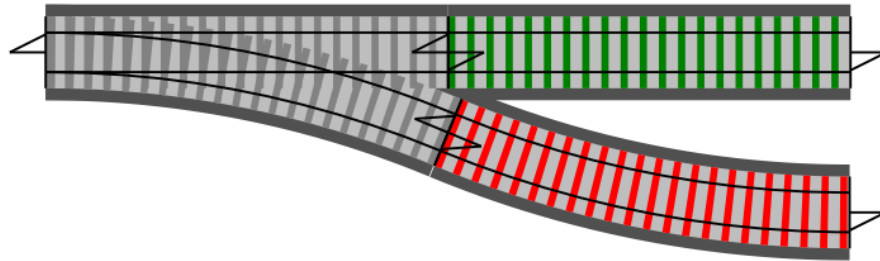


Modelling errors



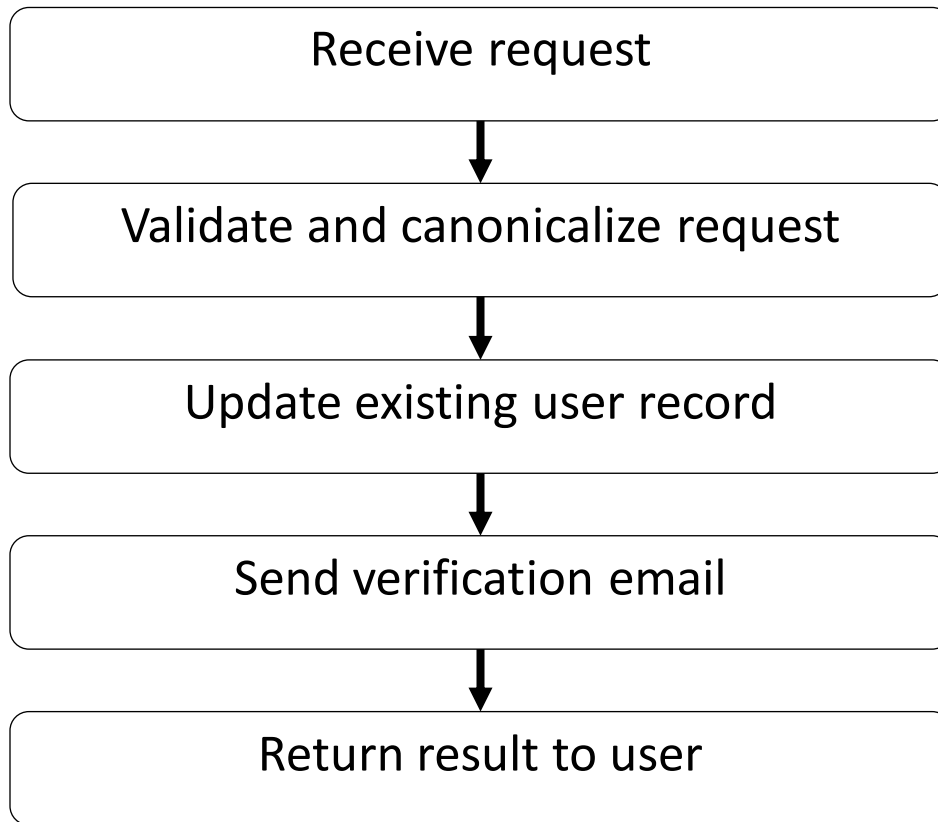
What do railways
have to do with
programming?

Happy path programming

Implementing a simple use case



"As a user I want to update my name and email address"

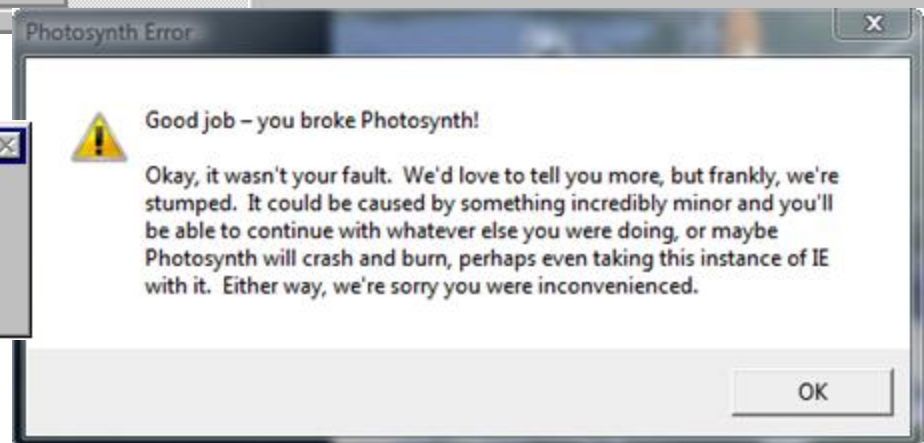


```
type Request = {  
  userId: int;  
  name: string;  
  email: string };
```

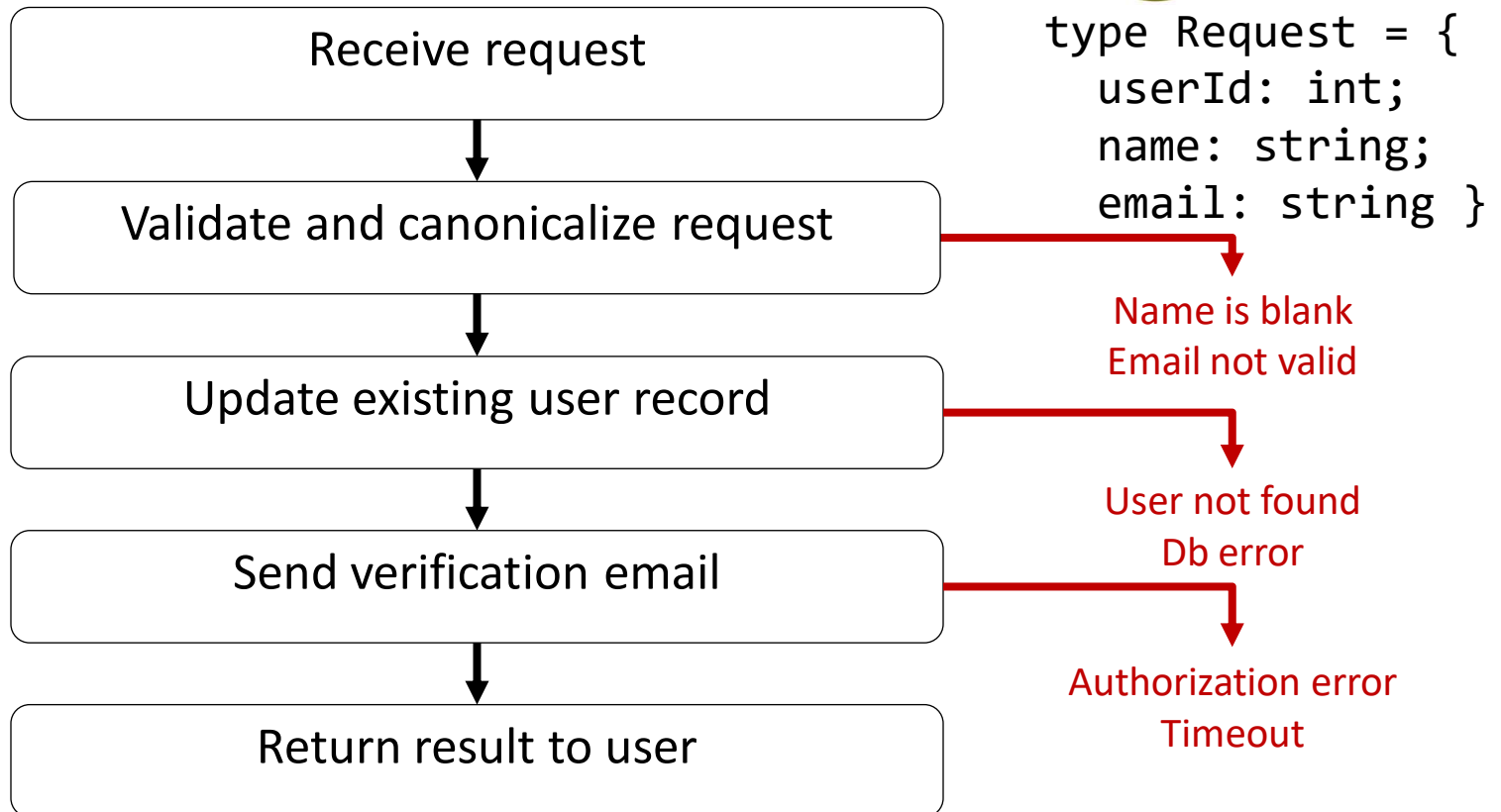
Straying from the happy path...

What do you do when
something goes wrong?





"As a user I want to update my name and email address"
- and see sensible error messages when something goes wrong!



```
string UpdateCustomerWithErrorHandling()  
{  
    var request = receiveRequest();  
    validateRequest(request);  
    canonicalizeEmail(request);  
    db.updateDbFromRequest(request);  
    smtpServer.sendEmail(request.Email)  
  
    return "OK";  
}
```



```
string UpdateCustomerWithErrorHandling()
{
    var request = receiveRequest();
    var isValidated = validateRequest(request);
    if (!isValidated) {
        return "Request is not valid"
    }
    canonicalizeEmail(request);
    db.updateDbFromRequest(request);
    smtpServer.sendEmail(request.Email)

    return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
    var request = receiveRequest();
    var isValidated = validateRequest(request);
    if (!isValidated) {
        return "Request is not valid"
    }
    canonicalizeEmail(request);
    var result = db.updateDbFromRequest(request);
    if (!result) {
        return "Customer record not found"
    }

    smtpServer.sendEmail(request.Email)

    return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
    var request = receiveRequest();
    var isValidated = validateRequest(request);
    if (!isValidated) {
        return "Request is not valid"
    }
    canonicalizeEmail(request);
    try {
        var result = db.updateDbFromRequest(request);
        if (!result) {
            return "Customer record not found"
        }
    } catch {
        return "DB error: Customer record not updated"
    }

    smtpServer.sendEmail(request.Email)

    return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
    var request = receiveRequest();
    var isValidated = validateRequest(request);
    if (!isValidated) {
        return "Request is not valid"
    }
    canonicalizeEmail(request);
    try {
        var result = db.updateDbFromRequest(request);
        if (!result) {
            return "Customer record not found"
        }
    } catch {
        return "DB error: Customer record not updated"
    }

    if (!smtpServer.sendEmail(request.Email)) {
        log.Error "Customer email not sent"
    }

    return "OK";
}
```

```

string UpdateCustomerWithErrorHandling()
{
    var request = receiveRequest();
    var isValidated = validateRequest(request);
    if (!isValidated) {
        return "Request is not valid"
    }
    canonicalizeEmail(request);
    try {
        var result = db.updateDbFromRequest(request);
        if (!result) {
            return "Customer record not found"
        }
    } catch {
        return "DB error: Customer record not updated"
    }

    if (!smtpServer.sendEmail(request.Email)) {
        log.Error "Customer email not sent"
    }

    return "OK";
}

```

Q: What is the functional equivalent of this code?

... and can we preserve the elegance of the original functional version?

↳ clean lines -> 18 ugly lines. 200% extra!
 Sadly this is typical of error handling code.

Use a *Result* type
for error handling



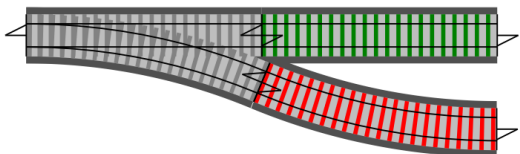
```
type Result =  
  | Ok of SuccessValue  
  | Error of ErrorValue
```

Define a choice type

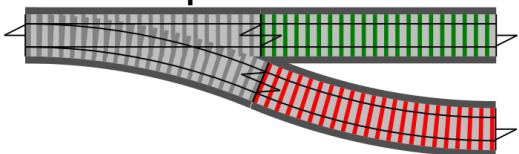


```
let validateInput input =  
  if input.name = "" then  
    Error "Name must not be blank"  
  else if input.email = "" then  
    Error "Email must not be blank"  
  else  
    Ok input // happy path
```

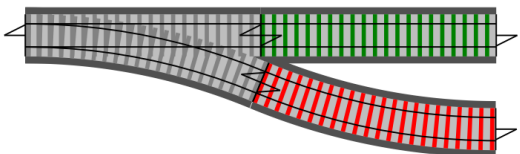

Validate

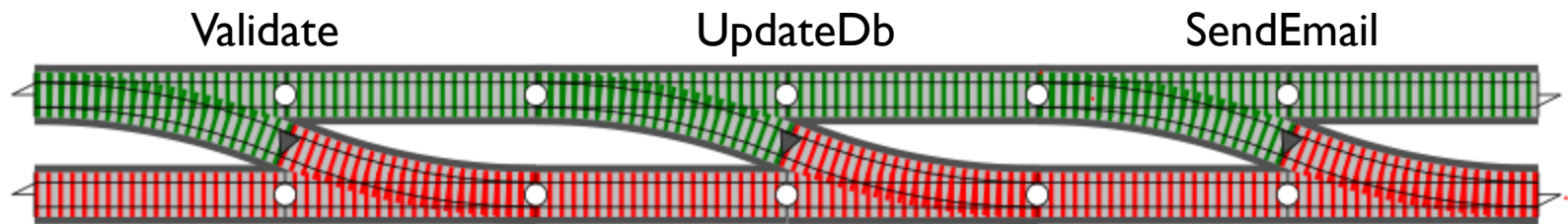


UpdateDb



SendEmail





This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

Functional flow without error handling

Before

```
let updateCustomer =  
  receiveRequest()  
  |> validateRequest  
  |> canonicalizeEmail  
  |> updateDbFromRequest  
  |> sendEmail  
  |> returnMessage
```

