume a *Person* and return a *Boolean*:

(1) `can-drink?` determines if the person can legally consume alcohol in Delaware, i.e., is the person at least 21 years of age?
(2) `can-vote?` determines if the person can vote in state and federal elections, i.e., is the person at least 18 years of age?
(3) `soc-sec-eligible?` determines if the person is eligible to receive social security benefits, which means s/he is at least 65 years old.

Since these functions are obviously very similar, you should define one or more *helper functions* first, and use those helpers in the definitions of the three main functions. This will prevent needless duplication of code, make the code easier to read, maintain, test, and understand, and make your solution shorter. You have to figure out what the appropriate helpers should be: what should they be named? what are their signatures? The helpers must be designed just like any other function. A hint is provided in the starter file.

3. Using structures for interactive programming: SpaceX

The company **SpaceX**, led by Elon Musk, is designing a re-useable Falcon rocket. After delivering its payload, the rocket should be able to land on a floating platform at sea. This turns out to be very difficult: `http://www.space.com/28295-spacex-rocket-landing-crash-photos-video.html`.

In the remaining problems, you will develop a game to aid in the design of the rocket. This game will show the rocket descending from the top of the scene. The rocket will naturally accelerate downwards due to the force of gravity. By firing its thrusters, it can more than counter gravity and

accelerate upwards. However, it has only a limited supply of fuel with which it can do this. There is also a horizontal component to the velocity, which is naturally constant (since there is no gravity in the horizontal direction). The horizontal velocity can be adjusted by small thrusters which do not consume significant fuel.

Images are provided in the starter file. You may use them, or find or make your own. As usual, creative graphics and particularly elegant solutions receive extra credit.

Problem 6. Define constants. Your solution will need *at least* the following constants (and probably more).

- images for the rocket, the fire beneath the rocket when it burns, the landing platform, some picture of an explosion
- dimensions of the overall scene which should have width 1000 and height 800
- constants for the positioning of the landing platform on the bottom of the scene: it should be in bottom middle
- constants defining the exact left and right borders of the safe landing zone, which should obviously be inside the platform; any landing outside of this zone results in failure (explosion)
- the acceleration due to gravity, which for this problem should be something like .01 pixels/tick^2. Note positive is *down*.
- the acceleration when the rocket is firing, which should be something like $-.05$ pixels/tick^2. Note that negative is *up*.
- the maximum safe speed for landing; any landing at a greater speed results in failure; approximately 1.5 pixels/tick
- the amount the horizontal (x-) velocity changes when a left or right arrow key is pressed, e.g., 0.5 pixels/tick
- `START-FUEL`, the amount of fuel at the beginning, e.g., 100 units
- `FUEL-PER-TICK`: the amount of fuel consumed in one clock tick, when the rocket is burning, e.g., 1.0 units

Most of these are given to you in the starter file, but there are two places you must fill in.

Problem 7. Define a structured type *RocketState* (the name of the struct should be `rocket-state`) which encompasses all aspects of the state of the game at any moment in time. It contains the following fields:

- `pos-x` : position of rocket in x-direction in pixels; 0 is the left edge of the scene; increases as you go to the right
- `pos-y` : position of rocket in y-direction in pixels; 0 is the top of the scene, increases as you go down
- `vel-x` : velocity of rocket in x-direction, in pixels per tick
- `vel-y` : velocity of rocket in y-direction, in pixels per tick
- `fire?` : a Boolean value telling you whether the rocket is currently firing (burning)
- `fuel` : amount of fuel remaining

Problem 8. Design helper function `update-vel-x` to update the `vel-x` component of a RocketState. This function consumes a RocketState and a number and produces a RocketState identical to the given one except that the `vel-x` component has been updated.

Problem 9. Design a helper function `update-fire` similar to above, but for the `fire?` component.

The rendering function for this program is complicated, so we will break it down into three major steps:

(1) `render-rocket` : draws the rocket, including possible fire underneath
(2) `render-data` : draws the data window with statistics in the upper right corner of the scene
(3) `render-final` : draws the final image, either "You Win!" or the picture of an explosion (you lose).

