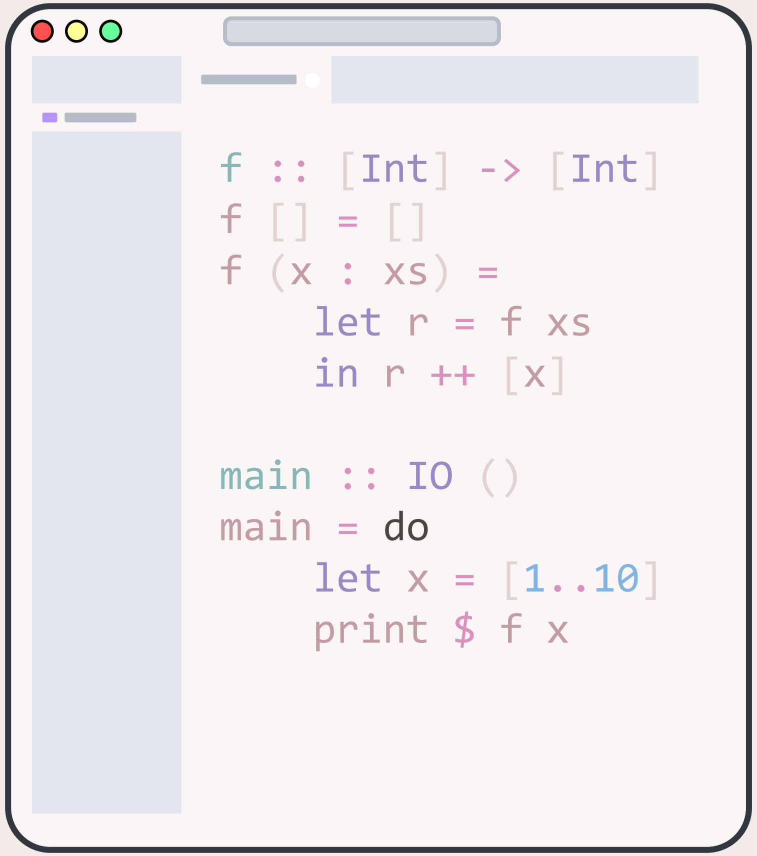
# IT5100A

## Industry Readiness: Typed Functional Programming

# Railway Pattern

Foo Yong Qi
yongqi@nus.edu.sg



# Railways

In an ideal world, you can compose computation very easily

```python
def add_one(x: int) -> int:
    return x + 1
def double(x: int) -> int:
    return x * 2
def div_three(x: int) -> float:
    return x / 3

print(div_three(double(add_one(4))))
```

# However, things are rarely perfect

```python
@dataclass
class Email:
    name: str
    domain: str

@dataclass
class User:
    username: str
    email: Email
    salary: int | float
```

```python
def parse_email(s: str) -> Email:
    if '@' not in s:
        raise ValueError
    s = s.split('@')
    if len(s) != 2 or '.' not in s[1]:
        raise ValueError
    return Email(s[0], s[1])

def parse_salary(s: str) -> int | float:
    try:
        return int(s)
    except:
        return float(s)
```

However, things are rarely perfect

```python
def main():
    n = input('Enter name: ')
    e = input('Enter email: ')
    s = input('Enter salary: ')
    try:
        print(User(n, parse_email(e), parse_salary(s)))
    except:
        print('Some error occurred')
```

Exceptions are being thrown everywhere, hard to keep track and compose exceptional functions!

With using the railway pattern and well-thought-out data structures and composition operators:

```haskell
data Email =
  Email { emailUsername :: String
        , emailDomain :: String   }
  deriving (Eq, Show)

data Salary = SInt Int
            | SDouble Double
  deriving (Eq, Show)

data User =
  User { username :: String
       , userEmail :: Email
       , userSalary :: Salary }
  deriving (Eq, Show)
```

```haskell
parseEmail :: String -> Maybe Email
parseEmail email = do
    guard $ '@' `elem` email
         && length e == 2
         && '.' `elem` last e
    return $ Email (head e) (last e)
  where e = split '@' email

parseSalary :: String -> Maybe Salary
parseSalary s =
  let si = SInt <$> readMaybe s
      sf = SDouble <$> readMaybe s
  in  si <|> sf
```

With using the railway pattern and well-thought-out data structures and composition operators:

```haskell
main :: IO ()
main = do
  n <- input "Enter name: "
  e <- input "Enter email: "
  s <- input "Enter salary: "
  let u = User n <$> parseEmail e <*> parseSalary s
  putStrLn $ maybe "Some error occurred" show u
```

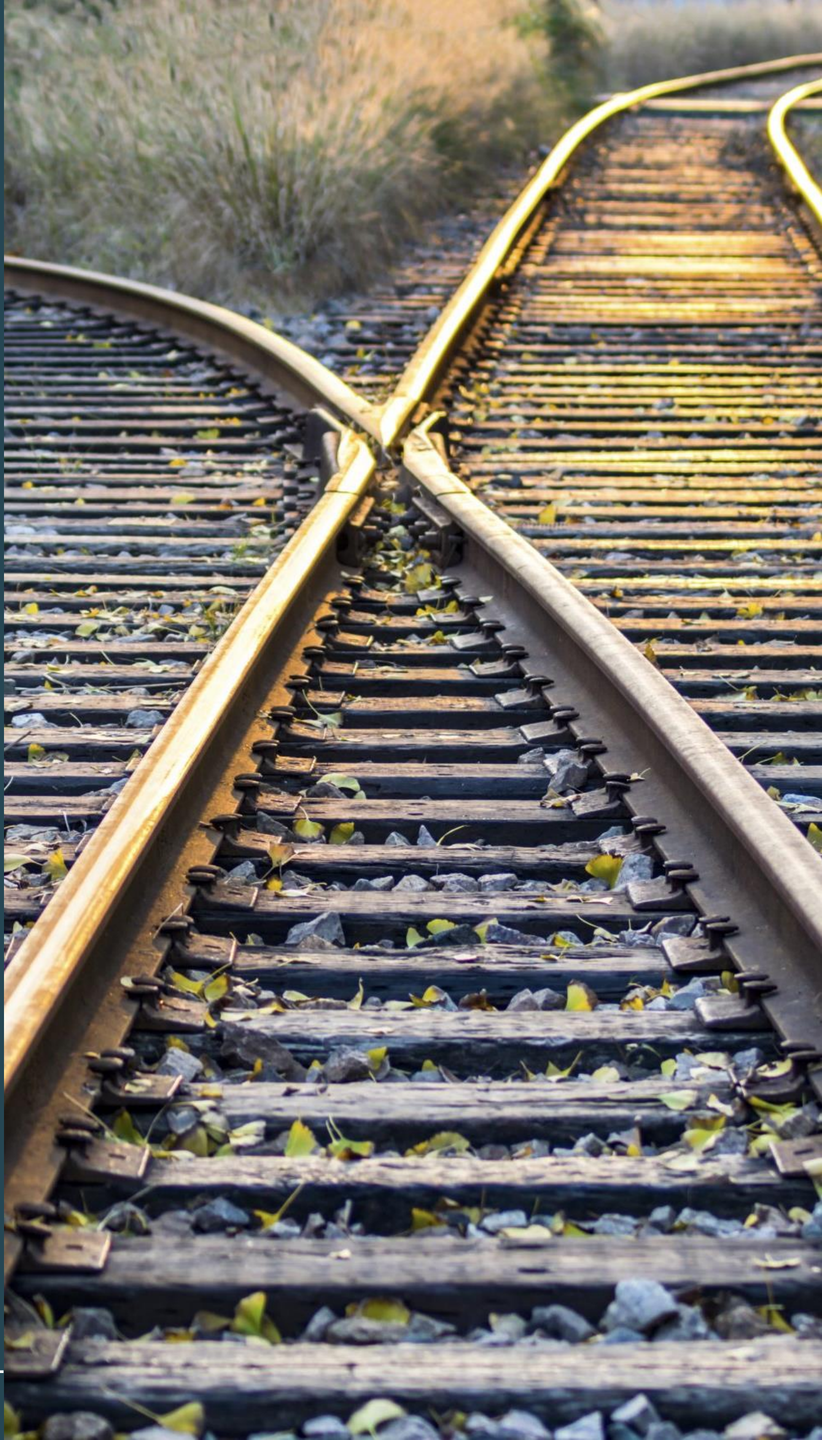
No exceptions, just purely-functional magic

# Railway Pattern

What is it?

What data structures and functions can we use to support this?

How do we code with the railway pattern?



# Context/Notions of Computation

# Context/Notions of Computation

Many popular languages lie to you in many ways:

```python
def happy(x: int) -> int:
    raise Exception
```

```java
String happy() {
    return null;
}
```

Python: exceptions not reported in type signature

Java: you can receive `null` without you knowing

We can't lie in Haskell and shouldn't lie in general… what now?