One track

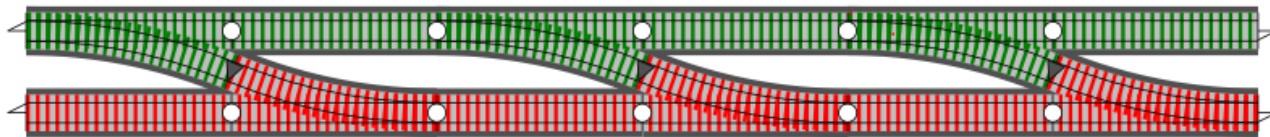


Functional flow with error handling

After

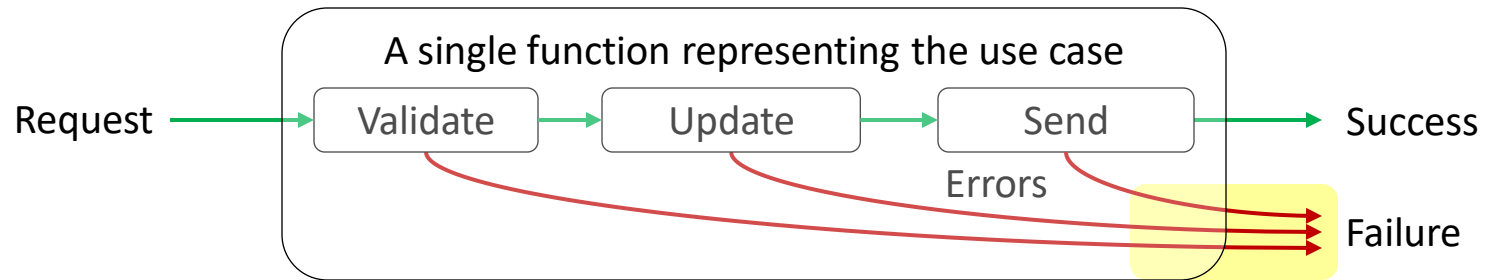
```
let updateCustomerWithErrorHandling =  
  receiveRequest()  
  |> validateRequest  
  |> canonicalizeEmail  
  |> updateDbFromRequest  
  |> sendEmail  
  |> returnMessage
```

Two track



See fsharpforfunandprofit.com/rop

Designing the unhappy path



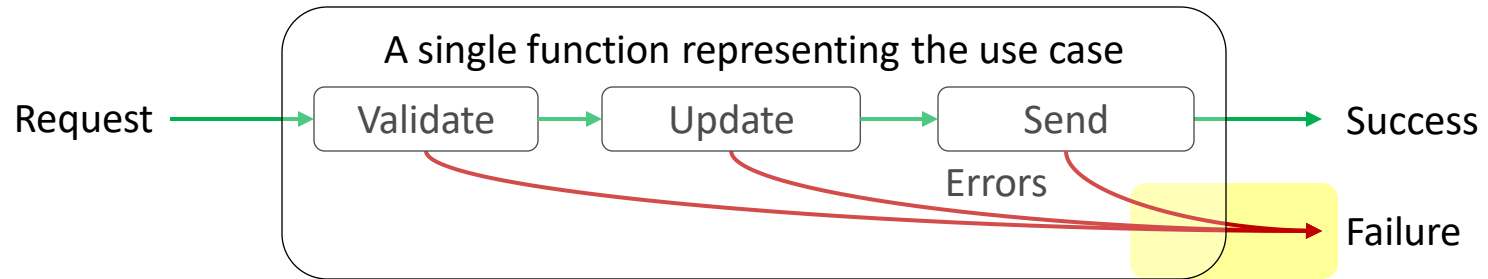
How can a function have more than one output?

Use a choice type!

```
type Result =  
  | Ok  
  | ValidationError  
  | UpdateError  
  | SmtplibError
```

But maybe too specific for this case?

Functional design

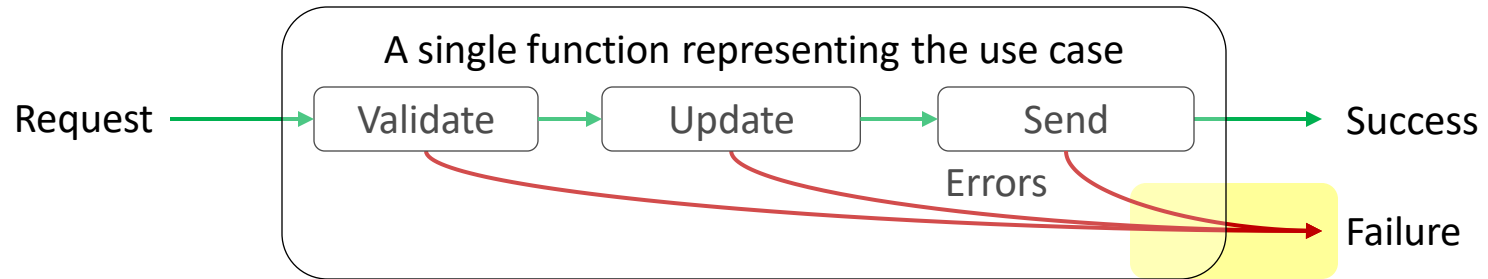


How can a function have more than one output?

```
type Result =  
  | Ok  
  | Error
```

Much more generic – but no data!

Functional design



How can a function have more than one output?

```
type Result<'SuccessType', 'ErrorType'> =  
  | Ok of 'SuccessType  
  | Error of 'ErrorType
```

This is just what we want

Designing error types

Unhappy paths are requirements too

```
let validateInput input =  
  if input.name = "" then  
    Error "Name must not be blank"  
  else if input.email = "" then  
    Error "Email must not be blank"  
  else  
    Ok input // happy path
```

```
// returns Result<'Input, string> =
```

Using strings is not good



```
let validateInput input =  
  if input.name = "" then  
    Error NameMustNotBeBlank  
  else if input.email = "" then  
    Error EmailMustNotBeBlank  
  else  
    Ok input // happy path
```

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank
```

```
// returns Result<'Input, ErrorMessage> =
```

Defined a special type
rather than string


```
let validateInput input =  
  if input.name = "" then  
    Error NameMustNotBeBlank  
  else if input.email = "" then  
    Error EmailMustNotBeBlank  
  else if (input.email doesn't match regex) then  
    Error EmailNotValid input.email  
  else  
    Ok input // happy path
```

Add invalid
email as data


```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank  
  | EmailNotValid of EmailAddress
```

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank  
  | EmailNotValid of EmailAddress
```

Documentation of everything
that can go wrong --



And it's type-safe
documentation that can't go
out of date!



