

THE PRINCIPLES OF FUNCTIONAL PROGRAMMING

Core principles of FP

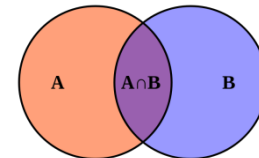
Functions are things



Composition everywhere



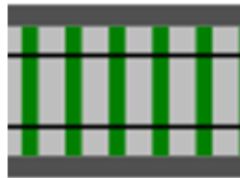
Types are not classes



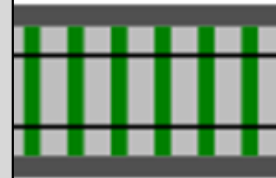
Core FP principle:
Functions are things



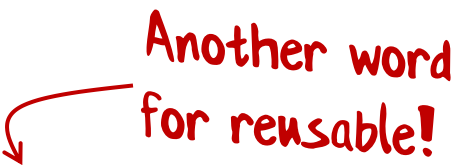
Functions as things



Function
apple -> banana



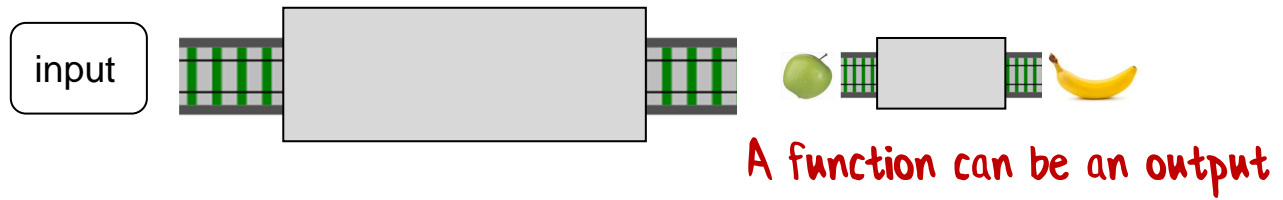
A function is a thing which
transforms inputs to outputs

Another word
for reusable!

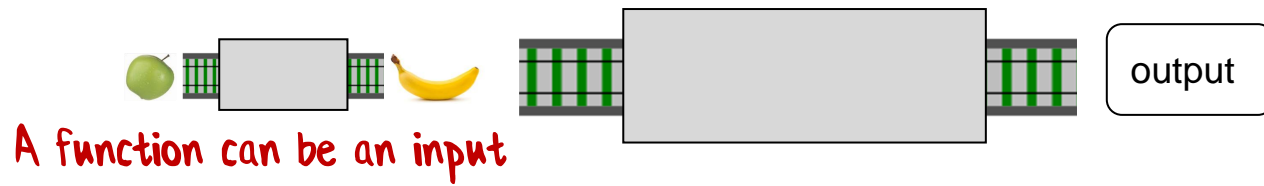
A function is a standalone thing,
not attached to a class

It can be used for inputs and outputs
of other functions

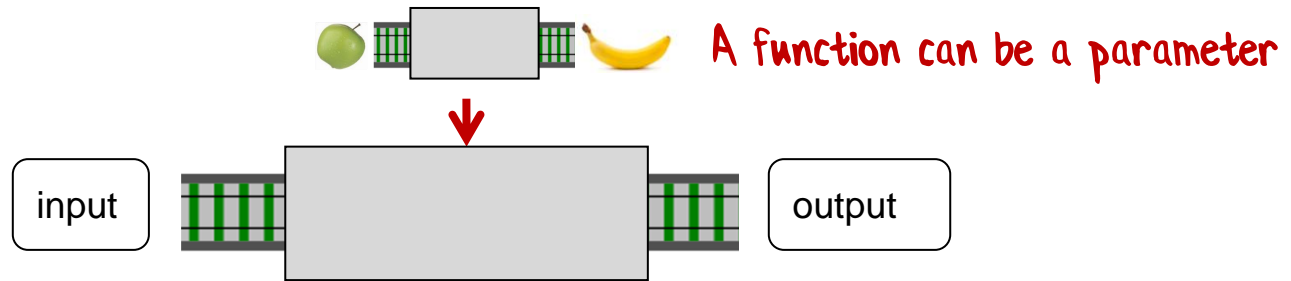
A function is a standalone thing



A function is a standalone thing



A function is a standalone thing



You can build very complex systems
from this simple foundation!

Core FP principle:
Composition everywhere



What is Composition?



