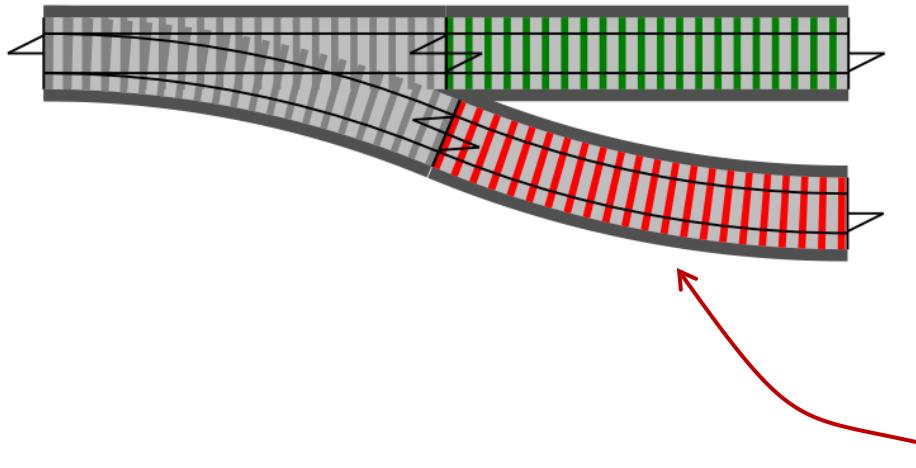


# Railway Oriented Programming

## A functional approach to error handling



What do railways  
have to do with  
programming?

# Railway Oriented Programming

## A functional approach to error handling

Scott Wlaschin  
@ScottWlaschin

Examples will be  
in F#...

fsharpforfunandprofit.com

...but OCaml and Haskell  
are very similar.

FPbridge.co.uk

# Overview

## Topics covered:

- Happy path programming
- Straying from the happy path
- Introducing "Railway Oriented Programming"
- Using the model in practice
- Extending and improving the design

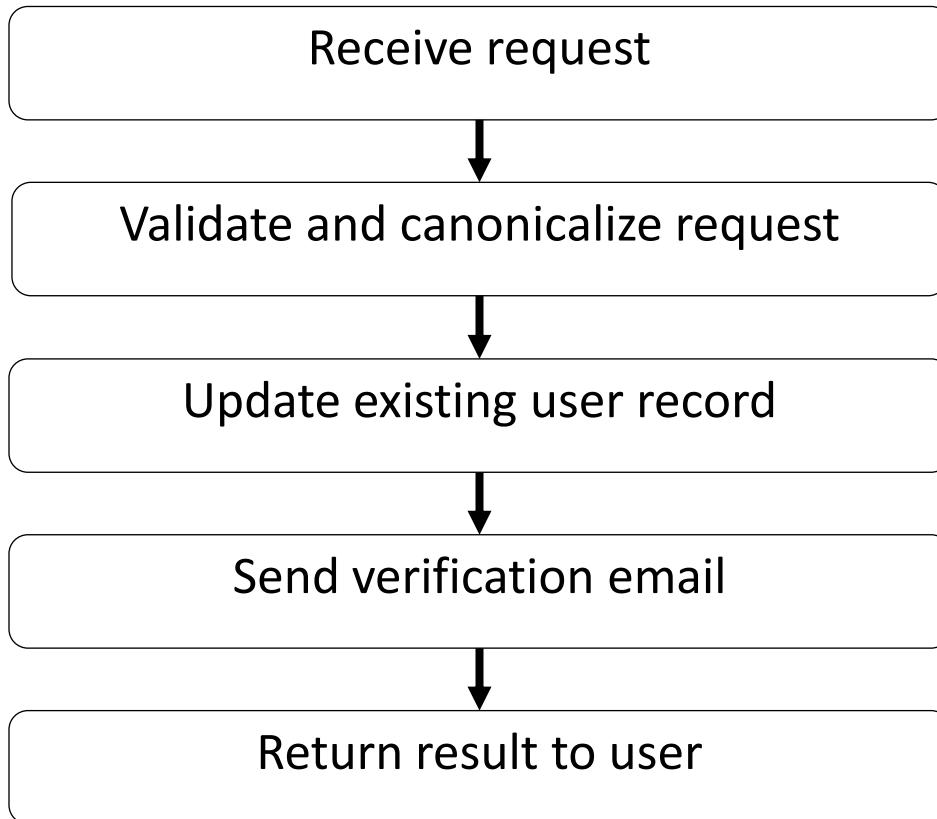
# Happy path programming

Implementing a simple use case



# A simple use case

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



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

# Imperative code

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

# Functional flow

```
let executeUseCase =  
    receiveRequest  
    >> validateRequest  
    >> canonicalizeEmail  
    >> updateDbFromRequest  
    >> sendEmail  
    >> returnMessage
```



F# left-to-right  
composition operator

# Straying from the happy path...

What do you do when  
something goes wrong?

# Straying from the happy path



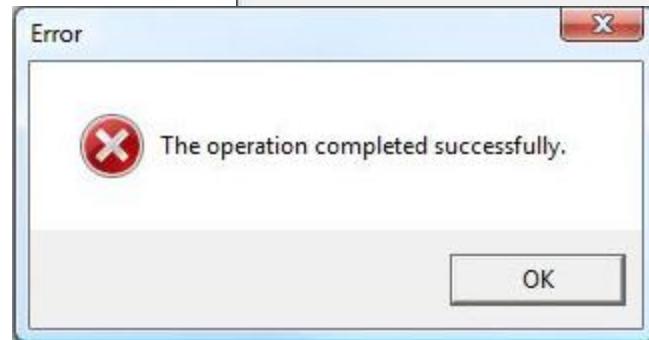
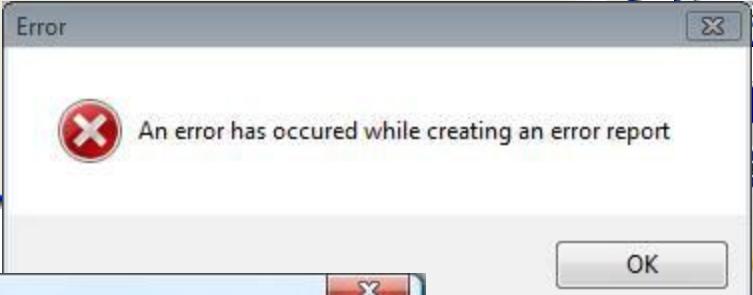
“A program is a spell cast over a  
computer, turning input  
into error messages”

## Microsoft Visual Studio



An exception of type 'System.NotImplementedException' occurred in UnhandledExceptionBlog.exe but was not handled in user code

Additional information: The developer needs to do his job.



## Form1



Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

ORA-1017: invalid username/password; logon denied.

Details

Continue

Quit

D! - Bad User!!!

You've been warned 3 times that this file does not exist. Now you've made us catch this worthless exception and we're upset. Do not do this again.

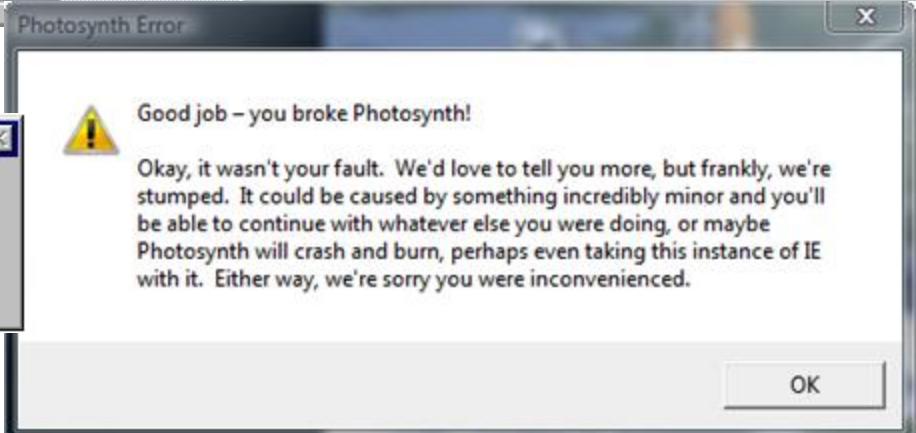
OK



## Microsoft Visual Basic

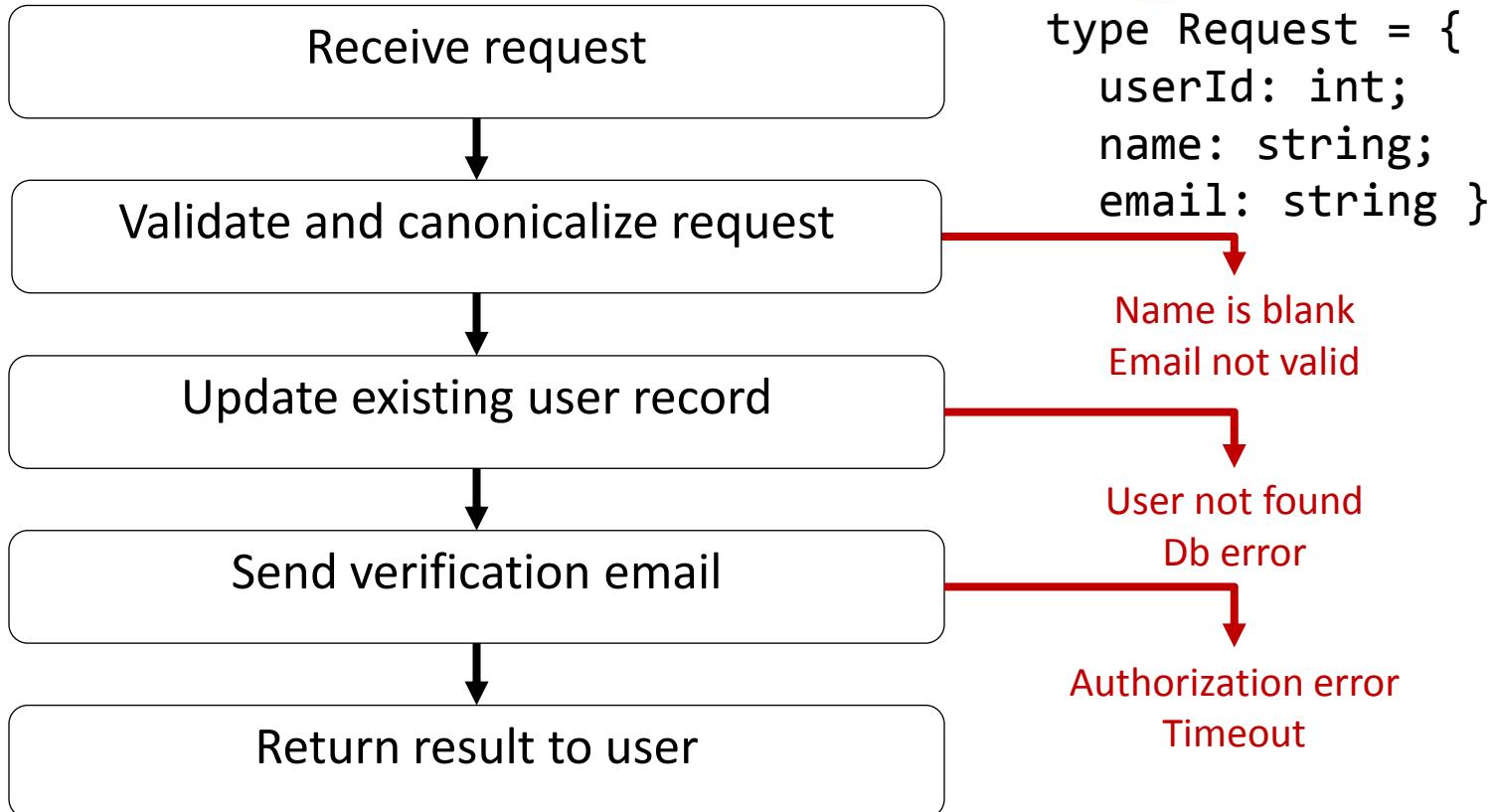
Run-time error '6':

Overflow



# Straying from the happy path

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



# Imperative code with error handling

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

    return "OK";
}
```

# Imperative code with error handling

```
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";
}
```

# Imperative code with error handling

```
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";
}
```

# Imperative code with error handling

```
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";
}
```

# Imperative code with error handling

```
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";
}
```

# Imperative code with error handling

```
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";
}
```

6 clean lines → 18 ugly lines. 200% extra!  
Sadly this is typical of error handling code.

Q: What is the functional equivalent of this code?

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

# Functional flow with error handling

Before



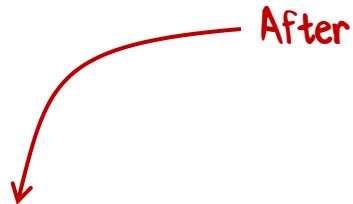
```
let updateCustomer =  
receiveRequest  
>> validateRequest  
>> canonicalizeEmail  
>> updateDbFromRequest  
>> sendEmail  
>> returnMessage
```

Q: What is the functional equivalent of this code?

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

# Functional flow with error handling

```
let updateCustomerWithErrorHandling =  
receiveRequest  
>> validateRequest  
>> canonicalizeEmail  
>> updateDbFromRequest  
>> sendEmail  
>> returnMessage
```



Does this look familiar?

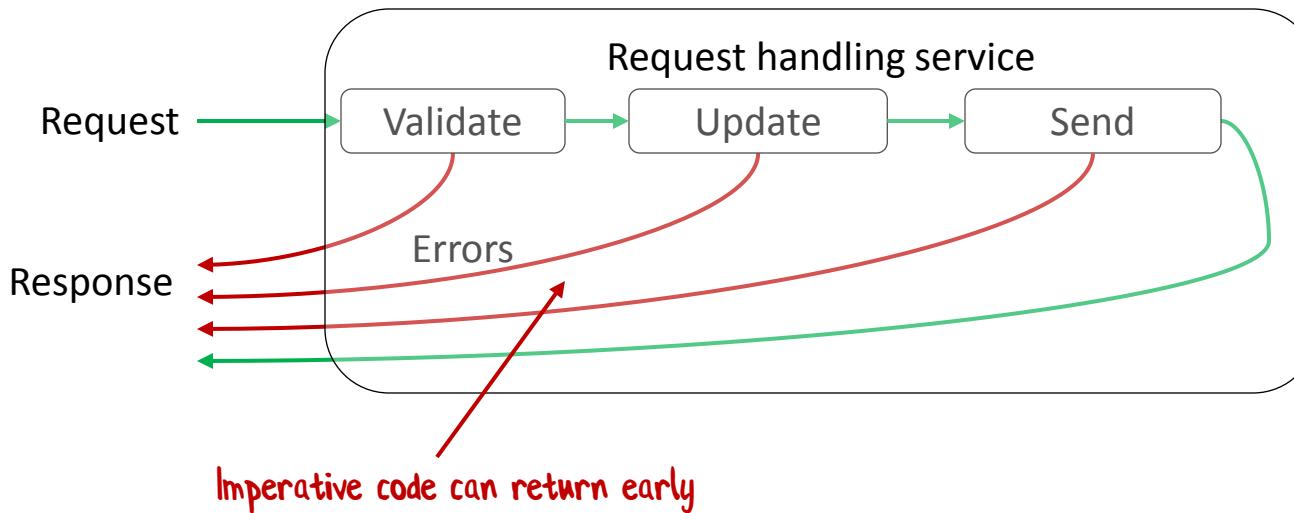
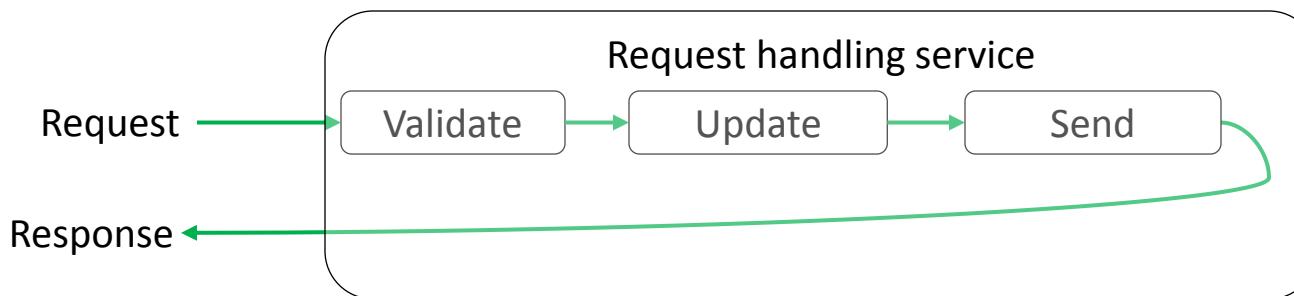
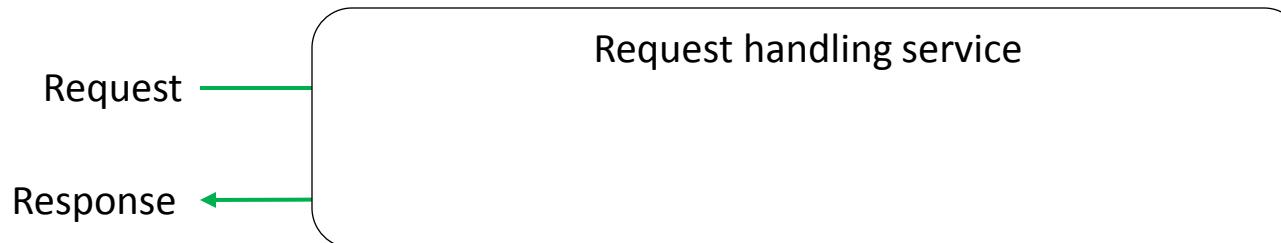
Don't believe me? Stay tuned!

Q: What is the functional equivalent of this code?

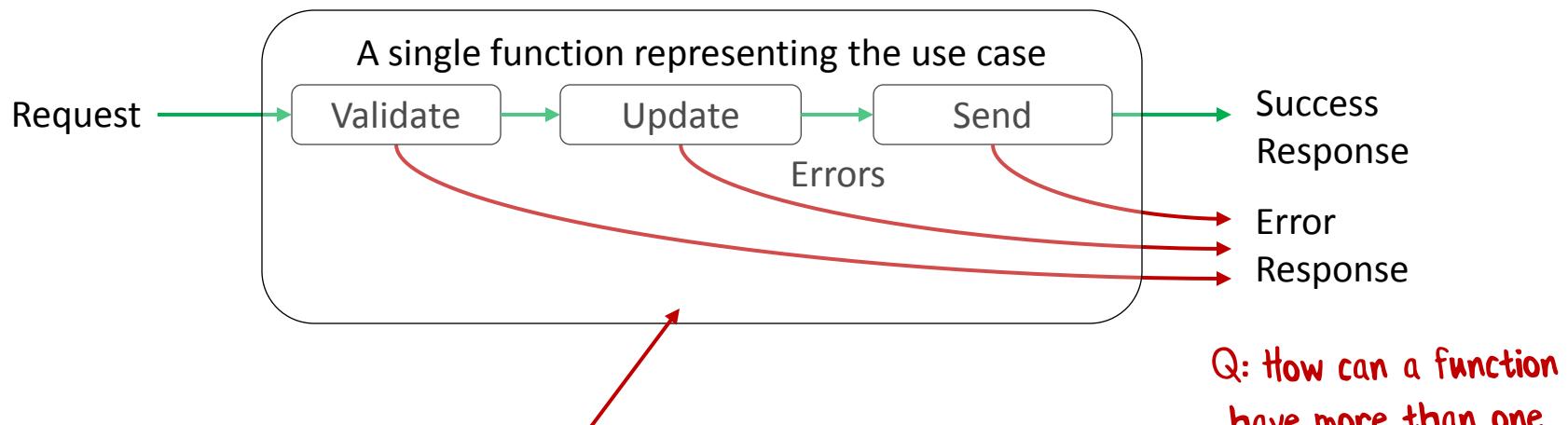
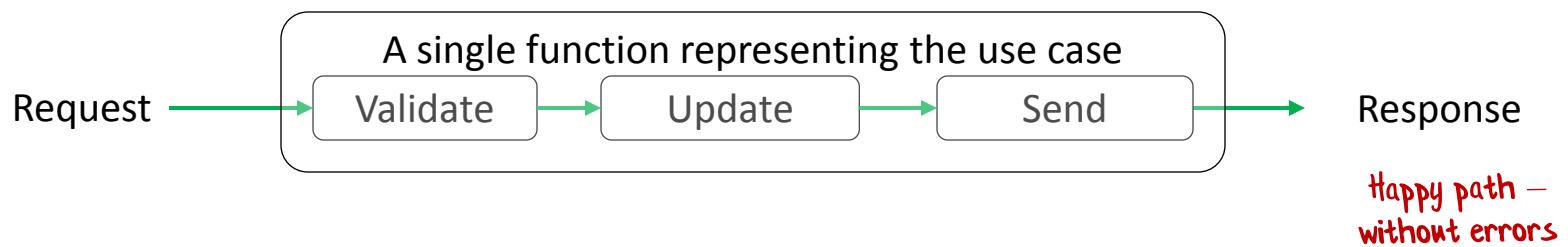
... and can we preserve the elegance of the original functional version?

# Designing the unhappy path

# Request/response (non-functional) design



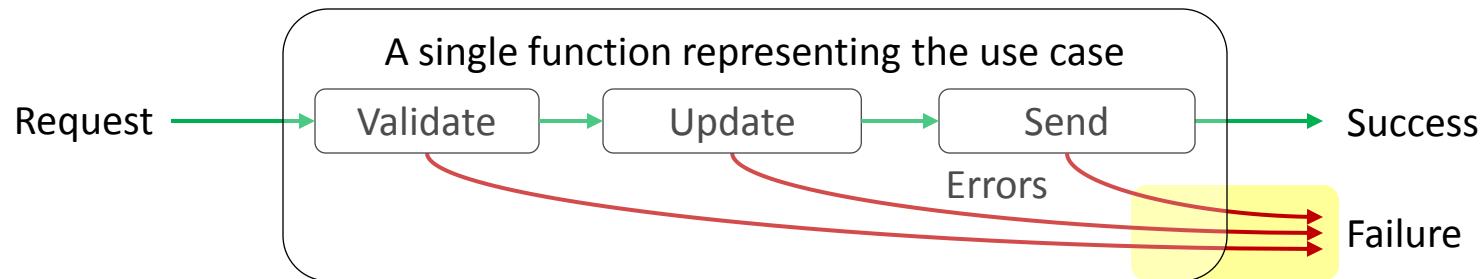
# Data flow (functional) design



Q: How can you bypass downstream functions when an error happens?

Q: How can a function have more than one output?

# Functional design

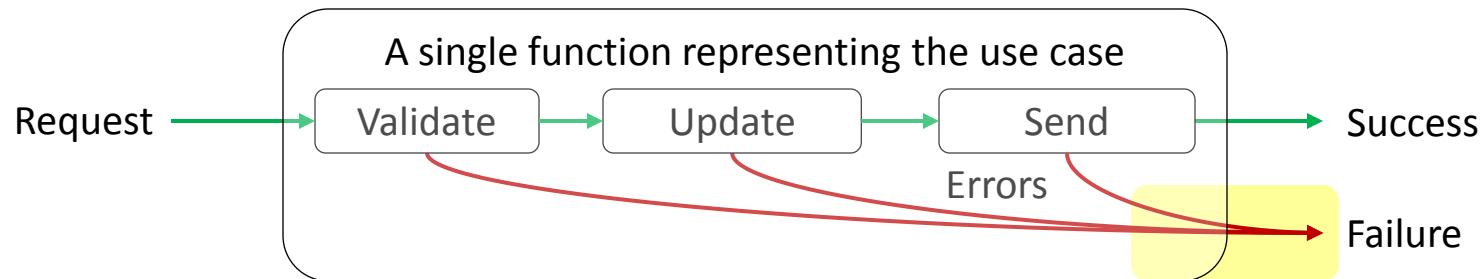


How can a function have more than one output?

```
type Result =  
| Success  
| ValidationError  
| UpdateError  
| SmtpError
```

I love sum types!  
But maybe too specific for this case?

# Functional design

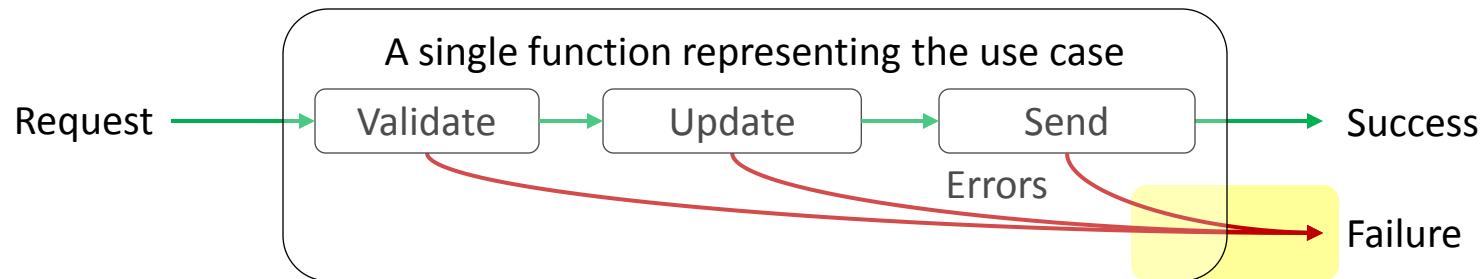


How can a function have more than one output?

```
type Result =  
| Success  
| Failure
```

Much more generic – but no data!

# Functional design

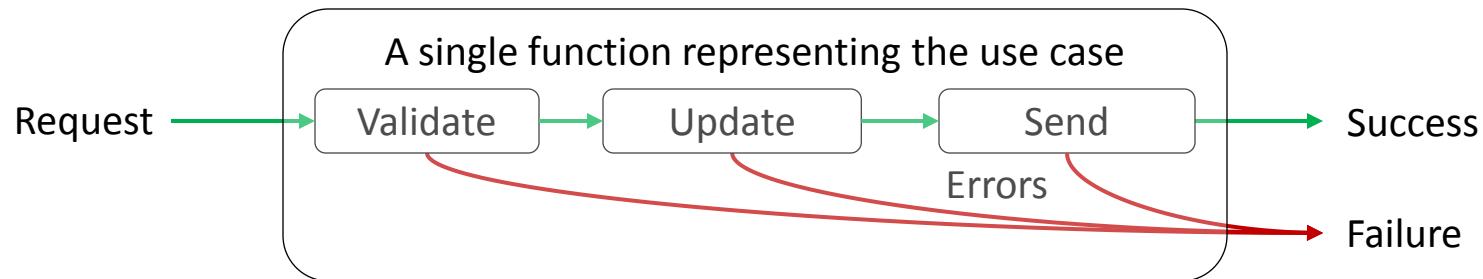


How can a function have more than one output?