Create types that accurately describes what our functions actually do!

```haskell
-- Something or nothing
data Maybe a = Nothing
             | Just a

-- If a is an error, then
-- Either an error or some b
data Either a b = Left a
                | Right b
```

These types act as **contexts** or **notions of computation**:

`Maybe a` — an **a** or nothing

`Either a b` — either **a** or **b**

`[a]` — a list of possible **a**s (nondeterminism)

`IO a` — an IO action resulting in **a**

These types act as **contexts** or **notions of computation**:

`Maybe a` — an `a` or nothing

`Either a b` — either `a` or `b`

`[a]` — a list of possible `a`s (nondeterminism)

`IO a` — an IO action resulting in `a`

`Maybe`, `Either a`, `[]` and `IO` all have kind `* -> *`, wraps around a type

`Maybe` `Int`

`Either Int` `String`

`[]` `Char`

`IO` `String`

Example: function that might return nothing

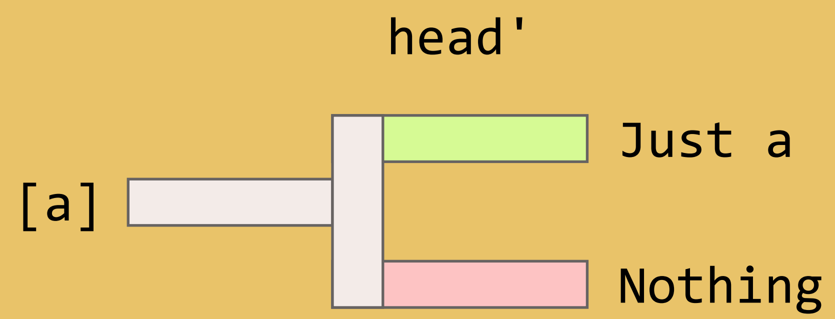
```haskell
head' :: [a] -> Maybe a
head' [] = Nothing
head' (x : _) = Just x
```

Examples: function maybe returning an error message

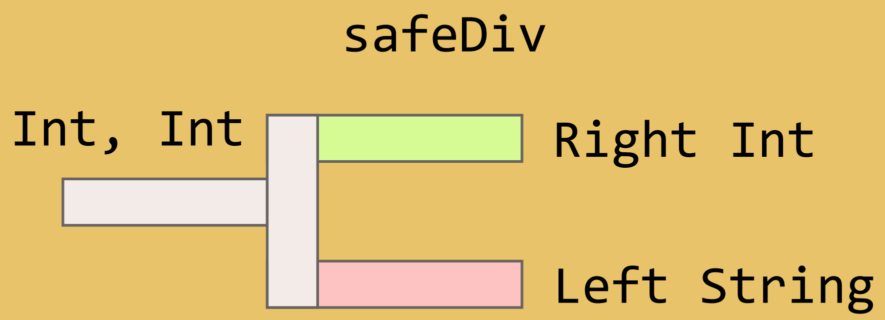
```haskell
safeDiv :: Int -> Int -> Either String Int
safeDiv x 0 = Left "Cannot divide by zero"
safeDiv x y = Right $ x `div` y
```

## We can see this is as branching railways

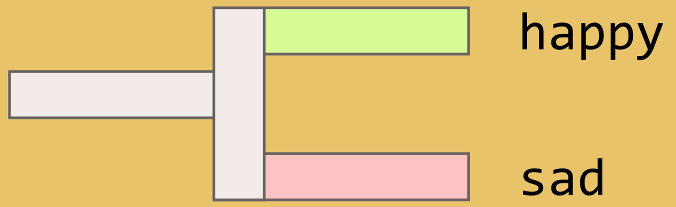
```haskell
head' :: [a] -> Maybe a
head' [] = Nothing
head' (x : _) = Just x
```




```haskell
safeDiv :: Int -> Int -> Either String Int
safeDiv x 0 = Left "Cannot divide by zero"
safeDiv x y = Right $ x `div` y
```



Railway pattern: pattern of using algebraic data types to encapsulate the different possible outputs from a function



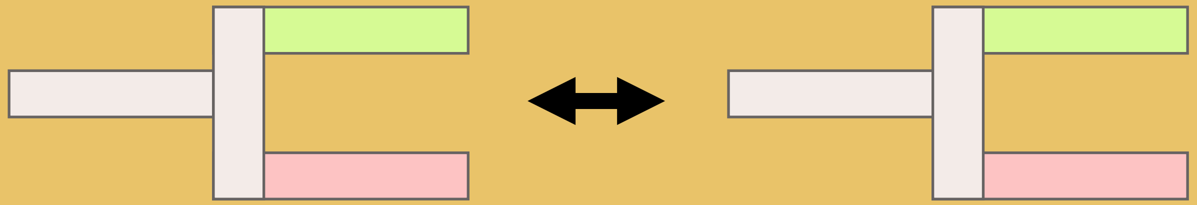

This is a **natural consequence** of purely functional programming—instead of writing functions that opaquely cause side-effects, the functions are made **transparent via the appropriate data structures**

The correct data structure to use depends on the **notion of computation** you want to express

| | | | |
|---|---|---|---|
| `Maybe` `Int` | `Either Int` `String` | `[]` `Char` | `IO` `String` |

If you produce nothing in some scenarios, use `Maybe`, if you want to produce something or something else (like an error), use `Either`, etc.

But, working with the railway pattern tedious without additional tools…



Let us take some ideas from… Category Theory!

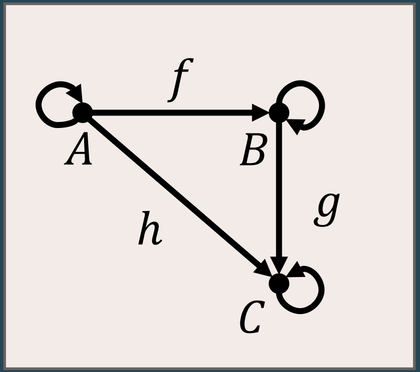
# Category Theory

Briefly…

# Category

A category $\mathcal{C}$ consists of

- Objects $A, B, C, \ldots$
- Morphisms $f: A \to B$, $g: B \to C, \ldots$

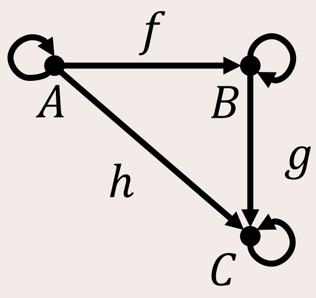



- For every $f: A \to B$ and $g: B \to C$ the composition $g \circ f: A \to C$ exists
- Composition is **associative**: for all morphisms $f\ g\ h$, $f \circ (g \circ h) = (f \circ g) \circ h$
- Composition is **unital**: every object $A$ has an identity morphism $1_{\mathrm{A}}$ such that for all $f: A \to B$, $1_B \circ f = f \circ 1_A = f$

# Category

A category $\mathcal{C}$ consists of

- Dots $A, B, C, \ldots$
- Arrows $f: A \to B$, $g: B \to C, \ldots$

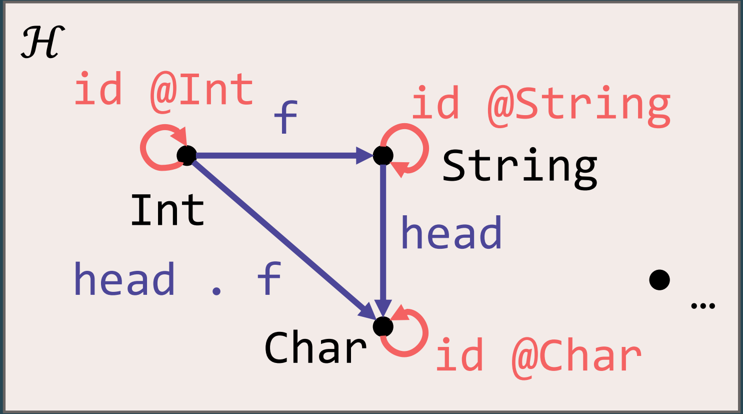



- Joining two arrows together gives another arrow
- There is a unique way to join three arrows together
- Every dot has an arrow pointing to itself, such that joining it with any other arrow $f$ gives $f$.

# Category of Types

Types and functions in Haskell form a category $\mathcal{H}$

- Objects are types like **`Int`** and **`String`**
- Morphisms are functions like **`(+1)`** and **`head`**




- The composition of two functions with **`(.)`** is a new function
- Composition of functions with **`(.)`** is associative
- Every type has the identity function **`id x = x`**

```
f :: Int -> String
f x = show (x + 2)
```

For example, $\texttt{f}: \texttt{Int} \to \texttt{String}$ is a morphism in $\mathcal{H}$

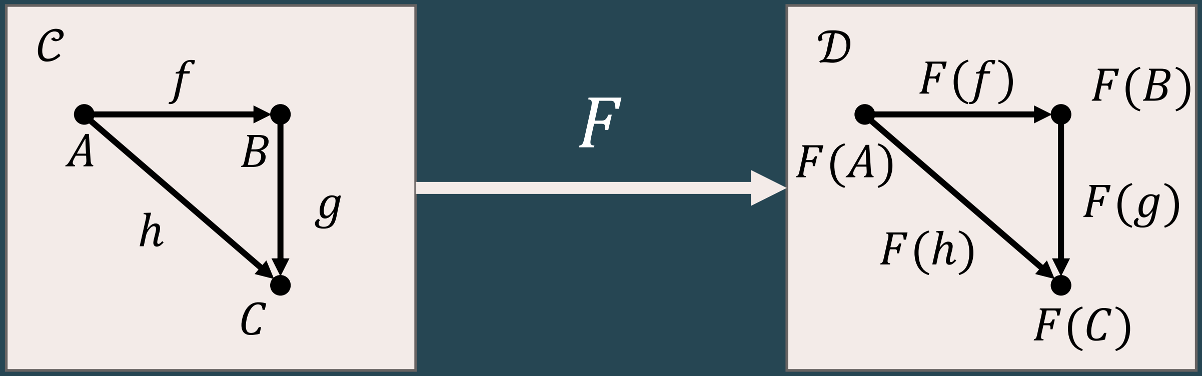
Who cares???

