Functor (1A)

Young Won Lim
12/14/17

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

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

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

Typeclasses and Instances

Typeclasses are like interfaces

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

Instances of that **typeclass**

types possessing such behavior

such *behavior* is defined by

- **function definition**
- function **type declaration** only

a function definition

```
(==) :: a -> a -> Bool
x == y = not (x /= y)
```

- **a type declaration**

a function type

```
(==) :: a -> a -> Bool
```

- **a type declaration**

A function definition can be **overloaded**

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

Typeclasses and Type

Typeclasses are like interfaces

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

Instances of that **typeclass**
types possessing such behavior

a **type** is an **instance** of a **typeclass** implies

the function types declared by the **typeclass**
are defined (implemented) in the **instance**

so that we can use the functions
that the **typeclass** defines with that **type**

No relation with classes in Java or C++

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

A Concrete Type and a Type Constructor

a : a concrete type

Maybe : not a concrete type
: a type constructor that takes one parameter
in order to 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*

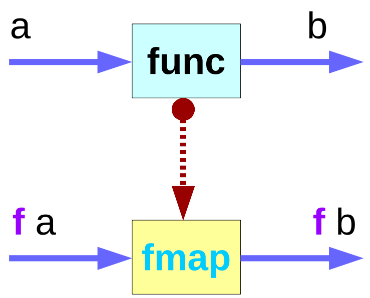
ex) mapping over **list**s

the **list** type is a 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
```



function fmap
function func
type constructor f

The **Functor typeclass**
 defines the function **fmap**
 without a default implementation

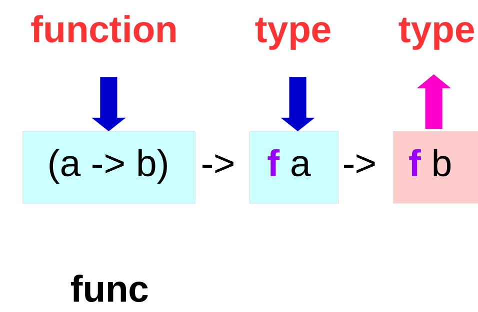
the type variable f
 not a concrete type (a concrete type can hold a value)
 a **type constructor** taking one type parameter

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

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)

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

map takes

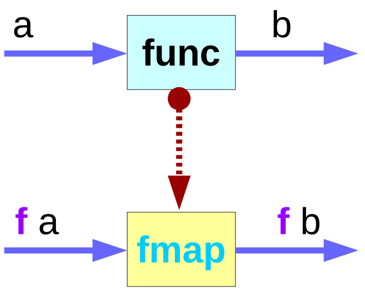
- a function from one type to another (* 2)
- take a list of one type [1, 2, 3]
- returns a list of another type [2, 4, 6]

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]
```

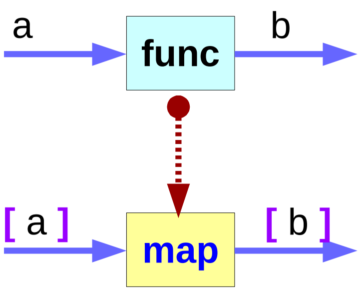


function fmap
function func
type constructor f

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]])

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

List Examples

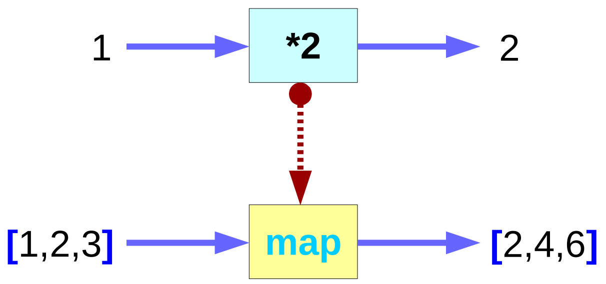
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

```
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]
```

function fmap	map
function func	(*2)
type constructor f	[ ]