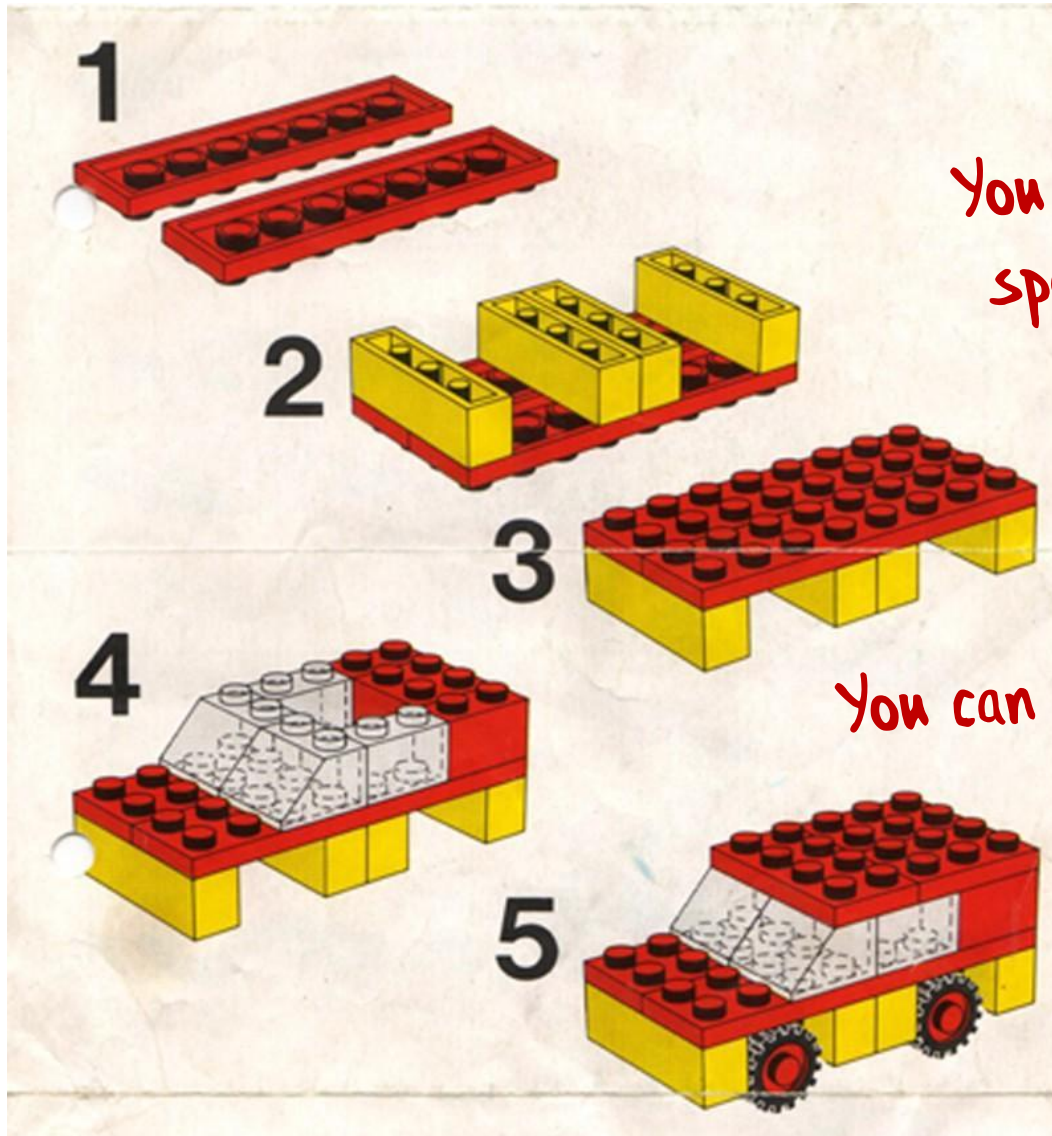
Lego Philosophy

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

All pieces are designed to be connected



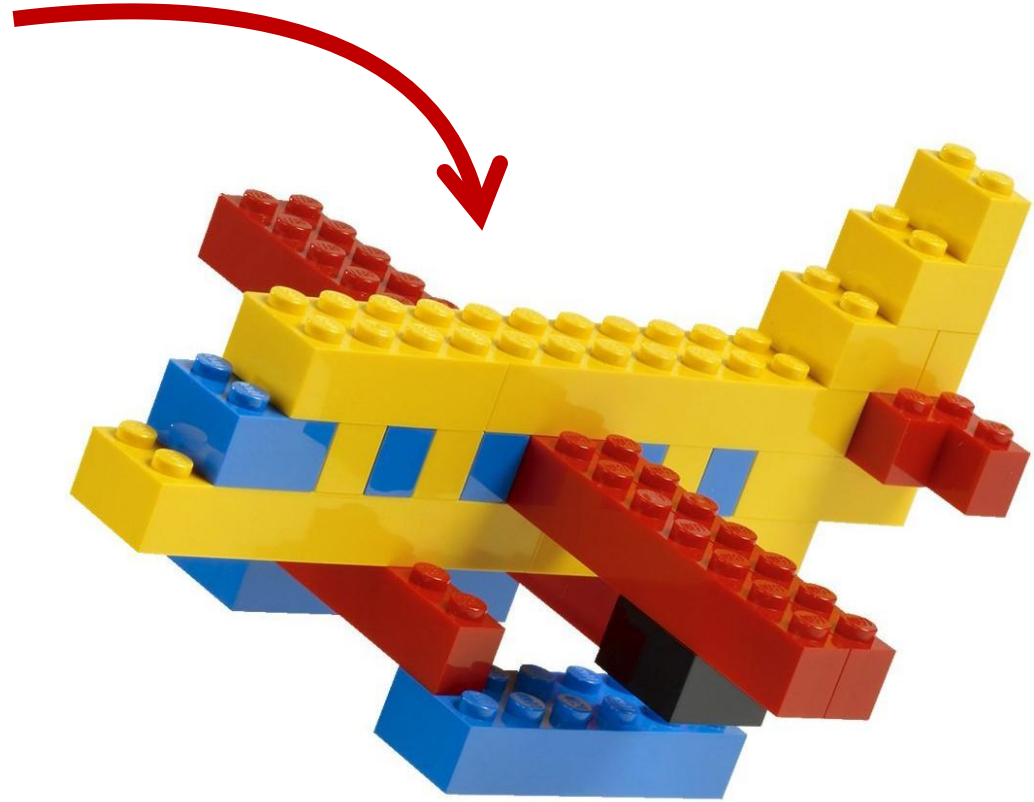
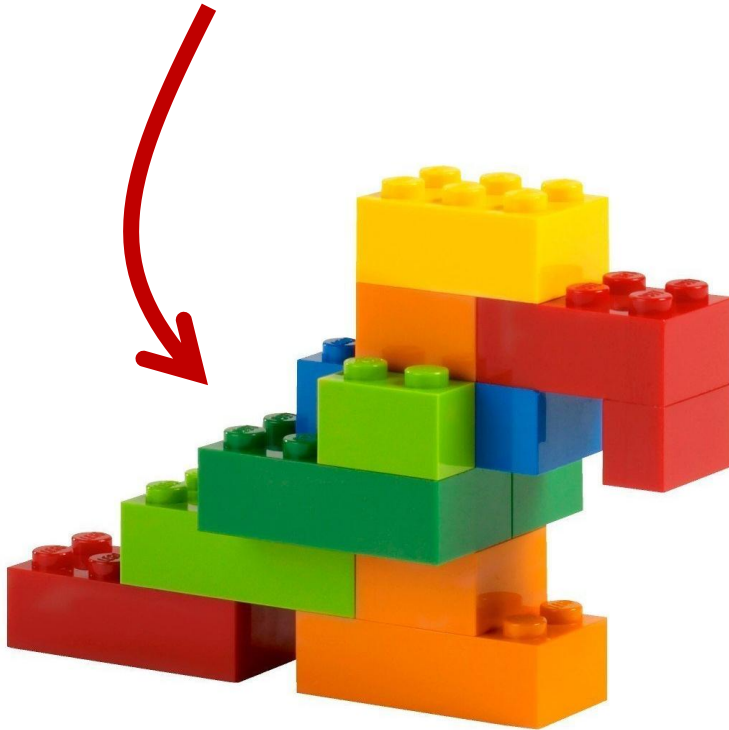
Connect two pieces together and
get another "piece" that can still be connected



You don't need to create a
special adapter to make
connections.

You can keep adding and adding.

The pieces are reusable in different contexts

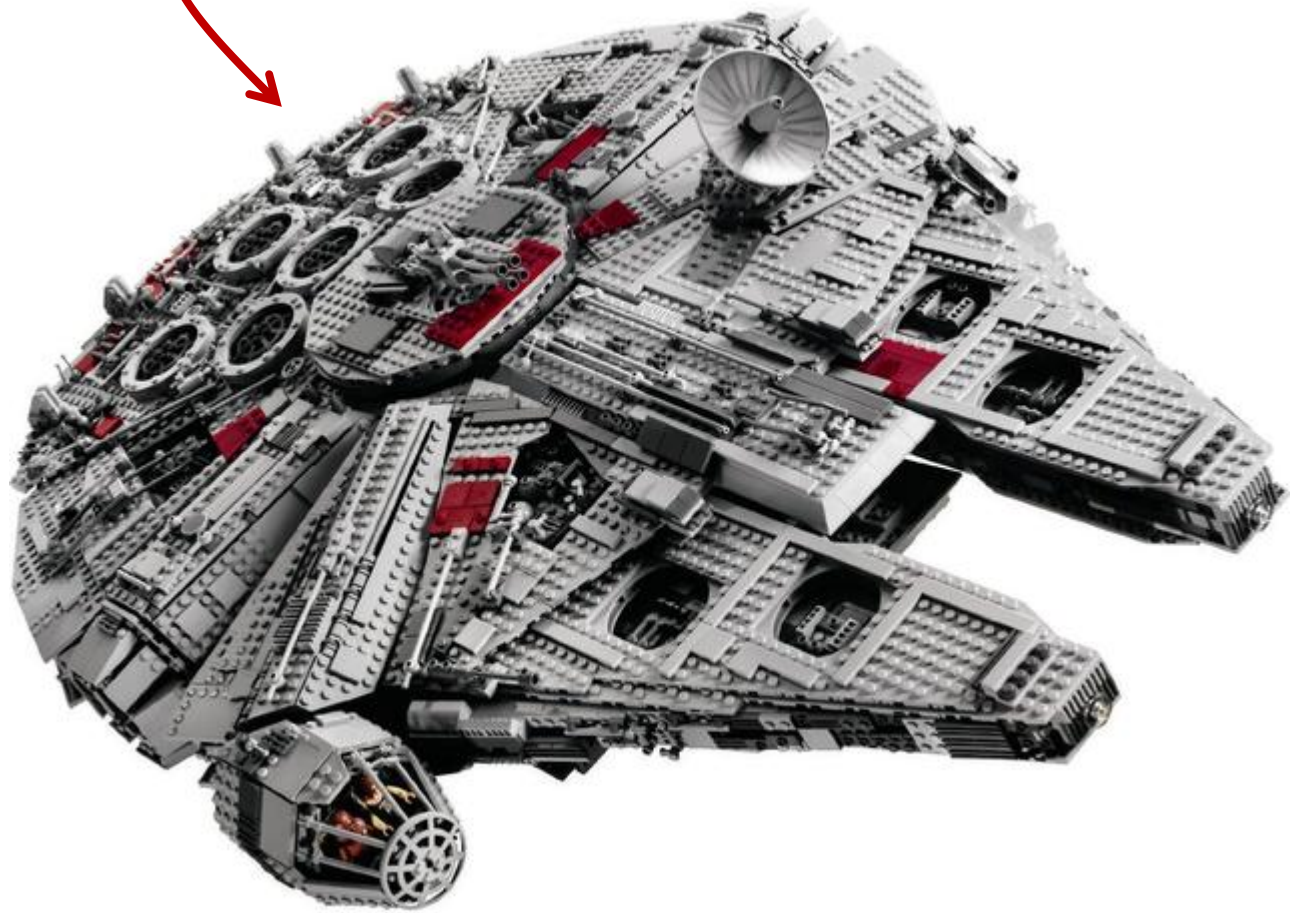


They are self contained.
No strings attached (literally).

