Comp160
Lab 10
Fall 2018

For this lab you'll look at a *toy problem* related to your final project. Toy problems simplify some real programming program down to its base components so that you can get a handle on the issues without the distraction of extra details. In this case we'll be looking at a space invaders like game where the player fires multiple missiles in an attempt to hit multiple UFOs. **Remember that *designing* a function means following the design recipe.** Help will not be given on function definitions unless you can prove that you've completed the three steps of the design recipe that proceed the definition[1]. As always, submit a stable set of code by the end of lab. We'll complete and review this assignment in class after break.

[1] i.e.

1. Signature, Purpose, and Header
2. Write examples as tests
3. Apply the template for your main input

Lab 10

A natural extension to our space invaders game is to allow the player to fire multiple missiles at multiple UFOs. One of the core tasks of such a program is to determine which UFOs have been hit and to remove those UFOs from the scene. In this lab you'll be explore the programming logic behind this problem in a simplified setting. Forget about things moving. Let's just worry about drawing things and removing the UFOs that have been hit. Rather than draw UFOs and missiles, let's just draw simple circles. UFOs have a radius of 15 and are red and missiles have a radius of 5 and are blue[2]. Both missiles and UFOS are represented by Posn structures. Our scene will be 900 pixels wide by 600 pixels high. The data definitions shown in Figure 1 are critical to our problem.

[2] or any two colors your choose

```
; A List-of-UFOs is one of the following:
; - '()
; - (cons (make-Posn x y) List-of-UFOs)
; interpretation: A list of zero or more UFOs where
; the posn indicates the UFO position from the upper left corner
; of the scene

; A List-of-Missiles is one of the following:
; - '()
; - (cons (make-Posn x y) List-of-Missiles)
; interpretation: A list of zero or more Missiles where
; the posn indicates the Missile position from the upper left
    corner
; of the scene
```

Figure 1: List Data Definitions for UFOs and Missiles

In a complete world program we'd need to define a structure to hold both of these lists, but we'll ignore that issue for now because our goal is to explore and understand the interaction between these two lists in games like our space invaders game. None the less, we can at least think about what kind of lists are required for different situations in our game.

1. Come up with example lists for the following situations. Define each list as a constant[3].

   (a) A scene with one UFO in the upper left part of the scene and one missile in the middle part of the scene.

   (b) A scene with three UFOs and no missiles.

   (c) A scene with no UFOs but two missiles.

[3] we'll use them for testing later

Let's start with drawing the scene so that we can actually visualize our different scenarios. The definition for *draw-scene* shown in figure 2 makes use a familiar design pattern. The helper functions *draw-missiles* and *draw-ufos* take their respective lists and an image and then place the missiles, respectively ufos, into that image.

```
; draw-scene : List-of-UFOs List-of-Missiles -> Image
; Draw a complete scene with all the UFOs and Missiles.

(define (draw-scene ufos missiles)
  (draw-ufos ufos (draw-missiles missiles BACKGROUND)))
```

Figure 2: Draw Scene Definition

2. Write two tests for *draw-scene*: one for a scene with no UFOs and Missiles and one for your scene with just one UFO and one missile.

3. Design the function *draw-missiles*. Test it with the same cases you used for *draw-scene*.

4. Design the function *draw-ufos*. Test it with the same cases you used for *draw-scene*.

5. In the interactions window, try out *draw-scene* for situations with multiple UFOs and Missiles to see that it does, in fact, scale up beyond the zero and one cases you used for testing.

Now that we can see the scene, let's take a look at the problem of removing the shot-down UFOs from the scene.

6. Design the function *hit-by?* which takes a missile (posn) and a ufo (posn) and returns true if the missile has struct the UFO and false otherwise. In this case hitting the UFO means that the missle posn is not more than 15 pixels (the radius of the UFO) from the UFO posn. You can compute the distance between the points (x_1, y_1) and x_2, y_2 as follows:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

7. Design the function *hit-by-any?* which takes a UFO (posn) and a list of Missiles and returns true if any of the missiles in the list have hit the UFO and false otherwise (Hint: Use *hit-by?* as a helper function).

8. Design the function *filter-safe* which takes a list of UFOs and a list of Missiles and computes the list of UFOs that have *not* been hit by any of the missiles. (Hint: Use *hit-by-any?* as a helper function.)

9. Come up with a scenario where one or more missiles has hit one or more UFOs. Develop lists for the before (all UFOs and Missiles in the scene) and after (all safe UFOs and all missiles) scenes. Then run a series of computations like those shown in figure 3 to see the effect of *filter-safe* in action.

```
(draw-scene BEFORE-UFOS MISSILES)
(draw-scene (filter-safe BEFORE-UFOS MISSILES)) MISSILES)
```

Figure 3: Visualizing the effect of *filter-safe*