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 -> b)	⟺	func

instance : implementing fmap

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

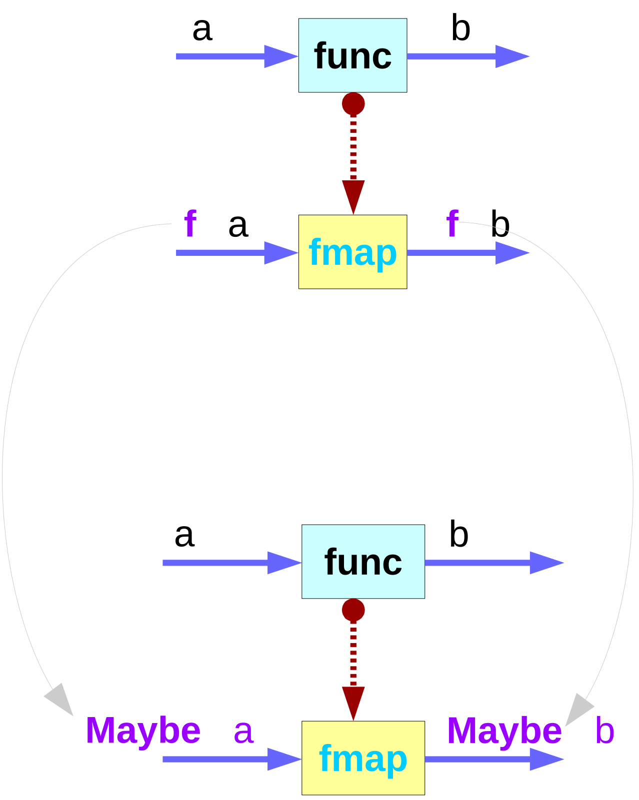
f : a type variable

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

f : a type variable

f ⟺ **Maybe**

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



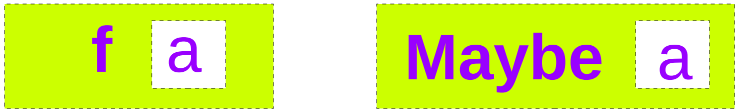
http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

f : a type constructor

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

f : a **type constructor** taking one type parameter

type	f a	⟺	Maybe a	type
type	f b	⟺	Maybe b	type




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

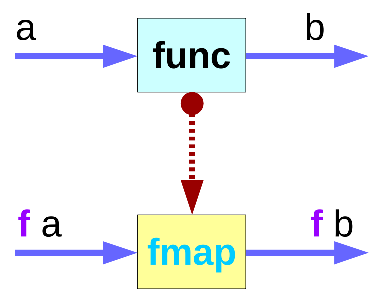
f ~~type~~ Maybe ~~type~~

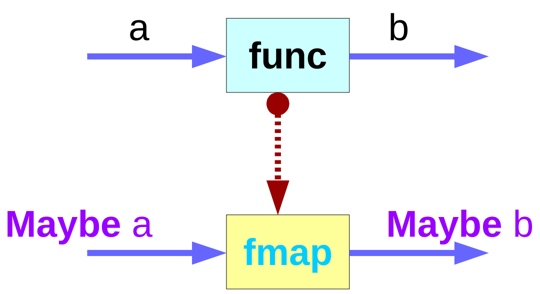
http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

f and Maybe

```
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 one type parameter

Maybe : an **instance** of **Functor** typeclass

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

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

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

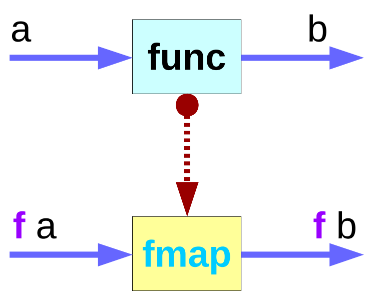
http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Maybe : an argument to fmap, together with a

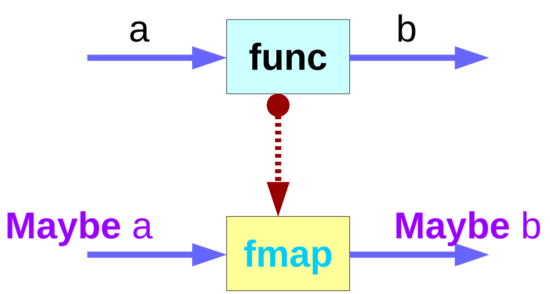
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

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

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



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

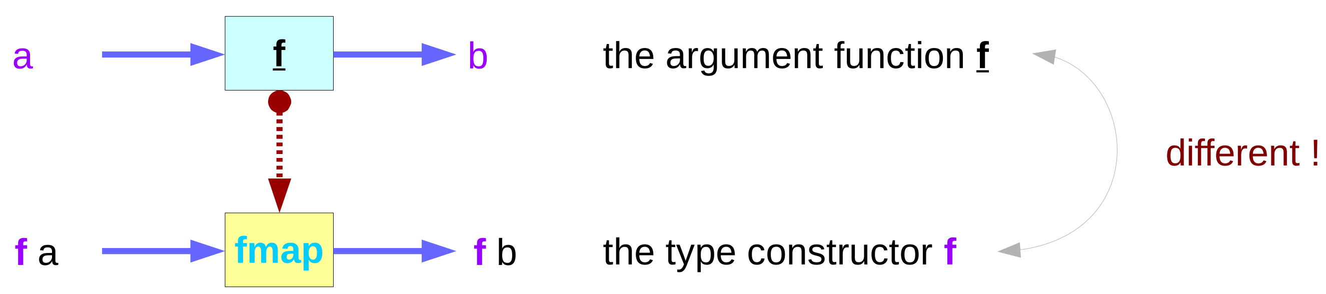


http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

The distinct two f's

