IT UNIVERSITY OF COPENHAGEN

Functional Programming

Patrick Bahr

Monads & Computation Expressions

Based on original slides by Jesper Bengtson

Last week

Last week

- Mutual Recursion

Last week

- Mutual Recursion
- Sequences
  - Lazy evaluation
  - Sequence expressions

This week

This week

- Monads
  - How to abstract common programming patterns
  - Analogy: "Railway-oriented programming"

This week

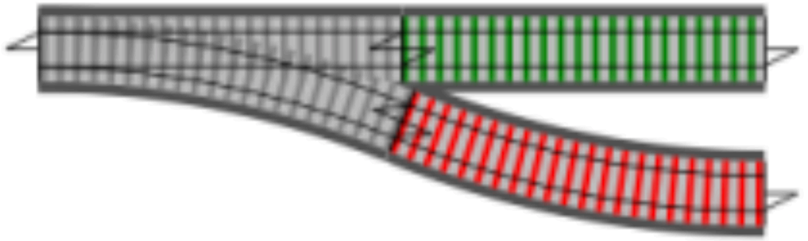
- Monads
  - How to abstract common programming patterns
  - Analogy: "Railway-oriented programming"
- Computation expressions
  - Ergonomic syntax for monads & similar structures

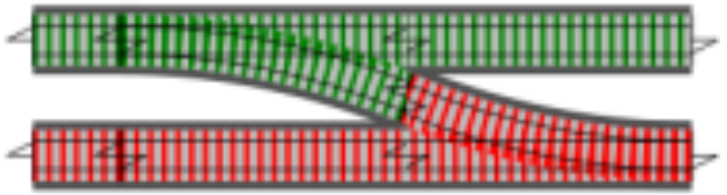
Part I

Railway-Oriented Programming

Credit where credit is due

Thanks to Scott Wlaschin for the use of his railway images from his site *F# for fun and profit*





What to do when things go wrong

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Sub of aExp * aExp
| Mul of aExp * aExp
```

Recall Exercise 3.2: write a function

```
arithEvalState : aExp -> Map<string,int> -> int
```

that evaluates arithmetic expressions

What to do when things go wrong

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Sub of aExp * aExp
| Mul of aExp * aExp
```

Recall Exercise 3.2: write a function

```
arithEvalState : aExp -> Map<string,int> -> int
```

that evaluates arithmetic expressions

```
let rec eval (e : aExp) s : int =
  match e with
   | N n           -> n
   | V x           -> Map.find x s
   | Add (a1, a2) -> eval a1 s + eval a2 s
   | Sub (a1, a2) -> eval a1 s - eval a2 s
   | Mul (a1, a2) -> eval a1 s * eval a2 s
```

What to do when things go wrong

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Sub of aExp * aExp
| Mul of aExp * aExp
```

Recall Exercise 3.2: write a function

```
arithEvalState : aExp -> Map<string,int> -> int
```

that evaluates arithmetic expressions

```
let rec eval (e : aExp) s : int =
  match e with
   | N n           -> n
   | V x           -> Map.find x s
   | Add (a1, a2) -> eval a1 s + eval a2 s
   | Sub (a1, a2) -> eval a1 s - eval a2 s
   | Mul (a1, a2) -> eval a1 s * eval a2 s
```

Observation:

- Map.find throws an exception if a key is not found
- Option types are generally better

What to do when things go wrong

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Sub of aExp * aExp
| Mul of aExp * aExp
```

Recall Exercise 3.2: write a function

```
arithEvalState : aExp -> Map<string,int> -> int
```

that evaluates arithmetic expressions

We want return type `int option` instead.

```
let rec eval (e : aExp) s : int =
  match e with
   | N n           -> n
   | V x           -> Map.find x s
   | Add (a1, a2) -> eval a1 s + eval a2 s
   | Sub (a1, a2) -> eval a1 s - eval a2 s
   | Mul (a1, a2) -> eval a1 s * eval a2 s
```

Observation:

- Map.find throws an exception if a key is not found
- Option types are generally better

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let rec eval (e : aExp) s : int =
  match e with
   | N n             -> n
   | V x             -> Map.find x s
   | Add (a1, a2) -> eval a1 s + eval a2 s
   | Sub (a1, a2) -> eval a1 s - eval a2 s
   | Mul (a1, a2) -> eval a1 s * eval a2 s
```

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let rec eval (e : aExp) s : int =
  match e with
   | N n              -> n
   | V x              -> Map.find x s
   | Add (a1, a2) -> eval a1 s + eval a2 s
   | Sub (a1, a2) -> eval a1 s - eval a2 s
   | Mul (a1, a2) -> eval a1 s * eval a2 s
```

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let rec eval (e : aExp) s : int =
  match e with
   | N n             -> n
   | V x             -> Map.find x s
   | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2) s
   | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2) s
   | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2) s
```

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let binop f a b s =
    f (a s) (b s)

let rec eval (e : aExp) s : int =
  match e with
  | N n           -> n
  | V x           -> Map.find x s
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2) s
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2) s
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2) s
```

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let binop f a b s =
    f (a s) (b s)

let rec eval (e : aExp) s : int =
  match e with
  | N n           -> n
  | V x           -> Map.find x s
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2) s
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2) s
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2) s
```

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let binop f a b s =
    f (a s) (b s)

let rec eval (e : aExp) (s : Map<string,int>) : int =
  match e with
  | N n           -> n
  | V x           -> Map.find x s
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2) s
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2) s
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2) s
```

What to do when things go wrong

Let’s refactor this implementation a bit.

```
let binop f a b s =
    f (a s) (b s)

let rec eval (e : aExp) : Map<string,int> -> int =
  match e with
   | N n           -> fun _ -> n
   | V x           -> Map.find x
   | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
   | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
   | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

What to do when things go wrong

Let's refactor this implementation a bit.

```
let binop f a b s =
    f (a s) (b s)
```

Question:
What is the type of `binop`?

```
let rec eval (e : aExp) : Map<string,int> -> int =
  match e with
   | N n           -> fun _ -> n
   | V x           -> Map.find x
   | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
   | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
   | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

What to do when things go wrong

Let's refactor this implementation a bit.

```
let binop f a b s =
    f (a s) (b s)
```

Question:
What is the type of `binop`?

```
let rec eval (e : aExp) : Map<string,int> -> int =
  match e with
   | N n           -> fun _ -> n
   | V x           -> Map.find x
   | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
   | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
   | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

Now we are ready to change `eval` so that it returns `int option` instead of `int`

What to do when things go wrong

```
let binop f a b s =
    f (a s) (b s)
```

```
let rec eval (e : aExp) : Map<string,int> -> int =
  match e with
   | N n           -> fun _ -> n
   | V x           -> Map.find x
   | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
   | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
   | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

What to do when things go wrong

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None

let rec eval (e : aExp) : Map<string,int> -> int option =
  match e with
  | N n           -> fun _ -> Some n
  | V x           -> Map.tryFind x
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

What to do when things go wrong

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None

let rec eval (e : aExp) : Map<string,int> -> int option =
  match e with
  | N n           -> fun _ -> Some n
  | V x           -> Map.tryFind x
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)

        where  Map.tryFind : 'a -> Map<'a,'b> -> 'b option
```

binop changed to accommodate option type

Some n instead of **n**

Map.tryFind instead of **Map.find**

Let’s add division

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None

let rec eval (e : aExp) : Map<string,int> -> int option =
  match e with
  | N n           -> fun _ -> Some n
  | V x           -> Map.tryFind x
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

Let’s add division

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None
```

```
type aExp =
          …
| Div of aExp * aExp
```

```
let rec eval (e : aExp) : Map<string,int> -> int option =
  match e with
  | N n           -> fun _ -> Some n
  | V x           -> Map.tryFind x
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
```

Let’s add division

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None
```

```
type aExp =
          …
| Div of aExp * aExp
```

```
let rec eval (e : aExp) : Map<string,int> -> int option =
  match e with
  | N n           -> fun _ -> Some n
  | V x           -> Map.tryFind x
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
  | Div (a1, a2) ->
     fun s -> match eval a1 s, eval a2 s with
              | Some x, Some y when y <> 0 -> Some (x / y)
              | _, _                       -> None
```

Let’s add division

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None
```

```
type aExp =
          …
| Div of aExp * aExp
```

```
let rec eval (e : aExp) : Map<string,int> -> int option =
  match e with
  | N n           -> fun _ -> Some n
  | V x           -> Map.tryFind x
  | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
  | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
  | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
  | Div (a1, a2) ->
     fun s -> match eval a1 s, eval a2 s with
              | Some x, Some y when y <> 0 -> Some (x / y)
              | _, _                       -> None
```

