I'll be using C# and F# code examples, but the concepts will work in most programming languages.

DotNext 2019
The Power Of Composition

@ScottWlaschin
fsharpforfunandprofit.com

The Power Of Composition

1. The philosophy of composition
2. Ideas of functional programming
   – Functions and how to compose them
   – Types and how to compose them
3. Composition in practice
   – Roman Numerals
   – FizzBuzz gone carbonated
   – Uh oh, monads!
   – A web service

THE PHILOSOPHY OF
COMPOSITION

Prerequisites for understanding composition

- ~~You must have been a child at some point~~
- You must have played with Lego
- You must have played with toy trains

Actually not true!
google "AFOL"

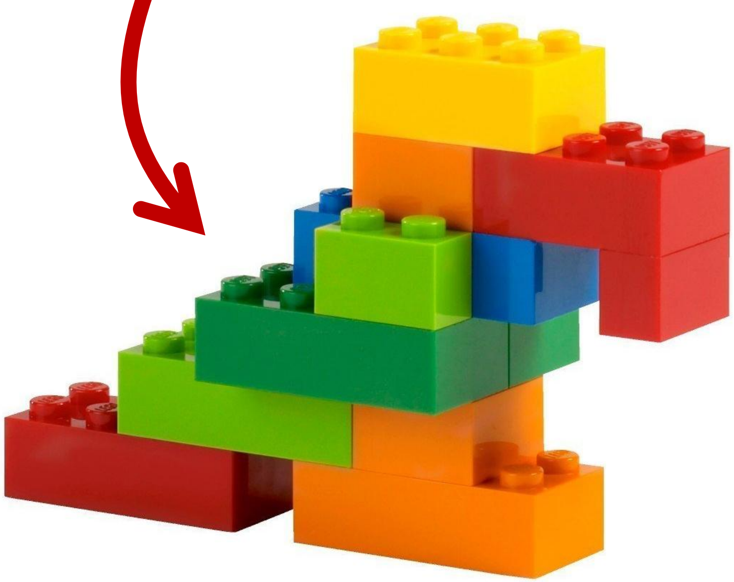
Lego Philosophy

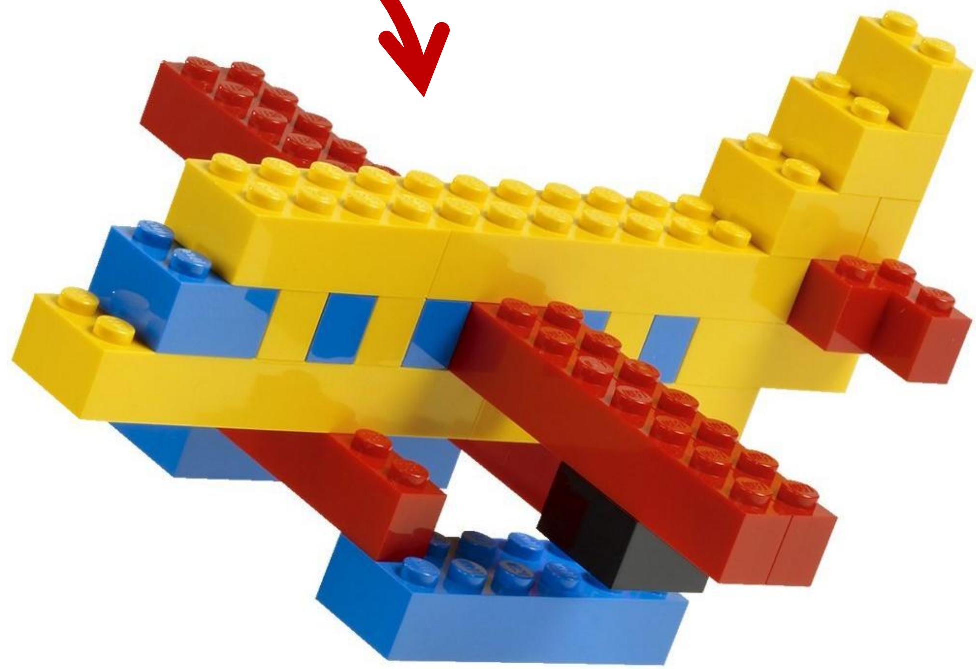
Lego Philosophy

1. All pieces are designed to be connected
2. The pieces are reusable in many contexts
3. Connect two pieces together and get another "piece" that can still be connected

All pieces are designed to be connected

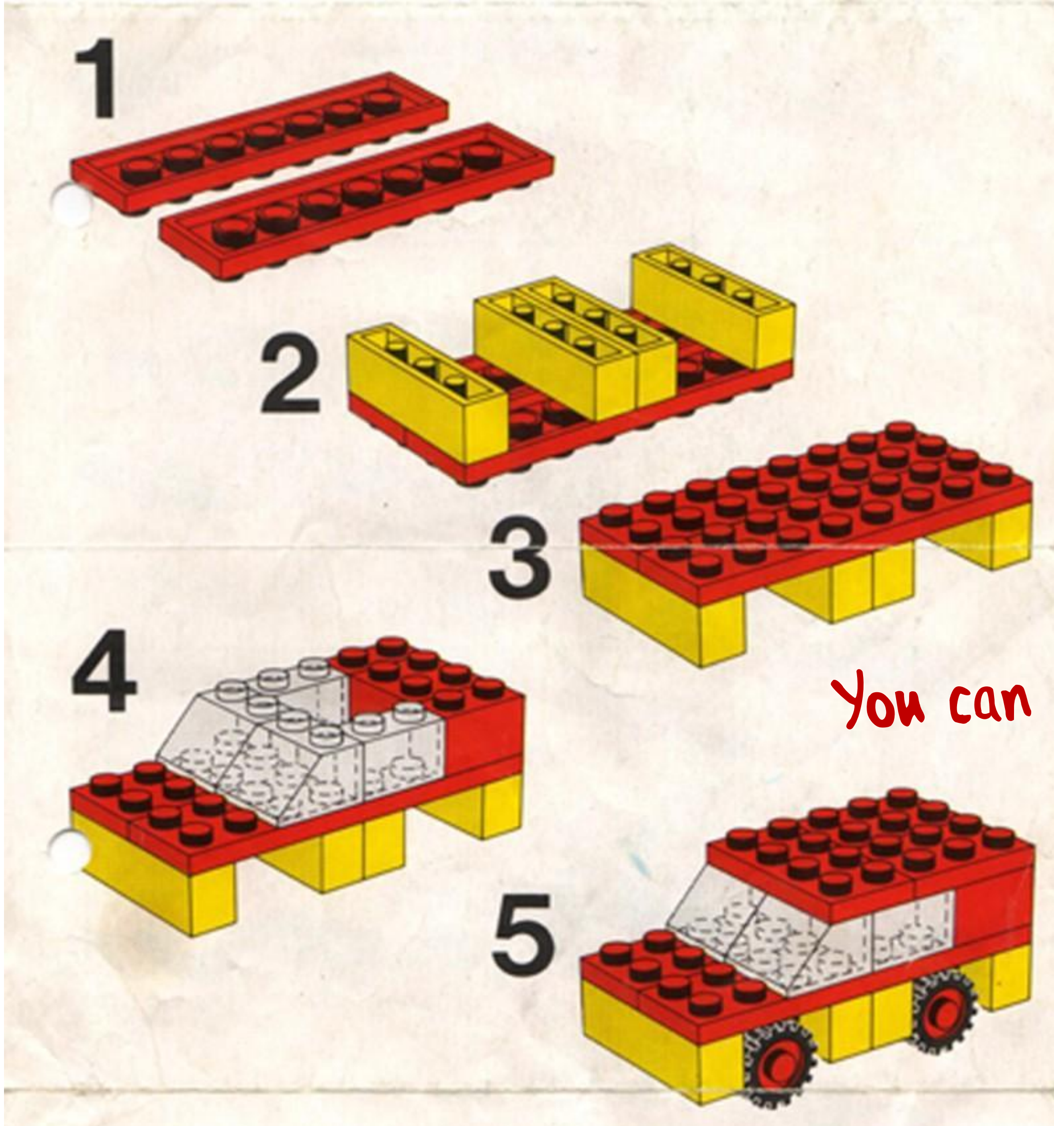
The pieces are reusable in different contexts





They are self contained.
No strings attached (literally).

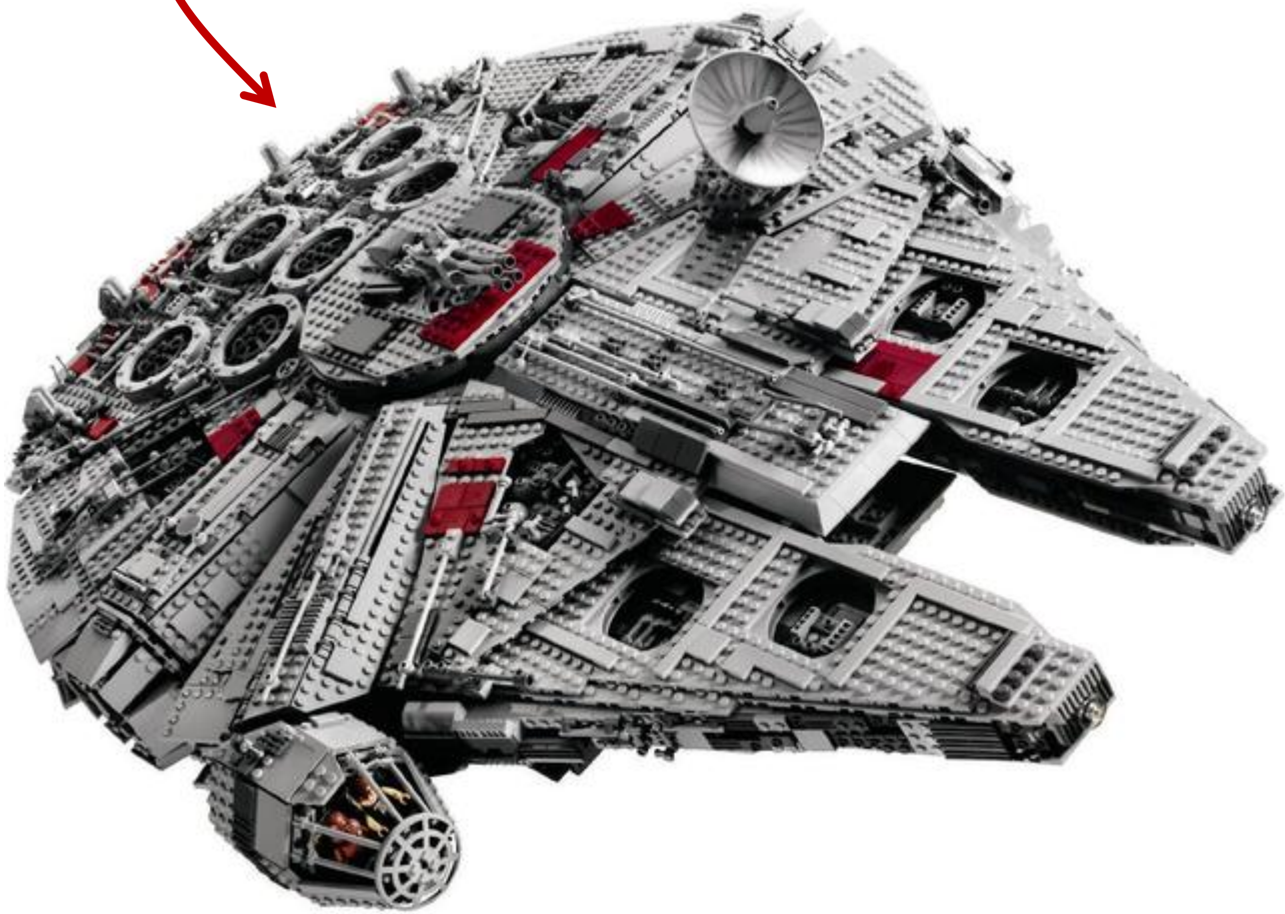
Connect two pieces together and
get another "piece" that can still be connected



You can keep adding and adding.

Make big things from small things in the same way

The Power of Composition



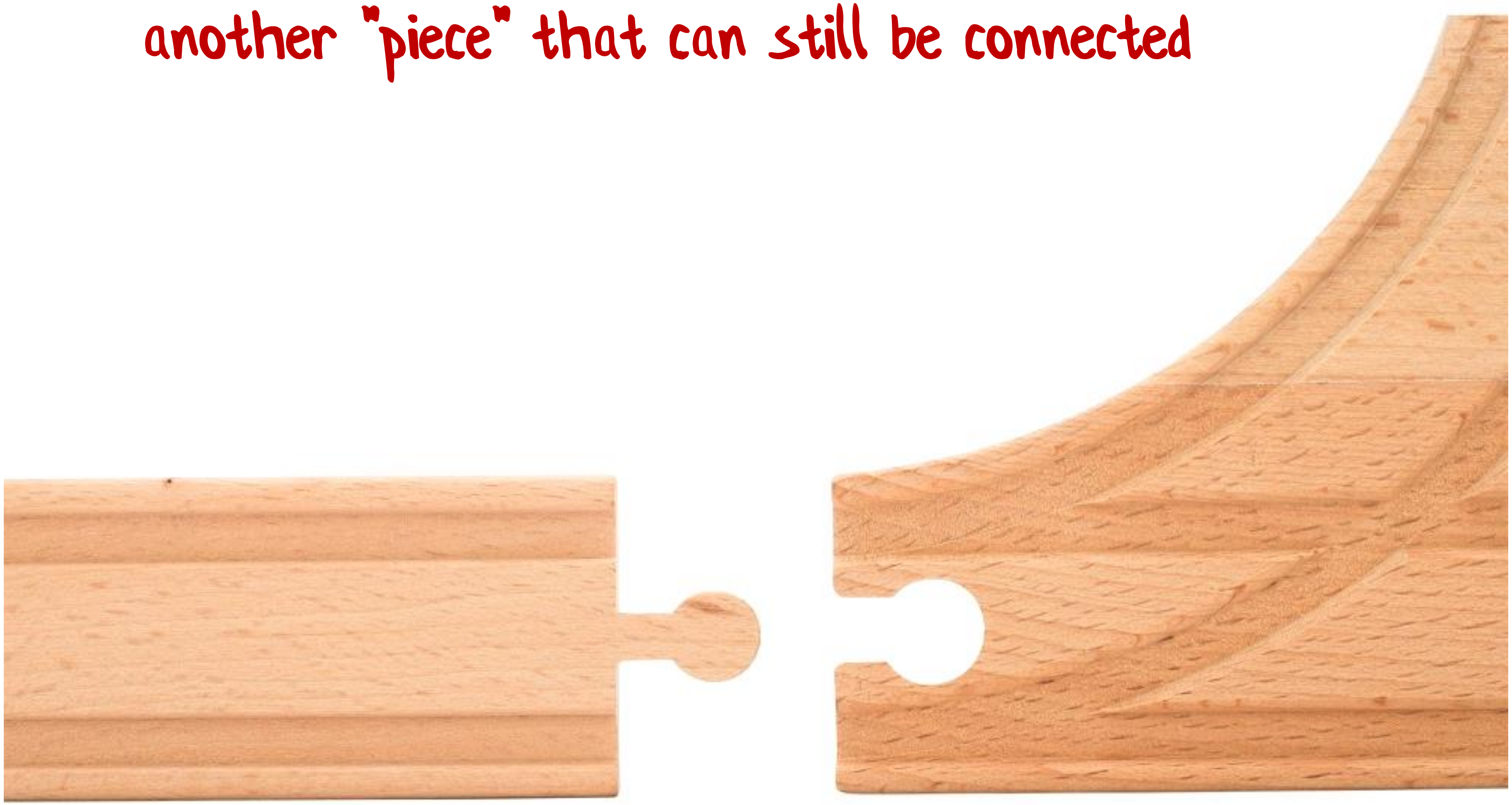
Wooden Railway Track Philosophy

1. All pieces are designed to be connected
2. The pieces are reusable in many contexts
3. Connect two pieces together and get another "piece" that can still be connected

All pieces are designed to be connected

The pieces are reusable in different contexts

Connect two pieces together and get another "piece" that can still be connected



You can keep adding and adding.

Make big things from small things in the same way

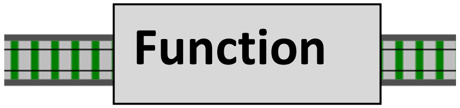
The Power of Composition

If you understand Lego and wooden railways, then you know everything about composition!

THE IDEAS OF
FUNCTIONAL PROGRAMMING

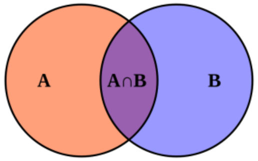
Four ideas behind FP

1. Functions are things



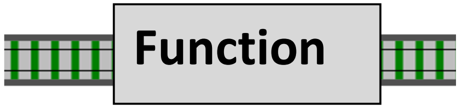
2. Build bigger functions using composition

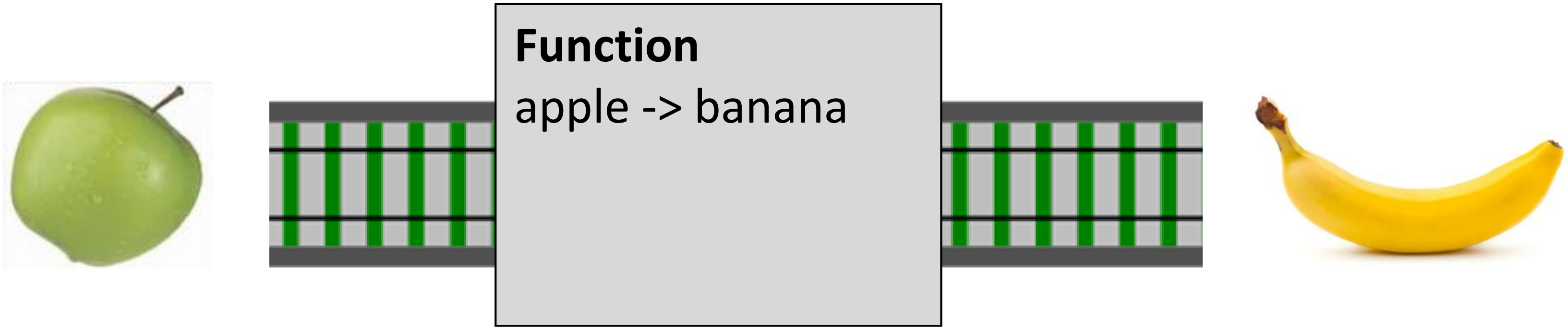
3. Types are not classes



4. Build bigger types using composition

FP idea #1:
Functions are things





A function is a thing which transforms inputs to outputs

Another word for reusable!

A function is a standalone thing, not attached to a class

No strings attached!