Make big things from small things in the same way



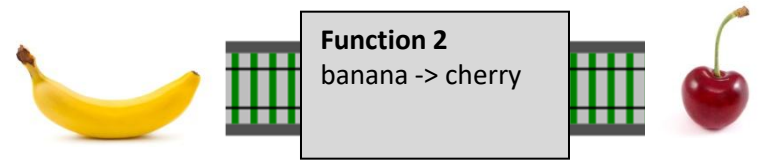
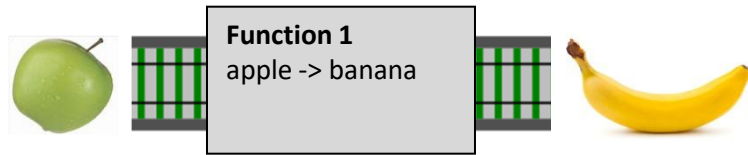
The Power of
Composition



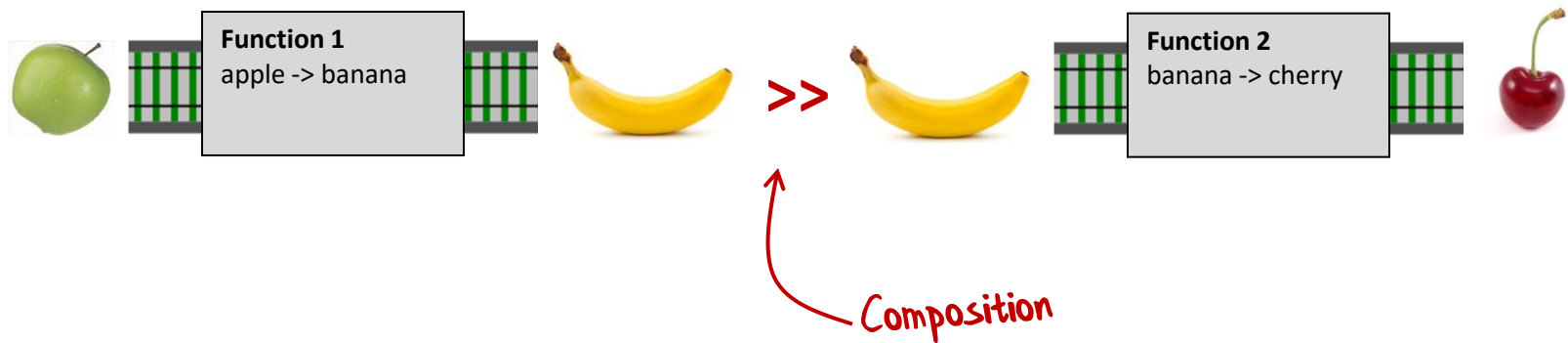
Function Composition



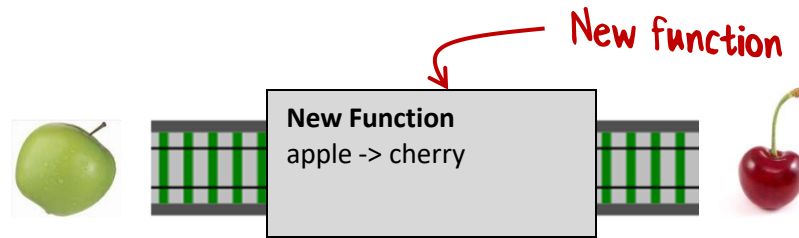
Function composition



Function composition



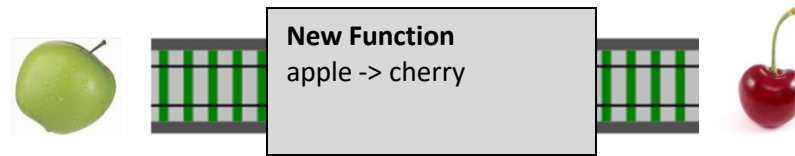
Function composition



Can't tell it was built
from smaller functions!

Where did the banana go?
(abstraction)

Function composition



A Very Important Point: For composition to work properly:

- Data must be immutable
- Functions must be self-contained, with no strings attached:
no side-effects, no I/O, no globals, etc

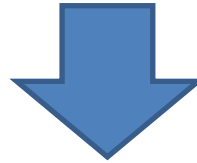
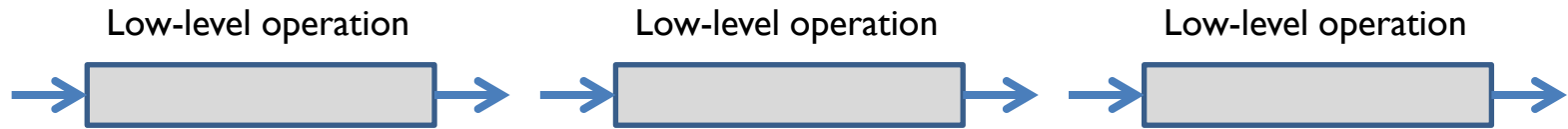
Building big things from functions

It's compositions all the way up



Low-level operation

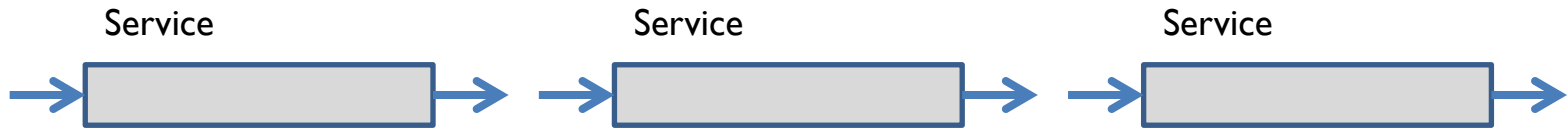




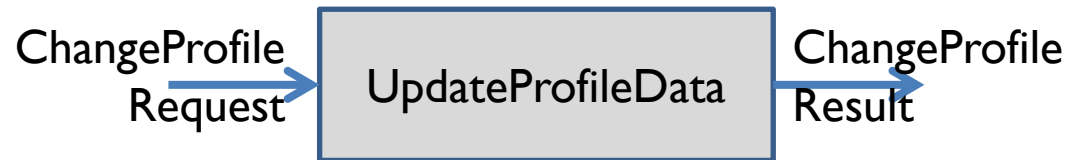
Service

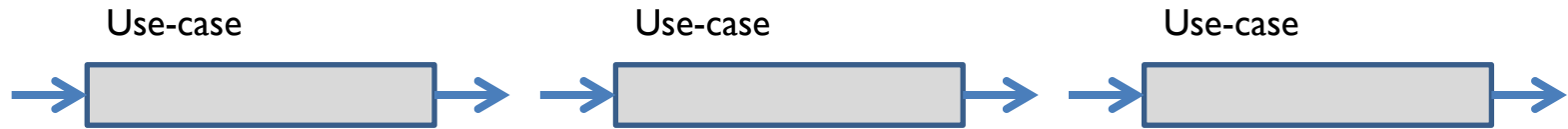


A "Service" is just like a microservice
but without the "micro" in front

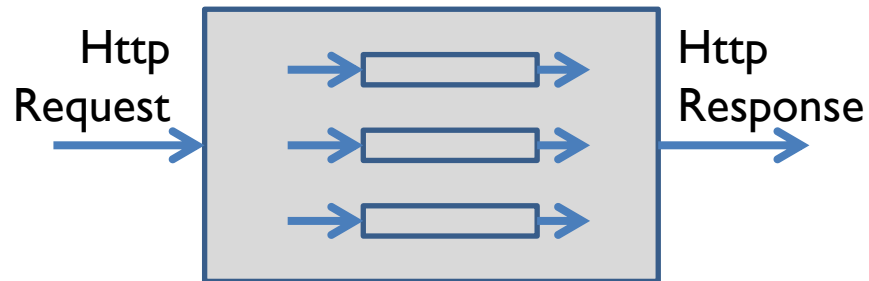


Use-case



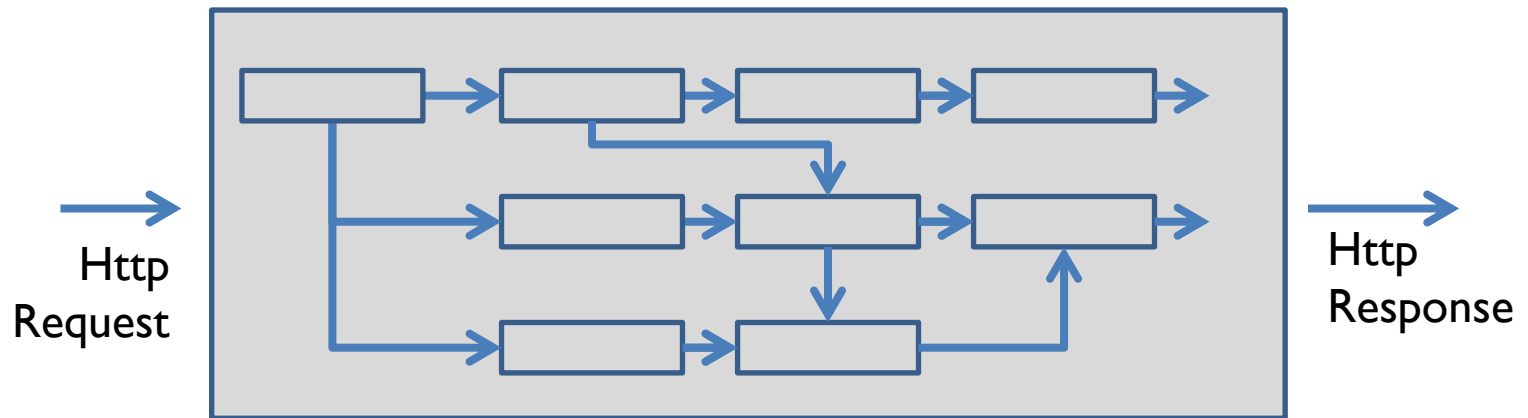


Web application

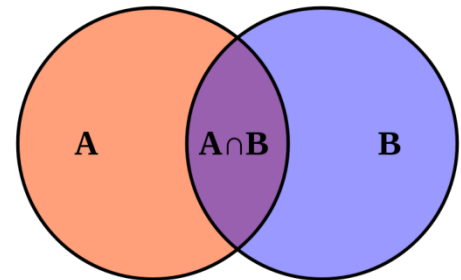


“Composition is fractal”

