Functor (1A)

Young Won Lim
7/21/17

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 possessing such behavior

Such behavior is defined by
- **function definition**
- **type declaration** to be implemented

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

No relation with classes in Java or Python

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

A Typeclass Example

the Eq typeclass

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

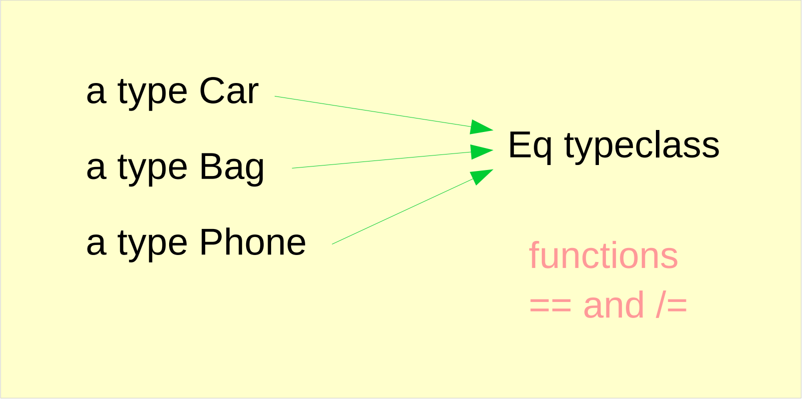
a type Car

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



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

Eq Typeclass Example

```
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

Show Typeclass Example

```
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"
```

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

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

Show Typeclass Example

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

class Num a where
  ...
```

class constraint on a class declaration
only we state that our type a must be an instance of **Eq**

an instance of **Eq**
before being an instance of **Num**

When defining the required function bodies
in the class declaration or
in instance declarations,

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

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

Show Typeclass Example

class constraints in class declarations

to make a typeclass a subclass of another typeclass

class constraints in instance declarations

to express requirements about the contents of some type.

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

Show Typeclass Example

the **a** : a concrete type

Maybe : not a concrete type
: a type constructor that takes one parameter
produces a concrete type.

Maybe a : a concrete type

```
instance (Eq m) => Eq (Maybe m) where
   Just x    ==  Just y          =      x == y
   Nothing  ==  Nothing          =      True
            _ == _               =      False
```

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
```

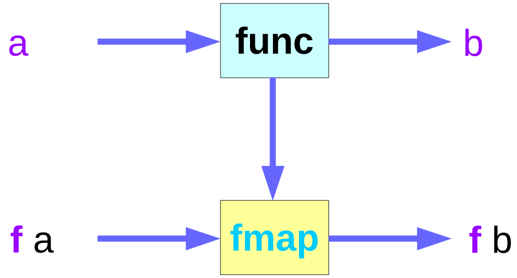
The Functor typeclass

- 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

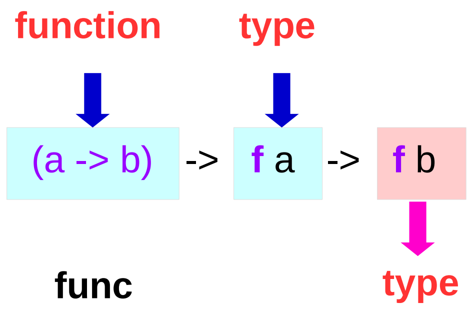

a → **func** → b
f a → **fmap** → f b

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)

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

map takes

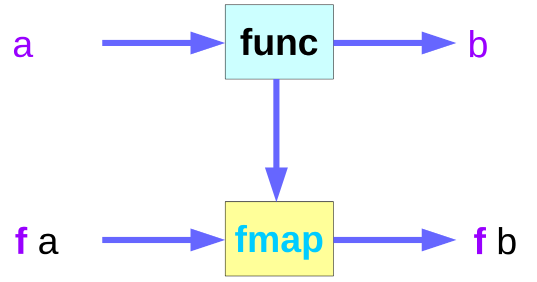
- a function from one type to another (* 2)
- take a list of one type [1, 2, 3]
- returns a list of another type [2, 4, 6]

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

List : an instance of Functor typeclass

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

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

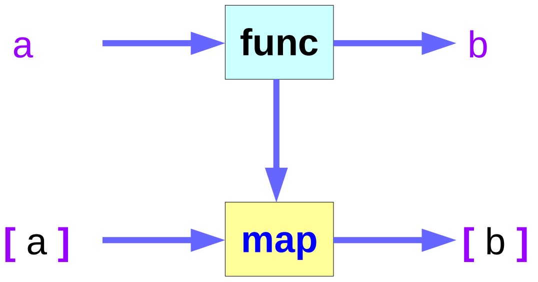


function fmap
function func
type constructor f

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

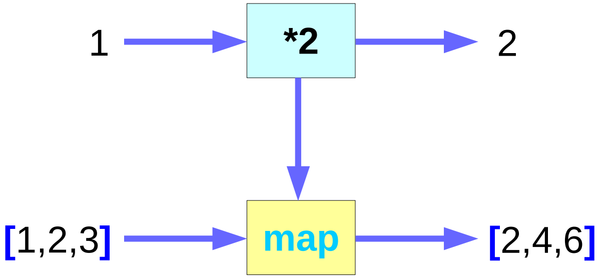
List Examples