```
type Result<'TEntity> =  
| Success of 'TEntity  
| Failure of string
```

Good for now – we'll revisit this design later.

# Functional design



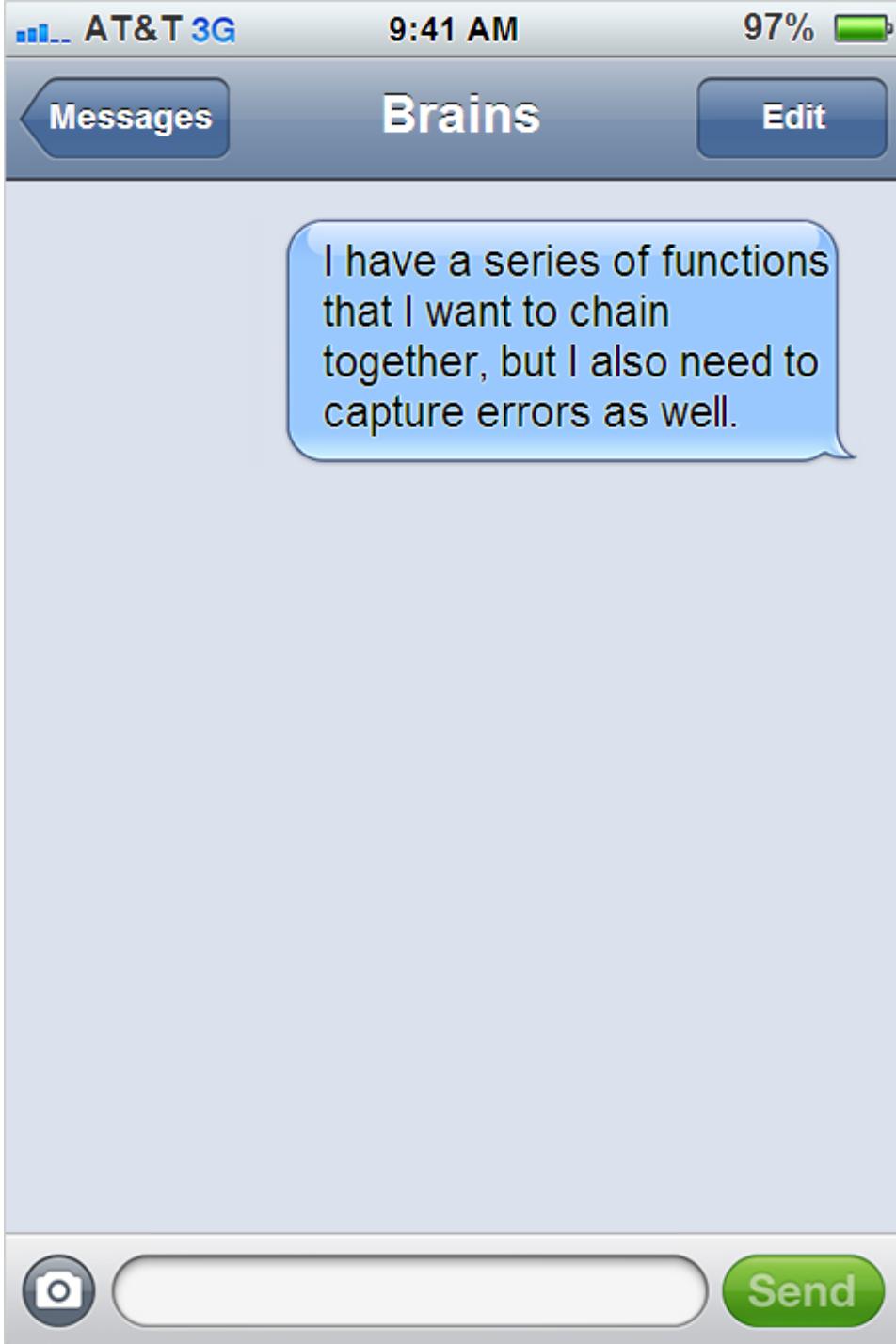
- Each use case will be equivalent to a single function
- The function will return a sum type with two cases: "Success" and "Failure".
- The use case function will be built from a series of smaller functions, each representing one step in a data flow.
- The errors from each step will be combined into a **single** "failure" path.

But we haven't answered the question:  
How can you bypass downstream functions  
when an error happens?

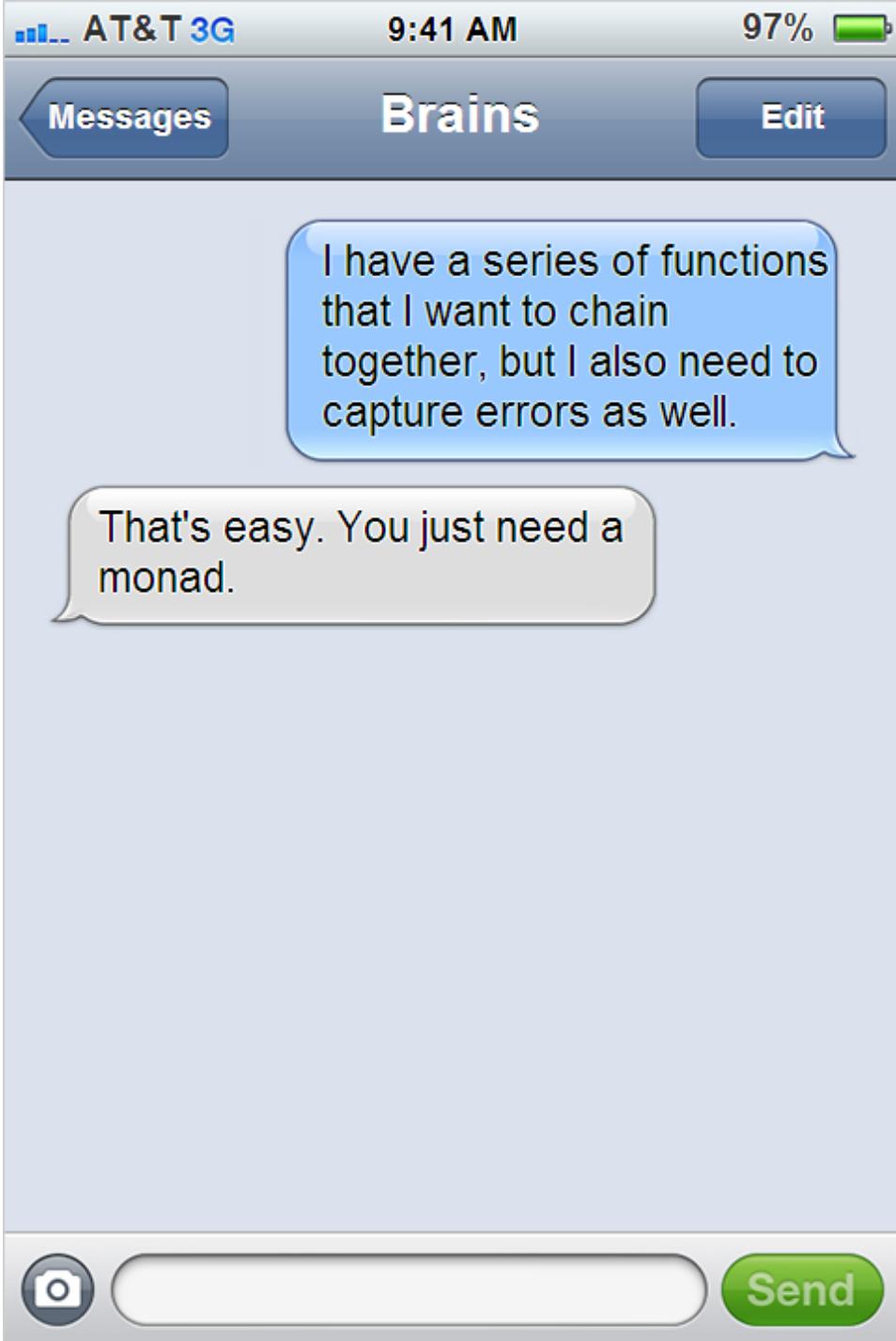
How do I work with errors  
in a functional way?

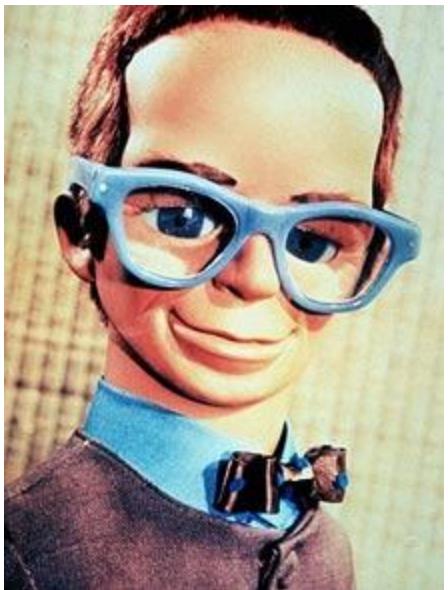


Very clever



A bear of very little brain





AT&T 3G 9:41 AM 97%

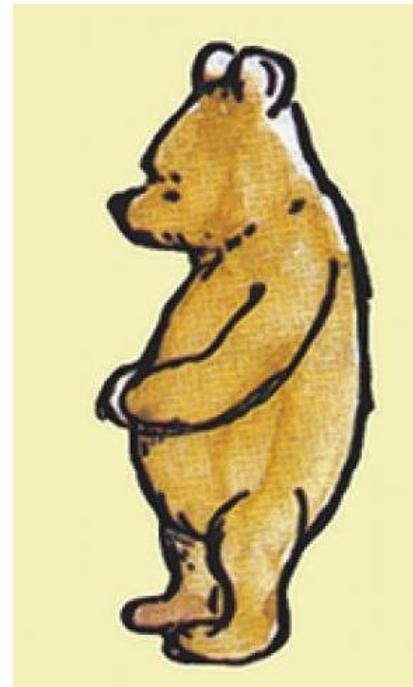
Messages Brains Edit

I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

That sounds complicated! What's a monad?

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

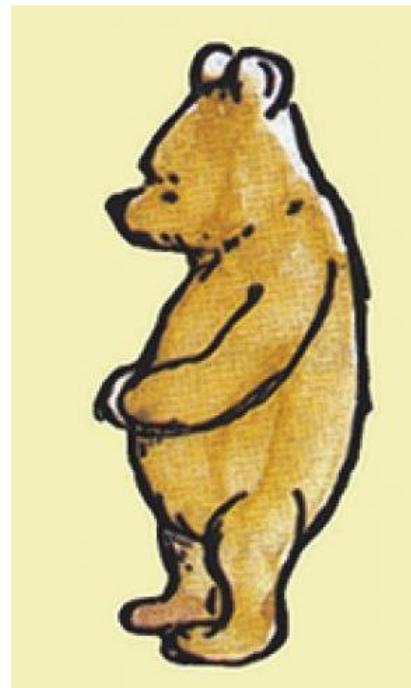
I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

That sounds complicated! What's a monad?

A monad is just a monoid in the category of endofunctors.

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

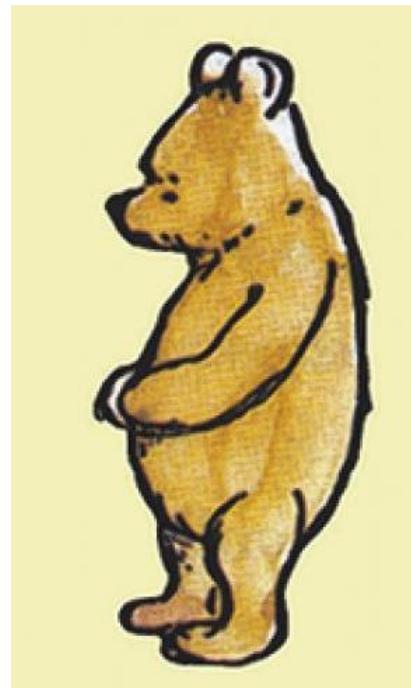
That sounds complicated! What's a monad?

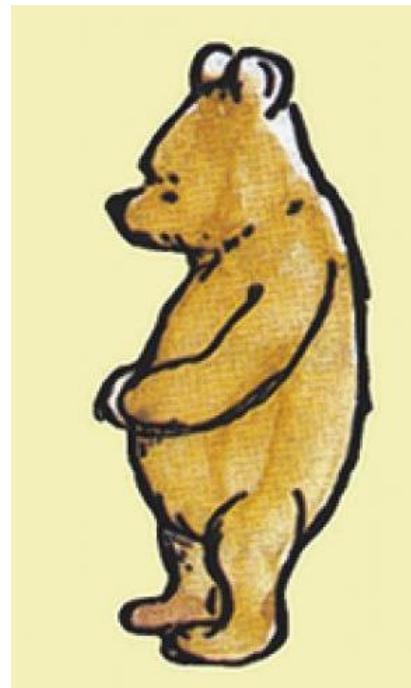
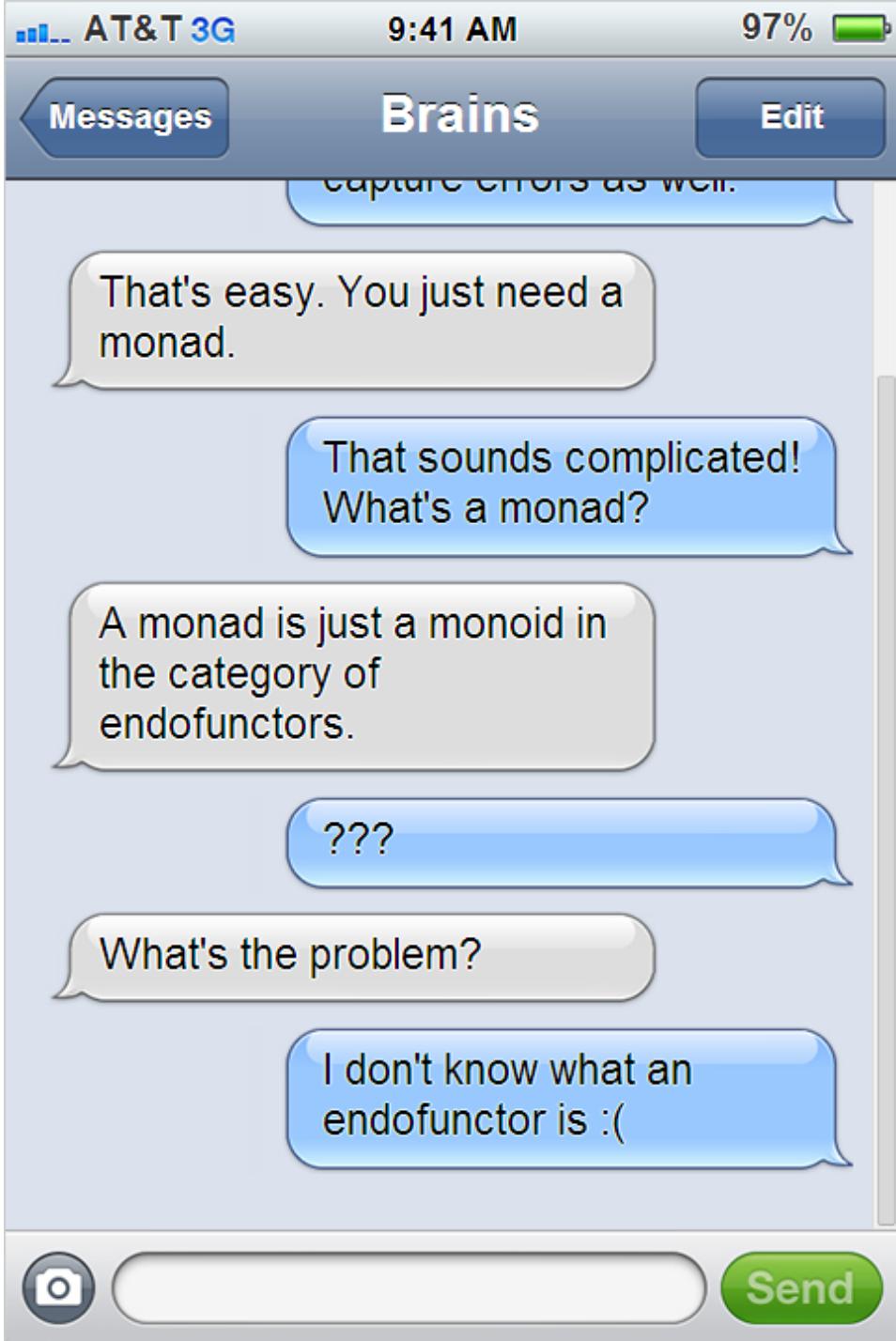
A monad is just a monoid in the category of endofunctors.

???

What's the problem?

Send







AT&T 3G 9:41 AM 97%

Messages Brains Edit

A monad is just a monoid in the category of endofunctors.

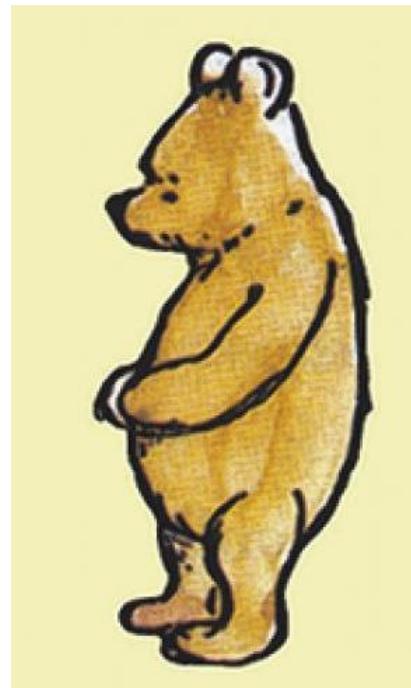
???

What's the problem?

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

the category of endofunctors.

???

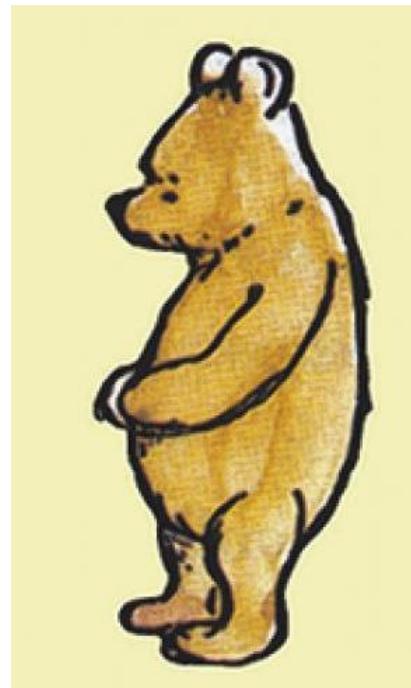
What's the problem?

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

???

What's the problem?

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

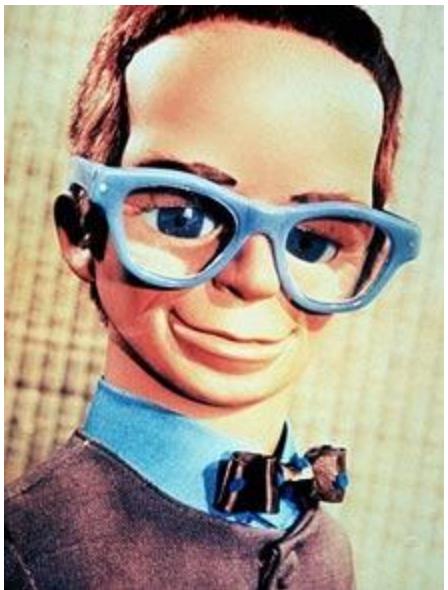
Of course! I understand completely now.

Send

A screenshot of an iPhone messaging screen. The title of the conversation is "Brains". The messages are as follows:

- ??? (blue bubble)
- What's the problem? (light gray bubble)
- I don't know what an endofunctor is :( (blue bubble)
- That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself. (light gray bubble)
- Simples! (blue bubble)
- Of course! I understand completely now. (blue bubble)

The bottom of the screen shows a camera icon, a text input field, and a green "Send" button.



AT&T 3G 9:41 AM 97%

Messages Brains Edit

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

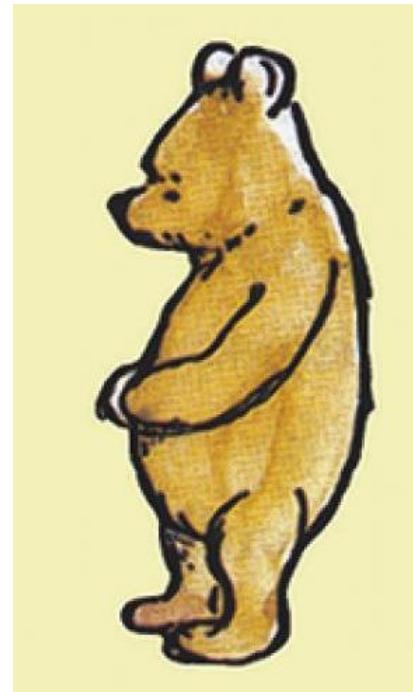
Of course! I understand completely now.

But, seriously, what do I have to do?

Send

A screenshot of a mobile phone messaging interface. The top bar shows signal strength, network (AT&T 3G), time (9:41 AM), battery level (97%), and a green battery icon. Below the top bar is a header with 'Messages' on the left, the contact name 'Brains' in the center, and 'Edit' on the right. The main area contains a text conversation:

- Person 1 (blue bubble): "I don't know what an endofunctor is :(
- Person 2 (grey bubble): "That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself."
- Person 1 (white bubble): "Simples!"
- Person 2 (blue bubble): "Of course! I understand completely now."
- Person 1 (blue bubble): "But, seriously, what do I have to do?"

At the bottom of the screen are standard messaging controls: a camera icon, a text input field, and a green 'Send' button.



AT&T 3G 9:41 AM 97%

Messages Brains Edit

homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

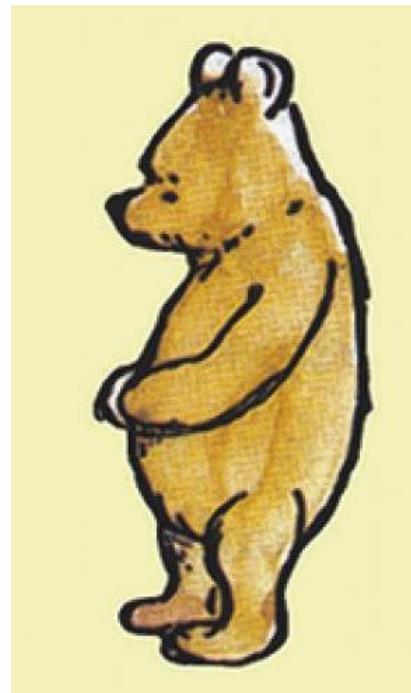
Simples!