Almost the same as **binop**, but with a check that **y <> 0**

Let's add division

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
```

```
type aExp =
        …
| Div of aExp * aExp
```

```
let
  m
   |
   |
   |
   |
   |
  | Div (a1, a2) ->
     fun s -> match eval a1 s, eval a2 s with
                | Some x, Some y when y <> 0 -> Some (x / y)
                | _, _                       -> None
```

ption =

)
)
)

This is getting hairier.

Matches against Some and None are starting to permeate the program and this will only get worse

Almost the same as **binop**, but with a check that **y <> 0**

To sum up

- We can account for errors using **`option` type**
- But: This may require some **refactoring**

To sum up

- We can account for errors using **`option` type**
- But: This may require some **refactoring**
- This is a **common pattern** when implementing
  - error handling
  - computations with state (e.g. Map<string,int>)
  - …
- There is a general technique to **encapsulate** these patterns …

Part IB

Part IB

Monads

Monads

Monads

- A **monad** is a mathematical concept that helps structuring code like this.

Monads

- A **monad** is a mathematical concept that helps structuring code like this.
- Boilerplate code **written only once** and is hidden behind the scenes, e.g.
  - propagating error messages
  - maintain some internal state
  - …

Monads

- A **monad** is a mathematical concept that helps structuring code like this.
- Boilerplate code **written only once** and is hidden behind the scenes, e.g.
  - propagating error messages
  - maintain some internal state
  - …
- Can be thought of as “programmable semicolons”

Monads

- A **monad** is a mathematical concept that helps structuring code like this.
- Boilerplate code **written only once** and is hidden behind the scenes, e.g.
  - propagating error messages
  - maintain some internal state
  - …
- Can be thought of as “programmable semicolons”

programmable semicolons

```
<do this> ;
<then do this> ;
...
```

Monads

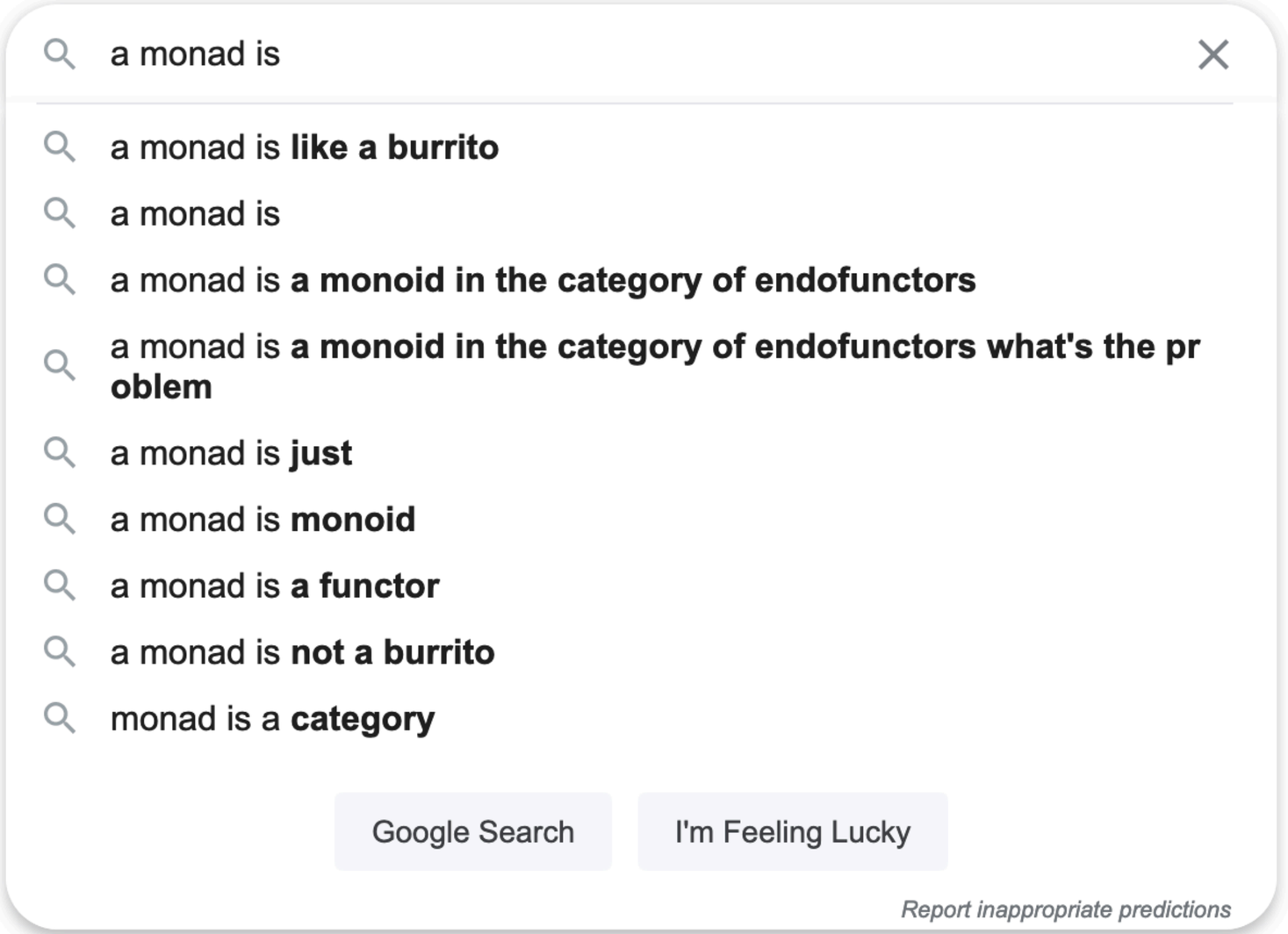
- A **monad** is a mathematical concept that helps structuring code like this.
- Boilerplate code **written only once** and is hidden behind the scenes, e.g.
  - propagating error messages
  - maintain some internal state
  - …
- Can be thought of as “programmable semicolons”

programmable semicolons

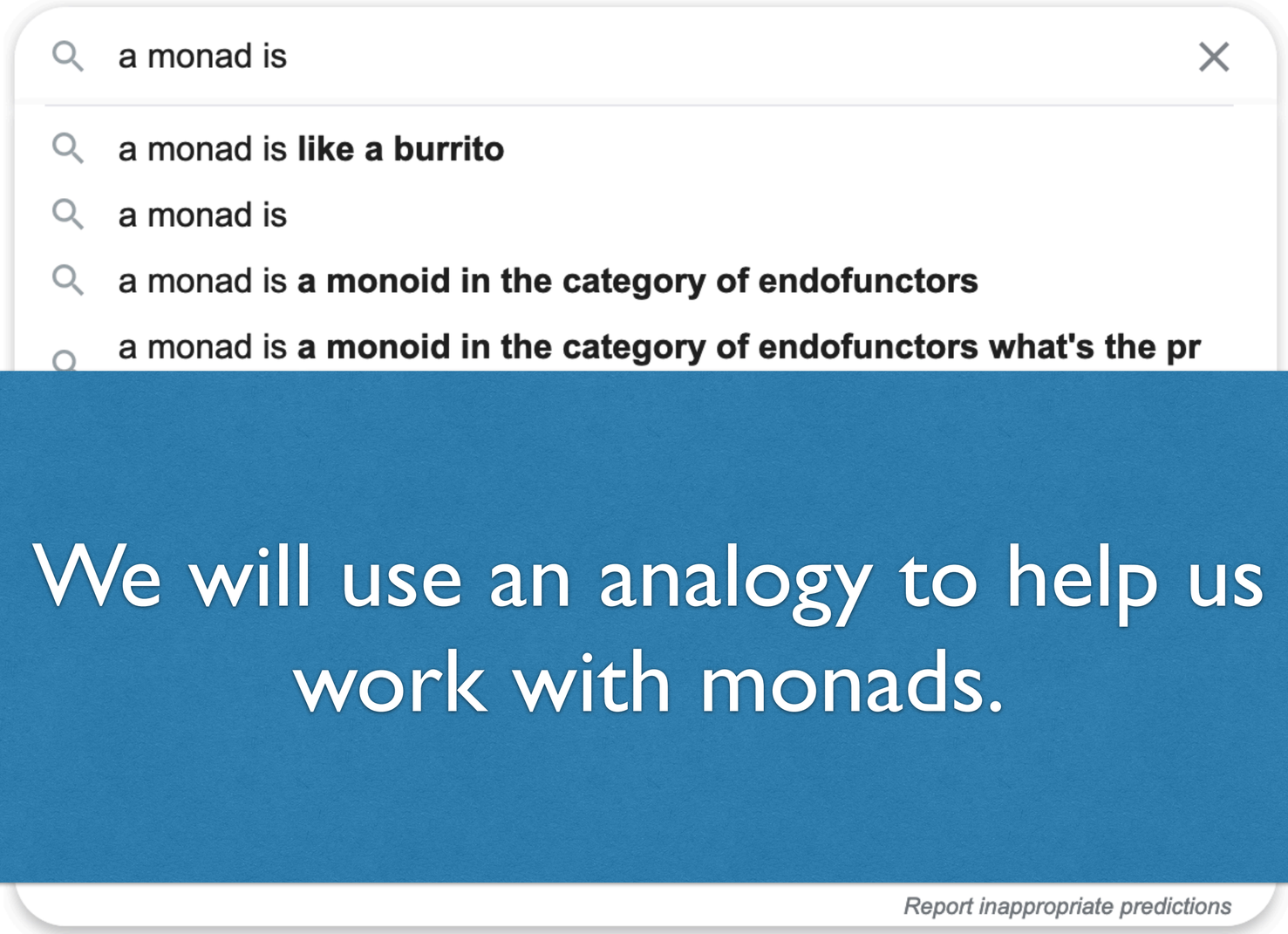
```
<do this> ;
<then do this> ;
...
```

What should ";" do?

`


a monad is
a monad is like a burrito
a monad is
a monad is a monoid in the category of endofunctors
a monad is a monoid in the category of endofunctors what's the problem
a monad is just
a monad is monoid
a monad is a functor
a monad is not a burrito
monad is a category
Google Search
I'm Feeling Lucky
Report inappropriate predictions

`



We will use an analogy to help us work with monads.

Railway-oriented programming

- Observation: Many functions have two kinds of outputs (success and error)
- Idea:
  - *Focus* on the success path.
  - Propagate errors to the top *automatically*.

Functions as railways

A **“normal” function** is just a straight railway with one input and one output