```
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]
```



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

Maybe : an instance of Functor typeclass

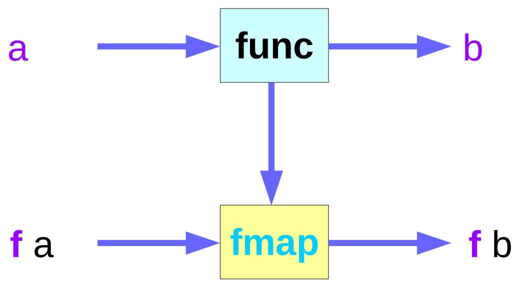
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

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

f ⟷ Maybe

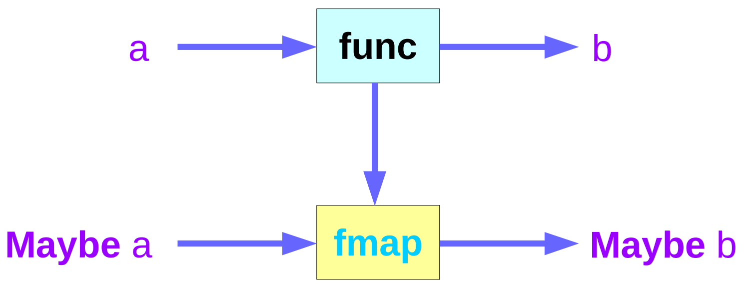
f a ⟷ Maybe a

f b ⟷ Maybe b



a → func → b

f a → fmap → f b



a → func → b

Maybe a → fmap → Maybe b

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

Maybe : a type constructor

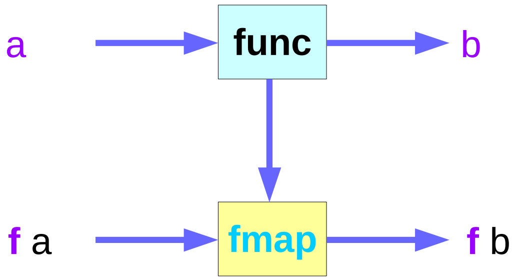
```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

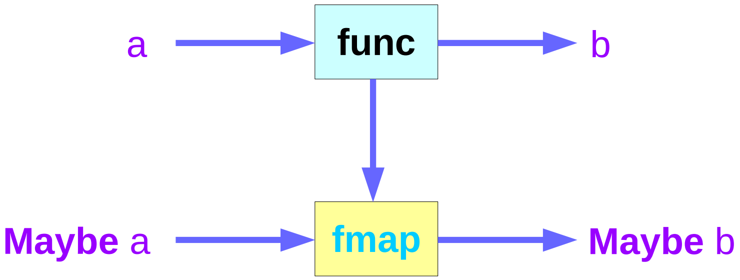
```
instance Functor Maybe where
   fmap func (Just x) = Just (func  x)
   fmap func Nothing = Nothing
```