The Power of Composition

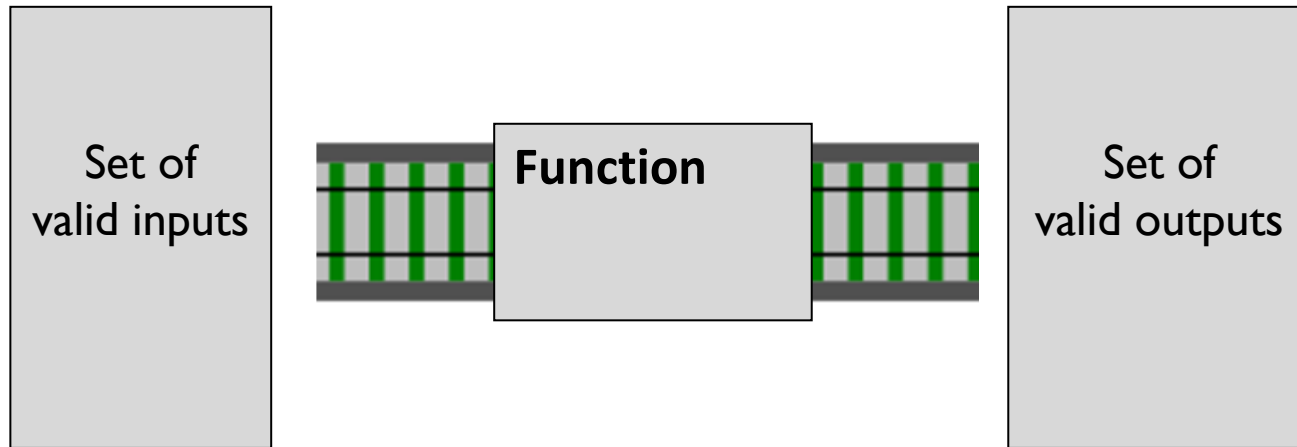


Core FP principle:
Types are not classes

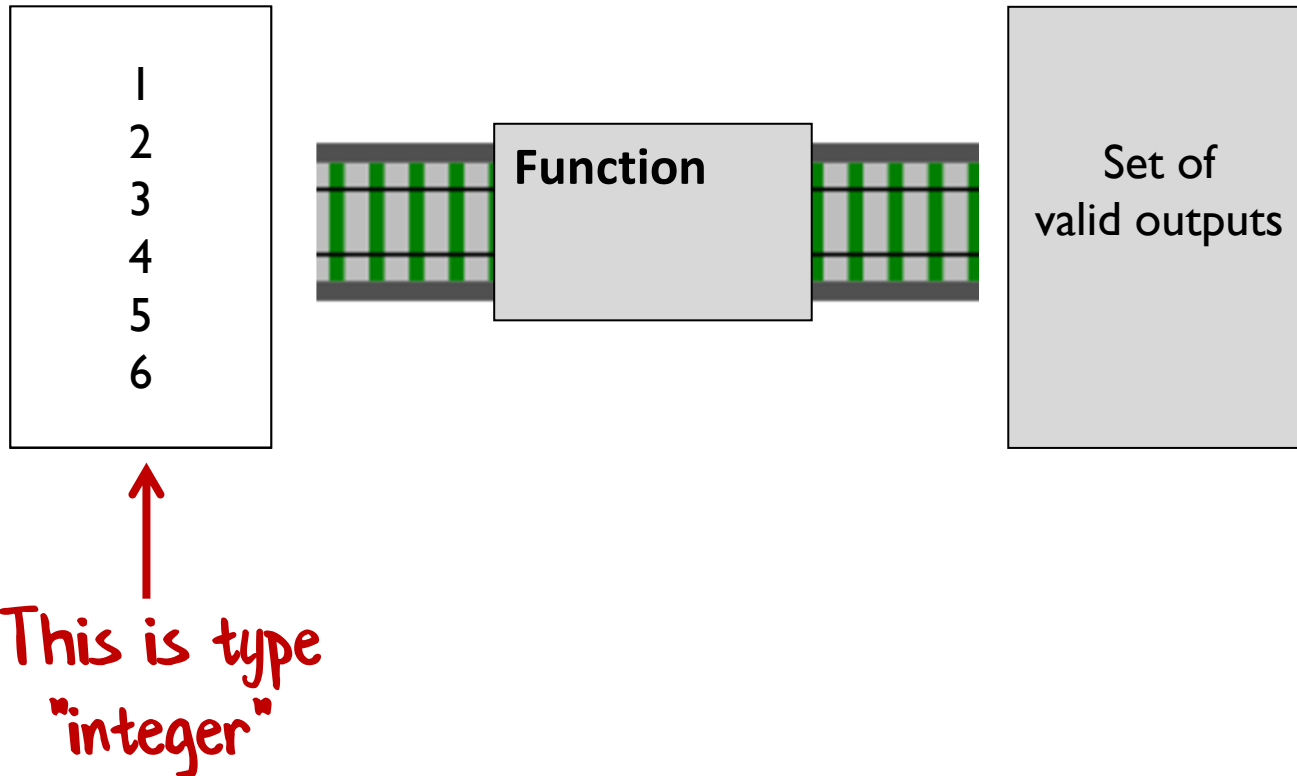


So, what is a type then?

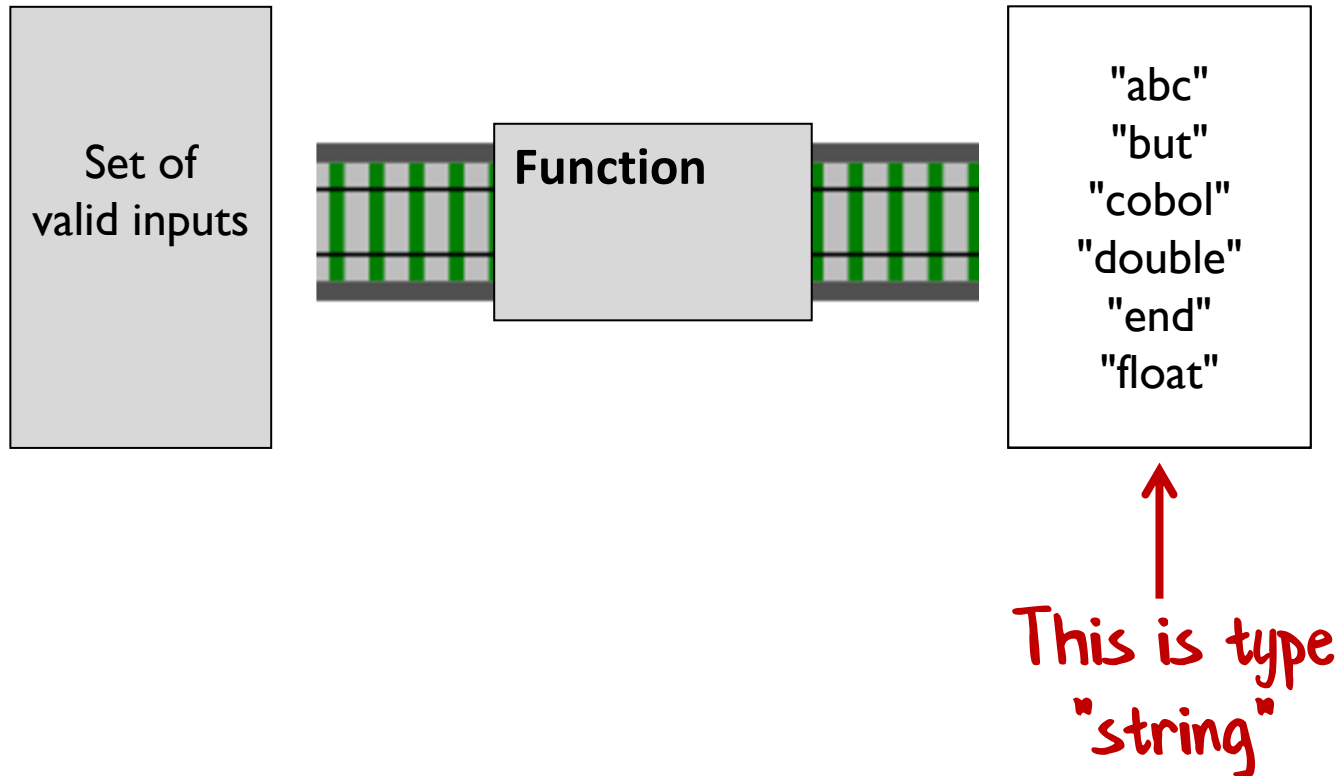
A type is a just a name
for a set of things