```
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
```



An argument **f** to fmap vs. Functor **f**

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

f

f : a **type variable**

f : a **type constructor** taking one type parameter

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

func

‖

f

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

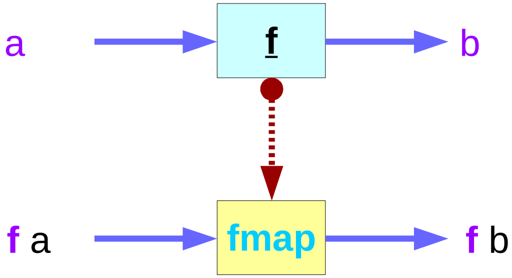
f an argument function to **fmap**

f is different from the type constructor **f**

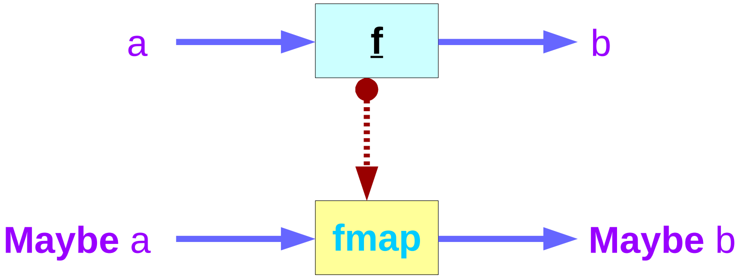
f : a -> b ⟺ **func** : a -> b

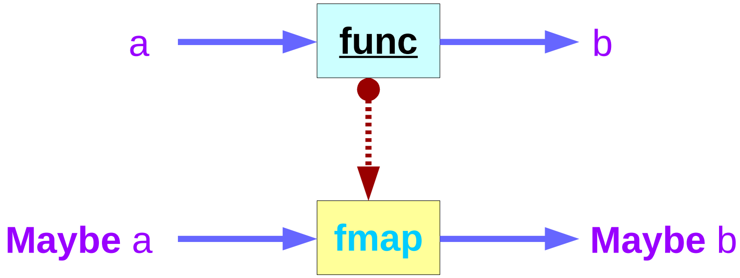
Maybe Functor

Type Class



Instance





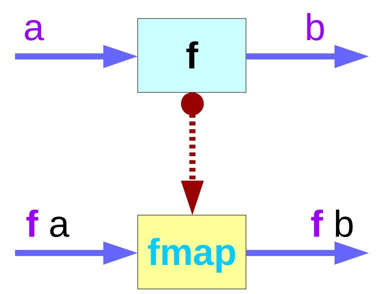
Maybe Functor Examples (1)

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

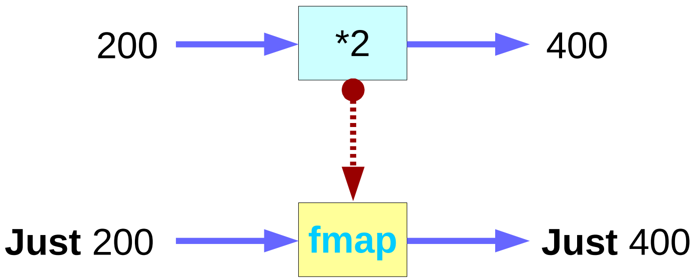
f

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

f




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



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

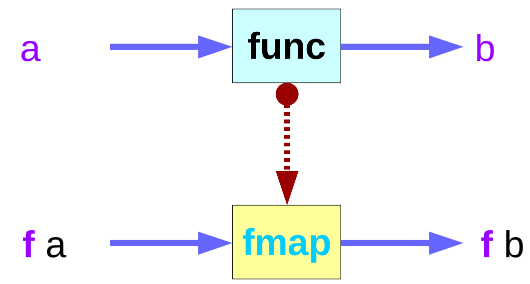
Maybe Functor Examples (2)

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

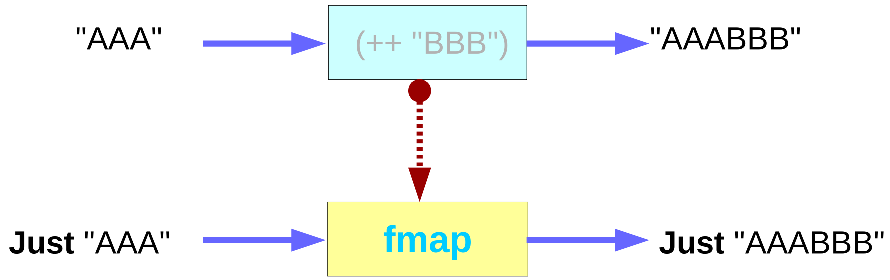
f

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

f




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



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

Maybe as a functor

Functor typeclass:

- transforming one type to another
- transforming operations of one type to those of another

Maybe a is an instance 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).

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

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
```

