Lec 23: Railway-Oriented Programming

CS220: Programming Principles

Sang Kil Cha

Railway-Oriented Programming

Disclaimer

This lecture is mainly inspired by `https://fsharpforfunandprofit.com/rop/`.

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

Motivation

ROP is a functional way of sequentially composing functions. It makes our code more elegant and easier to understand.

Recall: Maybe Computation Expression

```
type MaybeBuilder () =
  member __.Bind (m, f) = Option.bind f m
  member __.Return (m) = Some m

let maybe = MaybeBuilder ()
```

Example: Input Validation

```
type Input = { ID: string; PW: string }

let validateID input =
  if input.ID.Length > 4 && input.ID.Length <= 12 then Some input else None

let validatePWLength input =
  if input.PW.Length > 8 then Some input else None

let validatePWLowercase input =
  if input.PW |> String.exists System.Char.IsLower then Some input else None

let validatePWUppercase input =
  if input.PW |> String.exists System.Char.IsUpper then Some input else None

let validatePWDigit input =
  if input.PW |> String.exists System.Char.IsDigit then Some input else None
```

With Maybe Computation Expression

```
let validate input = maybe {
  let! _ = validateID input
  let! _ = validatePWLength input
  let! _ = validatePWLowercase input
  let! _ = validatePWUppercase input
  let! _ = validatePWDigit input
  return input
}
```

How about using the bind operator (»=)?

With Bind Operator

```
let (>>=) m f = Option.bind f m

let validate input =
  Some input
  >>= validateID
  >>= validatePWLength
  >>= validatePWLowercase
  >>= validatePWUppercase
  >>= validatePWDigit
```

Using Result Type

To get more information about the failure, we can use the `Result` type, which also provides the same high-order function, `bind`.

```
let (>>=) m f = Result.bind f m

let validate input =
  Ok input
  >>= validateID
  >>= validatePWLength
  >>= validatePWLowercase
  >>= validatePWUppercase
  >>= validatePWDigit
```

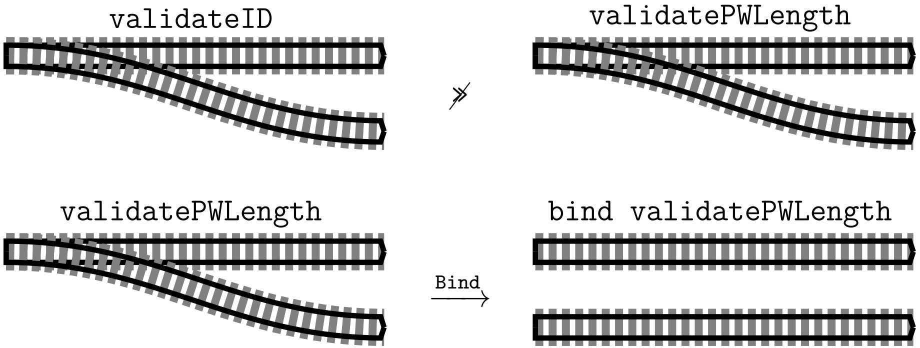
Composing Functions

We can compose functions using the » operator, too.

```
let validate =
  validateID
  >> Result.bind validatePWLength
  >> Result.bind validatePWLowercase
  >> Result.bind validatePWUppercase
  >> Result.bind validatePWDigit
```

We can think `bind` as a ***converter*** that converts a function into a new function that takes in a `Result` type as input, and returns a `Result` type as output.

Visual Explanation (Railway Analogy)



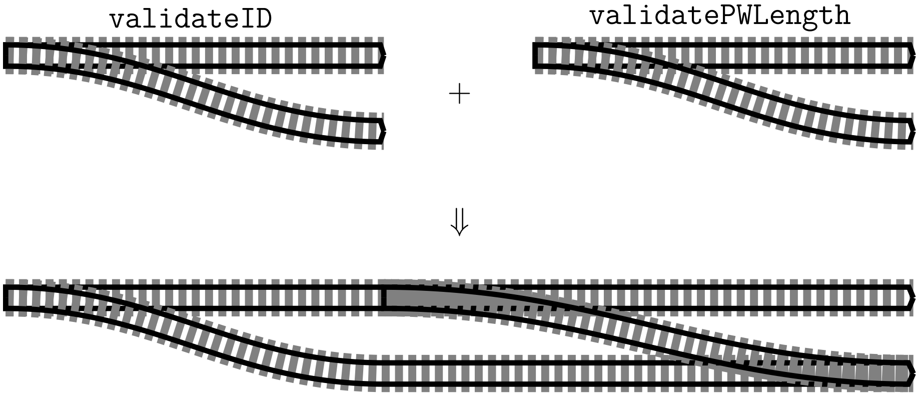
After the `bind` operation, a railway switch (one-track input) is converted into a two-track input.

