Functor (1A)

Young Won Lim
8/1/17

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Typeclasses

Typeclasses are like interfaces

defines some behavior
  comparing for *equality*
  comparing for *ordering*
  enumeration

Instances of that typeclass
  types <u>possessing</u> such behavior

Such *behavior* is defined by
- **function definition**
- **type declaration**

(**==**) :: a -> a -> Bool   - a type declaration
x == y = not (x /= y)   - a function definition

a **type** is an **instance** of a **typeclass** implies
  the functions defined (implemented)
  by the typeclass with that type can be used

No relation with classes in Java or C++

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Car Type Example
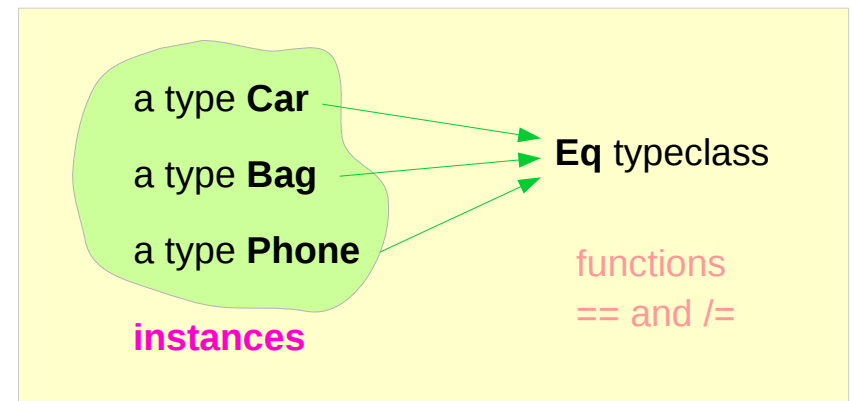
the Eq typeclass

defines the functions **==** and **/=**

a type Car

comparing two cars c1 and c2 with the equality function ==

The Car type is an **instance** of Eq **typeclass**

Instances : various types

Typeclass : a group or a class of these similar types

a type **Car**

a type **Bag**

a type **Phone**

Eq typeclass

functions
== and /=

instances

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

TrafficLight Type Example (1)

```
class Eq a where
    (==) :: a -> a -> Bool          - a type declaration
    (/=) :: a -> a -> Bool          - a type declaration
    x == y = not (x /= y)           - a function definition
    x /= y = not (x == y)           - a function definition
```

data TrafficLight = Red | Yellow | Green

```
instance Eq TrafficLight where
    Red    == Red    = True
    Green  == Green  = True
    Yellow == Yellow = True
    _      == _      = False
```

ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

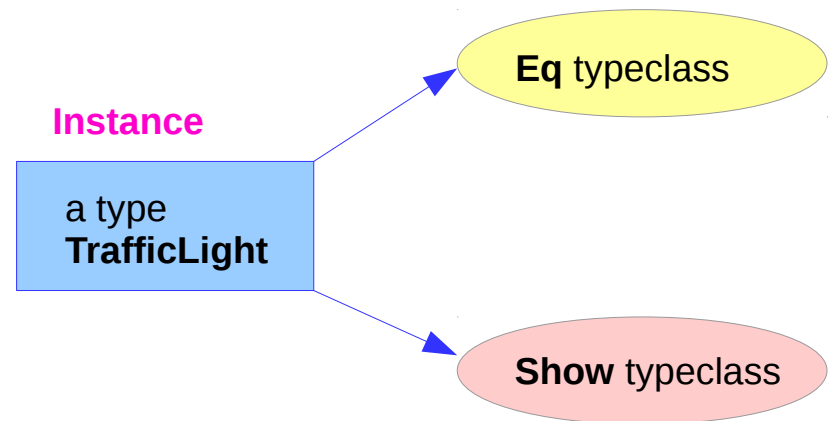# **TrafficLight** Type Example (2)

```
class Show  a where
    show :: a  -> String
    * * *
```

- a type declaration

data TrafficLight = Red | Yellow | Green

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Instance

Eq typeclass

a type
TrafficLight

Show typeclass

ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Class Constraints

class (**Eq** a) => **Num** a where
  ...

class Num a where
  ...

class constraint on a class declaration
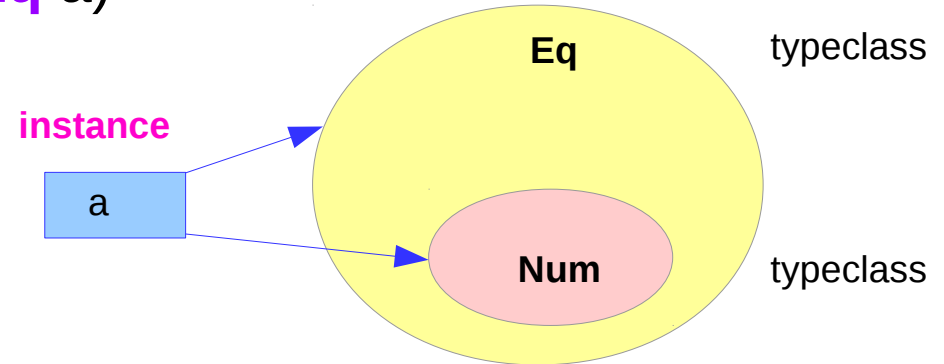
    an instance of **Eq**
    <u>before</u> being an instance of **Num**

the required function bodies can be defined in
- the class declaration
- an instance declarations,

    we can safely use == because a is a part of **Eq**

(**Eq** a) =>

instance

a

Eq   typeclass

Num   typeclass

Num   : a subclass of **Eq**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Class Constraints : class & instance declarations

class constraints in **class declarations**

      to make a typeclass a **subclass** of another typeclass

   class (**Eq** a) => **Num** a where

     …

class constraints in **instance declarations**

     to express **requirements** about the contents of some type.