f : a **type variable**

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

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



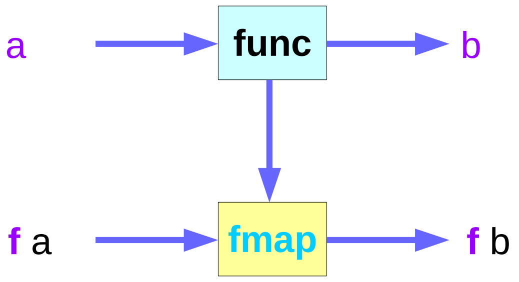


http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

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
```



a

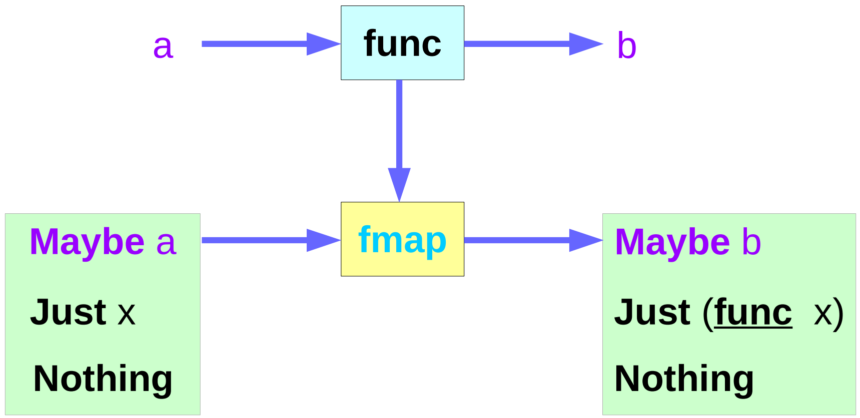
func

b

f a

fmap

f b



a

func

b

Maybe a

Just x

Nothing

fmap

Maybe b

Just (func x)

Nothing

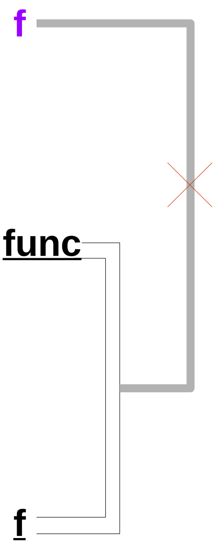
```
fmap :: (a -> b) -> f a -> f b

fmap func (Just x) = Just (func  x)

fmap func Nothing = Nothing
```

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

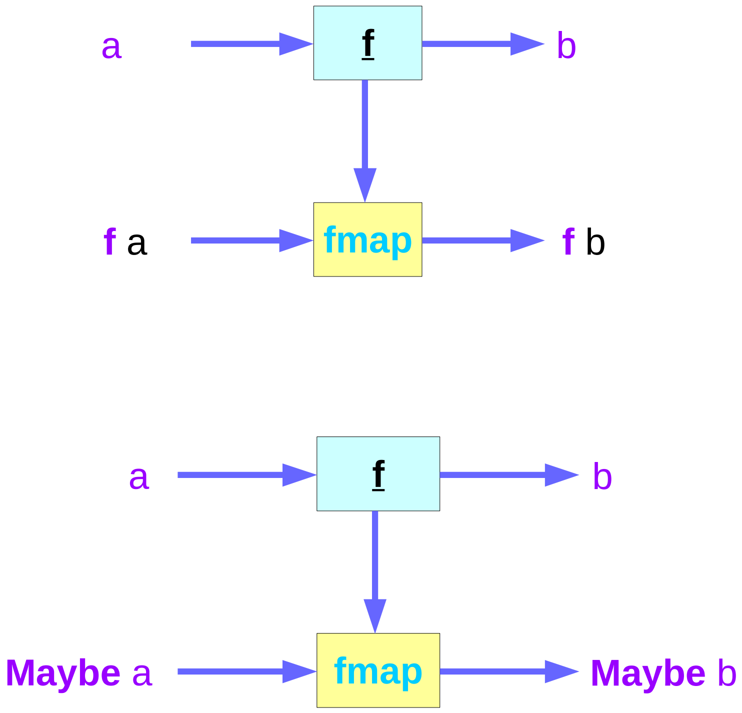
Maybe : fmap takes a function