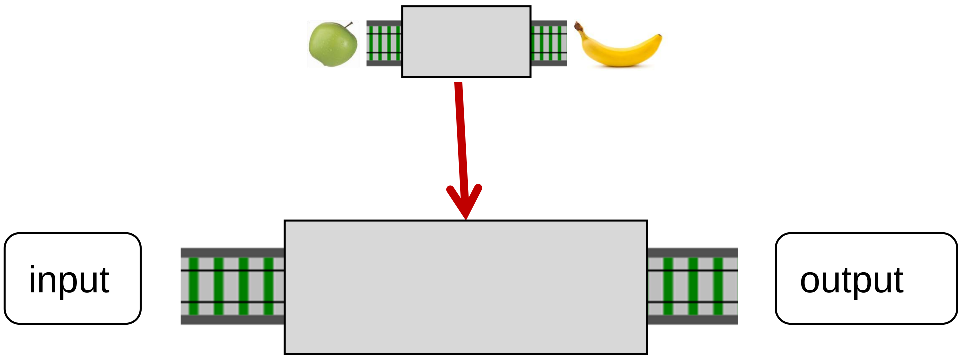
A function is a standalone thing, not attached to a class

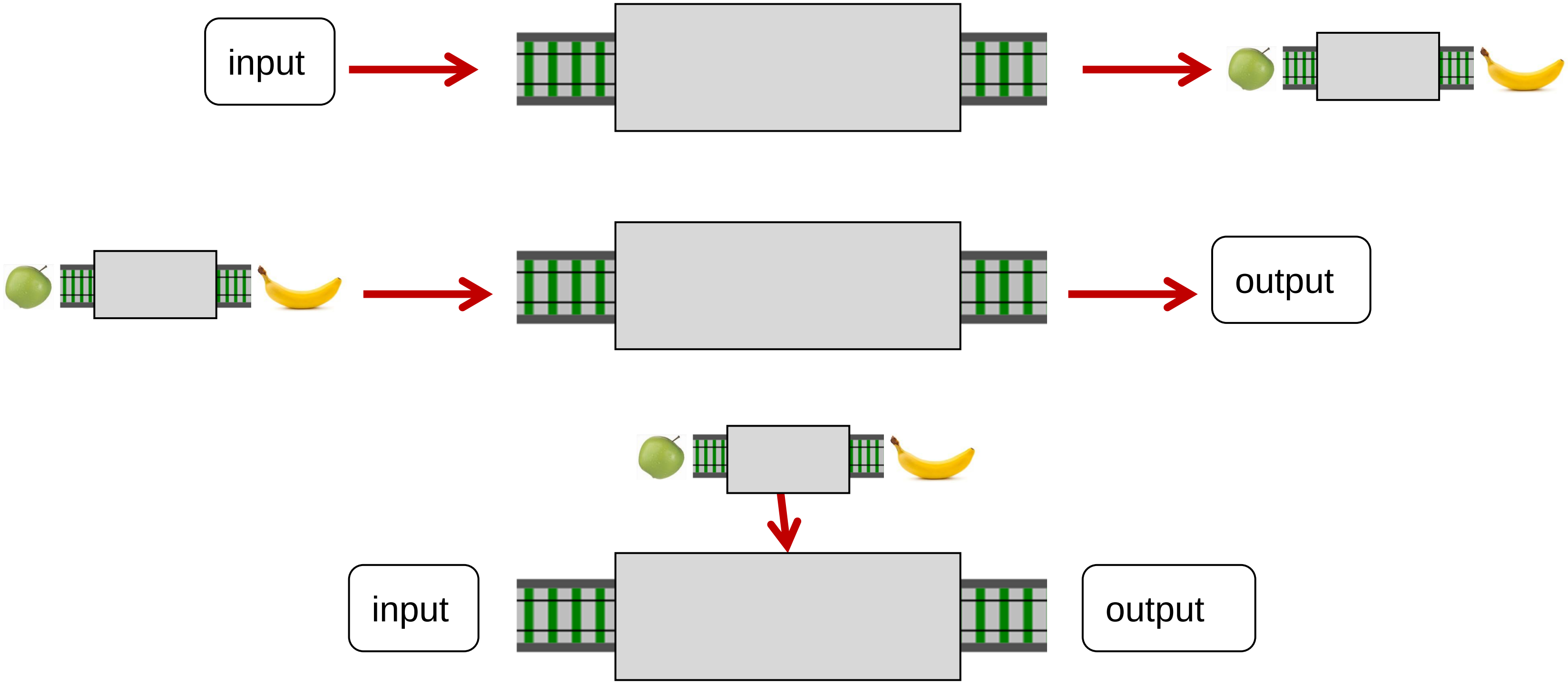
It can be used for inputs and outputs of other functions

A function can be an output thing

A function can be an input thing

A function can be a parameter



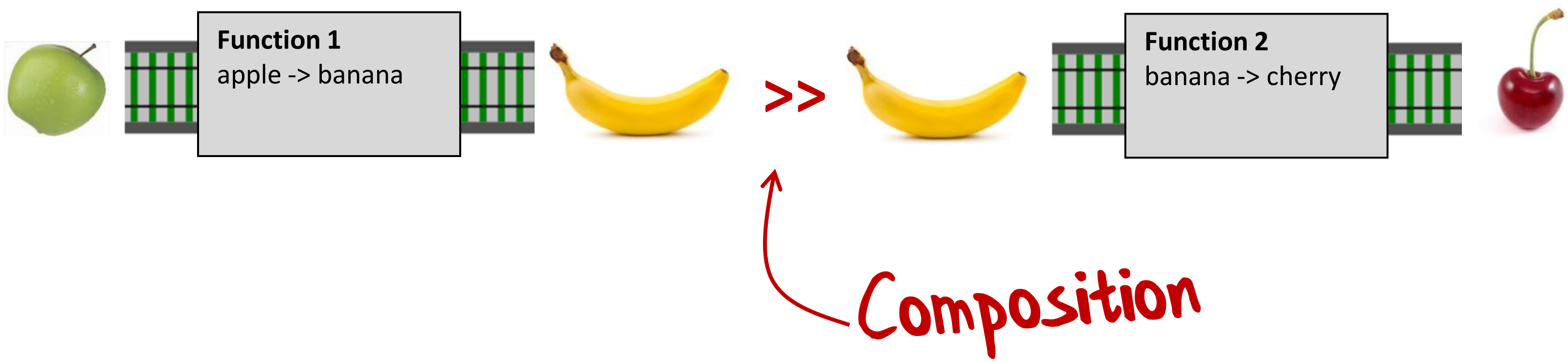


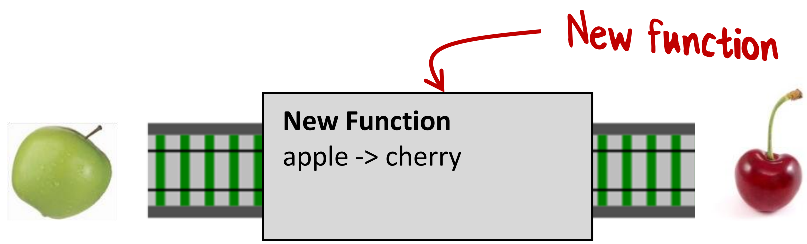
You can build very complex systems from this simple foundation!

FP idea #2:

Build bigger functions using composition


Function 1
apple -> banana
Function 2
banana -> cherry


Function 1
apple -> banana
>>
Function 2
banana -> cherry
Composition



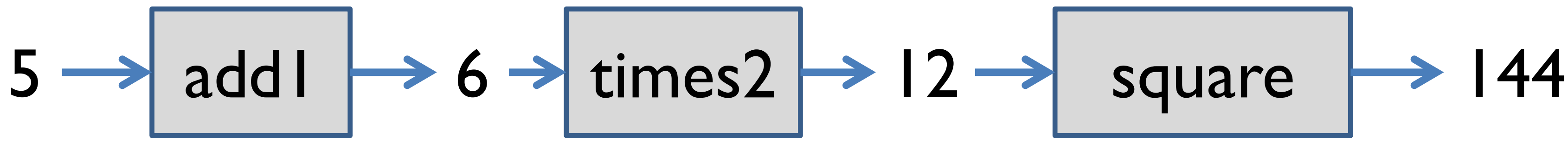
Can't tell it was built from smaller functions!

Where did the banana go?

Function composition in F# and C# using the "piping" approach

```
int add1(int x) => x + 1;
int times2(int x) => x * 2;
int square(int x) => x * x;

add1(5);                        // = 6
times2(add1(5));                // = 12
square(times2(add1(5)));        // = 144
```

Nested function calls
can be confusing if too deep



This is often easier to understand

Pipe symbol

F# example

```
5 |> add1                       // = 6
5 |> add1 |> times2             // = 12
5 |> add1 |> times2 |> square   // = 144
```

pipe

Pipe extension method

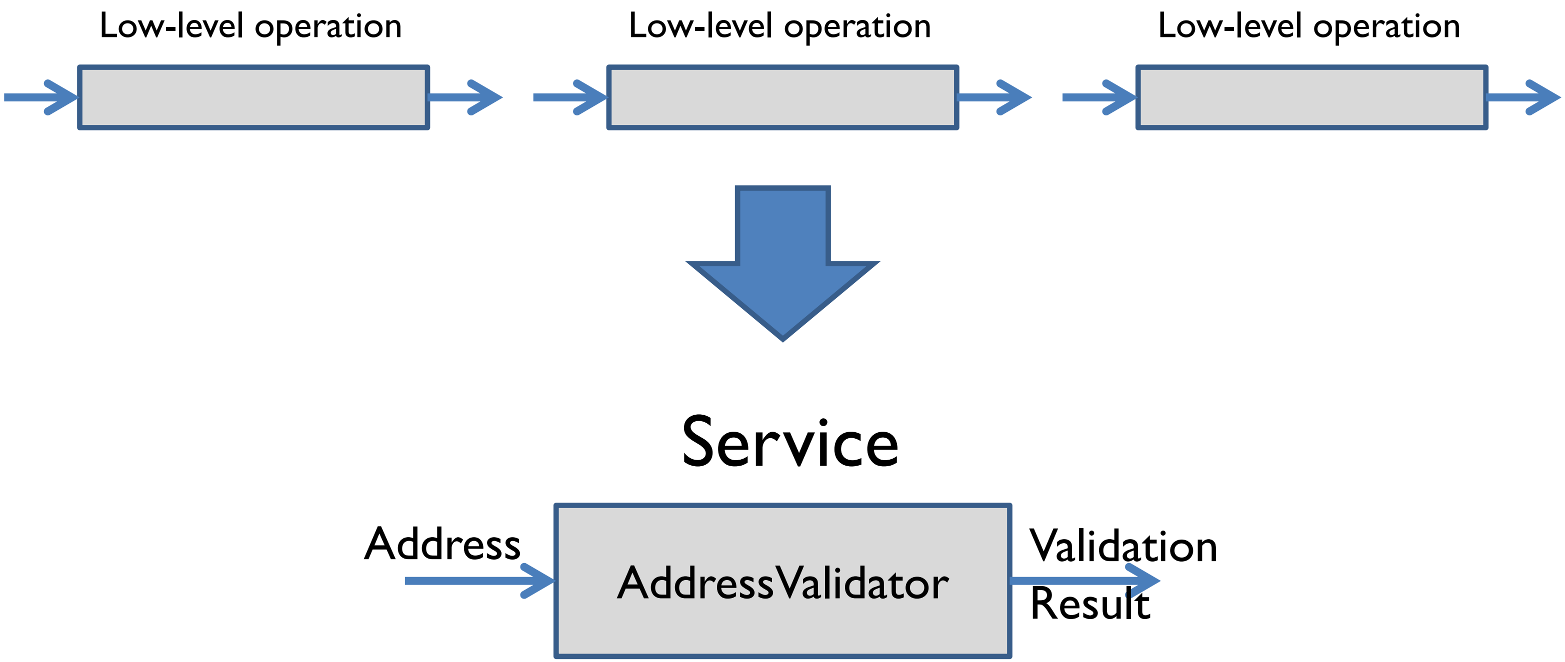
C# example

```
5.Pipe(add1);
5.Pipe(add1).Pipe(times2);
5.Pipe(add1).Pipe(times2).Pipe(square);
```

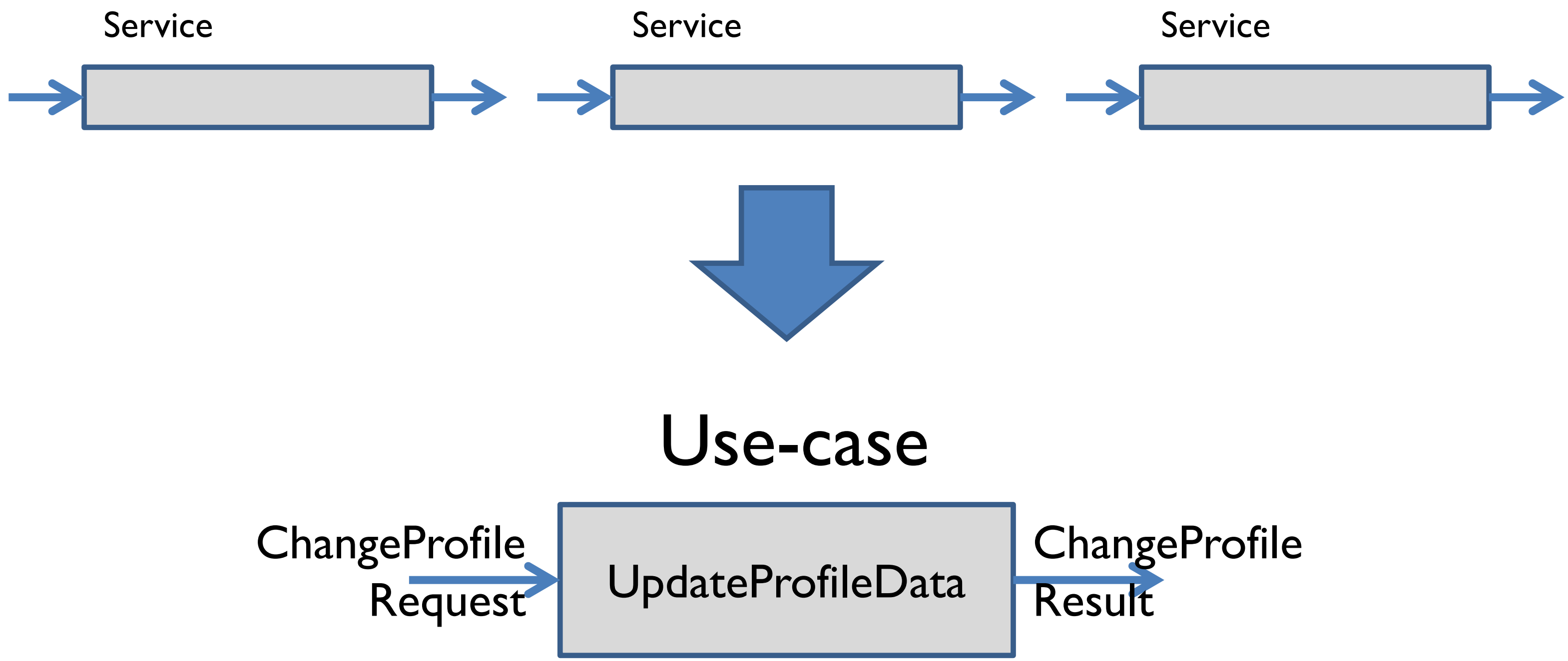
Building big things from functions

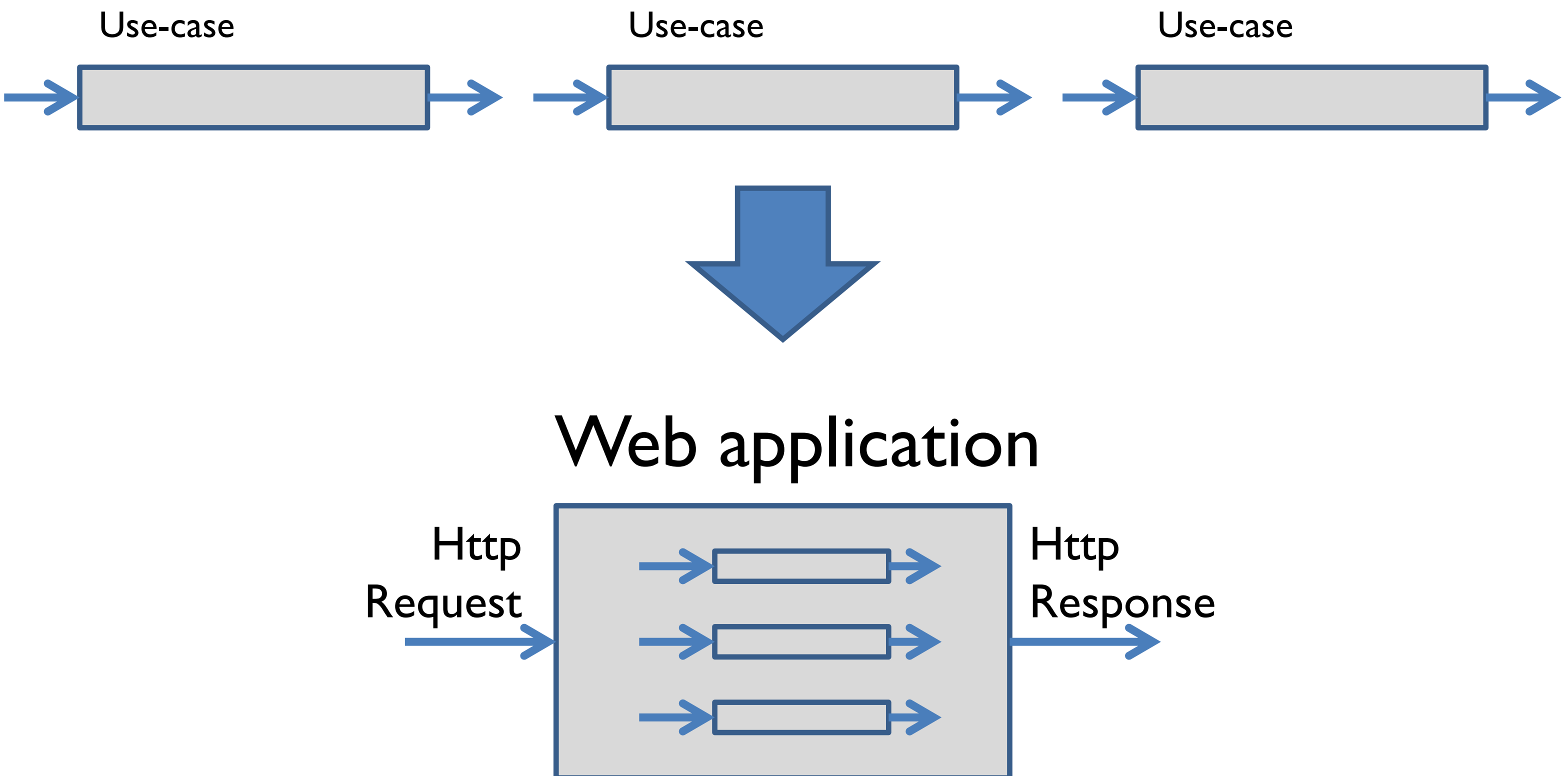
It's compositions all the way up

Low-level operation



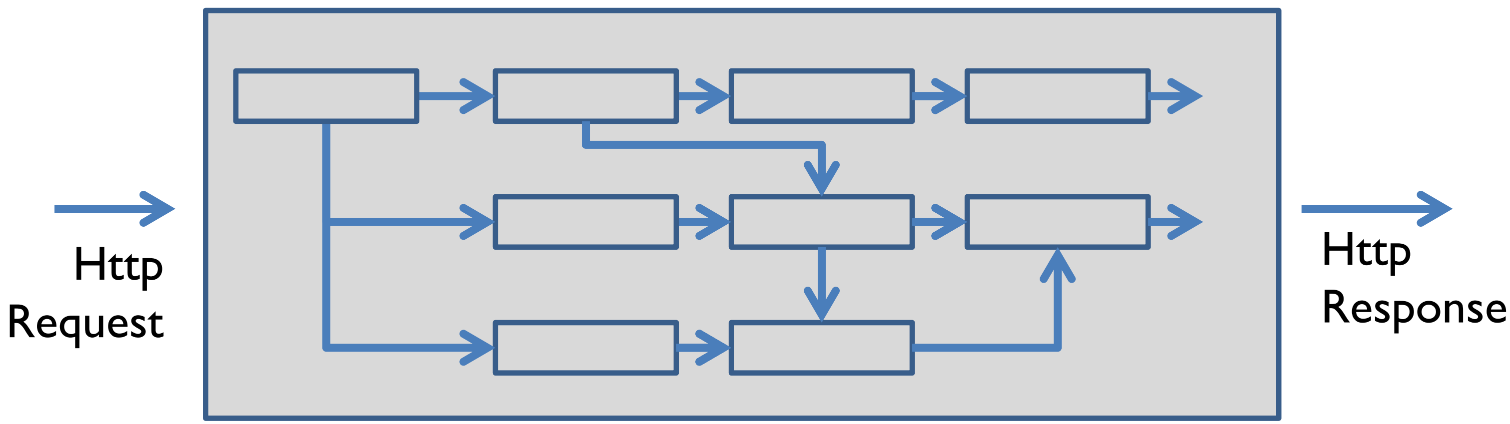
A "Service" is just like a microservice
but without the "micro" in front


Service
Service
Service
Use-case
ChangeProfile Request
UpdateProfileData
ChangeProfile Result



"Composition is the same at all scales"