```
'a -> 'b
```

Input  Output

Functions as railways

A **“normal” function** is just a straight railway with one input and one output

```
'a -> 'b
```

Input  Output

We can lengthen the track using **function composition**:

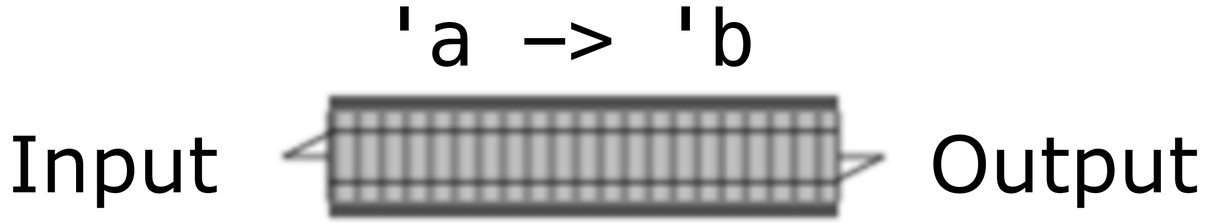
```
'a -> 'b
```

```
'b -> 'c
```

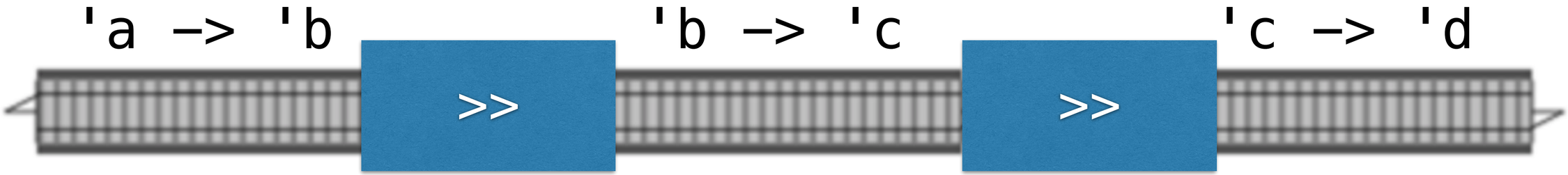
```
'c -> 'd
```

Functions as railways

A **“normal” function** is just a straight railway with one input and one output



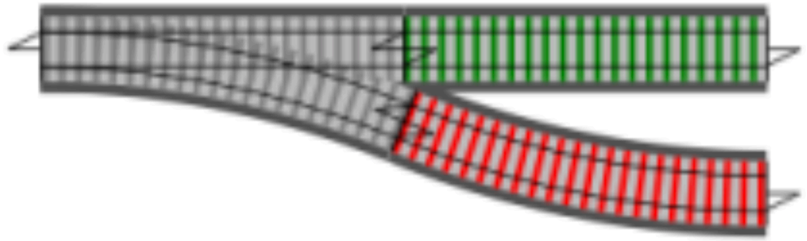
We can lengthen the track using **function composition**:



Functions as railways

A function that returns an option type can be seen as a **railway switch**

```
'a -> 'b option
```

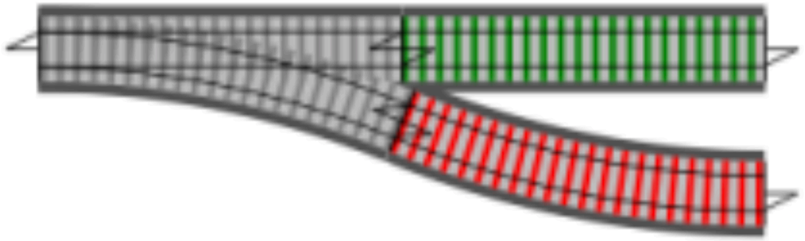


Functions as railways

A function that returns an option type can be seen as a **railway switch**

```
'a -> 'b option
```

Input

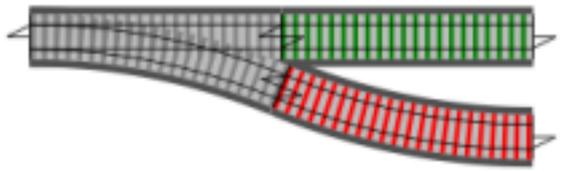


Some output

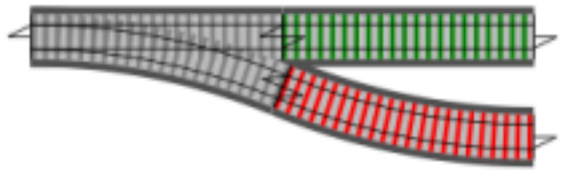
None

<u>But:</u> **Composing** these is problematic.

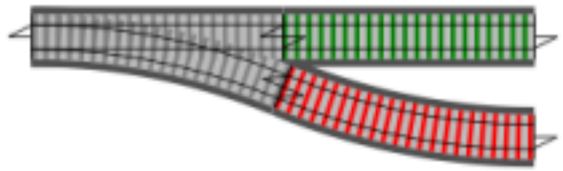
```
'a -> 'b option
```



```
'b -> 'c option
```



```
'c -> 'd option
```

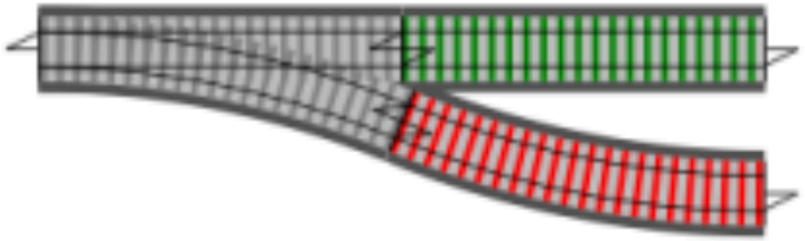


Functions as railways

A function that returns an option type can be seen as a **railway switch**

```
'a -> 'b option
```

Input

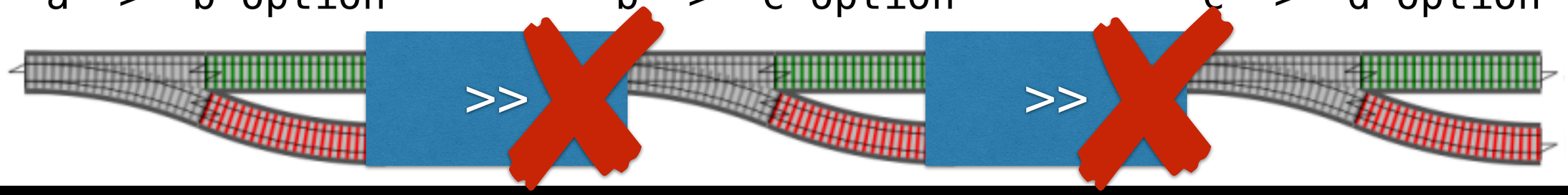


Some output

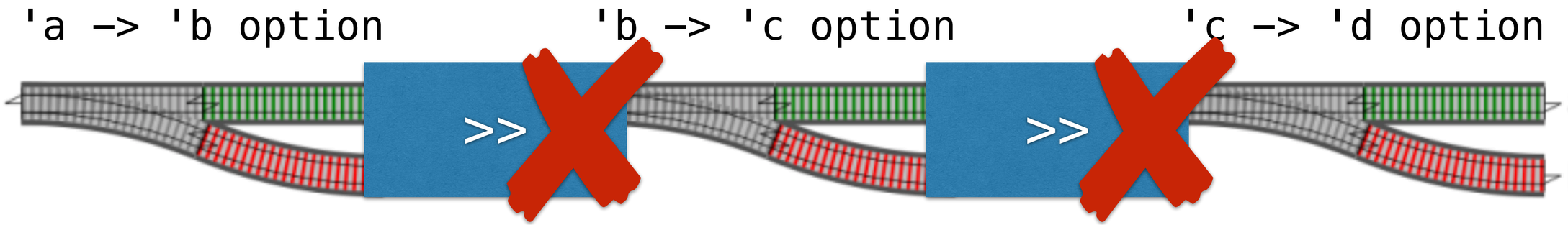
None

<u>But:</u> **Composing** these is problematic.

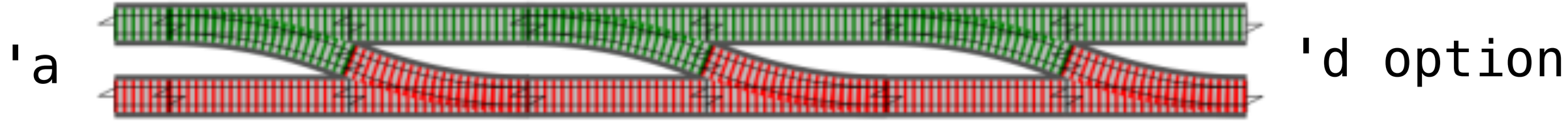
```
'a -> 'b option        'b -> 'c option        'c -> 'd option
```



Functions as railways



What we want is the following



Once you hit the error track you are stuck there

Our “Railway” Toolbelt

Recall that a “railway switch” has the type