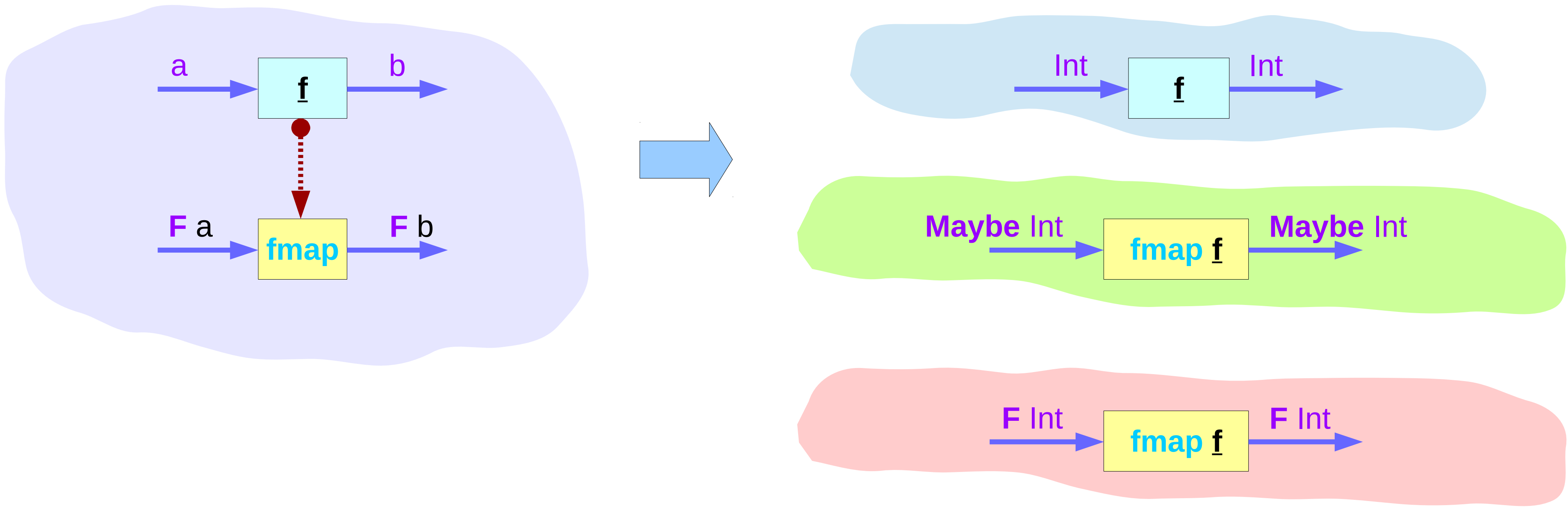
https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell

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).