   instance (**Eq** x, **Eq** y) => **Eq** (**Pair** x y) where
     Pair x0 y0 == Pair x1 y1 **=** x0 == x1 && y0 == y1

Derived instance

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/07-type-classes.php

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Class constraints in instance declaration examples

instance (**Eq** m) => **Eq** (**Maybe** m) where

    Just x   == **Just** y   =   x == y   ⬅   **Eq** m

    Nothing == **Nothing**   =   True

         _ == _   =   False

instance (**Eq** x, **Eq** y) => **Eq** (**Pair** x y) where

    Pair x0 y0 **==** Pair x1 y1 **=** x0 **==** x1 && y0 **==** y1

          ⬆   ⬆   ⬆

    Eq (**Pair** x y)   **Eq** x   **Eq** y

A Concrete Type and a Type Constructor

a   : a concrete type

Maybe   : <u>not</u> a concrete type

             : a type constructor that takes one parameter

              produces a concrete type.

Maybe a : a concrete type

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

the **Functor typeclass** is basically for things that can be *mapped over*

ex) mapping over lists

the list type is part of the Functor typeclass

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

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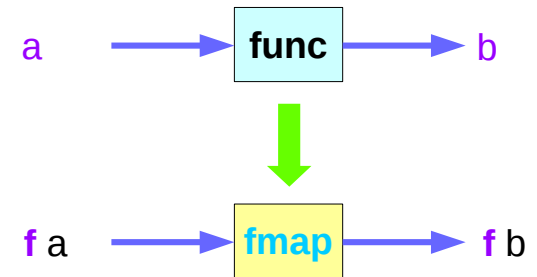
The **Functor typeclass**

    defines one function, **fmap**

    no default implementation

the **type variable f**

    not a concrete type (a concrete type can hold a value)

    a **type constructor** taking one type parameter

        Maybe Int : a concrete type

        Maybe   : a type constructor that takes one type as the parameter

a   → **func** → b

f a → **fmap** → f b

function fmap
function func
type constructor f

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Function **map** & **fmap**

```
class Functor f where

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

fmap takes
- a **function** from one type to another (a -> b)
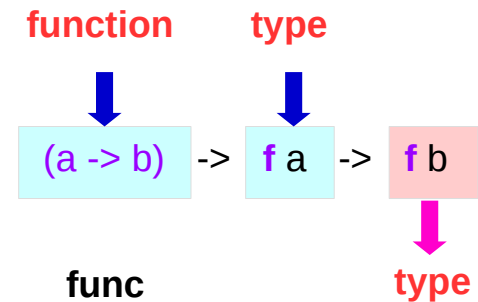- a **Functor f** applied with **one type** (f a)

fmap returns
- a **Functor f** applied with **another type** (f b)

function　　　**type**

$$(a \rightarrow b) \quad \text{->} \quad f\ a \quad \text{->} \quad f\ b$$

func　　　　　　　**type**

```
    map :: (a -> b) -> [a] -> [b]
```

map takes
- a function from one type to another   (* 2)
- take a list of one type   [1, 2, 3]
- returns a list of another type   [2, 4, 6]

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

List : an instance of the Functor typeclass

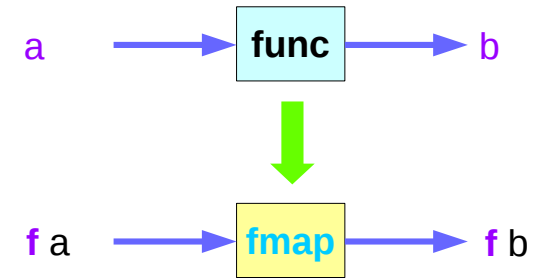**class** **Functor** **f** where

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

    map :: (a -> b) -> **[a]** -> **[b]**

map is just a **fmap** that works only on **list**s

a list is an **instance** of the Functor typeclass.

instance **Functor** **[ ]** where

    fmap = **map**

f :   a type constructor that takes one type

[ ] :   a type constructor that takes one type

[a] : a concrete type ([Int], [String] or [[String]])

function **fmap**
function **func**
type constructor f

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass
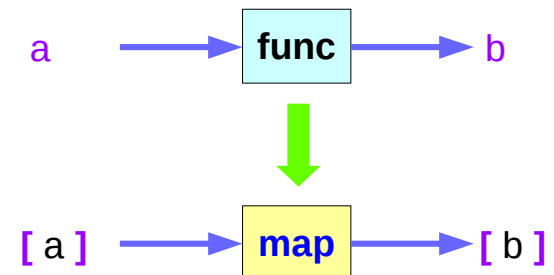
List Examples

class **Functor** **f** where
    fmap :: (a -> b) -> **f** a -> **f** b
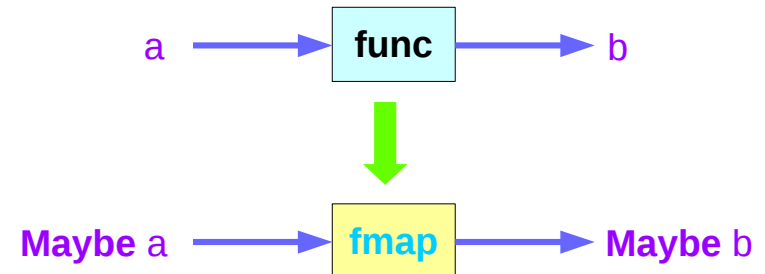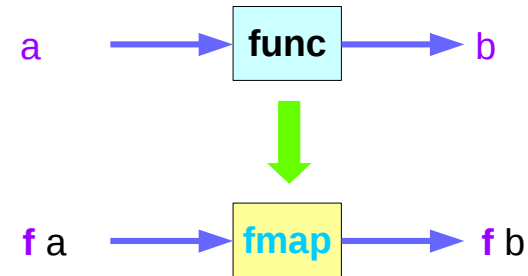
    map :: (a -> b) -> **[a]** -> **[b]**

instance **Functor** **[ ]** where
    fmap = **map**

map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]