```
'a -> 'b option
```

We need two helper functions

Our “Railway” Toolbelt

Recall that a “railway switch” has the type

```
'a -> 'b option
```

We need two helper functions

1. A **constant switch** that stays on the green track

```
ret : 'a -> 'a option
```

Our “Railway” Toolbelt

Recall that a “railway switch” has the type

```
'a -> 'b option
```

We need two helper functions

1. A **constant switch** that stays on the green track

```
ret : 'a -> 'a option
```

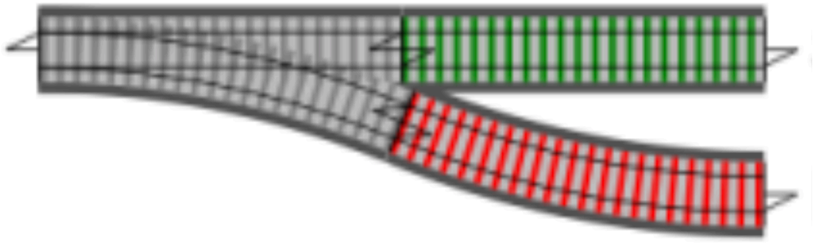
2. Turn **switches into two-track functions** in order to be able to compose them

```
bind : ('a -> 'b option) -> ('a option -> 'b option)
```

Turn constants into switches

We want to construct the constant switch (which just passes the argument through)

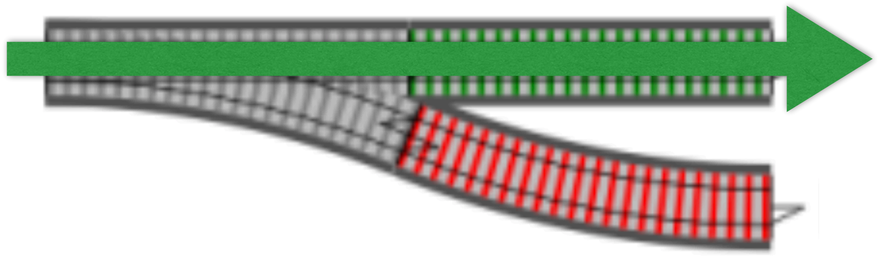
```
'a -> 'a option
```



Turn constants into switches

We want to construct the constant switch (which just passes the argument through)

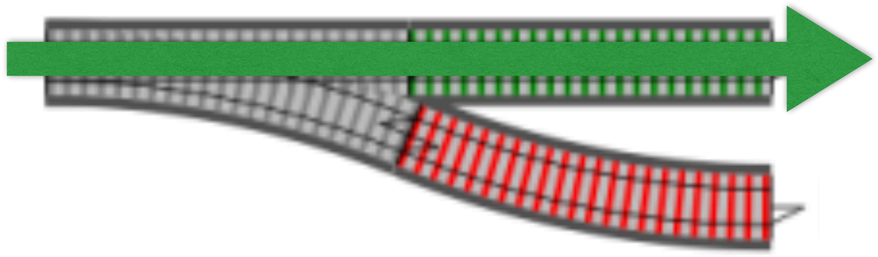
```
'a -> 'a option
```



Turn constants into switches

We want to construct the constant switch (which just passes the argument through)

```
'a -> 'a option
```



This is easy enough:

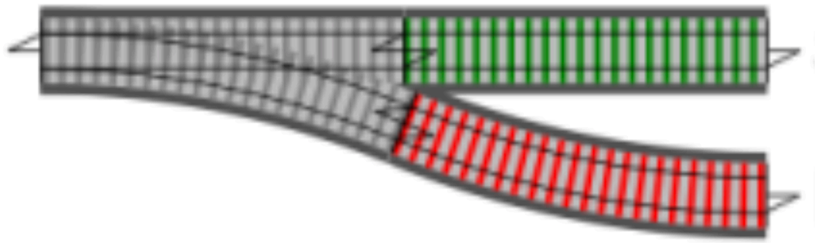
```
let ret x = Some x
```

```
ret : 'a -> 'a option
```

Turn switches into two-track functions

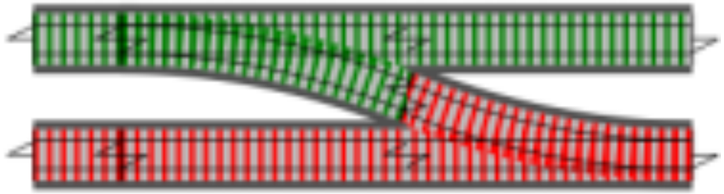
We want to turn

```
'a -> 'b option
```



into this

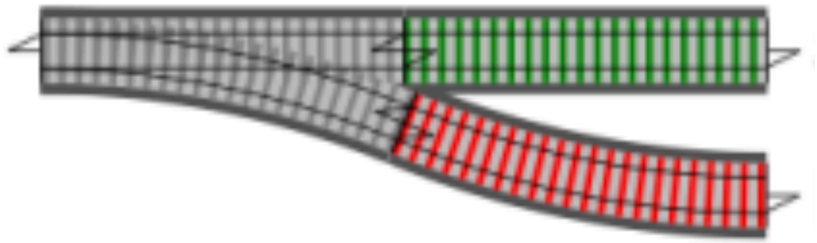
```
'a option -> 'b option
```



Turn switches into two-track functions

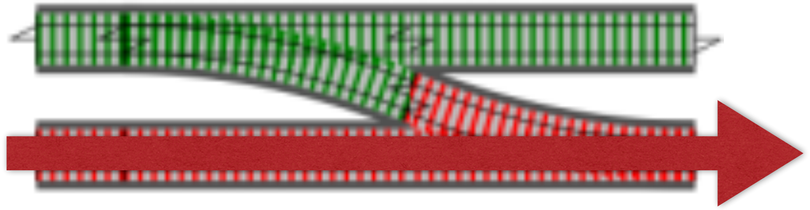
We want to turn

```
'a -> 'b option
```



into this

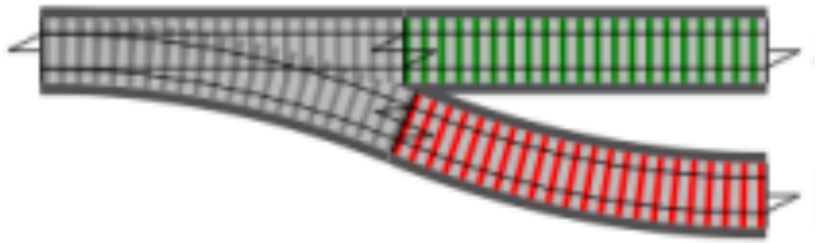
```
'a option -> 'b option
```



Turn switches into two-track functions

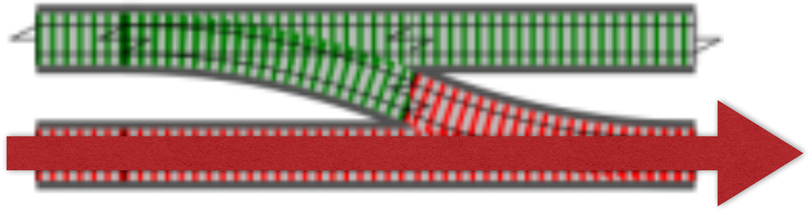
We want to turn

```
f : 'a -> 'b option
```



into this

```
'a option -> 'b option
```



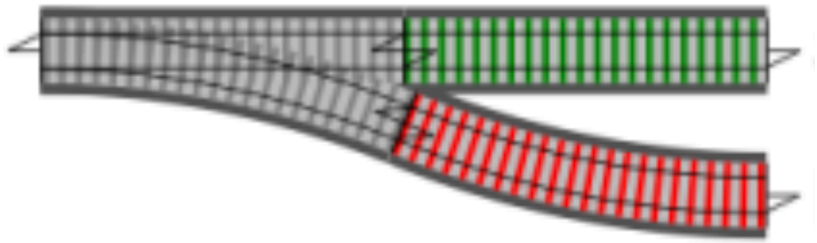
The following function does that

```
let bind f = fun x ->
    match x with
    | Some a -> f a
    | None   -> None
```

Turn switches into two-track functions

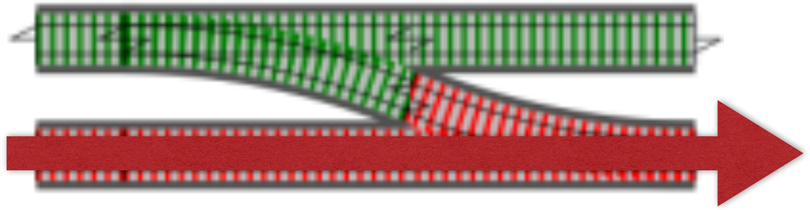
We want to turn

```
f : 'a -> 'b option
```



into this

```
'a option -> 'b option
```



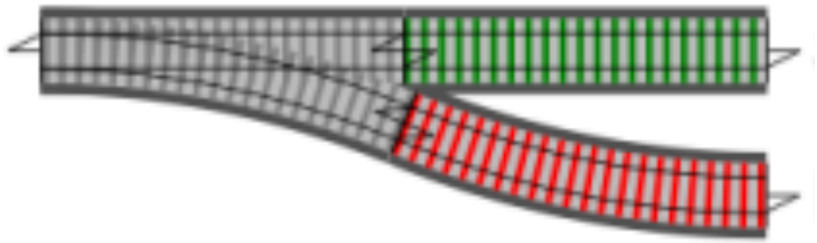
The following function does that