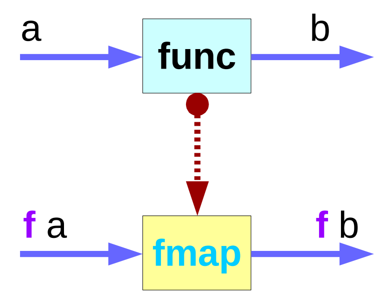


https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell

fmap func

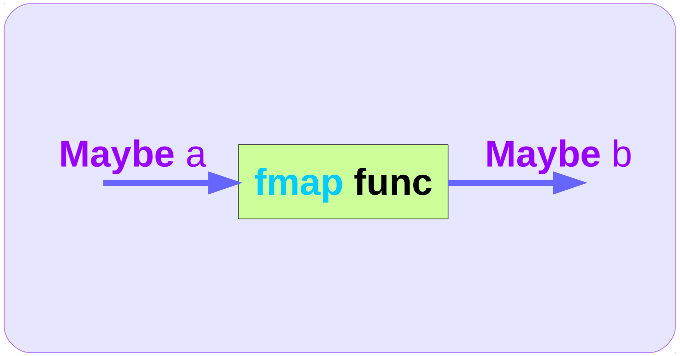
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

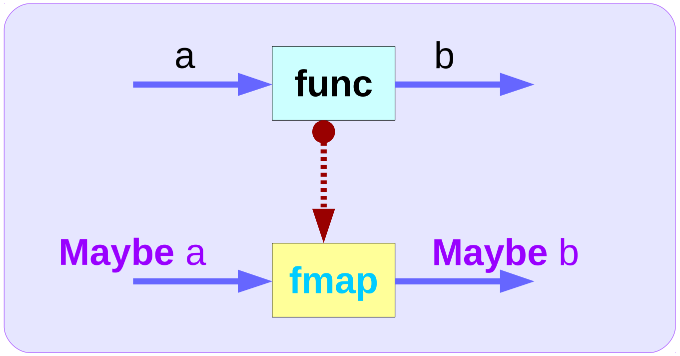
```
instance Functor Maybe where
    fmap func (Just x)  = Just (func  x)
    fmap func Nothing = Nothing
```



fmap func | Just x

fmap | func | Just x





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

Apply a function to lifted type values

m_x :: Maybe Integer   (Just 101, Nothing, …)

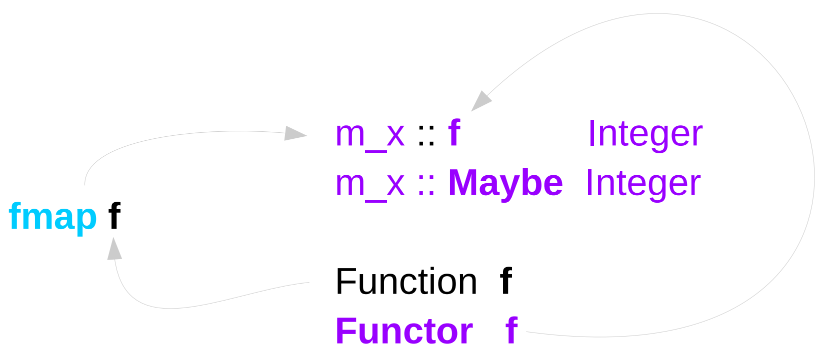
f :: Int -> Int

fmap f m_x

to <u>apply</u> the function **f** directly to the Maybe Integer
without concerning whether it is Nothing or not

```
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
```



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

Maybe as a functor

```
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
```

Function **f** of (a -> b)

fmap f m_x

lifted type

A **Functor f** applied with one type

f a or **f** b (**f** is not **f**)