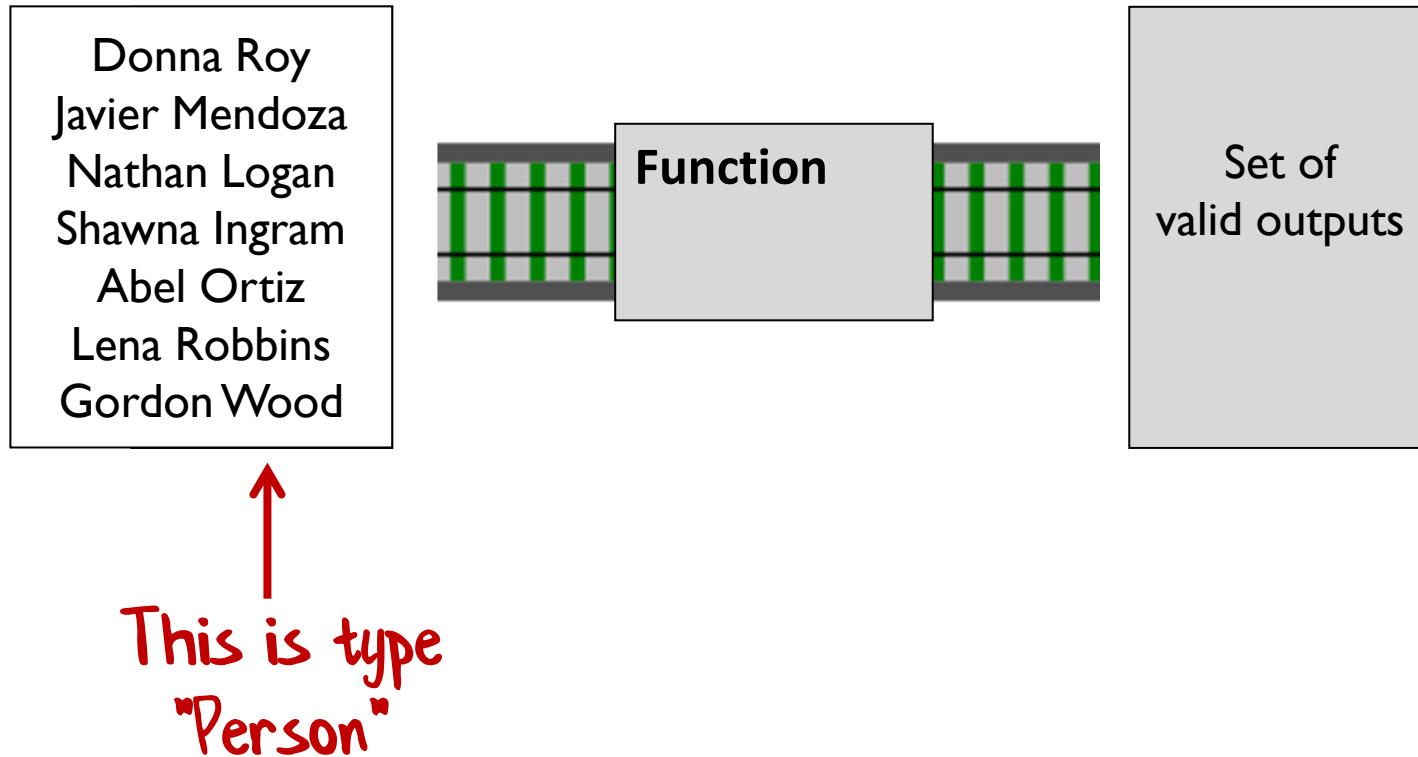
A type is a just a name
for a set of things



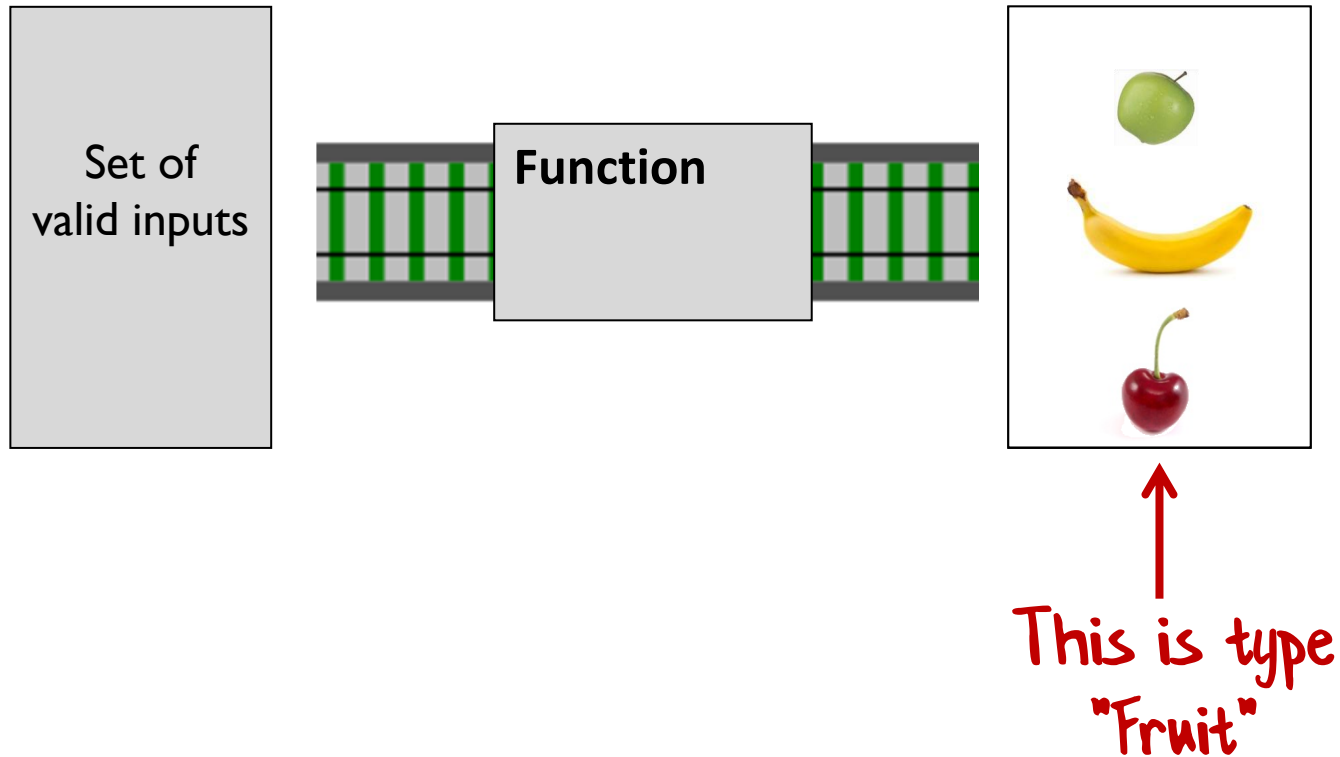
A type is a just a name
for a set of things



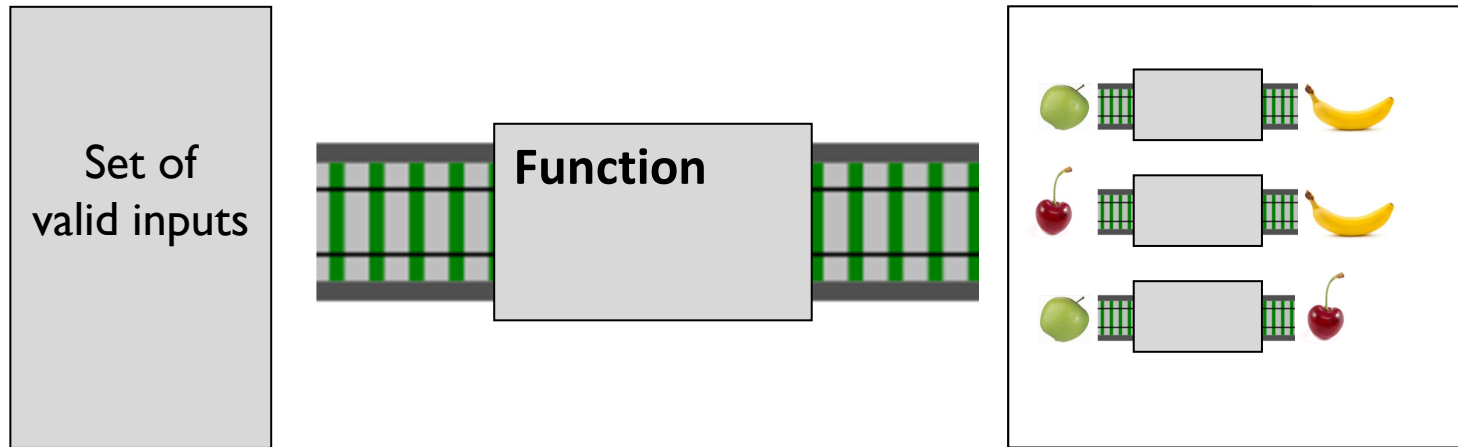
A type is a just a name
for a set of things