Because the types in Haskell assemble into a category, let's see if there is anything that category theory can tell us…

# Functors

A functor $F: \mathcal{C} \to \mathcal{D}$ is a mapping between two categories $\mathcal{C}$ and $\mathcal{D}$ where

- For every object $X$ in $\mathcal{C}$, $F(X)$ is in $\mathcal{D}$
- For every morphism $f: A \to B$ in $\mathcal{C}$, $F(f): F(A) \to F(B)$ is in $\mathcal{D}$
- For all $f$ and $g$ in $\mathcal{C}$, $F(g) \circ F(f) = F(g \circ f)$
- For all $A$, $F(1_A) = 1_{F(A)}$



# Functors

A functor $F: \mathcal{C} \rightarrow \mathcal{D}$ maps dots and arrows between $\mathcal{C}$ and $\mathcal{D}$, preserving composition and identities

$$\mathcal{C}\quad \begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\ & C \end{tikzcd} \quad \xrightarrow{\quad F \quad} \quad \mathcal{D}\quad \begin{tikzcd} F(A) \arrow[r, "F(f)"] \arrow[dr, "F(h)"'] & F(B) \arrow[d, "F(g)"] \\ & F(C) \end{tikzcd}$$

Two parts to a functor in $\mathcal{H}$:

(1) maps types to types

(2) maps functions to functions

(1) `[]` maps `a` to `[a]` for all types `a` in $\mathcal{H}$

(2) How do we map functions `f :: a -> b` to `F(f) :: [a] -> [b]` while preserving composition and identities?

```
ghci> f :: Int -> String
ghci> f x = show (x + 2)
ghci> f 3
"5"
ghci> :t map f
map f :: [Int] -> [String]
ghci> map f [3]
["5"]
```

- For all $f$ and $g$ in $\mathcal{C}$, $F(g) \circ F(f) = F(g \circ f)$
- For all $A$, $F(1_A) = 1_{F(A)}$

`map` preserves composition and identities—behaves in the **most obvious way**!

```
ghci> (map (*2) . map (+3)) [1, 2, 3]
[8, 10, 12]
ghci> map ((*2) . (+3)) [1, 2, 3]
[8, 10, 12]

ghci> :set -XTypeApplications
ghci> map (id @Int) [1, 2, 3]
[1, 2, 3]
ghci> id @[Int] [1, 2, 3]
[1, 2, 3]
```

`[]` and `map` form a functor over $\mathcal{H}$!

Two parts to a functor in $\mathcal{H}$:

(1) maps types to types

(2) maps functions to functions

(1) `Maybe` maps `a` to `Maybe a` for all types `a` in $\mathcal{H}$

(2) How do we map functions `f :: a -> b` to `F(f) :: Maybe a -> Maybe b` while preserving composition and identities?

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f (Just x) = Just (f x)
maybeMap f Nothing  = Nothing
```

- For all $f$ and $g$ in $\mathcal{C}$, $F(g) \circ F(f) = F(g \circ f)$
- For all $A$, $F(1_A) = 1_{F(A)}$

`maybeMap` preserves composition and identities—behaves in the **most obvious way**!

```
ghci> (maybeMap (*2) . maybeMap (+3)) (Just 1)
Just 8
ghci> maybeMap ((*2) . (+3)) (Just 1)
Just 8

ghci> :set -XTypeApplications
ghci> maybeMap (id @Int) (Just 1)
Just 1
ghci> id @(Maybe Int) (Just 1)
Just 1
```

`Maybe` and `maybeMap` form a functor over $\mathcal{H}$!

Maybe

Either a

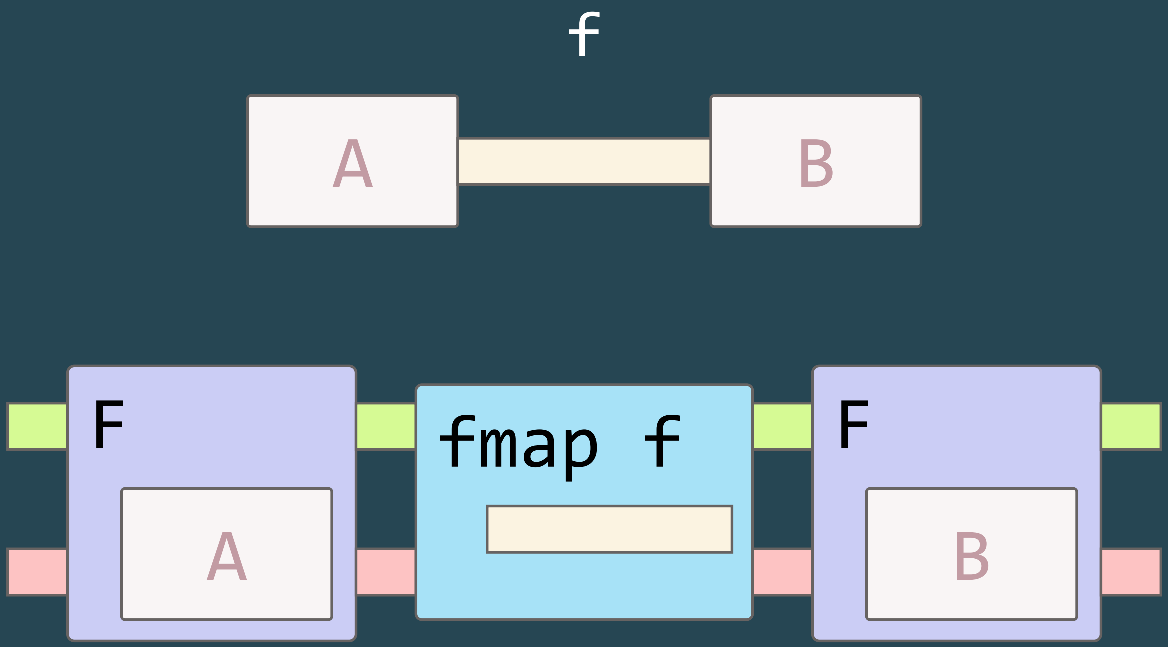
[]

IO

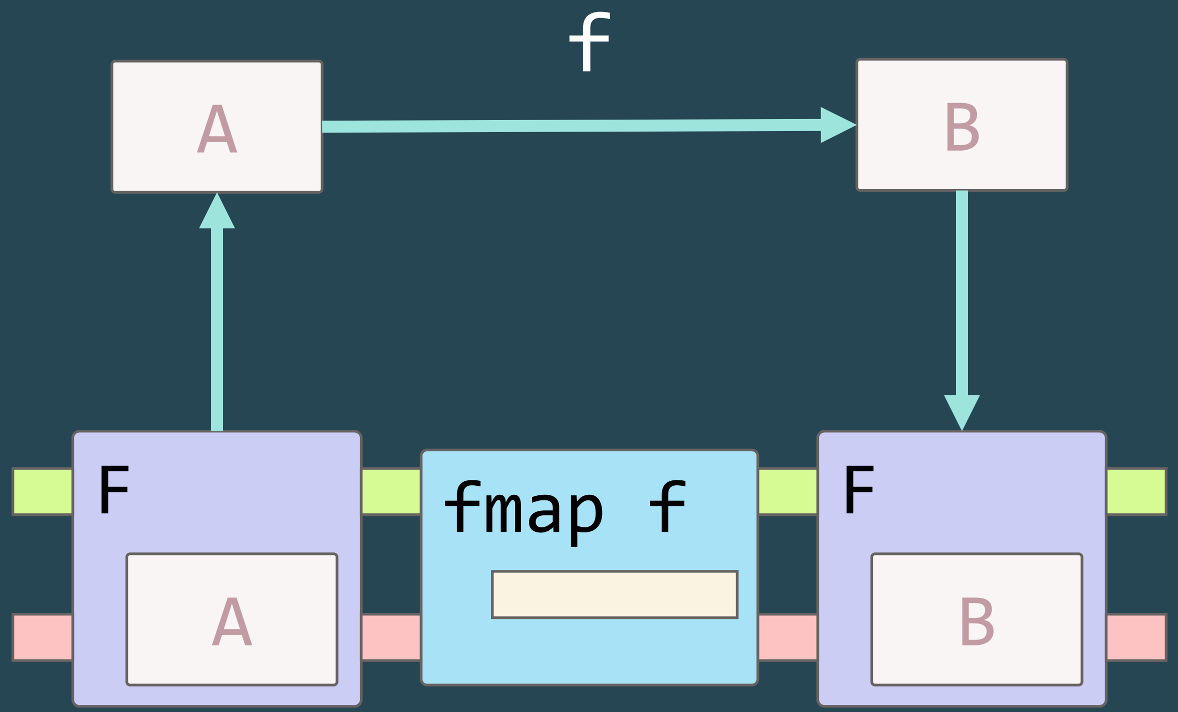
Given any context and a supporting function that preserves composition and identities, we have a `Functor`!