m_x :: **f** Integer

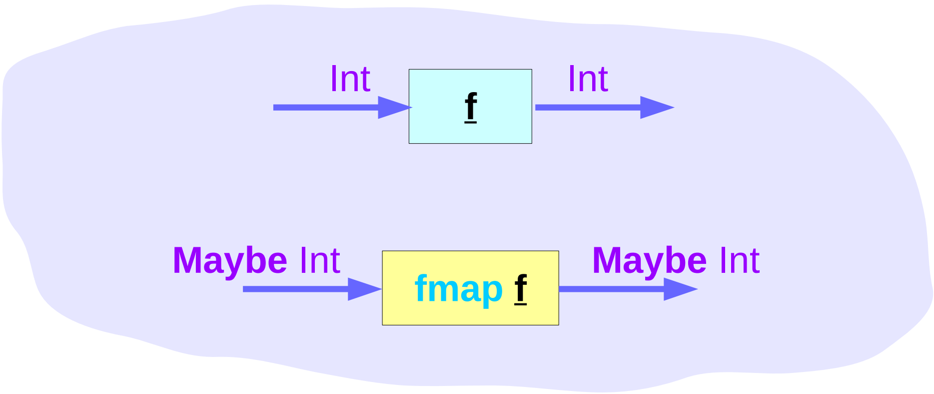
m_x :: **Maybe** Integer

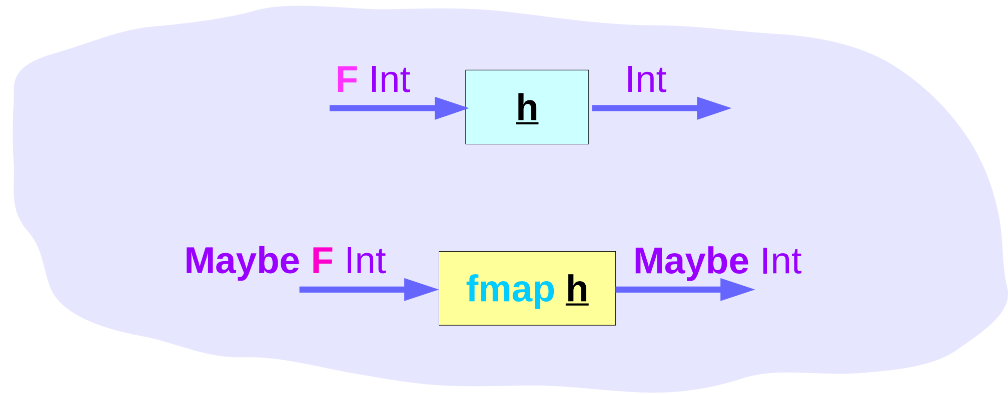
Function **f**

Functor f

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

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 :: (F Int -> Int) -> f Int -> f Int
```

```
instance Functor Maybe where
   fmap :: (F Int -> Int) -> Maybe F Int -> Maybe Int

   fmap h (Just f_x) = Just (h f_x)
   fmap h Nothing  = Nothing
```

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

Maybe as a functor

So if you have a Maybe Integer value m_x and an Int -> Int function f,
you can do **fmap f** m_x to apply the function **f** directly to the Maybe Integer
without worrying if it's actually got a value or not.

In fact, you could 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.

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

Maybe instances

Maybe is

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

https://wiki.haskell.org/Maybe

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)

https://wiki.haskell.org/Maybe

Maybe Functor

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

```
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
```

fmap **f** (**Just** x) = **Just** (**f** x)
fmap **f** **Nothing** = **Nothing**

https://wiki.haskell.org/Maybe

Maybe Functor

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

fmap **f** (**Just** x) = **Just** (**f** x)
fmap **f** **Nothing** = **Nothing**

```
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
```

https://wiki.haskell.org/Maybe

Maybe Monad

For **Monad**, the **bind** operation

passes through **Just**, while

Nothing will force the result to always be **Nothing**.

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
```

https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Maybe.html

Monad

a **Monad** is just a special **Functor** with extra features

Monads like IO map *types* to *new types*
that represent "computations that result in values"

can *lift* regular functions into **Monad** types
via a **liftM** function (like a **fmap** function)

liftM transform a regular function
into a "computations that results in the value obtained by evaluating the function."

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

Maybe as a Monad

Maybe is also a **Monad**

represents “computations that could *fail* to return a value"

an immediate abort
a valueless return in the middle of a computation.

enable a whole bunch of computations
without explicit checking for errors in each step

a computation on **Maybe** values *stops*
as soon as a **Nothing** is encountered

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

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.

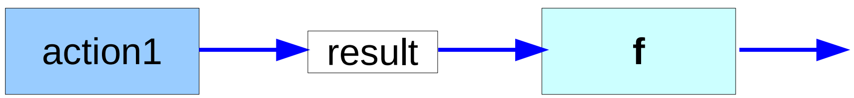
mapMaybe

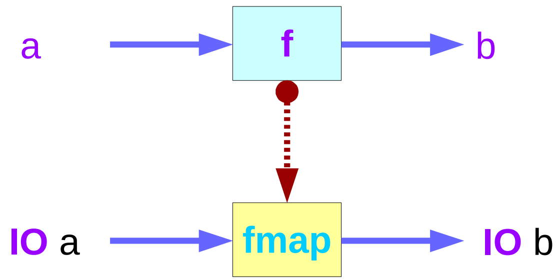
A different way to filter a list.