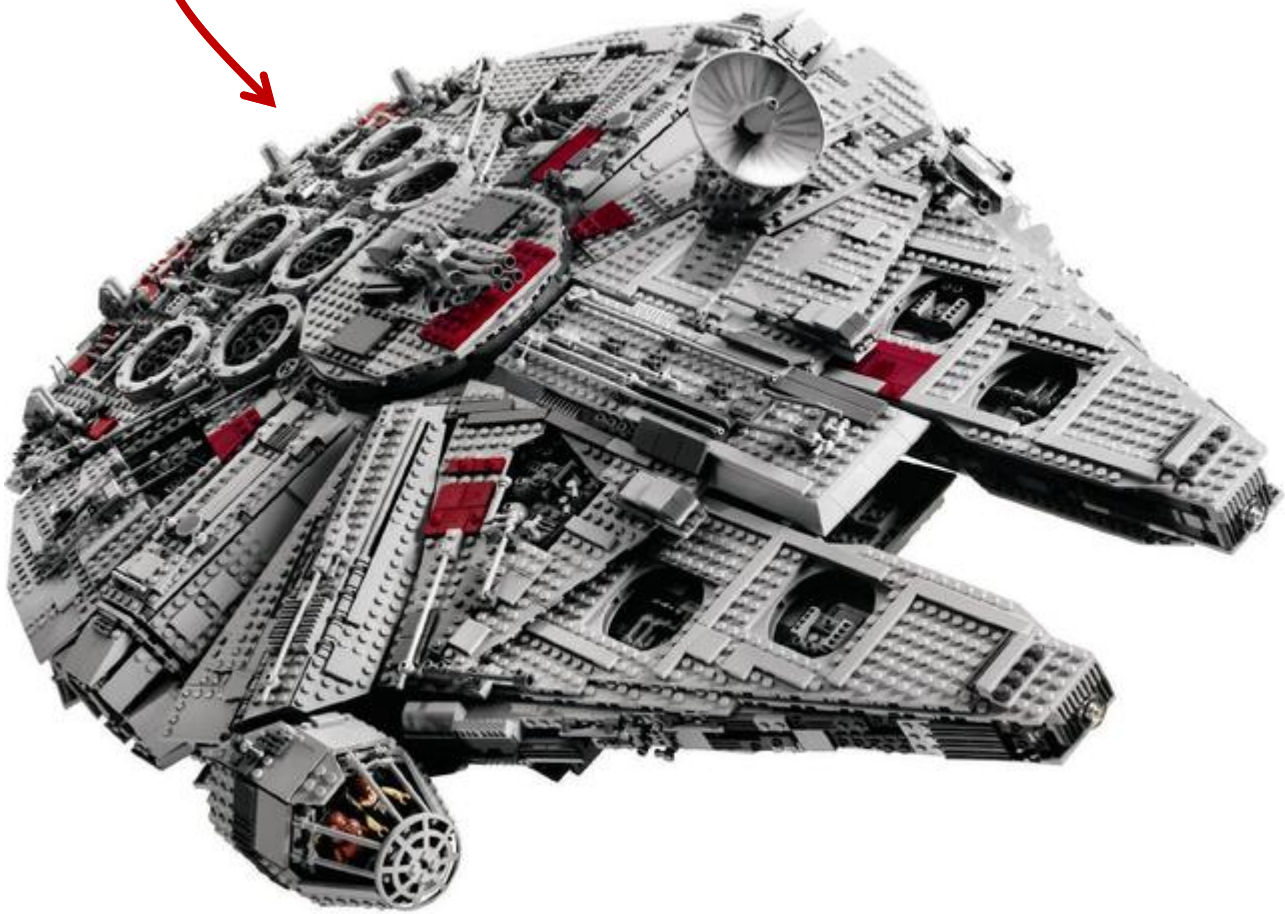
The Power of Composition



Even for complex applications, data flows only in one direction

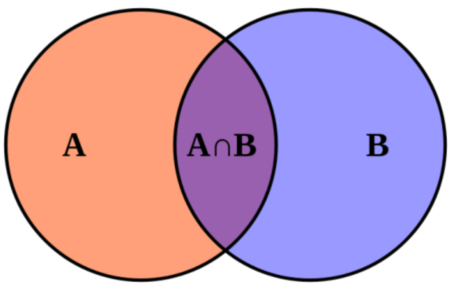
The Power of Composition

The Power of Composition

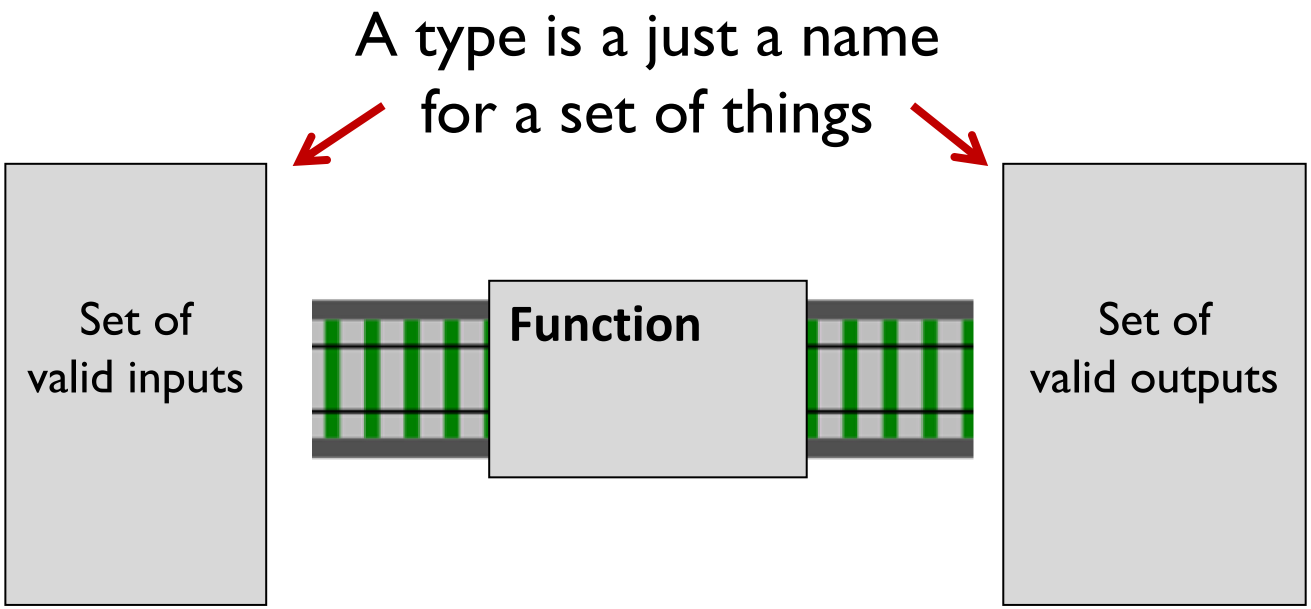


FP idea #3:

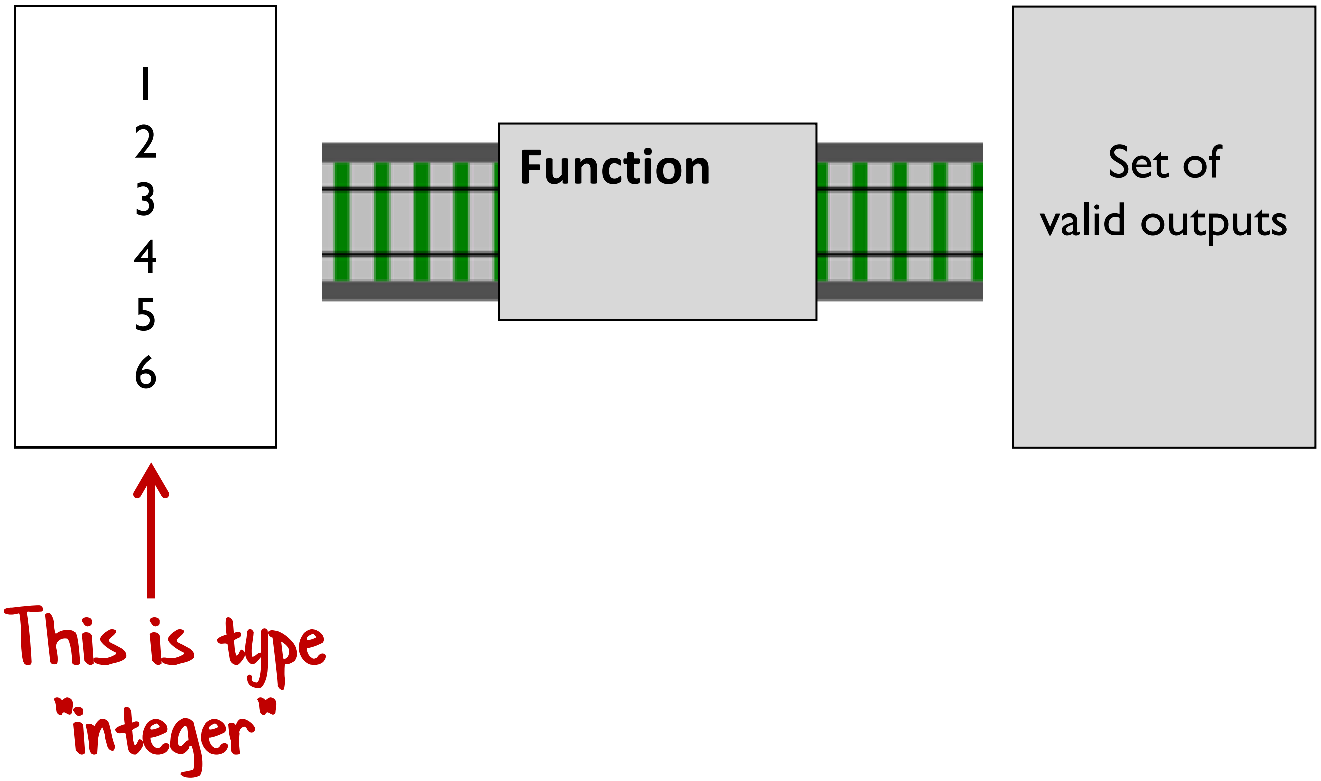
Types are not classes



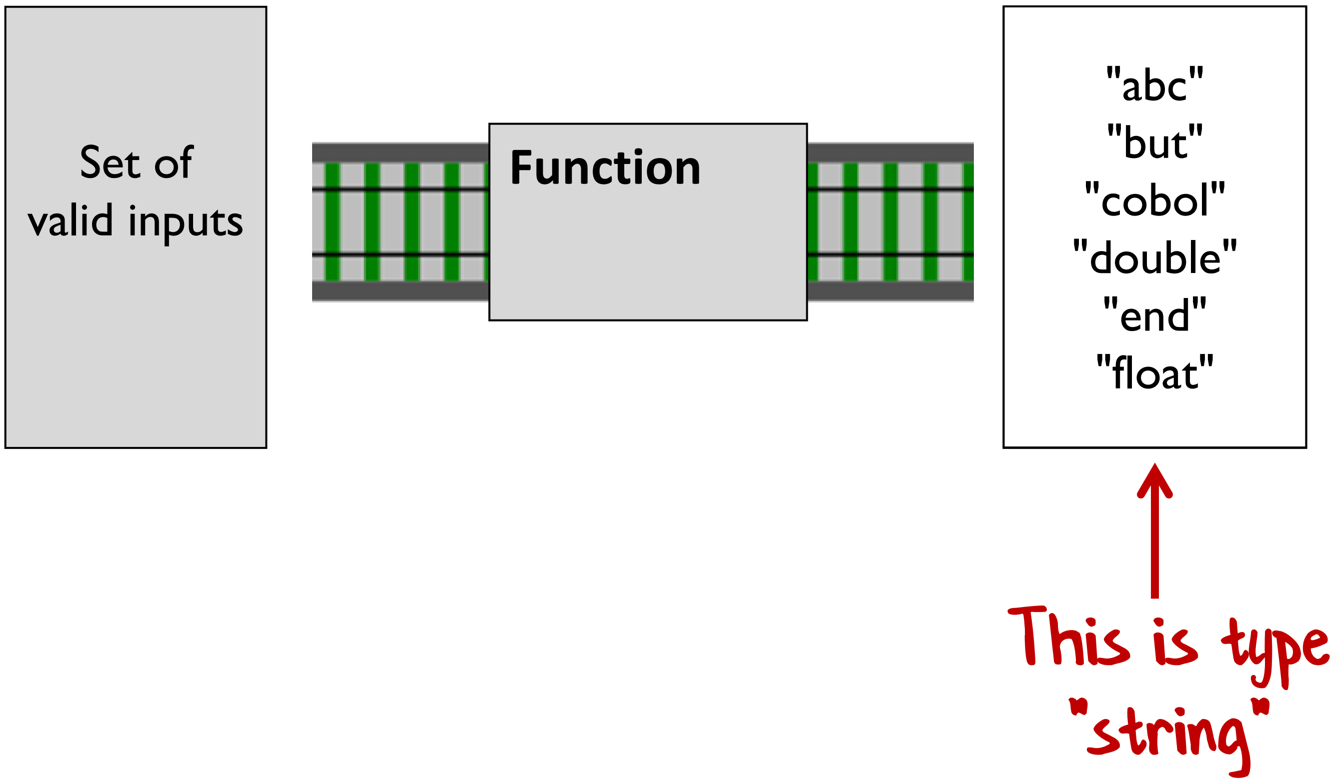
So, what is a type then?



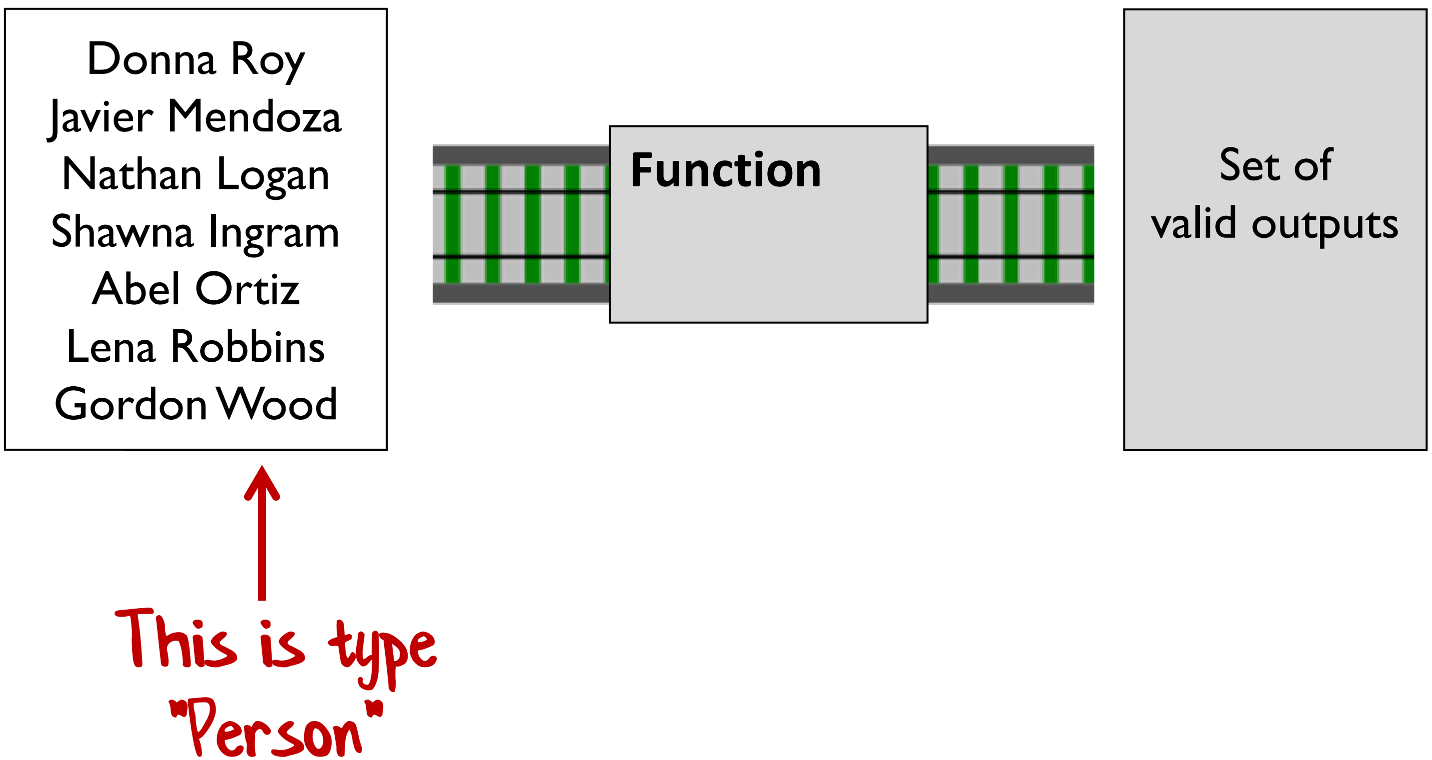
A type is a just a name for a set of things



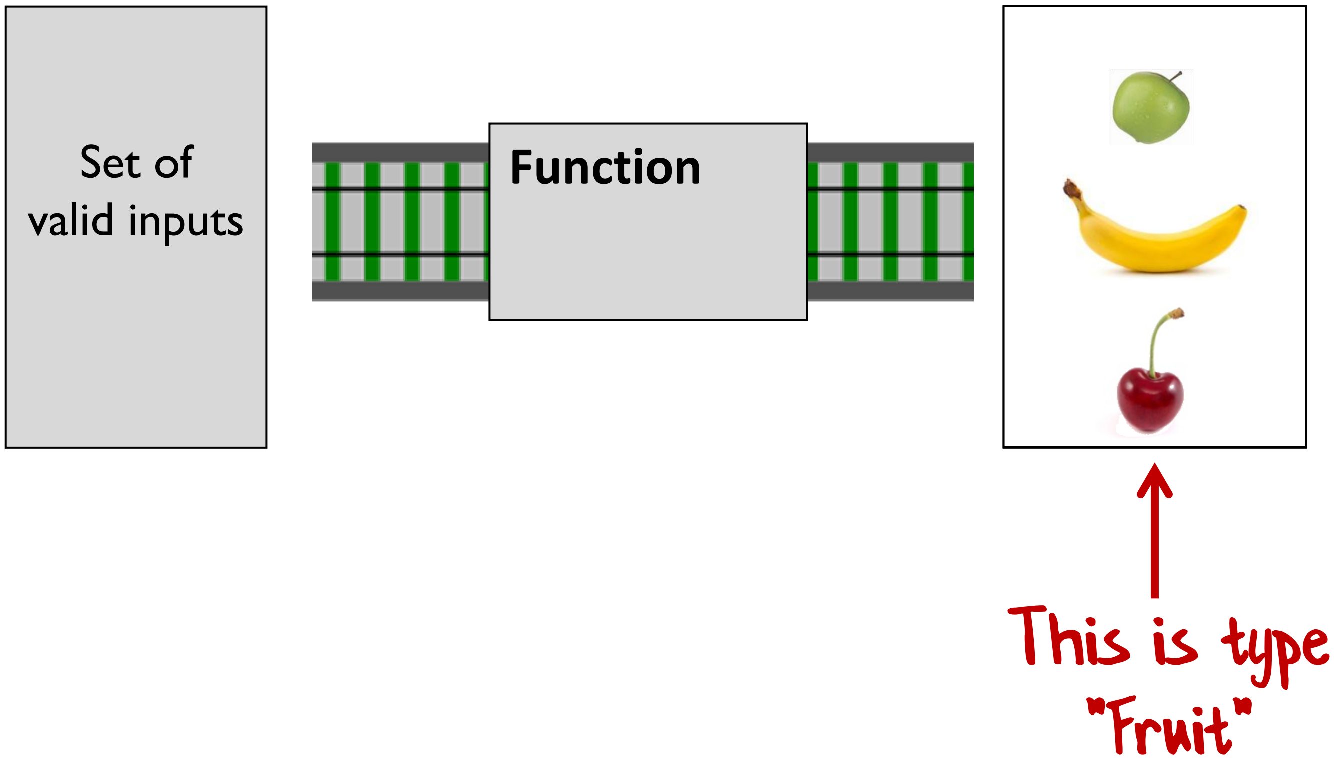
A type is a just a name for a set of things



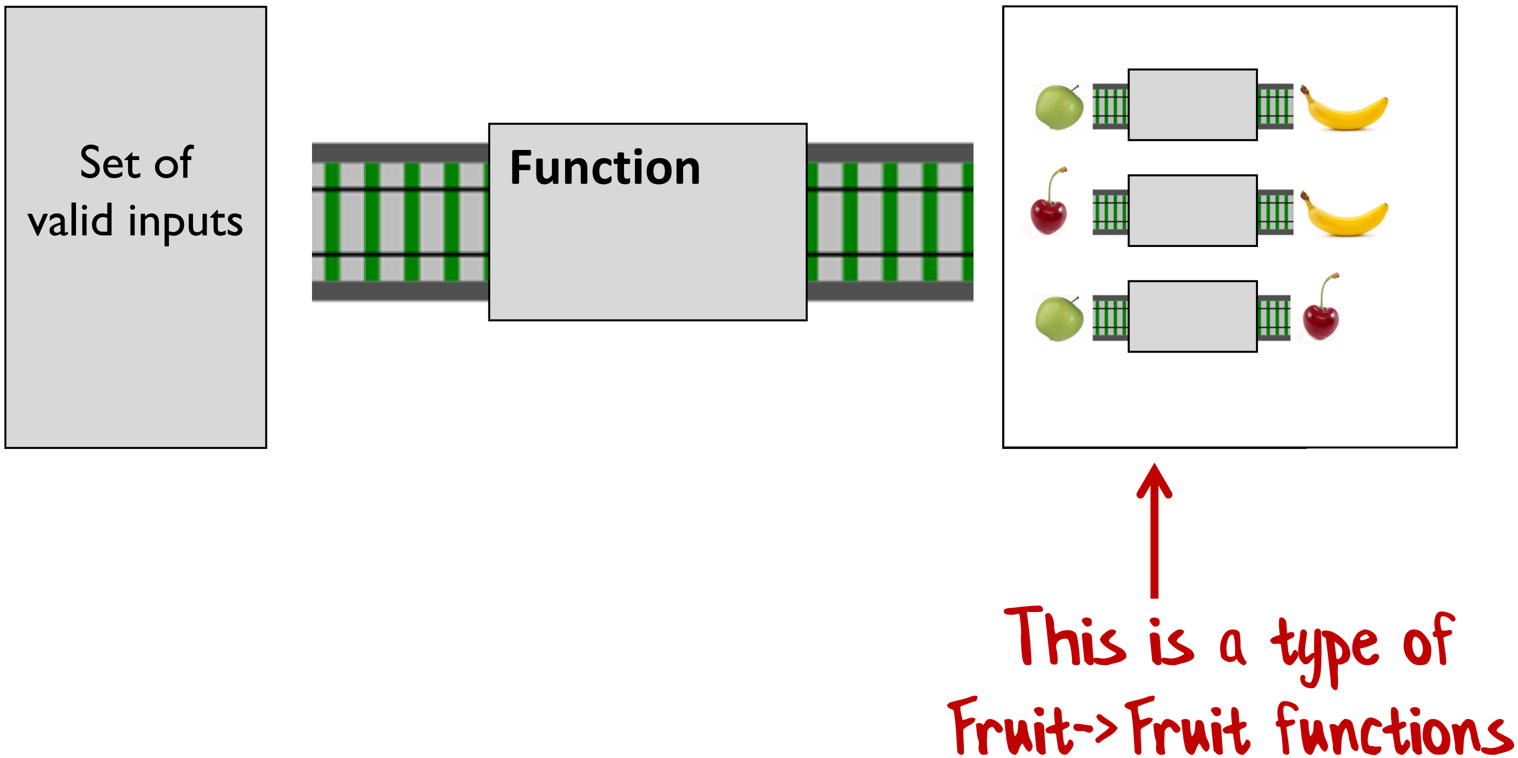
A type is a just a name for a set of things



A type is a just a name for a set of things



A type is a just a name for a set of things



FP idea #4:
Types can be composed too

Composable

~~Algebraic~~ type system

Only possible because behavior is separate from data!