```
1 ───→  *2  ───→ 2

        ↓

[1,2,3] ───→ map ───→ [2,4,6]
```

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass
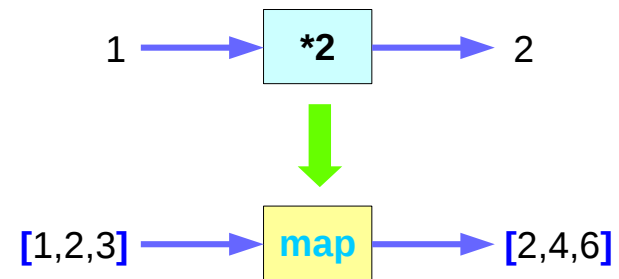
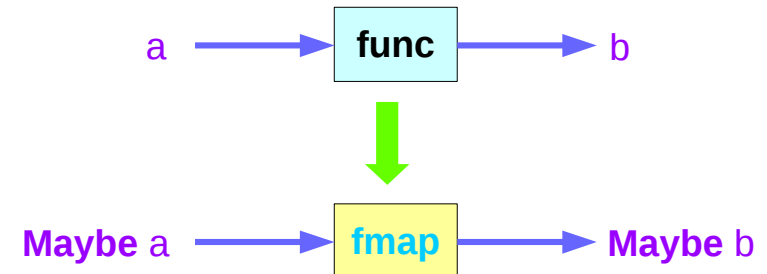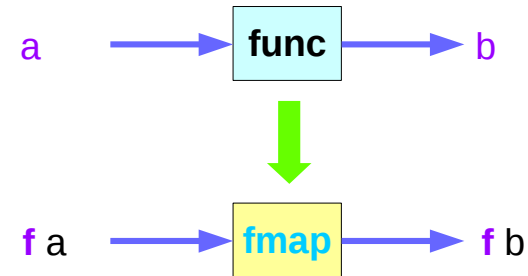# Maybe : an instance of the Functor typeclass

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

```
instance Functor Maybe where
    fmap func (Just x) = Just (func x)
    fmap func Nothing = Nothing
```

f	⟷	**Maybe**
f a	⟷	**Maybe** a
f b	⟷	**Maybe** b

a → **func** → b

↓ fmap

f a → **fmap** → **f** b

a → **func** → b

↓ fmap

Maybe a → **fmap** → **Maybe** b

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Maybe : a type constructor

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

```
instance Functor Maybe where
    fmap func (Just x) = Just (func  x)
    fmap func Nothing = Nothing
```

f : a **type variable**

f : a **type constructor** taking <u>one</u> type parameter
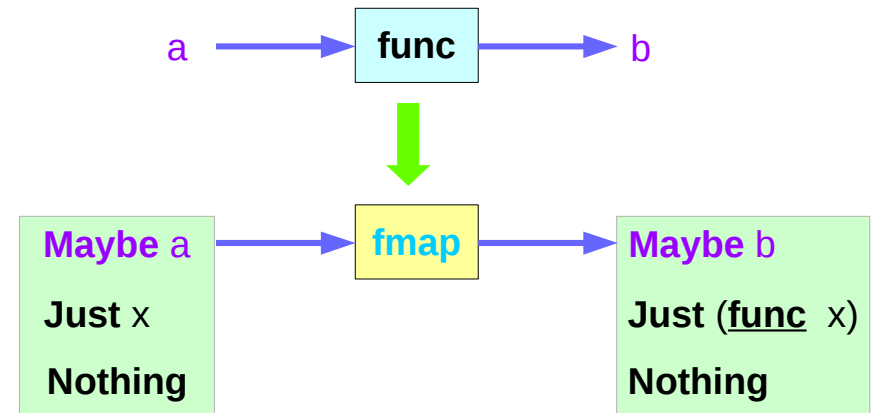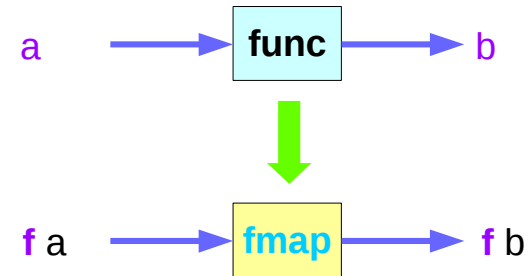
Maybe : an **instance** of **Functor** typeclass

a → **func** → b

f a → **fmap** → **f** b

a → **func** → b

Maybe a → **fmap** → **Maybe** b

Maybe : an argument to fmap, together with a

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

```
instance Functor Maybe where
    fmap func (Just x) = Just (func  x)
    fmap func Nothing = Nothing
```

```
fmap :: (a -> b) -> f a -> f b
fmap func (Just x) = Just (func  x)
fmap func Nothing = Nothing
```

a ⟶ **func** ⟶ b

f a ⟶ **fmap** ⟶ **f** b

a ⟶ **func** ⟶ b

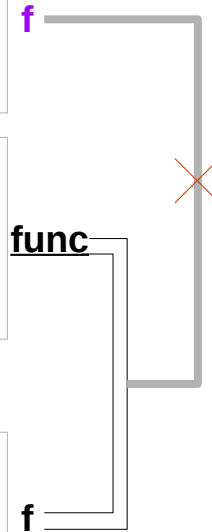| **Maybe** a	**fmap**	**Maybe** b
Just x		**Just** (**func** x)
Nothing		**Nothing**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Maybe : fmap takes a function

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

```
instance Functor Maybe where
    fmap func (Just x) = Just (func  x)
    fmap func Nothing = Nothing
```