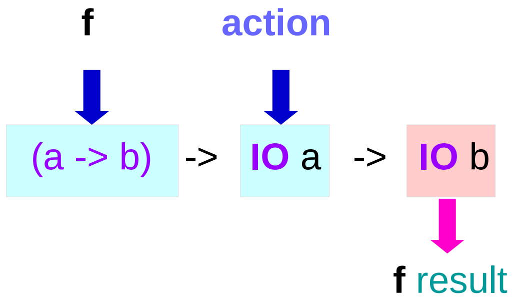
https://wiki.haskell.org/Maybe

Functor Typeclass Examples (1)

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








```haskell
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 Examples (2)

```
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

Functor Typeclass Examples (3)

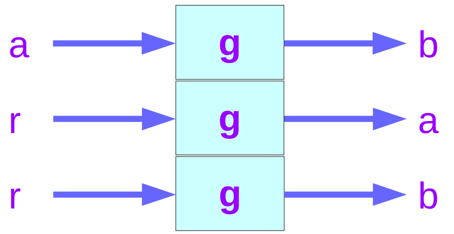
```
instance Functor ((->) r) where
  fmap f g = (\x -> f (g x))
```

```
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**

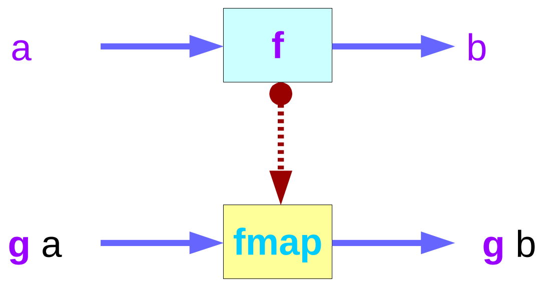
g :: **r -> a**



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

fmap :: (a -> b) -> ((->) r a) -> ((->) r b)

fmap :: (a -> b) -> (r -> a) -> (r -> b)



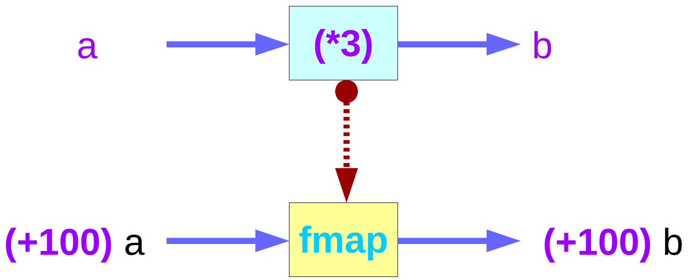
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functor Typeclass Examples (4)

```
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
```

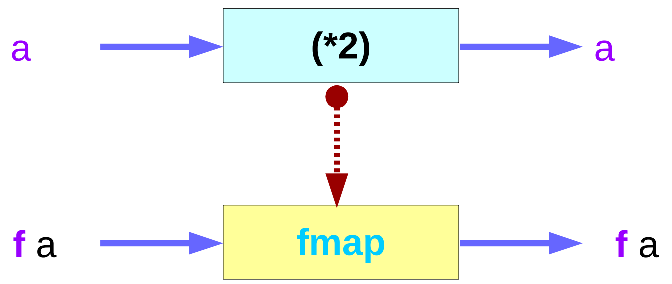
```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```



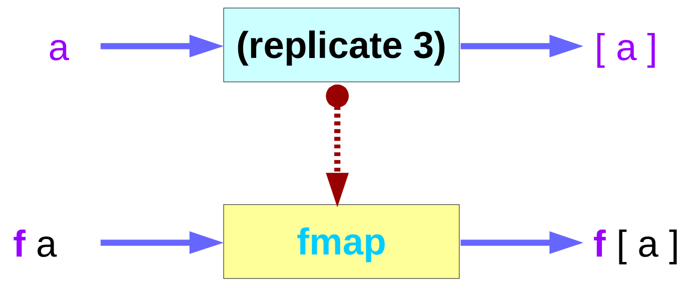
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Functor Typeclass Examples (5)

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 Examples (6)