```
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

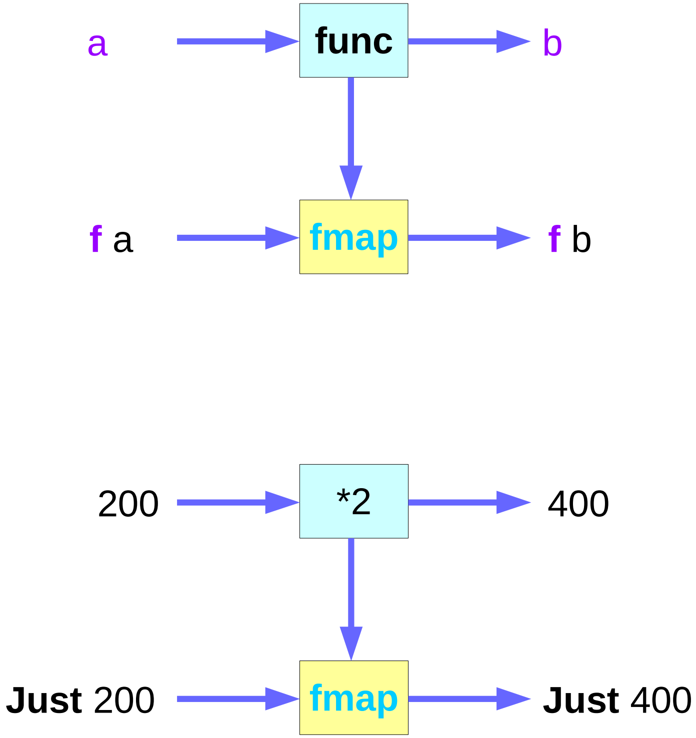
http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

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
```

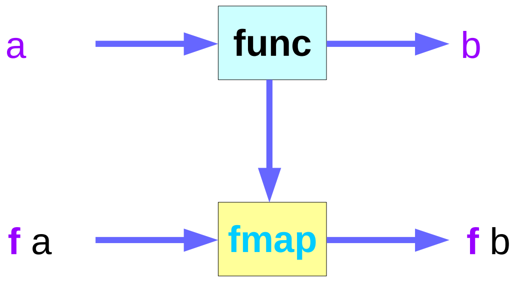


http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

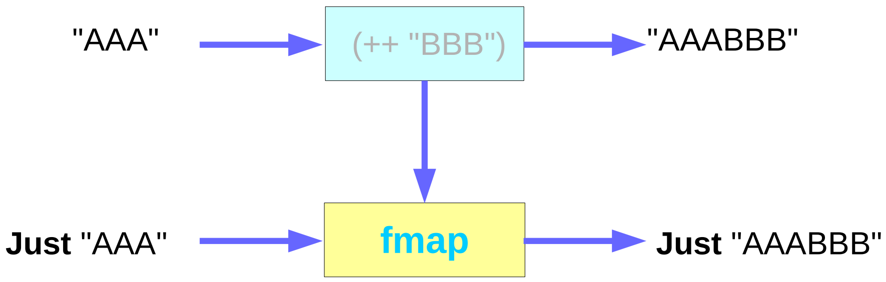
Maybe Examples (2)

```
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
```



```
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**

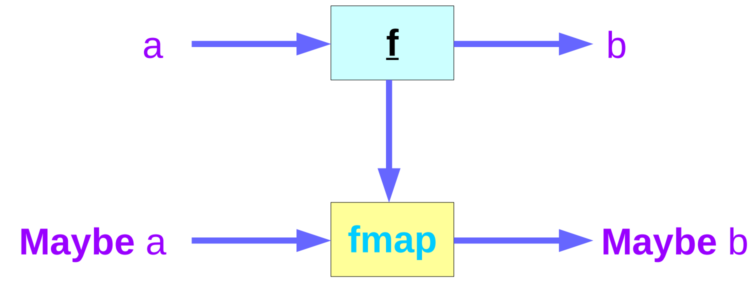
https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell

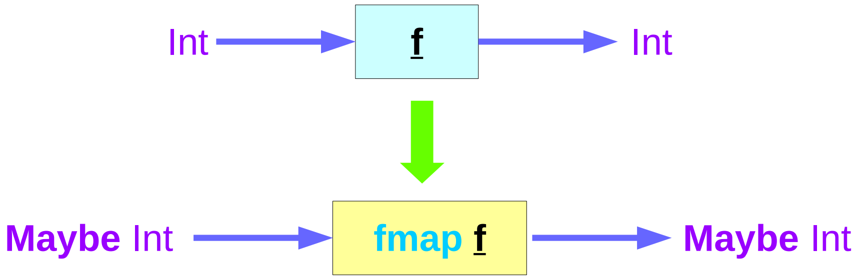
Transforming operations

Functor provides **fmap** method

maps functions of the base type (such as Integer)

to *functions* of the lifted type (such as Maybe Integer).