Of course! I understand completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

endofunctor is just a functor that maps a category to itself.

Simples!

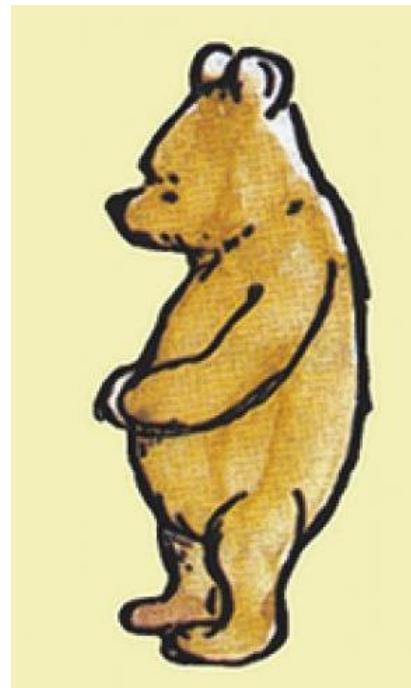
Of course! I understand completely now.

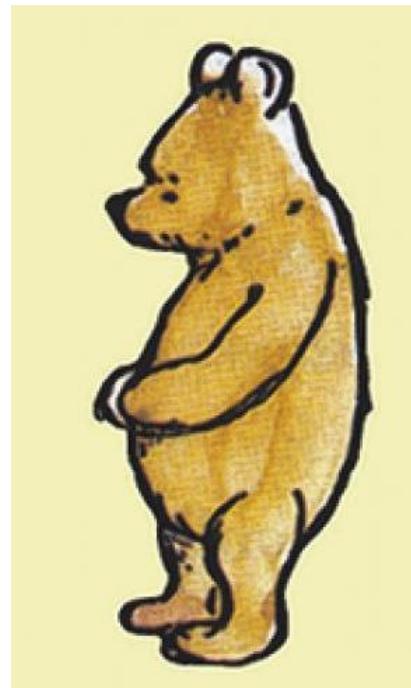
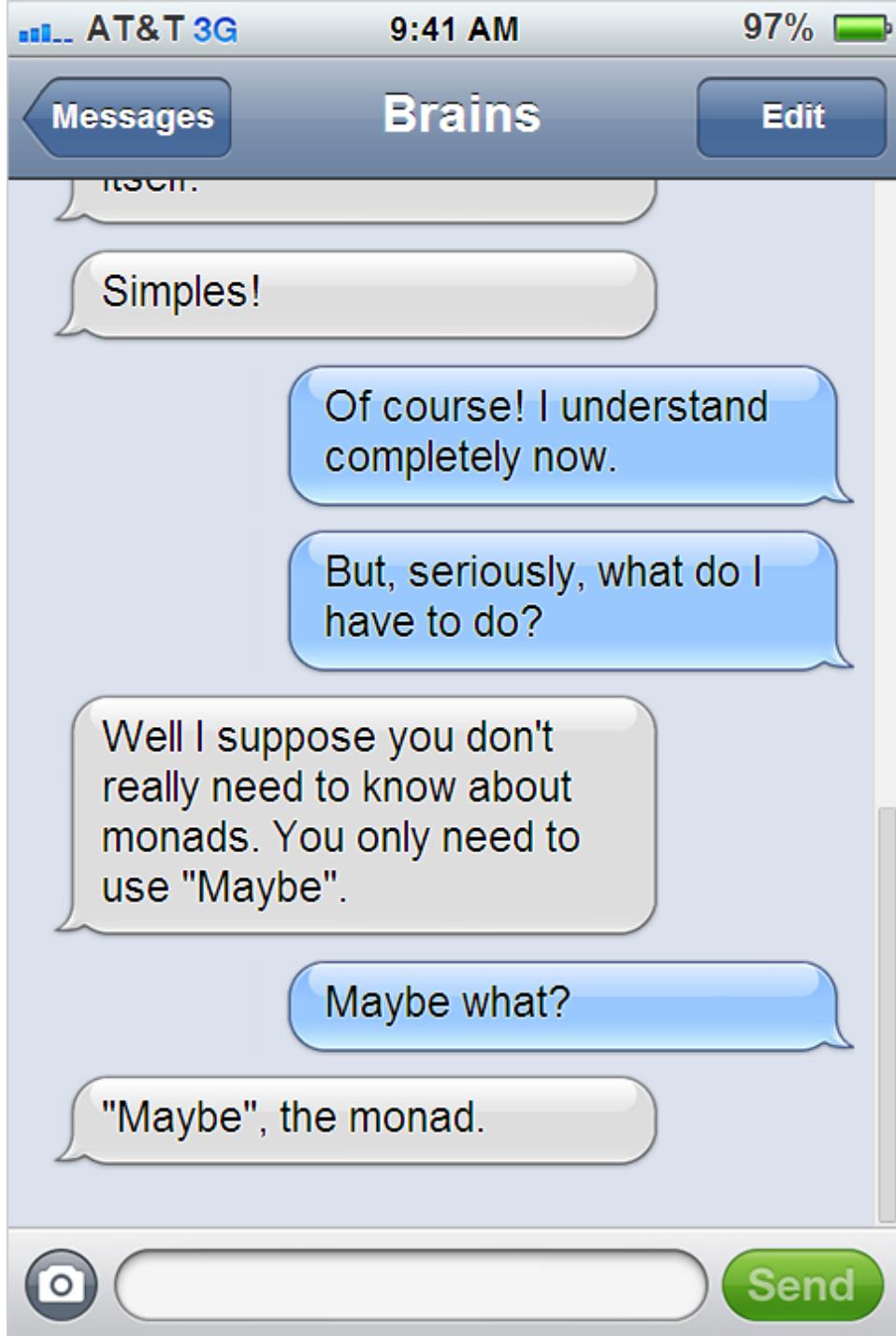
But, seriously, what do I have to do?

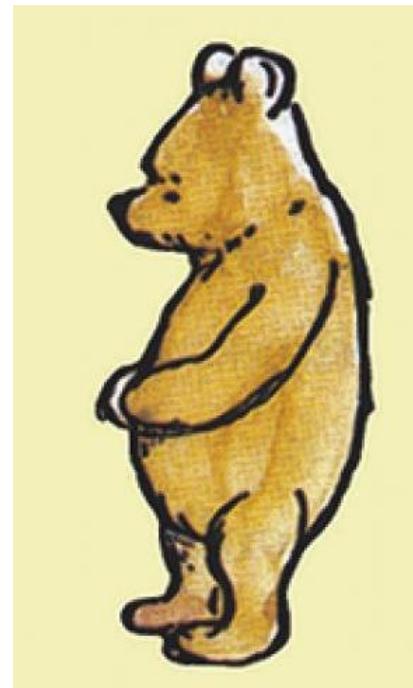
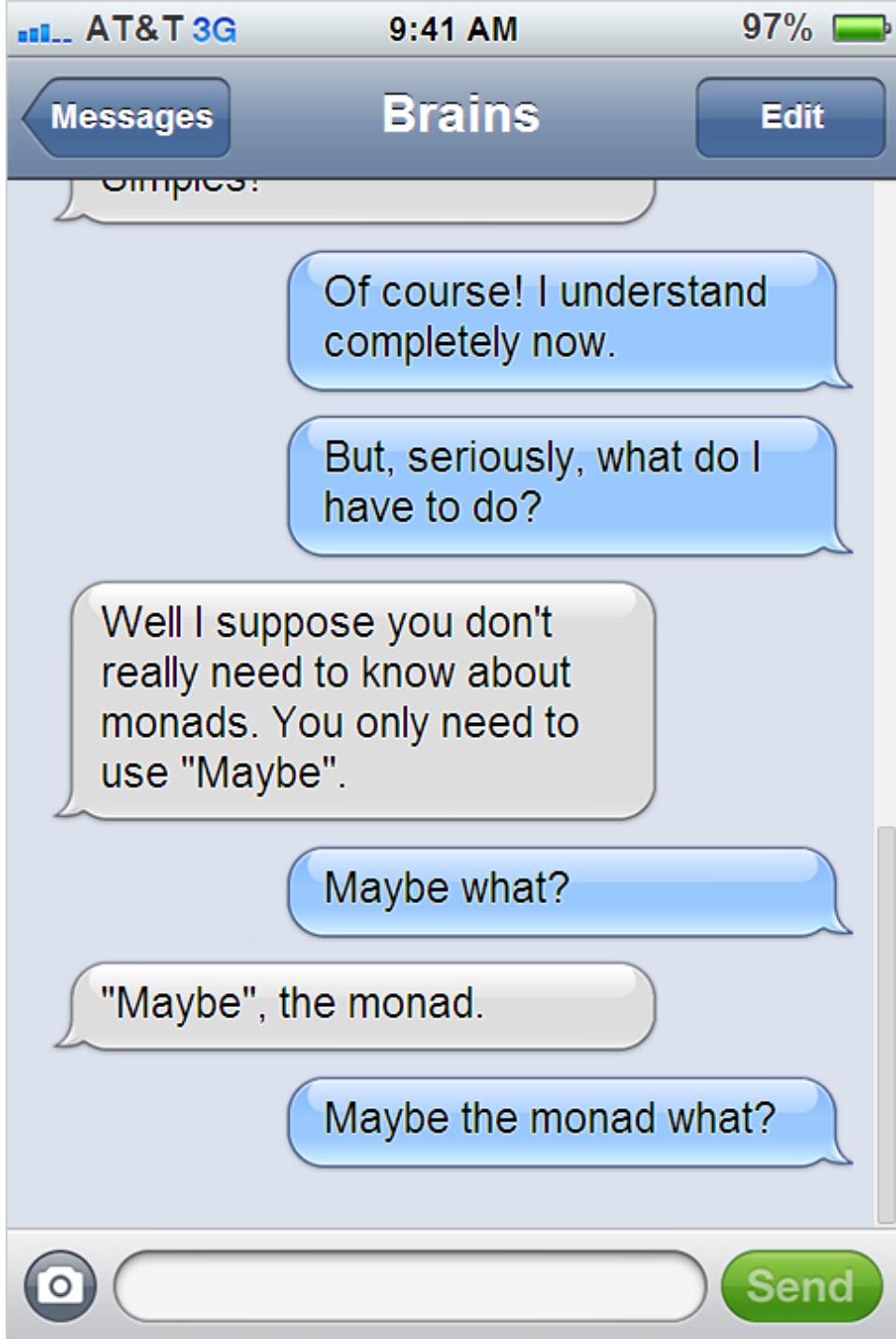
Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

Send









AT&T 3G 9:41 AM 97%

Messages Brains Edit

completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

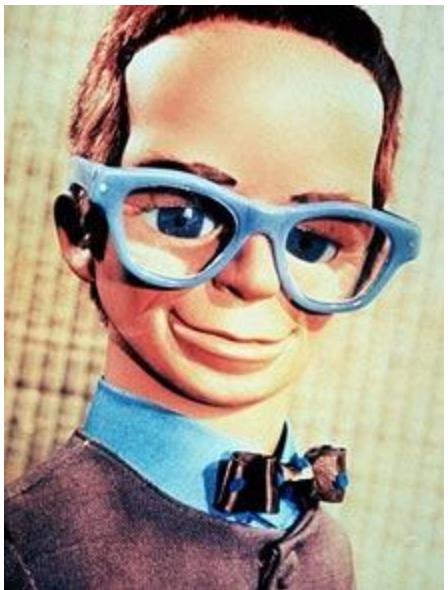
"Maybe", the monad.

Maybe the monad what?

"Maybe" is the \*name\* of the monad

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

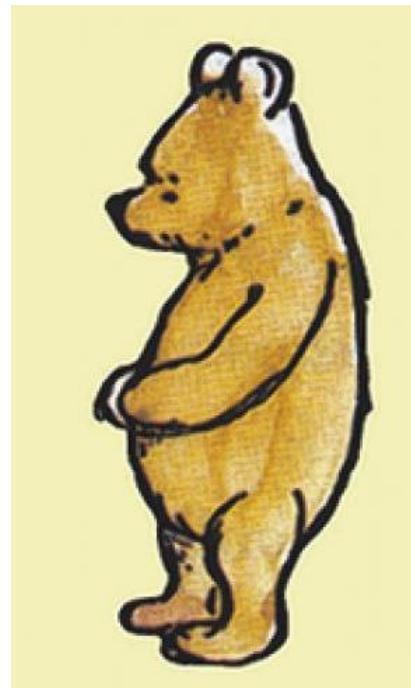
"Maybe", the monad.

Maybe the monad what?

"Maybe" is the \*name\* of the monad

Don't you mean, "Maybe the name of the monad is..."

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

use Maybe .

Maybe what?

"Maybe", the monad.

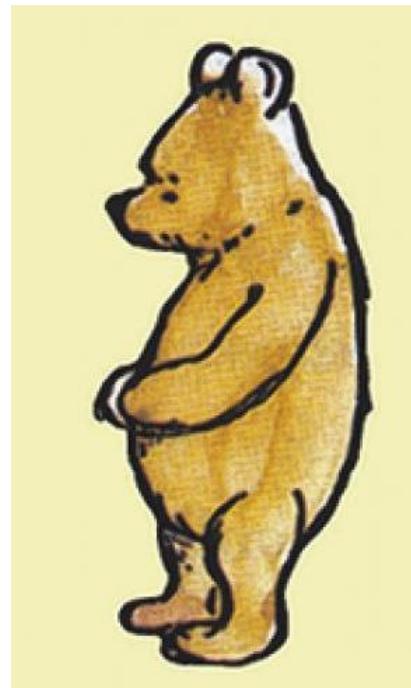
Maybe the monad what?

"Maybe" is the \*name\* of the monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

"Maybe", the monad.

Maybe the monad what?

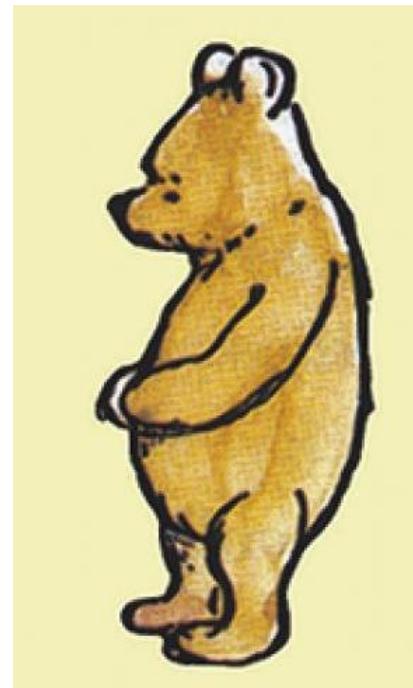
"Maybe" is the \*name\* of the monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

"Maybe" is the \*name\* of the monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

monad

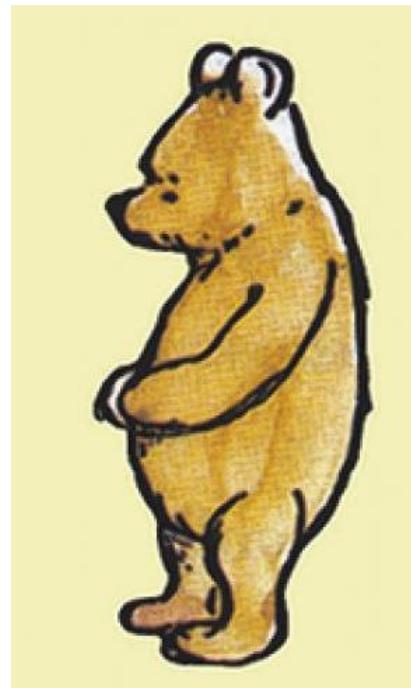
Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?





AT&T 3G 9:41 AM 97%

Messages Brains Edit

the name of the monad  
is..."

"Maybe" the name of the  
monad is? You're talking just  
like Yoda.

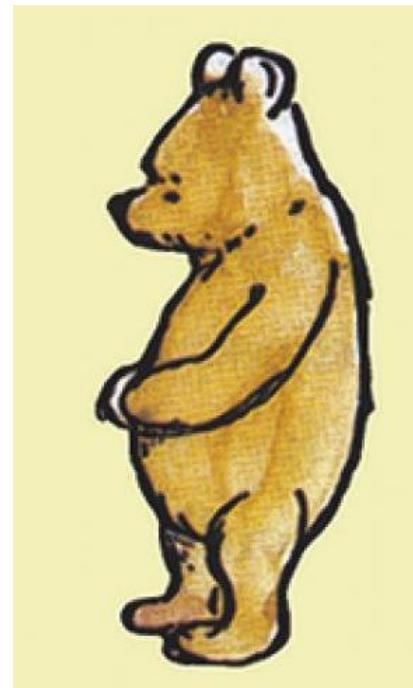
No, talking like Yoda you  
are!

Getting back on topic....  
"Maybe" is certainly what  
you want.

Definitely Maybe, then?

Actually, I prefer "(What's  
the Story) Morning Glory?"

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

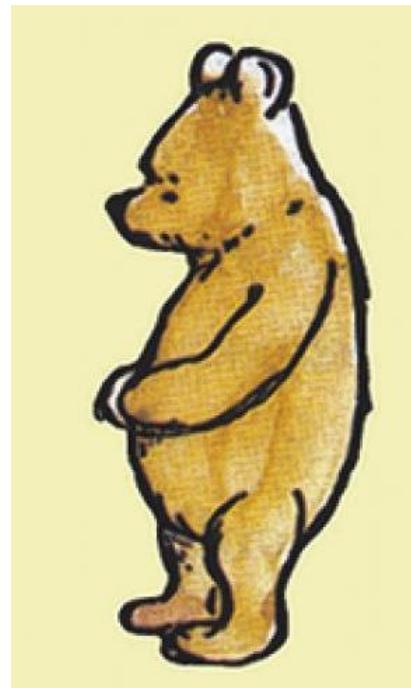
Getting back on topic...  
"Maybe" is certainly what you want.

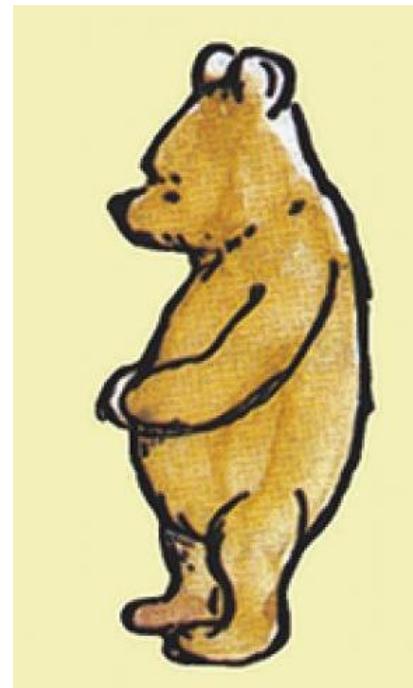
Definitely Maybe, then?

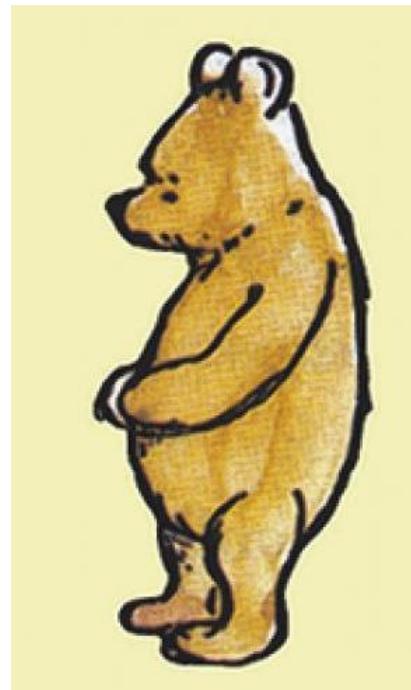
Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Send









AT&T 3G 9:41 AM 97%

Messages Brains Edit

Getting back on topic...  
"Maybe" is certainly what you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

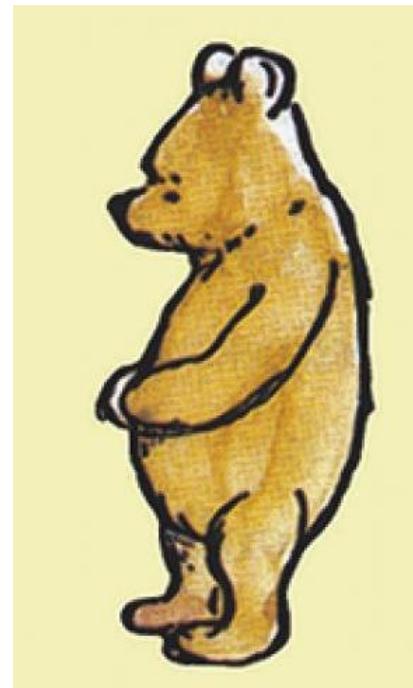
"Either", the monad

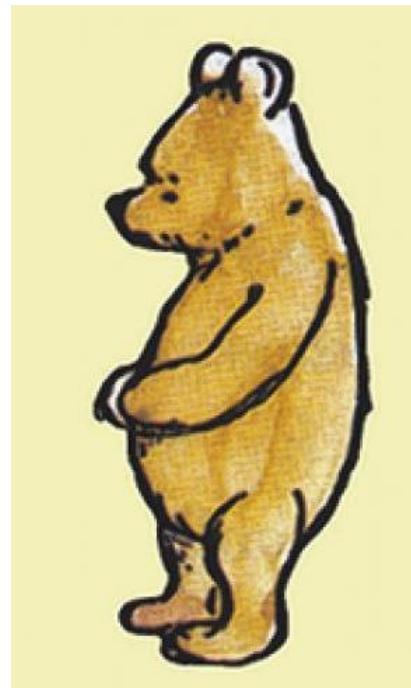
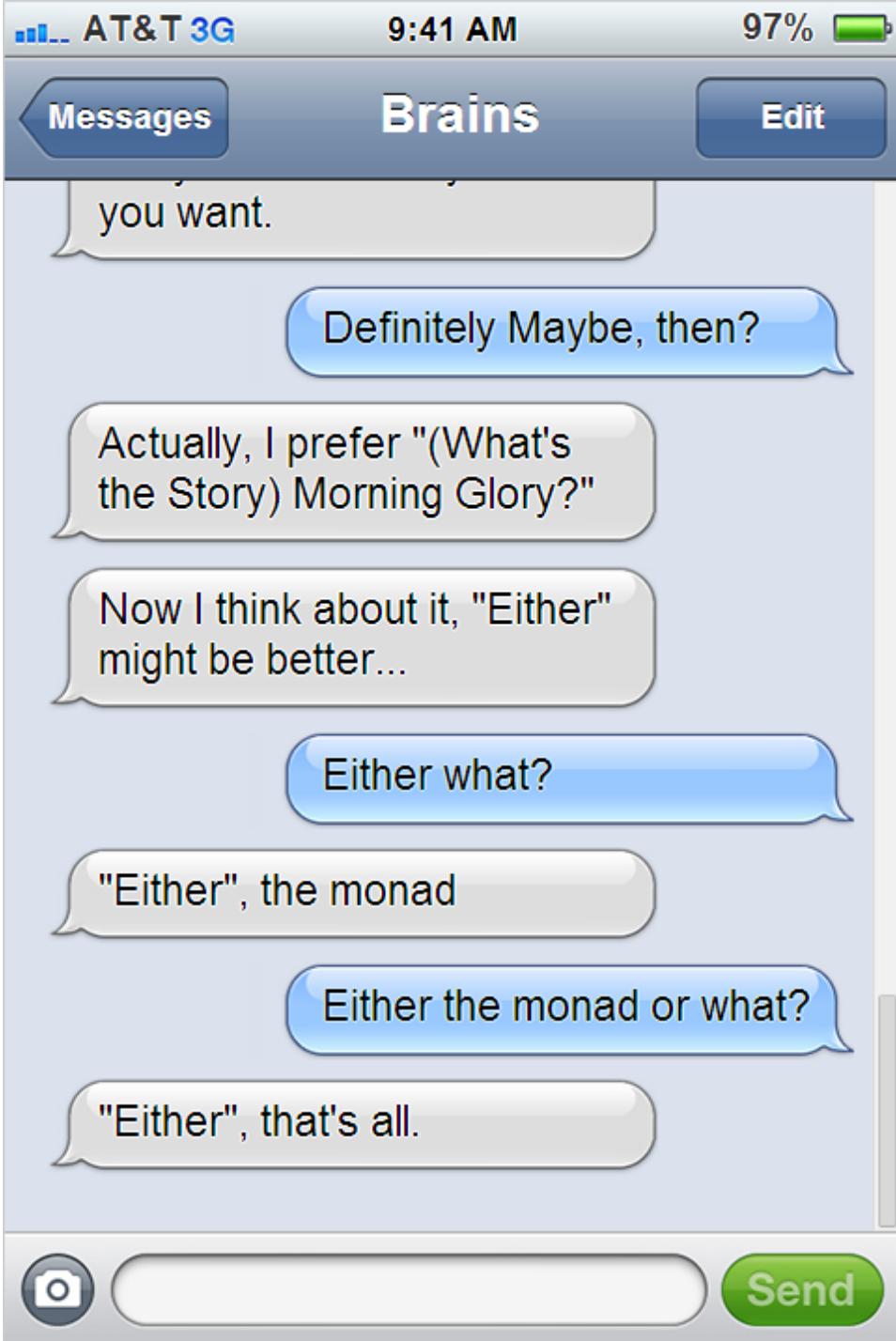
Either the monad or what?

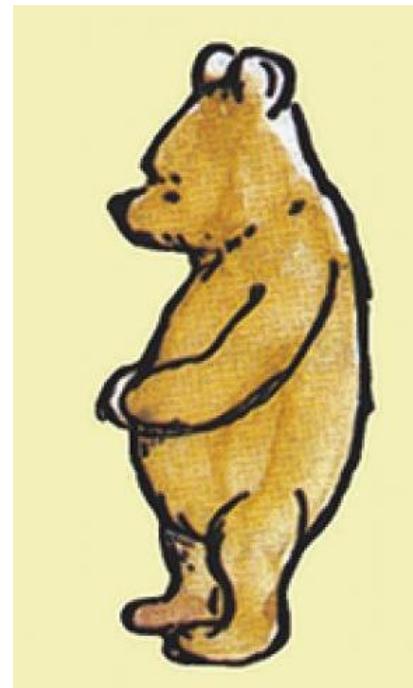
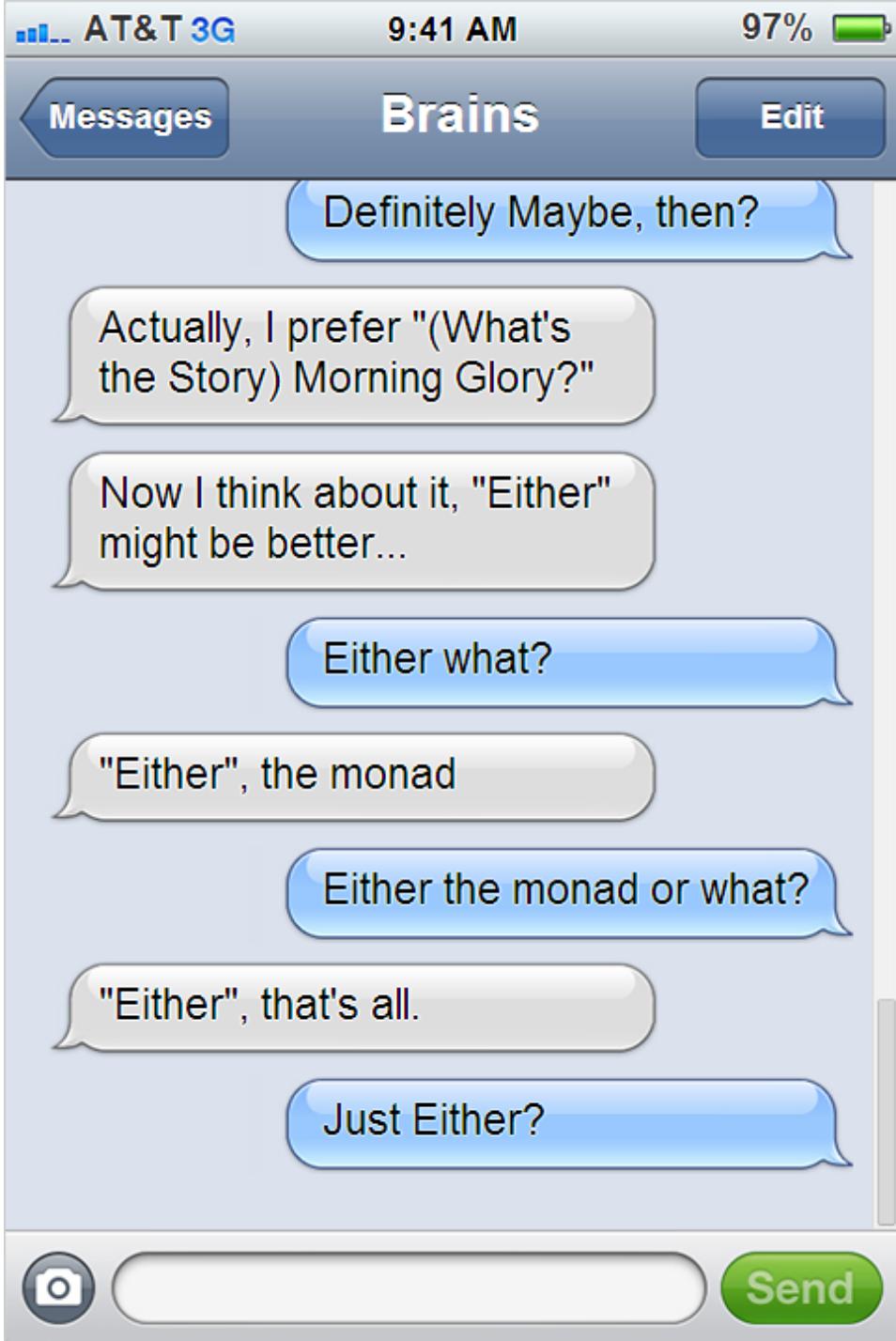
Send

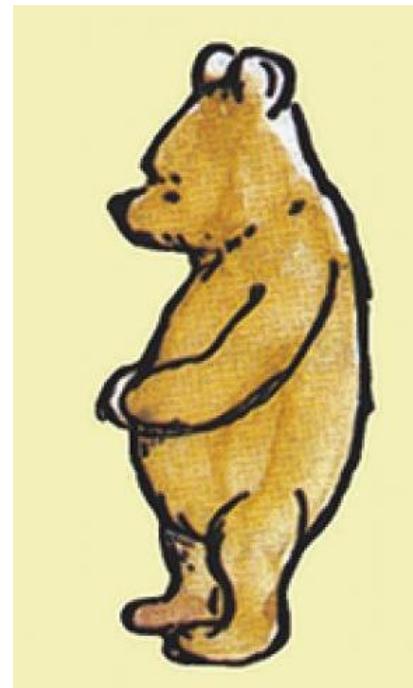
A screenshot of an iPhone messaging screen. The top status bar shows signal strength, AT&T 3G, the time 9:41 AM, and battery level at 97%. The message screen has tabs for 'Messages' and 'Brains', with 'Edit' on the right. The conversation is with a contact named 'Brains'. The messages are as follows:

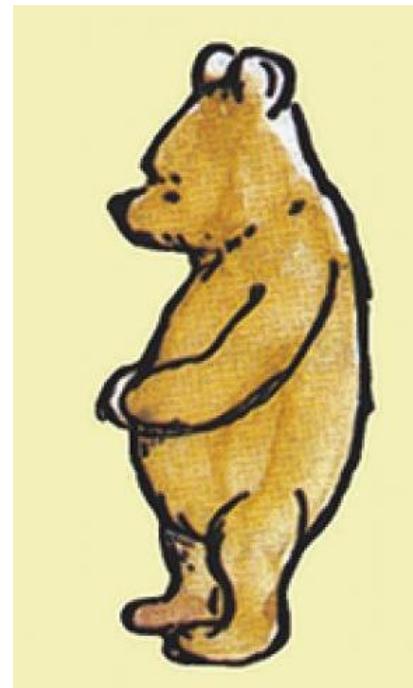
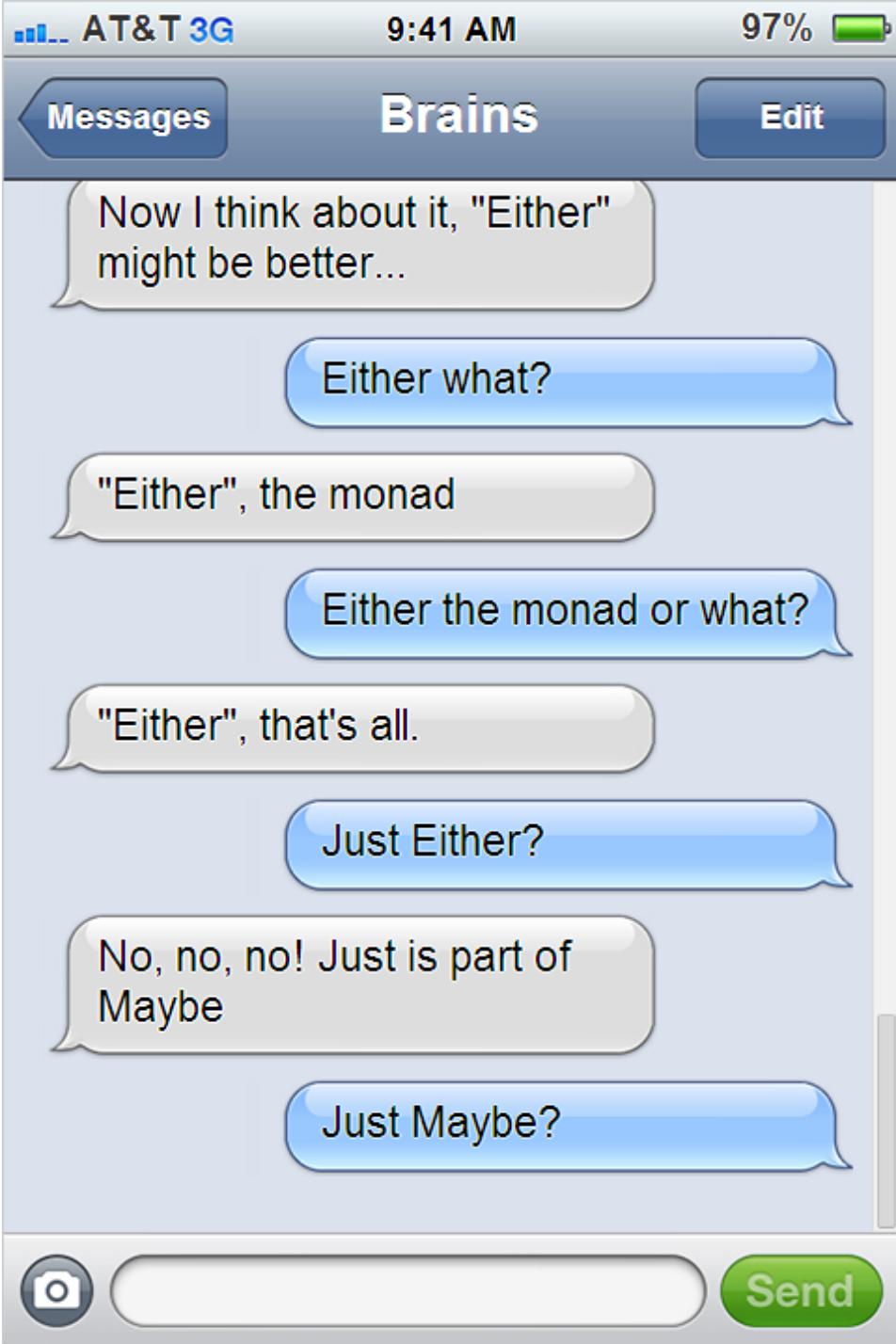
- Getting back on topic...  
"Maybe" is certainly what you want.
- Definitely Maybe, then?
- Actually, I prefer "(What's the Story) Morning Glory?"
- Now I think about it, "Either" might be better...
- Either what?
- "Either", the monad
- Either the monad or what?

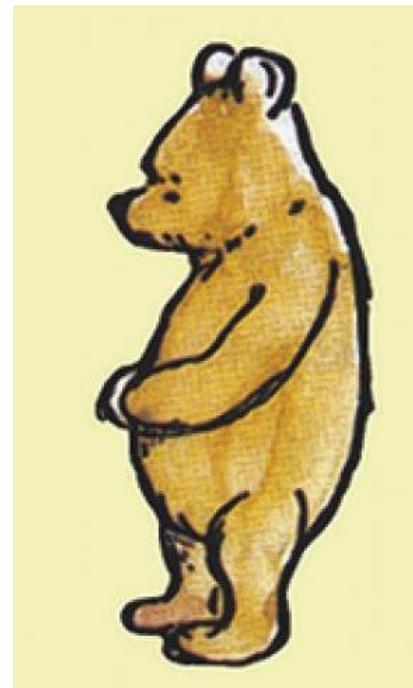
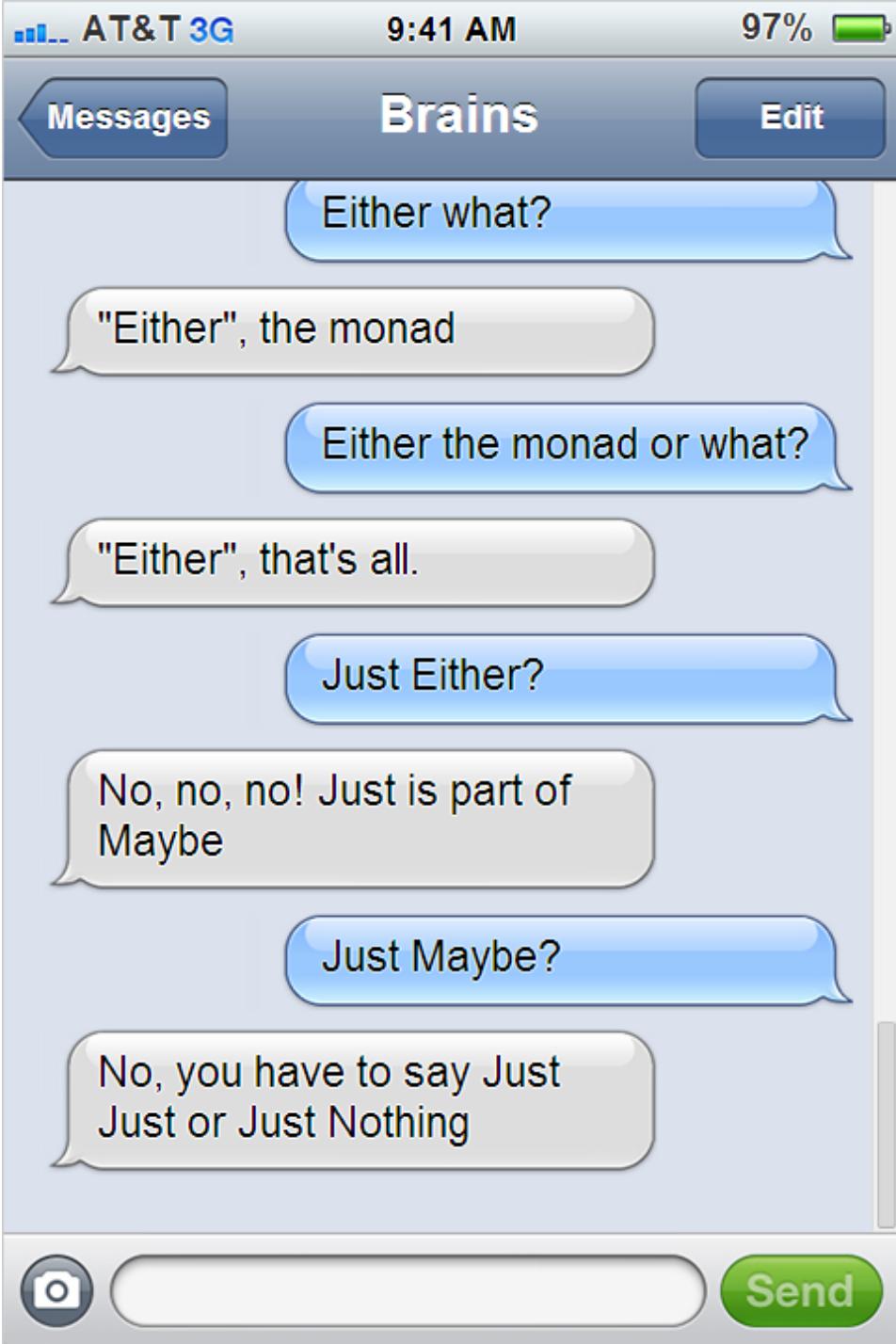
At the bottom are standard iOS message controls: a camera icon, a text input field, and a green 'Send' button.

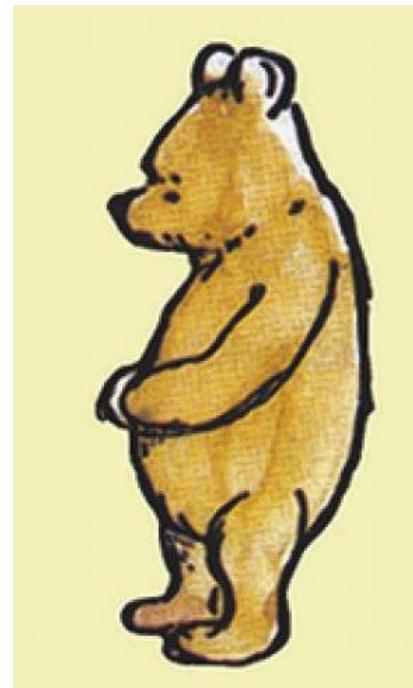


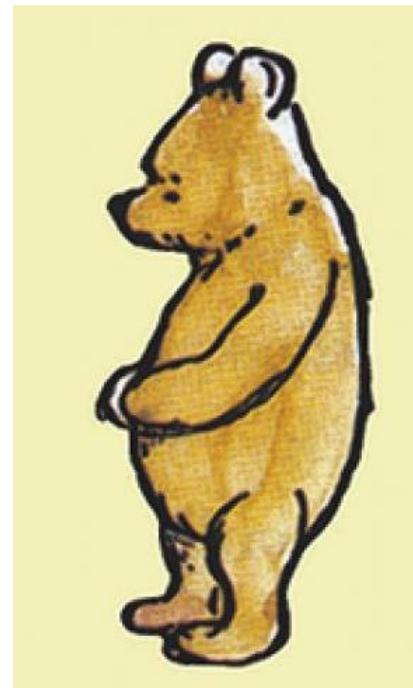
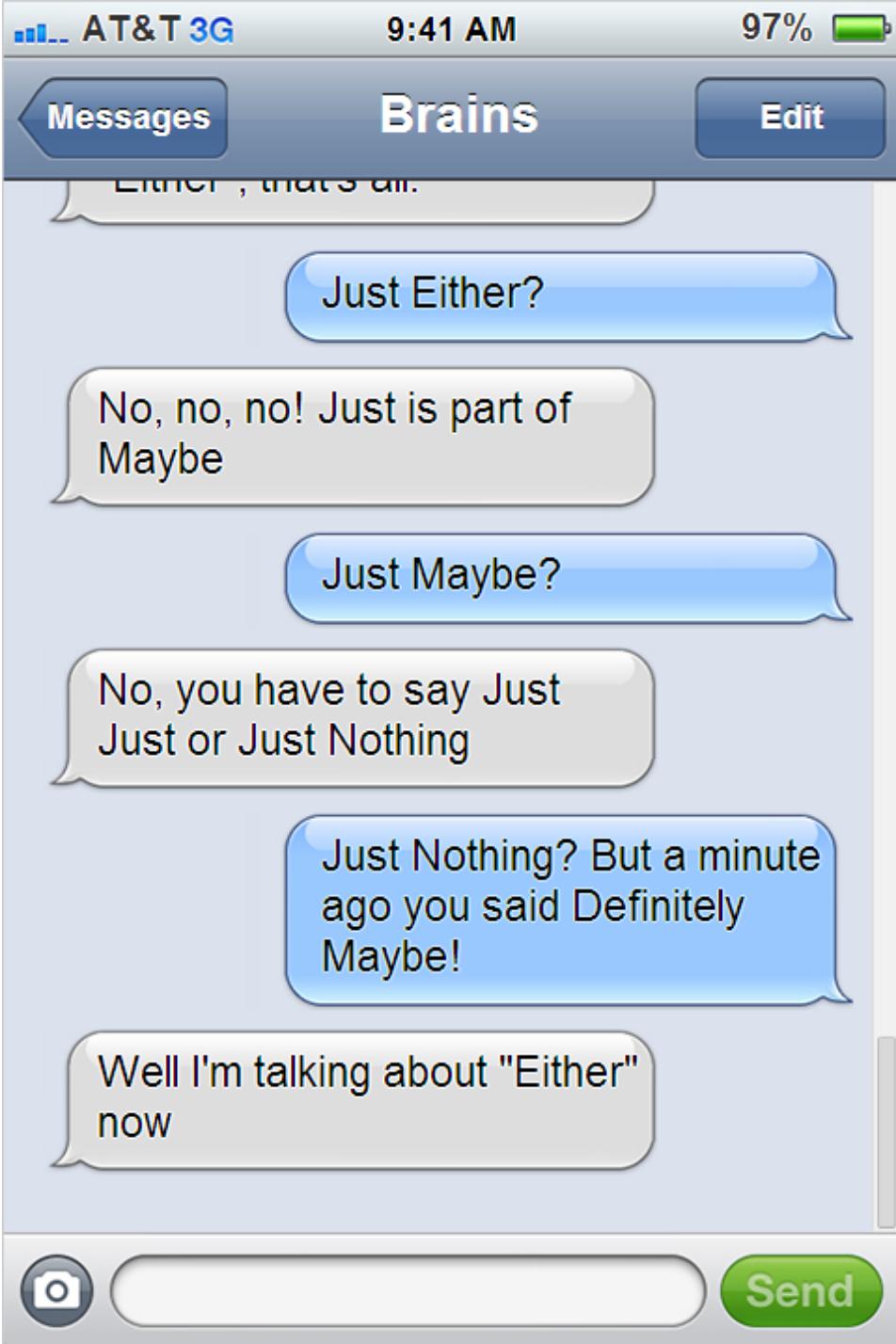


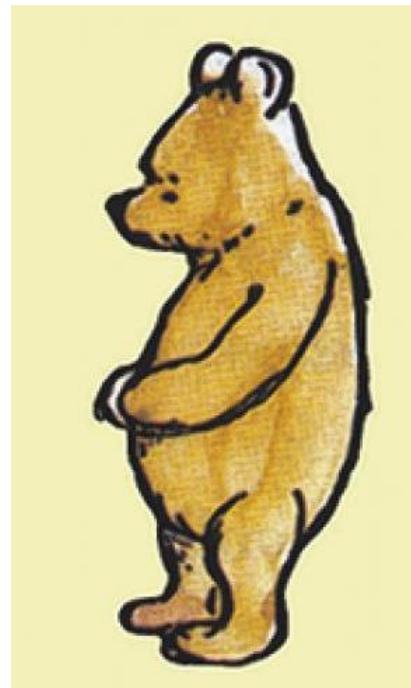