Bigger types are built from smaller types by:

- Composing with "AND"
- Composing with "OR"

Compose with “AND”

FruitSalad =

 AND  AND 

Compose with "AND"

```
enum AppleVariety { Red, Green }
enum BananaVariety { Yellow, Brown }
enum CherryVariety { Tart, Sweet }

struct FruitSalad
{
    AppleVariety Apple;
    BananaVariety Banana;
    CherryVariety Cherry;
}
```

C# example

Apple AND
Banana AND
Cherry

Compose with "AND"

```
type AppleVariety = Red | Green
type BananaVariety = Yellow | Brown
type CherryVariety = Tart | Sweet

type FruitSalad = {
    Apple: AppleVariety
    Banana: BananaVariety
    Cherry: CherryVariety
    }
```

F# example

F# "record" type

Compose with “OR”

Snack =  OR  OR 

F# "choice" type

```
type Snack =
    | Apple of AppleVariety
    | Banana of BananaVariety
    | Cherry of CherryVariety
```

Like an enum in C# but with extra information for each case

A real world example of composing types

Some requirements:

We accept three forms of payment:
Cash, Paypal, or CreditCard.

For Cash we don't need any extra information
For Paypal we need an email address
For Cards we need a card type and card number

How would you implement this?

In OO design you would probably implement it as an interface and a set of subclasses, like this:

```
interface IPaymentMethod
{..}

class Cash() : IPaymentMethod
{..}

class Paypal(string emailAddress): IPaymentMethod
{..}

class Card(string cardType, string cardNo) : IPaymentMethod
{..}
```

In F# you would probably implement by <u>composing</u> types, like this:

```
type EmailAddress = string
type CardNumber = string
```

Primitive types

```
type EmailAddress = ...
type CardNumber = …

type CardType = Visa | Mastercard
type CreditCardInfo = {
    CardType : CardType
    CardNumber : CardNumber
    }
```

Choice type (using OR)

Record type (using AND)

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...

type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
```

Choice type

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo

type PaymentAmount = decimal
type Currency = EUR | USD | RUB
```

Another primitive type

Another choice type

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD | RUB

type Payment = {
  Amount : PaymentAmount
  Currency : Currency
  Method : PaymentMethod  }
```

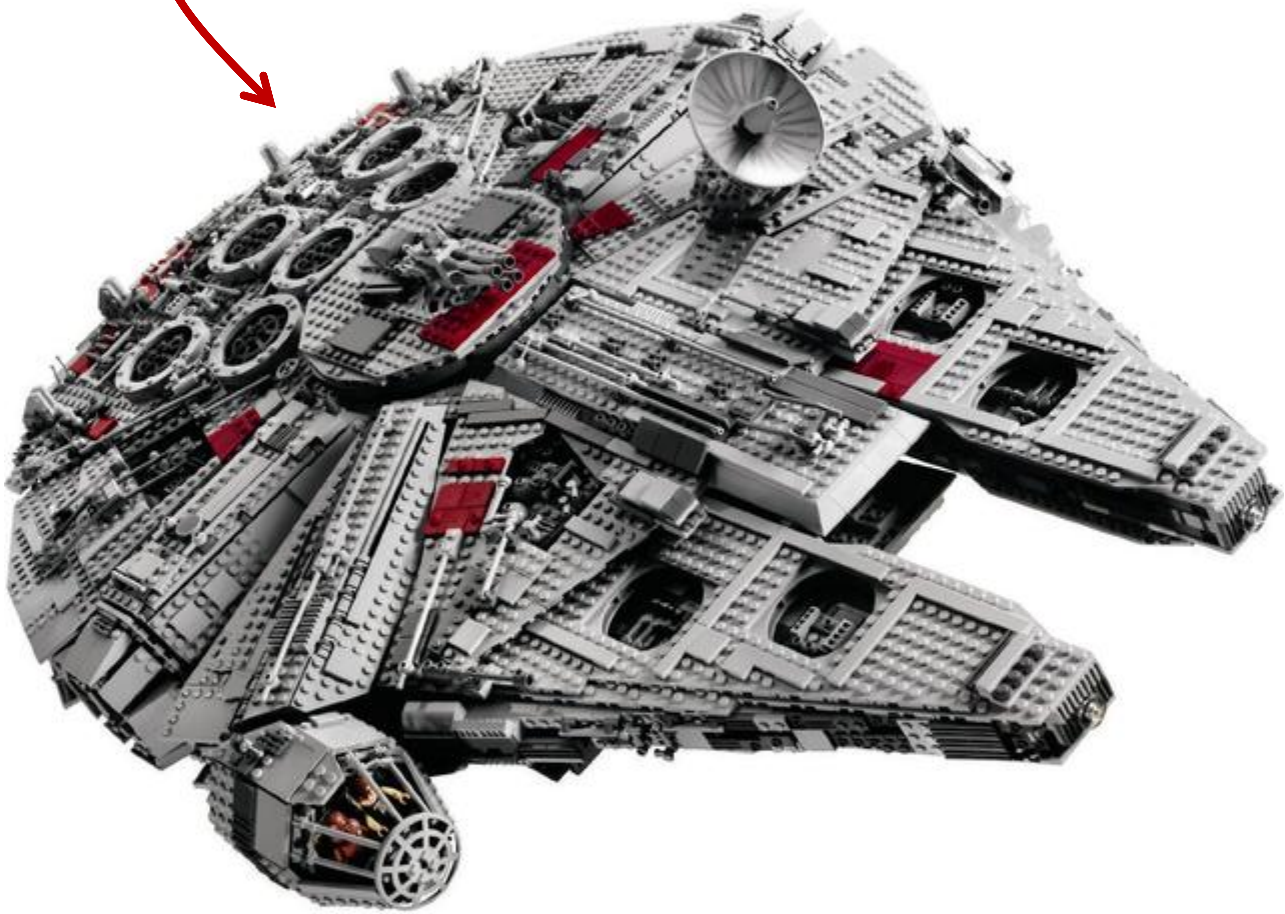
Record type

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD | RUB

type Payment = {
  Amount : PaymentAmount
  Currency : Currency
  Method : PaymentMethod  }
```

Final type built from many smaller types:

The Power of Composition

The Power of Composition



Composable types can be used as executable documentation

👍

Types can be nouns

The domain on one screen!

```
type Suit = Club | Diamond | Spade | Heart
type Rank = Two | Three | Four | Five | Six | Seven | Eight
          | Nine | Ten | Jack | Queen | King | Ace
type Card = { Suit:Suit; Rank:Rank }

type Hand = Card list
type Deck = Card list

type Player = {Name:string; Hand:Hand}
type Game = { Deck:Deck; Players:Player list }

type Deal = Deck -> (Deck * Card)
type PickupCard = (Hand * Card) -> Hand
```

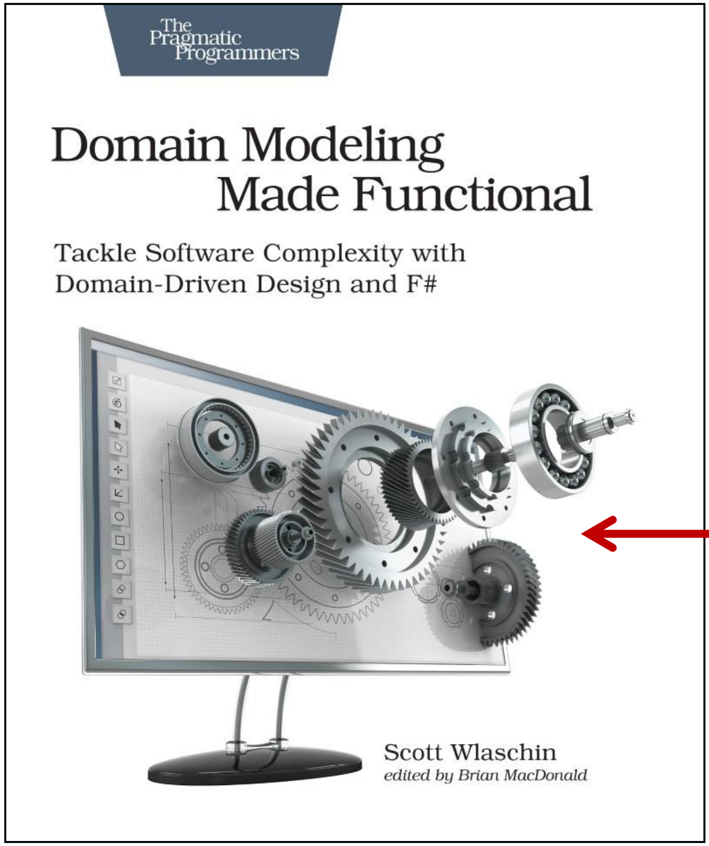
Types can be verbs

```
type CardType = Visa | Mastercard
type CardNumber = string
type EmailAddress = string

type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
  | Bitcoin of BitcoinAddress
```

Can you guess what payment methods are accepted?

A big topic and not enough time ☹ ☹

More on DDD and designing with types at
fsharpforfunandprofit.com/ddd



I have a book all about this!

Composition in practice:

Time for some real examples!

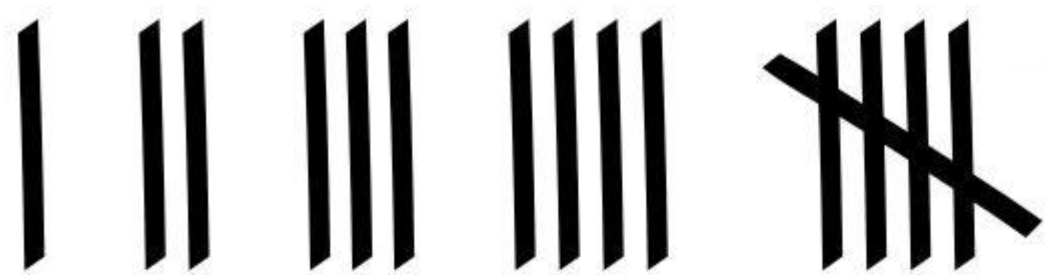
Technique #1

COMPOSITION WITH PIPING
(ROMAN NUMERALS)

To Roman Numerals

- *Task: How to convert an arabic integer to roman numerals?*
- 5 => "V"
- 12 => "XII"
- 107 => "CVII"

To Roman Numerals

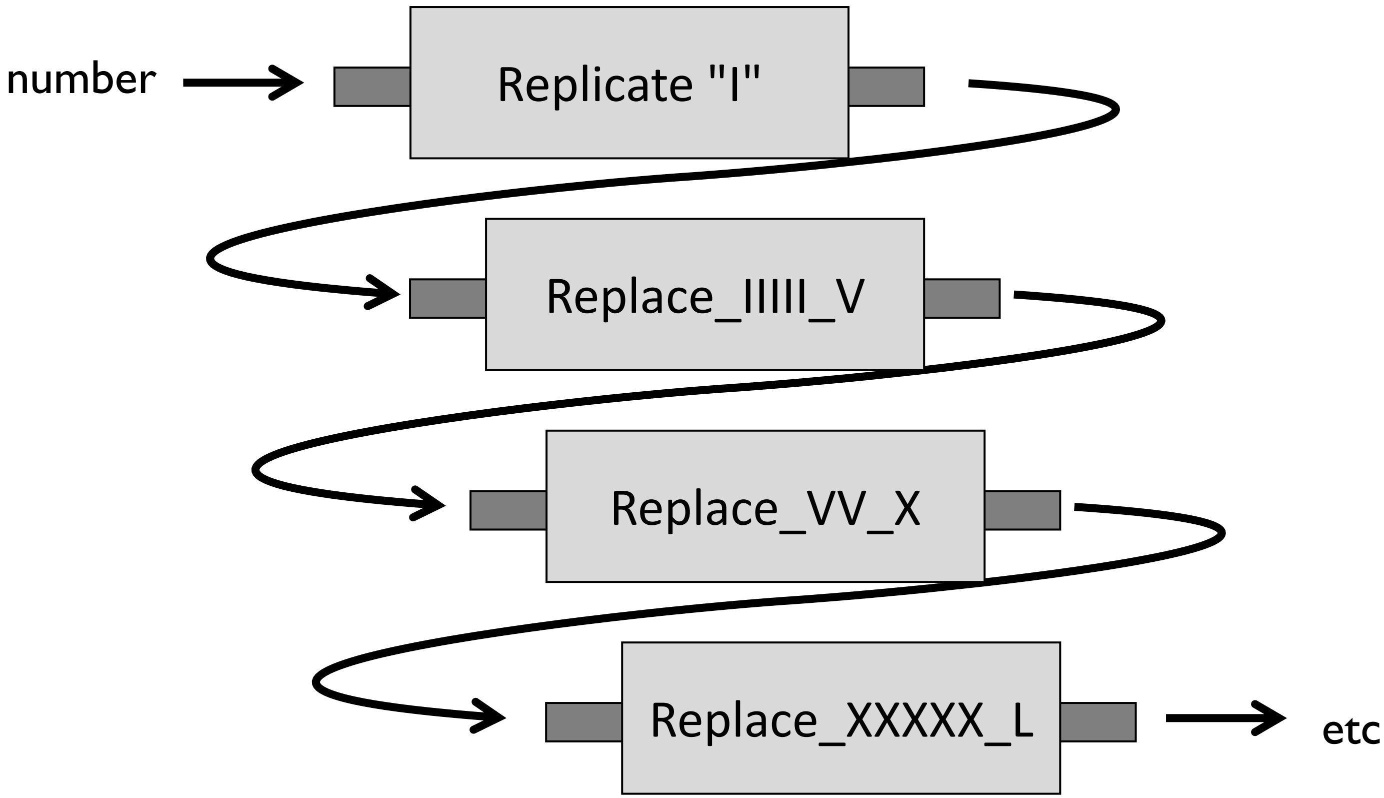


Roman numbers evolved from this

To Roman Numerals

- Use the "tally" approach
  - Start with N copies of "I"
  - Replace five "I"s with a "V"
  - Replace two "V"s with a "X"
  - Replace five "X"s with a "L"
  - Replace two "L"s with a "C"
  - etc

To Roman Numerals



C# example

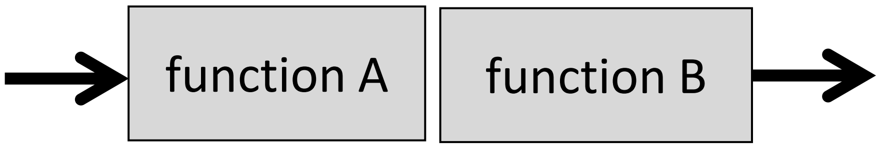
```
string ToRomanNumerals(int number)
{
  // define a helper function for each step
  string replace_IIIII_V(string s) =>
    s.Replace("IIIII", "V");
  string replace_VV_X(string s) =>
    s.Replace("VV", "X");
  string replace_XXXXX_L(string s) =>
    s.Replace("XXXXX", "L");
  string replace_LL_C(string s) =>
    s.Replace("LL", "C");

  // then combine them using piping
  return new string('I', number)
  .Pipe(replace_IIIII_V)
  .Pipe(replace_VV_X)
  .Pipe(replace_XXXXX_L)
  .Pipe(replace_LL_C);
}
```

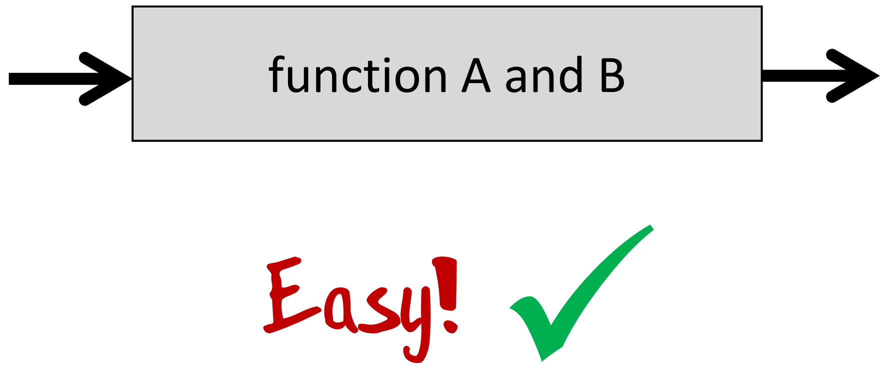
F# example

```
let toRomanNumerals number =
  // define a helper function for each step
  let replace_IIIII_V str =
    replace "IIIII" "V" str
  let replace_VV_X str =
    replace "VV" "X" str
  let replace_XXXXX_L str =
    replace "XXXXX" "L" str
  let replace_LL_C str =
    replace "LL" "C" str

  // then combine them using piping
  String.replicate number "I"
  |> replace_IIIII_V
  |> replace_VV_X
  |> replace_XXXXX_L
  |> replace_LL_C
```

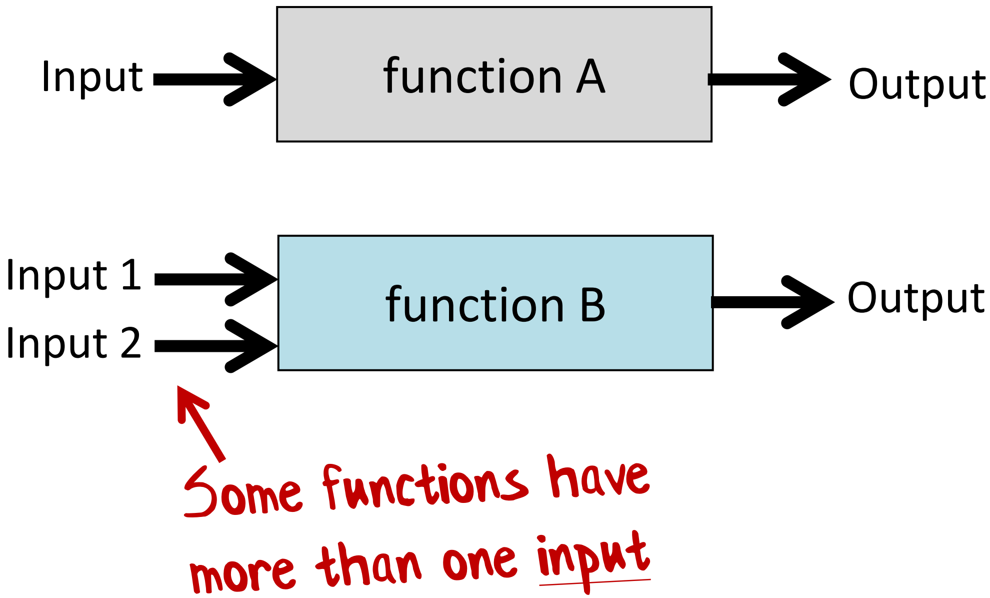
IT'S NOT ALWAYS THIS EASY…

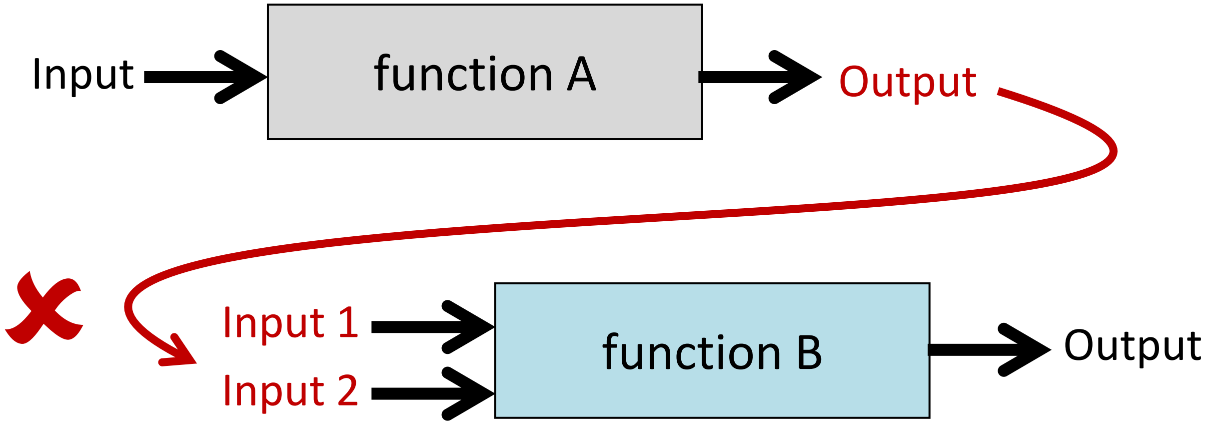

function A
Compose
function B


function A
function B


function A and B
Easy!