```haskell
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

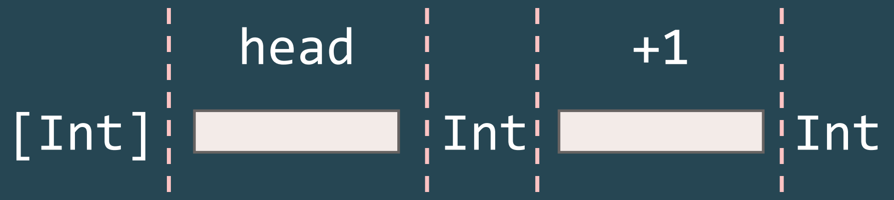
Given any functor `F` and a function `f` from `A` to `B`, `fmap f` is a function from `F A` to `F B` and **behaves as we should expect**



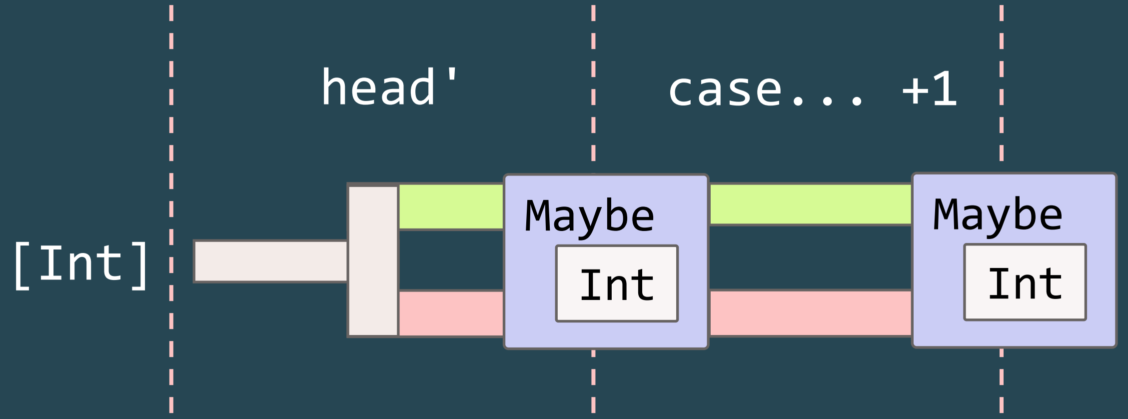
# Intuition: fmap



```haskell
ls = [1, 2, 3]
x = head ls
y = x + 1
```

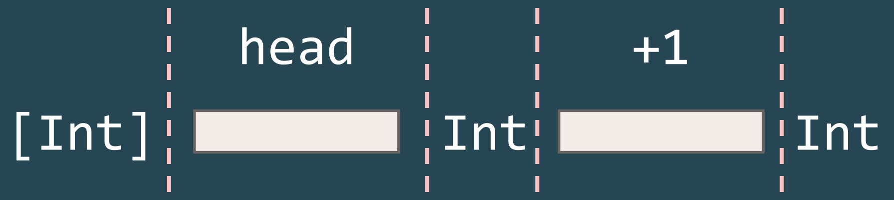



```haskell
head' :: [a] -> Maybe a
head' [] = Nothing
head' (x : _) = Just x

ls = [1, 2, 3]
x = head' ls
y = case x of
    Just z -> Just $ z + 1
    Nothing -> Nothing
```

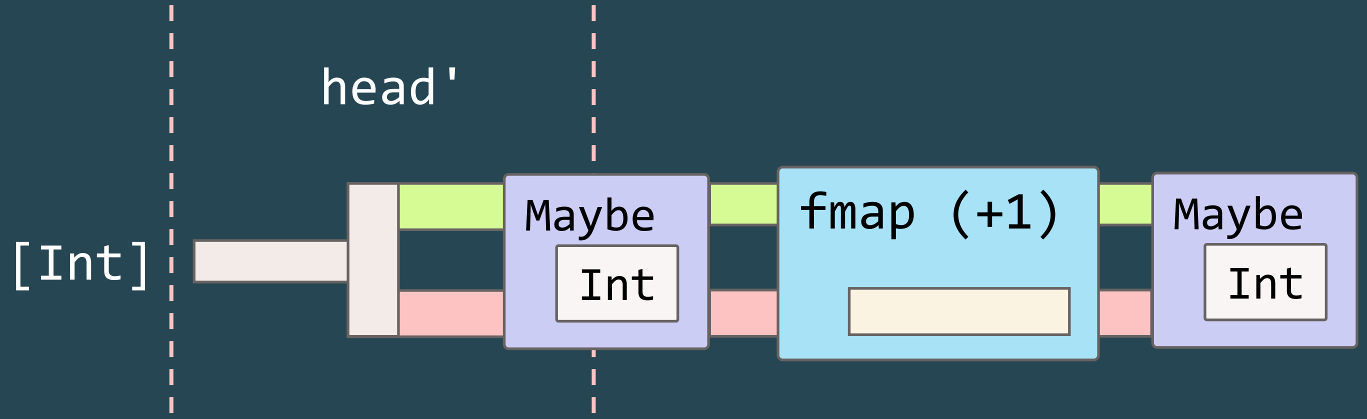



Don't torture yourself!

```haskell
ls = [1, 2, 3]
x = head ls
y = x + 1
```




```haskell
head' :: [a] -> Maybe a
head' [] = Nothing
head' (x : _) = Just x

ls = [1, 2, 3]
x = head' ls
y = fmap (+1) x
```




**fmap** can be used for exactly this purpose

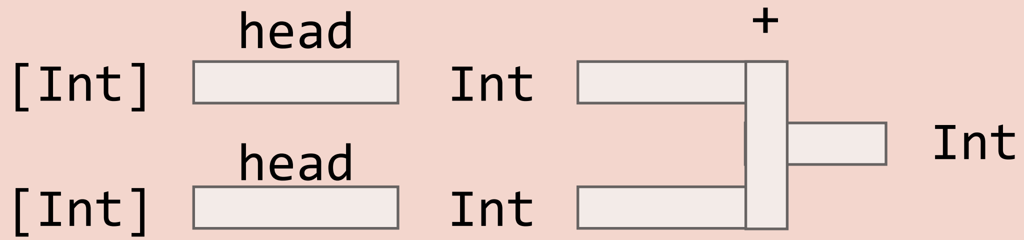
# Category Theory

- Category Theory is **not the main point** here
- Instead, Category Theory inspires tools that **support commonly-used programming patterns** backed by **well-defined notions**
- E.g. when we say something is a functor, it means that it obeys well-known laws and you can use it assuredly

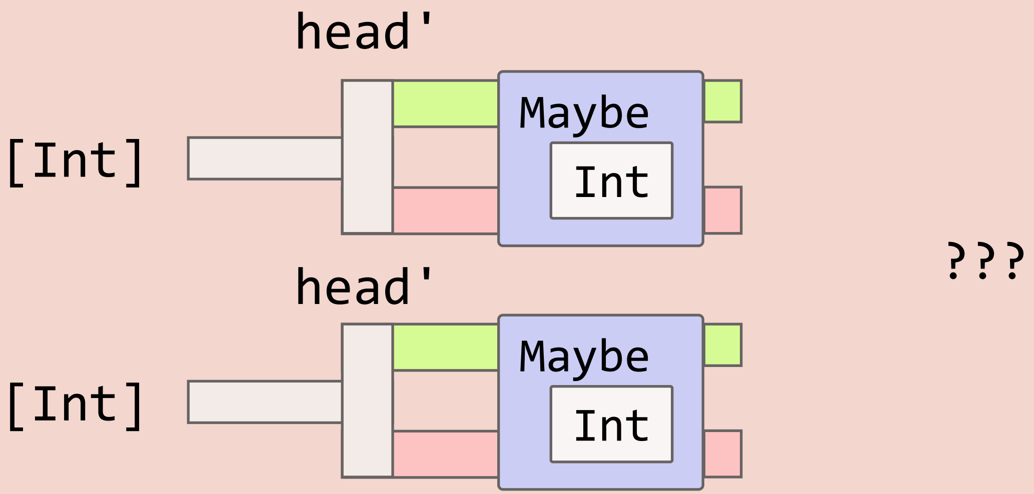
# Parallel Railways

What if we had 2 (or more) parallel railways and want to merge them?

```haskell
x = head [1, 2, 3]
y = head [4, 5, 6]
z = x + y -- 5
```




```haskell
x = head' [1, 2, 3]
y = head' [4, 5, 6]
z = x + y -- ???
```



# Monoidal Functors

A lax-monoidal functor $F: \mathcal{C} \to \mathcal{D}$ between two monoidal categories $(\mathcal{C}, \otimes, I_\mathcal{C})$ and $(\mathcal{D}, \circledast, I_\mathcal{D})$ is a functor $F: \mathcal{C} \to \mathcal{D}$ equipped with coherence maps

- A natural transformation $\phi_{A,B}: FA \circledast FB \to F(A \otimes B)$
- A morphism $\phi: I_\mathcal{D} \to FI_\mathcal{C}$

such that…

Not important!