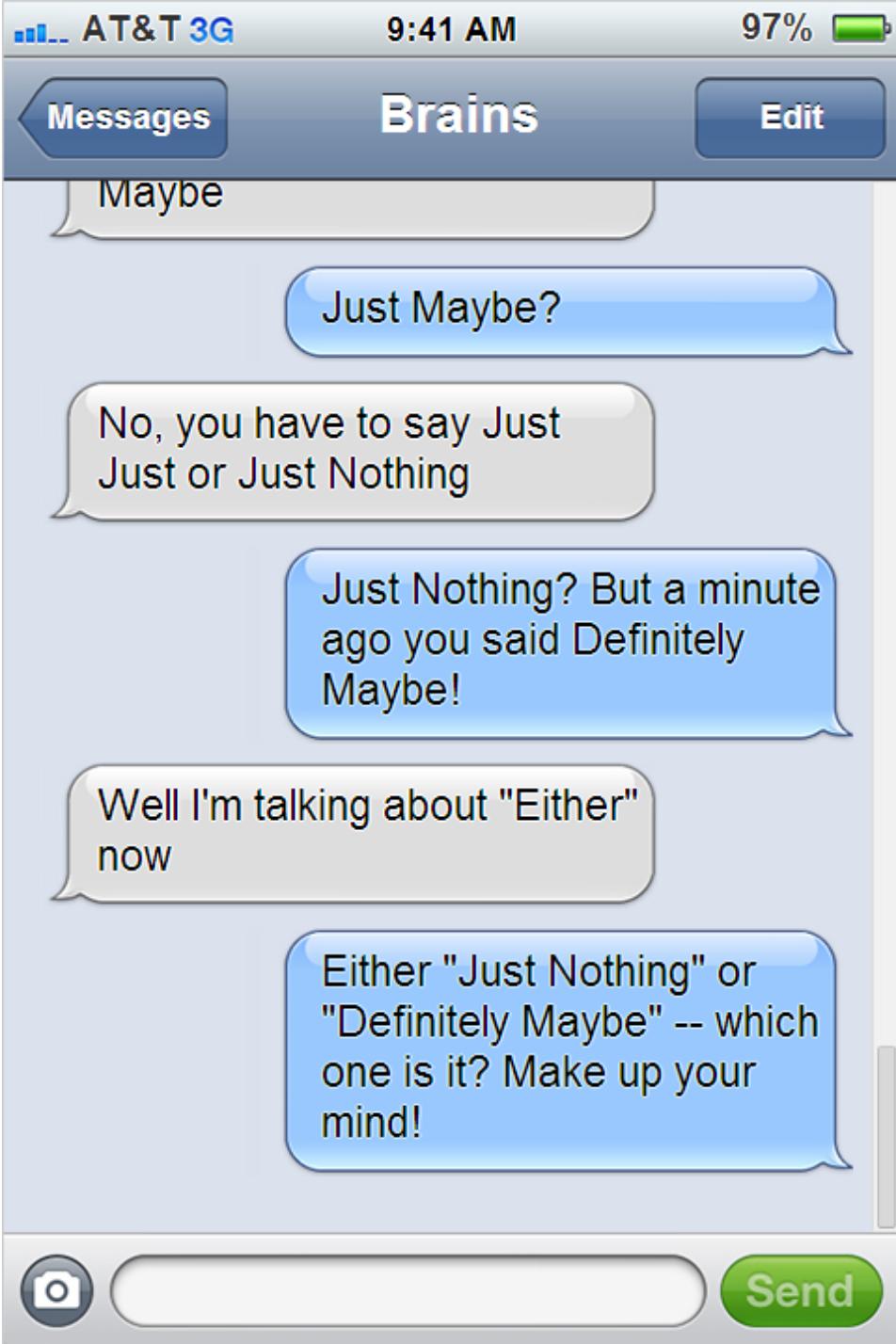
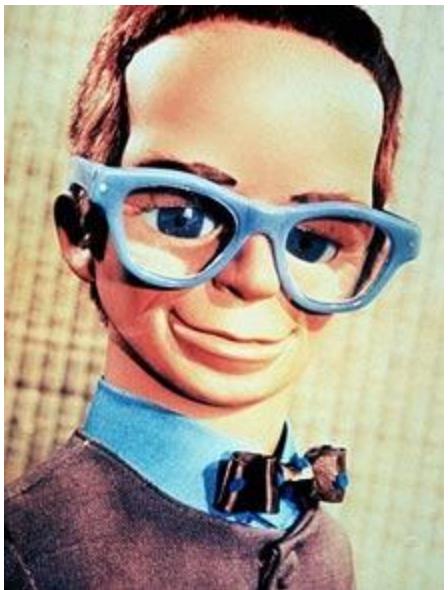














AT&T 3G 9:41 AM 97%

Messages Brains Edit

No, you have to say Just  
Just or Just Nothing

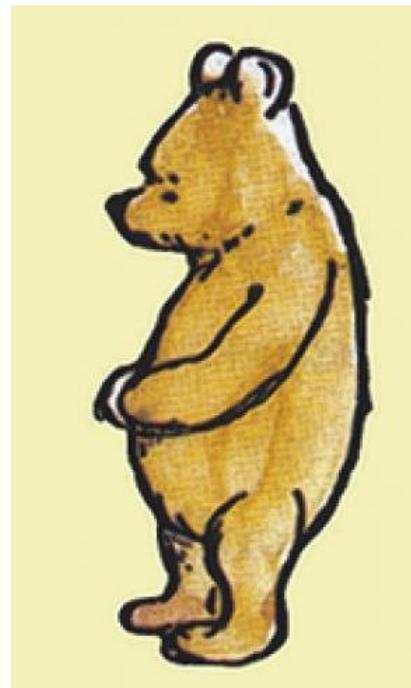
Just Nothing? But a minute  
ago you said Definitely  
Maybe!

Well I'm talking about "Either"  
now

Either "Just Nothing" or  
"Definitely Maybe" -- which  
one is it? Make up your  
mind!

Neither! I told you, you  
should use Either.

Send





AT&T 3G 9:41 AM 97%

Messages Brains Edit

Just or Just Nothing

Just Nothing? But a minute ago you said Definitely Maybe!

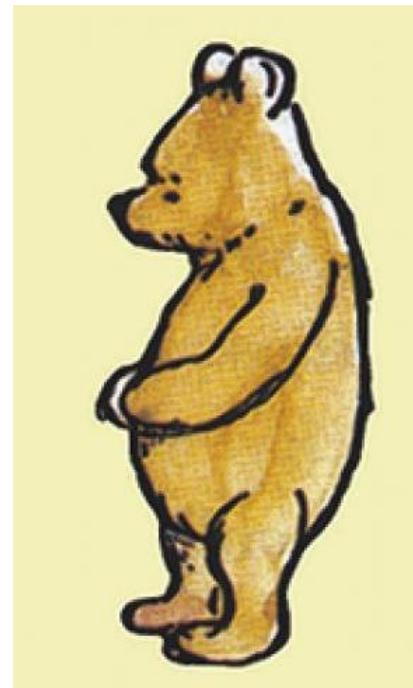
Well I'm talking about "Either" now

Either "Just Nothing" or "Definitely Maybe" -- which one is it? Make up your mind!

Neither! I told you, you should use Either.

Aaargh!

Send



# Monads are confusing?

Monads are not really that confusing

<http://bit.ly/monad-paper>

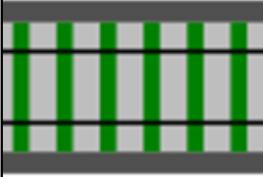
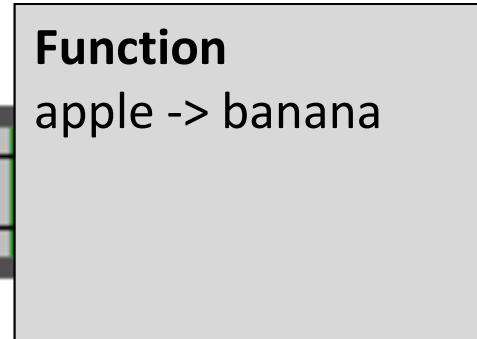
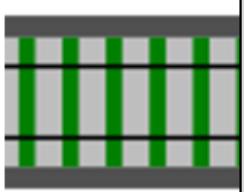


Read the original paper.

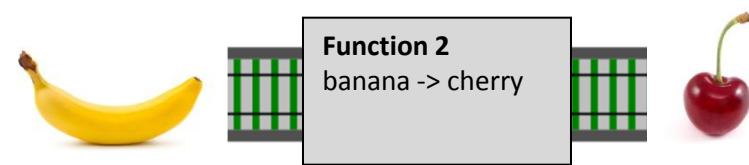
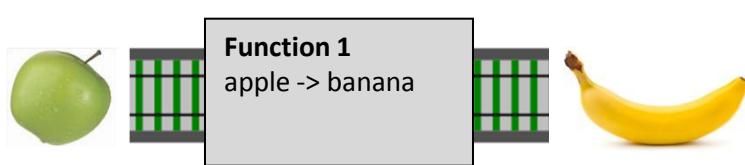
# Railway oriented programming

*This has absolutely nothing to do with monads.*

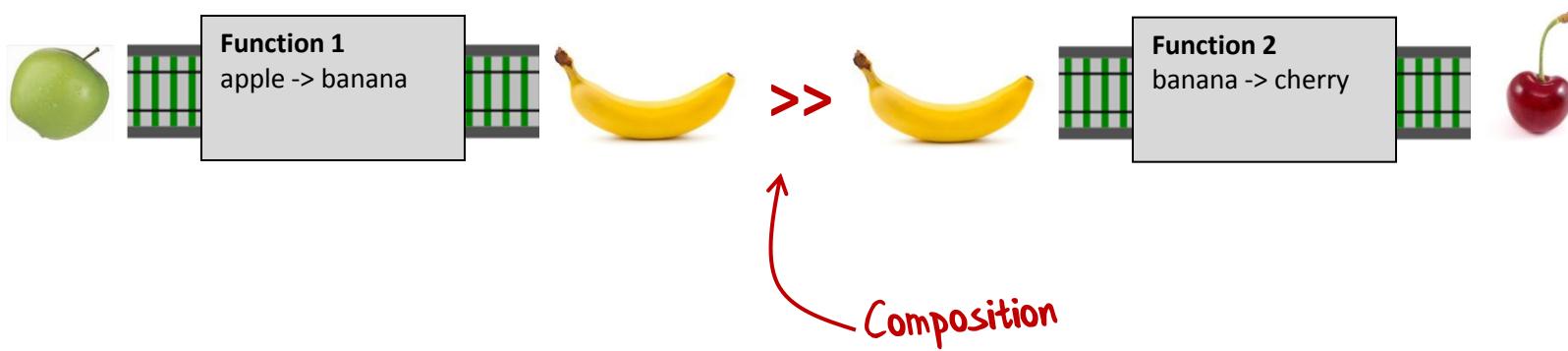
# A railway track analogy



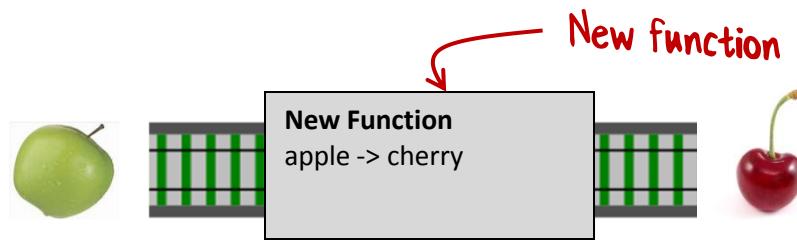
# A railway track analogy



# A railway track analogy



# A railway track analogy



Can't tell it was built from  
smaller functions!

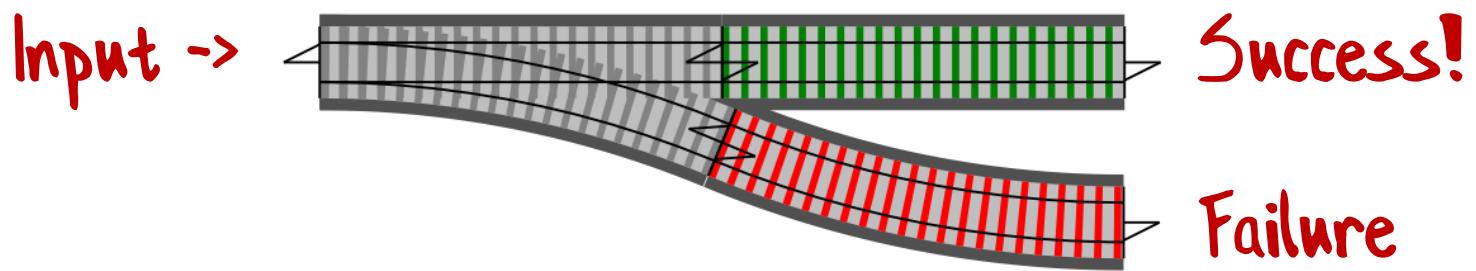
# An error generating function



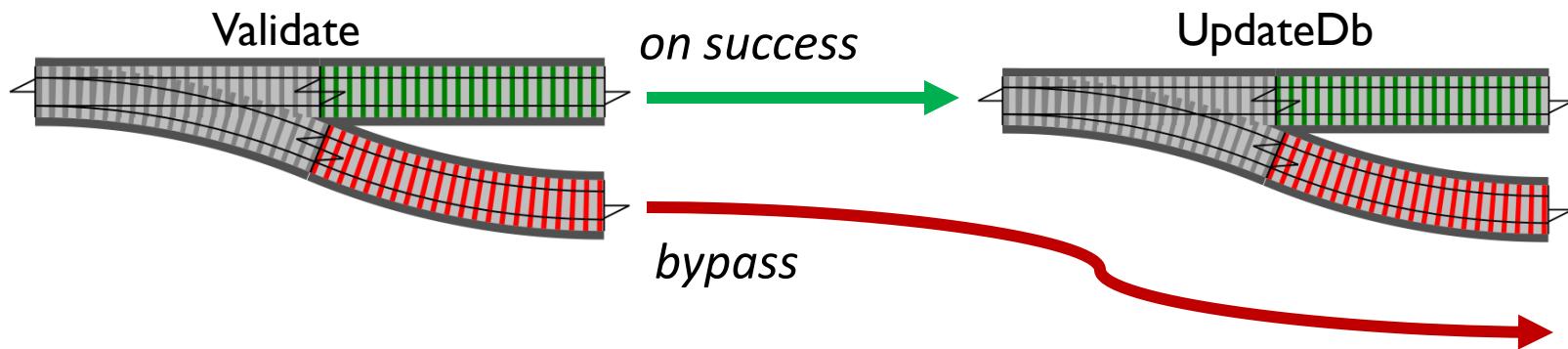
How do we model this  
as railway track?

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

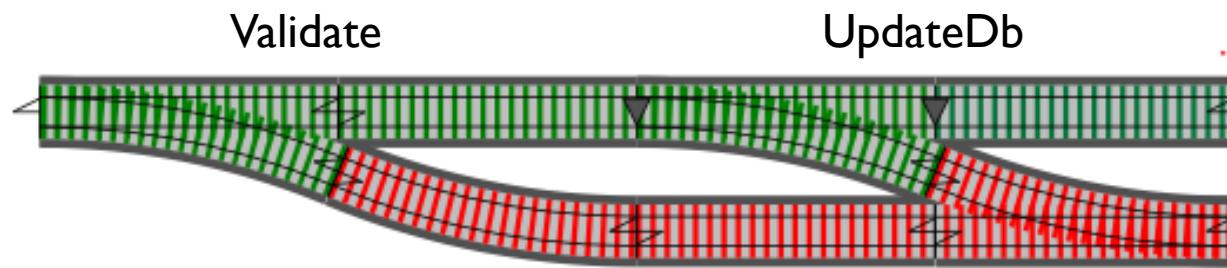
# Introducing switches



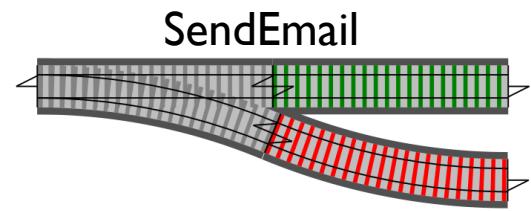
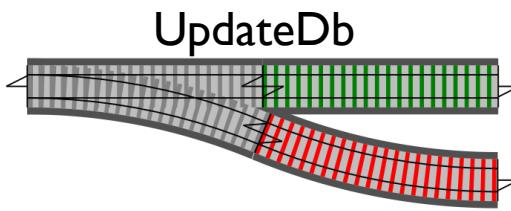
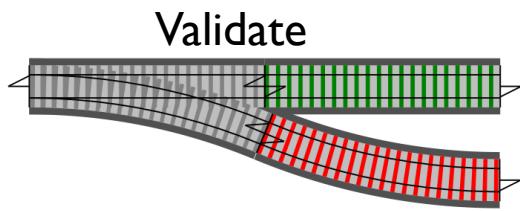
# Connecting switches



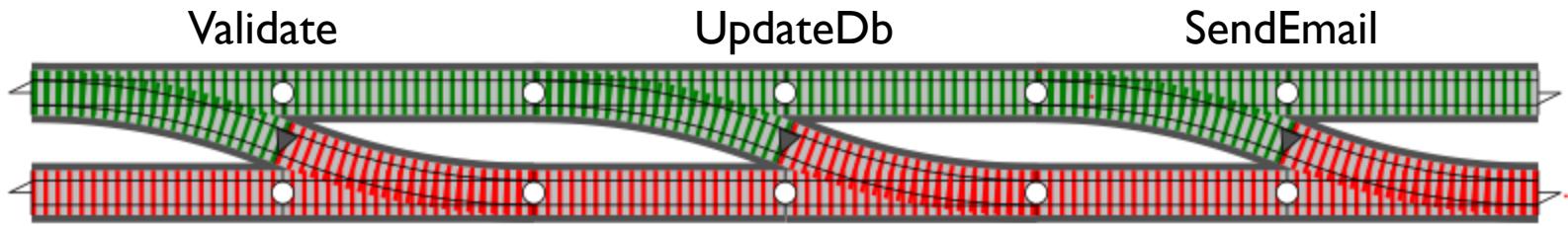
# Connecting switches



# Connecting switches



# Connecting switches



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

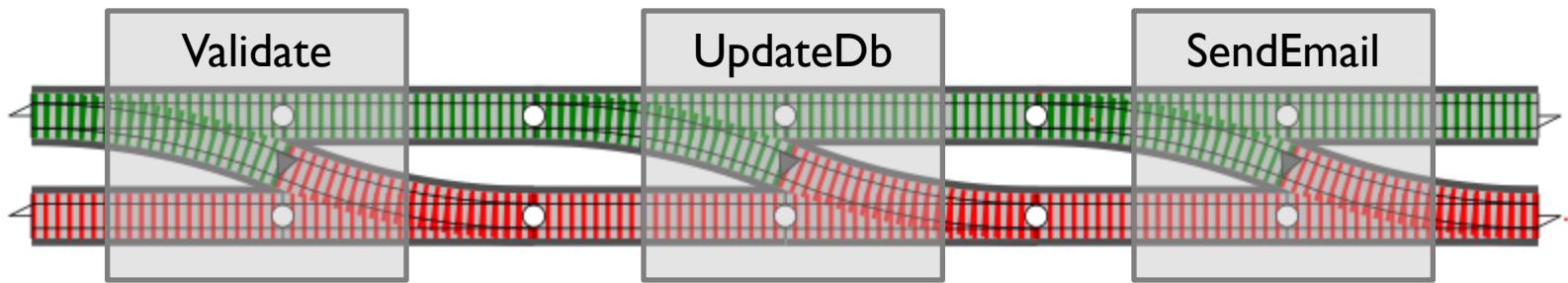
# The two-track model in practice

# Composing switches



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

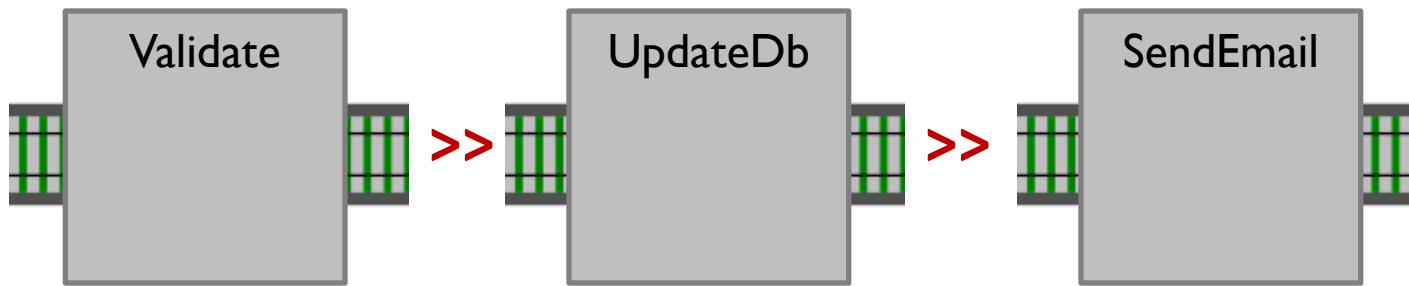
# Composing switches



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

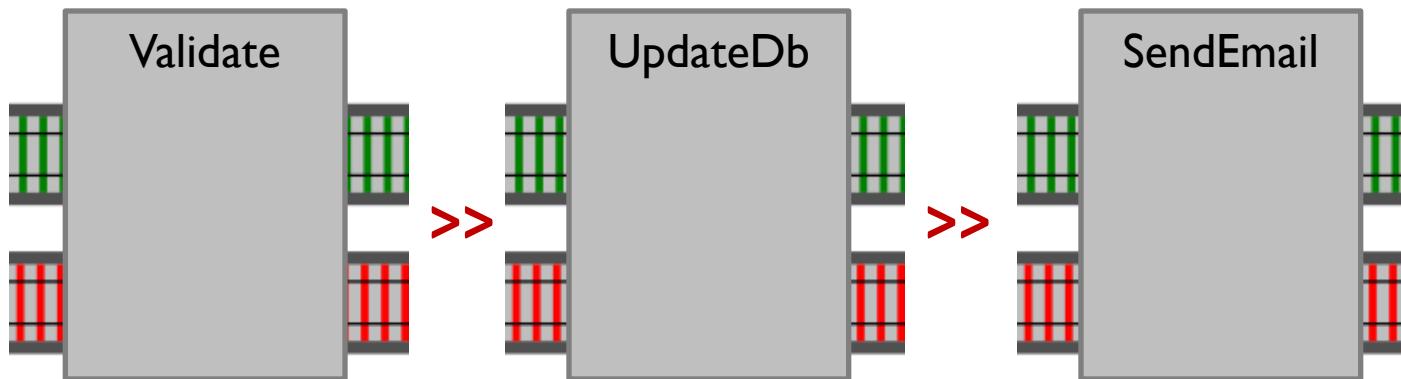
Inside each box there is a switch function.

# Composing switches



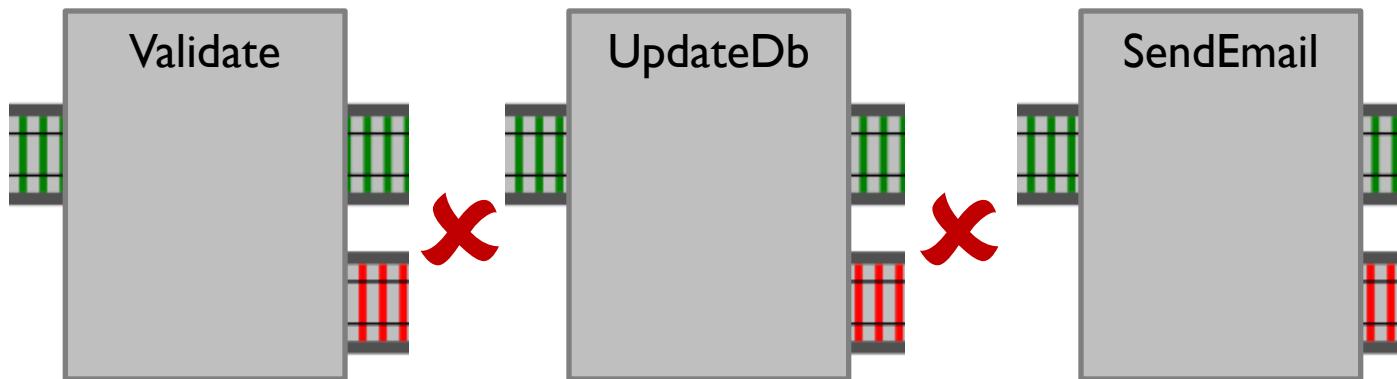
Composing one-track functions is fine...

# Composing switches



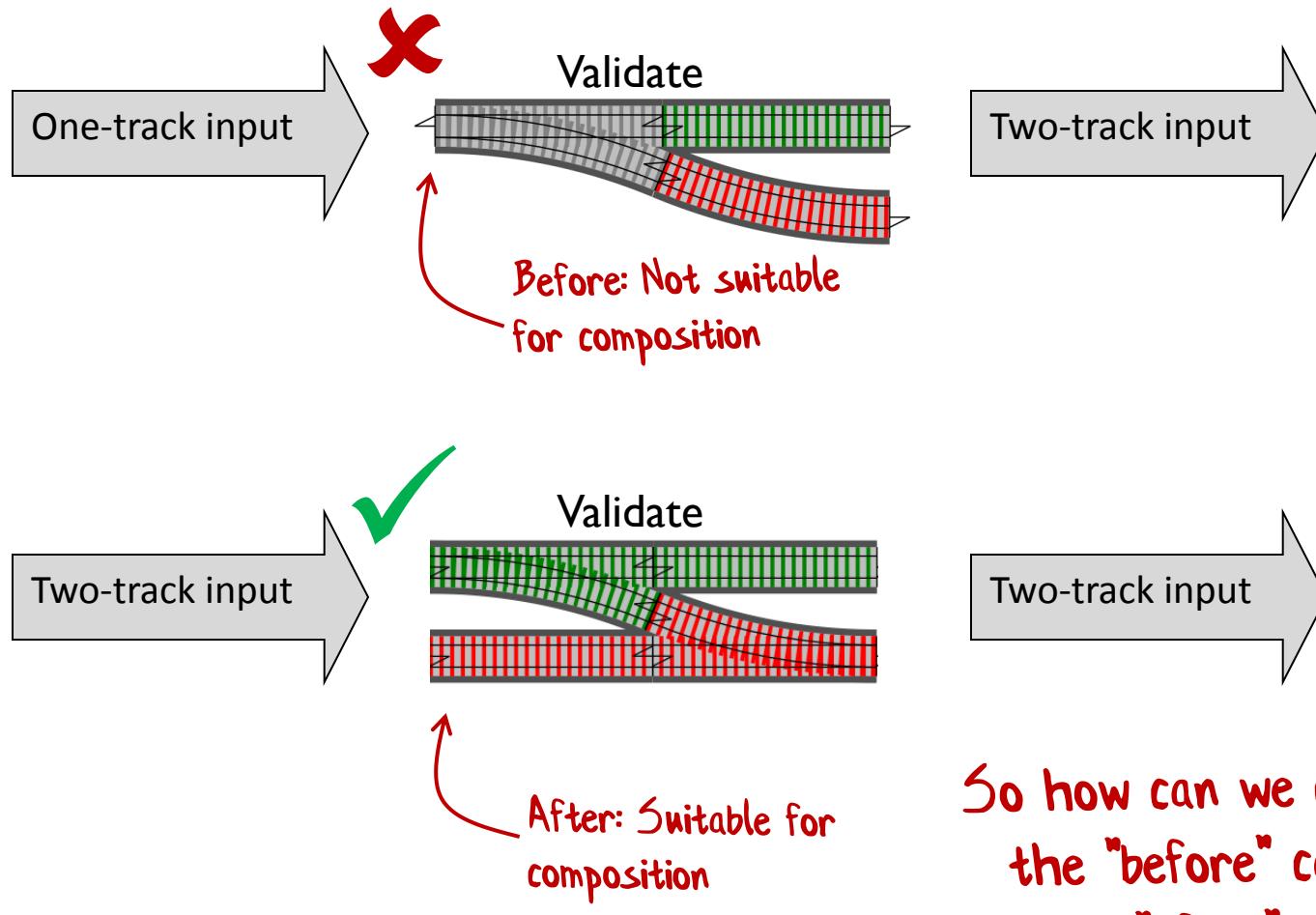
... and composing two-track functions is fine...

# Composing switches



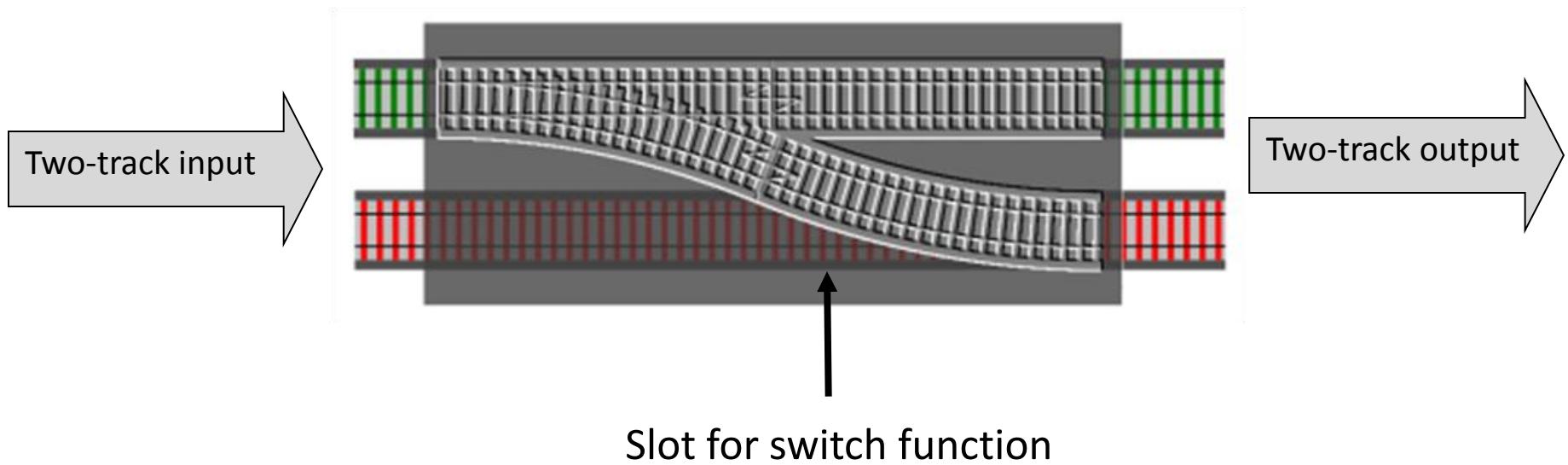
... but composing switches is not allowed!

# Composing switches

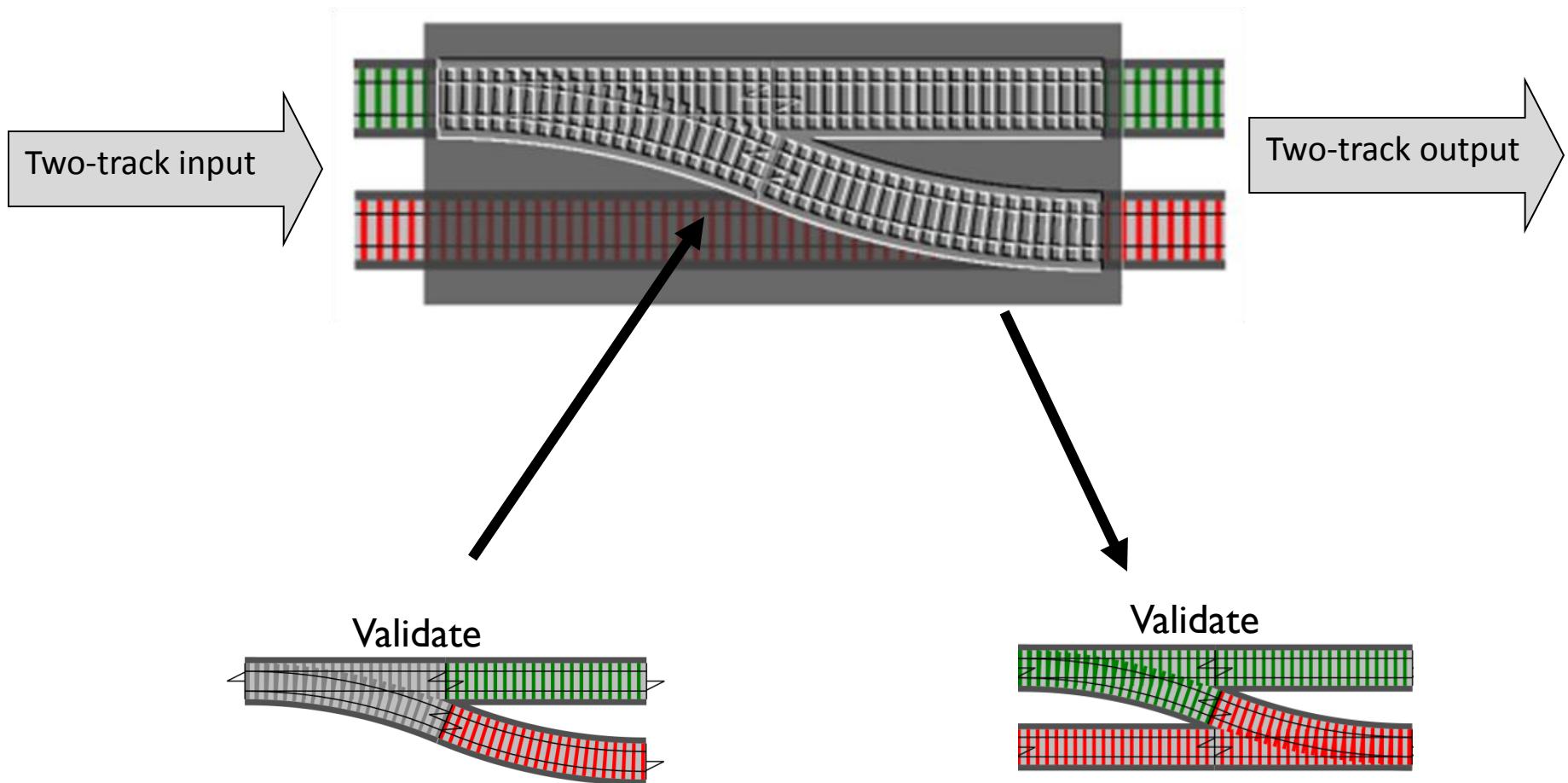


So how can we convert from  
the "before" case to the  
"after" case?

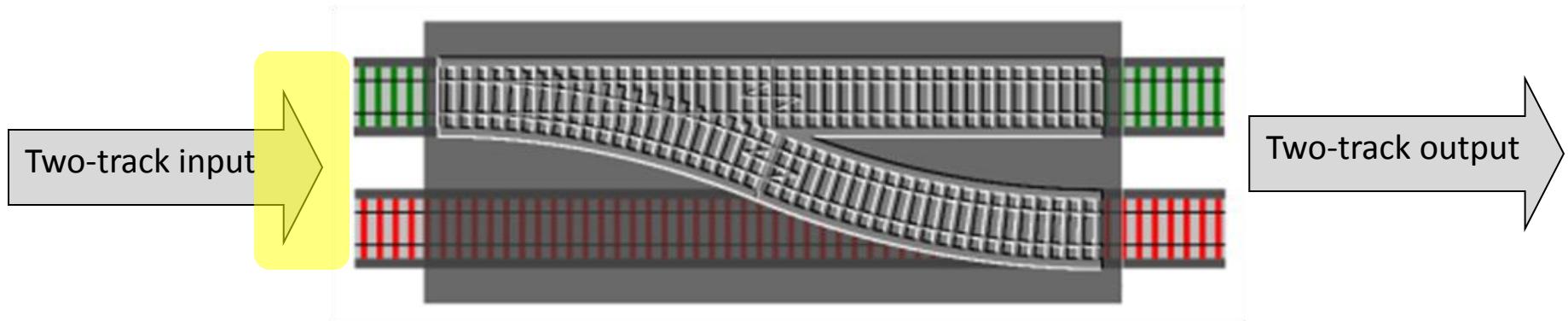
# Building an adapter block



# Building an adapter block

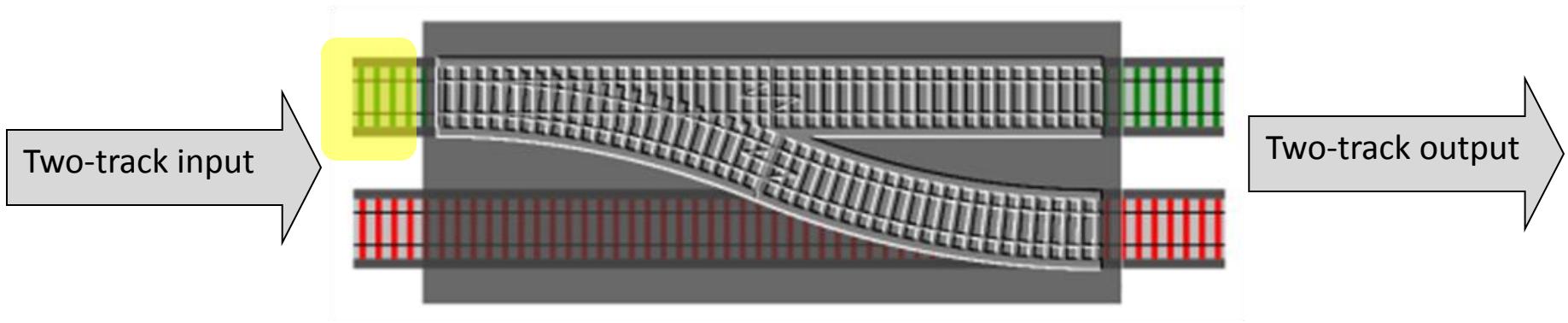


# Building an adapter block



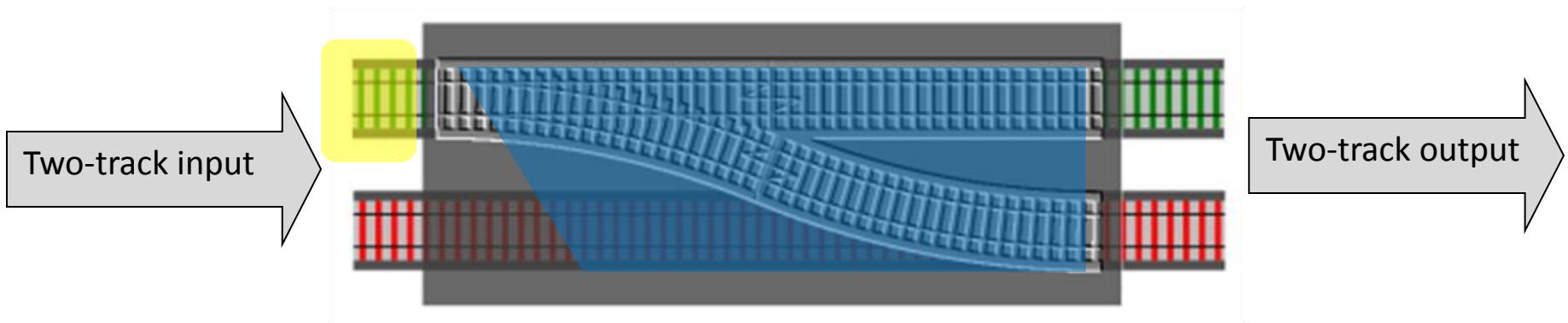
```
let adapt switchFunction =
  fun twoTrackInput ->
    match twoTrackInput with
    | Success s -> switchFunction s
    | Failure f -> Failure f
```

# Building an adapter block



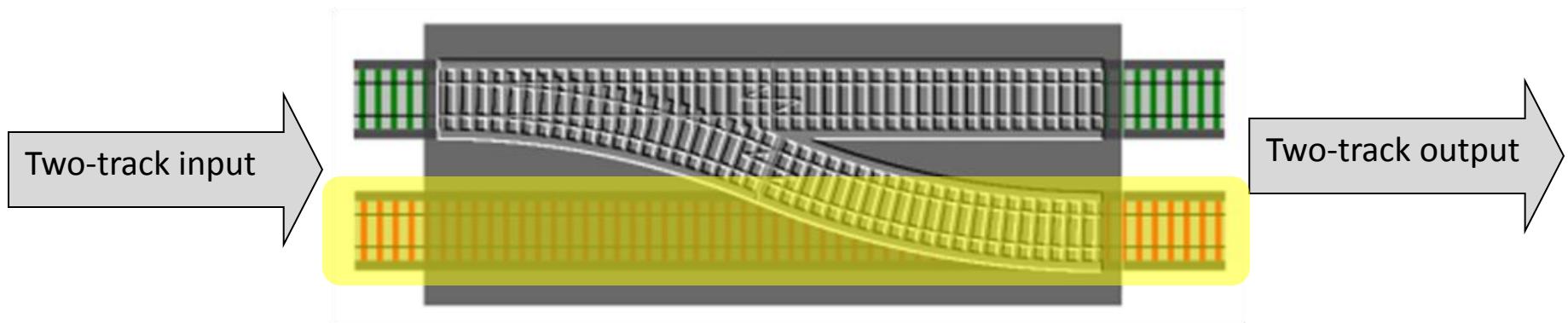
```
let adapt switchFunction =  
  fun twoTrackInput ->  
    match twoTrackInput with  
    | Success s -> switchFunction s  
    | Failure f -> Failure f
```

# Building an adapter block



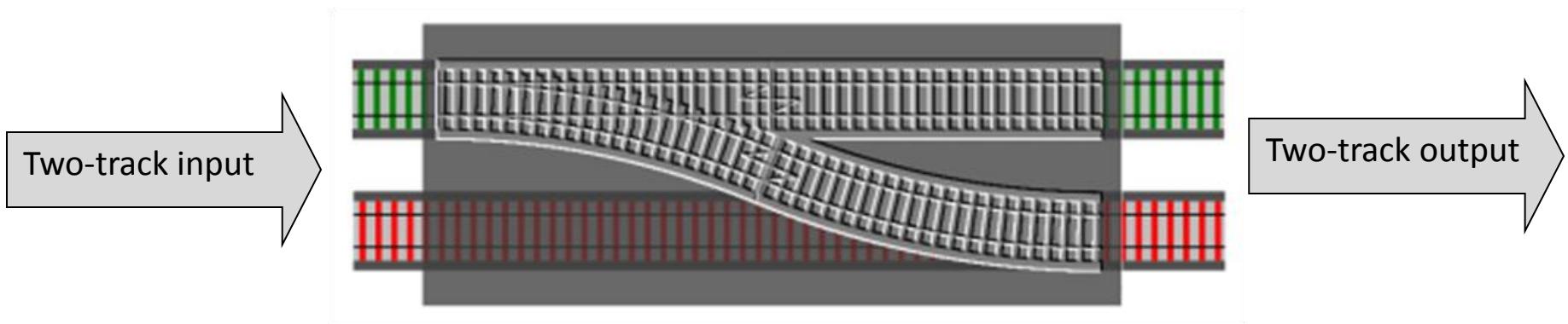
```
let adapt switchFunction =  
  fun twoTrackInput ->  
    match twoTrackInput with  
    | Success s -> switchFunction s  
    | Failure f -> Failure f
```

# Building an adapter block



