Lab 4:
Railway to Hell

Due February April 20 by the start of lecture.

Overview

In this lab you will complete two separate tasks:

1. Estimate the value of π using the Monte Carlo method... but in F# this time!
2. Extend the railway programming example from lecture to incorporate more validation functions and other patterns in functional design.

Program 1

You will redo the Monte Carlo simulation from Lab 1 in F#, following these guidelines:

1. Create a record type for a `Point`, which has two float fields `xCoord` and `yCoord`.
2. Write F# implementations of these functions:
   (a) makePoint, which takes two floats and returns a `Point`
   (b) randomPoint, which generates two random floats and calls makePoint
      i. To generate a random number:
      ```
      let r = System.Random()
      let x = r.NextDouble()
      ```
   (c) throwDarts, which takes an integer `n` and generates a list of `Point` objects of length `n`. Use `List.init` to create a list of elements; it requires the number of elements to create, as well as a function that takes an unused input and returns an object to place into the list[1].
   (d) isHit, which takes a `Point` object and determines if it is a hit.
      i. First, declare a discriminated union named `HitResult`, with two cases: `Hit`, and `Miss of float`.
      ii. Determine if the `Point` object is a hit, and return `Hit` if so.
      iii. Otherwise, determine the **distance** from the point to the nearest point on the imaginary circle of radius 1. (This is not as hard as it sounds.) Return `Miss` with that value.
   (e) countHits, which takes an integer for how many darts to throw, "throws" that many darts, maps each throw to a `HitResult`, and counts how many of those are `Hits`.
   (f) estimatePi, which takes an integer for how many darts to throw, and returns a Monte Carlo estimate for π.

Deliverables:

Turn in:

1. A printed copy of your F# source code.
2. The F# **type** of **each of** the functions from your program. **Try to do this without using the IDE, except to double check your work**... you WILL have to do something like this on your next midterm.
   Example: makePoint is of type `float->float->Point`

[1] This may help: https://stackoverflow.com/questions/6062191/f-getting-a-list-of-random-numbers

Program 2

Starting with the code given to you in the F# repository (RailwayProgramming), extend the registration validation example by following these instructions:

1. Create a list of strings named `existingAccounts` containing five distinct (but fake) email addresses, none of which contains a period or dash.
2. Create another list of strings named `blacklistedDomains` containing the values "`mailinator.org`" and "`throwawaymail.com`".
3. Write a new validation (switch) function `uniqueEmail`, which takes a `RegistrationAttempt` and validates that the email address is not in the `existingAccounts` list.
4. Write a new validation function `emailNotBlacklisted`, which takes a `RegistrationAttempt` and validates that the **domain of the email address** (following the `@`) is not in the `blacklistedDomains` list.
5. Consider a new class of function, a *single-track* function, which accepts a Success track and outputs (only) a Success track. Such functions might be used to transform the input in some way, as long as the input is still in a success state. We can't use `bind` or `>=>` with a single-track function, because `bind` only works with switch functions. What we need is a wrapper helper that takes a single-track function input and returns a switch function, where the Failure track is unused (since the one-track function cannot fail).
   (a) Write a function `switch`, which "promotes" a single-track function to a switch function. `switch` takes a function `f` and a `RegistrationAttempt` as parameters, invokes the function on the attempt, and returns `Success` with the resulting value. (Thereby converting `f` to a switch function, of one input -> two outputs.)
   (b) Write a function `lowercaseEmail`, which takes a `RegistrationAttempt` and returns a new `RegistrationAttempt` with the same username and the email address converted to all-lowercase.
   (c) Write a function `canonicalizeEmail`, which takes a `RegistrationAttempt` and "canonicalizes" the email address of the attempt by *removing all periods and dashes* to the left of the `@` symbol. The function returns a new `RegistrationAttempt` with the same username as before, and the new canonicalized email.
6. Incorporate the new functions into the existing validation system:
   (a) After determining that the email has a local part, use `switch` to promote `canonicalizeEmail` to a switch function, so that future validation functions see only the canonical email, and not the original email.
   (b) Do the same so that the email address is converted to all lowercase.
   (c) Next validate that the (canonical and lowercased) email is not of a blacklisted domain.
   (d) Next validate that the email is unique.

   Perform these additions by modifying the `finalValidateRegistration` function definition at the bottom of the F# example. Each change should require a single `>=>` operator added to the existing chain at an appropriate place.
7. Finally, demonstrate your code works by creating enough distinct `RegistrationAttempt` objects to cover each of the possible validation failure reasons.