Also triggers important
DDD conversations

Designing for errors - review

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank  
  | EmailNotValid of EmailAddress  
  // database errors  
  | UserIdNotValid of UserId  
  | DbUserNotFoundError of UserId  
  | DbTimeout of ConnectionString  
  | DbConcurrencyError  
  | DbAuthorizationError of ConnectionString * Credentials  
  // SMTP errors  
  | Smtptimeout of Smtptimeout  
  | Smtpbadrecipient of EmailAddress
```

Documentation of everything that
can go wrong.

Type-safe -- can't go out of date!

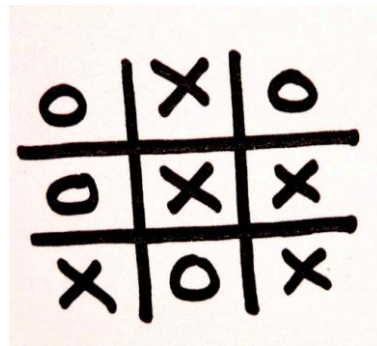
Surfaces hidden requirements.

Test against error codes,
not strings.

Makes translation easier.

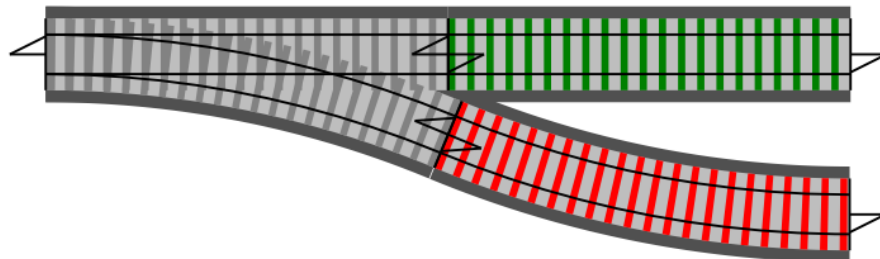
Exercise:

Add errors to the domain models



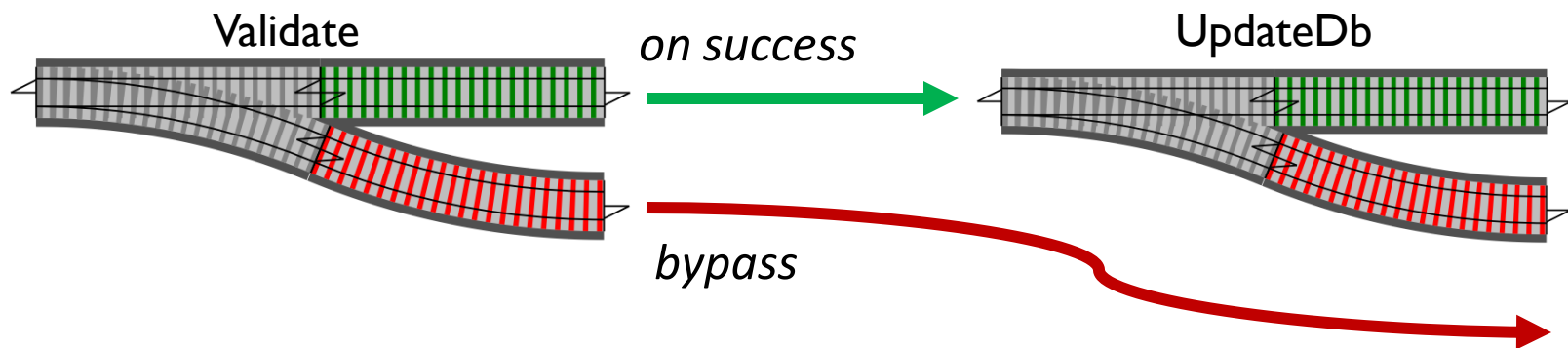
How to implement Railway Oriented Programming

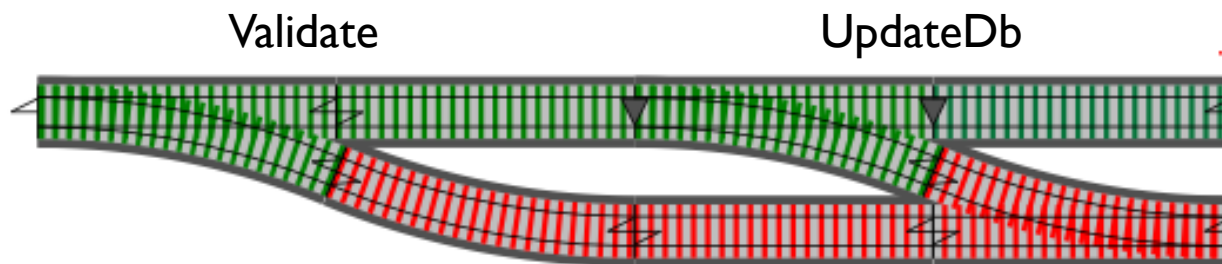
Input ->



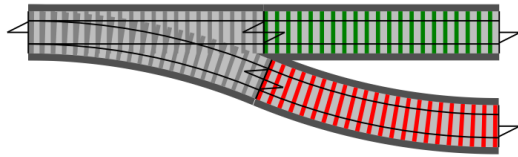
Success!

Failure

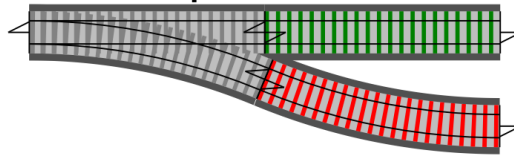




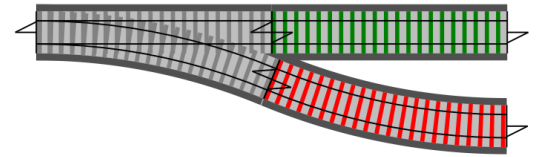
Validate

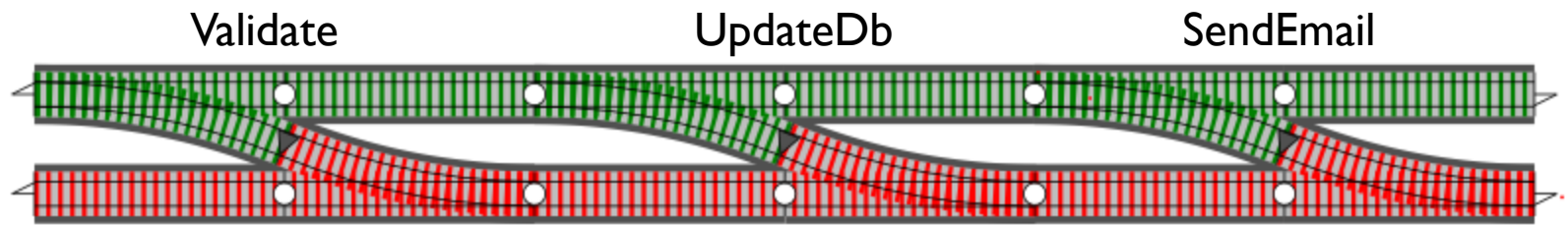


UpdateDb



SendEmail

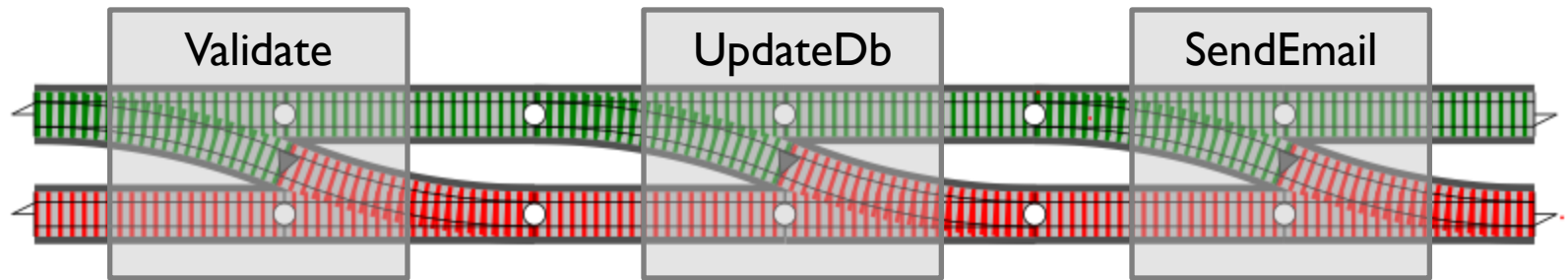




How to compose
these functions?

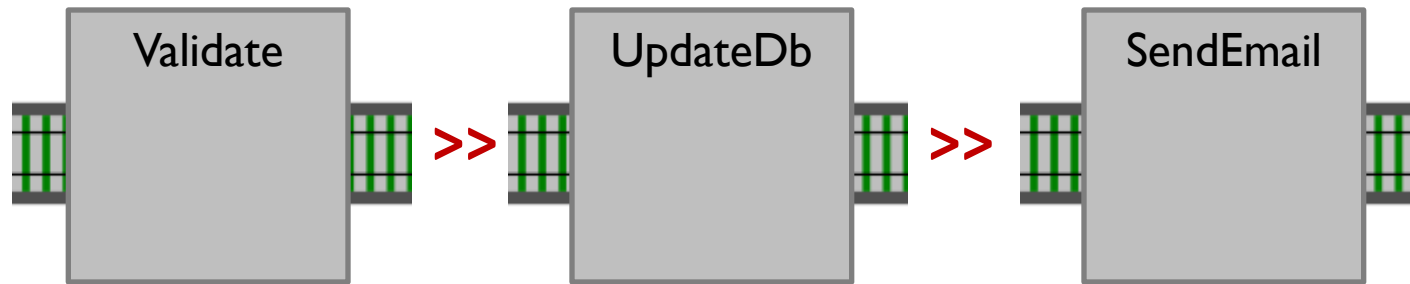


Here we have a series of black box functions
that are straddling a two-track railway.

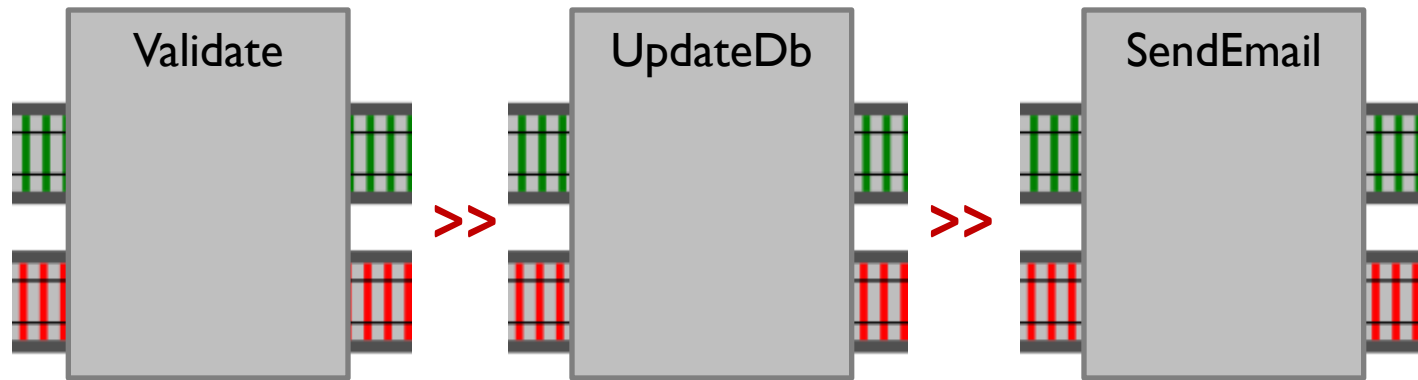


Here we have a series of black box functions
that are straddling a two-track railway.

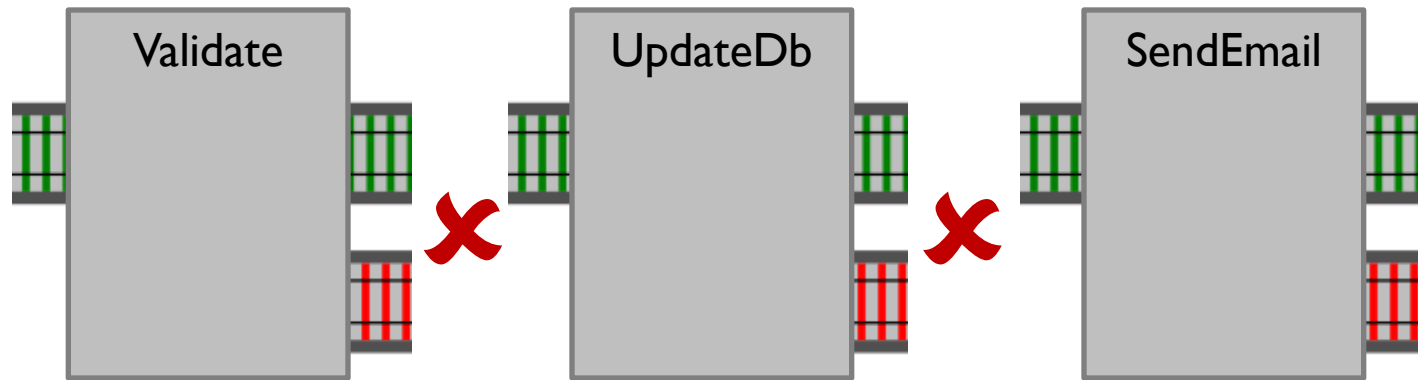
Inside each box there is a switch function.



Composing one-track functions is fine...



... and composing two-track functions is fine...

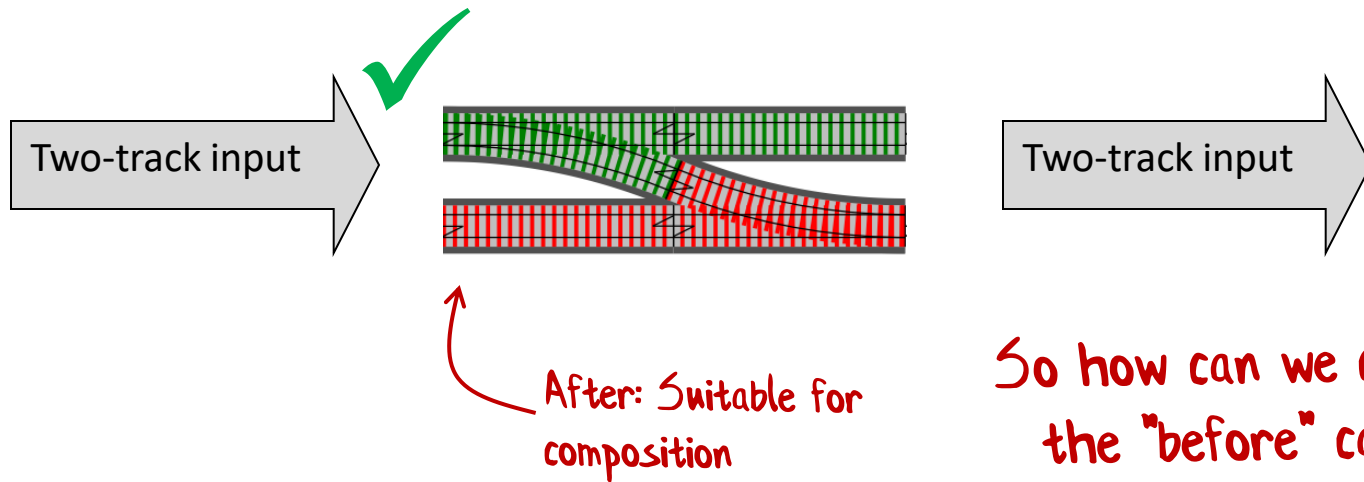
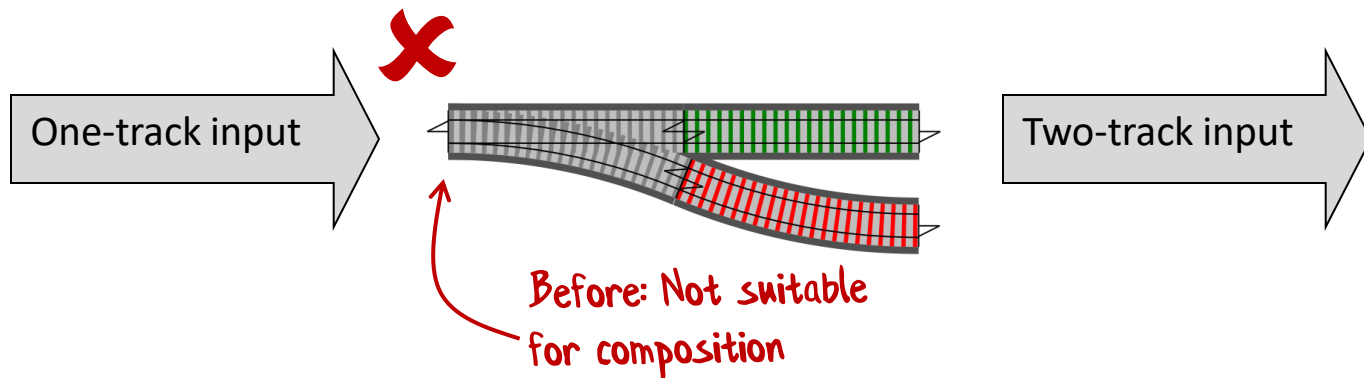


... but composing switches is not allowed!

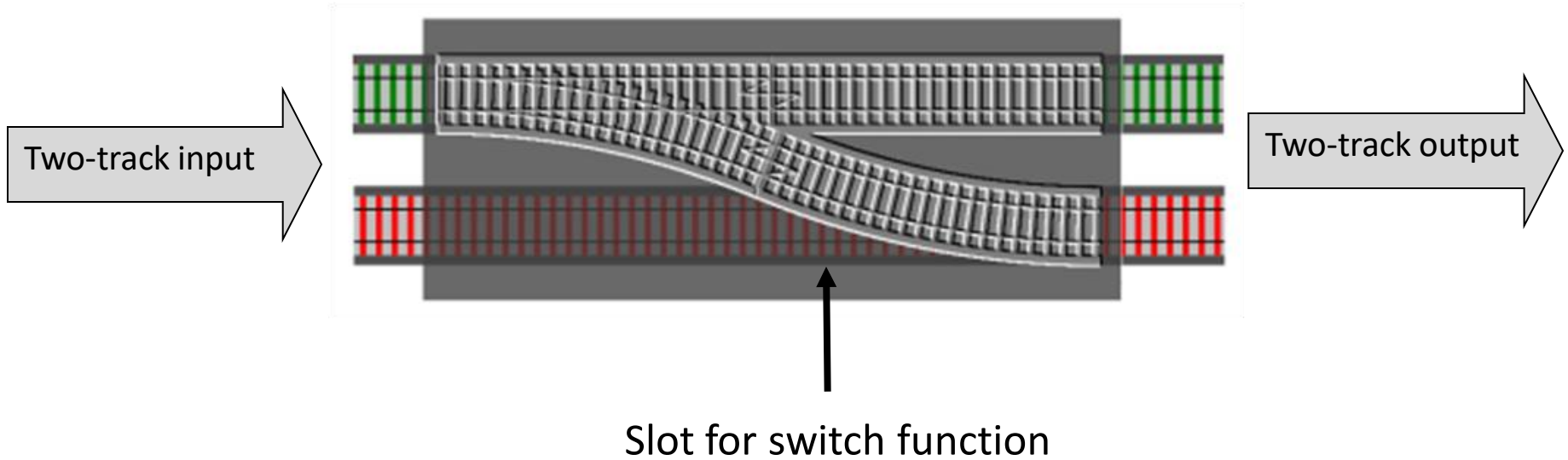
How to combine the
mismatched functions?

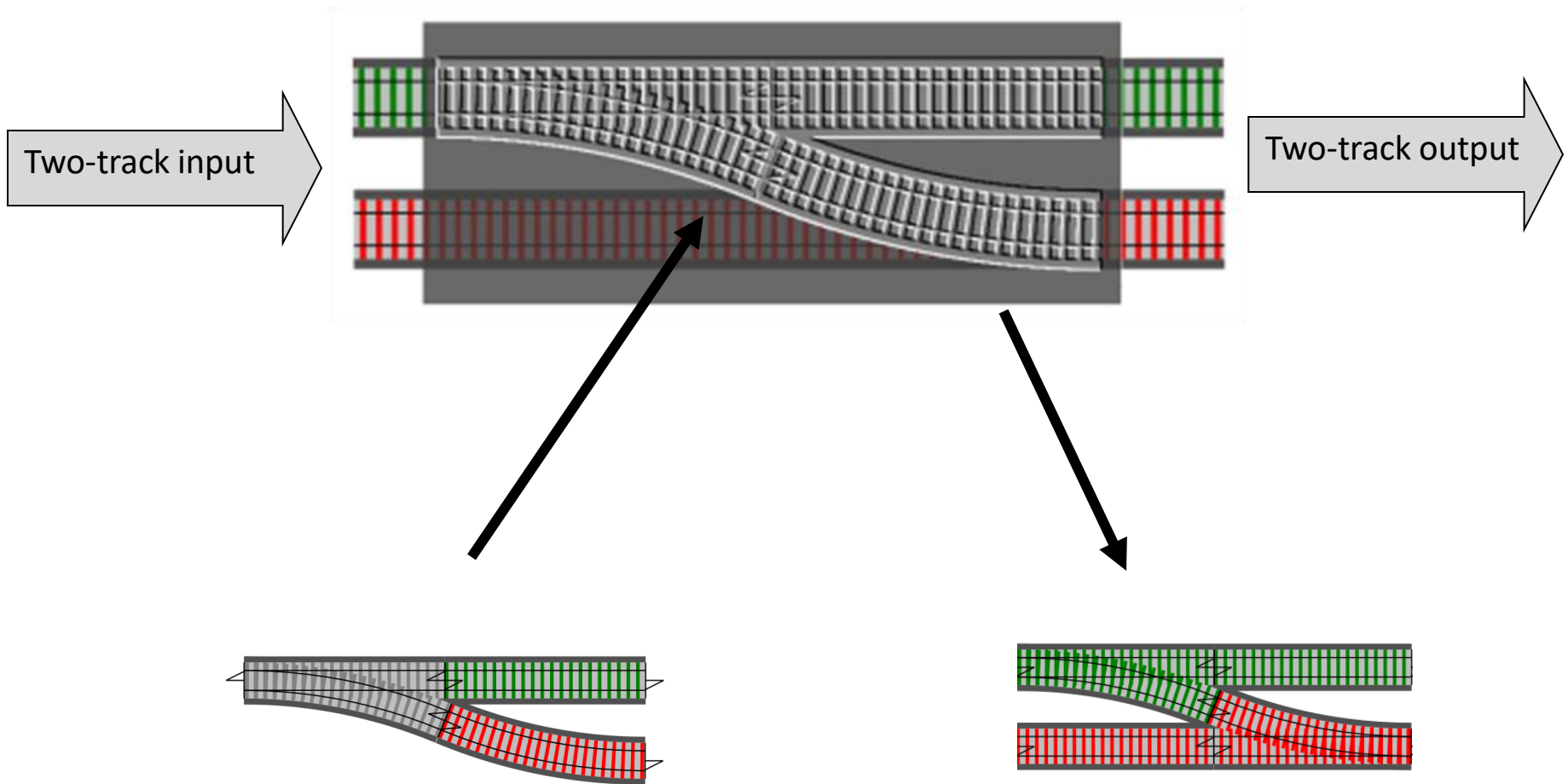
“Bind” is the answer!
Bind all the things!

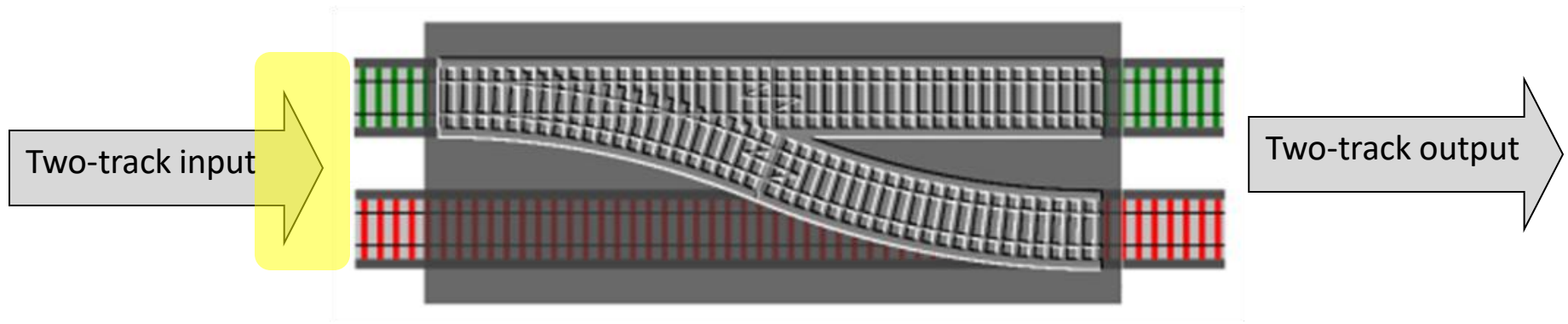
FP'ers get excited by bind



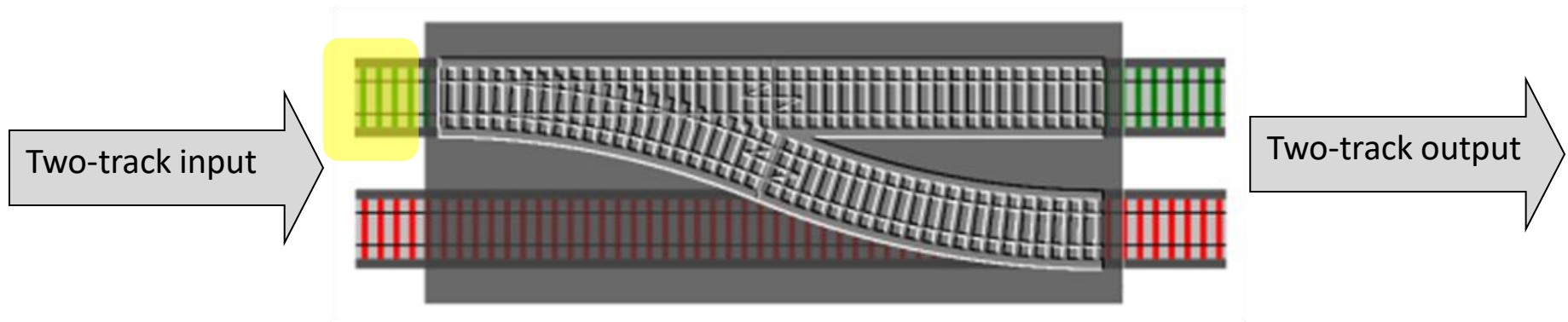
So how can we convert from the "before" case to the "after" case?



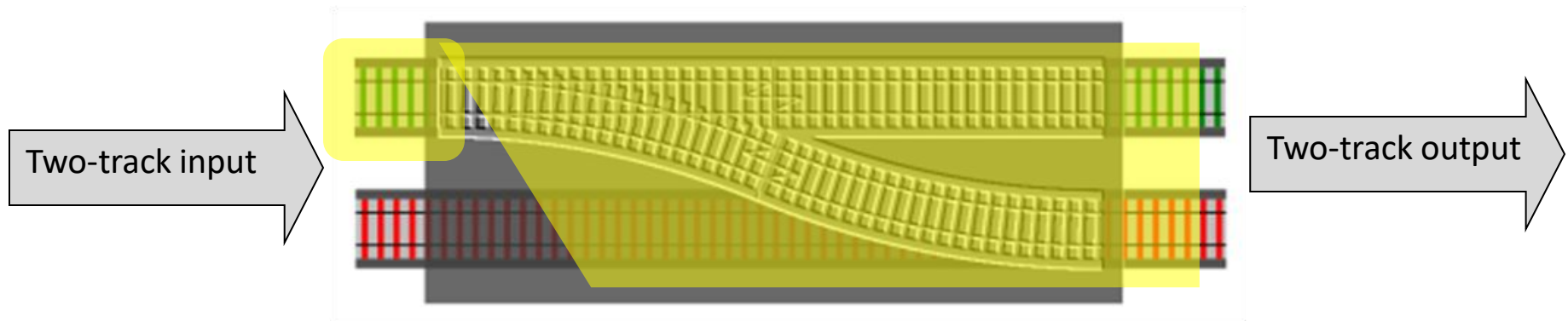




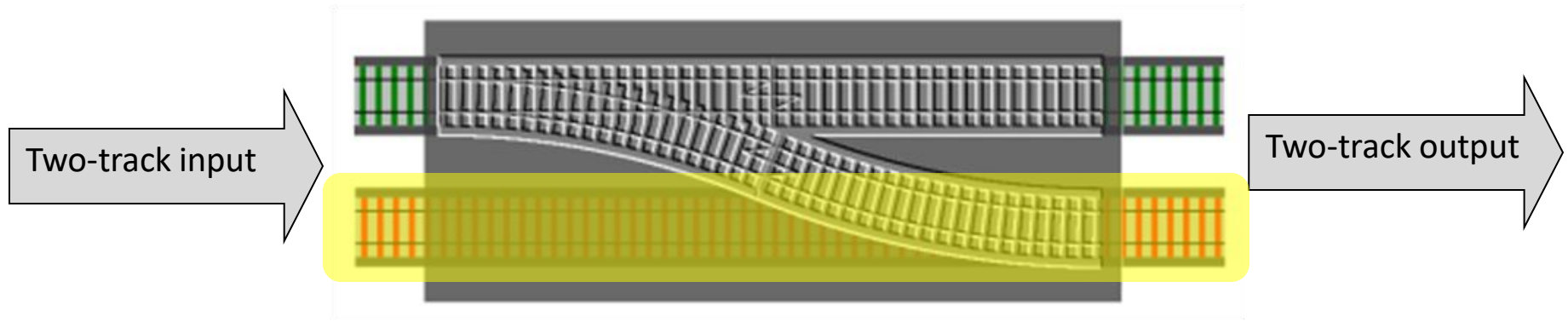
```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```