# Applicative Functors

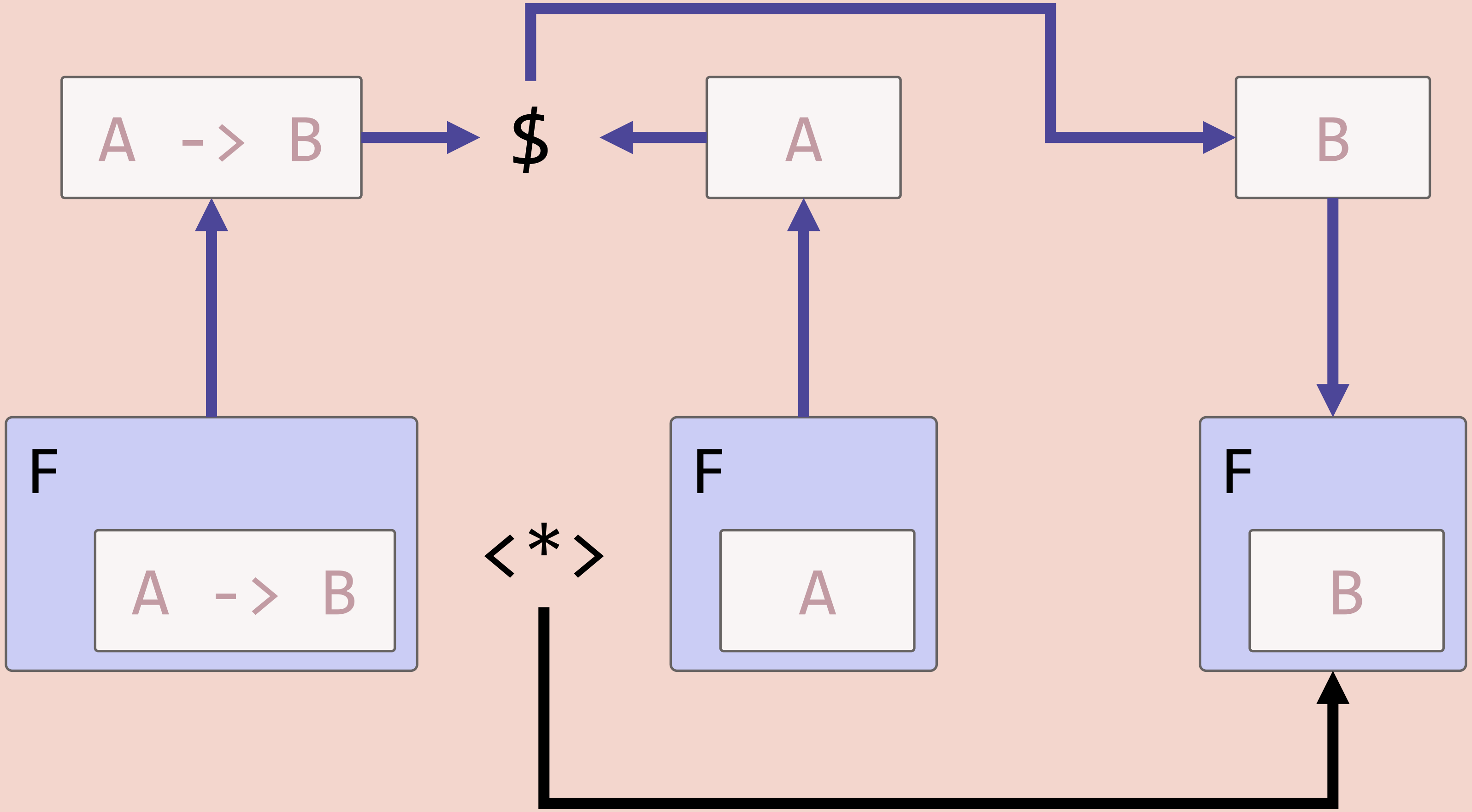
```haskell
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

`pure` : pure computation in context
`<*>` : function application in context

Subject to:

- Identity: `pure id <*> v = v`
- Homomorphism: `pure f <*> pure x = pure (f x)`
- Interchange: `u <*> pure y = pure ($ y) <*> u`
- Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

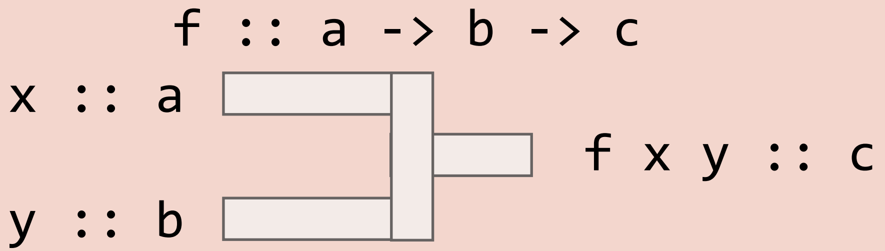
# Intuition: <*>



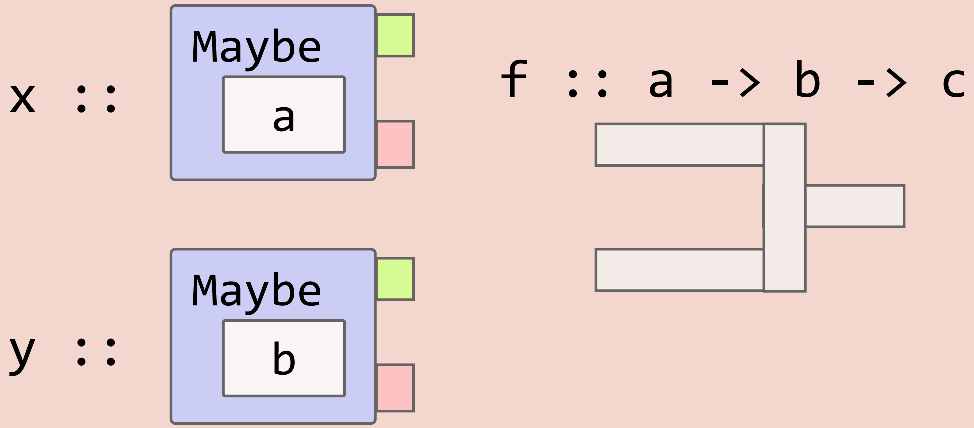
# Applicative Functors

```haskell
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    _ <*> Nothing = Nothing
    Just f <*> Just x = Just (f x)
```

`pure` and `<*>` behave in the obvious way

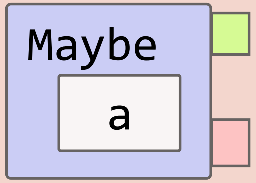



Previously we had two objects of type `a` and `b` and a function `f :: a -> b -> c`, and from these we can easily get `c`

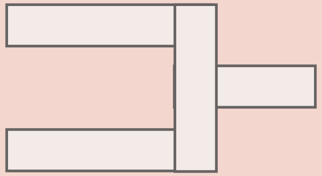



Now how do we apply `f` on `Maybe a` and `Maybe b` in the obvious way?

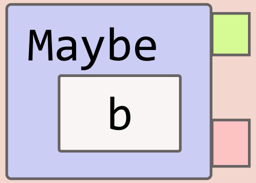
x ::



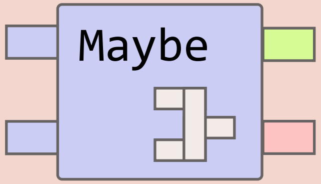

f :: a -> b -> c



y ::



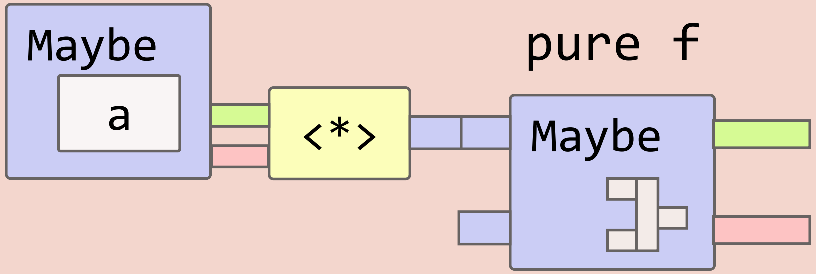

pure f

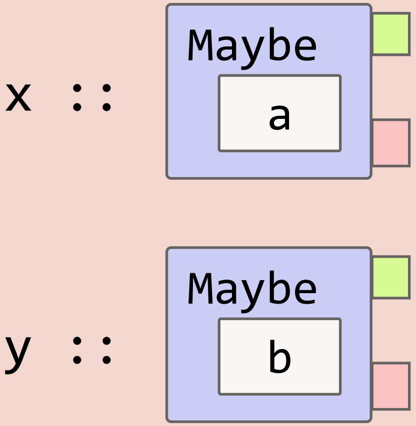