```
let bind f = fun x ->
    match x with
green track: | Some a -> f a
  red track: | None   -> None
```

Turn switches into two-track functions

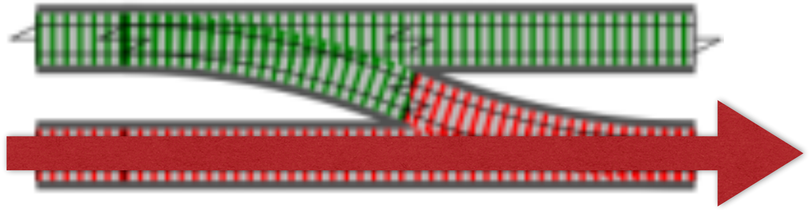
We want to turn

```
f : 'a -> 'b option
```



into this

```
'a option -> 'b option
```



The following function does that

```
let bind f = fun x ->
    match x with
green track: | Some a -> f a
  red track: | None   -> None
```

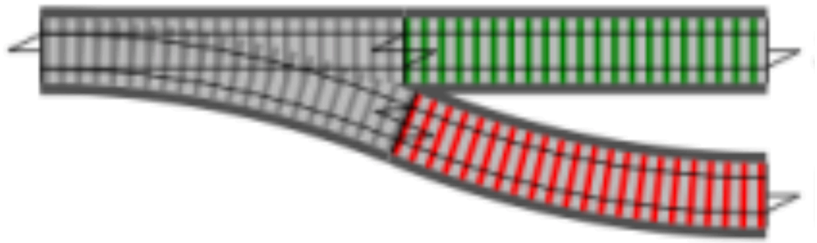
In practice, we use this variant:

```
let (>>=) x f = bind f x

(>>=) : 'a option ->
        ('a -> 'b option) ->
        'b option
```

Turn switches into two-track functions

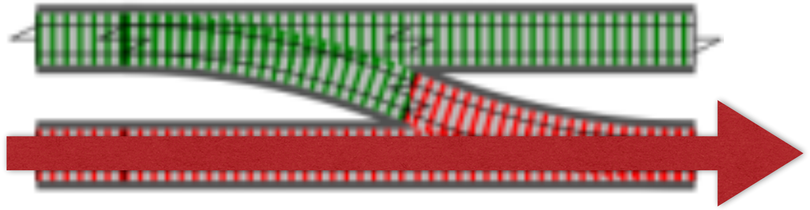
We want to turn

```
f : 'a -> 'b option
```



into this

```
'a option -> 'b option
```



The following function does that

```
let bind f = fun x ->
    match x with
green track: | Some a -> f a
  red track: | None   -> None
```

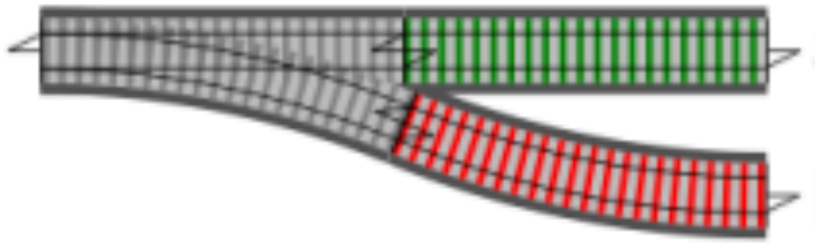
In practice, we use this variant:

```
let (>>=) x f = bind f x

(>>=) : 'a option ->
        ('a -> 'b option) ->
        'b option
```

Turn switches into two-track functions

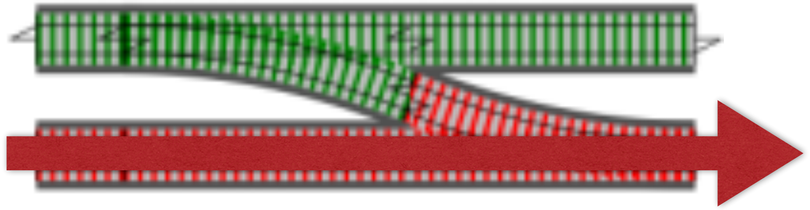
We want to turn

```
f : 'a -> 'b option
```



into this

```
'a option -> 'b option
```



Direct definition of (>>=) :

```
let (>>=) x f =
    match x with
green track: | Some a -> f a
  red track: | None   -> None
```

In practice, we use this variant:

```
let (>>=) x f = bind f x
```

```
(>>=) : 'a option ->
        ('a -> 'b option) ->
        'b option
```

Example

Let’s go back to our `binop` function

```
let binop f a b s =
    match a s, b s with
    | Some x,
      Some y -> Some (f x y)
    | _, _   -> None


ret : 'a -> 'a option

let ret x = Some x
```

Example

Let’s go back to our `binop` function

```
let binop f a b s =
    match a s, b s with
    | Some x,
      Some y -> Some (f x y)
    | _, _   -> None
```

use ret

```
let binop f a b s =
    match a s, b s with
    | Some x,
      Some y -> ret (f x y)
    | _, _   -> None
```

```
ret : 'a -> 'a option

let ret x = Some x
```

Example

Let's go back to our `binop` function

```
let binop f a b s =
    match a s, b s with
    | Some x,
      Some y -> ret (f x y)
    | _, _   -> None
```

```
ret : 'a -> 'a option

let ret x = Some x
```

```
(>>=) : 'a option ->
        ('a -> 'b option) ->
        'b option
let (>>=) x f = match x with
                | Some a -> f a
                | None   -> None
```

Example

Let's go back to our `binop` function

```
let binop f a b s =
    match a s, b s with
    | Some x,
      Some y -> ret (f x y)
    | _, _   -> None
```

use >>=

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

```
ret : 'a -> 'a option

let ret x = Some x
```

```
(>>=) : 'a option ->
        ('a -> 'b option) ->
        'b option
let (>>=) x f = match x with
                | Some a -> f a
                | None   -> None
```

Example

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

Example

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)

let rec eval e =
    match e with
    | N n           -> fun _ -> Some n
    | V x           -> Map.tryFind x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
      fun s -> match eval a1 s, eval a2 s with
                | Some x, Some y when y <> 0 -> Some (x / y)
                | _, _                       -> None
```

Example

```
let binop f a b s =
   a s >>= fun x ->
   b s >>= fun y ->
   ret (f x y)

let rec eval e =
    match e with
    | N n           -> fun _ -> ret n
    | V x           -> Map.tryFind x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
        eval a1 s >>= fun x ->
        eval a2 s >>= fun y ->
          if y <> 0 then ret (x / y) else None
```

Example

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)

let rec eval e =
    match e with
    | N n           -> fun _ -> ret n
    | V x           -> Map.tryFind x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
        eval a1 s >>= fun x ->
        eval a2 s >>= fun y ->
          if y <> 0 then ret (x / y) else None
```

Note

>>= is our 'programmable semicolon'

What did we gain here?!

- Abstracting away from failure case is useful
- We only have to write the code for error handling once
- In the code for eval we can focus on the “happy path”
- Much easier to add more features later

The big picture

- Next: We extend our “railway” with
  (1) error messages and (2) state.
- We only need to change its type (`a -> 'b option`), and the two operations `ret` & `>>=`
- Our code in the eval function needs little or no change!

- At the very end: We improve the syntax using computation expressions

Questions?

Part II

Error Messages

Adding error messages

Let’s modify our evaluator to return an error message on failure rather than just None

```
type Error =
    | VarNotFound of string
    | DivisionByZero


type Result<'a> =
    | Success of 'a
    | Failure of Error
```

Adding error messages

Let's modify our evaluator to return an error message on failure rather than just None

```
type Error =
    | VarNotFound of string
    | DivisionByZero
```

```
type Result<'a> =
    | Success of 'a
    | Failure of Error
```

```
type 'a option =
    | Some of 'a
    | None
```

Result is similar to option

Adding error messages

Let's modify our evaluator to return an error message on failure rather than just None

```
type Error =
    | VarNotFound of string
    | DivisionByZero
```

```
type Result<'a> =
    | Success of 'a
    | Failure of Error
```

```
type 'a option =
    | Some of 'a
    | None
```

Result is similar to option

Adding error messages

Let's start with the `binop` helper function (without `>>=` and `ret`)

```
let binop f a b s =
    match a s, b s with
    | Some x, Some y -> Some (f x y)
    | _, _           -> None
```

```
let binop f a b s =
    match a s, b s with
    | Success x, Success y -> Success (f x y)
    | Failure e, _         -> Failure e
    | _, Failure e         -> Failure e
```

"Standard" implementation

```
let rec eval e =
    match e with
    | N n -> fun _ -> Success n
    | V x ->
        fun s ->
        match Map.tryFind x s with
        | Some v -> Success v
        | None -> Failure (VarNotFound x)
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
        fun s -> match eval a1 s, eval a2 s with
                 | Success x, Success y when y <> 0 -> Success (x / y)
                 | Success x, Success y -> Failure DivisionByZero
                 | Failure e, _         -> Failure e
                 | _, Failure e         -> Failure e
```

"Standard" implementation