```
let adapt switchFunction =  
    fun twoTrackInput ->  
        match twoTrackInput with  
        | Success s -> switchFunction s  
        | Failure f -> Failure f
```

# Bind as an adapter block



```
let bind switchFunction =
  fun twoTrackInput ->
    match twoTrackInput with
    | Success s -> switchFunction s
    | Failure f -> Failure f
```

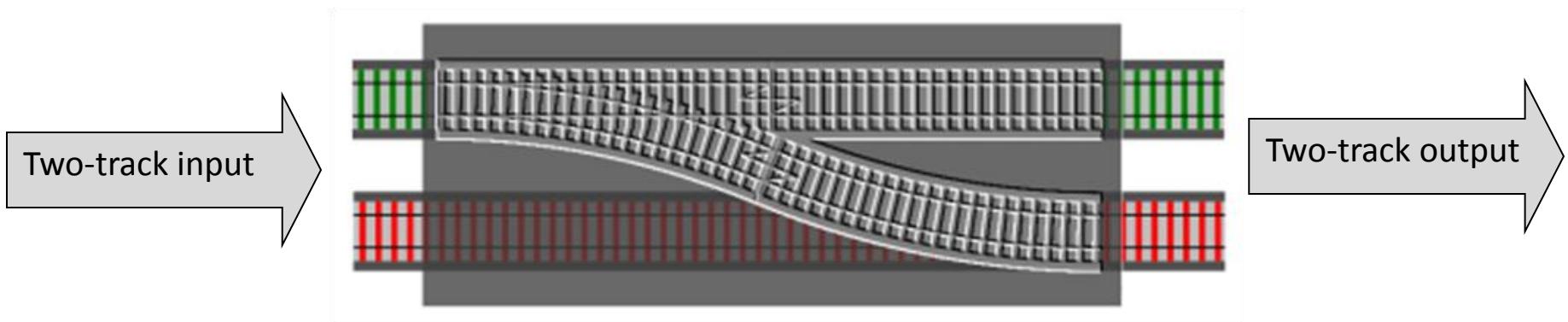
*bind : ('a -> TwoTrack<'b>) -> TwoTrack<'a> -> TwoTrack<'b>*

Switch  
function

2-track  
input

2-track  
output

# Bind as an adapter block