Revisiting »=

The operator »= combines one-to-two track railways by converting them into a two-to-two track railway.

An Alternative to Bind

Can we combine railways without converting them?



This just becomes another switch (one-to-two track railway).

Kleisli Composition

```
let (>=>) f g x =
  match f x with
  | Ok res -> g res
  | Error e -> Error e

let validate =
  validateID
  >=> validatePWLength
  >=> validatePWLowercase
  >=> validatePWUppercase
  >=> validatePWDigit
```

Bind vs. Kleisli Composition

- `bind` is an adapter that converts a one-to-two track railway into a two-to-two track railway, so that it can be combined with another two-to-two railways.
- `>=>` is a combinator that combines two one-to-two track railways into another one-to-two track railway.
- If we have an existing two-to-two track railway, we would use `bind` to combine a one-to-two track railway with it. If all the railways we have are one-to-two track railways, we would simply use `>=>` to combine them.

Revisiting Kleisli Composition

```
let (>=>) f g x =
  match f x with
  | Ok res -> g res
  | Error e -> Error e

// Rewritten with function composition.
let (>=>) f g =
  f >> Result.bind g
```

Inserting Regular Functions into the Railway

Previous functions are all one-to-two track railways, but how about one-to-one track railways? Can we combine them with the existing railways?

Suppose we want to trim whitespace from the given user input. We would write a one-to-one track railway function:

```
let trimID input =
  { input with ID = input.ID.Trim () }

let trimPW input =
  { input with PW = input.PW.Trim () }
```

Introducing `switch`

`switch` is a function that converts a one-to-one track railway into a one-to-two track railway.

```
let switch f x = Ok (f x)
// or more simply
let switch f = f >> Ok

let validate =
  switch trimID
  >=> switch trimPW
  >=> validateID
  >=> validatePWLength
  >=> validatePWLowercase
  >=> validatePWUppercase
  >=> validatePWDigit
```

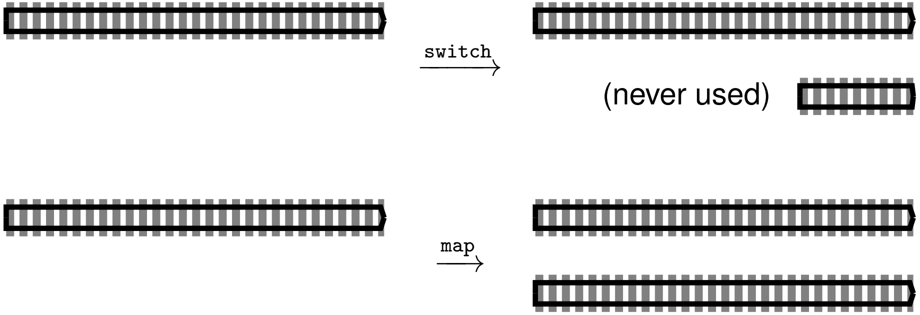
Map: One-to-One to Two-to-Two

How about converting a one-to-one track railway into a two-to-two track railway? This operation is typically called `map`, and it is defined in the standard library: `Result.map`.

```
let map = Result.map

let validate =
  switch trimID
  >> map trimPW
  >=> validateID
  >=> validatePWLength
  >=> validatePWLowercase
  >=> validatePWUppercase
  >=> validatePWDigit
```

Switch and Map



T-Splitter Function

- `tee` is a function that splits a railway into two railways, one of which is a dead-end railway[1]. It works like `tee` in Linux.
- `tee` is a command-line tool in Linux that reads from standard input and writes to standard output and files.
  - `echo "Hello" | tee file.txt`
  - The above command writes "Hello" to the file `file.txt` and also prints "Hello" to the standard output.
- The name `tee` comes from the T-splitter in plumbing.

