Lab 3:
Railway to Hell

Due February November 6 by the end of lab.

Overview

In this lab you will extend the railway programming example from lecture to incorporate more validation functions and other patterns in functional design.

Railway Terminology

Recall the following terms from the Railway Programming lecture:

1. **Two-track function**: a two-track function accepts a `ValidationResult` object as its input (which has two "tracks": `Success` and `Failure`), and produces a `ValidationResult` as its output. Two-track functions can be composed, because the output type of one matches the input type of another.
2. **Switch function**: a switch function accepts a `RegistrationAttempt` object as input and produces a `ValidationResult` output. It is like a railroad "switch": one track comes in, two tracks go out. A switch function cannot be composed with other switch functions, because its output type (two-track) does not match its input type (one-track).
3. **Bind function**: the bind function "wraps" a switch function – which normally expects a `ValidationResult` – inside of a two-track function called the *bind*. The bind accepts a two-track `ValidationResult` and if the input is on the `Success` track, invokes the wrapped switch function and returns its `ValidationResult` (which could result in either `Success` or `Failure`). If the two-track input is on the `Failure` track, the bind just returns `Failure`. Binding a switch function produces a new function that can be composed with other two-track functions, because its two-track output type now matches its two-track input.

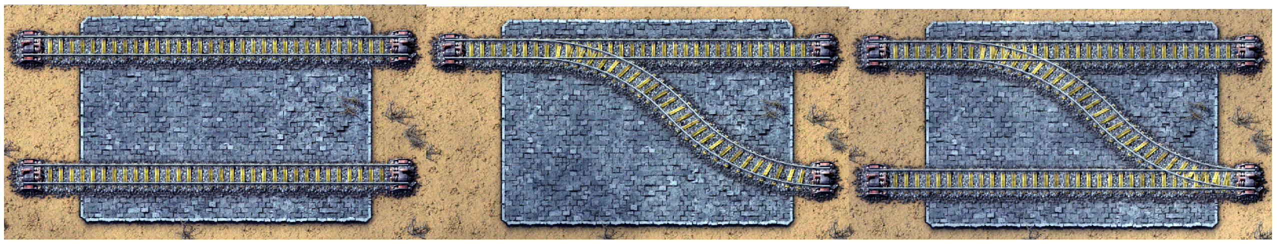

Figure 1: Example tracks. From left to right: a two-track function; a switch function; a binding over a switch function. In each, the upper track is success and the lower is failure. The gray "concrete" box encapsulates the function's "body".

Extensions:

We will extend these function types with new track types:

1. **One-track function**: a one-track function accepts a `RegistrationAttempt` and returns a `RegistrationAttempt`. It does not do any validation; its purpose is to transform the input in some way, perhaps by lower-casing the email address or otherwise manipulating the data to make it easier to validate later. These functions do not know how to handle `Failure` inputs; they can only be invoked on successful attempts. They can be composed with each other – since the output type matches the input type – but not with two-track functions.
2. **Bypass function**: a bypass function is a switch function that wraps a one-track function. The single-track input to the switch is passed to the one-track function, and the result is reported as `Success`. A

dummy `Failure` output track is never actually used, but gives the one-track function the same shape as a swich function, so it can then be binded into a two-track function.

3. **Terminal function**: a terminal function is a switch-type function that needs more parameters than just a `RegistrationAttempt` to do its validation. For example, to validate that an email address has not been used before, a function would need a parameter representing an existing set of user emails. A normal switch function can only take a single parameter (the registration attempt input); how then can we provide extra parameters to a switch function?

   Lucky for us, F# makes this easy. If we write our terminal function such that the *last parameter* is the `RegistrationAttempt` input expected of a switch function, then we can wrap a binding around a *partial invocation* of the terminal, in which we call the terminal function and provide *all but the last parameter.* (Therefore passing the "extra information" needed by the terminal.) Such a partial terminal function call can be placed into a bind operation, including the `>=>` operator.