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

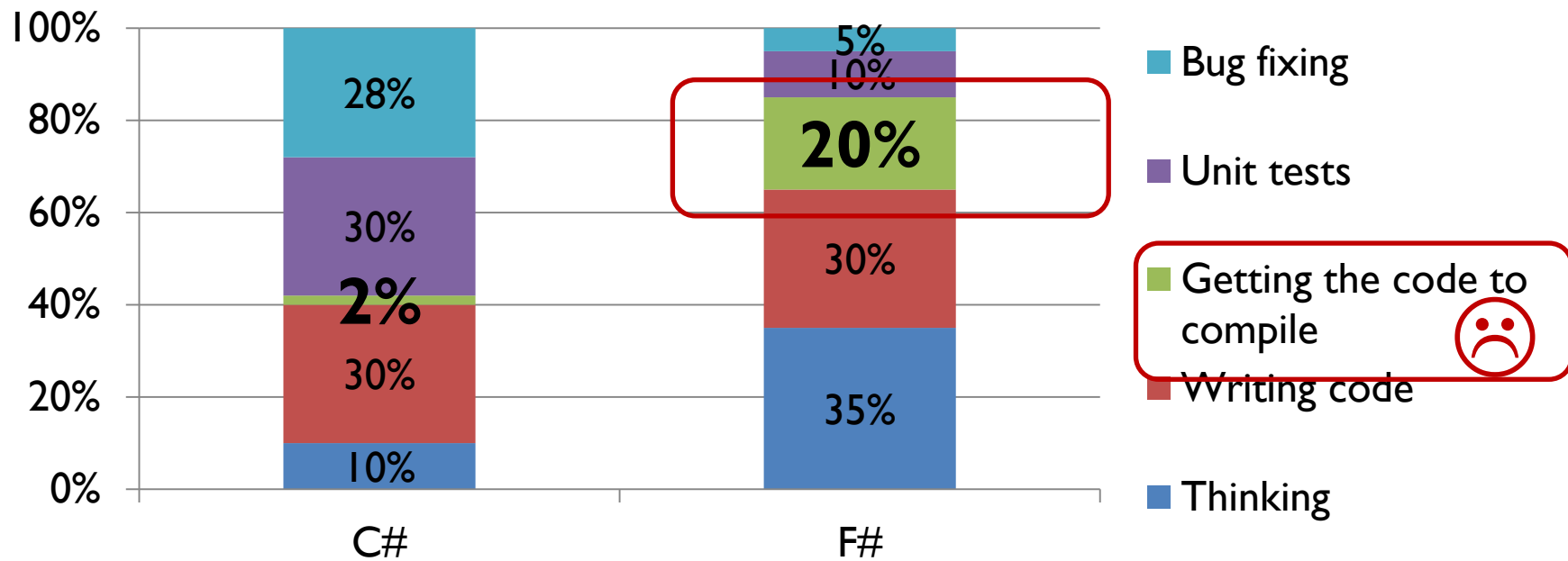
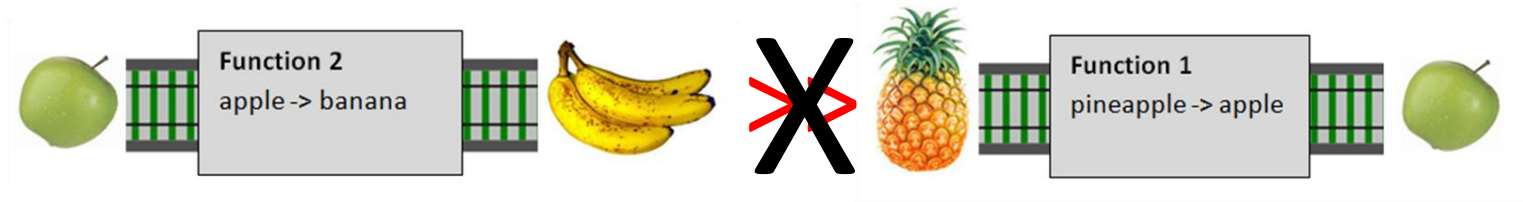


A type is a just a name
for a set of things



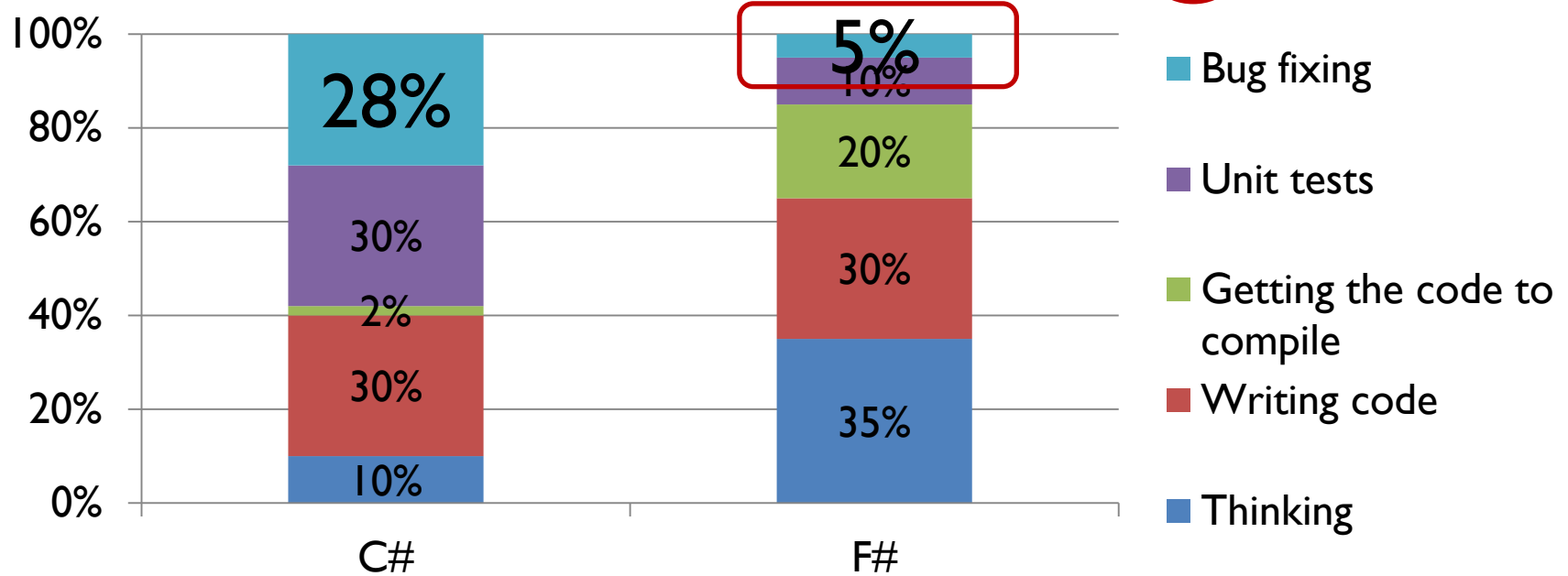
↑
This is a type of
Fruit- \rightarrow Fruit functions

Composition is type checked!



But the good news is...

A lot less bug fixing! 😊



Composition everywhere:
Types can be composed too

Composable

~~Algebraic~~ type system

New types are built from smaller types by:

Composing with “AND”

Composing with “OR”

*Only possible because behavior
is separate from data!*

Compose with “AND”

FruitSalad = One each of  and  and 




Example: pairs, tuples, records

A record type



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


Compose with “OR”

Snack =  or  or 

Not available in C#

A choice type

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

Real world example of type composition

Example of some requirements:

We accept three forms of payment:
Cash, Check, or Card.