```
let bind switchFunction twoTrackInput =  
  match twoTrackInput with  
  | Success s -> switchFunction s  
  | Failure f -> Failure f
```

Same function:  
alternative version with  
two parameters.

```
bind : ('a -> TwoTrack<'b>) -> TwoTrack<'a> -> TwoTrack<'b>
```

Switch function

2-track input

2-track output

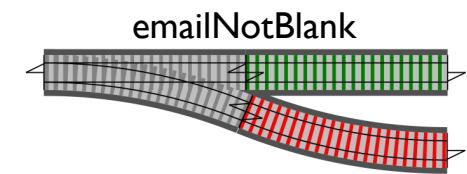
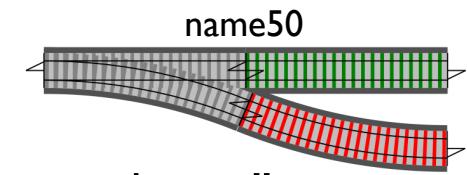
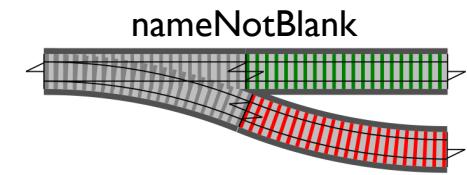
Diagram annotations: A red bracket underlines the type signature "('a -> TwoTrack<'b>) -> TwoTrack<'a> -> TwoTrack<'b>". Three red arrows point from the text labels "Switch function", "2-track input", and "2-track output" to the corresponding parts of the type signature: the first arrow points to the first parameter position, the second to the second parameter position, and the third to the return type position.

# Bind example

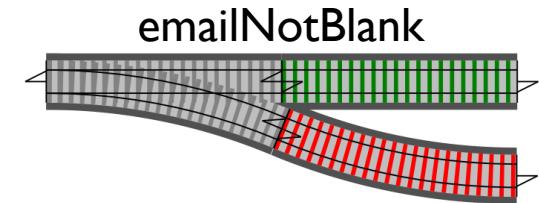
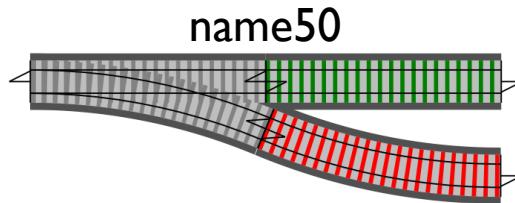
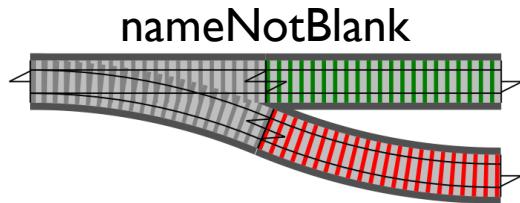
```
let nameNotBlank input =  
  if input.name = "" then  
    Failure "Name must not be blank"  
  else Success input
```

```
let name50 input =  
  if input.name.Length > 50 then  
    Failure "Name must not be longer than 50 chars"  
  else Success input
```

```
let emailNotBlank input =  
  if input.email = "" then  
    Failure "Email must not be blank"  
  else Success input
```

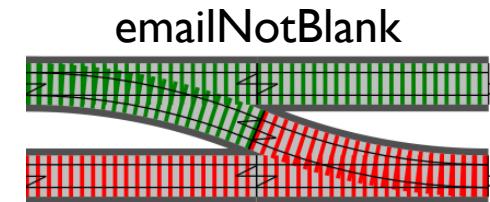
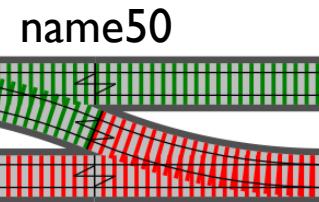
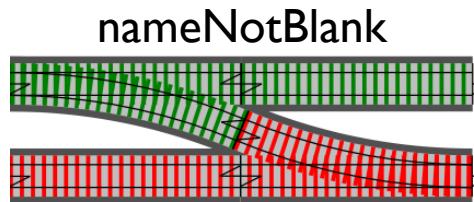


# Bind example



`nameNotBlank` (combined with)  
`name50` (combined with)  
`emailNotBlank`

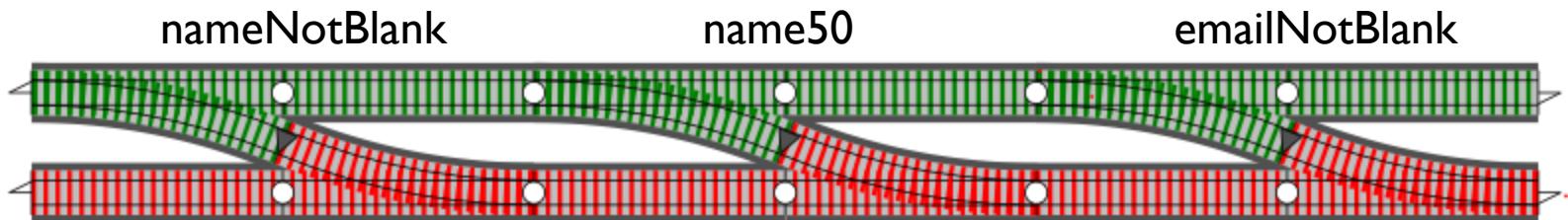
# Bind example



```
bind nameNotBlank  
bind name50  
bind emailNotBlank
```

use "bind" to convert to 2-track

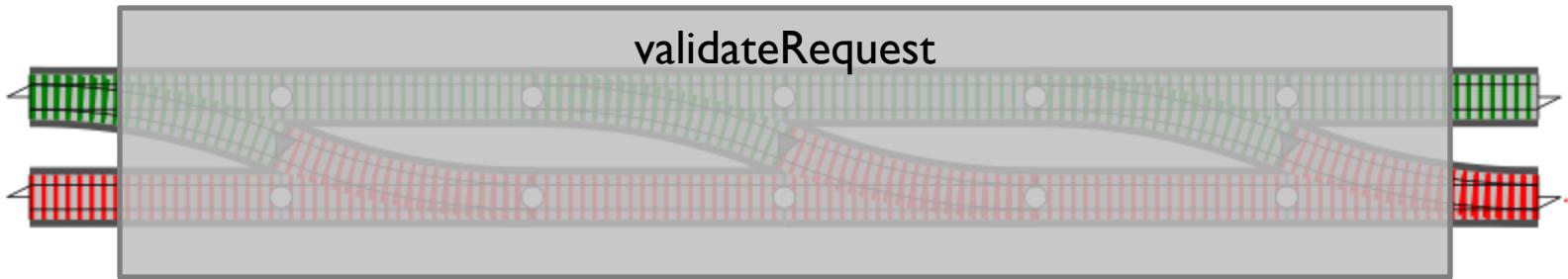
# Bind example



```
bind nameNotBlank  
>> bind name50  
>> bind emailNotBlank
```

*then compose together*

# Bind example

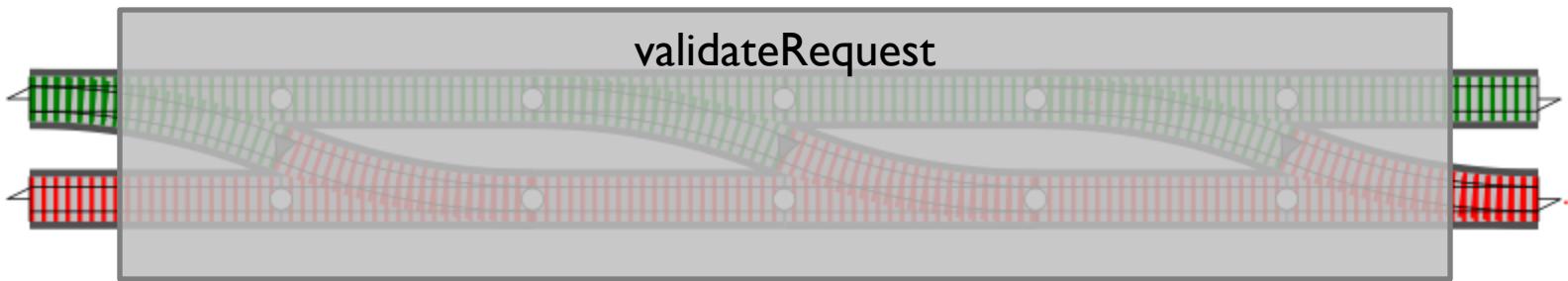


```
let validateRequest =  
  bind nameNotBlank  
  >> bind name50  
  >> bind emailNotBlank
```

// validateRequest : TwoTrack<Request> -> TwoTrack<Request>

Overall result is a new  
two-track function

# Bind example



```
let (>>=) twoTrackInput switchFunction =
    bind switchFunction twoTrackInput
```

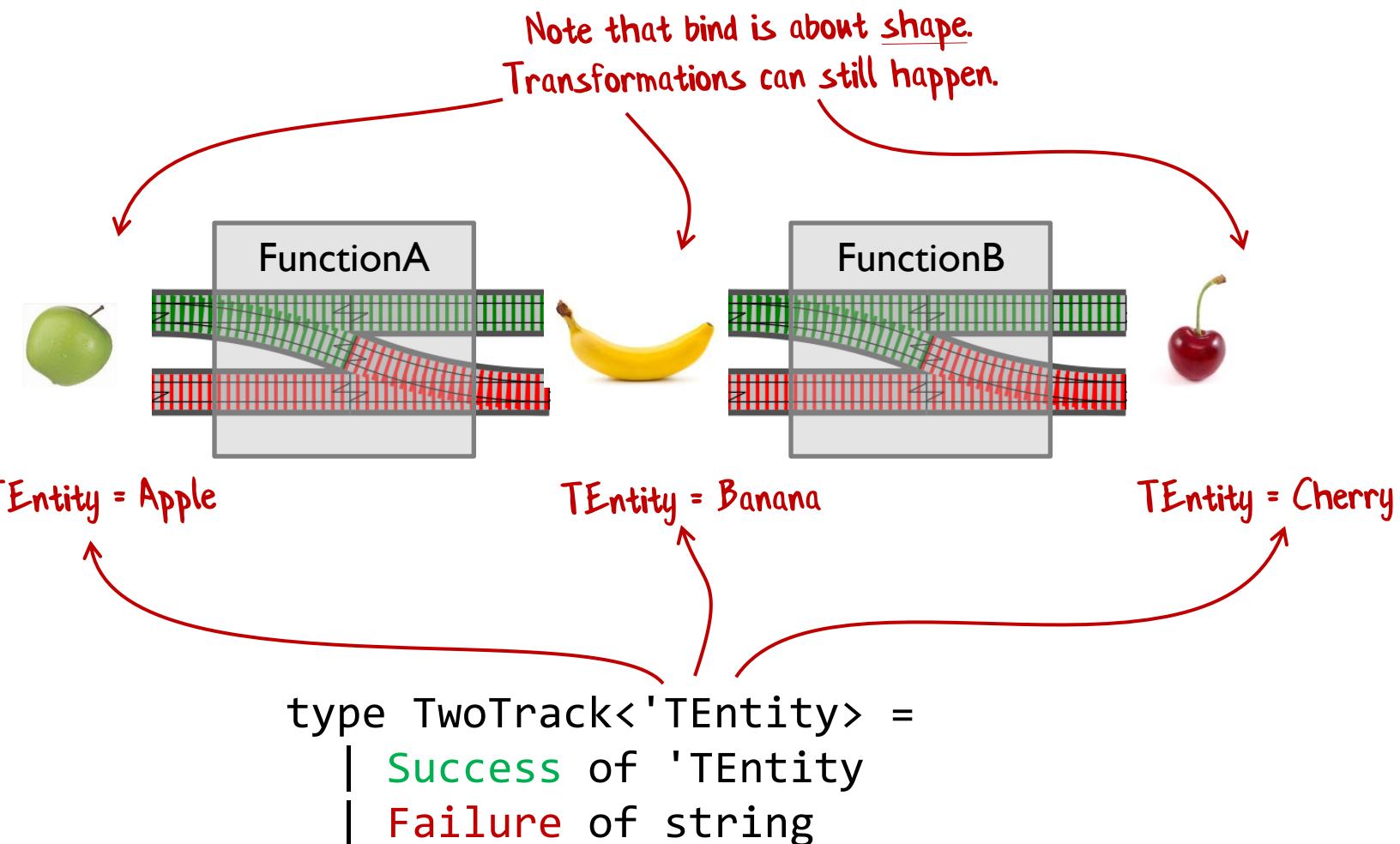
Common symbol for bind

```
let validateRequest twoTrackInput =
    twoTrackInput
    >>= nameNotBlank
    >>= name50
    >>= emailNotBlank
```

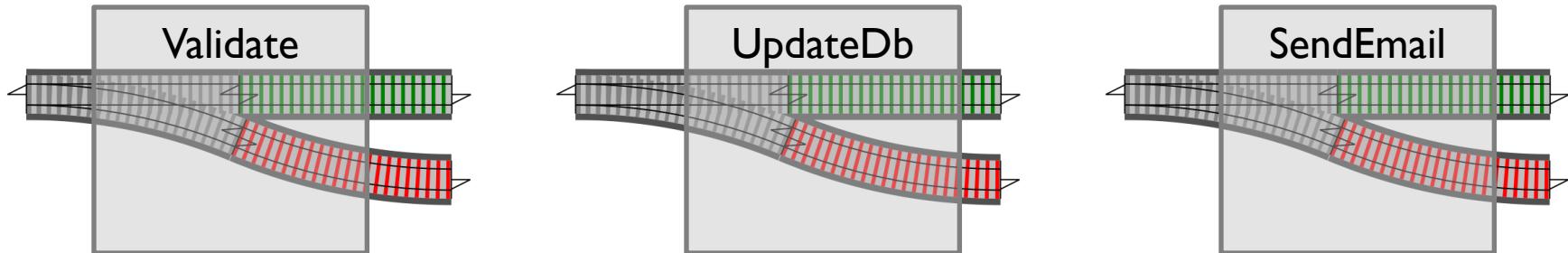
Needs a explicit parameter

Bind symbol = F# composition symbol +  
railway track symbol! Coincidence?

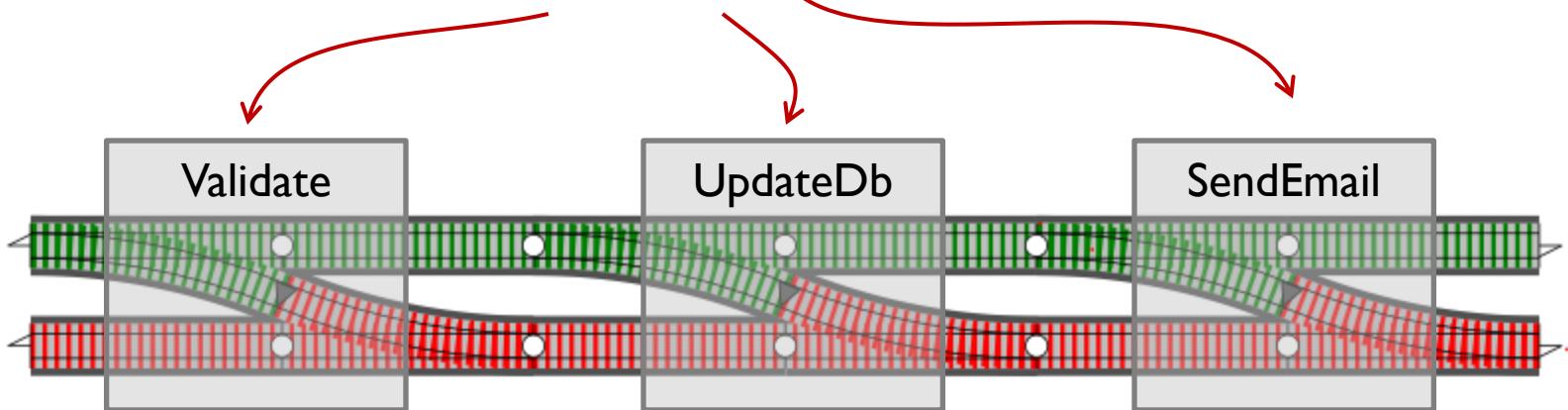
# Bind doesn't stop transformations



# Composing switches - review



Converted to two-track  
functions using bind



## Comic Interlude

What do you call a train that eats toffee?

I don't know, what do you call a train that eats toffee?

A chew, chew train!



# More fun with railway tracks...

Working with other functions

# More fun with railway tracks...

Fitting other functions into this framework:

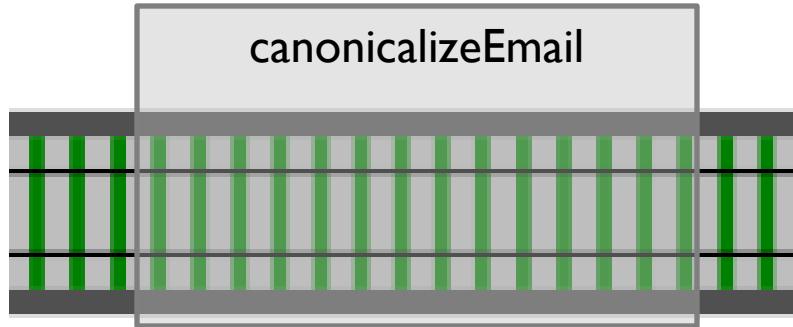
- Single track functions
- Dead-end functions
- Functions that throw exceptions
- Supervisory functions

# Converting one-track functions

Fitting other functions into this framework:

- **Single track functions**
- Dead-end functions
- Functions that throw exceptions
- Supervisory functions

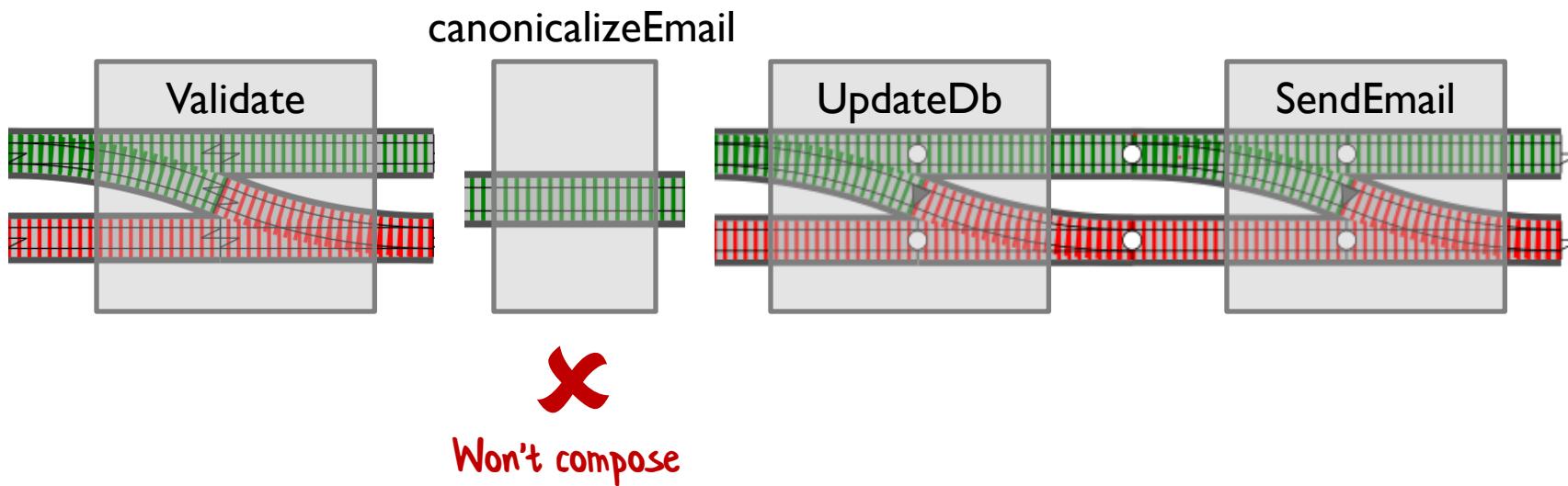
# Converting one-track functions



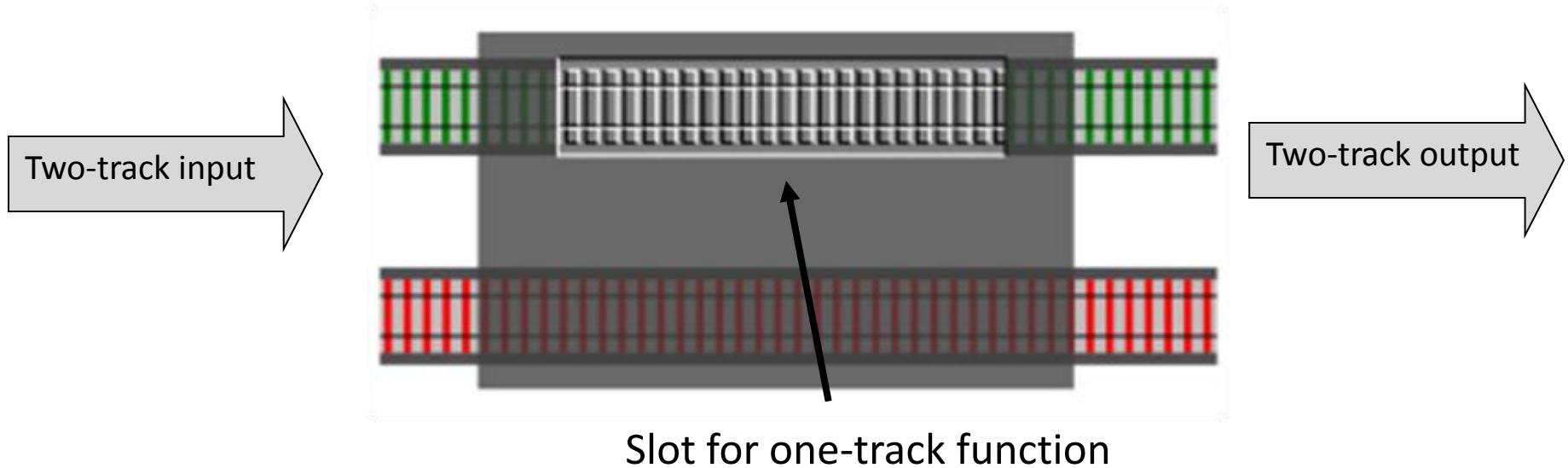
```
// trim spaces and lowercase
let canonicalizeEmail input =
    { input with email = input.email.Trim().ToLower() }
```

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

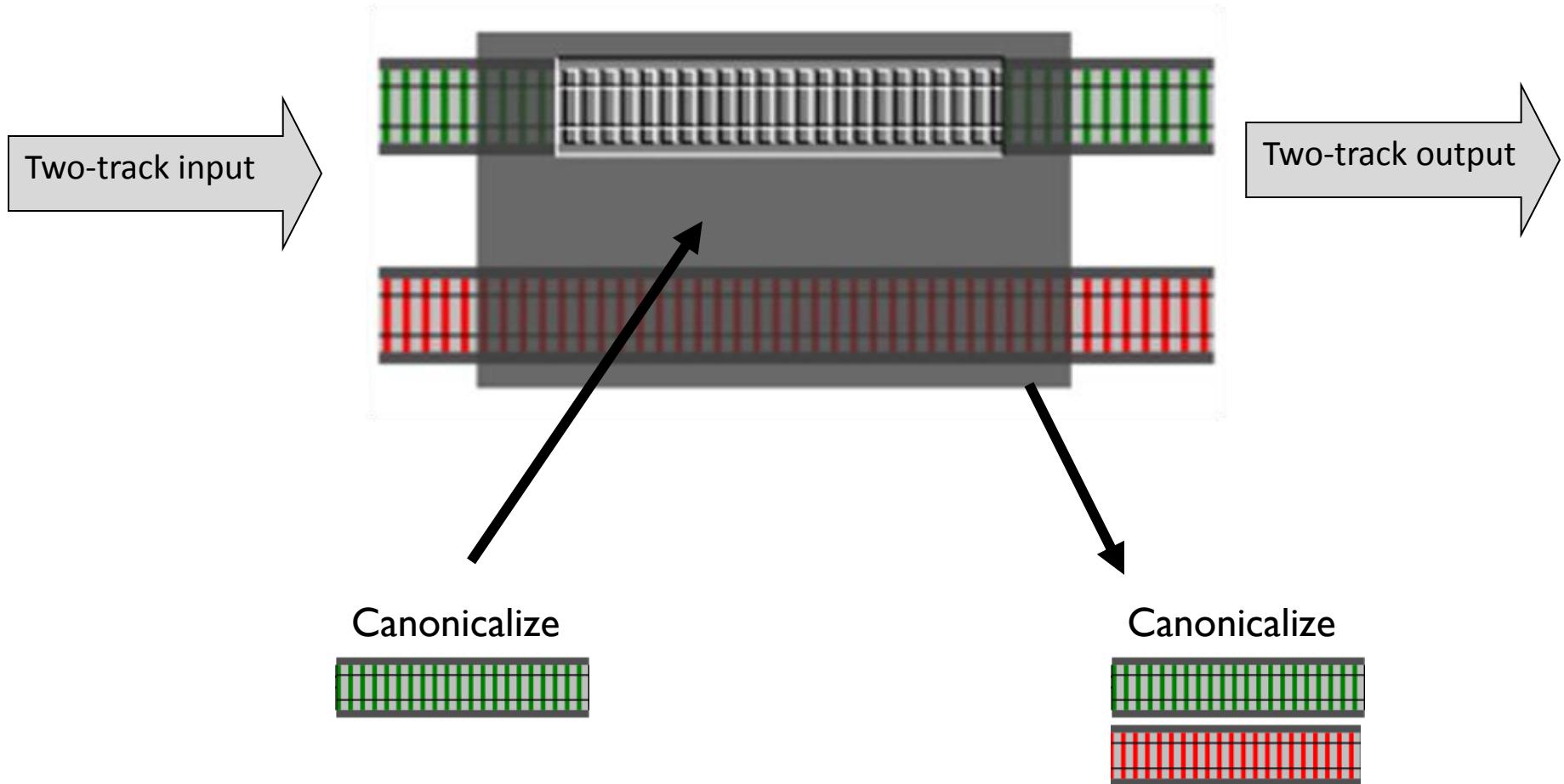
# Converting one-track functions



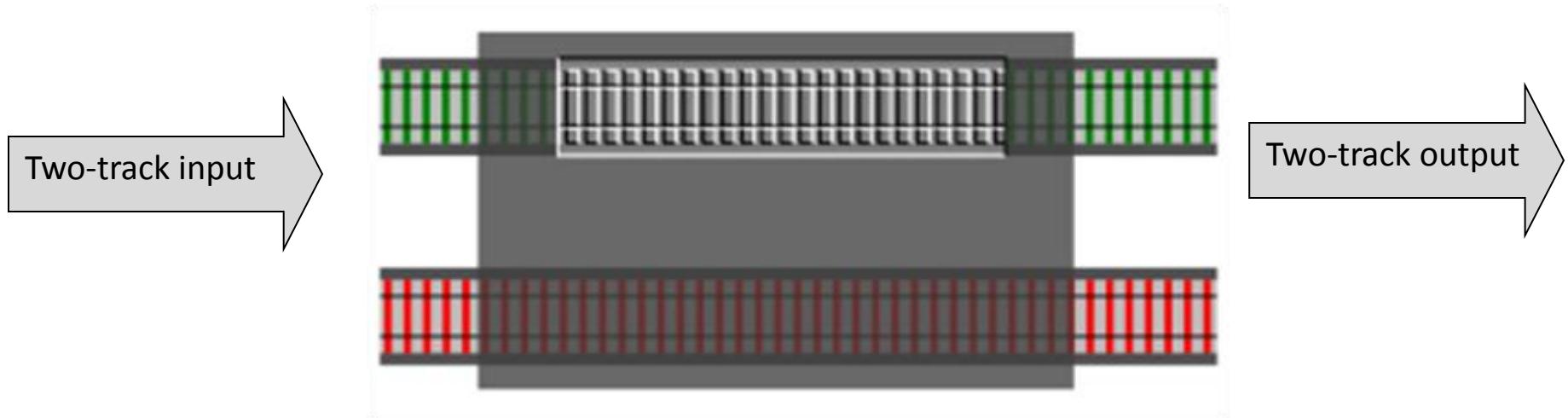
# Converting one-track functions



# Converting one-track functions



# Converting one-track functions



```
let map singleTrackFunction =  
  fun twoTrackInput ->  
    match twoTrackInput with  
    | Success s -> Success (singleTrackFunction s)  
    | Failure f -> Failure f
```

*map : ('a -> 'b) -> TwoTrack<'a> -> TwoTrack<'b>*

  └───┐

    Single track  
    function

  └───┐

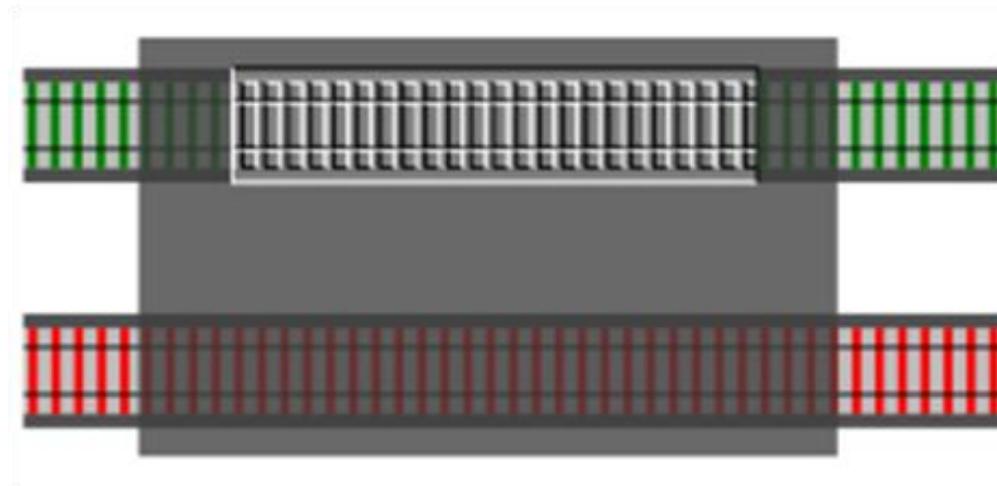
    2-track  
    input

  └───┐

    2-track  
    output

# Converting one-track functions

Two-track input



Two-track output