```
let rec eval e =
    match e with
    | N n -> fun _ -> Success n
    | V x ->
        fun s ->
        match Map
        | Some v
        | None ->
    | Add (a1, a2
    | Sub (a1, a2
    | Mul (a1, a2
    | Div (a1, a2,
        fun s -> match eval a1 s, eval a2 s with
                 | Success x, Success y when y <> 0 -> Success (x / y)
                 | Success x, Success y -> Failure DivisionByZero
                 | Failure e, _         -> Failure e
                 | _, Failure e         -> Failure e
```

This feels like several steps backwards

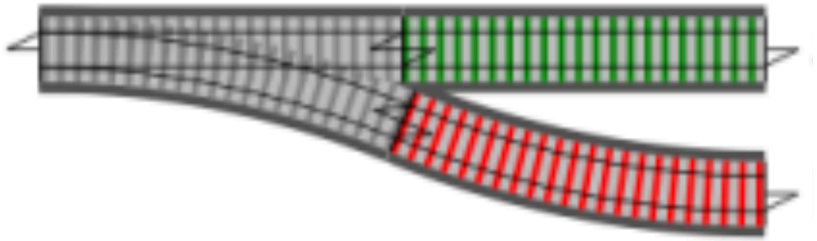
Instead, we will modify >>= and `ret` to work with `Result` rather than `option`

Constant switches

The functions turn out to be almost identical

Options

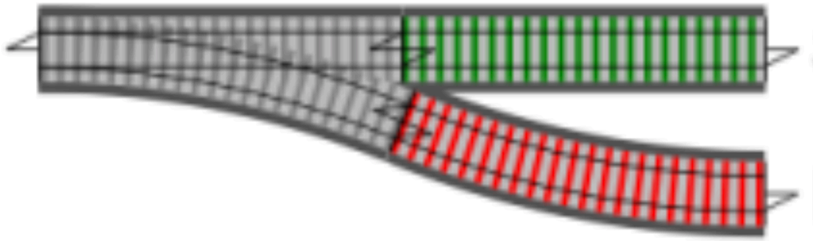
```
'a -> 'a option
```



```
let ret x = Some x

ret : 'a -> 'a option
```

Results

```
'a -> Result<'a>
```



```
let ret x = Success x

ret : 'a -> Result<‘a>
```

Turn switches into two-track functions

Options

```
let (>>=) x f =
    match x with
    | Some a -> f a
    | None   -> None
```

```
(>>=) : 'a option ->
        ('a -> 'b option) ->
        'b option
```

Results

```
let (>>=) x f =
    match x with
    | Success a -> f a
    | Failure e -> Failure e
```

```
(>>=) : Result<'a> ->
        ('a -> Result<'b>) ->
        Result<'b>
```

Comparing the binary operators

The code for the binary operators is identical

Options

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

Results

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

Comparing the binary operators

The code for the binary operators is identical

Options

```
let binop f a b s =
   a s >>= fun x ->
   b s >>= fun y ->
   ret (f x y)
```

Results

```
let binop f a b s =
   a s >>= fun x ->
   b s >>= fun y ->
   ret (f x y)
```

But their types differ

```
binop : ('a  -> 'b -> 'c) ->
        (state -> 'a option) ->
        (state -> 'b option) ->
        state -> 'c option
```

```
binop : ('a  -> 'b -> 'c) ->
        (state -> Result<‘a>) ->
        (state -> Result<‘b>) ->
        state -> Result<‘c>
```

Comparing the binary operators

The code for the binary operators is identical

Options

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

Results

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

```
type state =
  Map<string, int>
```

But their types differ

```
binop : ('a  -> 'b -> 'c) ->
        (state -> 'a option) ->
        (state -> 'b option) ->
        state -> 'c option
```

```
binop : ('a  -> 'b -> 'c) ->
        (state -> Result<‘a>) ->
        (state -> Result<‘b>) ->
        state -> Result<‘c>
```

Evaluator with error handling

```
let rec eval e =
    match e with
    | N n -> fun _ -> ret n
    | V x -> fun s ->
        match Map.tryFind x s with
        | Some v -> ret v
        | None -> Failure (VarNotFound x)
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
            eval a1 s >>= fun x ->
            eval a2 s >>= fun y ->
            if y <> 0 then ret (x / y) else Failure DivisionByZero
```

Evaluator with error handling

```
let rec eval e =
    match e with
    | N n -> fun _ -> ret n
    | V x -> fun s ->
        match Map.tryFind x s with
        | Some v -> ret v
        | None -> Failure (VarNotFound x)
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
            eval a1 s >>= fun x ->
            eval a2 s >>= fun y ->
            if y <> 0 then ret (x / y) else Failure DivisionByZero
```

The only place where we needed to change our code

Are we there yet?

```
let rec eval e =
    match e with
    | N n -> fun _ -> ret n
    | V x -> fun s ->
        match Map.tryFind x s with
        | Some v -> ret v
        | None -> Failure (VarNotFound x)
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
            eval a1 s >>= fun x ->
            eval a2 s >>= fun y ->
            if y <> 0 then ret (x / y) else Failure DivisionByZero
```

Are we there yet?

```
let rec eval e =
    match e with
    | N n -> fun _ -> ret n
    | V x -> fun s ->
        match Map.tryFind x s with
        | Some v -> ret v
        | None -> Failure (VarNotFound x)
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
          eval a1 s >>= fun x ->
          eval a2 s >>= fun y ->
          if y <> 0 then ret (x / y) else Failure DivisionByZero
```

If we look at our code now it still has a few flaws.

- The state is passed through the program
- Lookups are a bit clunky
- What about modifying the state?

Are we there yet?

```
let rec eval e =
    match e with
    | N n -> fun _ -> ret n
    | V x -> fun s ->
        match Map.tryFind x s with
        | Some v -> ret v
        | None -> Failure (VarNotFound x)
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) -> fun s ->
          eval a1 s >>= fun x ->
          eval a2 s >>= fun y ->
          if y <> 0 then ret (x / y) else Failure DivisionByZero
```

If we look at our code now it still has a few flaws.

- The state is passed through the program
- Lookups are a bit clunky
- What about modifying the state?

We will address this in the next step.

Questions?

Part III

Maintaining State

Adding state

Initially we worked on option types

```
'a option
```

We then switched to Result types
to distinguish different errors

```
Result<'a>
```

We now want a function that takes
a state and also returns a Result
with the updated state

```
state -> Result<'a * state>
```

Adding state

Initially we worked on option types

```
'a option
```

We then switched to Result types to distinguish different errors

```
Result<'a>
```

We now want a function that takes a state and also returns a Result with the updated state

```
state -> Result<'a * state>
```

Intuition

Now, all switches must also pass along additional data (namely the state).

Adding state

Initially we worked on option types

```
'a option
```

We then switched to Result types to distinguish different errors

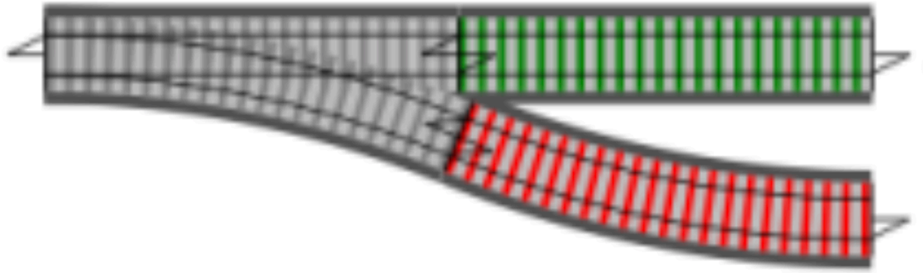
```
Result<'a>
```

We now want a function that takes a state and also returns a Result with the updated state

```
state -> Result<'a * state>
```

Intuition

Now, all switches must also pass along additional data (namely the state).

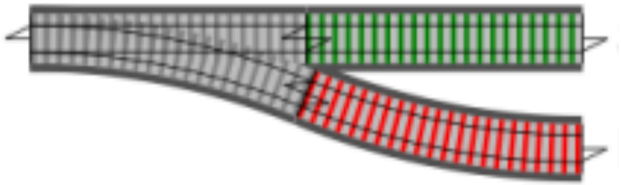


Constant switches

ret now also passes the state data through

Options

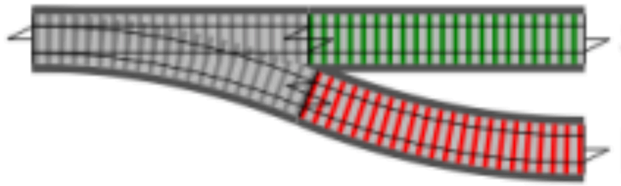
```
'a -> 'a option
```



```
let ret x = Some x

ret : 'a ->
      'a option
```

Results

```
'a -> Result<'a>
```



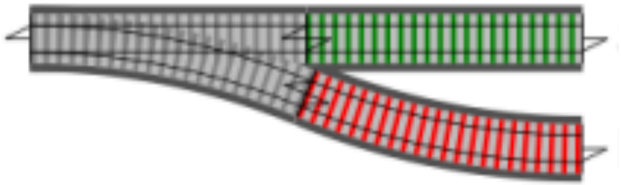
```
let ret x = Success x

ret : 'a -> Result<'a>
```

Constant switches

ret now also passes the state data through

Options

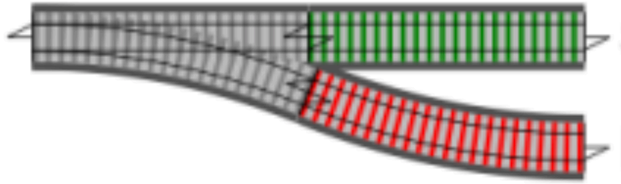
```
'a -> 'a option
```