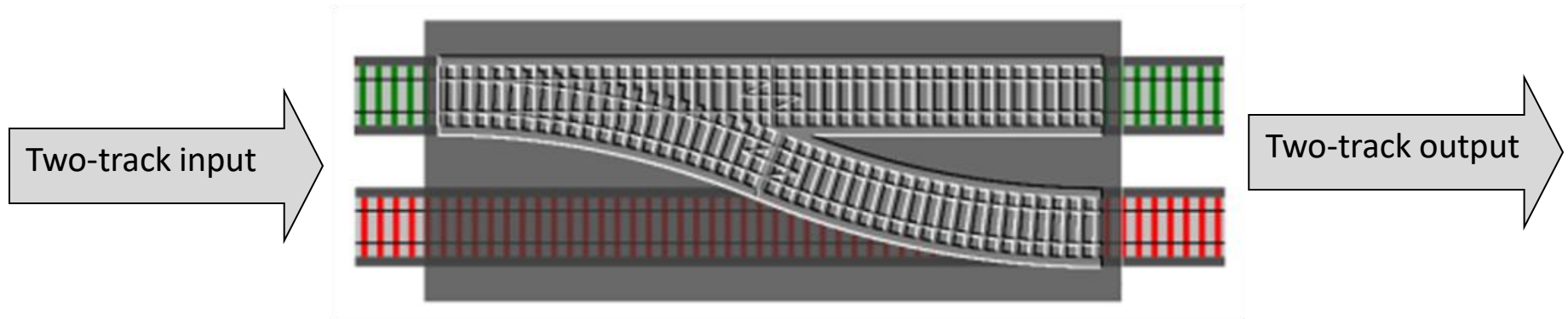
```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```



```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```



```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```



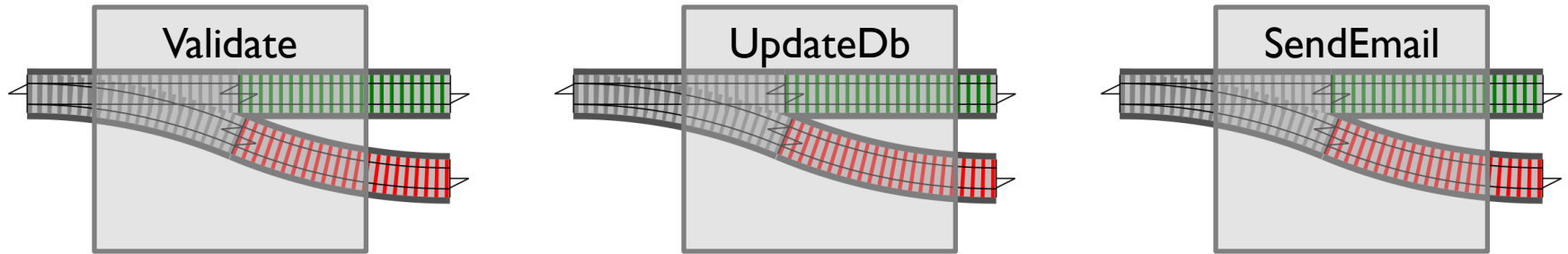
`Result.bind : ('a -> Result<'b>) -> Result<'a> -> Result<'b>`

*Switch
function*

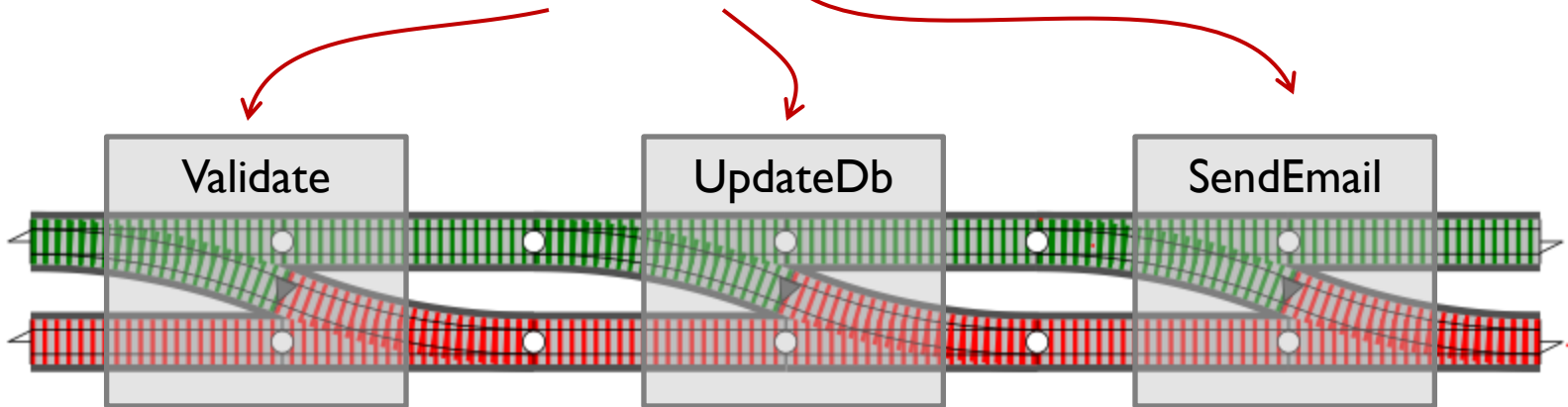
*2-track
input*

*2-track
output*

Composing switches - review



Converted to two-track
functions using bind



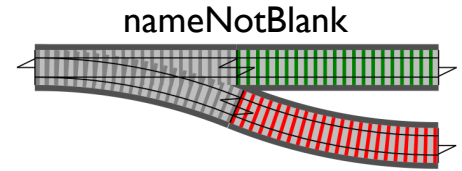
Bind example

Validating input

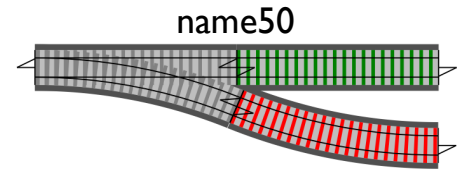
```
type Request = {  
    Name : string  
    Email : string  
}
```

Is this data valid?

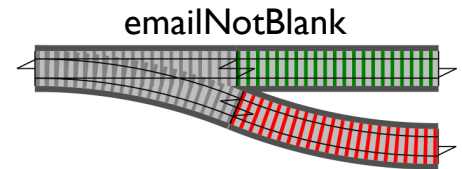