Each of these functions consumes a RocketState and an Image. The given Image should be an entire *scene*, i.e., the big empty rectangle, possibly with various things drawn on top of it. The function returns a new scene obtained by adding one more thing to the given scene.

We first focus on `render-rocket`. This function will require an auxiliary function which adds the fire underneath the rocket if the state indicates the rocket is firing.

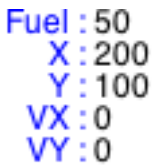
Problem 10. Design a function `render-rocket-fire` which consumes a RocketState and an Image (a scene) and returns an Image. If the state specifies that the rocket is *not* firing, then the scene returned should be the same one given. If the state specifies that the rocket *is* firing, then the scene returned is obtained by placing the fire image over the appropriate point (just underneath the point where the rocket should be):

Note that the purpose of this function is only to add the fire to the scene — not to add the rocket proper.

The starter file provides a good hint for this function.

Problem 11. Design a function `render-rocket` which consumes a RocketState and an Image and returns a new Image in which the rocket has been placed on the scene at the coordinates specified by the state, and, if the rocket is firing, the fire image appears underneath the rocket. Your function definition should use `render-rocket-fire`, which you defined in the previous problem.

Problem 12. Design a function `data-image` which consumes a RocketState and produces an Image. The image contains text and numbers which give the values of the remaining fuel, the x- and y-positions, and the x- and y-velocities. It should look something like this:



The values of x, y, vx, and vy should be multiplied by 10 and then rounded (i.e., these values will be presented in units of a tenth of a pixel). This will provide higher resolution information to the user. In designing this function, **define helper-functions as needed to avoid code duplication.** See the starter file for a hint.

Problem 13. Design a function `render-data` which consumes a RocketState and an Image and returns an Image in which the data image (defined above) is placed in the upper right hand corner of the scene.

Problem 14. Design a function `landed?` which consumes a RocketState and returns a Boolean. It tells you if the rocket has reached the ground.

Problem 15. Design a function `good-landing?` which consumes a RocketState in which the rocket has reached the ground, and tells you if that landing was successful. The landing is successful if the x-position is between the two acceptable boundaries and the y-velocity does not exceed the maximum safe speed.

Problem 16. Design a function `render-final` which consumes a RocketState and an Image and returns an Image as follows: if the rocket has not landed, the original scene is returned unchanged. If the rocket has landed: if the landing was safe, the words "You Win!" are placed on top of the scene in big letters in the middle of the scene; otherwise, a picture of an explosion is placed on top of the rocket.

Problem 17. Design a function `render` which consumes a RocketState and produces an Image. Using function composition, it applies the following functions, in order: `render-rocket`, `render-data`, and `render-final`. Note that in the composition, `render-rocket` will be the *innermost* function application, since it should be evaluated *first*.

That completes the rendering. Now for the tick- and key-handlers...

Problem 18. Design a function `handle-tick`, which consumes a RocketState and returns a RocketState. The state is updated as follows:

- the x-position is incremented by the x-velocity
- the y-position is incremented by the y-velocity — but don't let the rocket go below the ground
- the y-velocity is incremented by the acceleration due to gravity if the rocket is not firing, or by the acceleration due to firing if the rocket is firing
- if the fuel level has reached 0, the `fire?` field is set to `#false`
- if the rocket is firing, the fuel level is decremented by `FUEL-PER-TICK` — but don't let it go below 0.

Problem 19. Design a function `handle-key` which consumes a RocketState and a *Key* and returns a RocketState. The keys are used as follows:

- the space bar toggles the burning (rocket firing) on and off; of course, if the rocket is out of fuel this key should do nothing
- the right arrow increments the x-velocity of the rocket by some fixed small amount
- the left arrow decrements the x-velocity by the same fixed amount

- other keys have no effect

The final "main" function is provided for you. It calls `big-bang`. Note how the big-bang application wraps up the functions `render`, `handle-tick`, `handle-key`, and `landed?`. It also uses the type predicate `rocket-state?` to check that each state is actually a value of that structured type. (This will help you debug your program if you make a mistake.)

Play the game. Note that it should end with the rocket either safely landed on the platform and "You Win!" displayed, or with an explosion.