```
pure f :: Maybe (a -> b -> c)
```

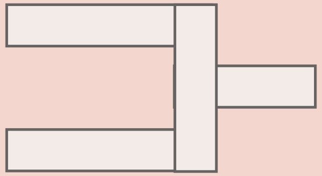
```
pure f <*> x :: Maybe (b -> c)
```

x ::

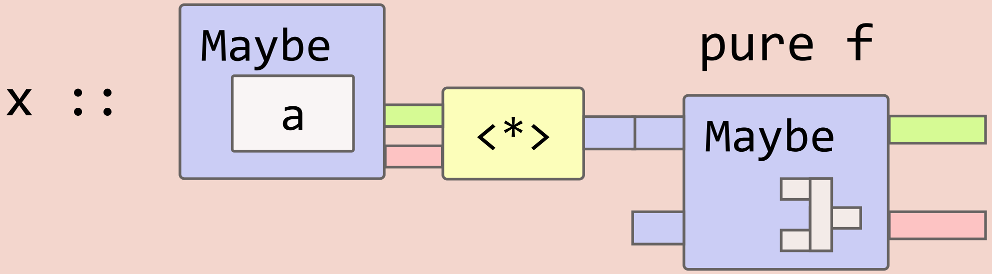




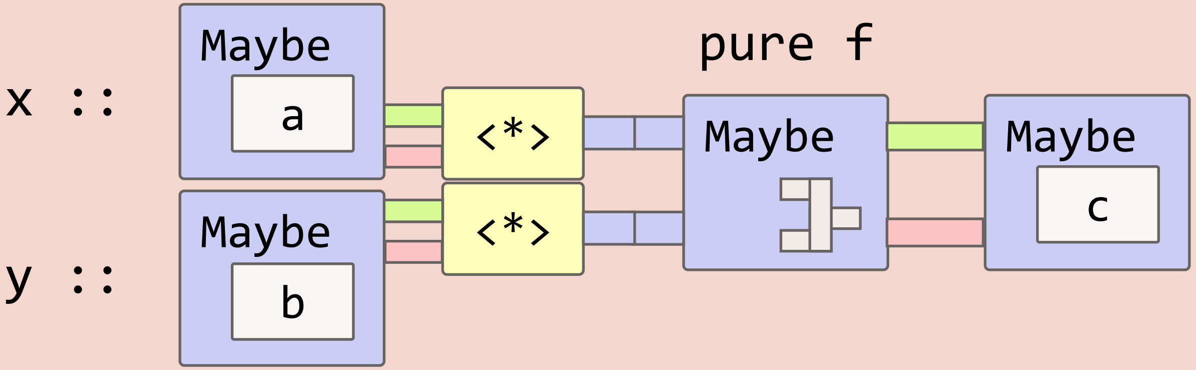

`f :: a -> b -> c`



```
pure f <*> x :: Maybe (b -> c)
```




```
pure f <*> x <*> y :: Maybe c
```



```haskell
pure f <*> x :: Maybe (b -> c)
```

If you noticed carefully, `pure f <*> x == fmap f x`

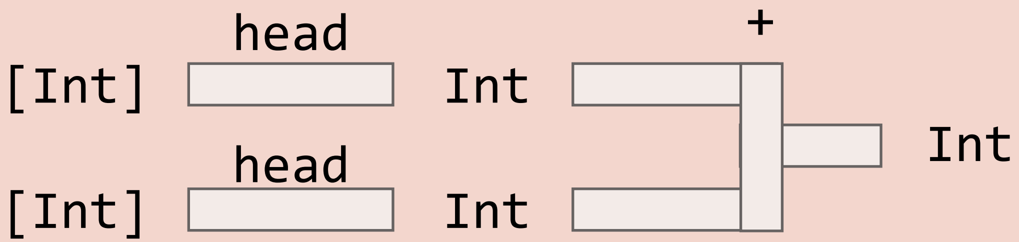
```haskell
pure f <*> x == Just f <*> x
             == case x of
                  Just y -> Just $ f y
                  Nothing -> Nothing
             == fmap f x
```

This is a natural consequence of the applicative laws

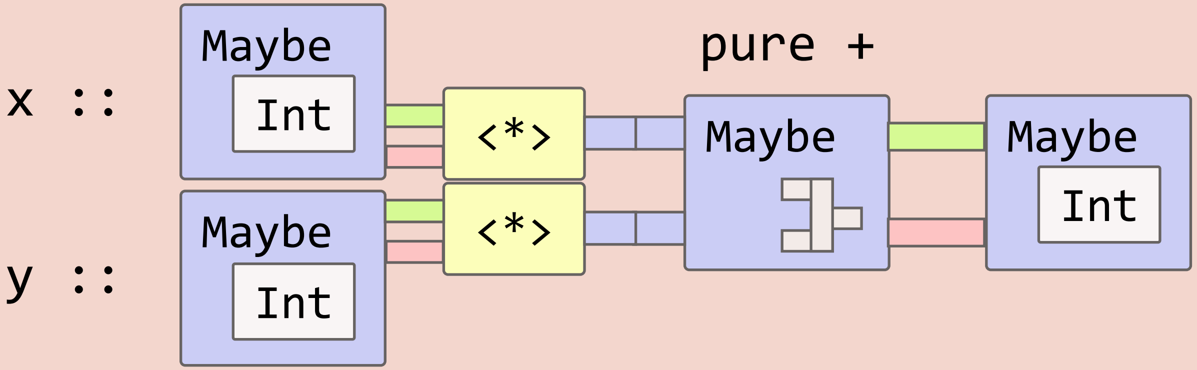
```haskell
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap

pure f <*> x <*> y == fmap  f x <*> y
                   == (<$>) f x <*> y
                   ==   f <$> x <*> y
```

Therefore, we can rewrite `pure f <*> x <*> y` as `f <$> x <*> y`

```haskell
x = head [1, 2, 3]
y = head [4, 5, 6]
z = x + y -- 5
```




```haskell
x = head' [1, 2, 3]
y = head' [4, 5, 6]
z = (+) <$> x <*> y
```



Functions and typeclasses seen so far perform the usual stuff, but **in context**

`fmap :: (a -> b) -> f a -> f b`

Applies a regular function onto a value in context

`pure :: a -> f a`

Puts pure computation in context

`<*> :: f (a -> b) -> f a -> f b`

Applies a function in context on an argument in context

`f x` becomes `fmap f x` or `pure f <*> x` if `x` is now in context

`f x` becomes `f <*> x` if both `f` and `x` are now in context

`f x y z` becomes `fmap f x <*> y <*> z` if `x`, `y` and `z` are now in context

# Validation

A common use of applicatives is **validation**

Example: a user has a name, a valid email and a valid salary

```haskell
data Email =
  Email { emailUsername :: String
        , emailDomain :: String   }
  deriving (Eq, Show)

data Salary = SInt Int
            | SDouble Double
  deriving (Eq, Show)

data User =
  User { username :: String
       , userEmail :: Email
       , userSalary :: Salary }
  deriving (Eq, Show)
```

# Validation

Data is stored as `String`s, must **validate** and make sure that the email and salary make sense

Return results in `Maybe` context to express this fact

```haskell
parseEmail :: String -> Maybe Email
parseEmail email = ...

parseSalary :: String -> Maybe Salary
parseSalary s     = ...
```

# Validation

Now let's define a function that parses strings into users!

```
parseUser :: String -- name
             -> String -- email
             -> String -- salary
             -> Maybe User -- user
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s
```

# Validation

Our validation function works just fine!

```
ghci> parseUser "Foo" "yong@qi.com" "1000"
Just (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Nothing
```

# Validation

```
ghci> parseUser "Foo" "yong@qi.com" "1000"
Just (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Nothing
```

However, a user who receives **`Nothing`** may not know what went wrong!

Let's have our functions return an error message instead of **`Nothing`**

```haskell
data Either a b = Left a   -- sad
                | Right b -- happy

instance Functor (Either a) where
    fmap :: (b -> c) -> Either a b -> Either a c
    fmap _ (Left x) = Left x
    fmap f (Right x) = Right (f x)

instance Applicative (Either a) where
    pure :: b -> Either a b
    pure x = Right x
    (<*>) :: Either a (b -> c) -> Either a b -> Either a c
    Left f <*> _ = Left f
    _ <*> Left x = Left x
    Right f <*> Right x = Right (f x)
```

Let us change the context of the computation our validators perform

```haskell
parseEmail :: String -> Either String Email
parseEmail email =
    if ... then
        Left $ "error: " ++ email ++ " is not an email"
    else
        Right $ Email ...

parseSalary :: String -> Either String Salary
parseSalary salary =
    if ... then
        Left $ "error: " ++ salary ++ " is not a number"
    else
        Right $ SInt ...
```

Our `parseUser` function doesn't need to change (other than its notion of computation) because we are using the same but overloaded applicative operators!

```haskell
parseUser :: String -- name
          -> String -- email
          -> String -- salary
          -> Either String User -- user
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s
```

```
ghci> parseUser "Foo" "yong@qi.com" "1000"
Right (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Left "error: yong is not an email"
ghci> parseUser "Foo" "yong@qi.com" "x"
Left "error: x is not a number"
```

Our validation function works better! But...

```
ghci> parseUser "Foo" "abc" "x"
Left "error: abc is not an email"
```

Both parsing email and salary fail, but the only error we report is from the email!

```haskell
instance Applicative (Either a) where
    ...
    Left f <*> _ = Left f
    _ <*> Left x = Left x
    Right f <*> Right x = Right (f x)
```

If both **`Either`**s are **`Left`**s, then only the left **`Left`** (lol) is preserved

We need a new data structure that allows us to combine two lefts together, so all error messages are reported!

Redefine **`Either`** as a new ADT called **`Validation`**, everything up to its **`Functor`** definition can be the same (we only need a different instance of **`Applicative`**)

```
data Validation err a = Success a
                      | Failure err

instance Functor (Validation err) where
    fmap _ (Failure e) = Failure e
    fmap f (Success x) = Success (f x)
```

Our error type can be anything that can be combined in the obvious way, this way we can accumulate errors together

$$E_1 \oplus (E_2 \oplus E_3) = (E_1 \oplus E_2) \oplus E_3$$

Make the error type a **Semigroup**!

A semigroup must have an associative binary operation that receive and return itself

```
class Semigroup a where
    (<>) :: a -> a -> a
```

As long as our error is a semigroup, we can freely combine them and know the combination will be in the most natural way

Now we can define our Applicative instance for Validations

```haskell
instance Semigroup err => Applicative (Validation err) where
    pure = Success
    Failure l <*> Failure r = Failure (l <> r)
    Failure l <*> _ = Failure l
    _ <*> Failure r = Failure r
    Success f <*> Success x = Success (f x)
```

Notice in the double failure case, the errors are combined with the semigroup operation

Now we need to think of an appropriate error type for our user validation function

```haskell
instance Semigroup [a] where
    (<>) = (++)
```

Lists together with concatenation form a semigroup!

Let us change the context of the computation our validators perform

```
parseEmail :: String -> Validation [String] Email
parseEmail email =
    if ... then
        Failure ["error: " ++ email ++ " is not an email"]
    else
        Success $ Email ...

parseSalary :: String -> Validation [String] Salary
parseSalary salary =
    if ... then
        Failure ["error: " ++ salary ++ " is not a number"]
    else
        Success $ SInt ...
```

Our `parseUser` function doesn't need to change (other than its notion of computation) because we are using the same but overloaded applicative operators!

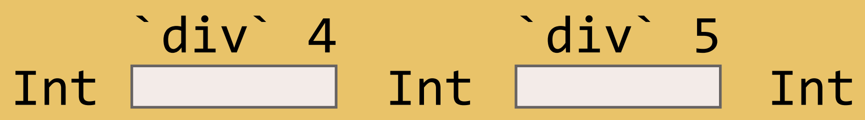
```haskell
parseUser :: String -- name
          -> String -- email
          -> String -- salary
          -> Validation [String] User -- user
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s
```

```
ghci> parseUser "Foo" "yong@qi.com" "1000"
Success (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Failure ["error: yong is not an email"]
ghci> parseUser "Foo" "yong@qi.com" "x"
Failure ["error: x is not a number"]
ghci> parseUser "Foo" "abc" "x"
Failure ["error: abc is not an email",
         "error: x is not a number"]
```

Our validation function works much better!

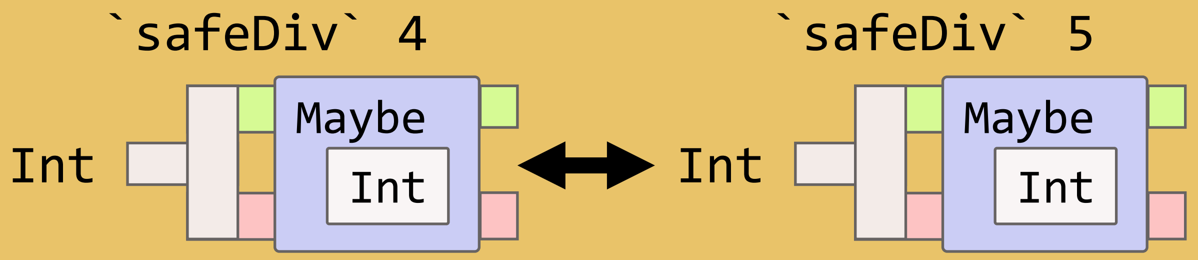
# Composing Railways

```haskell
x = 123
y = (`div` 4) x
z = (`div` 5) y -- 6
```




Composing regular functions is straightforward, but what if our functions are branching railways? How do we compose them?

```haskell
safeDiv x y :: Int -> Maybe Int
safeDiv x 0 = Nothing
safeDiv x y = div x y

x = 123
y = (`safeDiv` 4) x
z = (`safeDiv` 5) y -- ???
```



# Monads

Not important!

A monad on a category $\mathcal{C}$ is a monoid object in the (monoidal) category of endofunctors of $\mathcal{C}$.
A monad $(M, \mu, \eta)$ on $\mathcal{C}$ is an endofunctor $M: \mathcal{C} \to \mathcal{C}$ equipped with a natural transformation $\mu: M^2 \to M$ and natural transformation $\eta: 1_{\mathcal{C}} \to M$ such that the following diagrams commute:

$$\begin{tikzcd} M^3 \arrow[r, "M\mu"] \arrow[d, "\mu M"'] & M^2 \arrow[d, "\mu"] \\ M^2 \arrow[r, "\mu"'] & M \end{tikzcd}$$

$$\begin{tikzcd} M \arrow[r, "M\eta"] \arrow[d, "\eta M"'] \arrow[dr, equal] & M^2 \arrow[d, "\mu"] \\ M^2 \arrow[r, "\mu"'] & M \end{tikzcd}$$

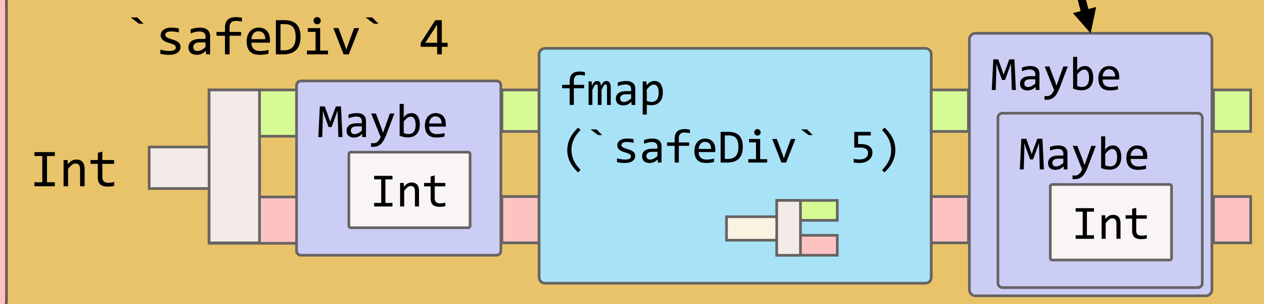
# Monads

A monad is an applicative that has an obvious way to collapse from a nested type into a flat type

```
join :: Monad m => m (m a) -> m a
```

If `Maybe` is a monad, we can collapse `Maybe (Maybe Int)` into `Maybe Int` in an obvious way

```haskell
y :: Maybe Int
y = safeDiv 123 4
z :: Maybe (Maybe Int)
z = fmap (`safeDiv` 5) y
```



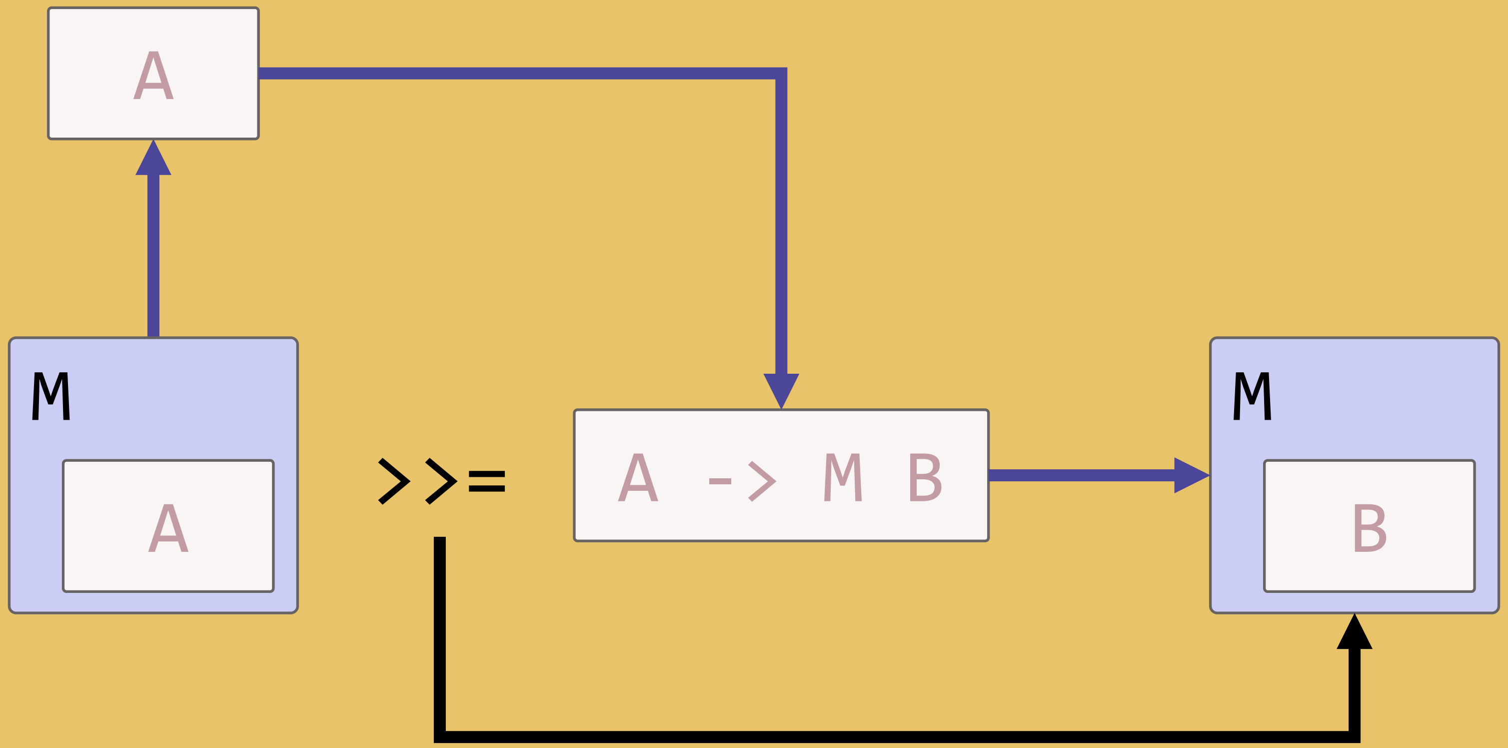
# Monads

```haskell
class Applicative m => Monad m where
    return :: a -> m a -- same as pure
    (>>=) :: m a -> (a -> m b) -> m b
    -- recall:
    --  fmap :: (a -> b) -> m a -> m b
```

`>>=` is called the monadic bind or `flatMap` in other languages, supports **composition in context**

```haskell
instance Monad Maybe where
    return = pure
    Nothing >>= _ = Nothing
    Just x >>= f  = f x
```

# Intuition: >>=



# Map vs >>=

```
ghci> fmap (`safeDiv` 1) (Just 1)
Just (Just 1)
ghci> Just 1 >>= (`safeDiv` 1)
Just 1
```

# Composition in Context

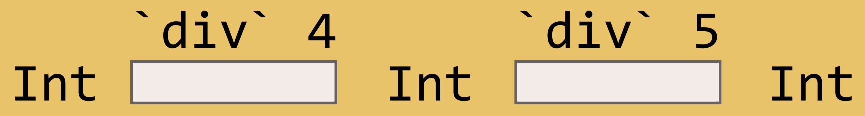
>>= supports **composition in context**, a way to perform **overloading on sequencing**

```haskell
f, g, h :: Int -> Int
a = 123
b = f a -- do f
c = g b -- AND THEN do g
d = h c -- AND THEN do h
```

```haskell
f, g, h :: Int -> Maybe Int
a = 123
b = f a -- do f
c = b >>= g -- AND THEN do g
d = c >>= h -- AND THEN do h
```

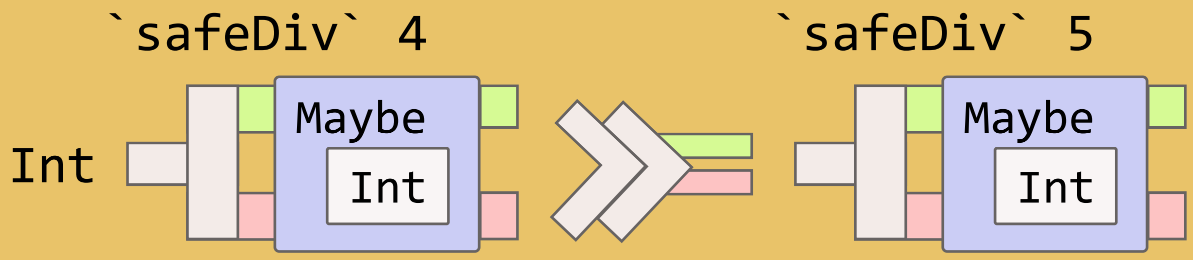
# Composition in Context

```haskell
x = 123
y = (`div` 4) x
z = (`div` 5) y -- 6
```




Use >>= to compose `safeDiv`!

```haskell
safeDiv x y :: Int -> Maybe Int
safeDiv x 0 = Nothing
safeDiv x y = div x y

x = 123
y = (`safeDiv` 4) x
z = y >>= (`safeDiv` 5)
```



# Overloading Sequencing

Lists are also monads; support composition in context

```
instance Monad [] where
    return x = [x]
    [] >>= f = []
    (x : xs) >>= f = f x ++ (xs >>= f)
```

```
ghci> fmap (\x -> [x, x + 1]) [1, 3]
[[1, 2], [3, 4]]

ghci> [1, 3] >>= (\x -> [x, x + 1])
[1, 2, 3, 4]
```

```haskell
cartesian_product :: [a] -> [b] -> [(a, b)]
cartesian_product xs ys = xs >>= (\x ->
                          ys >>= (\y ->
                          return (x, y)))
```

Take each `x` in `xs` and produce every result from the following in a list:

- Take each `y` in `ys` and produce every result from the following in a list:
  - Produce `[(x, y)]`

```
ghci> cartesian_product [1,2] [3]
[(1,3),(2,3)]
```

What if we rewrite `x >>= (\y -> E)` as `y <- x; E`?

```
xs >>= (\x ->
ys >>= (\y ->
return (x, y)))
```

What if we rewrite `x >>= (\y -> E)` as `y <- x; E`?

```
x <- xs
ys >>= (\y ->
return (x, y))
```

What if we rewrite `x >>= (\y -> E)` as `y <- x; E`?

```
x <- xs
y <- ys
return (x, y)
```

This is called do notation! Notice anything similar?

```haskell
cartesian_product :: [a] -> [b] -> [(a, b)]
cartesian_product xs ys = xs >>= (\x ->
                          ys >>= (\y ->
                          return (x, y)))
```

```haskell
cartesian_product :: [a] -> [b] -> [(a, b)]
cartesian_product xs ys
  = do x <- xs
       y <- ys
       return (x, y)
```

```haskell
cartesian_product :: [a] -> [b] -> [(a, b)]
cartesian_product xs ys
  = do x <- xs
       y <- ys
       return (x, y)
```

```python
# Python
def cartesian_product(xs, ys):
    for x in xs:
        for y in ys:
            yield (x, y)
```

We have just recovered imperative programming with monads and **do** notation!

Most importantly… **do** notation works with **any monad**!

# Imperative Programming with Monads

```
[]
```

```haskell
pairs :: [a] -> [(a, a)]
pairs ls = do
    x <- ls
    y <- ls
    return (x, y)
```

```haskell
z :: Maybe Int
z =
 do y <- 123 `safeDiv` 4
    y `safeDiv` 5
```

```
Maybe
```

```
Either a
```

```haskell
parseUser :: String
    -> String
    -> String
    -> Either String User
parseUser name email salary = do
    e <- parseEmail email
    s <- parseSalary salary
    return $ User name e s
```

In other languages, language itself defines keywords like for, while, if/else that each define what "and then" means

```
cartesian_product :: m a -> m b => m (a, b)
cartesian_product xs ys = do
  = do x <- xs
       y <- ys
       return (x, y)
```

In FP, monads decide what "**and then**" means—you get to define your own monads!

```
ghci> cartesian_product [1, 2] [3]
[(1, 3), (2, 3)]
ghci> cartesian_product (Just 1) (Just 2)
Just (1, 2)
ghci> cartesian_product (Just 1) Nothing
Nothing
ghci> cartesian_product (Right 1) (Right 2)
Right (1, 2)
-- getLine is like input() in Python
ghci> cartesian_product getLine getLine
alice -- user input
bob   -- user input
("alice","bob")
```

# Key Takeaways

- Instead of functions with side-effects, pure functions can emulate the desired effects (like branching railways) using the right data structures as notions of computation
- We can operate in context using regular functions when the context is a functor
- We can combine context when the context is an applicative
- We can compose functions in context sequentially when they are monads

# Railway Pattern in Python

# Railway Pattern

Railway pattern can be used in many languages as long as you have the right data structures

Types like `Maybe`, lists etc. are widespread in popular industry languages

```
import java.util.Optional;
public class Main {
  static Optional<Integer> safeDiv(int num, int den) {
    if (den == 0) {
      return Optional.empty();
    }
    return Optional.of(num / den);
  }
  public static void main(String[] args) {
    Optional<Integer> x = safeDiv(123, 4)
        .flatMap(y -> safeDiv(y, 5))
        .flatMap(z -> safeDiv(z, 2));
    x.ifPresent(System.out::println);
  }
}
```

# Railway Pattern

Define the right data structures and supporting methods (that abide by the relevant laws) and you can begin writing pure functions with notions of computation in Python!

```
type Maybe[A] = Just[A] | Nothing
class Just[A]:
    val: A
    def map(self, f):
        return Just(f(self.val))
class Nothing:
    def map(self, f):
        return self
```

# Thank you

**Foo** Yong Qi

yongqi@nus.edu.sg

https://yongqi.foo/