For Cash we don't need any extra information
For Checks we need a check number
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 Check(int checkNo): IPaymentMethod  
{..}
```

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

In F# you would probably implement by composing types, like this:

```
type CheckNumber = int  
type CardNumber = string
```

← Primitive types

```
type CheckNumber = ...
```

```
type CardNumber = ...
```


Choice type
(using OR)



```
type CardType = Visa | Mastercard
```

```
type CreditCardInfo = {  
    CardType : CardType  
    CardNumber : CardNumber  
}
```

Record type (using AND)



```
type CheckNumber = ...  
type CardNumber = ...  
type CardType = ...  
type CreditCardInfo = ...
```

```
type PaymentMethod =
```

```
| Cash
```

```
| Check of CheckNumber
```

```
| Card of CreditCardInfo
```

← Choice type

```
type CheckNumber = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
```

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

← Another primitive type

← Another choice type


```
type CheckNumber = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD
```

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

← Record type

Final type built from many
smaller types:

The Power of Composition

```
type CheckNumber = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD

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

FP design principle:

Types are executable documentation

Types are executable documentation

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 * Rank
```

← Types can be nouns

```
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)
```

← Types can be verbs

```
type PickupCard = (Hand * Card) → Hand
```

Types are executable documentation

```
type CardType = Visa | Mastercard
```

```
type CardNumber = CardNumber of string
```

```
type CheckNumber = CheckNumber of int
```


```
type PaymentMethod =
```

```
| Cash
```

```
| Check of CheckNumber
```

```
| Card of CardType * CardNumber
```

*Can you guess what
payment methods are
accepted?*

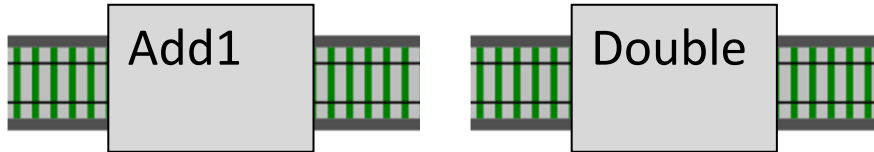


The End

This is everything you need to know
about functional programming

Composition in practice

Composition



```
let add1 x = x + 1
```

```
let double x = x + x
```

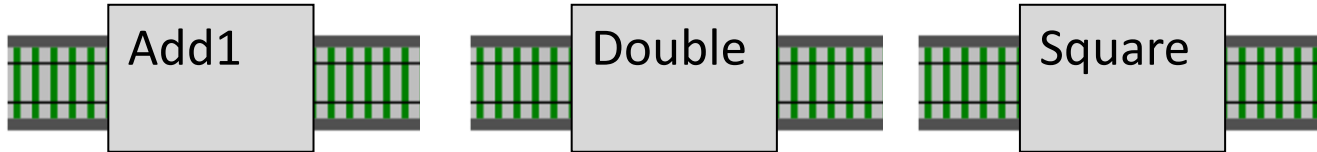
```
let add1_double = add1 >> double
```

```
let x = add1_double 5 // 12
```

Composition
operator

Note: using "snake_case" naming to
make the composition clear...

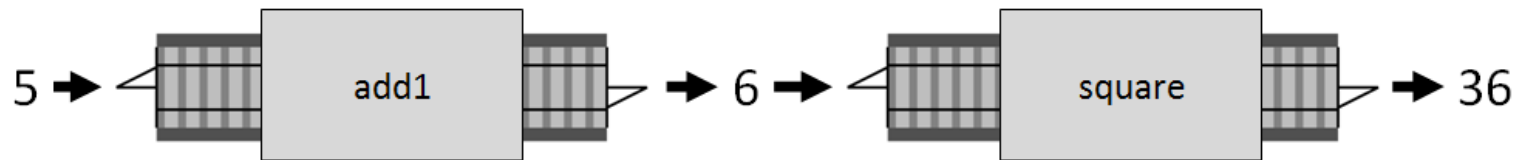
Composition



```
let add1_double_square =  
    add1 >> double >> square
```

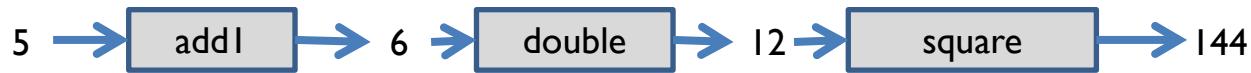
```
let x = add1_double_square 5    // 144
```

Piping



```
add1 5 // = 6  
double (add1 5) // = 12  
square (double (add1 5)) // = 144
```

*Standard way of nesting function calls
can be confusing if too deep*



This is easier to understand

Pipe symbol

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

pipe

Exercise: Try piping for yourself

04a-Composition-and-piping.fsx

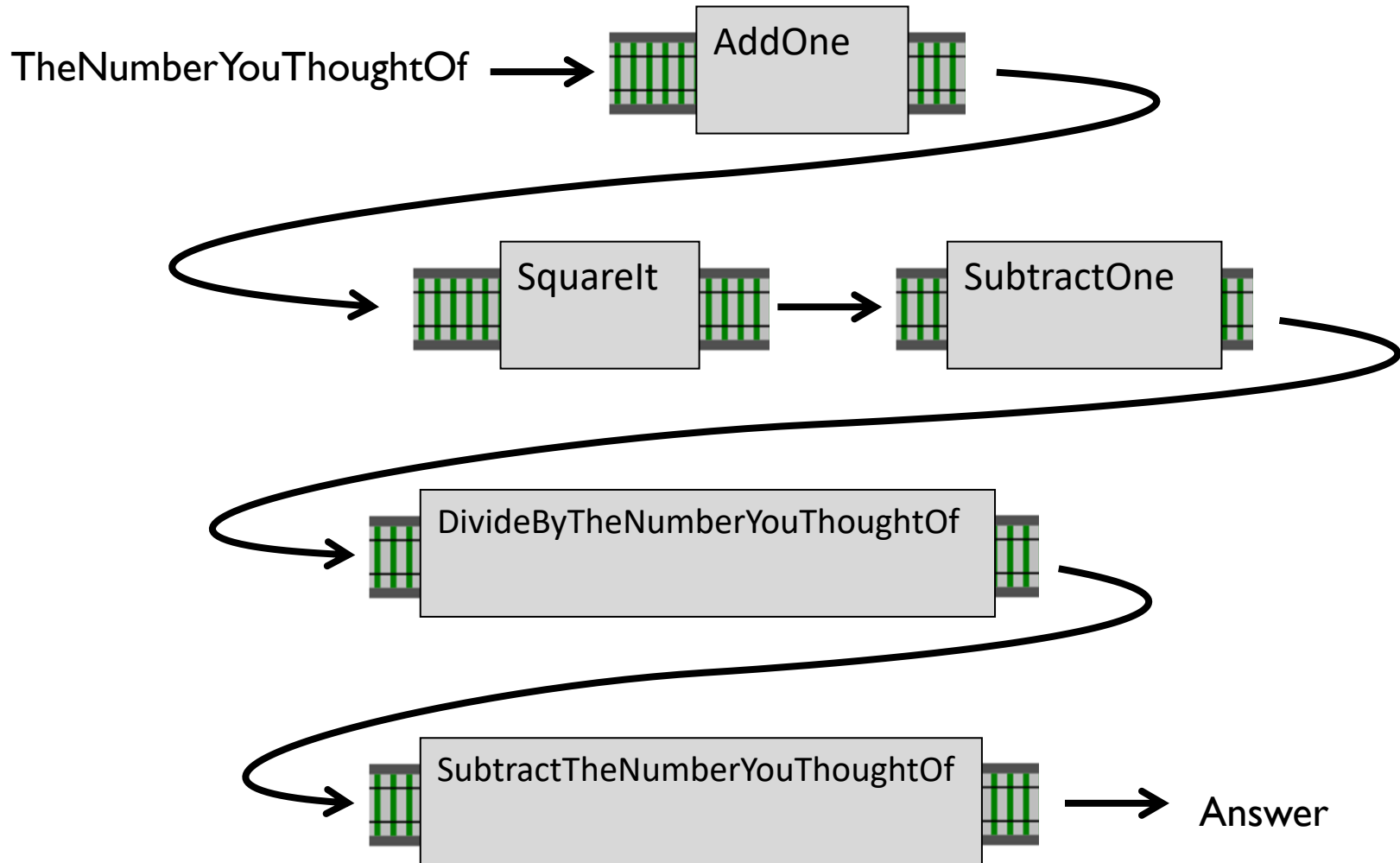
BASIC COMPOSITION:

THINK OF A NUMBER

Think of a number

- Think of a number.
- Add one to it.
- Square it.
- Subtract one.
- Divide by the number you first thought of.
- Subtract the number you first thought of.
- The answer is TWO!

Think of a number



Exercise: Think of a number

05a-Exercise-ThinkOfANumber.fsx