```
let map singleTrackFunction =  
  bind (singleTrackFunction >> Success)
```

Tip: "map" can also be built  
from "bind" and "Success"

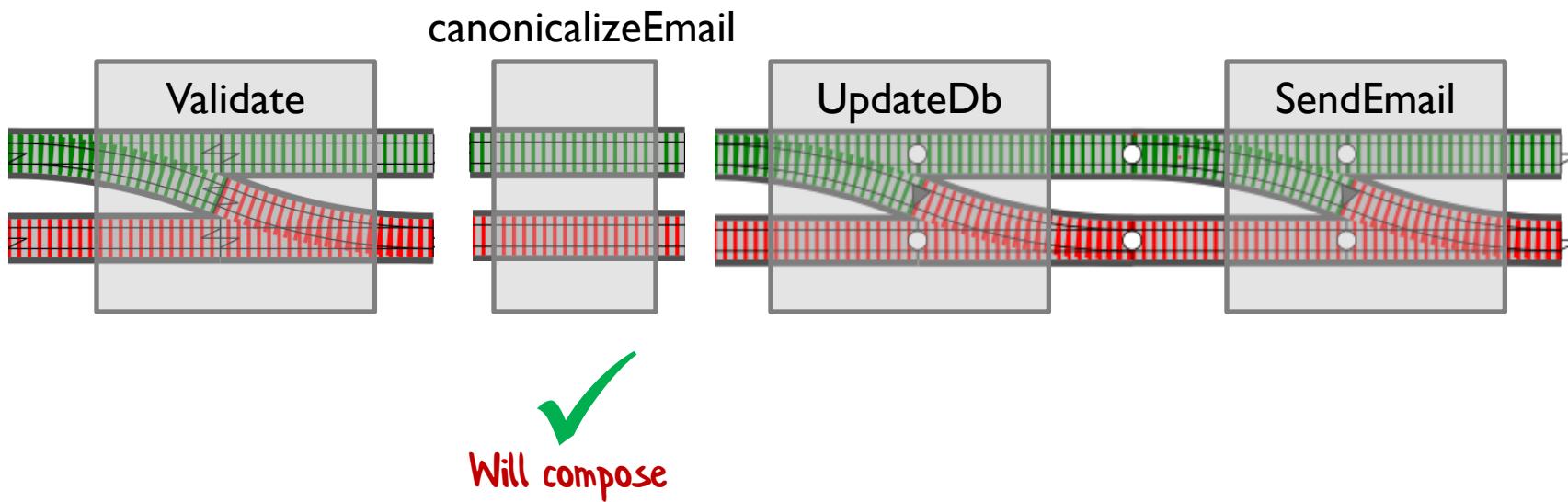
*map* : ('a -> 'b) -> TwoTrack<'a> -> TwoTrack<'b>

Single track  
function

2-track  
input

2-track  
output

# Converting one-track functions

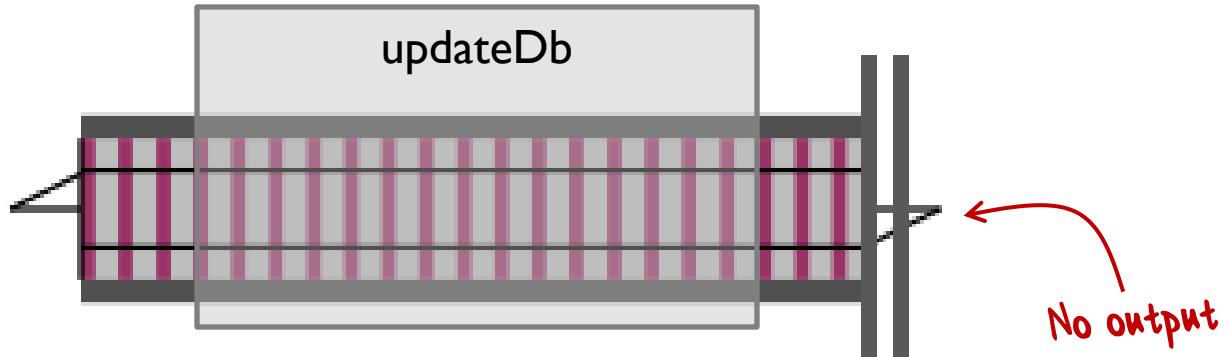


# Converting dead-end functions

Fitting other functions into this framework:

- Single track functions
- **Dead-end functions**
- Functions that throw exceptions
- Supervisory functions

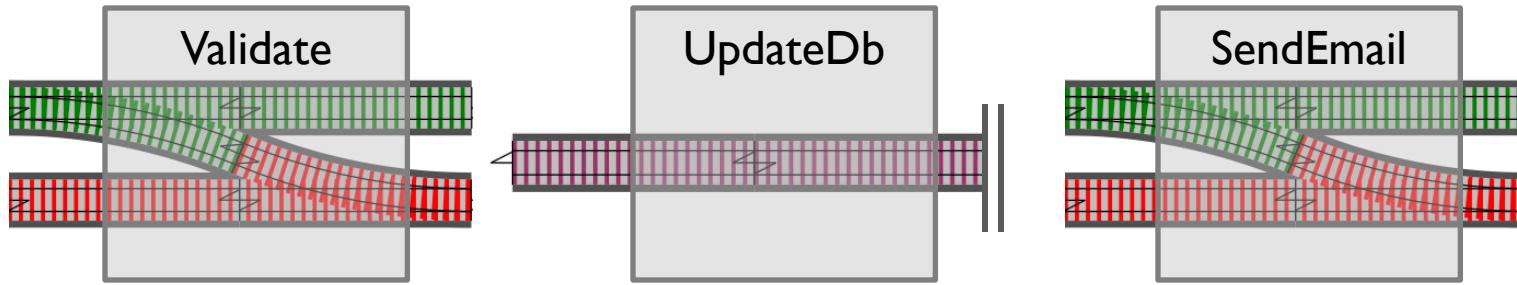
# Converting dead-end functions



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

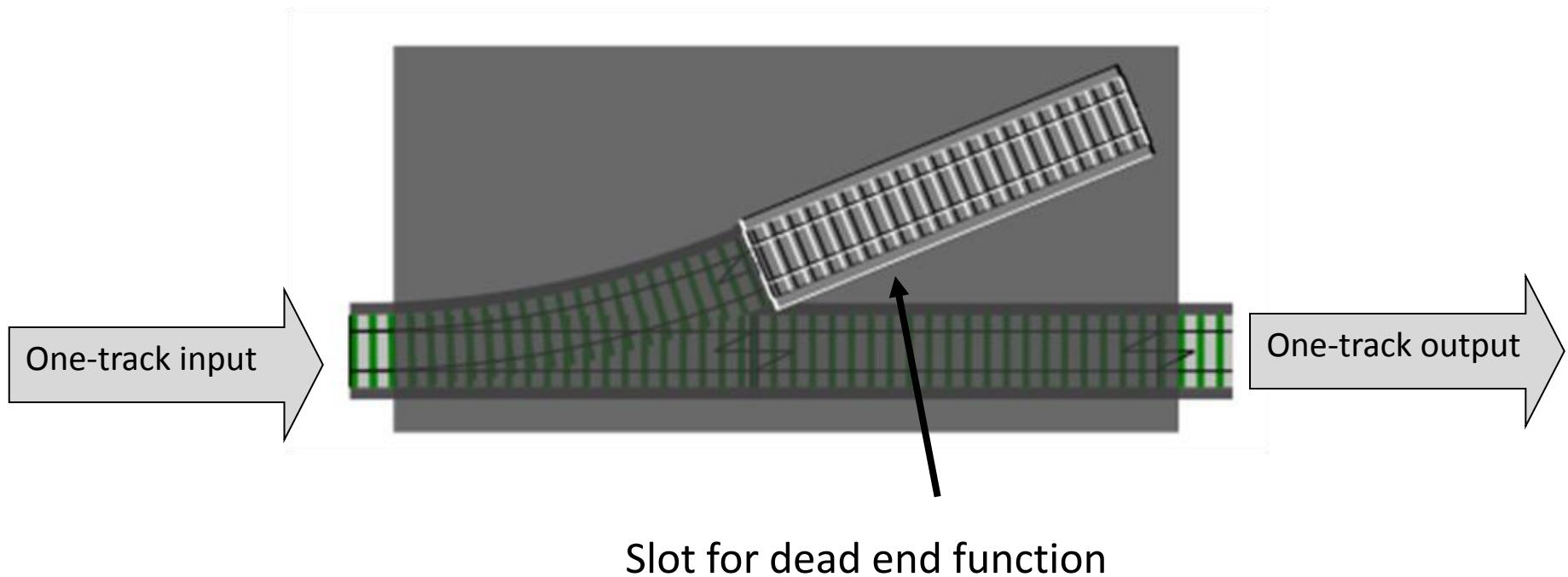
A function that doesn't return anything— a "dead-end" function.

# Converting dead-end functions

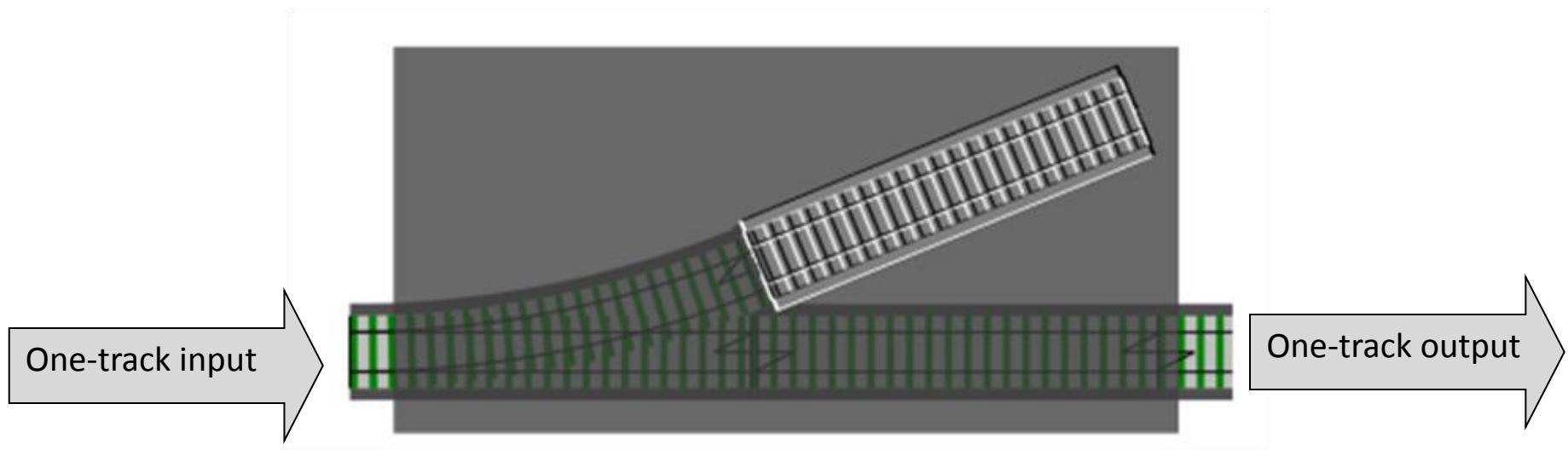


Won't compose

# Converting dead-end functions



# Converting dead-end functions

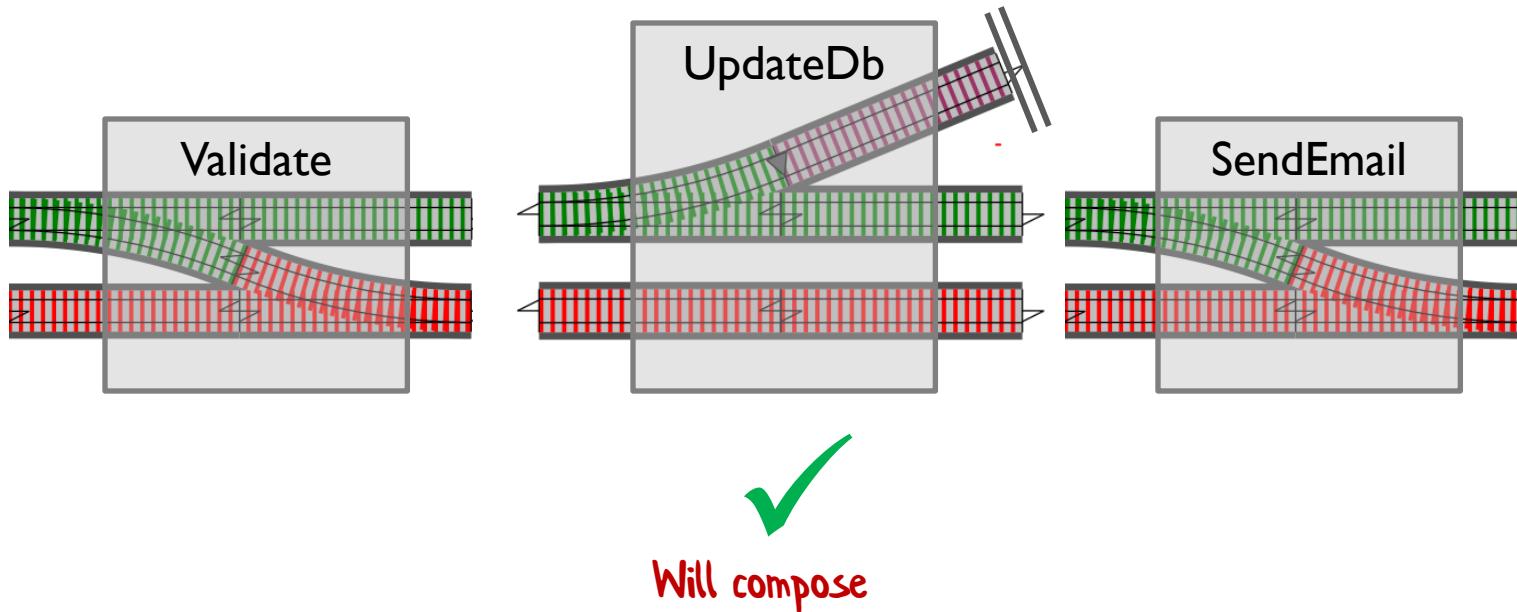


```
let tee deadEndFunction oneTrackInput =  
    deadEndFunction oneTrackInput  
    oneTrackInput
```

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

Dead end function      one-track input      one-track output

# Converting dead-end functions

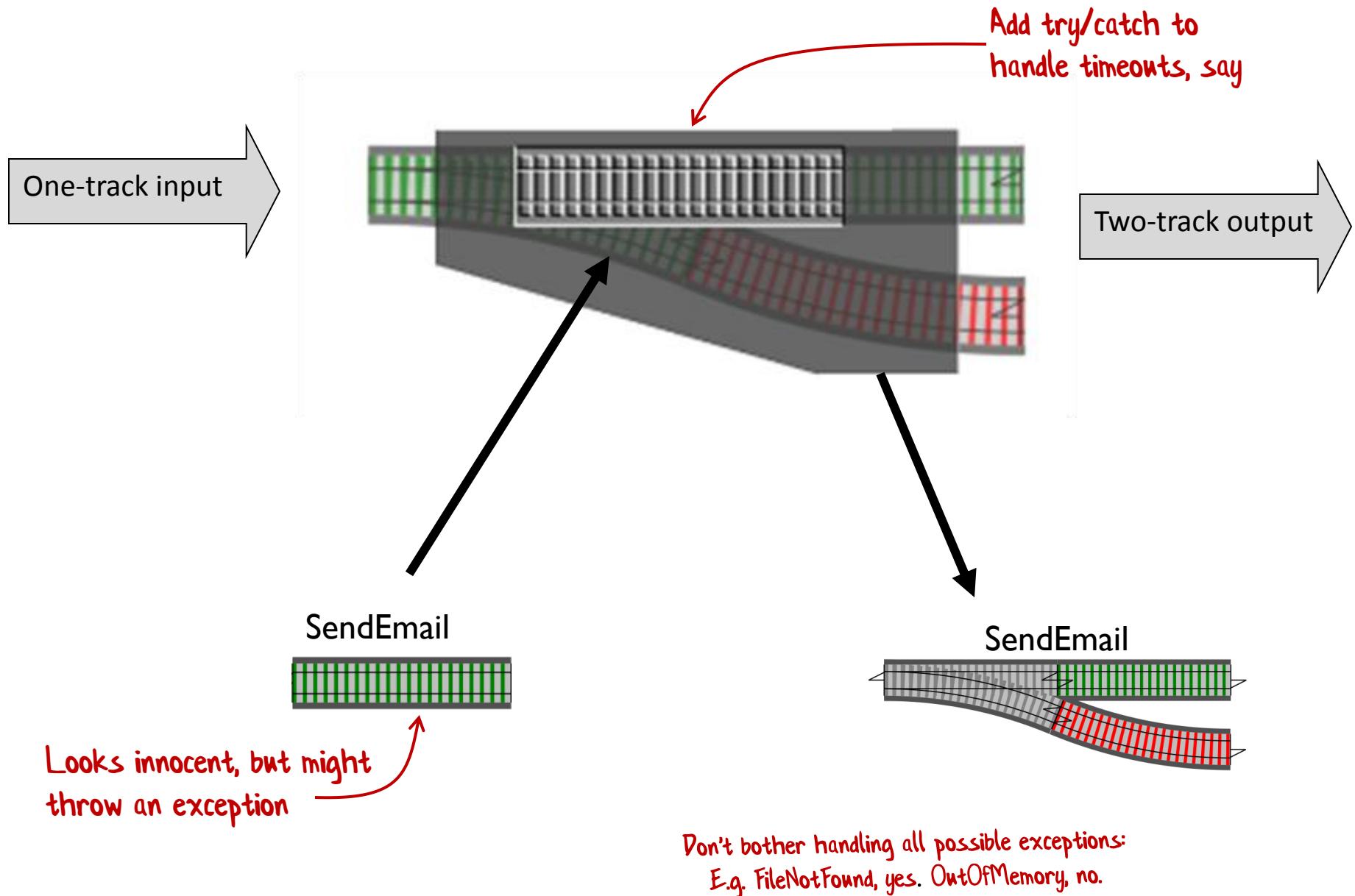


# Functions that throw exceptions

Fitting other functions into this framework:

- Single track functions
- Dead-end functions
- **Functions that throw exceptions**  
*Especially to wrap an I/O call*
- Supervisory functions

# 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".

# Supervisory functions

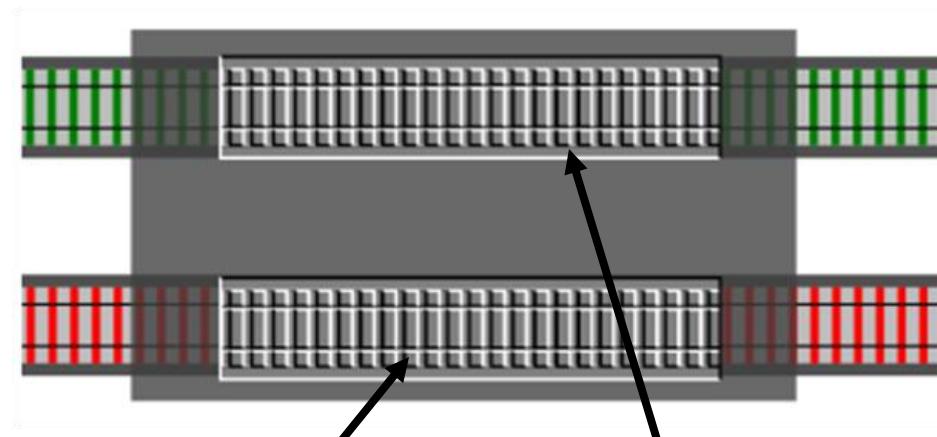
Fitting other functions into this framework:

- Single track functions
- Dead-end functions
- Functions that throw exceptions
- **Supervisory functions**

For when you need to handle \*both\* tracks  
– e.g. tracing, logging, etc.

# Supervisory functions

Two-track input



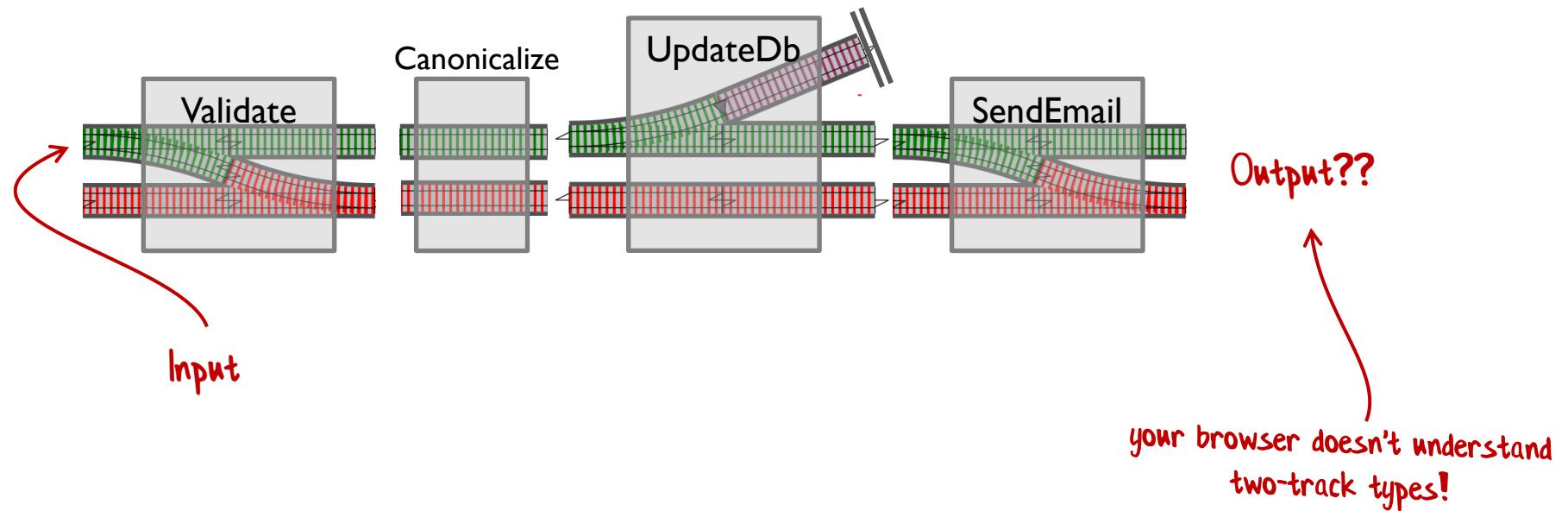
Two-track output

Slot for one-track function for Success case

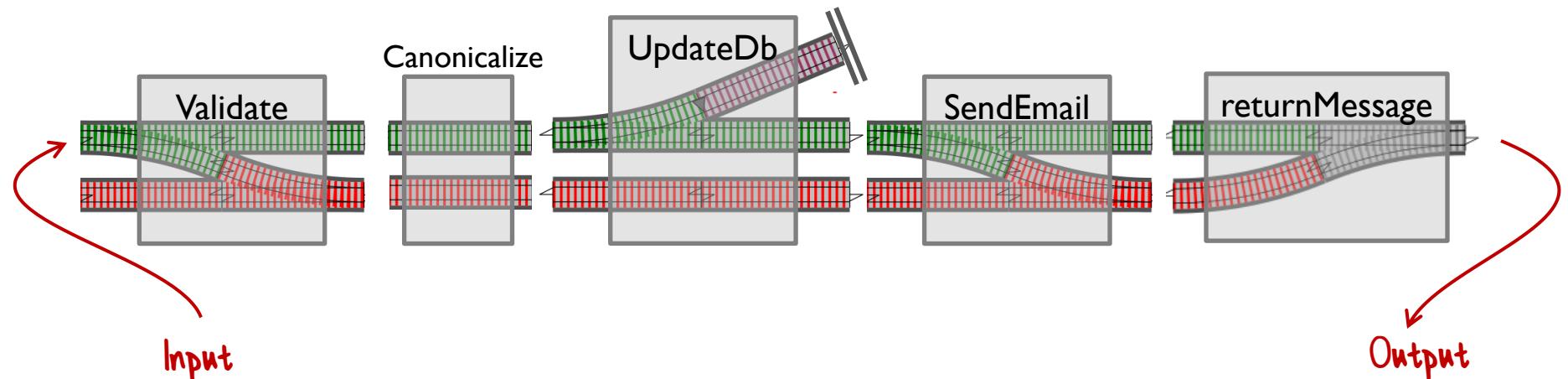
Slot for one-track function for Failure case

# Putting it all together

# Putting it all together



# Putting it all together



```
let returnMessage result =
  match result with
  | Success obj -> OK objToJson()
  | Failure msg -> BadRequest msg
```

# Putting it all together - review

The "two-track" framework is a useful approach for most use-cases.

You can fit most functions into this model.

Not a solution for everything, but a good starting point.

# Putting it all together - review

The "two-track" framework is a useful approach for most use-cases.

Let's look at the code -- before and after adding error handling

```
let updateCustomer =  
  receiveRequest  
  >> validateRequest  
  >> updateDbFromRequest  
  >> sendEmail  
  >> returnMessage
```

Before – without error handling

```
let updateCustomer =  
  receiveRequest  
  >> validateRequest  
  >> updateDbFromRequest  
  >> sendEmail  
  >> returnMessage
```

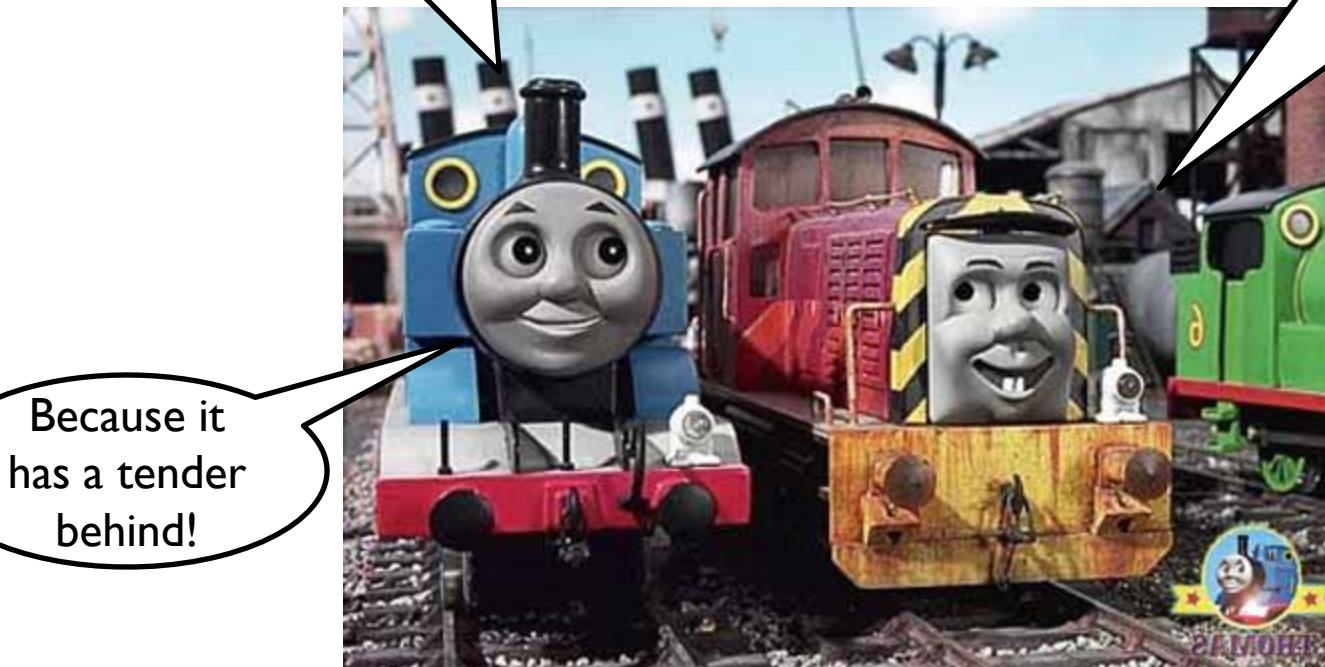
After – with error handling

Still clean and elegant

# Comic Interlude

Why can't a steam locomotive sit down?

I don't know,  
why can't a steam locomotive sit down?



## Extending the framework:

- Designing for errors
- Parallel tracks
- Domain events

# Designing for errors

Unhappy paths are requirements too

# Designing for errors

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

```
type TwoTrack<'TEntity> =  
| Success of 'TEntity  
| Failure of string
```

Using strings is not good

# Designing for errors

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

```
type TwoTrack<'TEntity> =  
| Success of 'TEntity  
| Failure of ErrorMessage
```

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

Special type rather  
than string

# Designing for errors

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

Add invalid  
email as data

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

# Designing for errors

```
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!

# Designing for errors

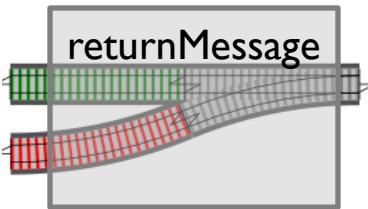
```
type ErrorMessage =  
| NameMustNotBeBlank  
| EmailMustNotBeBlank  
| EmailNotValid of EmailAddress  
// database errors  
| UserIdNotValid of UserId  
| DbUserNotFoundError of UserId  
| DbTimeout of ConnectionString  
| DbConcurrencyError  
| DbAuthorizationError of ConnectionString * Credentials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

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

As we develop the code, we can build up a complete list of everything that could go wrong

one single place to see all the error cases  
Also triggers important DDD conversations

# Designing for errors – converting to strings

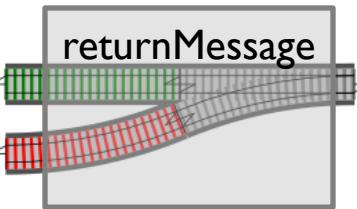


```
let returnMessage result =
  match result with
  | Success _ -> "Success"
  | Failure msg -> msg
```

No longer works – each case must now  
be explicitly converted to a string

# Designing for errors – converting to strings

```
let returnMessage result =
  match result with
  | Success _ -> "Success"
  | Failure err ->
    match err with
    | NameMustNotBeBlank -> "Name must not be blank"
    | EmailMustNotBeBlank -> "Email must not be blank"
    | EmailNotValid (EmailAddress email) ->
        sprintf "Email %s is not valid" email
    // database errors
    | UserIdNotValid (UserId id) ->
        sprintf "User id %i is not a valid user id" id
    | DbUserNotFoundError (UserId id) ->
        sprintf "User id %i was not found in the database" id
    | DbTimeout (_,TimeoutMs ms) ->
        sprintf "Could not connect to database within %i ms" ms
    | DbConcurrencyError ->
        sprintf "Another user has modified the record. Please resubmit"
    | DbAuthorizationError _ ->
        sprintf "You do not have permission to access the database"
    // SMTP errors
    | SmtpTimeout (_,TimeoutMs ms) ->
        sprintf "Could not connect to SMTP server within %i ms" ms
    | SmtpBadRecipient (EmailAddress email) ->
        sprintf "The email %s is not a valid recipient" email
```



Each case must be converted to a string – but this is only needed once, and only at the last step.

Different conversions can be used depending on the target.  
E.g. user messages vs. logging.

All strings are in one place,  
so translations are easier.  
(or use resource file)

# 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 SmtpConnection  
| SmtpBadRecipient of EmailAddress
```

Documentation of everything that can go wrong.