```
instance Functor Maybe where
    fmap  f  (Just x) = Just ( f  x)
    fmap  f  Nothing = Nothing
```
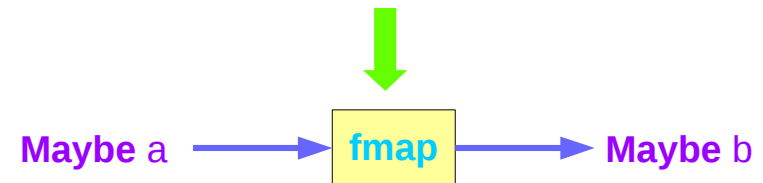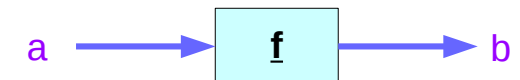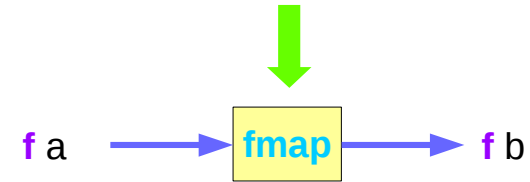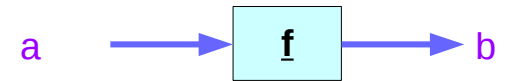
f is different from the type constructor **f**

func : a -> b

 f : a -> b

Maybe Examples (1)

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

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing

a → **func** → b

f a → **fmap** → f b

200 → *2 → 400

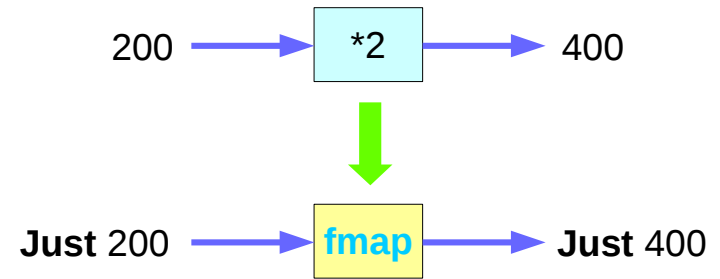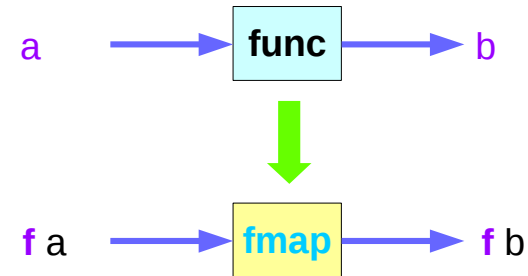**Just** 200 → **fmap** → **Just** 400

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Maybe Examples (2)

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

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

a → **func** → b

f a → **fmap** → f b

ghci> fmap (++ "BBB") (Just "AAA")
Just "AAABBB"
ghci> fmap (++ "BBB") Nothing
Nothing

"AAA" → (++ "BBB") → "AAABBB"

Just "AAA" → **fmap** → **Just** "AAABBB"

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass
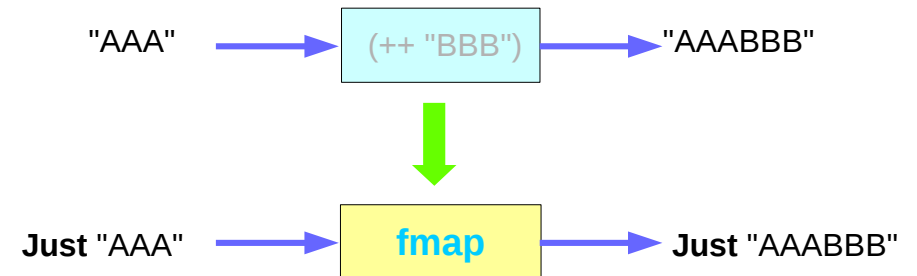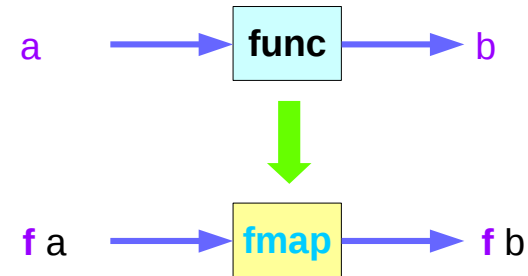
Maybe as a functor

Functor typeclass:
- transforming one <u>type</u> to another
- transforming <u>operations</u> of one type to those of another

Maybe a is an <u>instance</u> of a **functor** type class

Functor provides **fmap** method
      maps functions of the base type (such as Integer)
      to *functions* of the lifted type (such as Maybe Integer).

Maybe as a functor

A *function* **f** transformed with **fmap**
can work on a **Maybe** value

case maybeVal **of**
  Nothing -> **Nothing**   -- there is nothing, so just return Nothing
  Just val -> **Just** (**f** val)   -- there is a value, so apply the function to it

   father :: Person -> **Maybe** Person
   mother :: Person -> **Maybe** Person

      f :: Int   -> Int
  fmap f :: **Maybe** Integer -> **Maybe** Integer

a **Maybe Integer** value: **m_x**

fmap f   **m_x**

Transforming operations

Functor provides **fmap** method
  maps *functions* of the base type (such as Integer)
  to *functions* of the lifted type (such as Maybe Integer).

a ⟶ **f** ⟶ b

Maybe a ⟶ **fmap** ⟶ **Maybe** b

Int ⟶ **f** ⟶ Int

Maybe Int ⟶ **fmap f** ⟶ **Maybe** Int

https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell
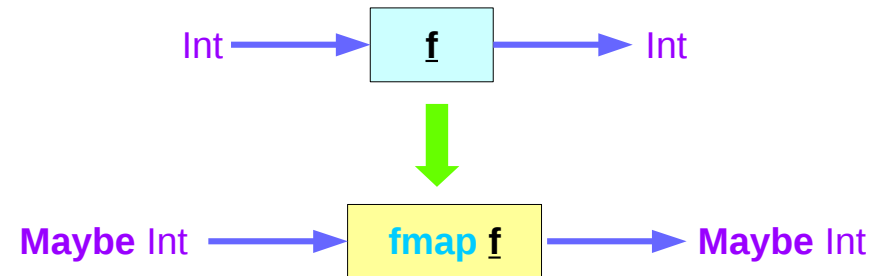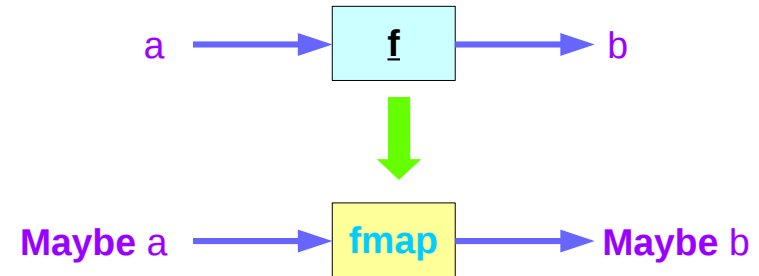
Maybe as a functor

m_x : a Maybe Integer value (Just 101, Nothing, …)
f :: Int -> Int

you can do **fmap** **f** m_x
to apply the function f directly to the Maybe Integer
without worrying whether it is Nothing or not

I

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

instance **Functor** **Maybe** where
    fmap **f** (**Just** x) = **Just** (**f** x)
    fmap **f** **Nothing** = **Nothing**

Maybe as a functor

Can apply a whole chain of
lifted Integer -> Integer functions
to Maybe Integer values
and only have to worry about explicitly checking for Nothing
once when you're finished.

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

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```
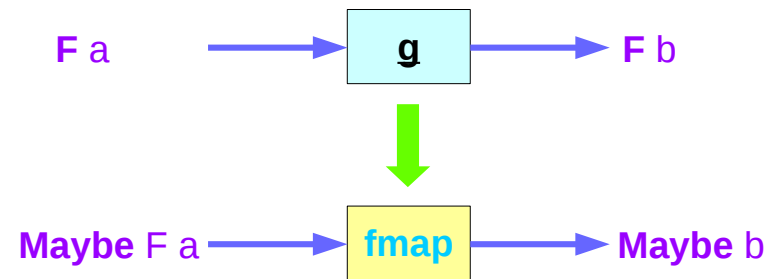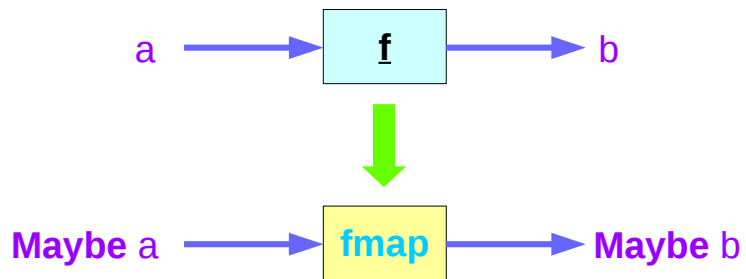
Maybe class

The Maybe type definition

 data Maybe a = Just a | Nothing
   deriving (Eq, Ord)

Maybe is
    an instance of **Eq** and **Ord** (as a base type)
    an instance of **Functor**
    an instance of **Monad**

    For Functor, the fmap function f
        moves inside the Just constructor and
        is identity on the Nothing constructor.

    For Monad,
        the bind operation passes through Just, while
        Nothing will force the result to always be Nothing.

https://wiki.haskell.org/Maybe

Young Won Lim
8/1/17

Maybe as a Monad

f::Int -> Maybe Int
f 0 **= Nothing**
f x **= Just** x

if x==0 then Nothing else Just x

g :: Int -> Maybe Int
g 100 **= Nothing**
g x   **= Just** x

if x==100 then Nothing else Just x

h ::Int -> Maybe Int
h x **= case f** x **of**
        Just n **-> g** n
        Nothing -> Nothing

if f x==Nothing then Nothing else g n

h' :: Int -> Maybe Int
h' x **= do** n **<- f** x
        g n

g (f x)

h & h' give the same results
h 0 = h' 0 = h 100 = h' 100 = Nothing;
h x = h' x = Just x

https://wiki.haskell.org/Maybe

Maybe as a Library Function

When the module is imported   **import Data.Maybe**

maybe :: b->(a->b) -> Maybe a -> b

   Applies the second argument (a->b) to the third Maybe a,
   when it is Just x, otherwise returns the first argument (b).

isJust, **isNothing**
   Test the argument, returing a Bool based on the constructor.

ListToMaybe , **maybeToList**
   Convert to/from a one element or empty list.

mapMaybe
   A different way to filter a list.

https://wiki.haskell.org/Maybe

Maybe as Monad

maybe :: b->(a->b) -> Maybe a -> b
The maybe function takes
      a default value (b),
      a function (a->b), and
      a Maybe value (Maybe a).

If the Maybe value is Nothing,
      the function returns the default value.
Otherwise, it applies the function to the value inside the Just and returns the result.

>>> maybe False odd (Just 3)
True

>>> maybe False odd Nothing
False

Then Operator (**>>**) and **do** Statements