… But here is a challenge


Input
function A
Output
Input 1
Input 2
function B
Output
Some functions have
more than one input



Challenge #1: How can we compose these?

Technique #2

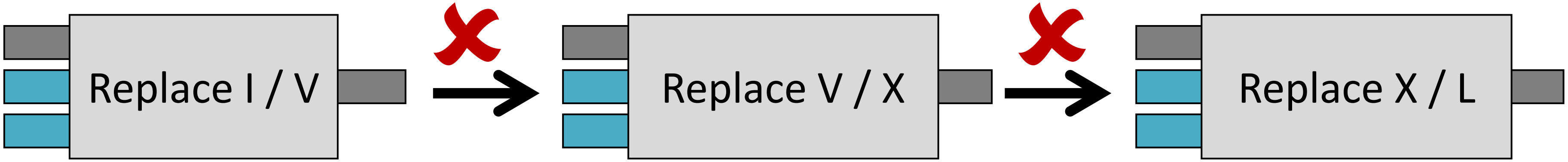
COMPOSITION WITH CURRYING

(ROMAN NUMERALS)

The Replace function

We use this a lot!

Uh-oh! Composition problem



Bad news:

Composition patterns only work for functions that have one parameter! ☹

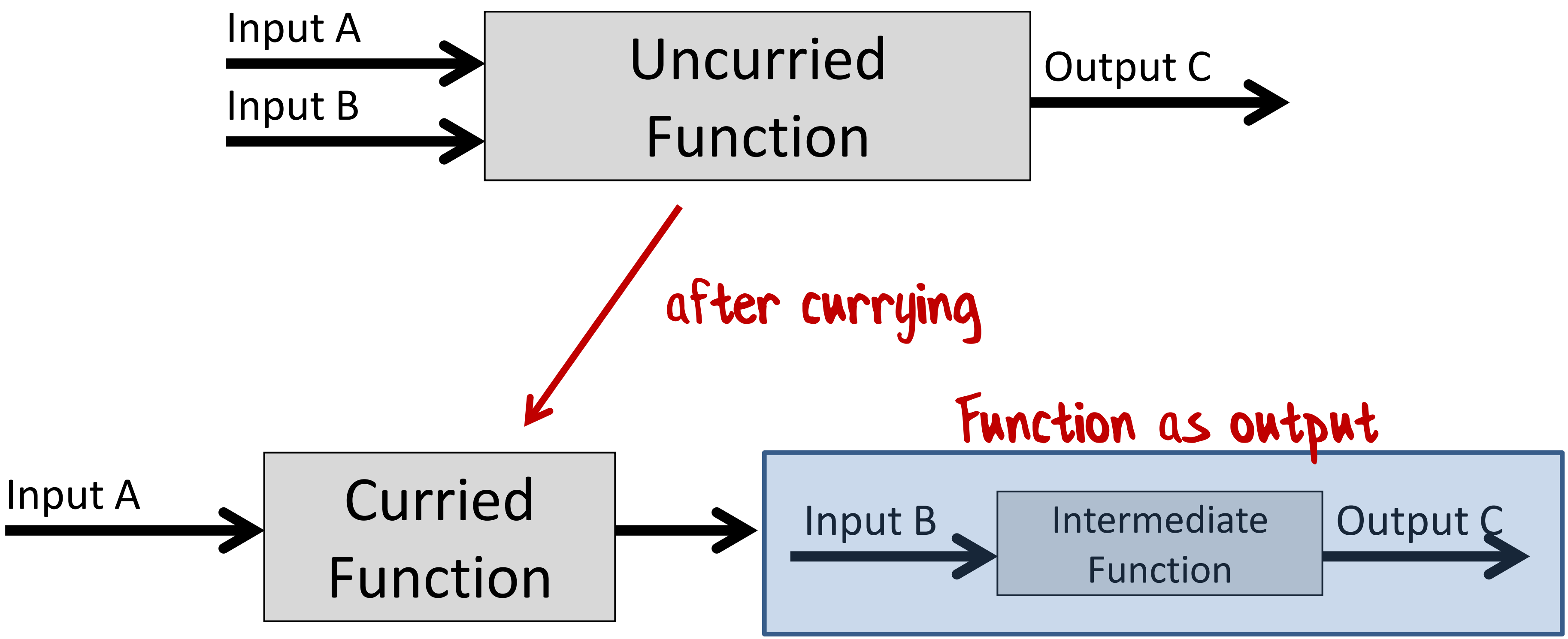
Good news!

Every function can be turned into a one parameter function ☺

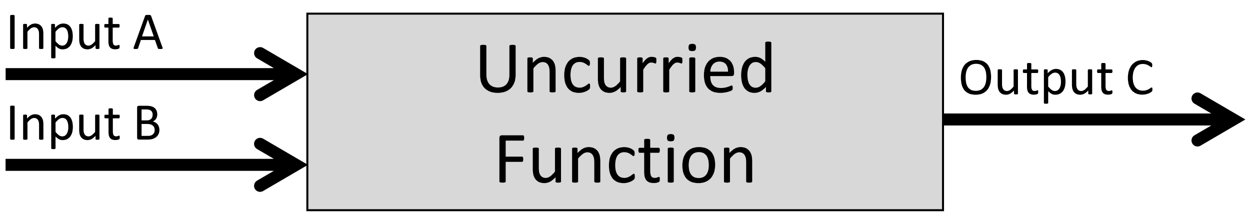
Haskell Curry

We named this technique after him

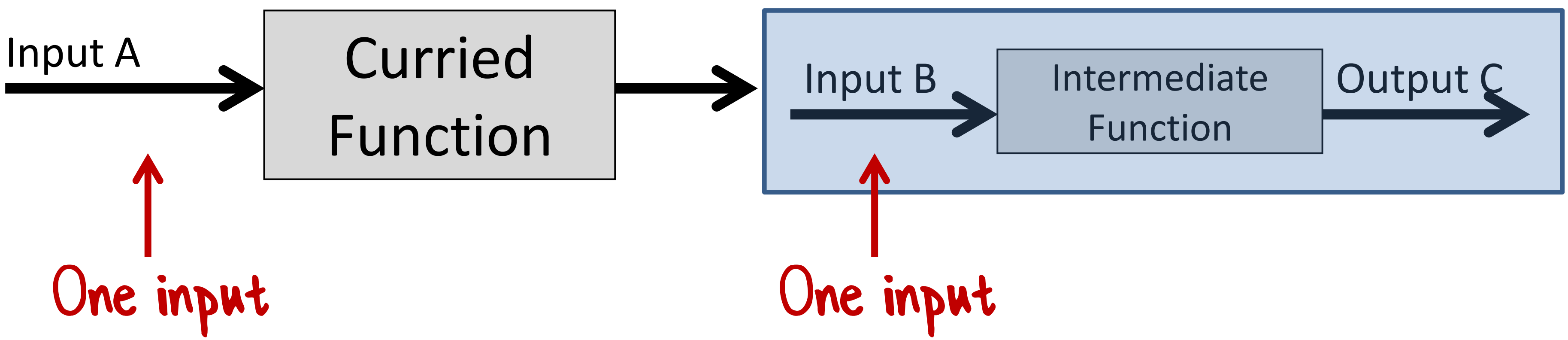
What is currying?



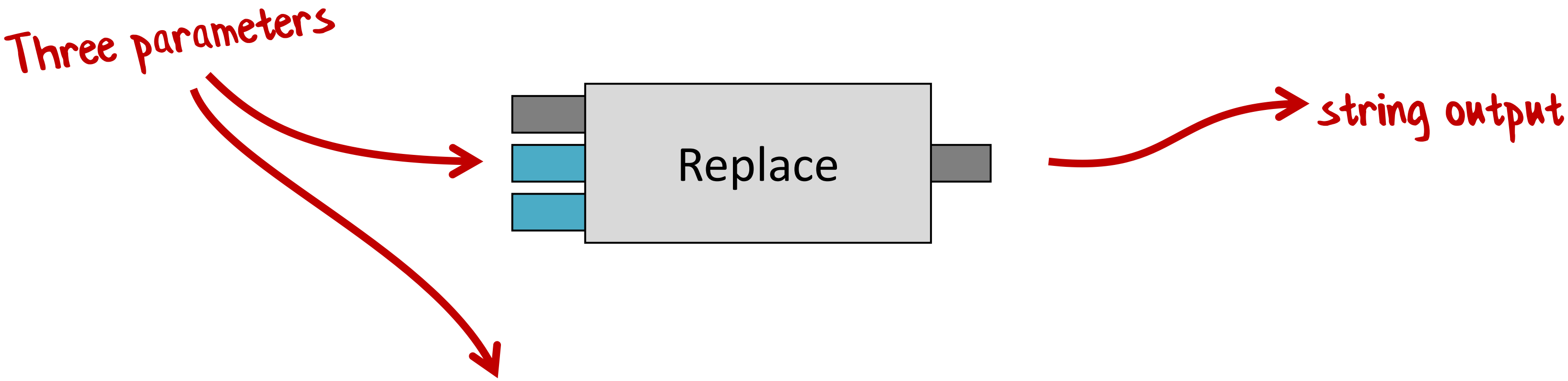
What is currying?



Currying means that *every* function can be converted to a series of one input functions

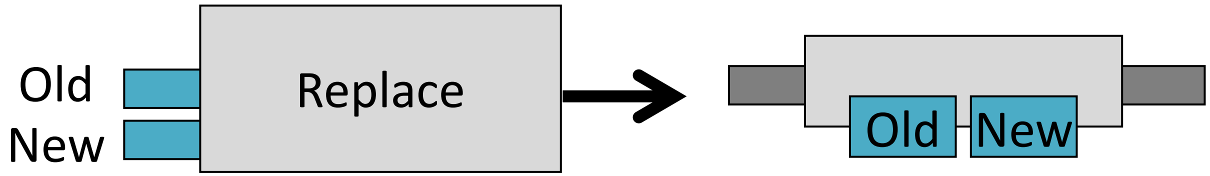


Before currying




```
input.Replace(oldValue, newValue);
```

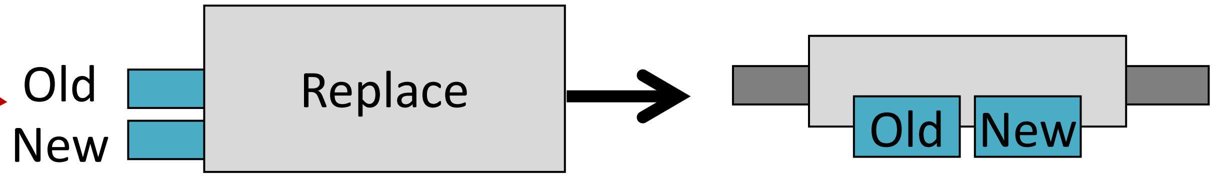
After currying




```
Func<string,string> replace(string oldVal, string newVal) =>
        input => input.Replace(oldVal, newVal);
```

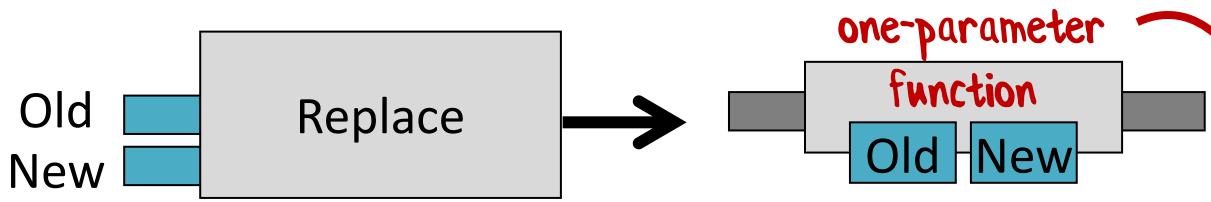
After currying

Two parameters




```
Func<string,string> replace(string oldVal, string newVal) =>
        input => input.Replace(oldVal, newVal);
```

After currying




```
Func<string,string> replace(string oldVal, string newVal) =>
        input => input.Replace(oldVal, newVal);
```

This lambda (function) is returned

C# example

```
string ToRomanNumerals(int number)
{
  // define a general helper function
  Func<string,string> replace(
    string oldValue, string newValue) =>
        input => input.Replace(oldValue, newValue);

  // then use piping
  return new string('I', number)
  .Pipe(replace("IIIII","V"))
  .Pipe(replace("VV","X"))
  .Pipe(replace("XXXXX","L"))
  .Pipe(replace("LL","C"));
}
```

F# example

```
let toRomanNumerals number =
  // no helper function needed.
  // currying occurs automatically in F#

  // combine using piping
  String.replicate number "I"
  |> replace "IIIII" "V"
  |> replace "VV" "X"
  |> replace "XXXXX" "L"
  |> replace "LL" "C"
```

```
let toRomanNumerals number =
  // no helper function needed.
  // currying occurs automatically in F#

  // combine using piping
  String.replicate number "I"
  |> replace "IIIII" "V"
  |> replace "VV" "X"
  |> replace "XXXXX" "L"
  |> replace "LL" "C"
```

Only 2 of the 3 parameters are passed ?

Partial Application

Very important technique!

Partial Application

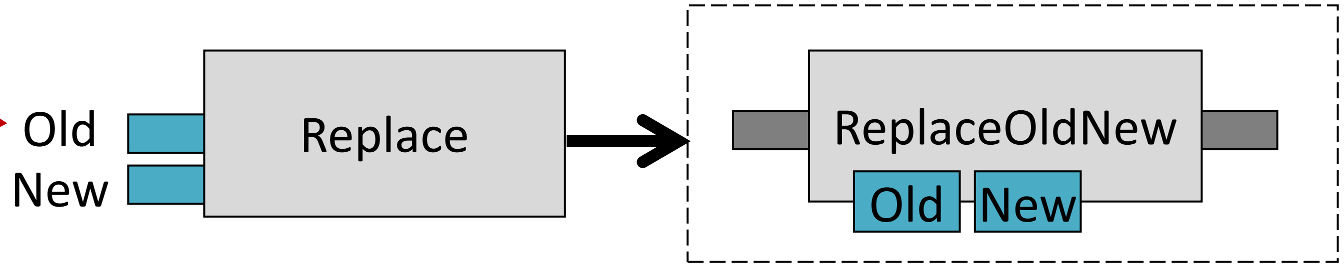
```
let add x y = x + y
let multiply x y = x * y



5
|> add 2
|> multiply 3
```

partial application

Piping provides the missing argument

Partial Application

Pass in 2 of the 3 inputs




```
String.replicate number "I"
   |> replace "IIIII" "V"
   |> replace "VV" "X"
   |> replace "XXXXX" "L"
   |> replace "LL" "C"
```

Only 2 parameters passed in

Piping provides the missing argument

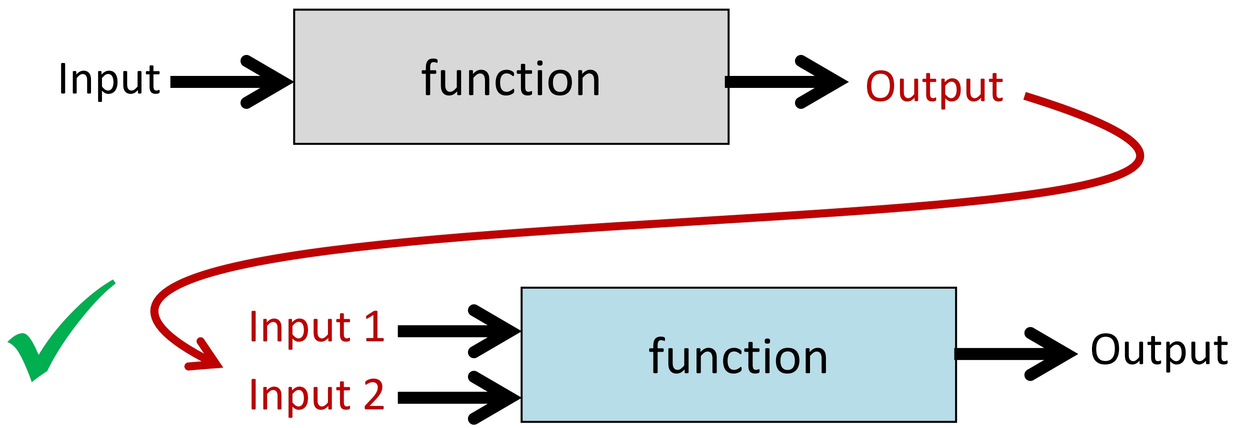
Pipelines
are extensible

Composable => extensible

```
let toRomanNumerals number =
  String.replicate number "I"
   |> replace "IIIII" "V"
   |> replace "VV" "X"
   |> replace "XXXXX" "L"
   |> replace "LL" "C"
  // can easily add new segments to the pipeline
   |> replace "VIIII" "IX"
   |> replace "IIII"  "IV"
   |> replace "LXXXX" "XC"
```

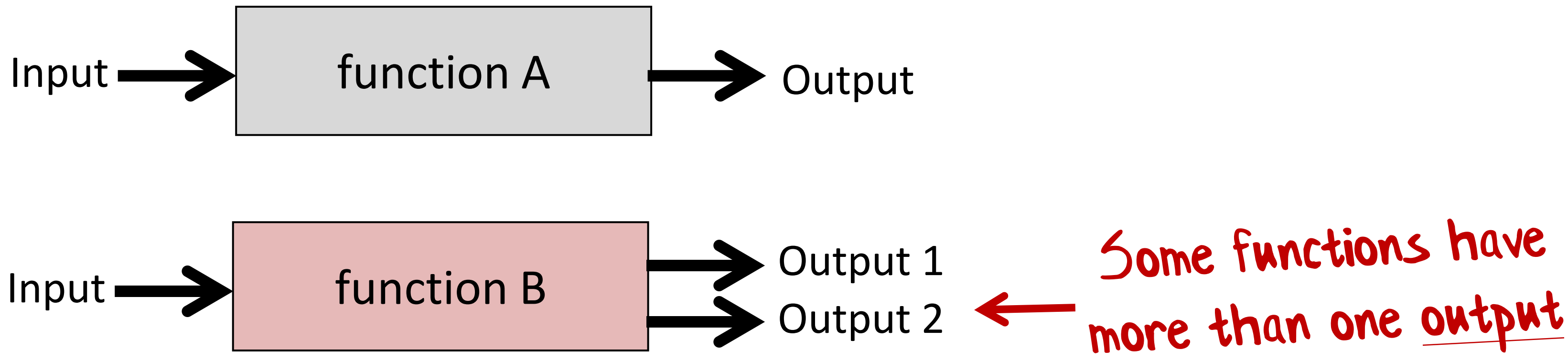
Can add new functionality
without touching existing code!