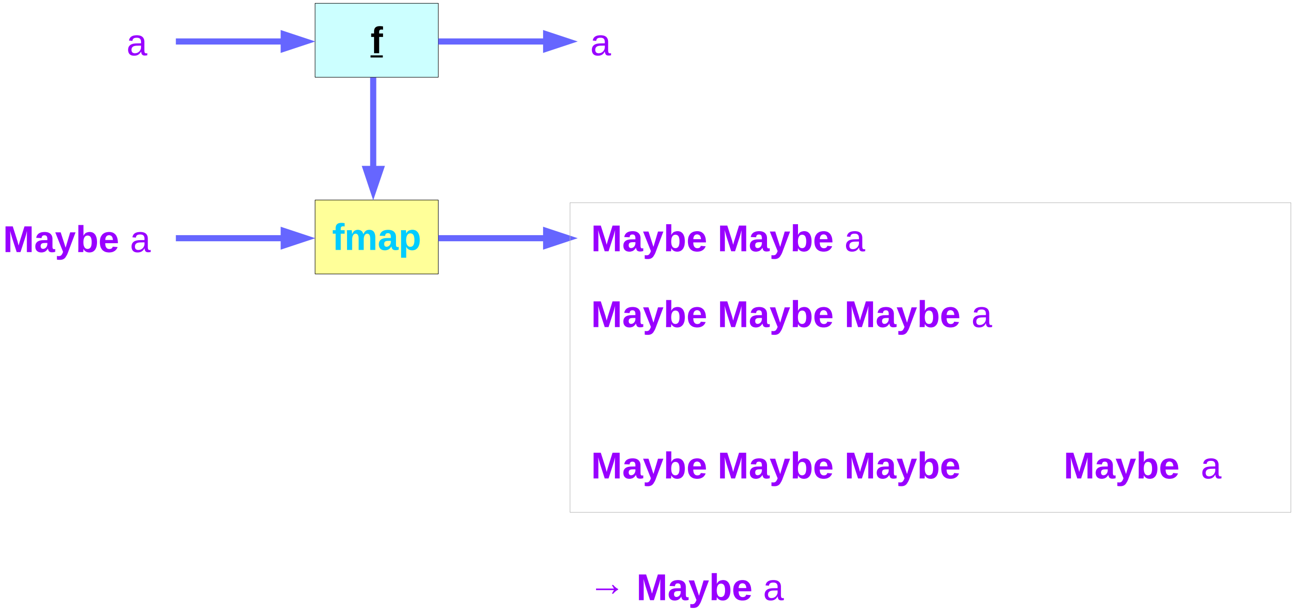




https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell

Maybe as a functor

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.