```
putStr "Hello"  >>
putStr " "      >>
putStr "world!" >>
putStr "\n"
```

```
do { putStr "Hello"
   ; putStr " "
   ; putStr "world!"
   ; putStr "\n" }
```
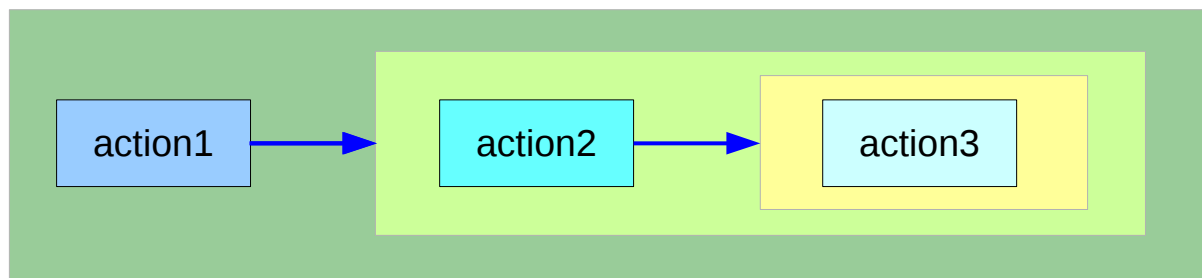
https://en.wikibooks.org/wiki/Haskell/do_notation