```
let checkNameNotBlank input =  
  if input.Name = "" then  
    Error "Name must not be blank"  
  else Ok input
```

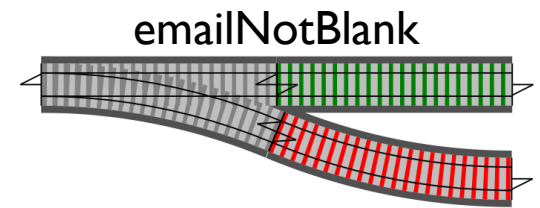
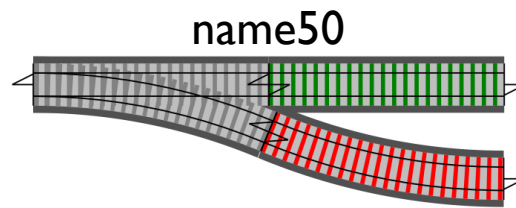
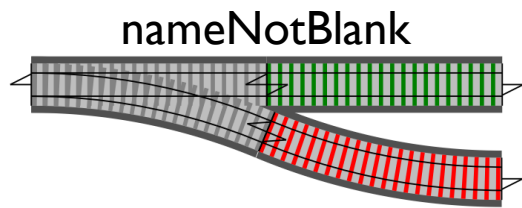


```
let checkName50 input =  
  if input.Name.Length > 50 then  
    Error "Name must not be longer than 50 chars"  
  else Ok input
```

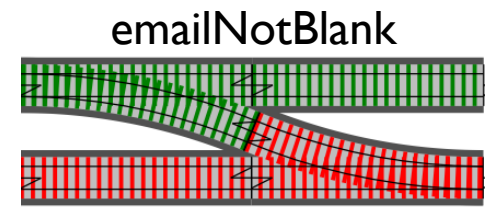
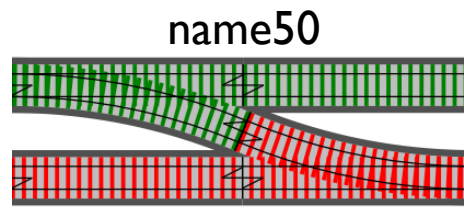
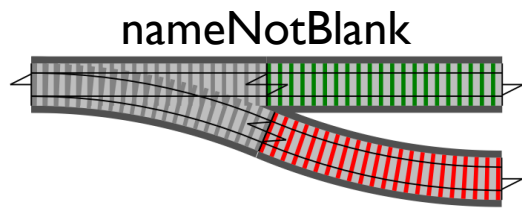


```
let checkEmailNotBlank input =  
  if input.Email = "" then  
    Error "Email must not be blank"  
  else Ok input
```





checkNameNotBlank (composed with)
 checkName50 (composed with)
 checkEmailNotBlank

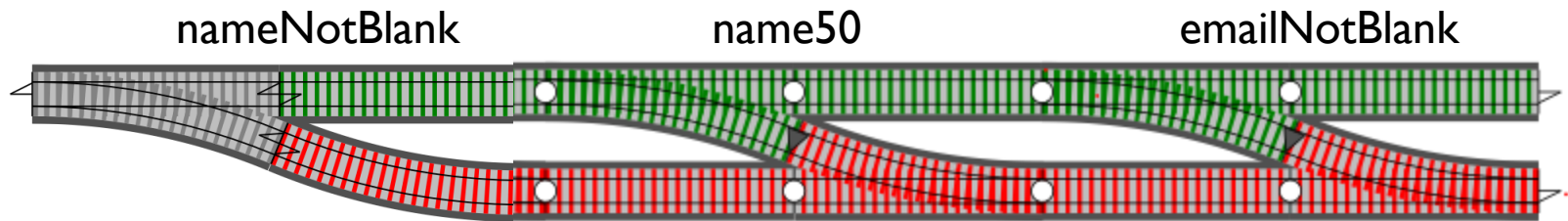


checkNameNotBlank

bind checkName50

bind checkEmailNotBlank

use "bind" to
convert to 2-track



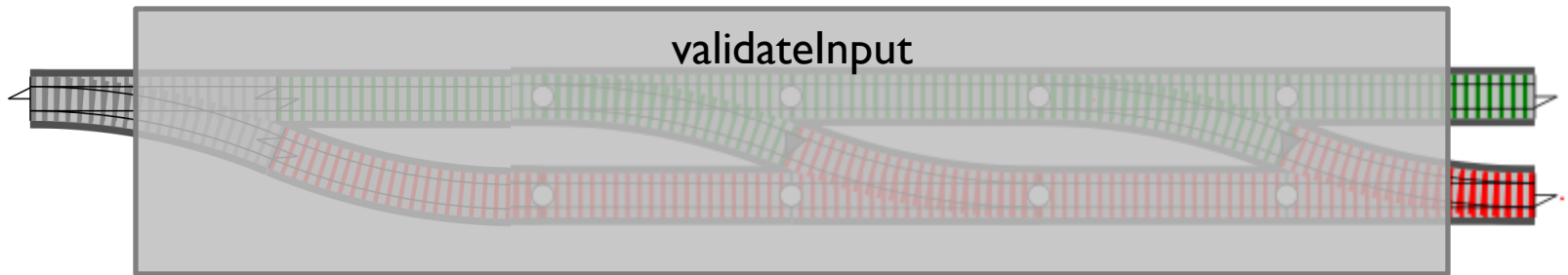
request

```
|> checkNameNotBlank
```

```
|> Result.bind checkName50
```

```
|> Result.bind checkEmailNotBlank
```

then compose together

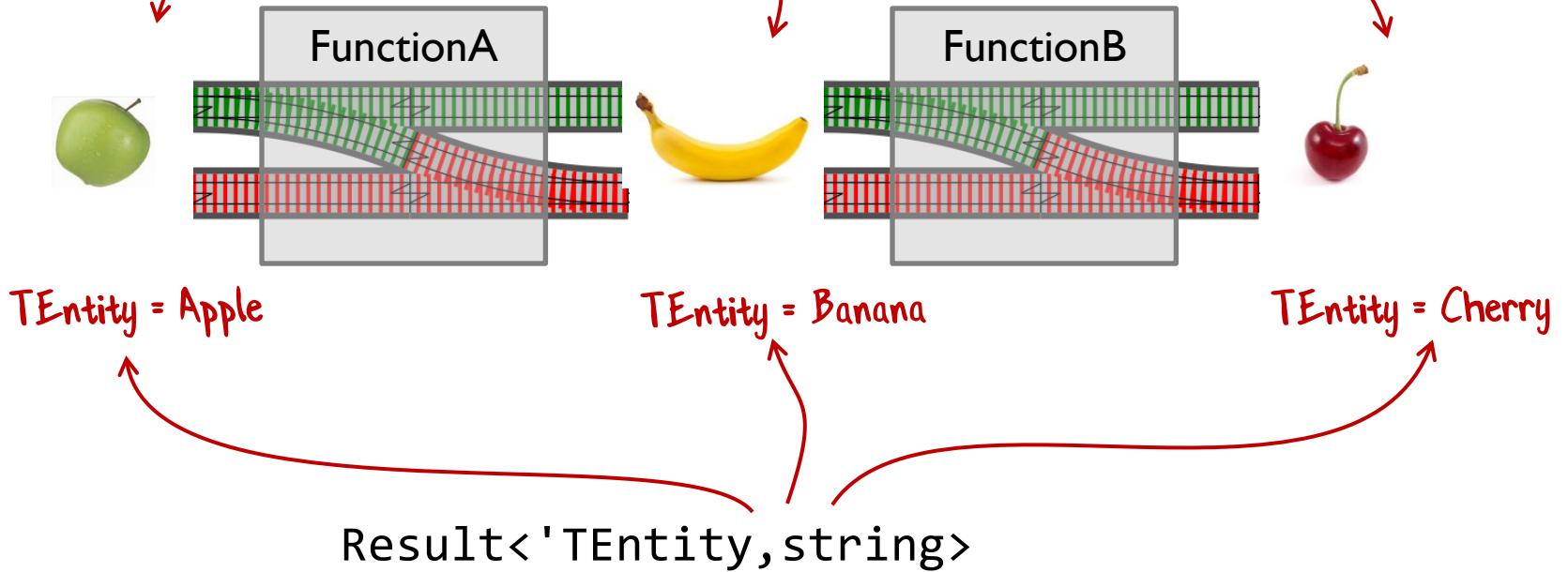


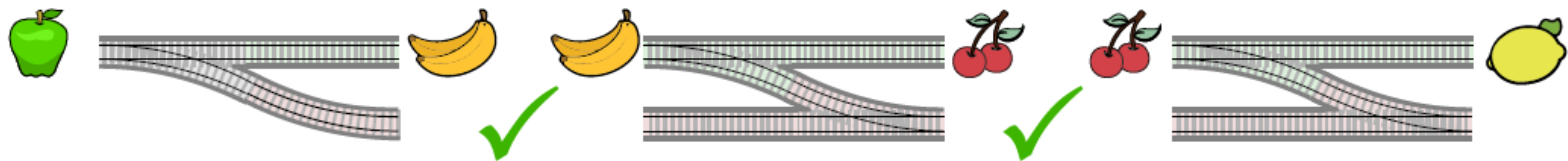
Define a function