Challenge #1: How can we compose these?

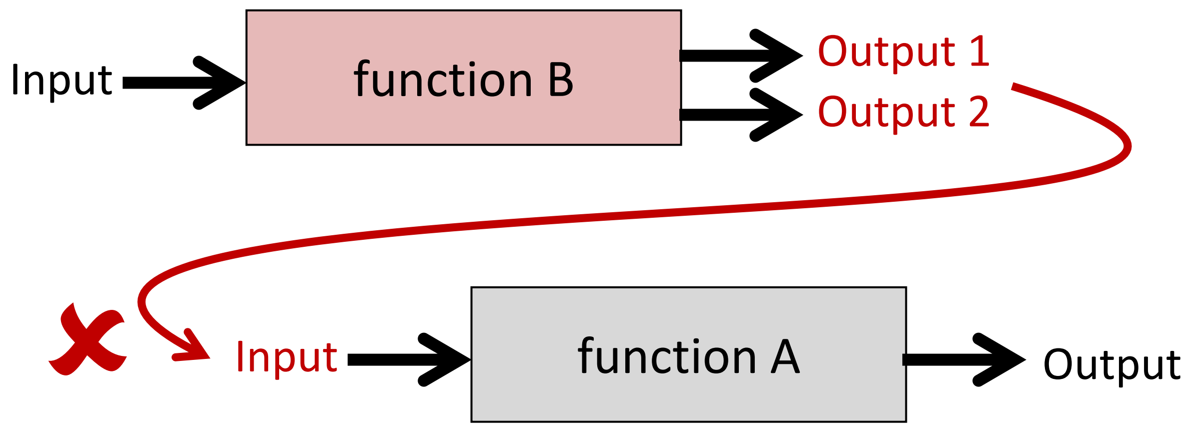


Solved with currying and partial application!

Here is another challenge


Input
function A
Output
Input
function B
Output 1
Output 2
Some functions have more than one output

Challenge #2: How can we compose these?



Technique #3

COMPOSITION WITH BIND
(FIZZBUZZ)

FizzBuzz definition

- Write a program that prints the numbers from 1 to 100
- But:
  - For multiples of three print "Fizz" instead
  - For multiples of five print "Buzz" instead
  - For multiples of both three and five print "FizzBuzz" instead.

A simple F# implementation

```
let fizzBuzz max =
  for n in [1..max] do
    if (isDivisibleBy n 15) then
      printfn "FizzBuzz"
    else if (isDivisibleBy n 3) then
      printfn "Fizz"
    else if (isDivisibleBy n 5) then
      printfn "Buzz"
    else
      printfn "%i" n

let isDivisibleBy n divisor =
  (n % divisor) = 0  // helper function
```

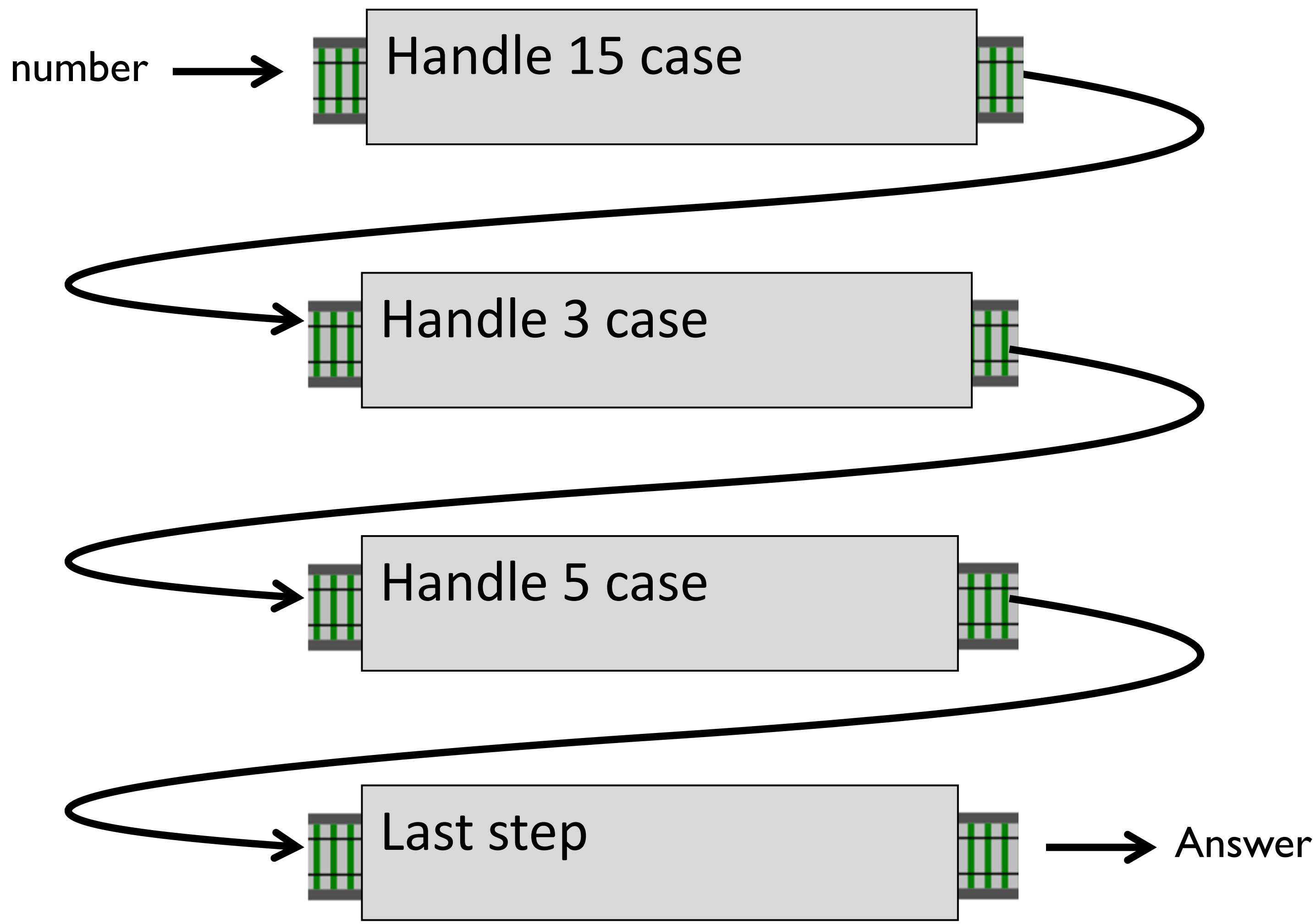
A simple F# implementation

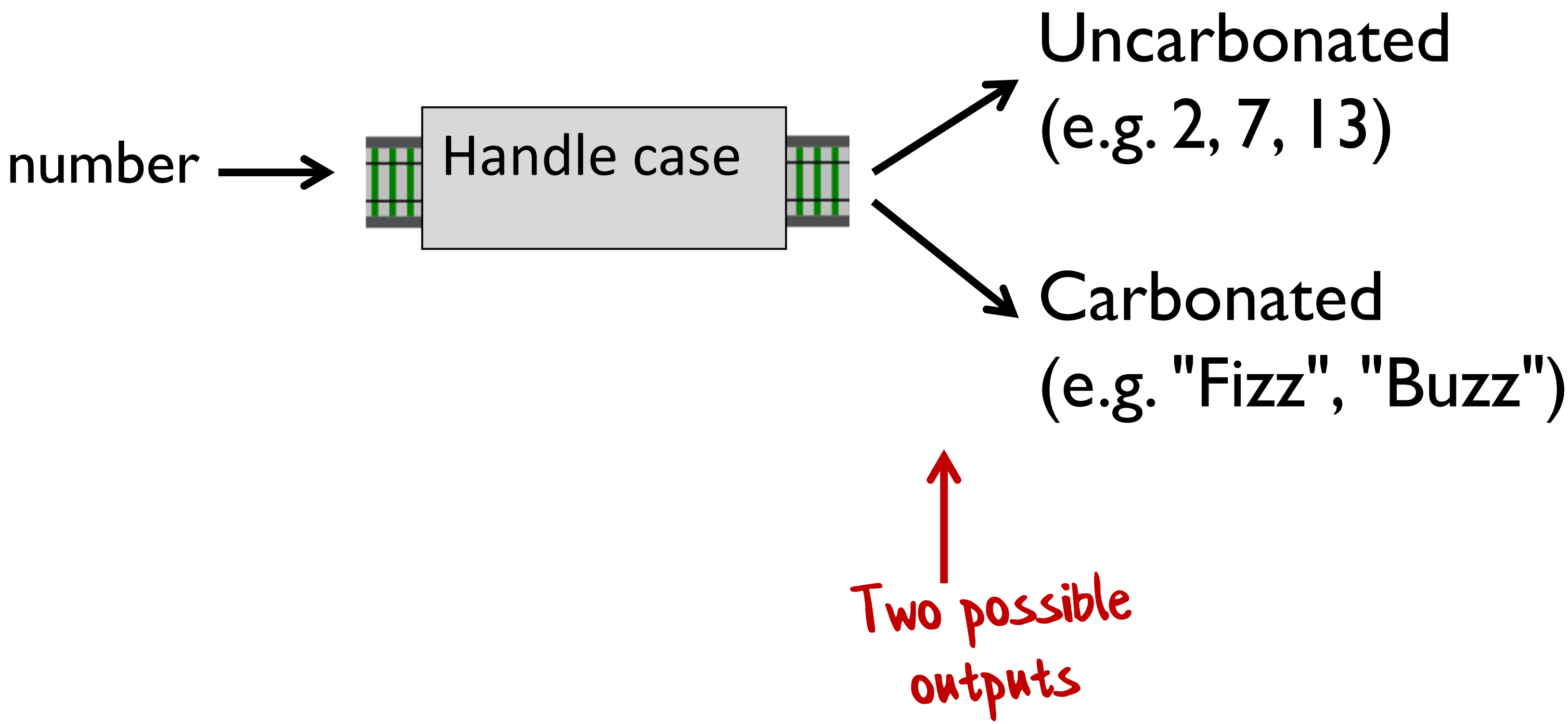
```
let fizzBuzz max =
  for n in [1.     x] do
    if (isDivisi    By n 15) th
      printfn "Fi     zz"
    else if (isDivi     eBy     ) then
      printfn "Fizz"
    else if (isDivisib     n 5) then
      printfn "Buzz"
    else
      printfn "%i"

let isDivisibl     n divisor =
  (n % divis     = 0  // helper fun     n
```

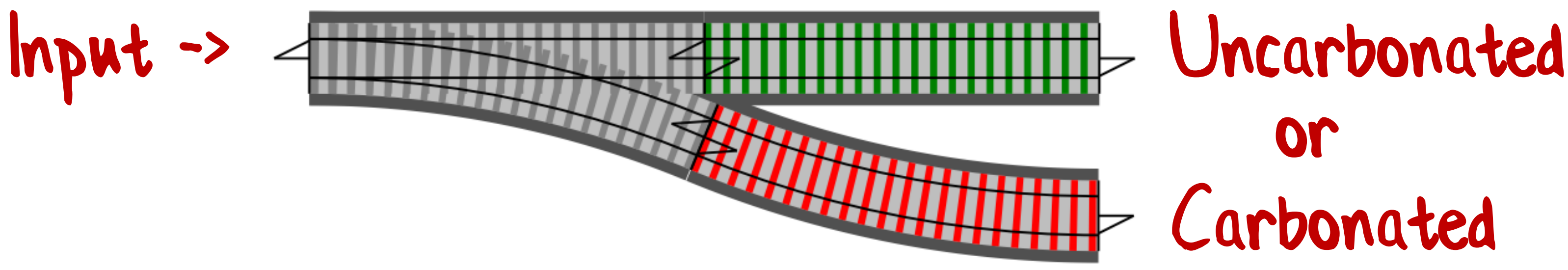
Too easy!

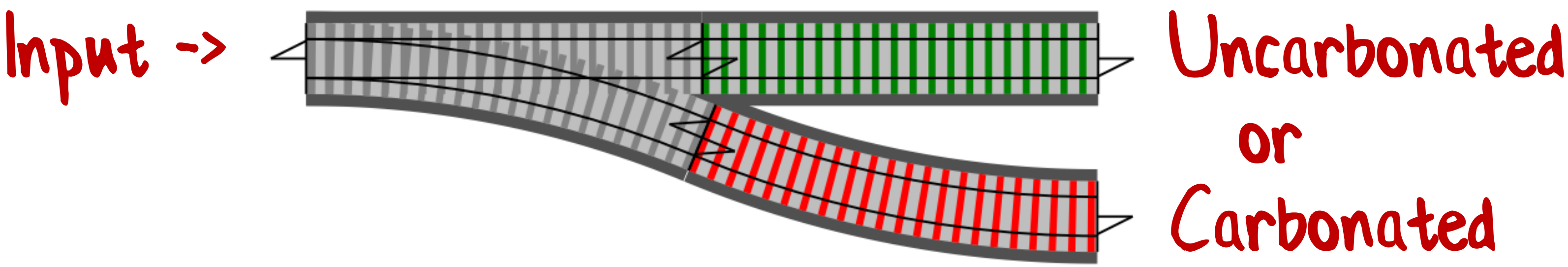
Also, not composable!

Pipeline implementation




number
Handle case
Uncarbonated
(e.g. 2, 7, 13)
Carbonated
(e.g. "Fizz", "Buzz")
Two possible
outputs


Input ->
Uncarbonated
or
Carbonated




```
type CarbonationResult =
  | Uncarbonated of int    // unprocessed
  | Carbonated of string  // "Fizz", Buzz", etc
```

Choice type

Idea from http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html

```
type CarbonationResult =
  | Uncarbonated of int    // unprocessed
  | Carbonated of string  // "Fizz", Buzz", etc


let carbonate divisor label n =
  if (isDivisibleBy n divisor) then
    Carbonated label
  else
    Uncarbonated n
```

return one case

return the other case

Idea from http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html

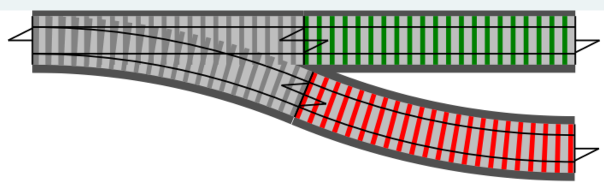
```
type CarbonationResult =
  | Uncarbonated of int   // unprocessed
  | Carbonated of string  // "Fizz", Buzz", etc

let carbonate divisor label n =
  if (isDivisibleBy n divisor) then
    Carbonated label
  else
    Uncarbonated n
```

Three parameters

```
12 |> carbonate 3 "Fizz"  // Carbonated "Fizz"

10 |> carbonate 3 "Fizz"  // Uncarbonated 10

10 |> carbonate 5 "Buzz"  // Carbonated "Buzz"
```

Two parameters!

carbonate 5 "Buzz"



First implementation attempt

```
let fizzbuzz n =
  let result15 = n |> carbonate 15 "FizzBuzz"
  match result15 with
    | Carbonated str ->
      str
    | Uncarbonated n ->
      let result3 = n |> carbonate 3 "Fizz"
      match result3 with
        | Carbonated str ->
          str
        | Uncarbonated n ->
          let result5 = n |> carbonate 5 "Buzz"
          match result5 with
            | Carbonated str ->
              str
            | Uncarbonated n ->
              string n     // convert to string
```

Really ugly code...

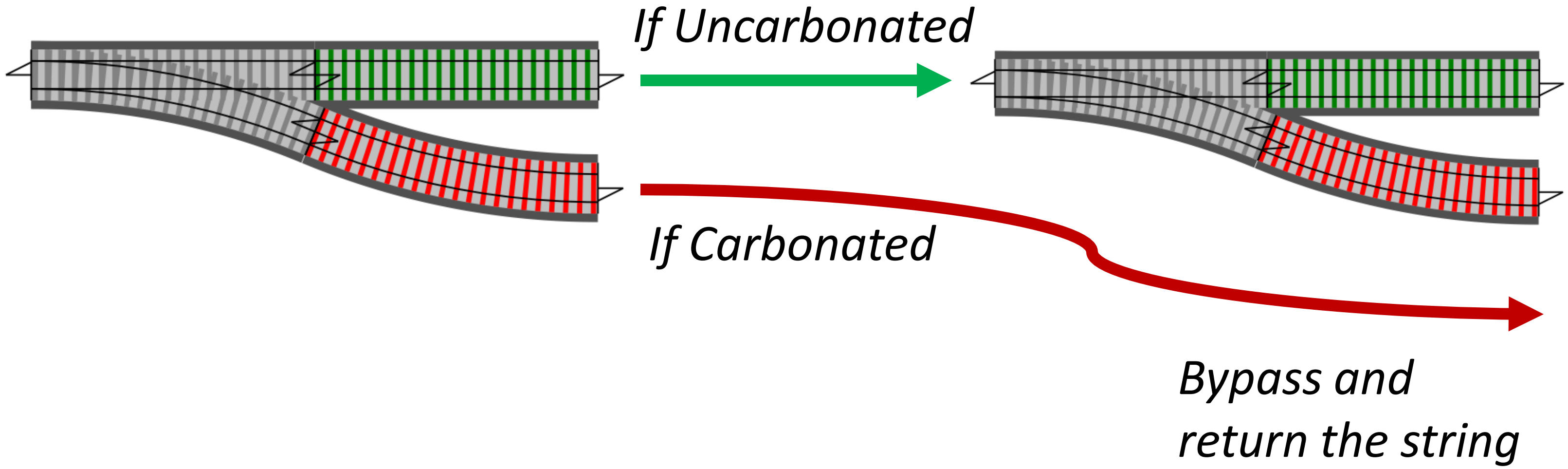
But wait – there's a pattern...

```
let fizzbuzz n =
  let result15 = n |> carbonate 15 "FizzBuzz"
  match result15 with
    | Carbonated str ->
      str
    | Uncarbonated n ->
      let result3 = n |> carbonate 3 "Fizz"
      match result3 with
        | Carbonated str ->
          str
        | Uncarbonated n ->
          let result5 = n |> carbonate 5 "Buzz"
          match result5 with
            | Carbonated str ->
              str
            | Uncarbonated n ->
              // do something with Uncarbonated value
```

```
let fizzbuzz n =
  let result15 = n |> carbonate 15 "FizzBuzz"
  match result15 with
    | Carbonated str ->
      str
    | Uncarbonated n ->
      let result3 = n |> carbonate 3 "Fizz"
      match result3 with
        | Carbonated str ->
          str
        | Uncarbonated n ->
          // do something with Uncarbonated value
          // ...
          // ...
```

```
let fizzbuzz n =
  let result15 = n |> carbonate 15 "FizzBuzz"
  match result15 with
    | Carbonated str ->
      str
    | Uncarbonated n ->
      // do something with Uncarbonated value
      // ...
      // ...
```

```
if Carbonated then
    // return the string
if Uncarbonated then
    // do something with the number
```

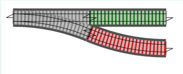

If Uncarbonated
If Carbonated
Bypass and
return the string

Parameterize all the things!