```
let thinkOfANumber numberYouThoughtOf =
```

```
// define a function for each step
```

```
let addOne x = x + 1
```

```
let squareIt x = x * x
```

```
let subtractOne x = x - 1
```

```
let divideByTheNumberYouThoughtOf x = x / numberYouThoughtOf
```

```
let subtractTheNumberYouThoughtOf x = x - numberYouThoughtOf
```

```
// then combine them using piping
```

```
numberYouThoughtOf
```

```
|> addOne
```

```
|> squareIt
```

```
|> subtractOne
```

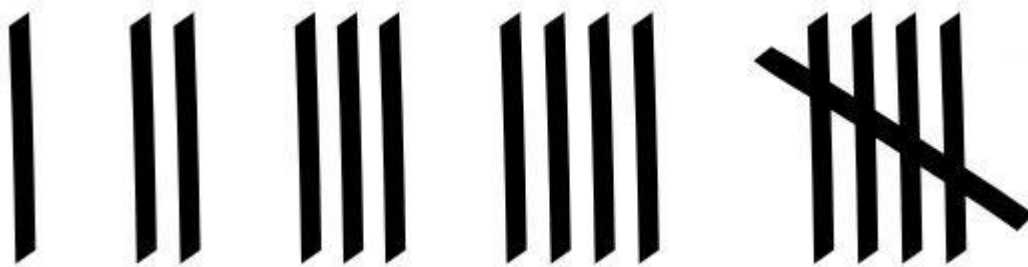
```
|> divideByTheNumberYouThoughtOf
```

```
|> subtractTheNumberYouThoughtOf
```

BASIC COMPOSITION:
ROMAN NUMERALS

To Roman Numerals

- Task: convert an integer to Roman Numerals
- 5 => "V", 10 => "X", 100 => "C" etc

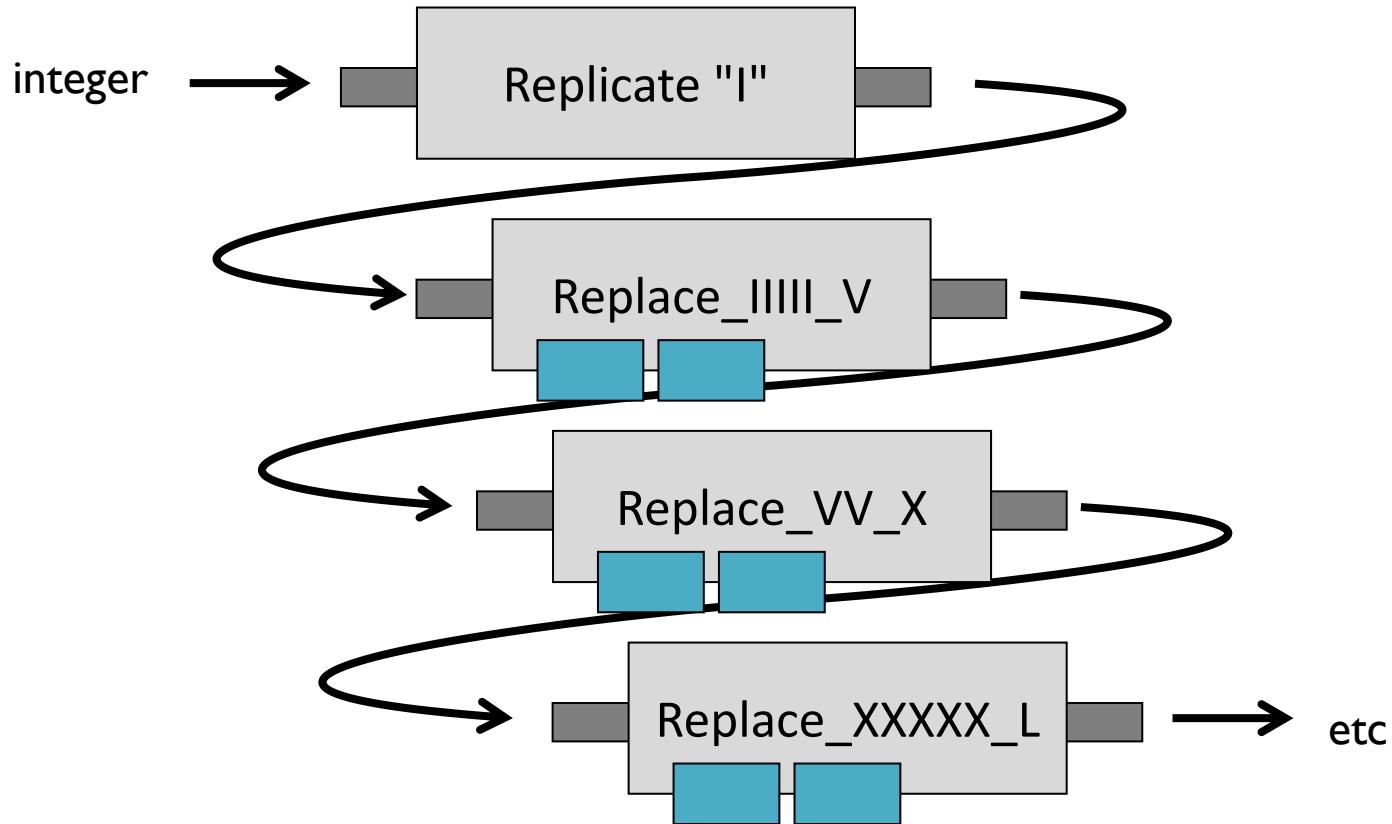


Roman numbers evolved
from this

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"
 - Replace five "C"s with a "D"
 - Replace two "D"s with a "M"

To Roman Numerals



Exercise: Roman numerals

05c-Exercise-ToRomanNumbers.fsx

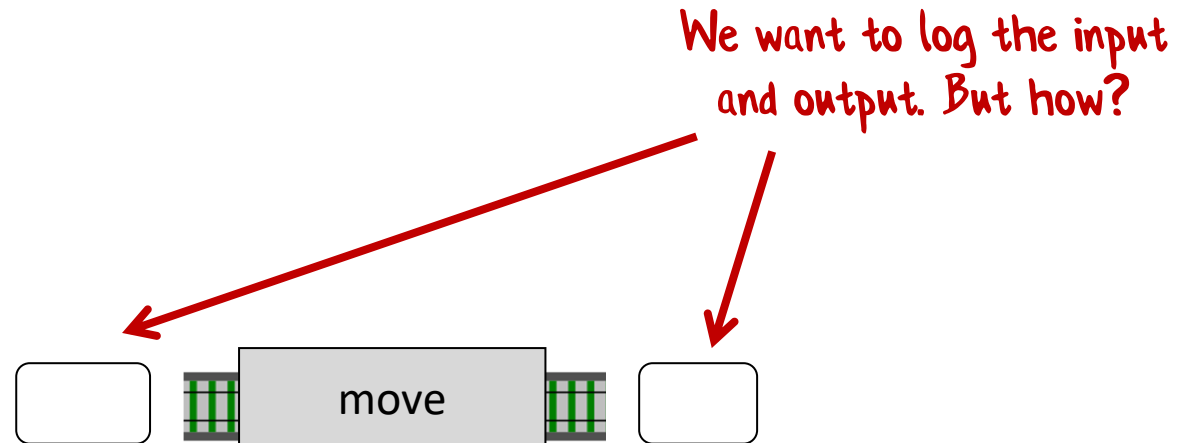
```
let toRomanNumerals number =  
  let replace_IIII_V = replace "IIII" "V"  
  let replace_VV_X = replace "VV" "X"  
  let replace_XXXXXX_L = replace "XXXXXX" "L"  
  let replace_LL_C = replace "LL" "C"  
  let replace_CCCCCC_D = replace "CCCCCC" "D"  
  let replace_DD_M = replace "DD" "M"
```

String.replicate number "I"

```
|> replace_IIII_V  
|> replace_VV_X  
|> replace_XXXXXX_L  
|> replace_LL_C  
|> replace_CCCCCC_D  
|> replace_DD_M
```


Decorator pattern using composition (Logging example)

Logging



Decorator pattern? Aspect-oriented?

Logging

Step 1: Create a log function



Logging

Step 2: glue all the functions
together using composition



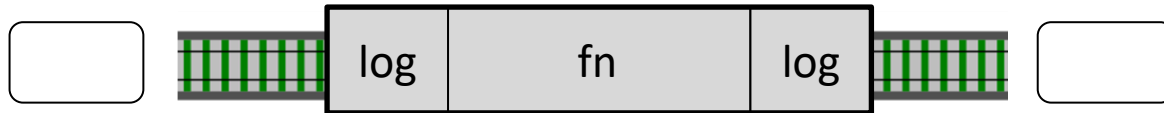
Logging

Step 2: glue all the functions
together using composition



Logging

Step 3: use the new function in
place of the old function
(same interface)



There's no need for a "decorator pattern"
in FP - it's just regular composition

Exercise: Decorator pattern

05e-Exercise-Decorator.fsx