```
let validateInput input =  
  input  
  |> checkNameNotBlank  
  |> Result.bind checkName50  
  |> Result.bind checkEmailNotBlank
```

*Overall result is a new
two-track function*

Note that bind is about shape.
Transformations can still happen.

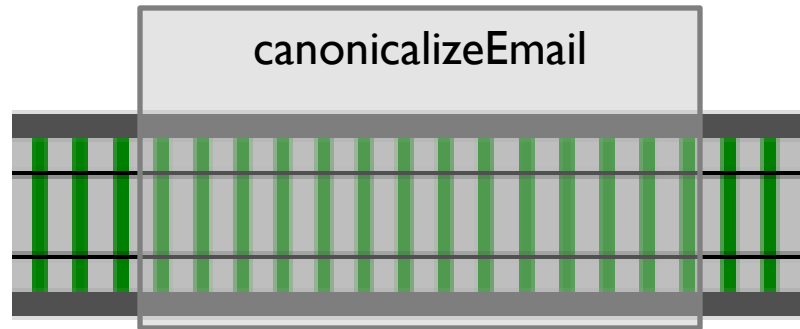




Exercise:

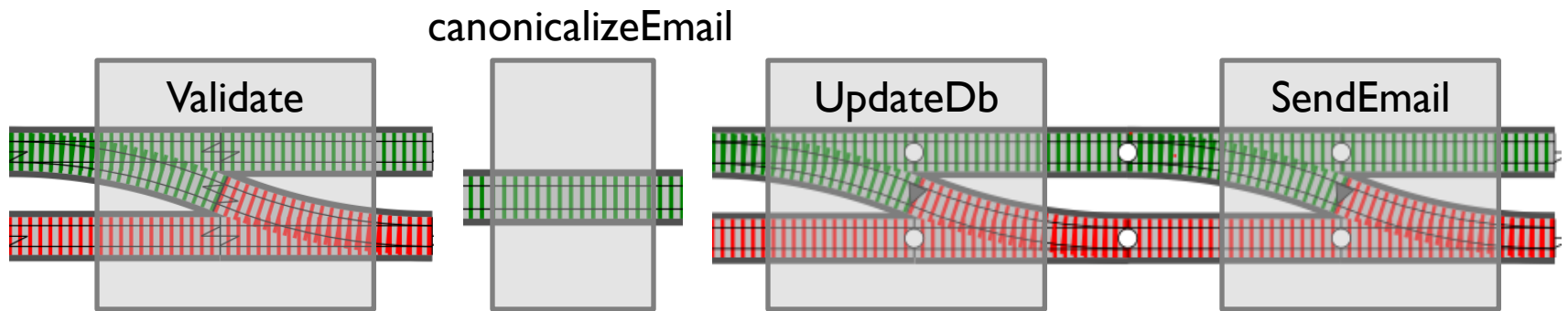
`02a-Exercise-ValidateRequest.fsx`

Mapping the success track

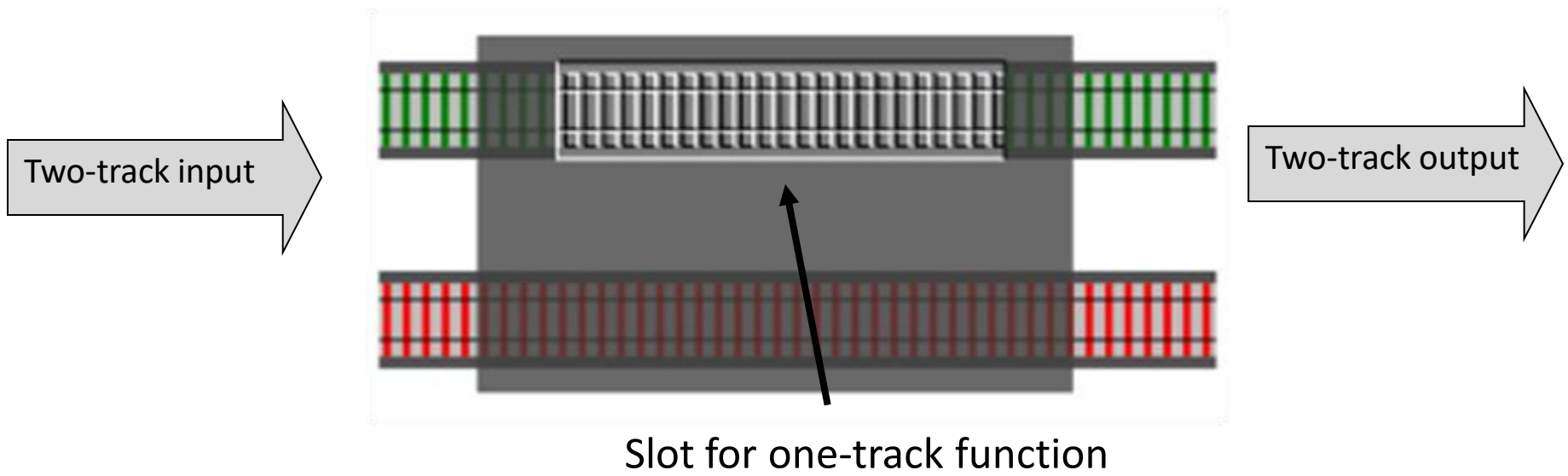


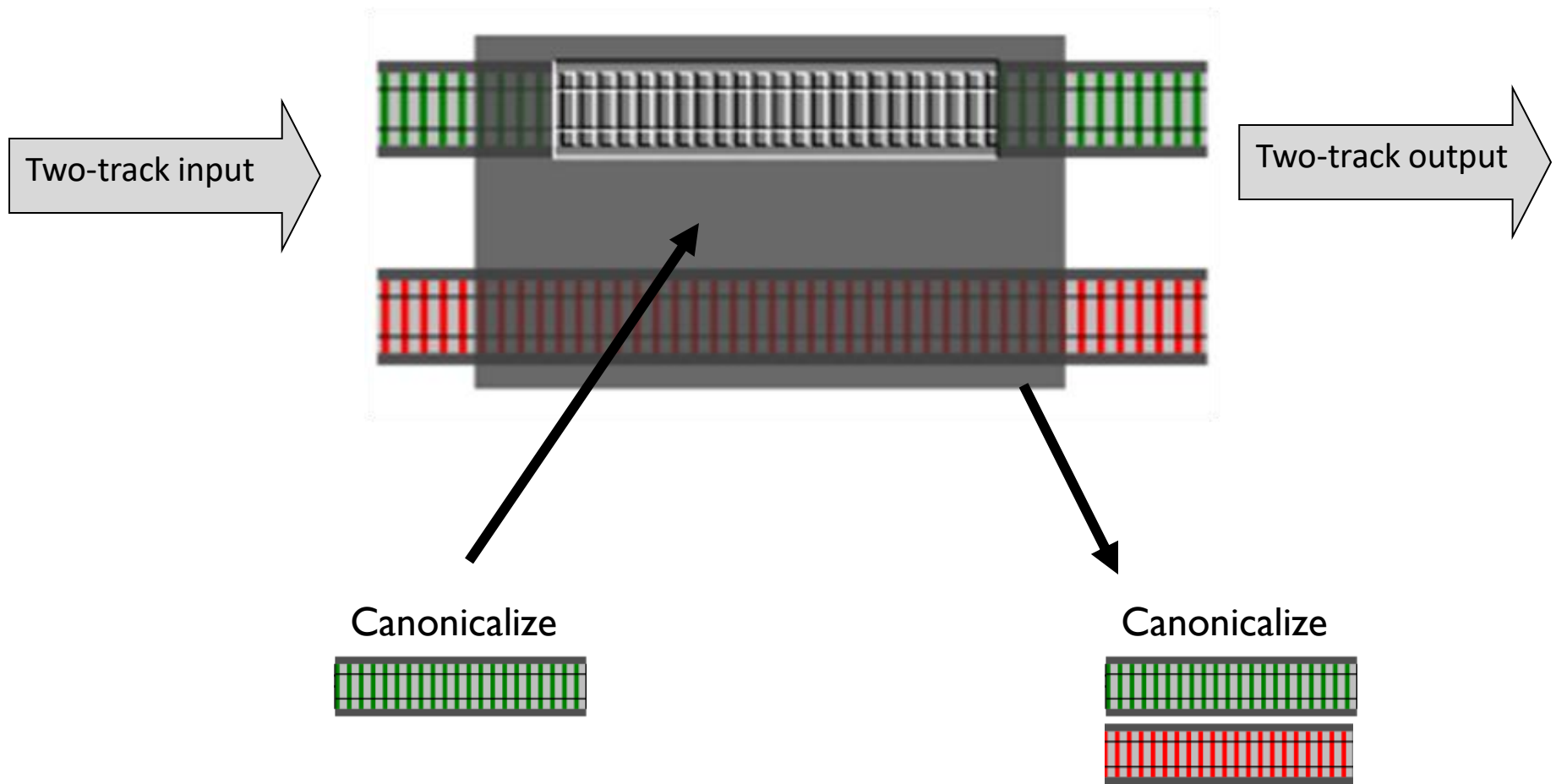
```
// trim spaces and lowercase  
let canonicalizeEmail input =  
  { input with email = input.email.Trim().ToLower() }
```

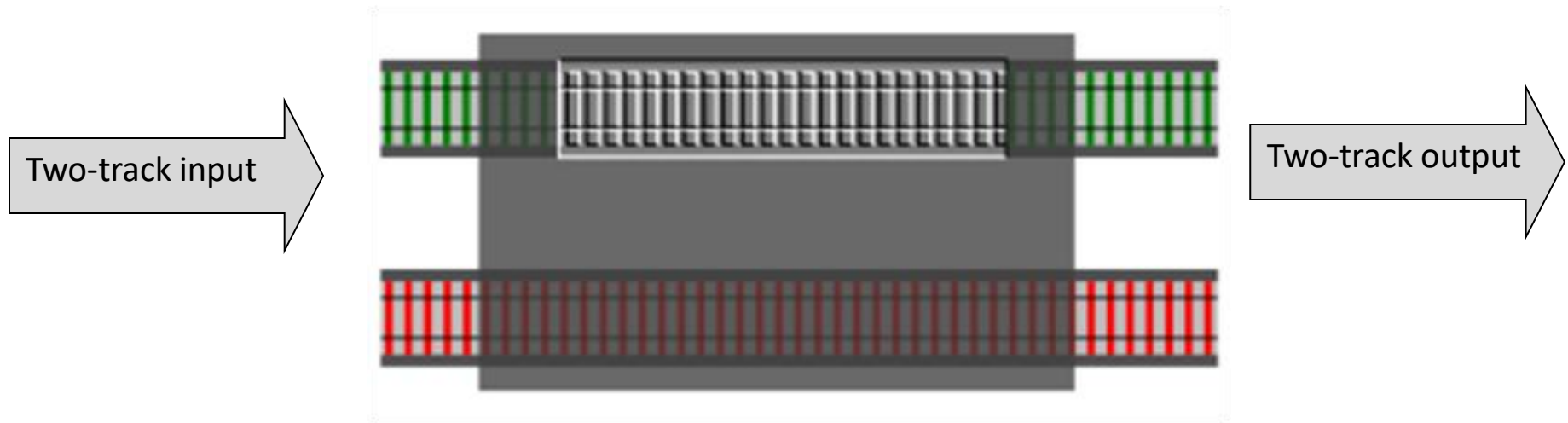
A simple function that doesn't generate errors – a "one-track" function.



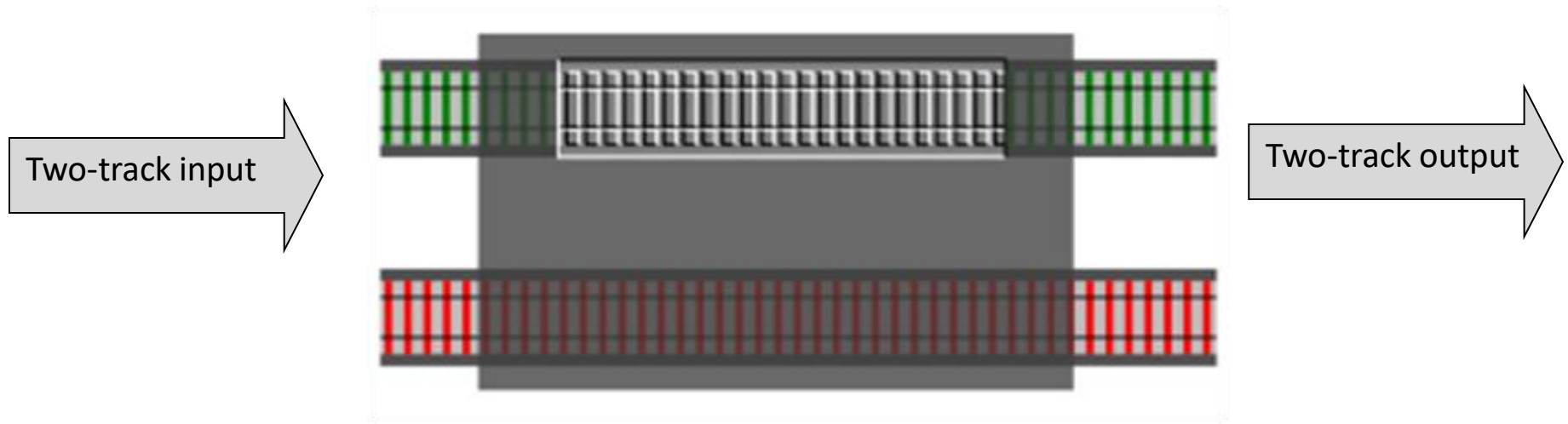
Won't compose







```
let map singleTrackFunction twoTrackInput =  
  match twoTrackInput with  
  | Ok s -> Ok (singleTrackFunction s)  
  | Error f -> Error f
```



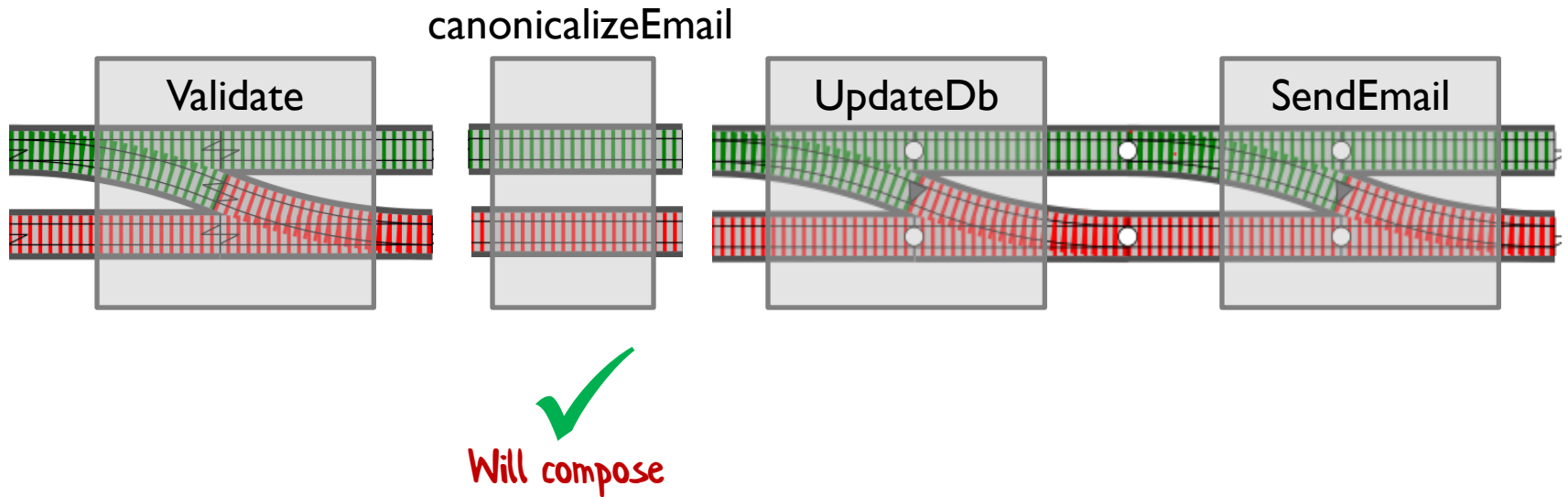
`Result.map : ('a -> 'b) -> Result<'a, 'c> -> Result<'b, 'c>`

Single track
function

2-track
input

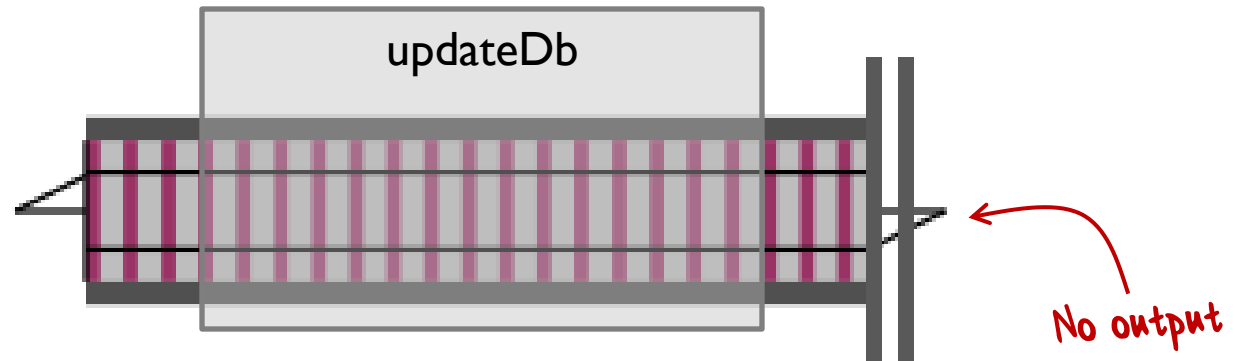
2-track
output

Converting one-track functions



Converting dead-end functions

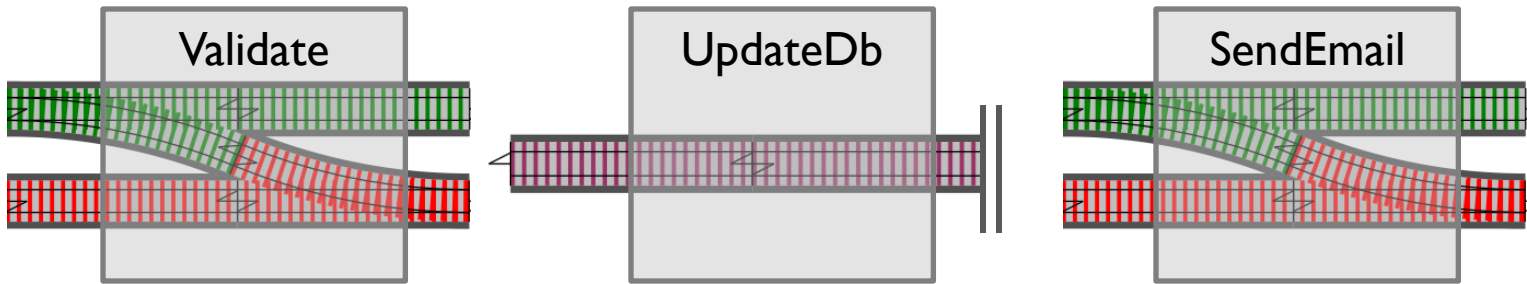
Converting dead-end functions