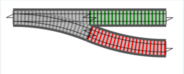
```
let ifUncarbonatedDo f result =
  match result with
  | Carbonated str ->
      Carbonated str
  | Uncarbonated n ->
     f n
```

```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> lastStep
```

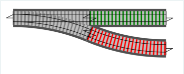
```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> lastStep
```



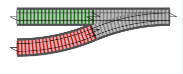
```
let fizzbuzz n =
  n
   |> carbonate 15 "FizzBuzz"
   |> ifUncarbonatedDo (carbonate 3 "Fizz")
   |> ifUncarbonatedDo (carbonate 5 "Buzz")
   |> lastStep
```



```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> lastStep
```



```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> lastStep
```



```
let lastStep result =
  match result with
  | Carbonated str ->
     str
  | Uncarbonated n ->
     string(n)  // still not fizzy, so
                // convert to string
```

Composable => easy to extend

```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> lastStep
```

```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> ifUncarbonatedDo (carbonate 7 "Baz")
  |> lastStep
```

Composable => easy to extend

Composable => easy to extend

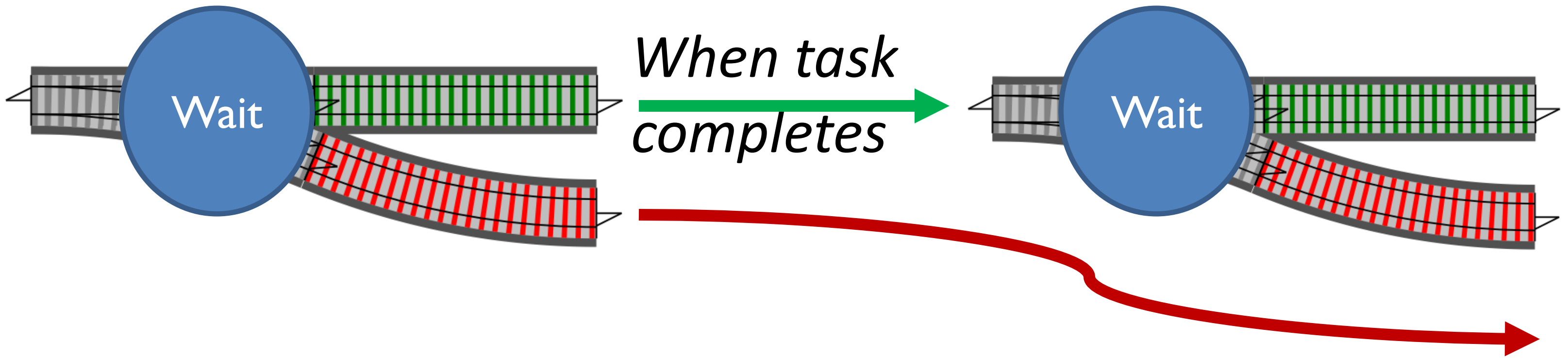
```
let fizzbuzz n =
  n
   |> carbonate 15 "FizzBuzz"
   |> ifUncarbonatedDo (carbonate 3 "Fizz")
   |> ifUncarbonatedDo (carbonate 5 "Buzz")
   |> ifUncarbonatedDo (carbonate 7 "Baz")
   |> ifUncarbonatedDo (carbonate 11 "Pozz")
   |> lastStep
```

Composable => easy to extend

```
let fizzbuzz n =
  n
  |> carbonate 15 "FizzBuzz"
  |> ifUncarbonatedDo (carbonate 3 "Fizz")
  |> ifUncarbonatedDo (carbonate 5 "Buzz")
  |> ifUncarbonatedDo (carbonate 7 "Baz")
  |> ifUncarbonatedDo (carbonate 11 "Pozz")
  |> ifUncarbonatedDo (carbonate 13 "Tazz")
  |> lastStep
```

Not touching existing code means more confidence that you haven't broken anything!

Another example: Chaining tasks



a.k.a "promise", "future"

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                etc
```

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                do something
```

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
          do something
```

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
      do something
```

Parameterize the next step

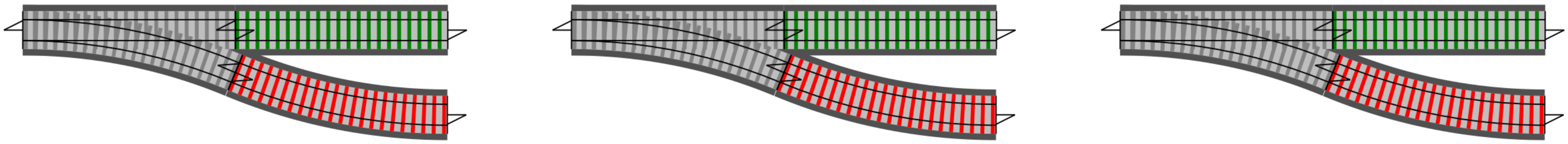
```
let whenFinishedDo f task =
    task.WhenFinished (fun taskResult ->
        f taskResult)
```

```
let taskExample input =
    startTask input
    |> whenFinishedDo startAnotherTask
    |> whenFinishedDo startThirdTask
    |> whenFinishedDo ...
```

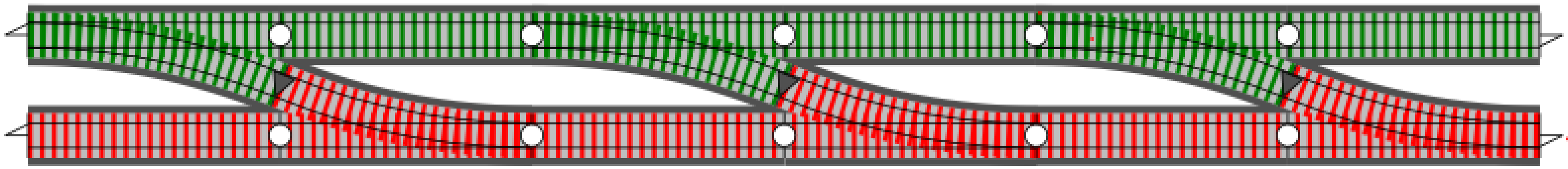
MONADS!

Is there a general solution to handling functions like this?

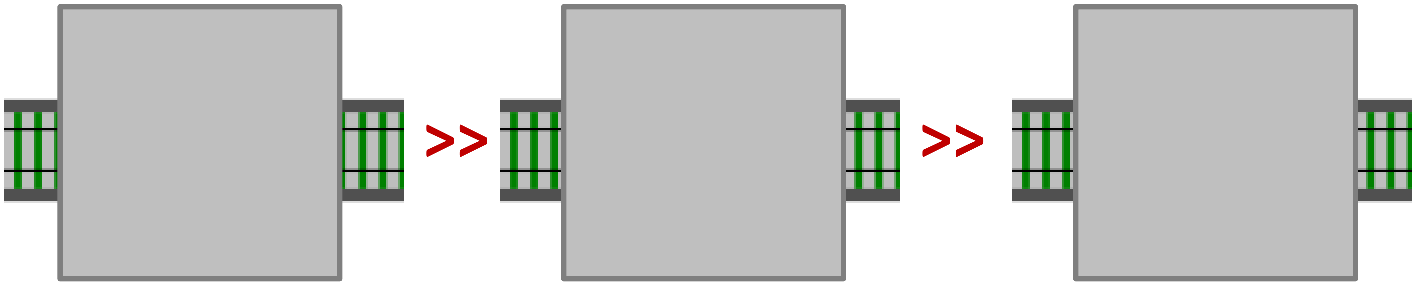
Yes! “Bind” is the answer!
Bind all the things!



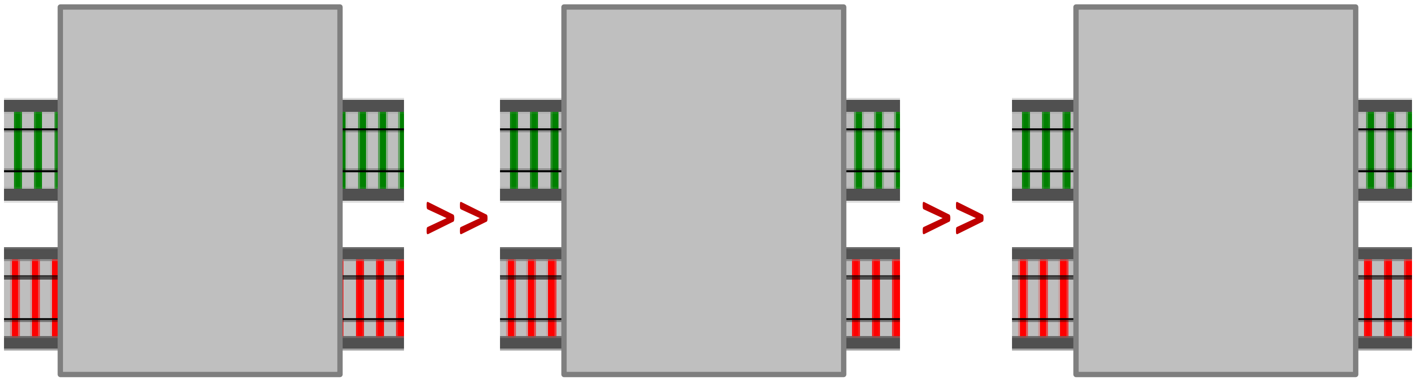
How do we compose these?



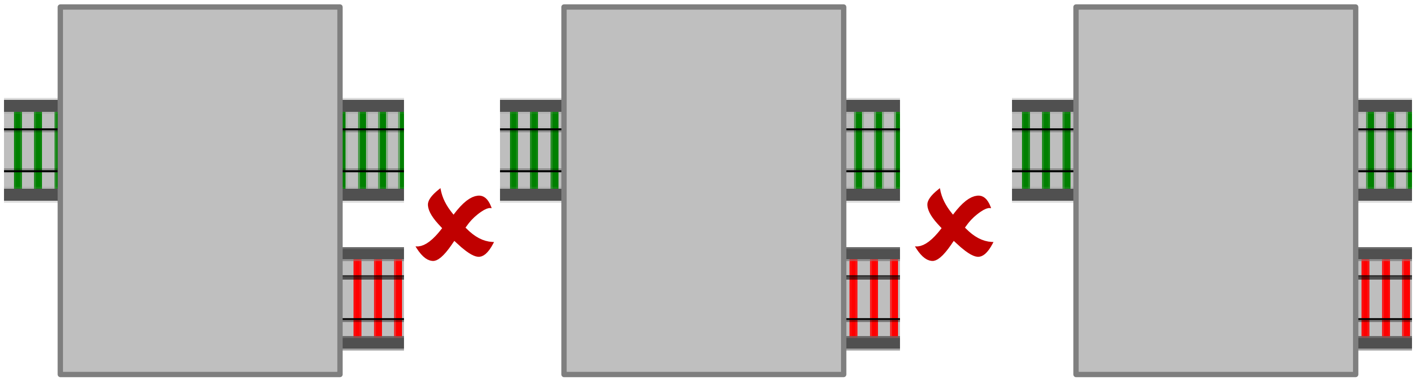
This is the "two track" model –
a.k.a "Railway Oriented Programming".
See fsharpforfunandprofit.com/rop



Composing one-track functions is fine...

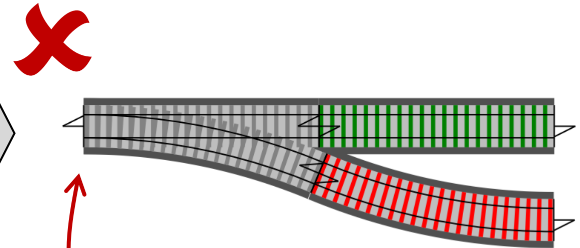


... and composing two-track functions is fine...



... but composing points/switches is not allowed!