```
let ret x = Some x

ret : 'a ->
      'a option
```

Results

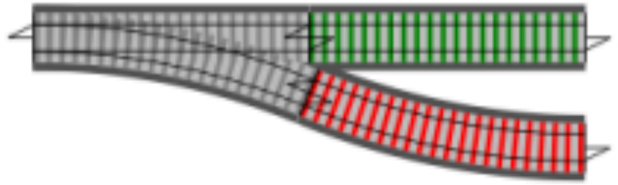
```
'a -> Result<'a>
```



```
let ret x = Success x

ret : 'a -> Result<'a>
```

Results + State

```
'a -> (state ->
       Result<‘a * state>)
```



```
let ret x =
    fun s -> Success (x, s)

ret : 'a -> state ->
      Result<‘a * state>
```

Turn switches into two-track functions

Options

```
let (>>=) a f =
  match a with
  | Some x -> f x
  | None   -> None
```

```
(>>=) :
   'a option ->
   ('a ->
    'b option) ->
   'b option
```

Results

```
let (>>=) a f =
  match a with
  | Success x -> f x
  | Failure e ->
      Failure e
```

```
(>>=) :
   Result<'a> ->
   ('a ->
    Result<'b>) ->
   Result<'b>
```

Results + State

```
let (>>=) a f = fun s ->
  match a s with
  | Success (x, s1) -> f x s1
  | Failure e -> Failure e
```

```
(>>=) :
  (state ->
   Result<'a * state>) ->
  ('a -> state ->
   Result<'b * state>) ->
  state -> Result<'b * state>
```

Comparing the binary operators

The code for the binary operators is now **much** cleaner

Results

```
let binop f a b s =
   a s >>= fun x ->
   b s >>= fun y ->
   ret (f x y)
```

```
binop : ('a  -> 'b -> 'c) ->
        (state -> Result<'a>)
        (state -> Result<'b>)
        state -> Result<‘c>
```

Results + State

```
let binop f a b =
   a >>= fun x ->
   b >>= fun y ->
   ret (f x y)
```

```
binop : ('a  -> 'b -> 'c) ->
    (state -> Result<‘a * state>)
    (state -> Result<‘b * state>)
    state -> Result<‘c * state>
```

Comparing the binary operators

The code for the binary operators is now **much** cleaner

Results

```
let binop f a b s =
    a s >>= fun x ->
    b s >>= fun y ->
    ret (f x y)
```

```
binop : ('a
        (state
        (state
        state -> Result< c>
```

Results + State

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
c) ->
 * state>)
 * state>)
state -> Result< c * state>
```

Not having to thread around the state, and supporting updated states is a big deal, so the term "**much** cleaner" is warranted

Updating the state

We create the following switches to update the state

```
let find x = fun s ->
  match Map.tryFind x s with
  | Some v -> Success (v, s)
  | None -> Failure (VarNotFound x)
```

```
let set x v = fun s ->
    Success ((), (Map.add x v s))
```

```
find : string ->
  (state -> Result<int * state>)
```

```
set : string -> int ->
  (state -> Result<unit * state>)
```

Updating the state

We create the following switches to update the state

```
let find x = fun s ->
  match Map.tryFind x s with
  | Some v -> Success (v, s)
  | None -> Failure (VarNotFound x)
```

```
let set x v = fun s ->
    Success ((), (Map.add x v s))
```

`set` returns a unit; updating the state can be seen as a side effect. It is, however, perfectly functional

```
find : string ->
  (state -> Result<int * state>)
```

```
set : string -> int ->
  (state -> Result<unit * state>)
```

Updating the state

We create the following switches to update the state

```
let find x = fun s ->
  match Map.tryFind x s with
  | Some v -> Success (v, s)
  | None -> Failure (VarNotFound x)
```

Note how both functions return results + state and the these can be used in binds

```
find : string ->
  (state -> Result<int * state>)
```

```
let set x v = fun s ->
    Success ((), (Map.add x v s))
```

`set` returns a unit; updating the state can be seen as a side effect. It is, however, perfectly functional

```
set : string -> int ->
  (state -> Result<unit * state>)
```

Error signalling

Let’s also write a function for signalling errors

```
let fail e = fun s -> Failure e

fail : Error ->
       state -> Result<'a * state>
```

Now we can just write `fail e` to signal errors, without bothering passing along the state.

Evaluator with stateful railways

```
let rec eval e =
    match e with
    | N n -> ret n
    | V x -> find x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
        eval a1 >>= fun x ->
        eval a2 >>= fun y ->
        if y <> 0 then ret (x / y) else fail DivisionByZero
```

Evaluator with stateful railways

```
let rec eval e =
    match e with
    | N n -> ret n
    | V x -> find x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
        eval a1 >>= fun x ->
        eval a2 >>= fun y ->
        if y <> 0 then ret (x / y) else fail DivisionByZero
```

Let's extend the expression language with an increment operator (like ++ in Java, C#)

```
type aExp =
    …
| Inc of string
```

Evaluator with stateful railways

```
let rec eval e =
    match e with
    | N n -> ret n
    | V x -> find x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
        eval a1 >>= fun x ->
        eval a2 >>= fun y ->
        if y <> 0 then ret (x / y) else fail DivisionByZero
    | Inc x ->
        find x >>= fun v ->
        set x (v + 1) >>= fun () ->
        ret (v + 1)
```

Evaluator with stateful railways

```
let rec eval e =
    match e with
    | N n -> ret n
    | V x -> find x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
        eval a1 >>= fun x ->
        eval a2 >>= fun y ->
        if y <> 0 then ret (x / y) else fail DivisionByZero
    | Inc x ->
        find x >>= fun v ->
        set x (v + 1) >>= fun () ->
        ret (v + 1)
```

The state (`Map<string,int>`) is still not mentioned, even though we read and write it

Evaluator with stateful railways

```
| Inc x ->
    find x >>= fun v ->
    set x (v + 1) >>= fun () ->
    ret (v + 1)
```

Evaluator with stateful railways

```
| Inc x ->
    find x >>= fun v ->
    set x (v + 1) >>= fun () ->
    ret (v + 1)
```

Evaluator with stateful railways

This is a common pattern, so we define a specialised variant of >>= for it:

```
let (>>>=) a b = a >>= (fun () -> b)
```

```
  | Inc x ->
      find x >>= fun v ->
      set x (v + 1) >>= fun () ->
      ret (v + 1)
```

Evaluator with stateful railways

This is a common pattern, so we define a specialised variant of >>= for it:

```
let (>>>=) a b = a >>= (fun () -> b)
```

```
  | Inc x ->
      find x >>= fun v ->
      set x (v + 1) >>>=
      ret (v + 1)
```

Evaluator with stateful railways

```
let rec eval e =
    match e with
    | N n -> ret n
    | V x -> find x
    | Add (a1, a2) -> binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) -> binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) -> binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
        eval a1 >>= fun x ->
        eval a2 >>= fun y ->
        if y <> 0 then ret (x / y) else fail DivisionByZero
    | Inc x ->
        find x >>= fun v ->
        set x (v + 1) >>>=
        ret (v + 1)
```

Compare the type of >>= & ret

Option

```
ret :
  'a -> 'a option


(>>=) :
  'a option ->
  ('a ->
   'b option) ->
  'b option
```

Result

```
ret :
  'a -> Result<'a>


(>>=) :
  Result<'a> ->
  ('a ->
   Result<'b>) ->
  Result<'b>
```

Result + State

```
ret :
  'a ->
  state -> Result<'a * state>

(>>=) :
  state -> Result<'a * state> ->
  ('a ->
   state -> Result<'b * state>) ->
  state -> Result<'b * state>
```

Compare the type of >>= & ret

Option

```
ret :
  'a -> 'a option


(>>=) :
  'a option ->
  ('a ->
   'b option  ->
  'b option
```

Result

```
ret :
  'a -> Result<'a>


(>>=) :
  Result<'a> ->
  ('a ->
   Result<'b>) ->
  Result<'b>
```

Result + State

```
ret :
  'a ->
  state -> Result<'a * state>

(>>=) :
  state -> Result<'a * state> ->
  ('a ->
   state -> Result<'b * state>) ->
  state -> Result<'b * state>
```

Compare the type of >>= & ret

Option

```
ret :
  'a -> 'a option


(>>=) :
  'a option ->
  ('a ->
   'b option  ->
  'b option
```

Result

```
ret :
  'a -> Result<'a>


(>>=) :
  Result<'a> ->
  ('a ->
   Result<'b>) ->
  Result<'b>
```

Result + State

```
ret :
  'a ->
  state -> Result<'a * state>

(>>=) :
  state -> Result<'a * state> ->
  ('a ->
   state -> Result<'b * state>) ->
  state -> Result<'b * state>
```

Compare the type of >>= & ret

Option

```
ret :
  'a -> 'a option


(>>=) :
  'a option ->
  ('a ->
   'b option)  ->
  'b option
```

Result

```
ret :
  'a -> Result<'a>


(>>=) :
  Result<'a> ->
  ('a ->
   Result<'b>) ->
  Result<'b>
```

Result + State

```
ret :
  'a ->
  state -> Result<'a * state>


(>>=) :
  state -> Result<'a * state> ->
  ('a ->
   state -> Result<'b * state>) ->
  state -> Result<'b * state>
```

Compare the type of >>= & ret

Option

```
ret :
  'a -> M<'a>


(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Result

```
ret :
  'a -> M<'a>


(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Result + State

```
ret :
  'a -> M<'a>


(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Compare the type of >>= & ret

Option

```
M<'a> = 'a option

ret :
  'a -> M<'a>

(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Result

```
M<'a> = Result<'a>

ret :
  'a -> M<'a>

(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Result + State

```
M<'a> = state -> Result<'a * state>

ret :
  'a -> M<'a>

(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Compare the type of >>= & ret

They are **exactly** the same

```
ret :
  'a -> M<'a>
```

```
(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

Compare the type of >>= & ret

They are **exactly** the same

```
ret :
  'a -> M<'a>


(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

A type M<'a> with these functions is called a monad*

Compare the type of >>= & ret

They are **exactly** the same

```
ret :
  'a -> M<'a>


(>>=) :
  M<'a> ->
  ('a ->
   M<'b>) ->
  M<'b>
```

A type M<'a> with these functions is called a monad*

* Conditions apply; see next slides

Getting to monads

- Railway-oriented programming offers a **versatile abstraction**
- We only need 2 ingredients:
  - `>>= : M<'a> -> (a -> M<'b>) -> M<'b>`
  - `ret : 'a -> M<'a>`

Getting to monads

- Railway-oriented programming offers a **versatile abstraction**
- We only need 2 ingredients:
  - `>>= : M<'a> -> (a -> M<'b>) -> M<'b>`
  - `ret : 'a -> M<'a>`
- Technically, `>>=` and ret need to satisfy the **monad laws**

Monad Laws

```
    ret a >>= f  =  f a                              (1)

      m >>= ret  =  m                                (2)

(m >>= f) >>= g  =  m >>= (fun x -> f x >>= g)       (3)
```

If these equations are satisfied,
then `>>=` and `ret` form a monad.

Monad Laws

```
    ret a >>= f  =  f a                          (1)

      m >>= ret  =  m                            (2)

(m >>= f) >>= g  =  m >>= (fun x -> f x >>= g)   (3)
```

I only mention them for the sake of completion.

You don't need to know them.

Function composition

```
        id >> f   =   f                    (1)

        f >> id   =   f                    (2)

(h >> f) >> g   =   h >> (f >> g)          (3)
```

Monad laws correspond to these laws of function composition (>>) and the identity function (id).

Questions?

Part IV

Computation Expressions

Computation expressions

- **Ergonomic syntax** to work with monads (and similar structures) more easily.
- **Extensible** by the user.

So we can write this:

```
let binop f a b =
  ev { let! x = a
       let! y = b
       return f x y }
```

Instead of this:

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

Computation expressions

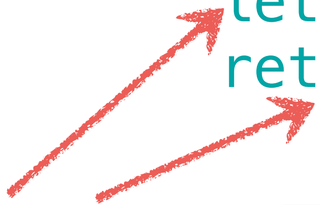
- **Ergonomic syntax** to work with monads (and similar structures) more easily.
- **Extensible** by the user.

So we can write this:

```
let binop f a b =
  ev { let! x = a
       let! y = b
       return f x y }
```

Instead of this:

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```



These are F# keywords (but you define what they mean).

The binop-operator revisited

Without computation expr.

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

The binop-operator revisited

Without computation expr.

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

These names must match

The binop-operator revisited

Our last binop operation

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

Monadic bind

The binop-operator revisited

Our last binop operation

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Monadic return

The binop-operator revisited

Our last binop operation

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

Useful for returning failure values (keyword return!)

The binop-operator revisited

Our last binop operation

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

```
let binop f a b =
  ev.Bind (a, fun x ->
  ev.Bind (b, fun y ->
  ev.Return(f x y)))
```

The binop-operator revisited

Our last binop operation

```
let binop f a b =
    a >>= fun x ->
    b >>= fun y ->
    ret (f x y)
```

```
type MyResultBuilder() =
    member bld.Bind (a, f)  =
        a >>= f
    member bld.Return e     =
        ret e
    member bld.ReturnFrom e =
        e
let ev = MyResultBuilder()
```

Computation expressions

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

Syntactic sugar for this

```
let binop f a b =
    ev.Bind (a, fun x ->
    ev.Bind (b, fun y ->
    ev.Return(f x y)))
```

Translations into standard syntax

Translation function T translates computational expressions to standard syntax

```
T(return v)          = Return v

T(return! e)         = ReturnFrom e

T(let! x = e
  ce)                = Bind(e, fun x -> T(ce))

T(do! e
  ce)                = Bind(e, fun () -> T(ce))
```

Example

```
T(return v)  = Return v
T(let! x = e
  ce)        = Bind(e, fun x -> T(ce))
```

```
let binop f a b =
    ev { let! x = a
         let! y = b
         return f x y }
```

Example

```
T(return v)  = Return v
T(let! x = e
  ce)        = Bind(e, fun x -> T(ce))
```

```
let binop f a b =
    T ( let! x = a
        let! y = b
        return f x y )
```

Example

```
T(return v)  = Return v
T(let! x = e
  ce)        = Bind(e, fun x -> T(ce))
```

```
let binop f a b =
      Bind(a, fun x ->
       T (let! y = b
          return f x y))
```

Example

```
T(return v)  = Return v
T(let! x = e
  ce)        = Bind(e, fun x -> T(ce))
```

```
let binop f a b =
      Bind(a, fun x ->
      Bind(b, fun y ->
       T (return f x y)))
```

Example

```
T(return v)  = Return v
T(let! x = e
  ce)        = Bind(e, fun x -> T(ce))
```

```
let binop f a b =
      Bind(a, fun x ->
      Bind(b, fun y ->
      Return (f x y)))
```

Example

```
T(return v)  = Return v
T(let! x = e
  ce)        = Bind(e, fun x -> T(ce))
```

```
let binop f a b =
      a >>= fun x ->
      b >>= fun y ->
      ret (f x y)
```

Evaluator w/ computation expressions

```
let binop f a b =
  ev { let! x = a
       let! y = b
       return f x y }

let div a b = ev {
  let! x = a
  let! y = b
  if y <> 0 then
    return (x / y)
  else
    return! fail DivisionByZero }
```

```
let rec eval e =
    match e with
    | N n -> ev {return n}
    | V x -> find x
    | Add (a1, a2) ->
      binop ( + ) (eval a1) (eval a2)
    | Sub (a1, a2) ->
      binop ( - ) (eval a1) (eval a2)
    | Mul (a1, a2) ->
      binop ( * ) (eval a1) (eval a2)
    | Div (a1, a2) ->
      div (eval a1) (eval a2)
    | Inc x ->
      ev { let! v = find x
           do!  set x (v + 1)
           return v + 1 }
```

Evaluator as one big ev{…}

```
let rec eval (a : aExp) : state -> Result<int * state> = ev {
    match a with
      | N n -> return n
      | V x -> return! find x
      | Add (a1, a2) -> return! binop ( + ) (eval a1) (eval a2)
      | Sub (a1, a2) -> return! binop ( - ) (eval a1) (eval a2)
      | Mul (a1, a2) -> return! binop ( * ) (eval a1) (eval a2)
      | Div (a1, a2) ->
        let! x = eval a1
        let! y = eval a2
        if y <> 0 then return (x / y)
        else return! fail DivisionByZero
      | Inc x ->
        let! v = find x
        do!  set x (v + 1)
        return v + 1 }
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}

val it : seq<int> = seq [1; 4; 9; 16; …]
```

- Sequence expressions are a special case of **computation expressions**.
- Uses For & Yield Instead of Bind & Return

The computation expression for sequences is defined like this:

```
type MySeqBuilder() =
  member this.Yield x = …
  …
let seq = MySeqBuilder()
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

The above syntax is translated as follows:

```
T(yield v)                  = Yield v

T(yield! e)                 = YieldFrom e

T(for x in e do ce)         = For(e, fun x -> T(ce))
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

The above syntax is translated as follows:

T is the function that does the translation.

```
T(yield v)                  = Yield v

T(yield! e)                 = YieldFrom e

T(for x in e do ce)         = For(e, fun x -> T(ce))
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

In addition, computation expressions
for sequences use the function Delay

```
T(seq {ce})      = Delay (fun () -> T (ce))
```

Sequence expressions

```
> seq {for x in 1..10 do yield x * x}
```

In addition, computation expressions for sequences use the function `Delay`

```
T(seq {ce})      = Delay (fun () -> T (ce))
```

This makes sure that sequences defined by `seq {…}` are delayed by default (⇒ can be used in recursive functions)

Summary

- The **railway** approach (a.k.a. **monads**) allows you to encapsulate behind-the-scenes boilerplate code (handling errors, reading/updating state)
- Then you can **focus on the main logic** in your code (e.g. evaluating an expression)
- **Computation expressions** provide a nicer syntax to do all of this.
- Computation expressions are also used for **sequences** and for **asynchronous** programming (see later lecture)