a
f
a

Maybe a
fmap

Maybe Maybe a

Maybe Maybe Maybe a

Maybe Maybe Maybe **Maybe** a

→ **Maybe** a

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

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

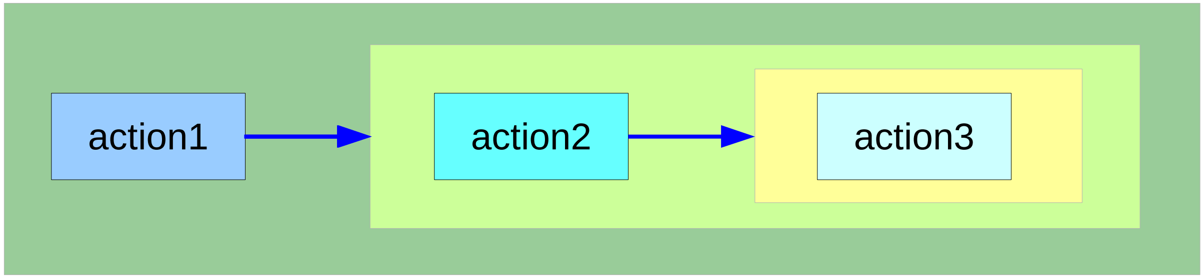
Translating in **do** notation

```
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 assigns 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

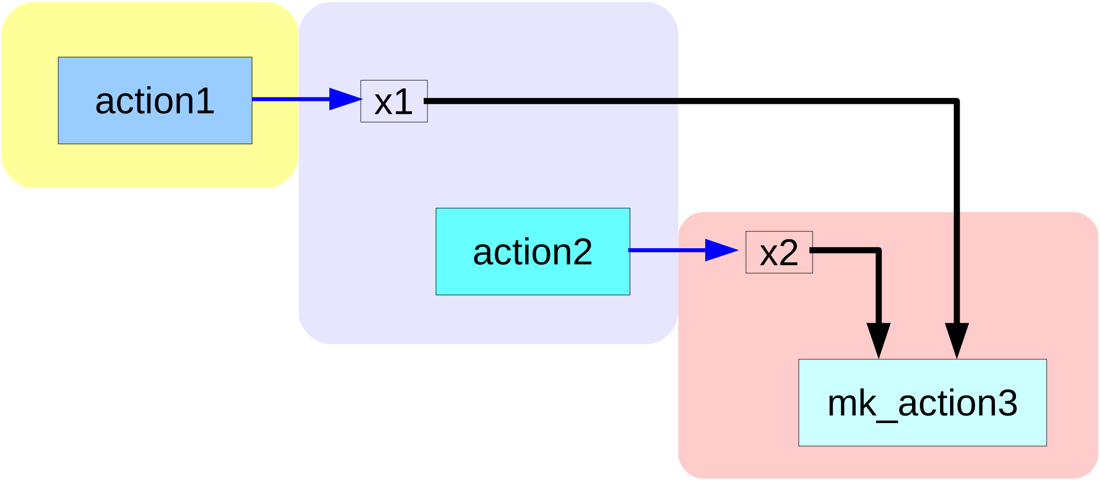
Translation using the bind operator (>>=)

```
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 ))
```



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

Anonymous Function

```
\x -> x + 1
```

```
(\x -> x + 1) 4
5 :: Integer
```

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

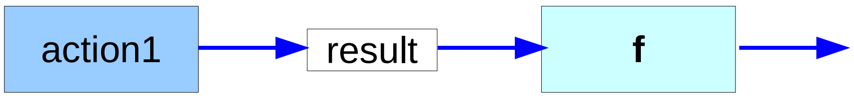
Lambda Expression

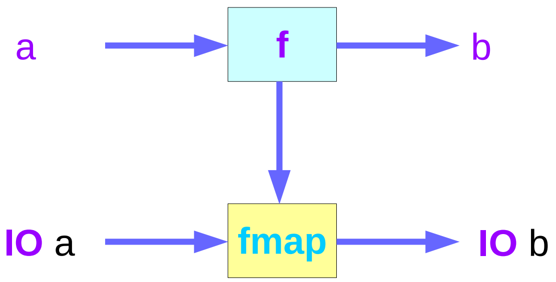
```
addOne = \x -> x + 1
```

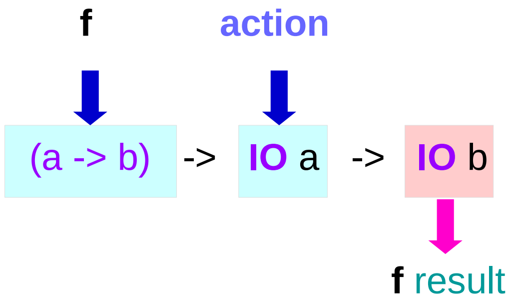
https://wiki.haskell.org/Anonymous_function

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
```