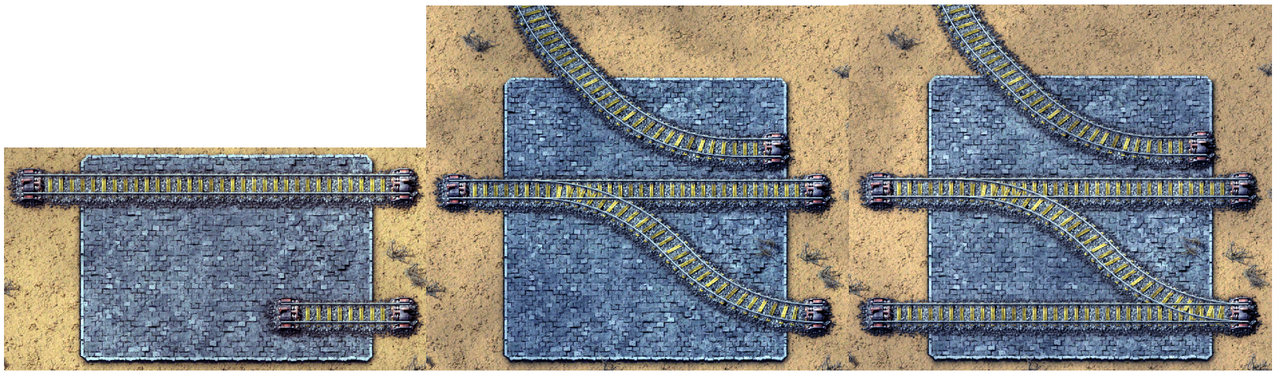

Figure 2: More track types. A bypass function over a single-track function; a terminal function with an extra parameter; a binding over the same terminal function, making it two-track.

Assignment

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 these validation functions:
   (a) A terminal function `uniqueEmail`, which takes a list of strings and a `RegistrationAttempt` as parameters, and validates that the attempt's email address is not in the list of strings. You cannot access the global list you created in step (1); you must use the parameter to the function.
   (b) A terminal function `emailNotBlacklisted`, which takes a list of strings and a `RegistrationAttempt` and validates that the **domain of the email address** (following the `@`) is not in the list of strings. Same rule as (a).

   Both of these terminal functions can be added to your validation "chain" using partial invocation; simply adding `>=> uniqueEmail` *`myListOfEmails`* should do the trick.
4. Write helper functions to create a bypass over a single-track function:
   (a) Write the function `bypass`, which takes a single-track function and a `RegistrationAttempt` as parameters, invokes the single-track function on the `RegistrationAttempt`, and returns the result as a `Success`.

   (b) Write an operator `>->` by mimicking the `>=>` operator, except that:
      i. the second parameter is not a switch function, but a `bypassFunction`.
      ii. the `bypassFunction` needs to be promoted to switch using `bypass` before passing it to `bind`. Everything else in the operator stays the same.

5. Write these single-track functions:
   (a) `lowercaseEmail`, which takes a `RegistrationAttempt` and returns a new `RegistrationAttempt` with the same username and the email address converted to all-lowercase.
   (b) `canonicalizeEmail`, which takes a `RegistrationAttempt` and "canonicalizes" the email address of the attempt **only if** the domain of the email (to the right of the `@`) is `gmail.com`. To canonicalize such an email address:
      i. remove all periods and dashes in the local-part (the left of the `@` symbol). Example: `neal.terrell@gmail.com` -> `nealterrell@gmail.com` (This is not my email address.)
      ii. if a `+` is present in the local-part, remove it **and** all characters that follow it in the local part. Example: `nealterrell+spam@gmail.com` -> `nealterrell@gmail.com`

      The function returns a new `RegistrationAttempt` with the same username as before, and the new canonicalized email. Any non-gmail address is not changed.

   Both single-track functions can be added to your validation chain using the `>->` operator.

6. Incorporate the new functions into the existing validation system:
   (a) After determining that the email has a local part, convert the email address to all lowercase, then canonicalize it.
   (b) Next validate that the (canonical and lowercased) email is not of a blacklisted domain.
   (c) Next validate that the email is unique.

   Perform these additions by modifying the `validate3` function definition at the bottom of the F# example. Each change should require a single `>=>` or `>->` operator added to the existing chain at an appropriate place.

Deliverables

Turn in the following when the lab is due:

1. A printed copy of your code, **printed from your IDE when possible.** If you cannot print from your editor, copy your code into Notepad or another program with a fixed-width (monospace) font and print from there.
2. Create five registration attempt records and pass each to your final validation function chain; print the output of each registration attempt. Your five records should involve:
   (a) At least two `@gmail.com` email addresses that need to be canonicalized in different ways.
      i. One of those emails should match an entry in your `existingAccounts` list, after it is canonicalized.
   (b) At least one email address that is not a gmail address.
   (c) At least one non-gmail address that uses characters which would be removed during canonicalization *if it were a gmail address.*
   (d) At least one email address that has uppercase characters.
   (e) At least one email address that uses a blacklisted domain.
   (f) At least one email address with no local part.
   (g) At least one username that is empty.
   (h) At least one registration attempt that succeeds.

Your five records should cover every possible way of failing the validation functions. Turn in a copy of the output of your program showing the result of each registration attempt.