[1] A dead-end here means that the function will not return any value and just produce a side effect.

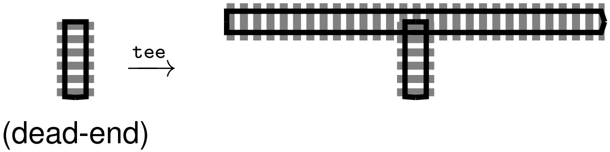
Debugging with `tee`

```
let tee f x =
  f x |> ignore
  x

let debugPrint input = printfn "%A" input

let validate =
  switch trimID
  >=> switch (tee debugPrint) // or >> tee debugPrint
  >=> switch trimPW
  >=> validateID
  >=> validatePWLength
  >=> validatePWLowercase
  >=> validatePWUppercase
  >=> validatePWDigit
```

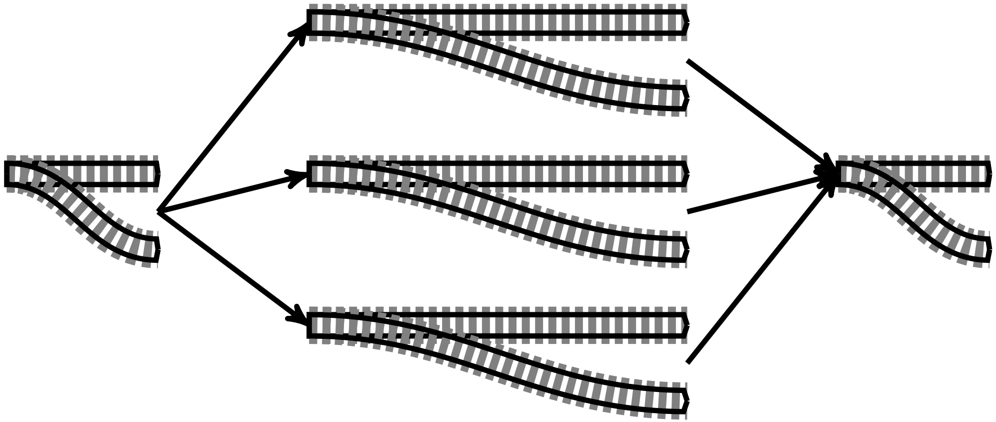
tee



Parallel Combination of Railways

- So far, we have been combining railways in a sequential manner.
- But parallel combination is often desirable. For example, we may want to get two or more validation errors at once, instead of just one. Current validator will stop at the first error!

Parallel Combination



Parallel Combinator: `&&&`

`&&&` is a parallel combinator that combines two railways into one.

```
let (&&&) f g x =
  match f x, g x with
  | Ok res1, Ok _ -> Ok res1 (* returning either one is ok *)
  | Error e, Ok _ -> Error e
  | Ok _, Error e -> Error e
  | Error e1, Error e2 -> Error (e1 + Environment.NewLine + e2)

let validateIDAndPW =
  validateID
  &&& validatePWLength
  &&& validatePWLowercase
  &&& validatePWUppercase
  &&& validatePWDigit

let validate =
  switch trimID
  >=> switch trimPW
  >=> validateIDAndPW
```

Key Takeaway

Once we define the railway operators, such as `bind`, `>=>`, `switch`, etc., we can easily compose functions in a railway-oriented manner, which makes our code more elegant and easier to understand. ROP is a good example showing the power of functional abstraction.

There are more railway operators didn't cover in this lecture. For example, one could consider handling exceptions in a railway-oriented manner. For more information, please refer to the original article.

In-Class Activity #23

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.

```
- git clone https://github.com/KAIST-CS220/CS220-Main.git
```

2. Move in to the directory `CS220-Main/Activities`

```
- cd CS220-Main
- cd Activities
```

Problem

Modify the given implementation as well as the parallel combinator `&&&` so that the program can handle a list of error messages (`string list`) instead of a single error message (`string`).

Conclusion

Further Readings

- https://fsharpforfunandprofit.com/posts/recipe-part2/.

Question?