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

. 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)

putStrLn . show $ 1 + 1

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

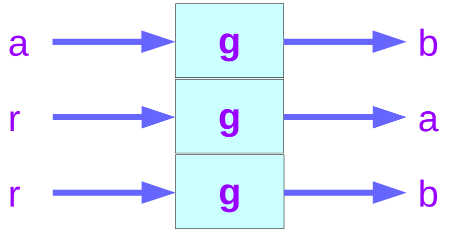
Functor Typeclass

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

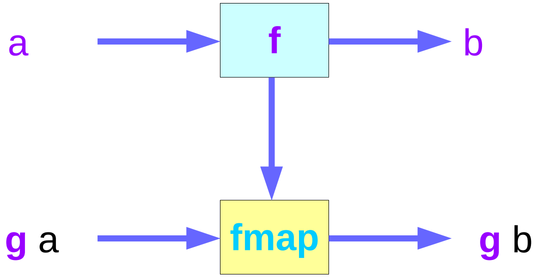
```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

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)
```



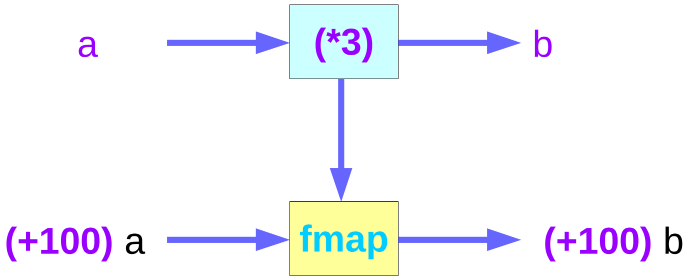
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

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
```

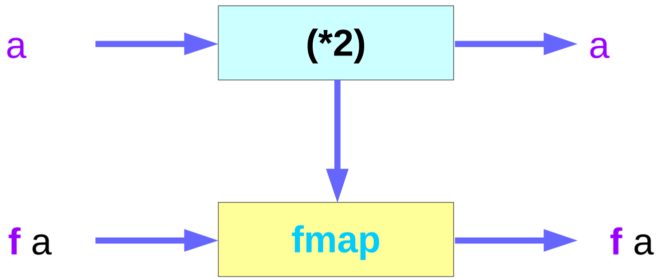
```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```



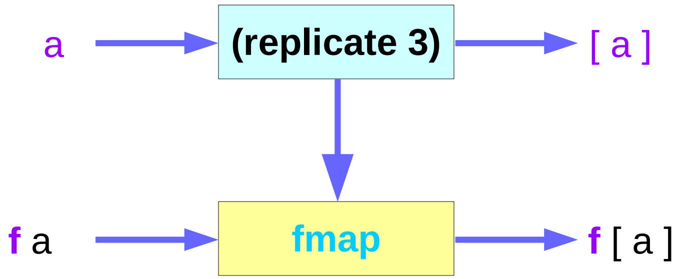
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