```
do { action1
    ; action2
    ; action3 }
```

```
action1 >>
do { action2
    ; action3 }
```

```
do { action1
    ; do { action2
        ; action3 } }
```

```
do { action1
    ; do { action2
        ; do { action3 } } }
```


can **chain** any actions
as long as all of them are
in **the same monad**

https://en.wikibooks.org/wiki/Haskell/do_notation

Bind Operator (**>==**) and **do** statements

The bind operator (**>>=**)

      passes a value (the result of an action or function),

      downstream in the binding sequence.

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```
            anonymous function
            (lambda expression)
            is used

do notation <u>assigns</u> a variable name
to the passed value using the **<-**

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

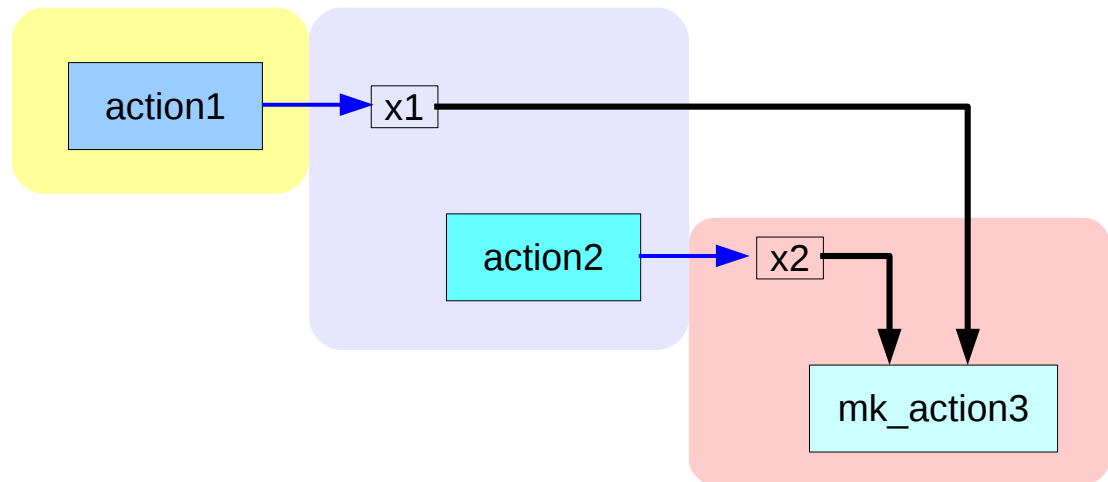https://en.wikibooks.org/wiki/Haskell/do_notation

Translation using the bind operator (**>>=**)

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

action1 **>>=** (\ x1 -> action2 **>>=** (\ x2 -> mk_action3 x1 x2 **)**)