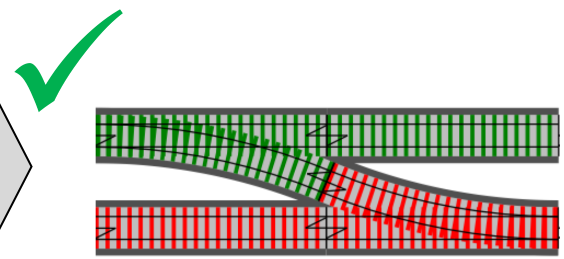
One-track input



Two-track output

Before: Not suitable for composition

Two-track input

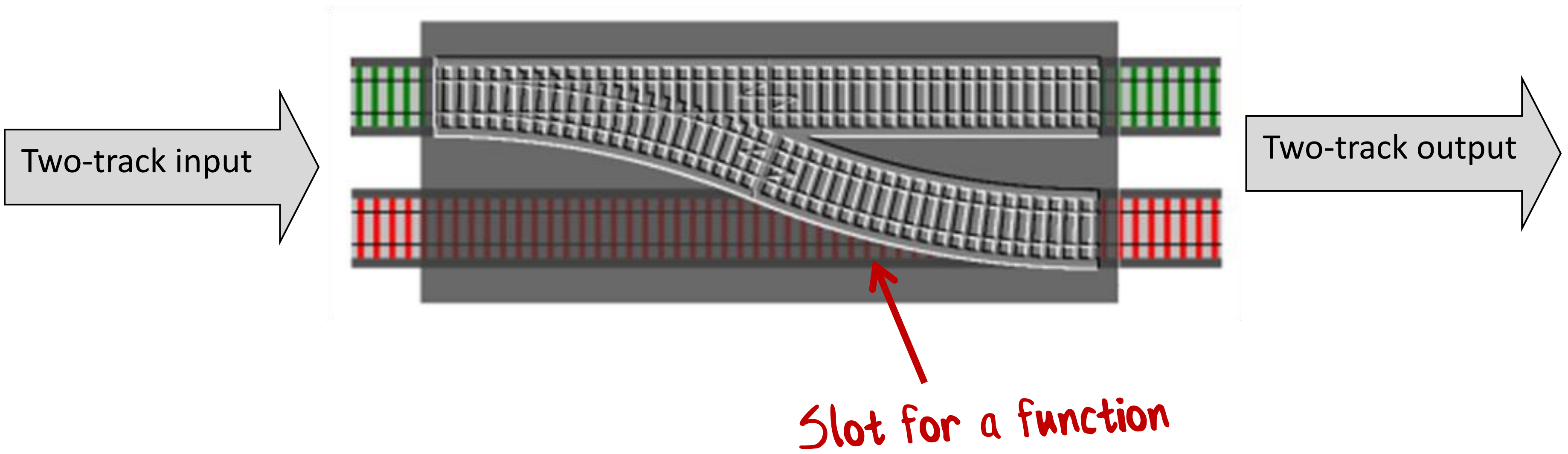


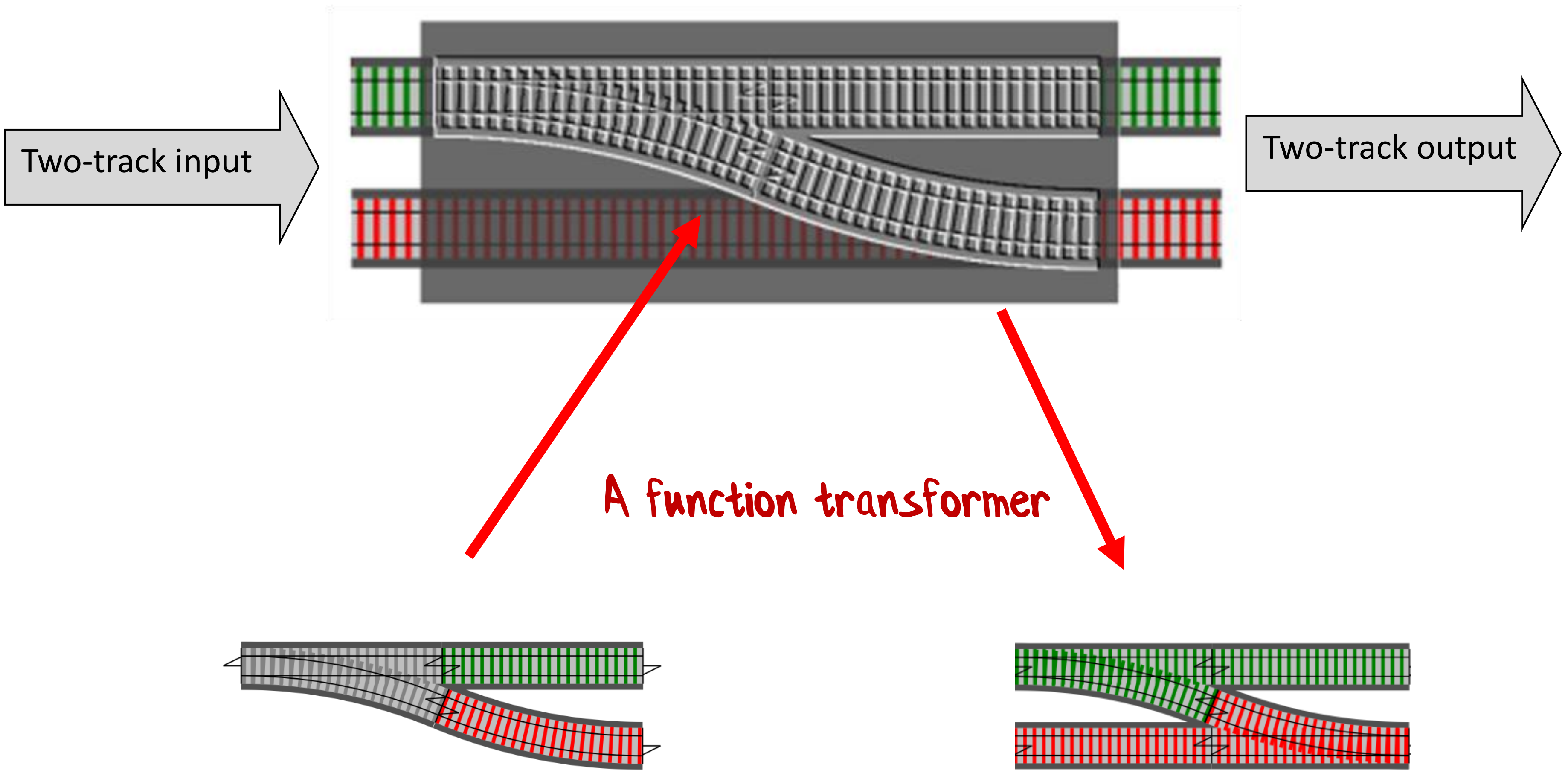
Two-track output

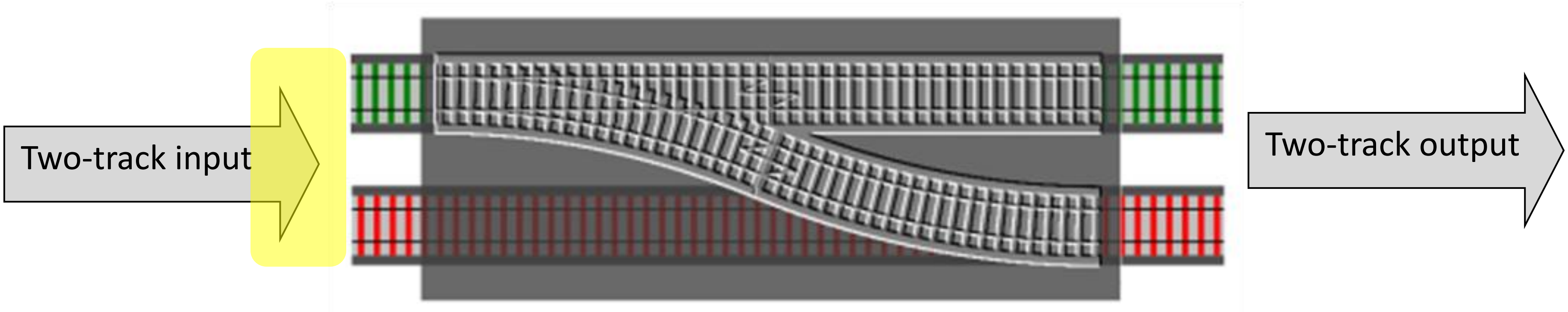
After: Suitable for composition

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

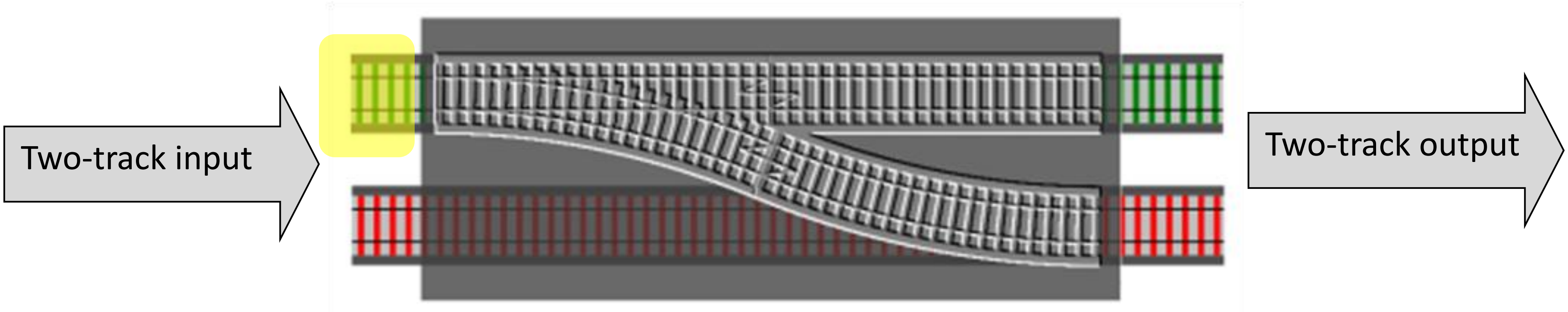
The "bind" adapter block



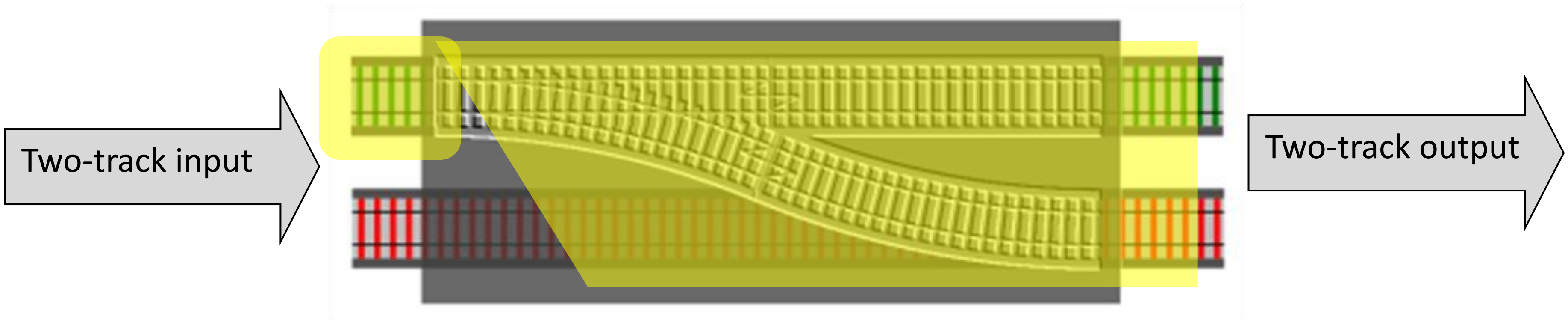

Two-track input
Two-track output
A function transformer



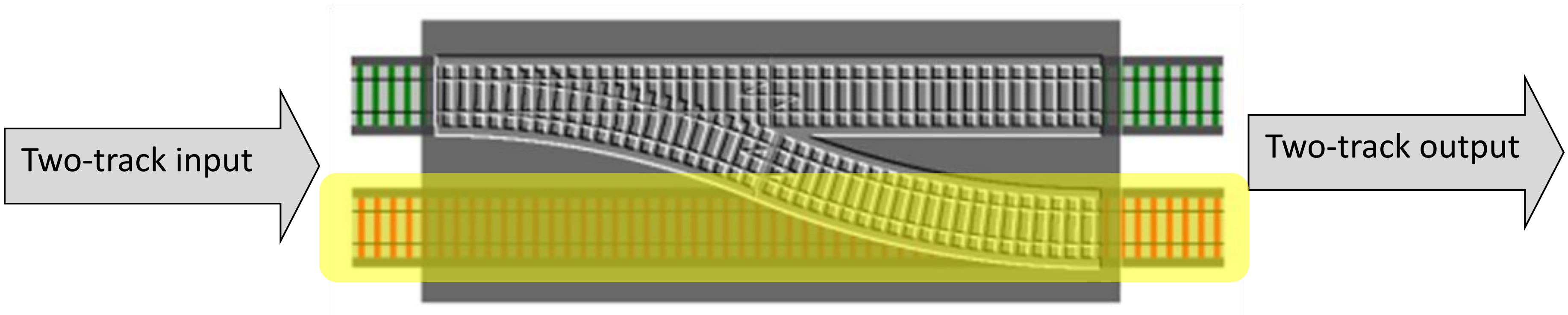

```
let bind nextFunction result =
  match result with
  | Uncarbonated n ->
      nextFunction n
  | Carbonated str ->
      Carbonated str
```



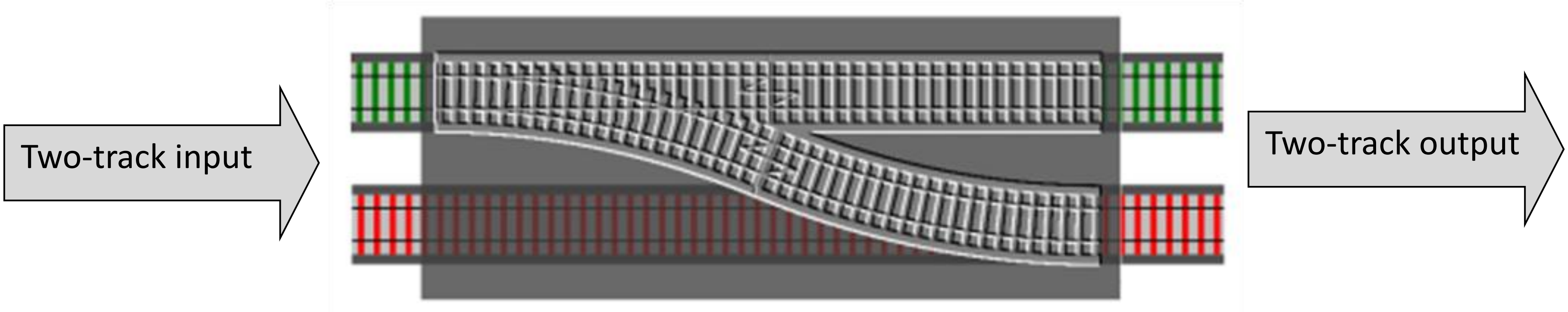

```
let bind nextFunction result =
  match result with
  | Uncarbonated n ->
      nextFunction n
  | Carbonated str ->
      Carbonated str
```




```
let bind nextFunction result =
  match result with
  | Uncarbonated n ->
      nextFunction n
  | Carbonated str ->
      Carbonated str
```




```
let bind nextFunction result =
  match result with
  | Uncarbonated n ->
      nextFunction n
  | Carbonated str ->
      Carbonated str
```




```
let bind nextFunction result =
  match result with
   | Uncarbonated n ->
       nextFunction n
   | Carbonated str ->
       Carbonated str
```

"ifUncarbonatedDo"

FP terminology

- A **monad** is
  - A data type
  - With an associated **"bind"** function
  - (and some other stuff)

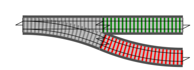
"CarbonationResult"

```
type CarbonationResult =
   | Uncarbonated of int
   | Carbonated of string
```

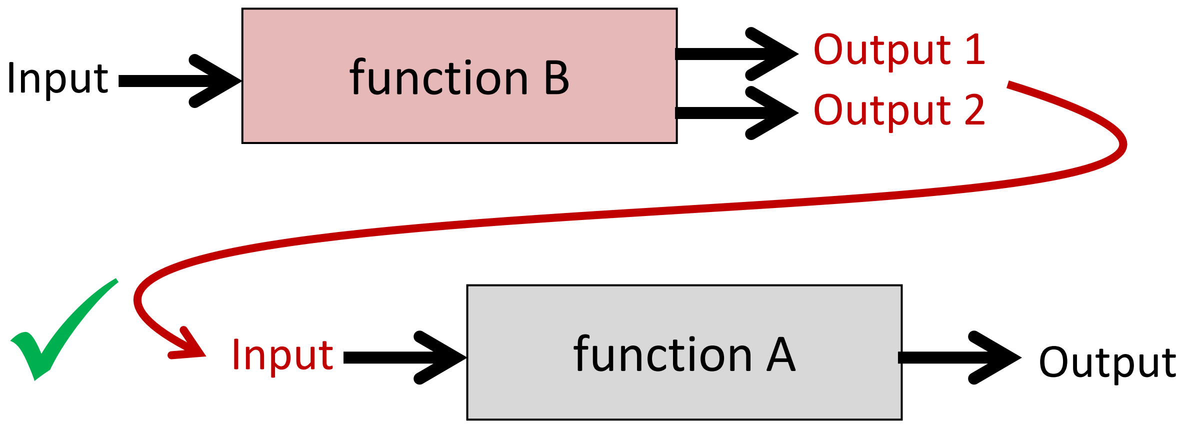
"ifUncarbonatedDo"

- A **monadic function** is
  - A switch/points function
  - "bind" is used to compose them

"carbonate"



Challenge #2: How can we compose these?



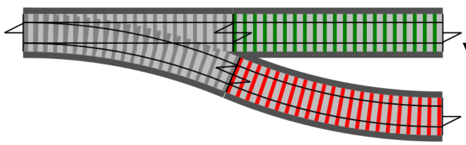
Solved with monads!

Technique #4

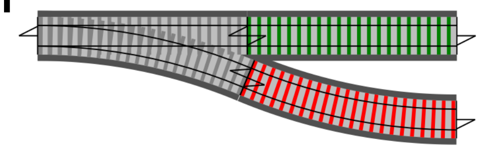
KLEISLI COMPOSITION

(WEB SERVICE)

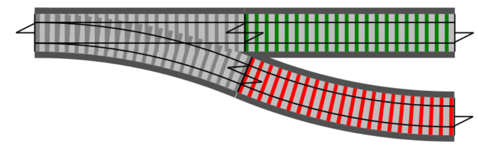
Kleisli Composition



compose
with



=

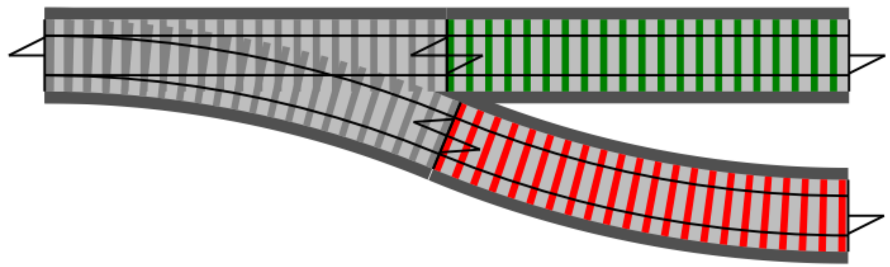


The result is the
same kind of thing

A HttpHandler "WebPart"

HttpContext

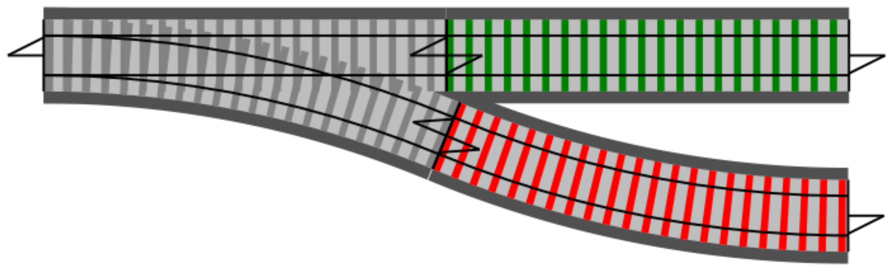
Async<HttpContext option>



Might succeed

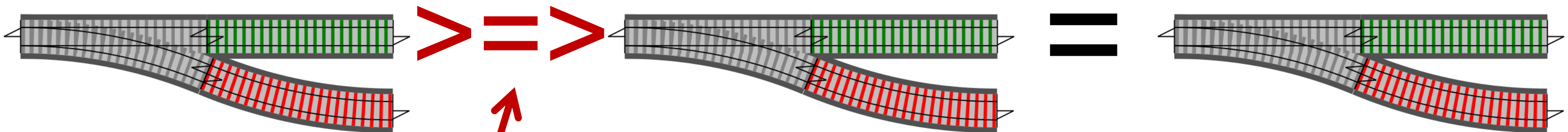
Contains request, response, etc

A HttpHandler "WebPart"



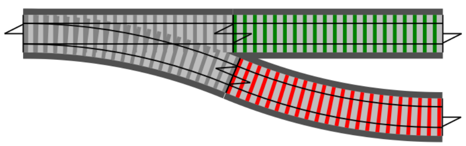
Search for "F# Giraffe" for more

Composition of HttpHandlers



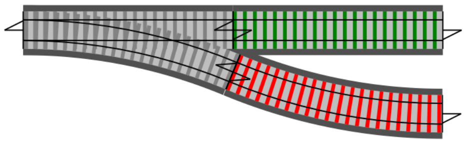
Kleisli composition symbol

The result is another HttpHandler so you can keep adding and adding



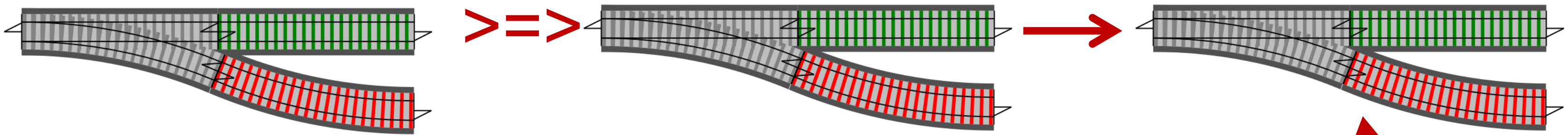

```
path "/hello"
```

Checks request path
(might fail)




```
OK "Hello"
```

Sets response



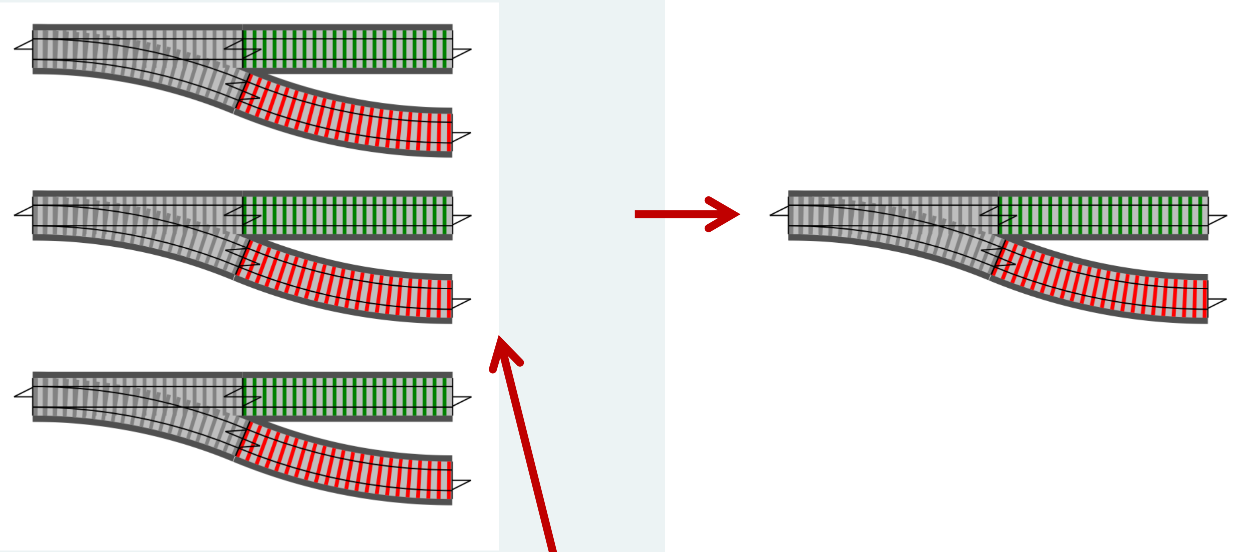

```
path "/hello" >=> OK "Hello"
```

A new WebPart

Checks request path (might fail)

Sets response

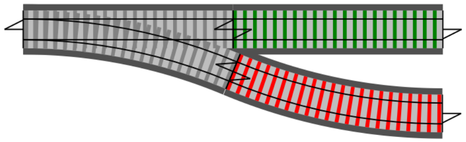
```
choose [

       ]
```



Picks first HttpHandler that succeeds

Pick first path that succeeds

```
choose [
  path "/hello" >=> OK "Hello"
  path "/goodbye" >=> OK "Goodbye"
  ]
```



```
GET
```

Only succeeds if
request is a GET

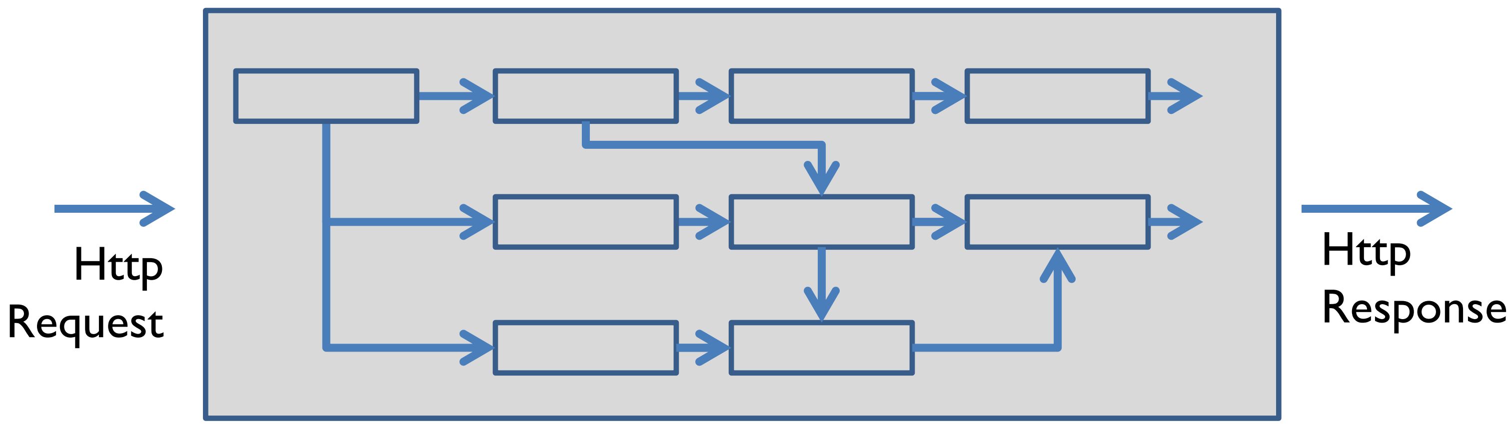
```
GET >=> choose [
  path "/hello" >=> OK "Hello"
  path "/goodbye" >=> OK "Goodbye"
  ]
```

A complete web app

```
let app = choose [
  GET >=> choose [
    path "/hello" >=> OK "Hello"
    path "/goodbye" >=> OK "Goodbye"
    ]
  POST >=> choose [
    path "/hello" >=> OK "Hello POST"
    path "/goodbye" >=> OK "Goodbye POST"
    ]
]

startWebServer defaultConfig app
```

HttpHandlers are composable, reusable, testable, etc.

The Power of Composition



No classes, no inheritance, one-directional data flow!

Review

- The philosophy of composition
  - Connectable, reusable parts
- FP principles:
  - Composable functions
  - Composable types

Review

A taste of various composition techniques:

- Piping with "|>"
- Currying/partial application
- Composition using "bind" (monads!)
- Kleisli composition using ">=>"

Don't worry about understanding it all, but hopefully it's not so scary now!

Why bother? 

Benefits of composition:

- Reusable – no strings attached
- Understandable – data flows in one direction
- Testable – parts can be tested in isolation
- Maintainable – all dependencies are explicit
- Extendable – can add new parts without touching old code
- Different way of thinking – it's good for your brain to learn new things!

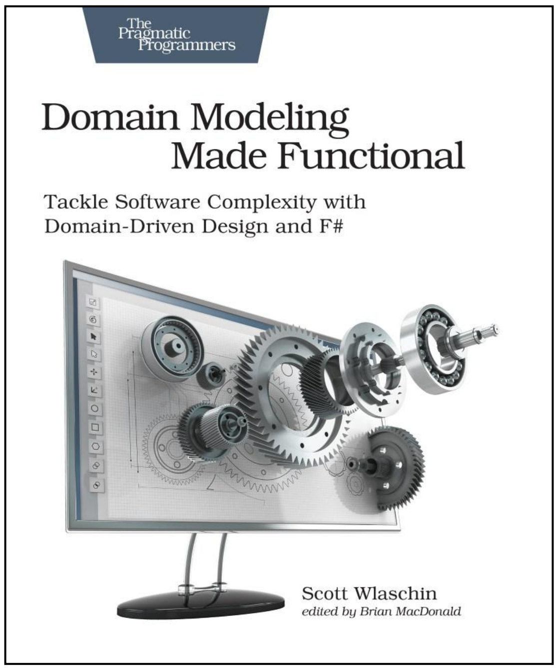
Thank you, DotNext!

fsharpforfunandprofit.com/composition

Slides and video here

@ScottWlaschin

Me on twitter



My book

Ask me anything about railways!