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"
```

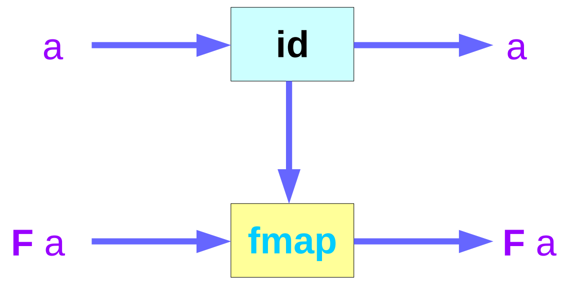
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

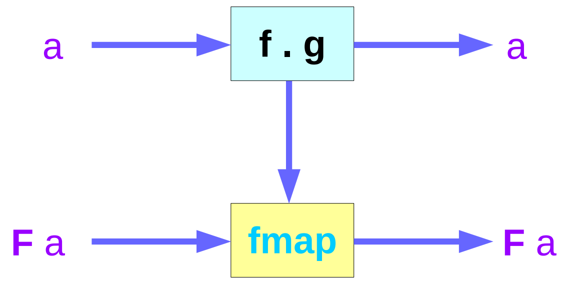
Functor Laws

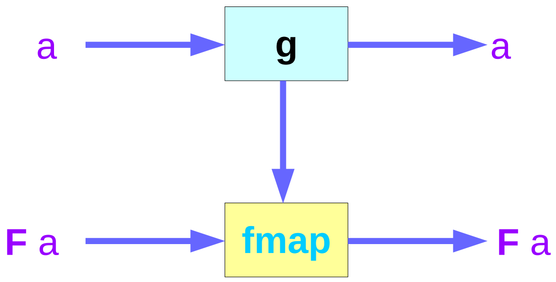
fmap **id** = **id**

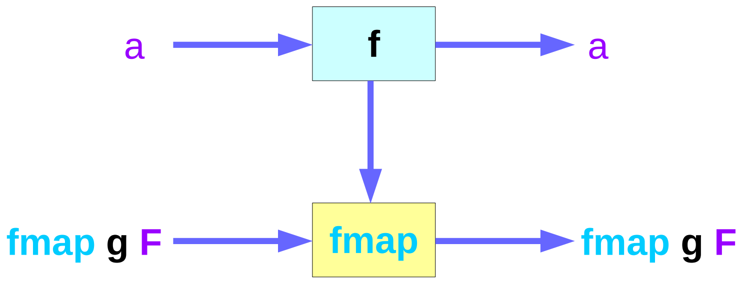
fmap (**f . g**) = fmap **f** . fmap **g**

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









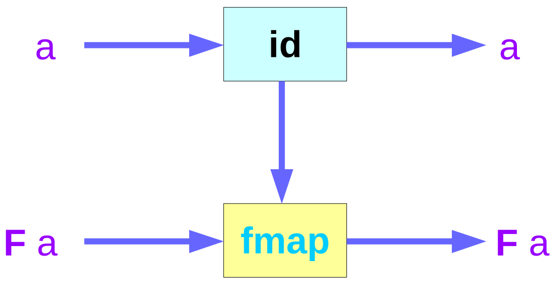
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

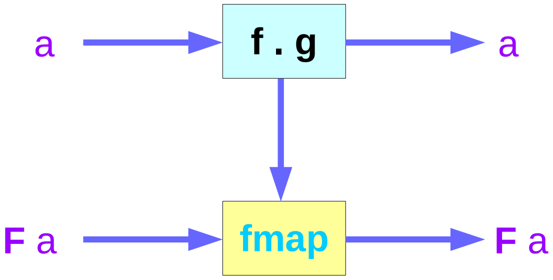
Functor Laws

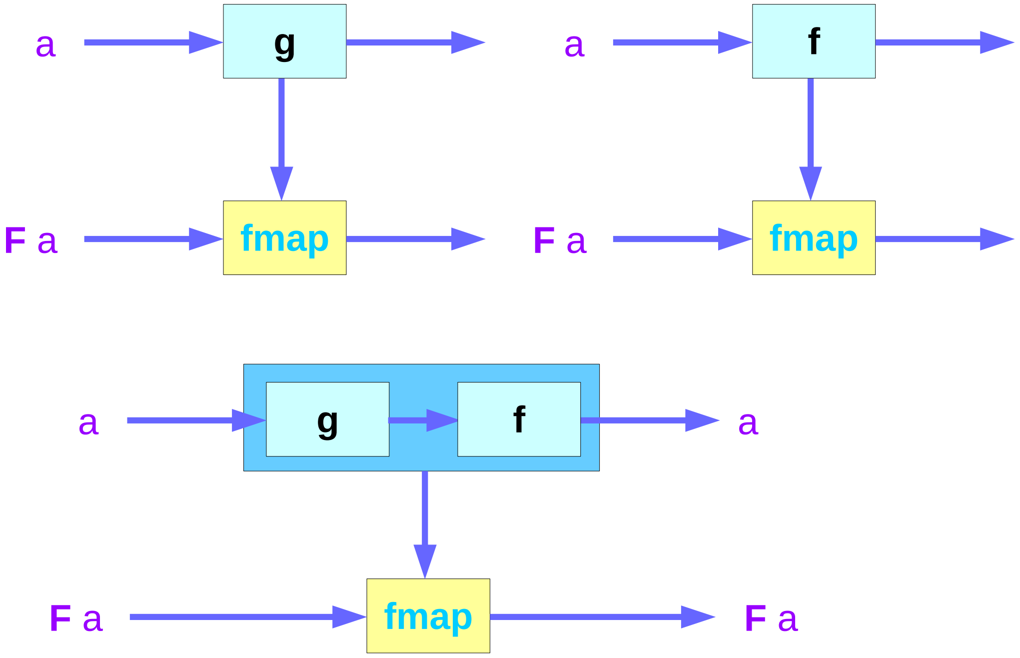
fmap **id** = **id**

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

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







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