```
action1
  >>=
    (\ x1 -> action2
      >>=
        (\ x2 -> mk_action3 x1 x2 ))
```

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```

Anonymous Function

\x **->** x + 1

(****x **->** x + 1**)** 4
5 :: Integer

(****x y **->** x + y**)** 3 5
8 :: Integer

Lambda Expression

addOne = ****x **->** x + 1

https://wiki.haskell.org/Anonymous_function

Functor Typeclass

instance **Functor IO** where
    fmap f action = **do**
        result **<- action**
        return (**f** result)

instance **Functor Maybe** where
    fmap func (**Just** x) = **Just** (**func** x)
    fmap func Nothing = **Nothing**

http://learnyouahaskell.com/functors-applicative-functors-and-monoids
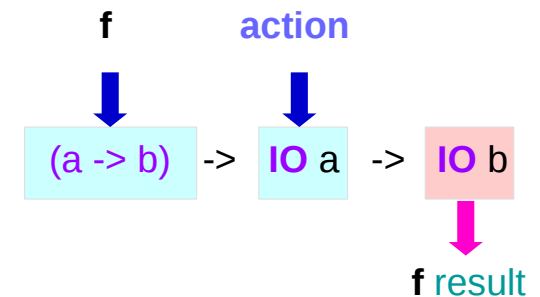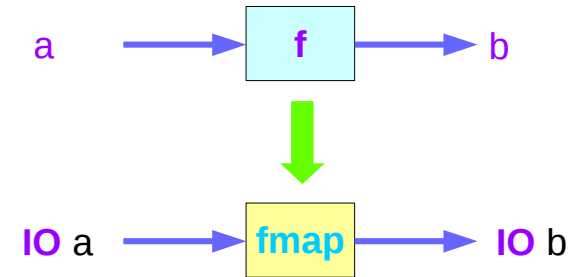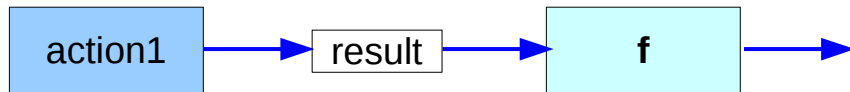
Functor Typeclass

main = **do** line **<- getLine**

    let line' = **reverse** line

    putStrLn $ "You said " ++ line' ++ " backwards!"

    putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"

main = **do** line **<- fmap reverse getLine**

    putStrLn $ "You said " ++ line ++ " backwards!"

    putStrLn $ "Yes, you really said" ++ line ++ " backwards!"

instance Functor IO where

    fmap f action = **do**

        result **<- action**

        return (**f** result)

    fmap reverse getLine = **do**

        result **<- getLine**

        return (**reverse** result)

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

$ Operator

$ operator to avoid parentheses

Anything appearing after $

will take precedence over anything that comes before.

putStrLn (show (1 + 1))

putStrLn (show **$** 1 + 1)

putStrLn **$** show (1 + 1)

putStrLn **$** show **$** 1 + 1

. Operator

. operator to chain functions

putStrLn (**show** (1 + 1))

(1 + 1) is not a function, so the . operator cannot be applied
show can take an Int and return a String.
putStrLn can take a String and return an IO().

(**putStrLn . show**) (1 + 1)

Int → [**show**] → String → [**putStrLn**] → IO()

putStrLn . show $ 1 + 1

Functor Typeclass

instance Functor ((->) r) where
    fmap f g = (\x -> **f** (**g** x))

instance Functor Maybe where
    fmap f (**Just** x) = **Just** (**f** x)
    fmap f Nothing = **Nothing**

A function takes any thing and returns any thing
    g :: **a -> b**
    g :: **r -> a**

fmap :: (a -> b) -> f a -> f b
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
fmap :: (a -> b) -> (r -> a) -> (r -> b)
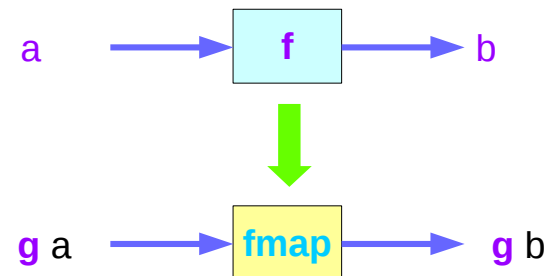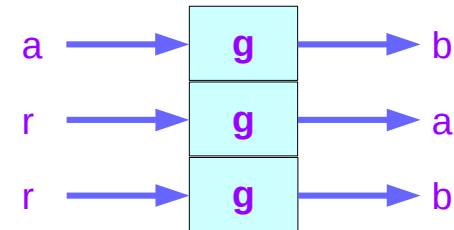
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functor Typeclass

instance Functor ((->) r) where
    fmap f g = (\x -> **f** (**g** x))

instance Functor ((->) r) where
    fmap = (**.**)

instance Functor Maybe where
    fmap f (**Just** x) = **Just** (**f** x)
    fmap f Nothing = **Nothing**