```
let updateDb request =  
  // do something  
  // return nothing at all
```

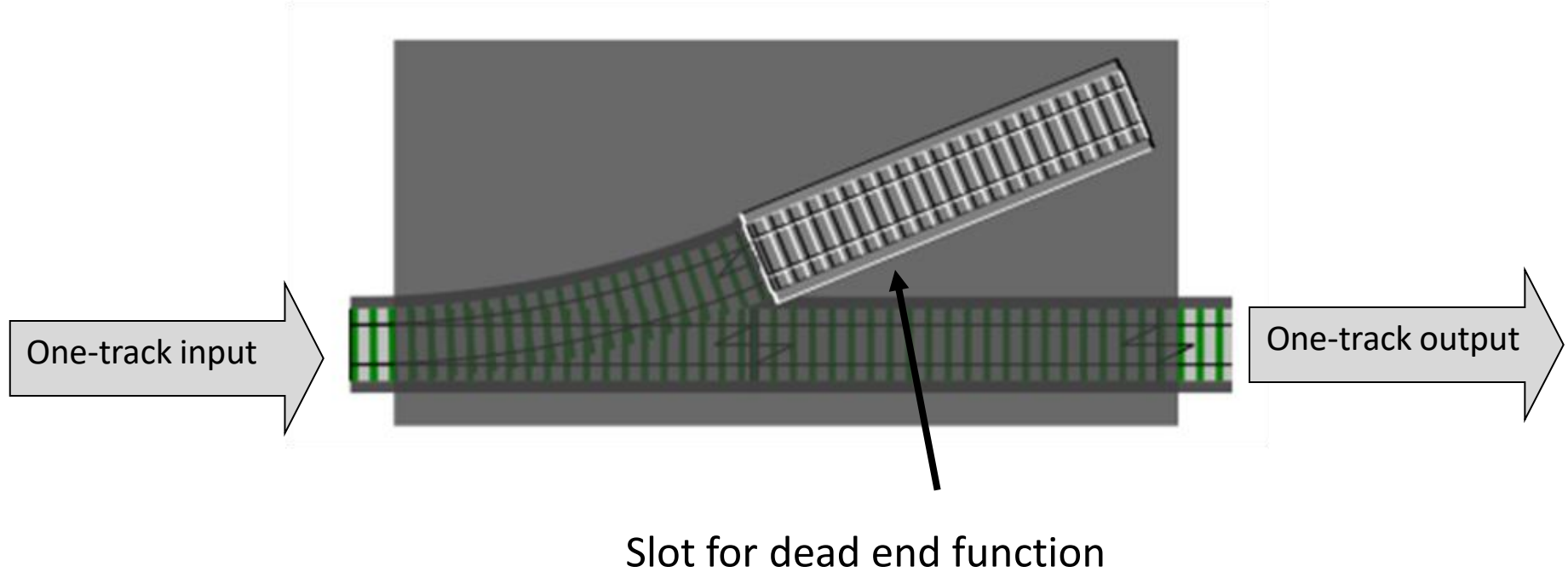
A function that doesn't return anything— a "dead-end" function.

Converting dead-end functions

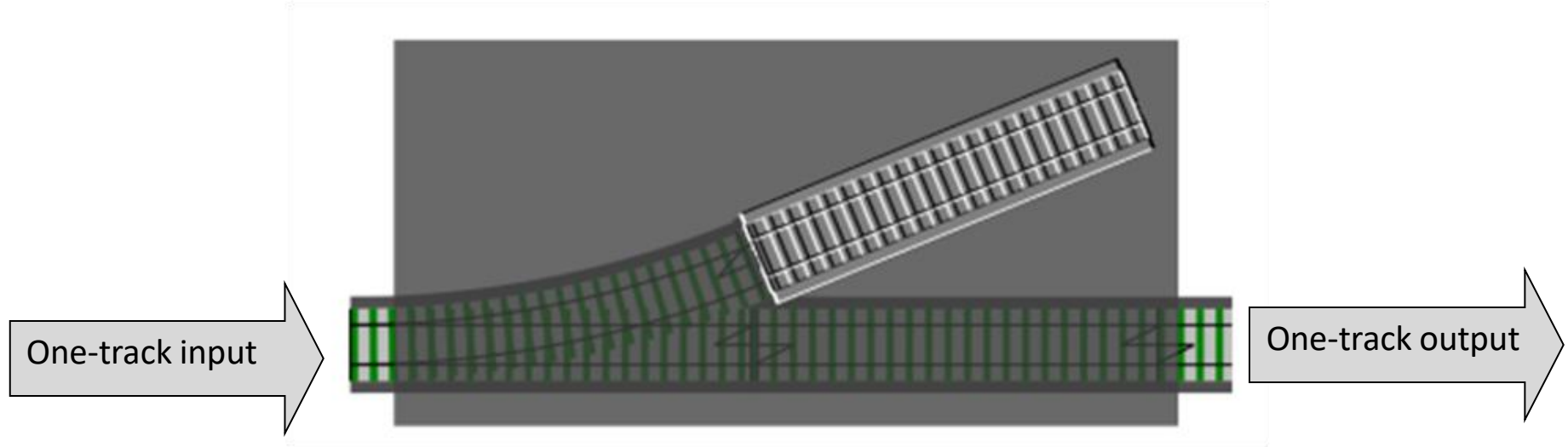


Won't compose

Converting dead-end functions

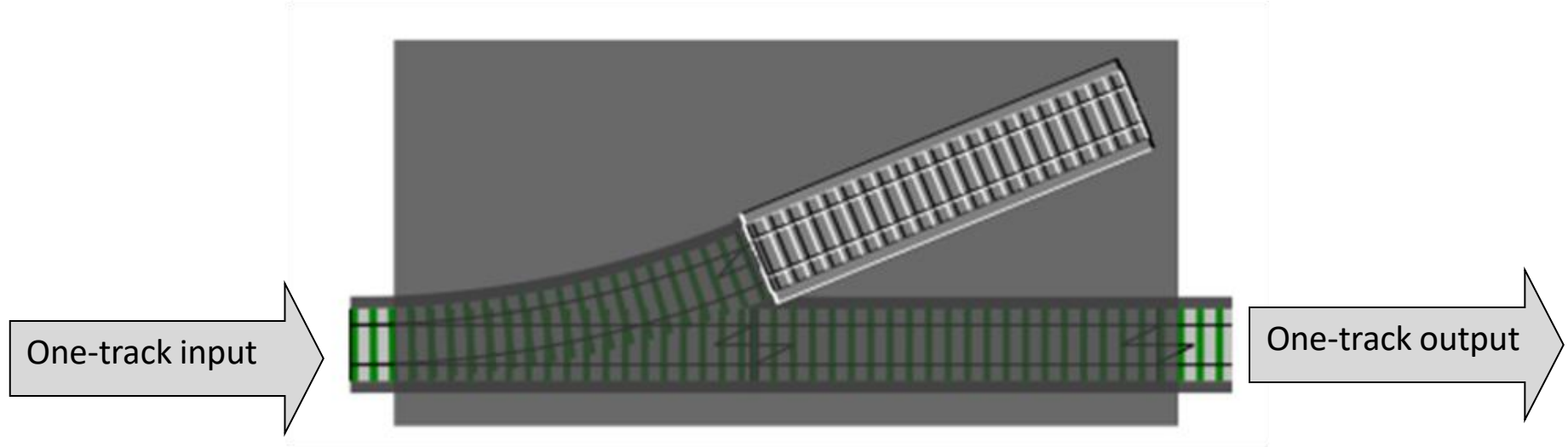


Converting dead-end functions



```
let tee deadEndFunction oneTrackInput =  
  deadEndFunction oneTrackInput  
  oneTrackInput
```

Converting dead-end functions



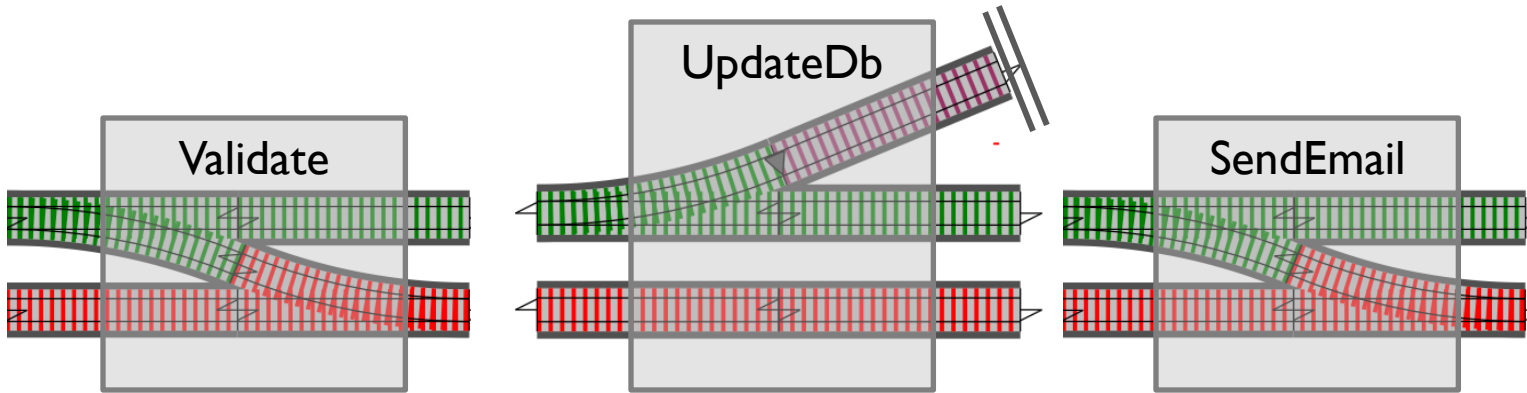
```
tee : ('a -> unit) -> 'a -> 'a
```

Dead end
function

one-track
input

one-track
output

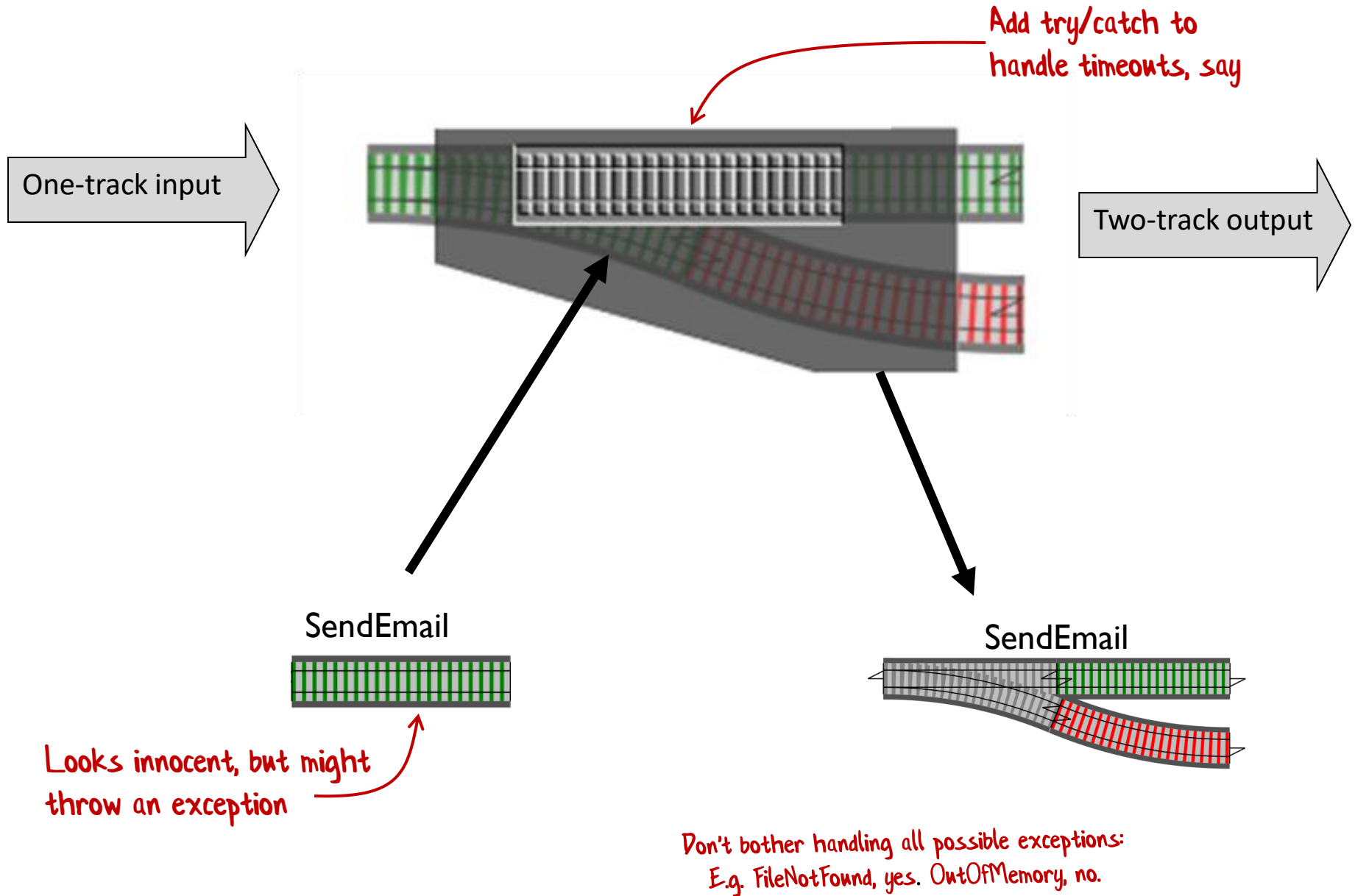
Converting dead-end functions



Will compose

Functions that throw exceptions

Functions that throw exceptions



Functions that throw exceptions

Guideline: Convert exceptions into Failures

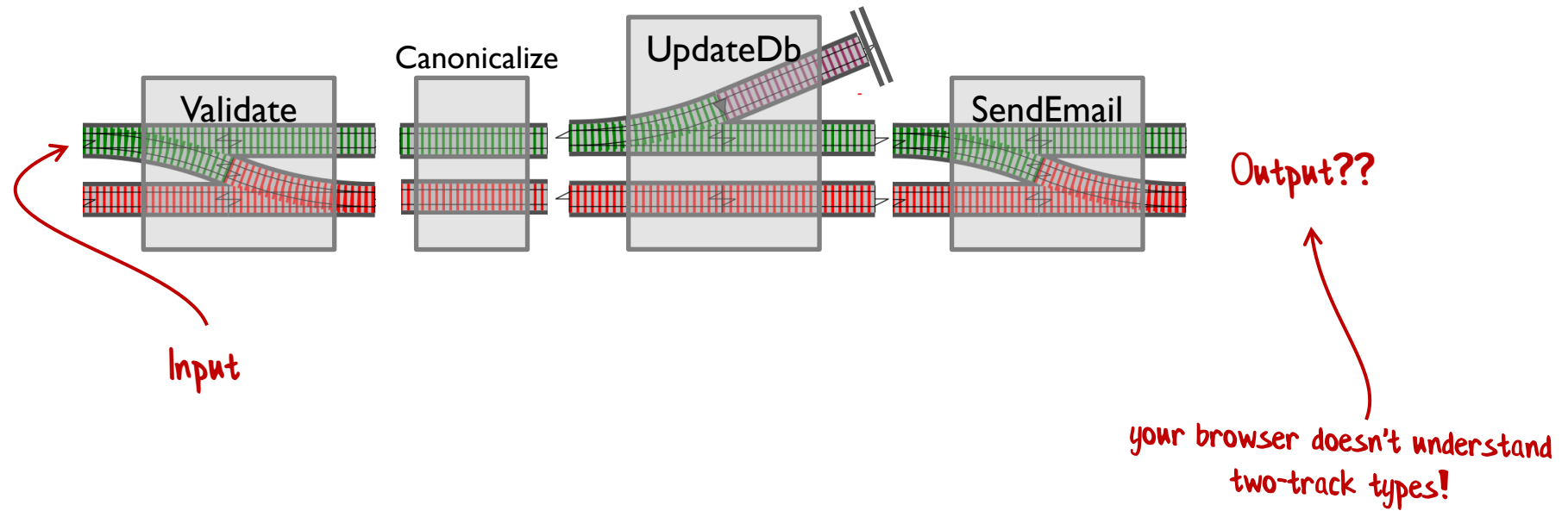


Even Yoda recommends not to use exception handling for control flow:

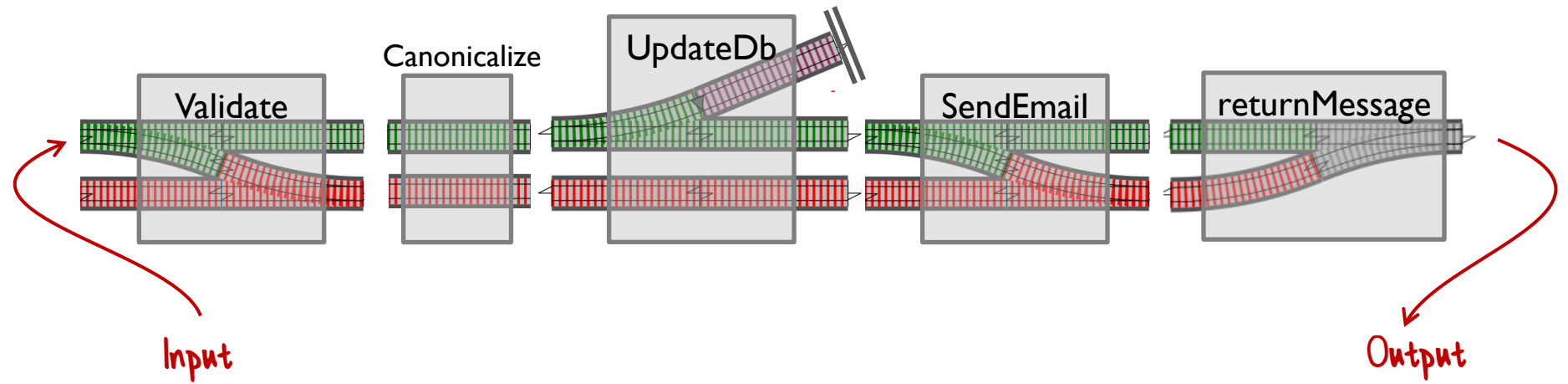
"Do or do not, there is no try".

Putting it all together

Putting it all together



Putting it all together