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

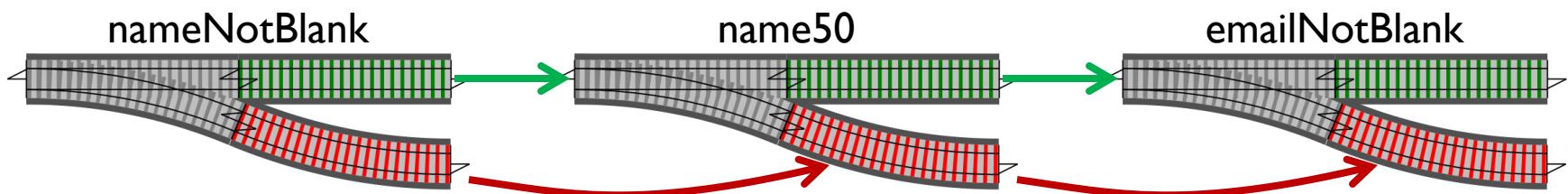
Surfaces hidden requirements.

Test against error codes, not strings.

Makes translation easier.

# Parallel tracks

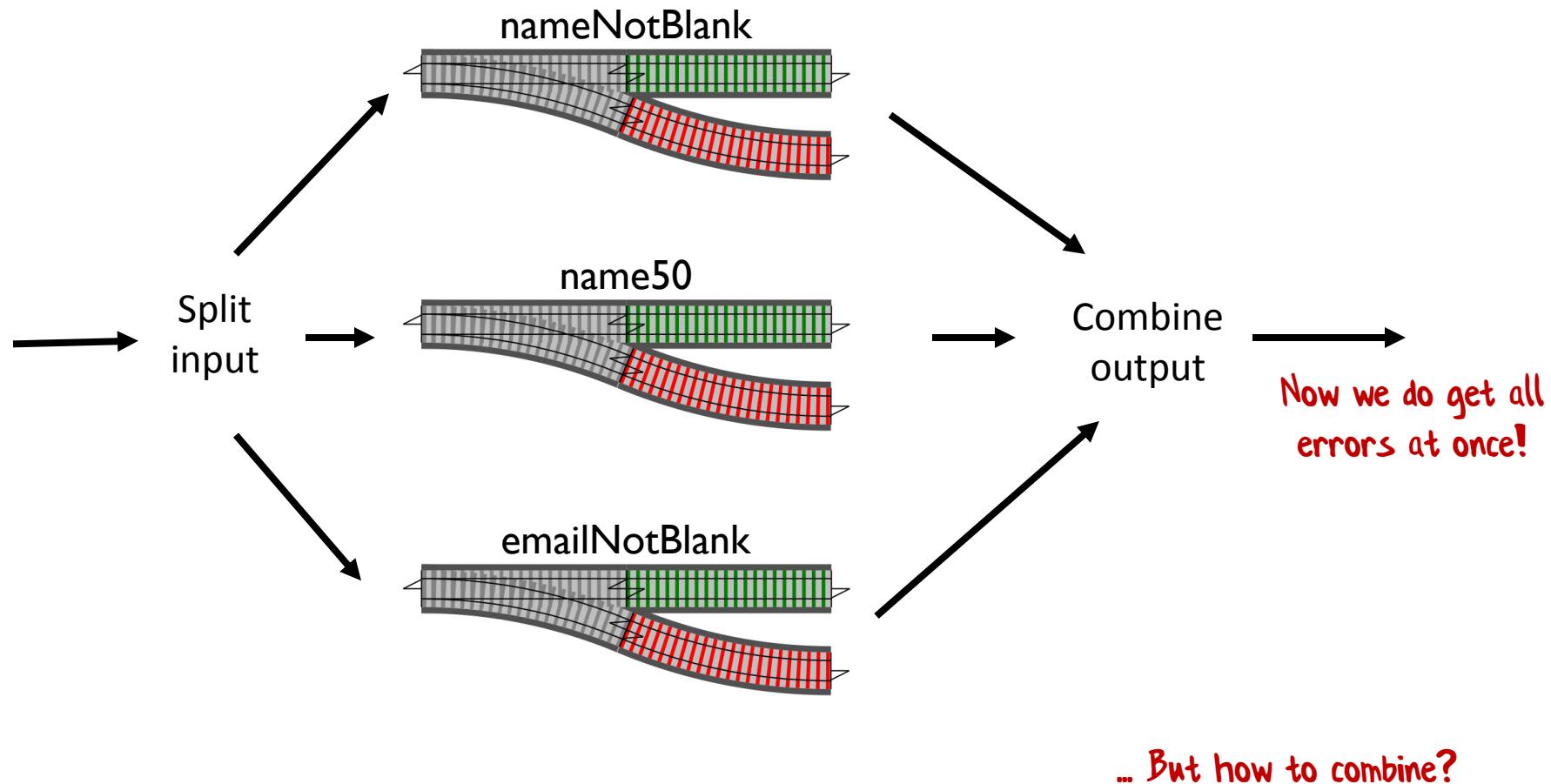
# Parallel validation



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

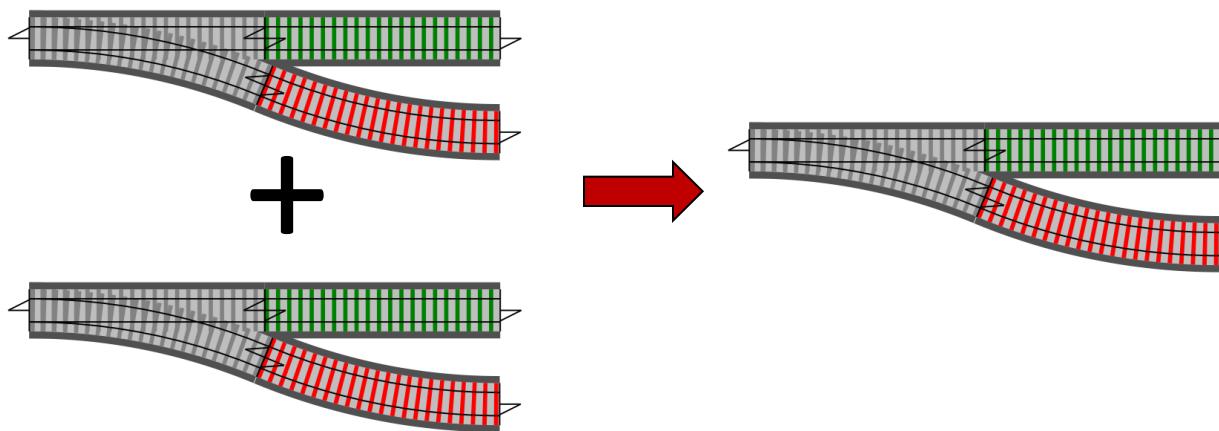
It would be nice to return all validation errors at once.

# Parallel validation



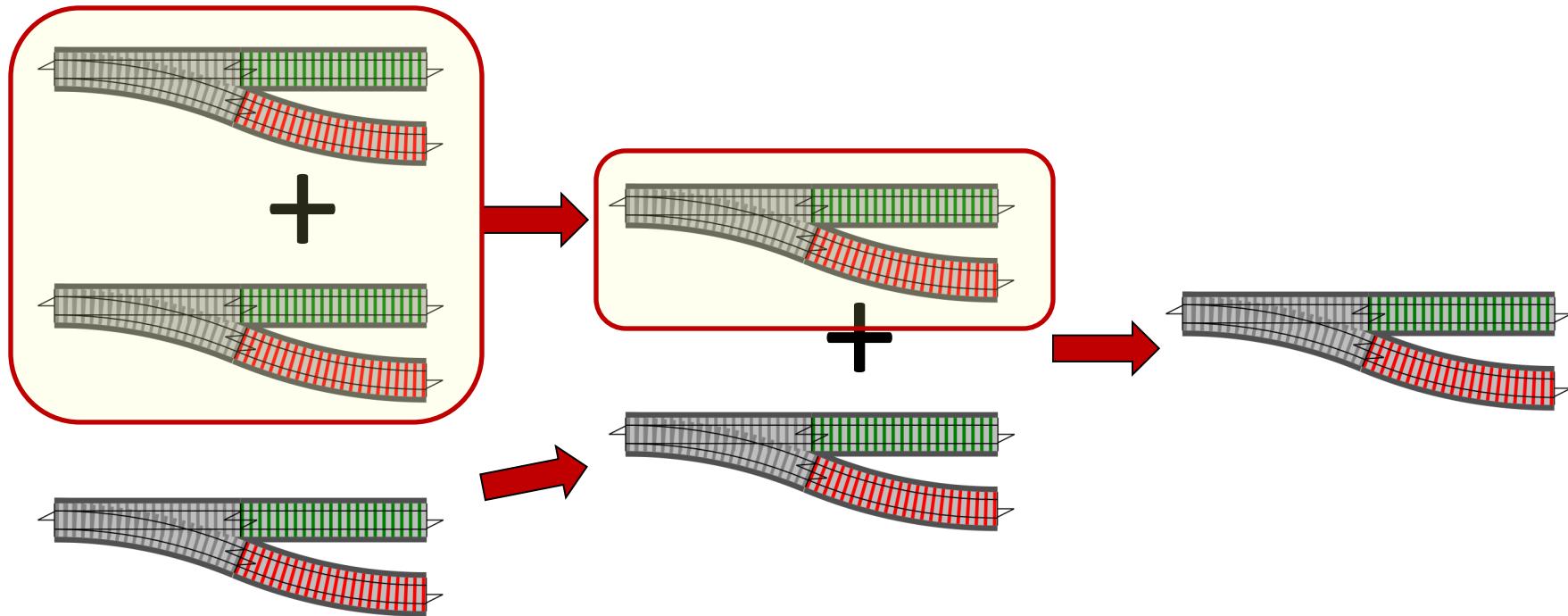
# Combining switches

Trick: if we create an operation that combines pairs into a new switch, we can repeat to combine as many switches as we like.



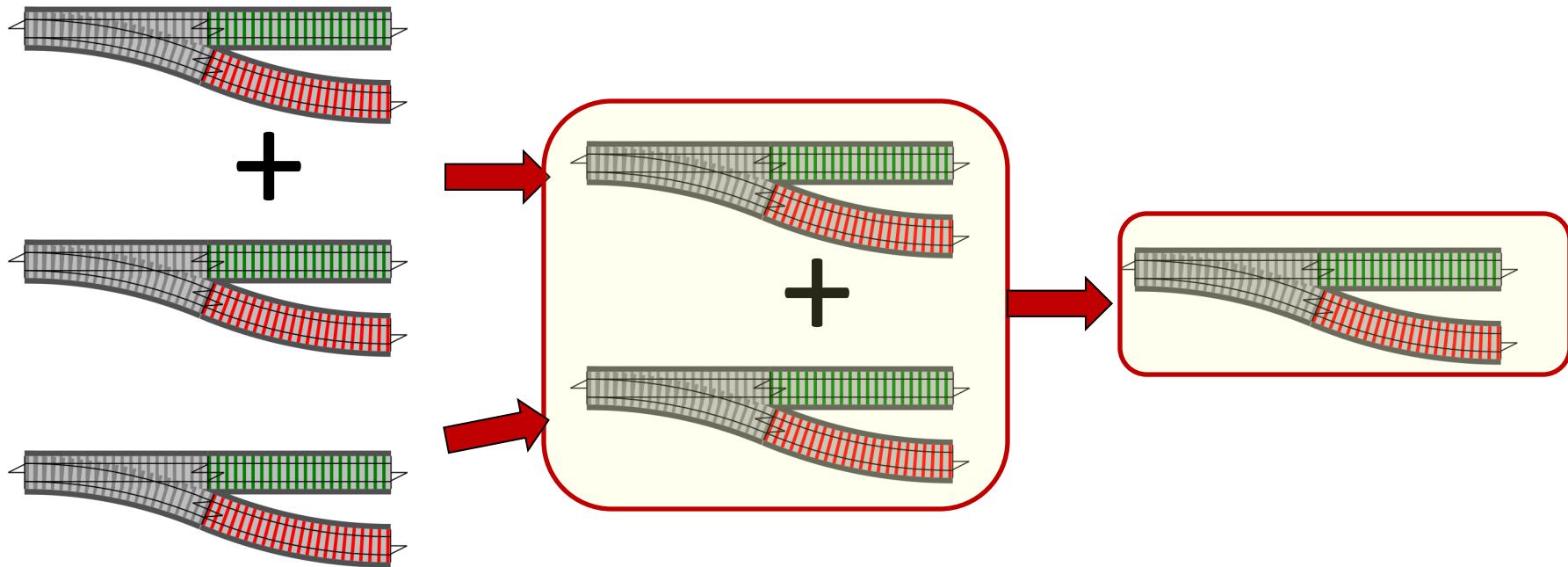
# Combining switches

Trick: if we create an operation that combines pairs into a new switch, we can repeat to combine as many switches as we like.



# Combining switches

Trick: if we create an operation that combines pairs into a new switch, we can repeat to combine as many switches as we like.

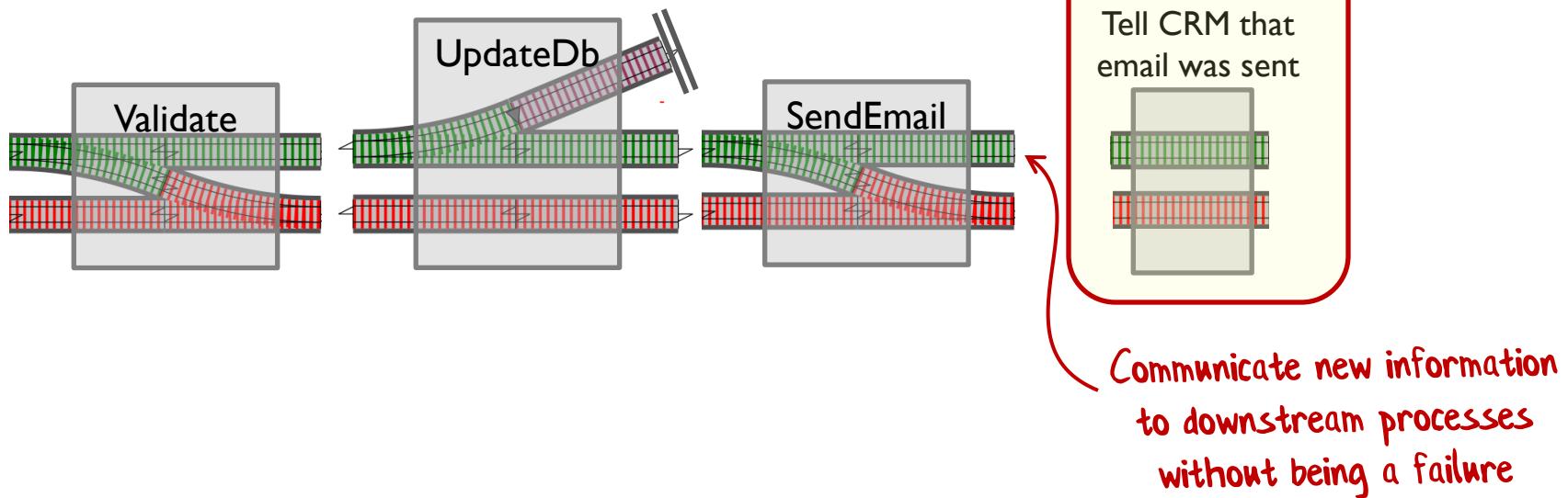


-> For more, see "monoids without tears"

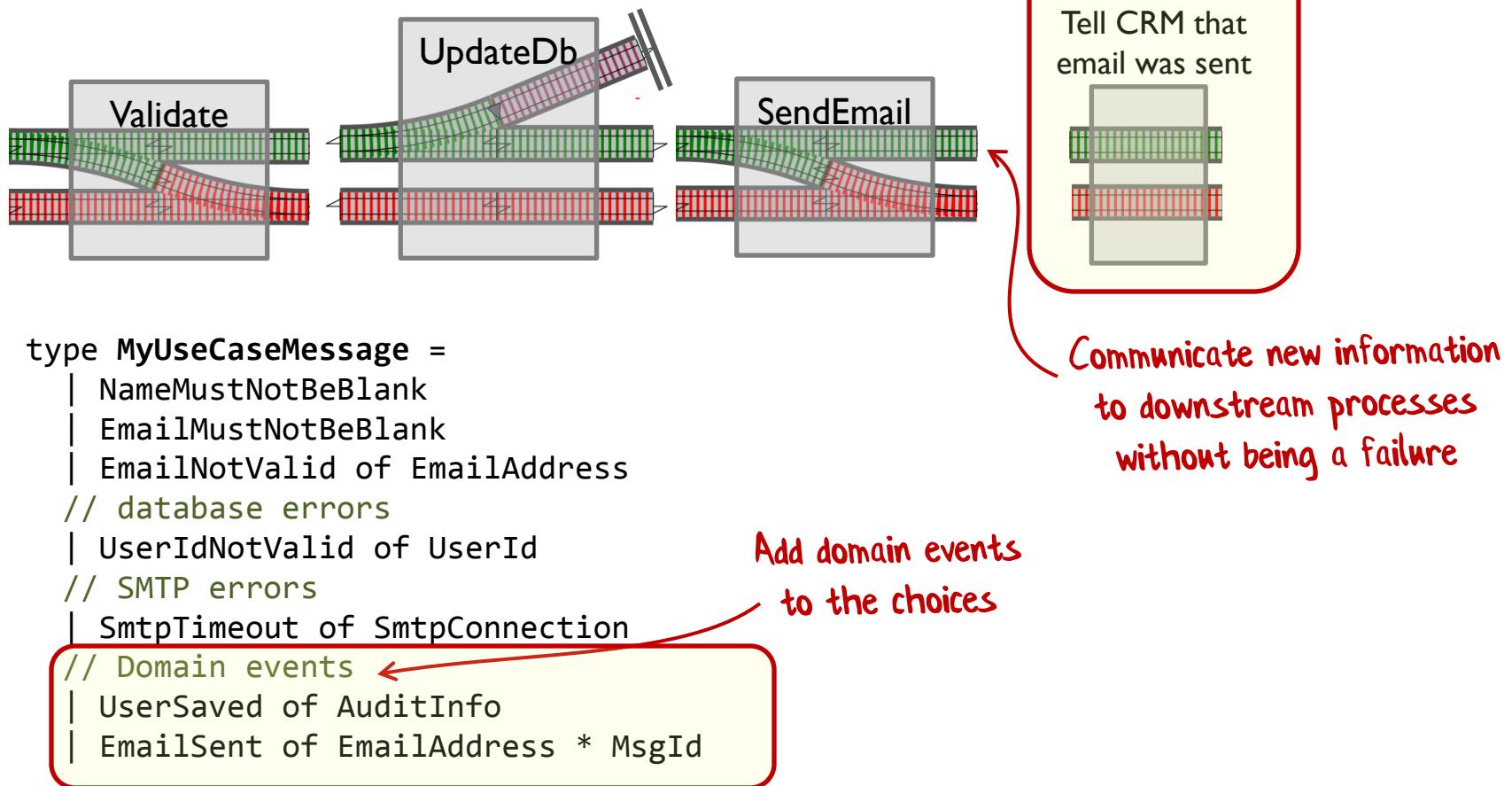
# Domain events

Communicating information to  
downstream functions

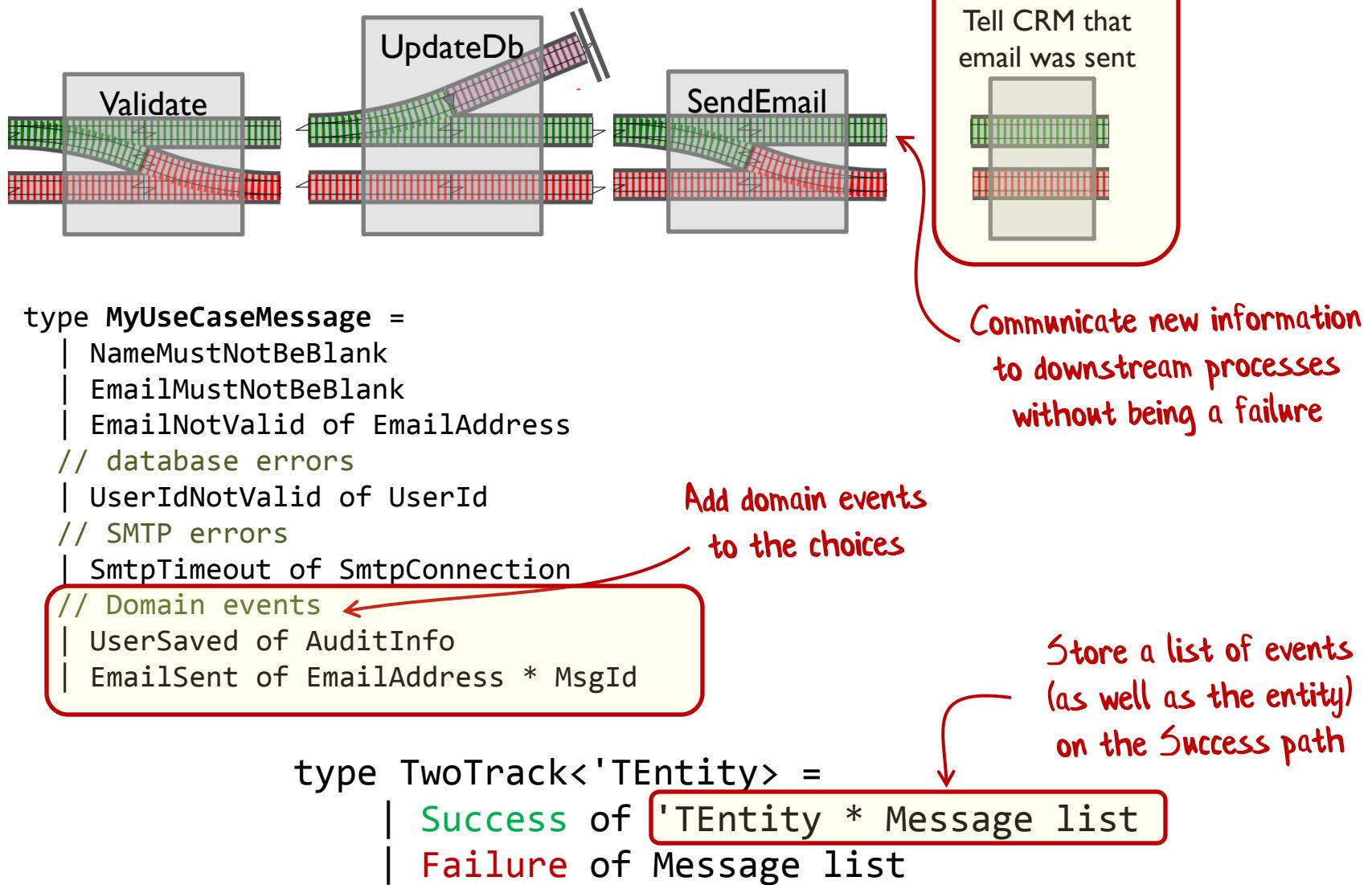
# Events are not errors



# Events are not errors



# Events are not errors



# Comic Interlude

Why can't a train  
driver be electrocuted?

I don't know,  
why can't a train driver  
be electrocuted?

Because he's not  
a conductor!



# Some topics not covered...

...but could be handled  
in an obvious way.

## Topics not covered

- Errors across service boundaries
- Async on success path (instead of sync)
- Compensating transactions  
(instead of two phase commit)
- Logging (tracing, app events, etc.)

# Summary

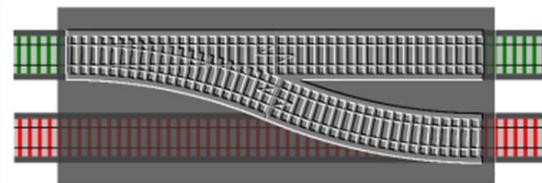
A recipe for handling errors in a  
functional way

# Recipe for handling errors in a functional way

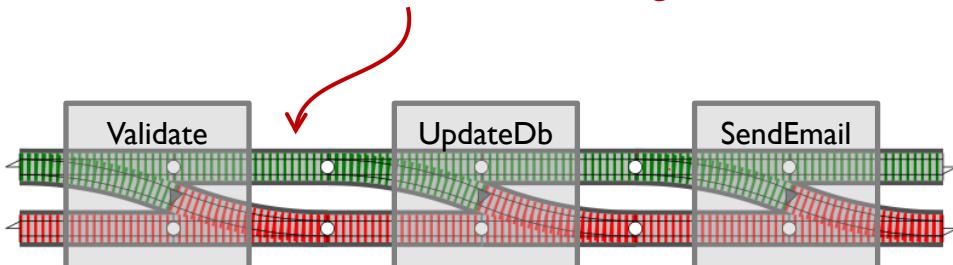
Step 1: Create a Result type

```
type TwoTrack<'TEntity> =  
| Success of 'TEntity * Message list  
| Failure of Message list
```

Step 2: Use "bind" to convert switches to two-track functions



Step 3: Use composition to glue the two-track functions together



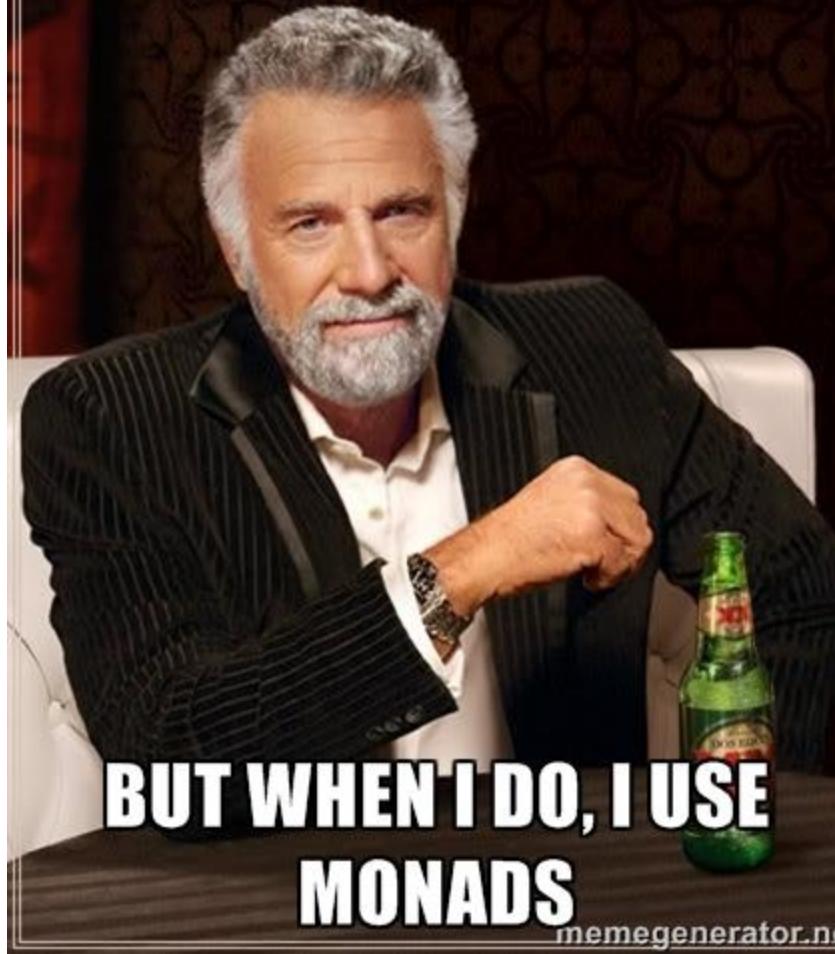
Step 4: Make error cases first class citizens

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

# Demo

<http://bit.ly/rop-example>

I DON'T ALWAYS HAVE ERRORS



BUT WHEN I DO, I USE  
MONADS

[memegenerator.net](http://memegenerator.net)



## Railway Oriented Programming

@ScottWlaschin

[fsharpforfunandprofit.com/rop](http://fsharpforfunandprofit.com/rop)

FPbridge.co.uk

Let me know if you  
need help with F#

Slides will be  
available here  
& code too.

<http://bit.ly/rop-example>

Example code