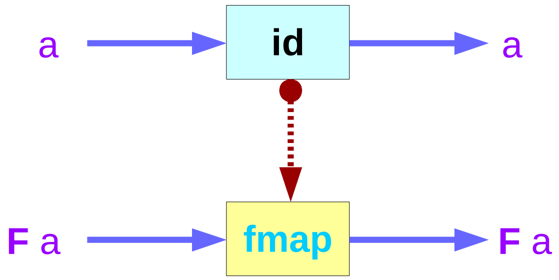
```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

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

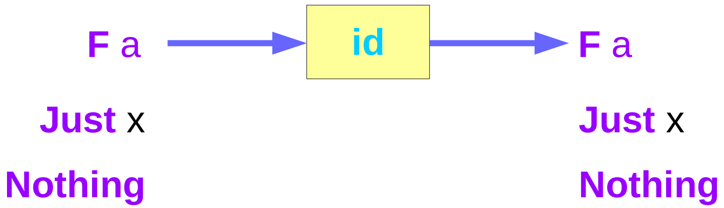
Functor Laws

fmap **id** = **id**

id :: a -> a
id   x = x



instance **Functor** Maybe where
    fmap **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

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

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

Functor Typeclass

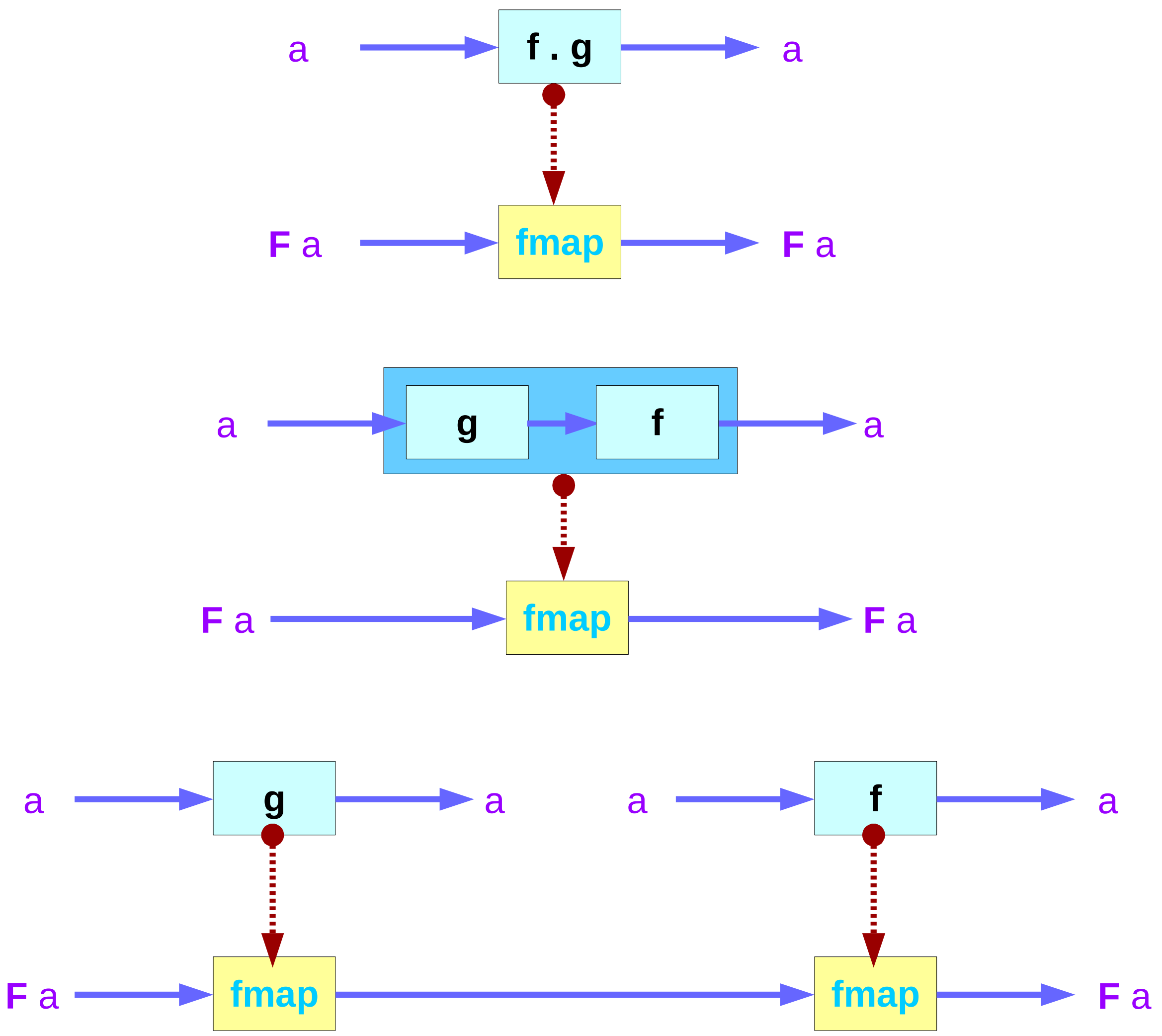
```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

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 f (Just x) = Just (f x)
        fmap f 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