```
let returnMessage result =  
  match result with  
  | Ok obj -> OK obj.ToJson()  
  | Error msg -> BadRequest msg
```

Demo:

03a-RopWithStringError.fsx

Exercise:

03b-Exercise-RopWithErrorType.fsx

Mapping the error track

Converting to a common error type

Make sure all errors are the same type

- The error track has the same type all the way along the track.
- So, we may need to change the error types to make them compatible.
- This what "Result.mapError" is for


```
type FunctionA =
```

```
  Apple -> Result<Bananas, AppleError>
```

```
type FunctionB =
```

```
  Bananas -> Result<Cherries, BananaError>
```



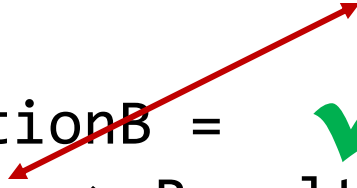
```
// define a common superset  
type FruitError =  
| AppleErrorCase of AppleError  
| BananaErrorCase of BananaError
```

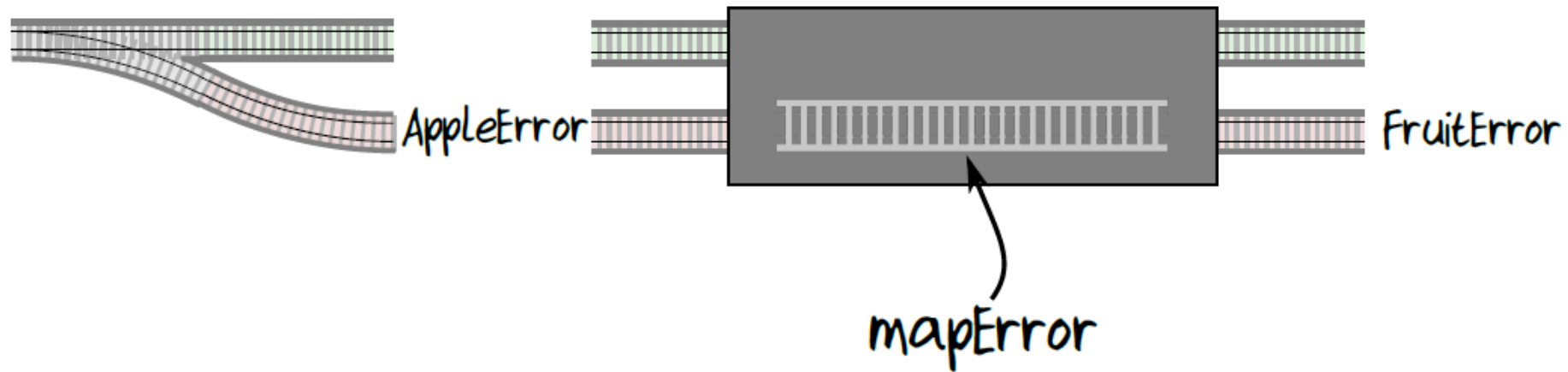
```
type FunctionA =
```

```
  Apple -> Result<Bananas, FruitError>
```

```
type FunctionB =
```

```
  Bananas -> Result<Cherries, FruitError>
```





```
let functionA' input =  
  input  
  |> functionA // original function  
  |> Result.mapError AppleErrorCase  
  // Apple -> Result<Bananas,FruitError>  
  
// do the same for the other function  
let functionB' input =  
  input  
  |> functionB // original function  
  |> Result.mapError BananaErrorCase  
  // Bananas -> Result<Cherries,FruitError>  
  
// now they can be composed!
```

Review of "implementing workflows"

Demo:

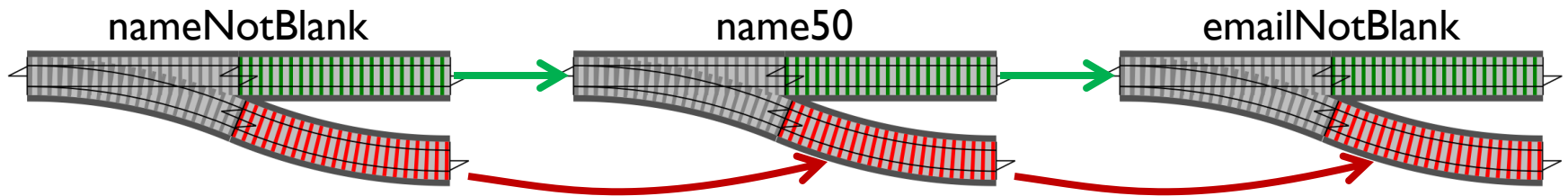
04a-RopWithTicTacToe.fsx

Exercise:

04b-Exercise-RopWithCoffeeMaker.fsx

Validation

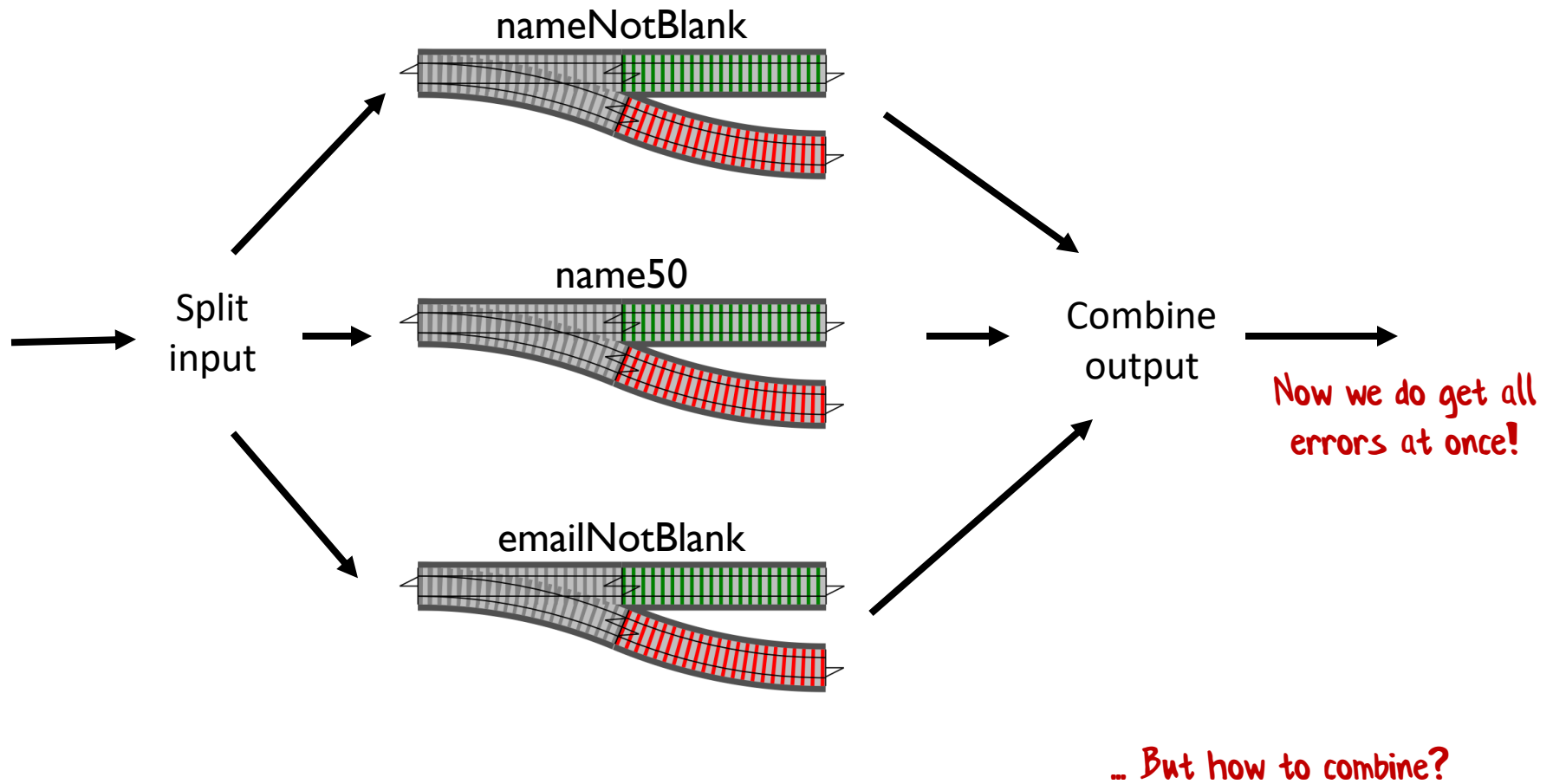
Parallel validation



Problem: Validation done in series.
So only one error at a time is returned

It would be nice to return all
validation errors at once.

Parallel validation



How does it work?

- A bunch of functions that return a `ValidationType`
- A constructor
- Use `<!*>` and `<*>`

How does it work?

```
let createPerson first last age =  
  {First=first; Last=last; Age=age}
```

```
let firstR = validateFirst first  
  // returns Validation<First,_>
```

```
let lastR = validateLast last  
  // returns Validation<Last,_>
```

```
let ageR = validateAge age  
  // returns Validation<Age,_>
```

```
let personR =  
  createPerson <!> firstR <*> lastR <*> ageR  
  // returns Validation<Person,_>
```

Demo:

05a-ValidationWithMultipleErrors.fsx

Exercise:

05b-Exercise-DtoValidation.fsx

Computation expressions

Demo:

06a-ResultComputationExpression.fsx