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```
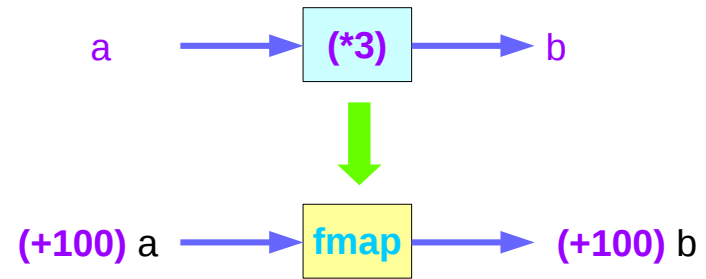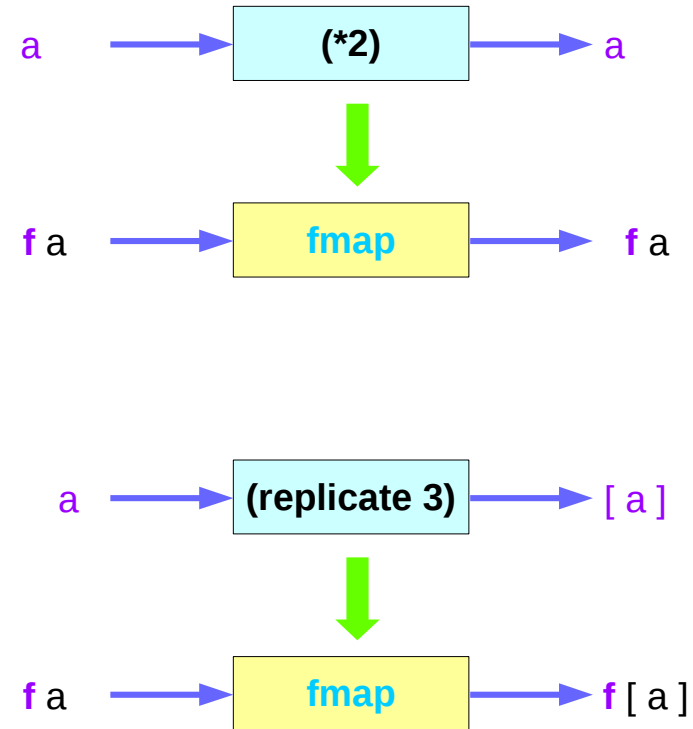
Functor Typeclass

ghci> :t **fmap** (*2)

fmap (*2) :: (Num a, Functor **f**) => **f** a -> **f** a

ghci> :t **fmap** (replicate 3)

fmap (replicate 3) :: (Functor **f**) => **f** a -> **f** [a]

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functor Typeclass

ghci> **fmap** (replicate 3) **[**1,2,3,4**]**
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

ghci> **fmap** (replicate 3) **(Just** 4**)**
Just [4,4,4]

ghci> **fmap** (replicate 3) **(Right** "blah"**)**
Right ["blah","blah","blah"]

ghci> **fmap** (replicate 3) **Nothing**
Nothing

ghci> **fmap** (replicate 3) **(Left** "foo"**)**
Left "foo"

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functor Laws

fmap id = **id**

id :: a -> a
id   x = x

instance **Functor** **Maybe** where
    fmap <u>**func**</u> (**Just** x) = **Just** (<u>**func**</u> x)
    fmap <u>**func**</u> **Nothing** = **Nothing**

instance **Functor** **Maybe** where
    fmap <u>**f**</u> (**Just** x) = **Just** (<u>**f**</u> x)
    fmap <u>**f**</u> **Nothing** = **Nothing**

instance **Functor** **Maybe** where
    fmap <u>**id**</u> (**Just** x) = **Just** (<u>**id**</u> x)
    fmap <u>**id**</u> **Nothing** = **Nothing**

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

a ⟶ [id] ⟶ a

⬇

F a ⟶ [fmap] ⟶ F a

F a ⟶ [id] ⟶ F a
Just x   **Just** x
Nothing   **Nothing**
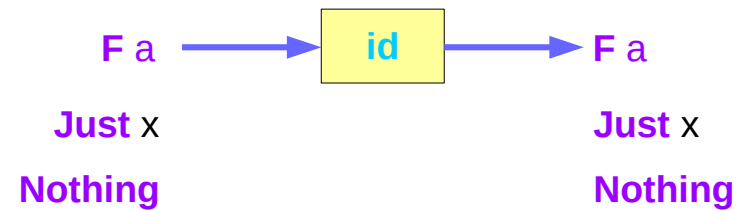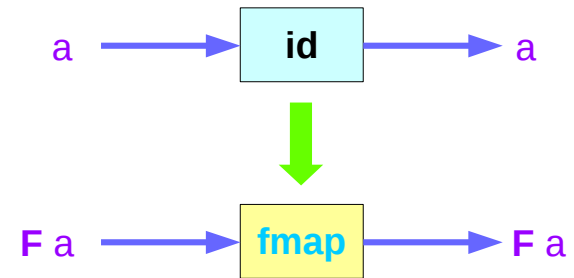
Functor Typeclass

ghci> **fmap** **id** (Just 3)

Just 3

ghci> **id** (Just 3)

Just 3

ghci> **fmap** **id** [1..5]

[1,2,3,4,5]

ghci> **id** [1..5]

[1,2,3,4,5]

ghci> **fmap** **id** []

[]

ghci> **fmap** **id** Nothing

Nothing

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functor Laws

fmap (**f . g**) = **fmap f** . **fmap g**

fmap (**f . g**) **F** = **fmap f** (**fmap g F**)

http://learnyouahaskell.com/functors-applicative-functors-and-monoids
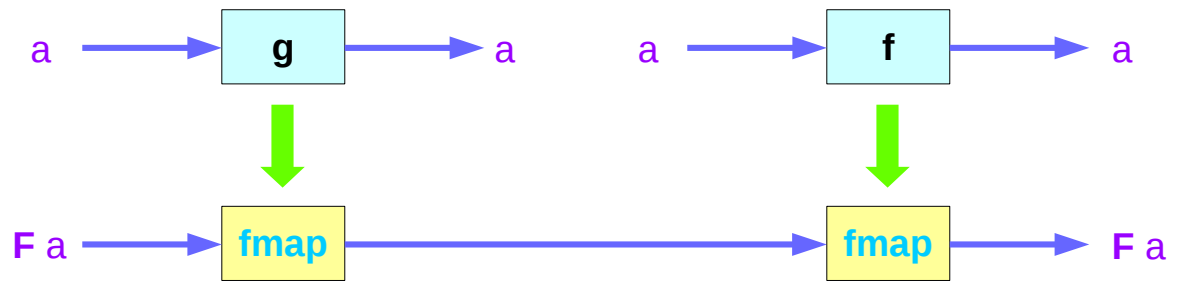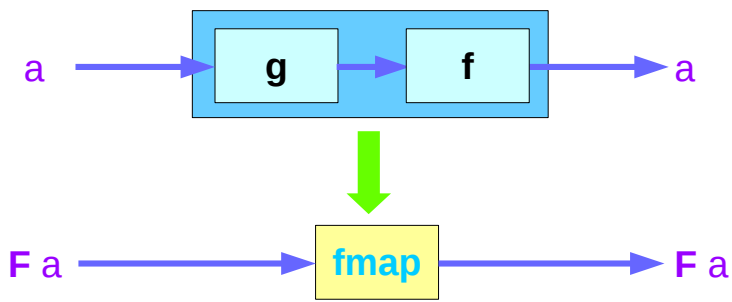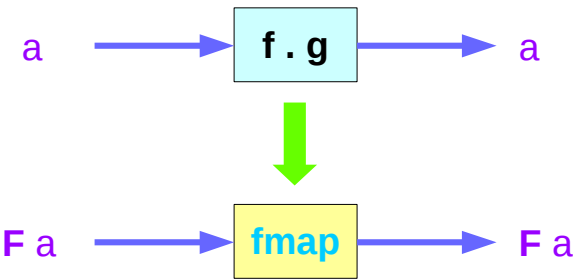
Functor Laws

fmap (**f** . **g**) = **fmap** **f** . **fmap** **g**

fmap (**f** . **g**) **F** = **fmap** **f** (**fmap** **g** **F**)

    instance **Functor** **Maybe** where

        fmap <u>**f**</u> (**Just** x) = **Just** (<u>**f**</u> x)

        fmap <u>**f**</u> **Nothing** = **Nothing**

fmap (**f** . **g**) **Nothing** = **Nothing**

fmap **f** (**fmap** **g** **Nothing**) = **Nothing**

fmap (**f** . **g**) (**Just** x) = **Just** ((**f** . **g**) x) = **Just** (**f** (**g** x))

fmap **f** (**fmap** **g** (**Just** x)) = **fmap** **f** (**Just** (**g** x)) = **Just** (**f** (**g** x))

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf