Functor (1A)

Young Won Lim
11/27/17

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

Function Definition

Function Definition I.

square x = x * x   - function type is inferred → not efficient **Type Inference**

Function Definition II.

square :: Double -> Double – function type declaration

square x = x * x

Function Type Declaration

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

Type Declaration

the declaration of an identifier's type

the identifier name :: the type name ...

type names in Haskell <u>always</u> begin with a <u>capital</u> letter,

identifier names (including function identifiers) must <u>always</u> begin with a <u>lower</u>-case letter

http://www.toves.org/books/hsfun/

Function Types and Type Classes

Function Definition I.

square x = x * x

Function Definition II.

square :: Double -> Double

square x = x * x

function definition

=

function definition

- function **type declaration**

=

type class – a set of types

- function **type** 1
- function **type** 2
-
- function **type** n

Requirements

Subclasses

http://www.toves.org/books/hsfun/

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

$$(==) :: a \rightarrow a \rightarrow Bool$$ - **a type declaration**

$$x == y = not\ (x\ /=\ y)$$

a function type

$$(==) :: a \rightarrow a \rightarrow Bool$$ - **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

Car Type Example

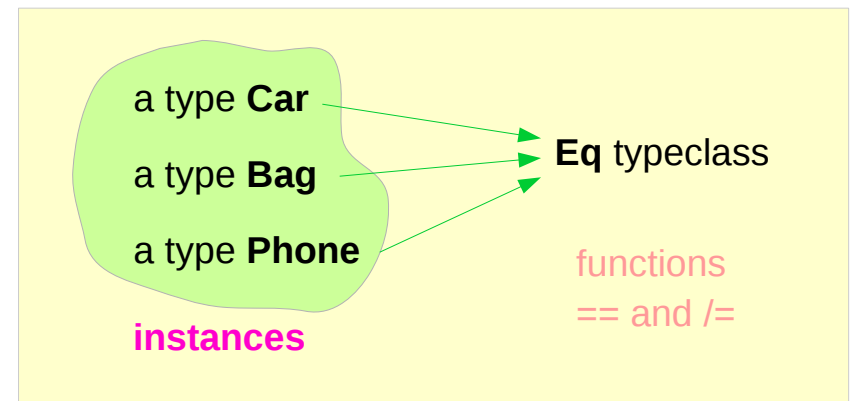**the Eq typeclass**

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**

instances

Eq typeclass

functions
== and /=

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
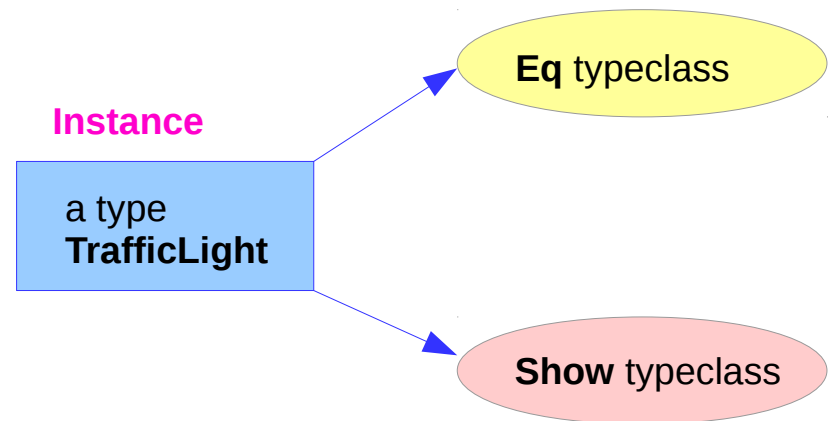
TrafficLight Type Example (2)

```
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

a type
TrafficLight

Eq typeclass

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
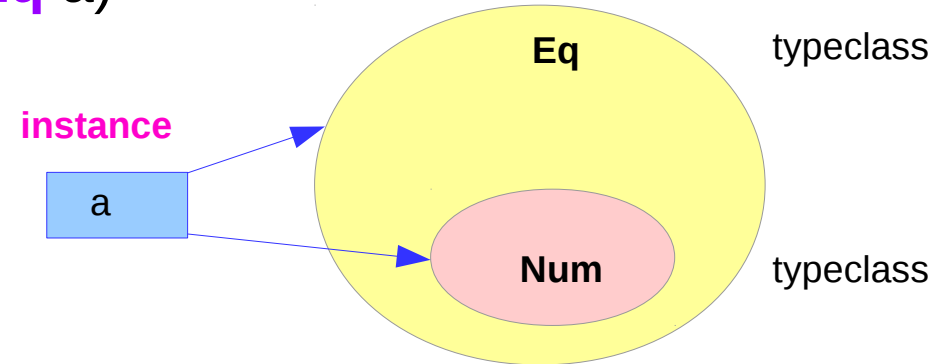
an instance of **Eq**
<u>before</u> being an instance of **Num**

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**

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

  …

subclass

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

requirements

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

Derived instance

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

Functions : First-class Data Types

functions are **first-class data types**
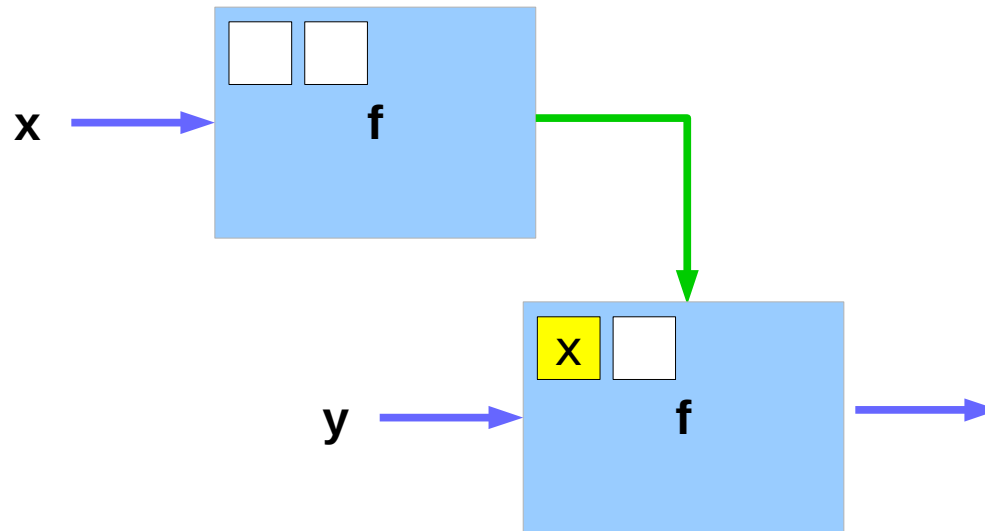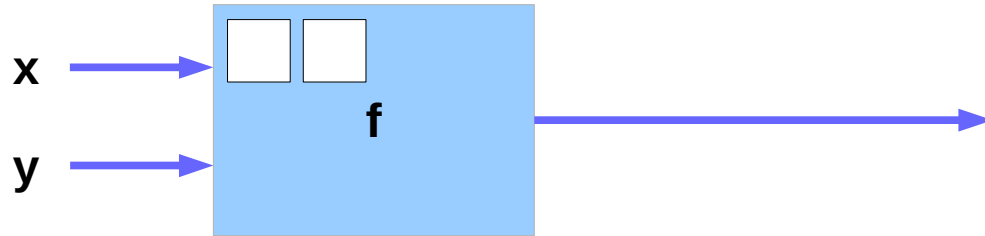
Haskell treats functions as regular data,

      just like integers, or floating-point values, or other types.

- a function can take other functions as parameters

- a function takes a parameter and produces another function (curried function)

f x y	**f :: a -> b -> c**
(f x) y	**f :: a -> (b -> c)**
f x returns a function of type **b -> c**	
g y	**g :: b -> c**

a
x → **f**   **(b->c)**

f :: a -> b -> c

b
y → **g** → **z**   **c**

http://www.toves.org/books/hsfun/

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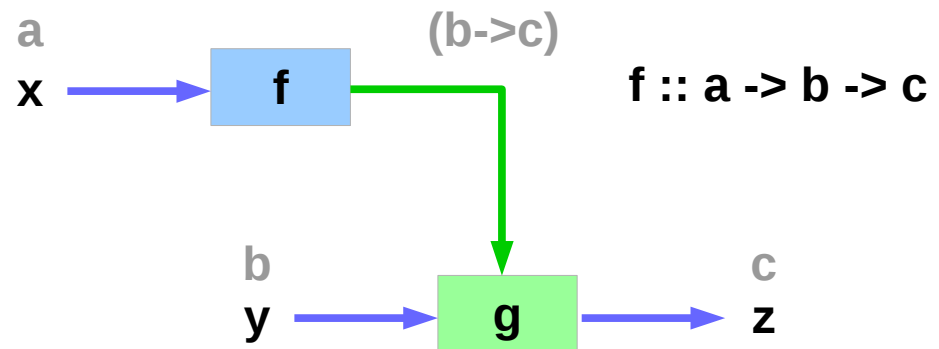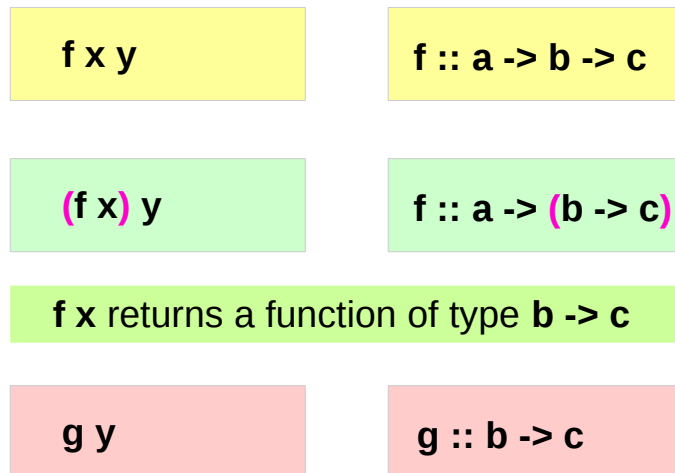
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Polymorphic Functions

specific types vs. arbitrary types

a **polymorphic** functions – an abstract type

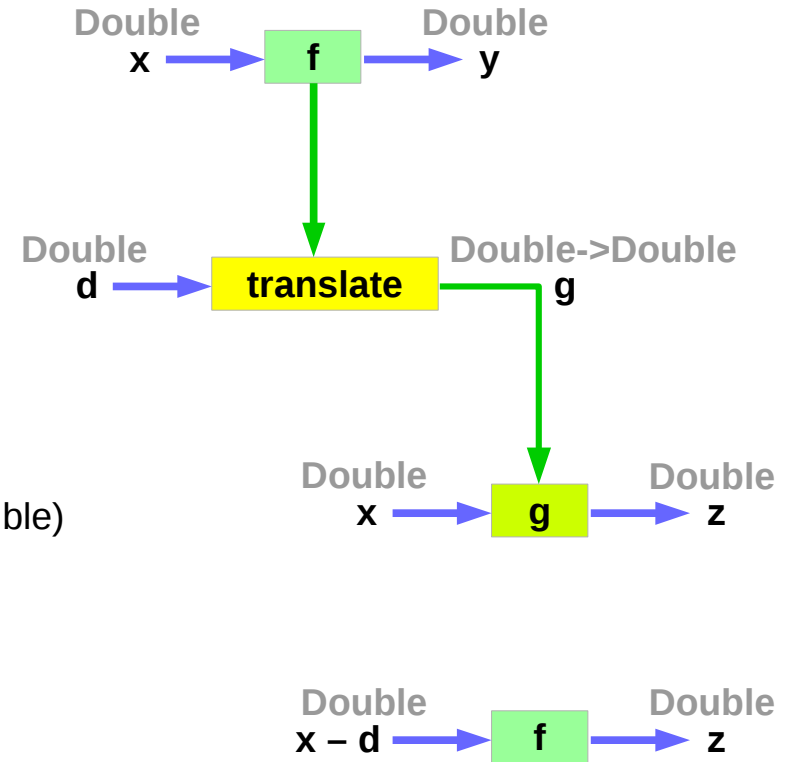each type variable is generally a lower-case letter.

Example) A translate function

      takes a function f and a distance d

      returns a new function g

         that is f "translated" d units to the right

http://www.toves.org/books/hsfun/

Polymorphic Function Examples

Double   Double

x → **f** → y

Double   Double->Double

d → **translate** → g

Double   Double

x → **g** → z

translate :: (Double -> Double) -> Double -> (Double -> Double)

translate f d = **g where g** x = **f** (x – d)

Double   Double

x – d → **f** → **z**

translate :: (Double -> a) -> Double -> (Double -> a)

http://www.toves.org/books/hsfun/

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

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**

    <u>defines</u> one function, **fmap**

    <u>no</u> <u>default</u> implementation

the **type variable f**

    <u>not</u> 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
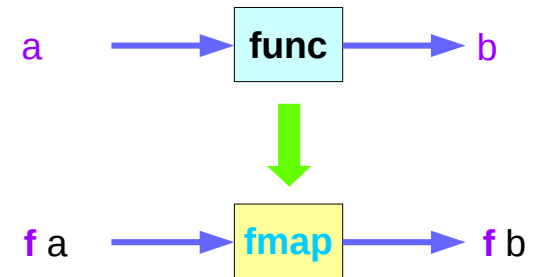
Function **map** & **fmap**

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

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

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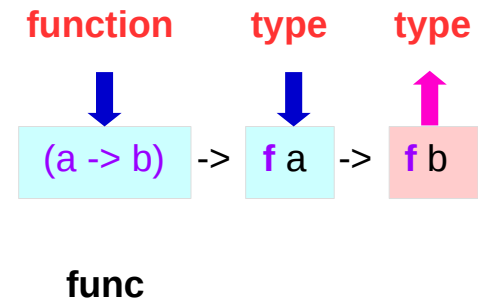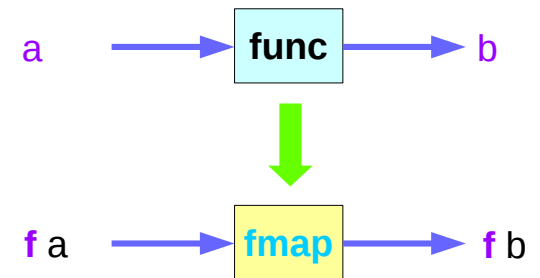- a **Functor** **f** applied with **one type** (**f** a)

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

```
function     type     type
   ↓           ↓         ↑
(a -> b)  ->  f a  ->  f b
```

func

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

map takes
- a function from one type to another
- take a list of one type
- returns a list of another type

(* 2)

[1, 2, 3]

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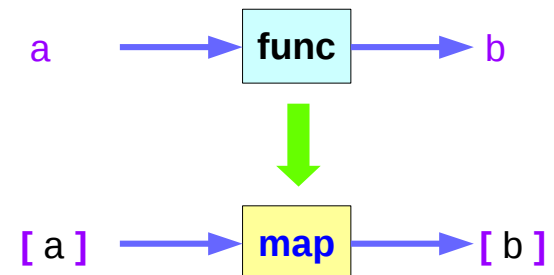
a **list** is an **instance** of the **Functor typeclass**.

```
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]])

a ⟶ **func** ⟶ b

f a ⟶ **fmap** ⟶ f b

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

a ⟶ **func** ⟶ b

[a] ⟶ **map** ⟶ **[b]**

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
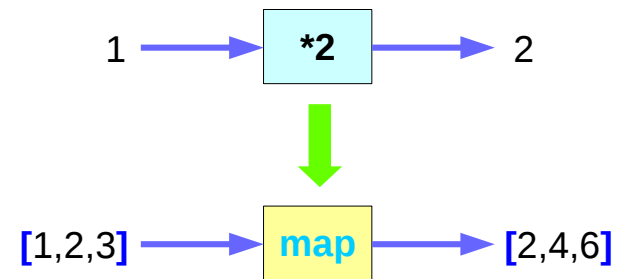
Maybe : an instance of the Functor typeclass
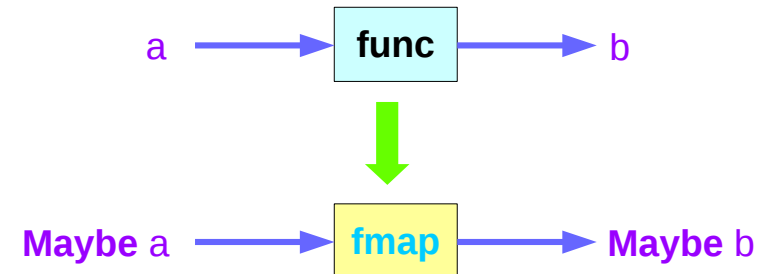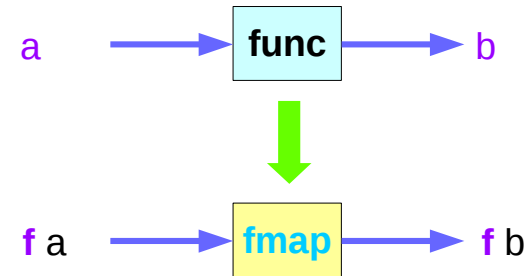
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

```
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 -> b)	⟹	**func**

a ⟶ **func** ⟶ b

f a ⟶ **fmap** ⟶ **f** b

a ⟶ **func** ⟶ b

Maybe a ⟶ **fmap** ⟶ **Maybe** b
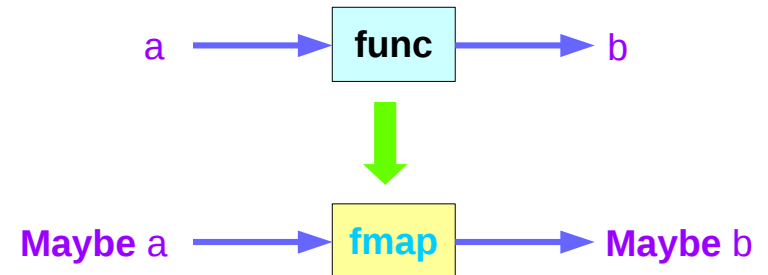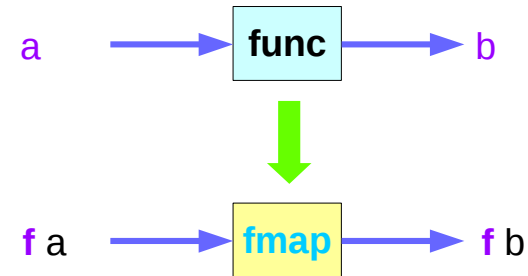
Maybe : a type constructor

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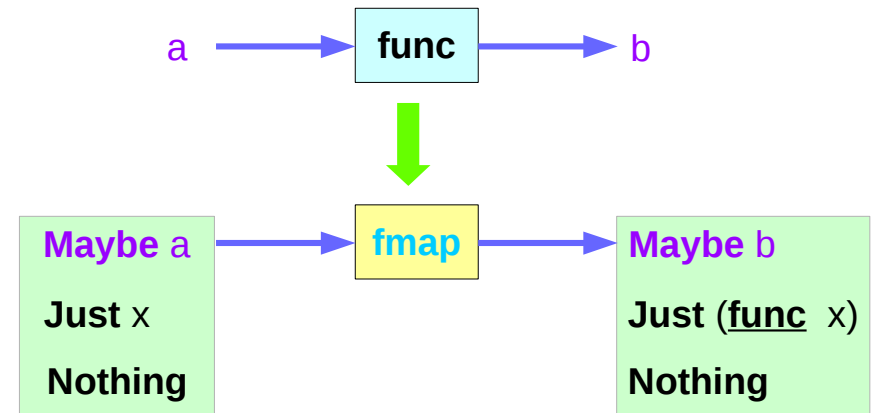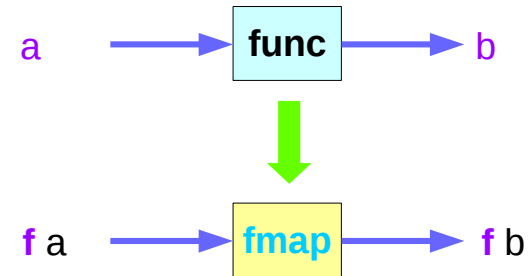
Maybe : an **instance** of **Functor** typeclass

a → func → b

f a → fmap → f b

a → func → b

Maybe a → fmap → **Maybe** b

25

Young Won Lim
11/27/17

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

A function argument to fmap and a Functor f


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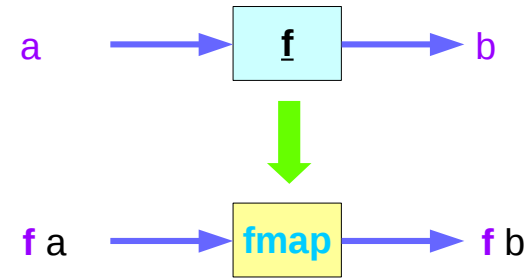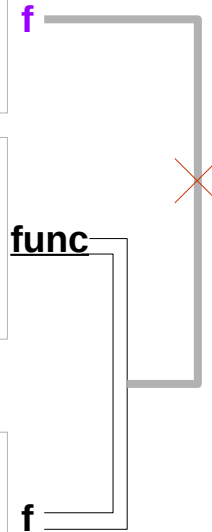
```
instance Functor Maybe where
    fmap  f  (Just x) = Just ( f  x)
    fmap  f  Nothing = Nothing
```
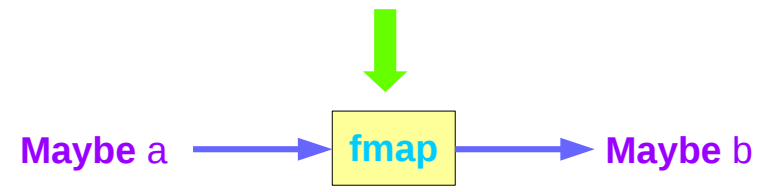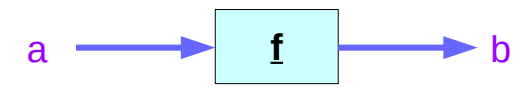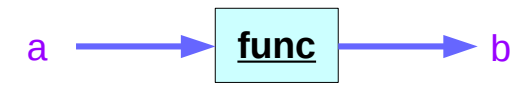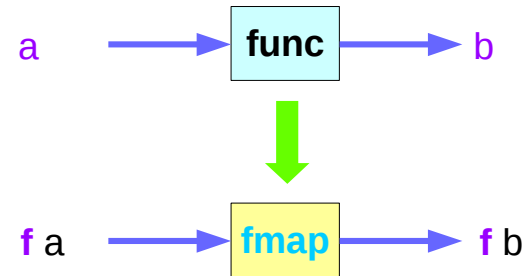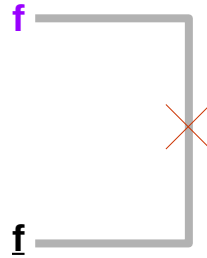
f is different from the type constructor f

func : a -> b
 f : a -> b

Young Won Lim
11/27/17

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
```

f ─┐
   │ ✕
f ─┘

a ──→ **func** ──→ b

⬇

f a ──→ **fmap** ──→ **f** b

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

200 ──→ *2 ──→ 400

⬇

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

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass
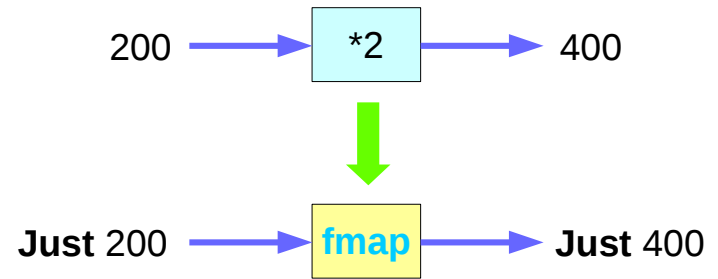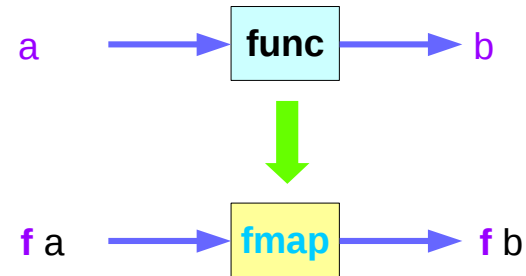
Maybe Examples (2)

```
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
```

f

f

a → func → b

f a → fmap → f b
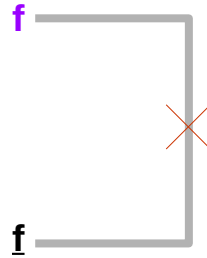
ghci> fmap (++ "BBB") (Just "AAA")
Just "AAABBB"
ghci> fmap (++ "BBB") Nothing
Nothing

"AAA" → (++ "BBB") → "AAABBB"

Just "AAA" → fmap → Just "AAABBB"

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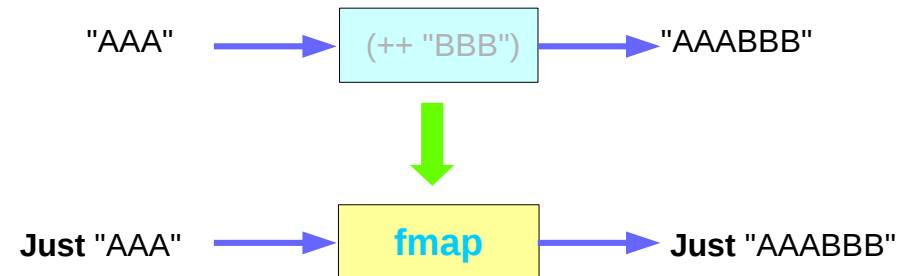
Maybe as a functor

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).
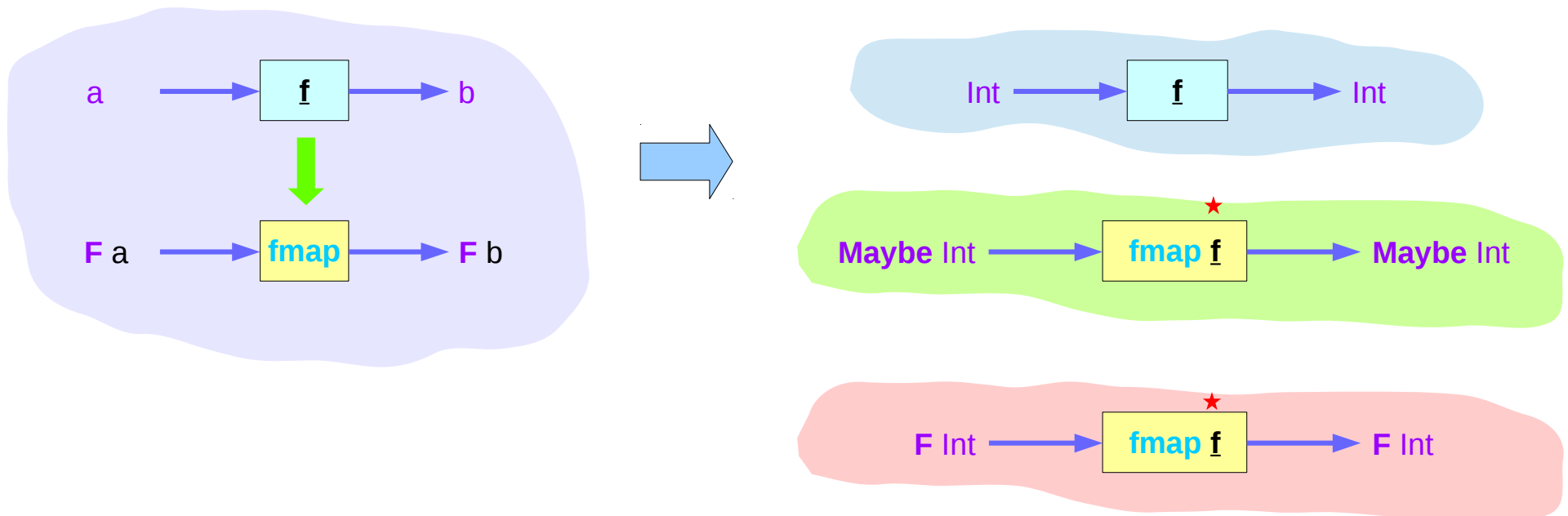
Maybe as a functor

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

```
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 of (a -> b)

fmap f m_x

lifted type

A Functor f applied with
one type **f** a or **f** b
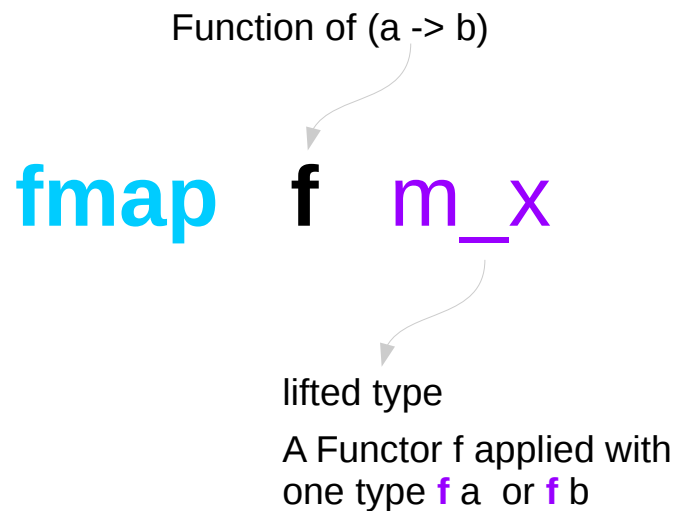
Maybe as a functor

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.
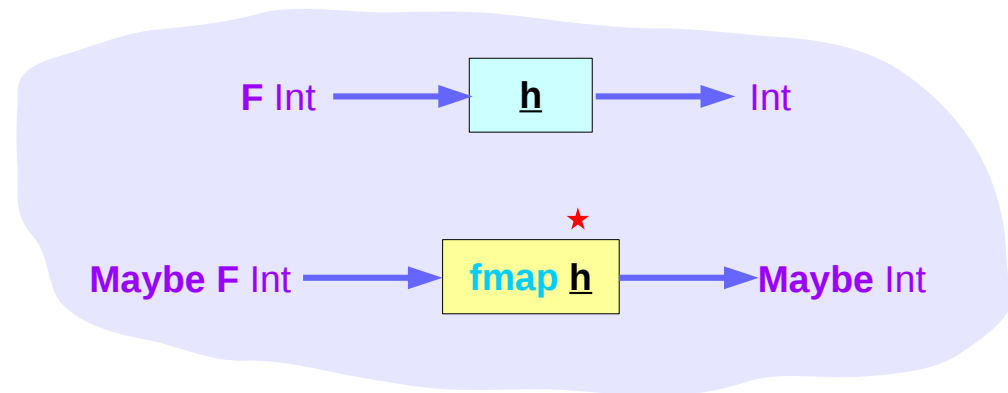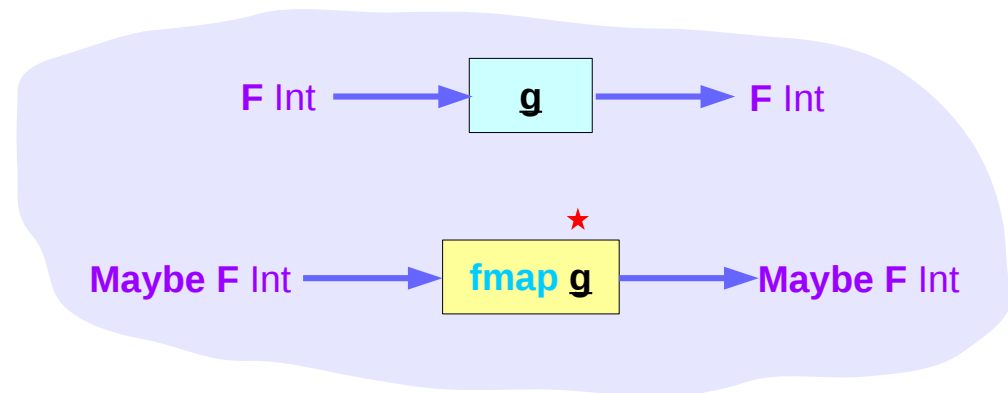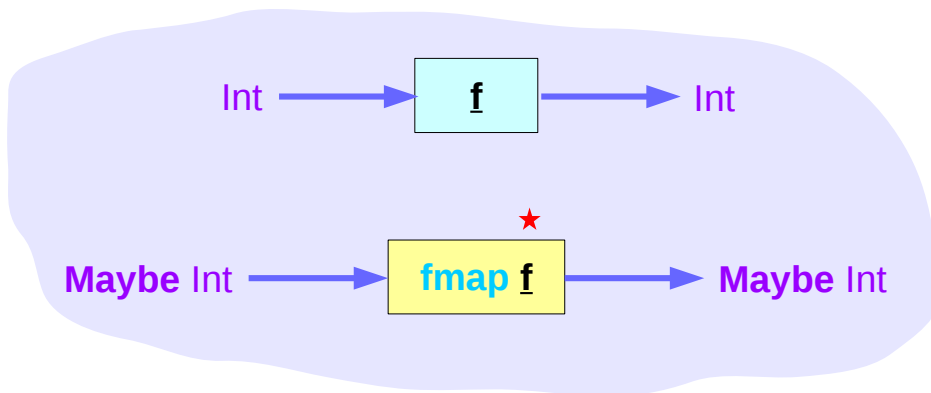
Maybe Type Definition

The Maybe type definition

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

Maybe is
- an <u>instance</u> of **Eq** and **Ord** (as a base type)
- an <u>instance</u> of **Functor**
- an <u>instance</u> 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.

```
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 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

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."

Young Won Lim
11/27/17

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

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
```

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

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
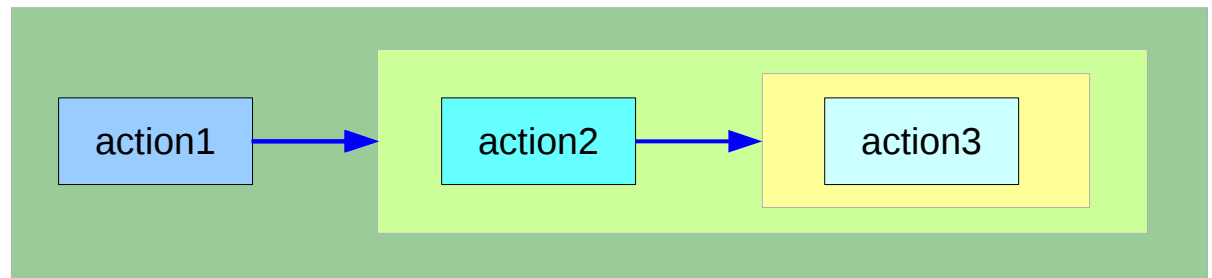
Young Won Lim
11/27/17

Translating in **do** notation

do { action1
    ; action2
    ; action3 **}**

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

do { action1
    ; **do {** action2
       ; action3 **} }**

do { action1
    ; **do {** action2
       ; **do {** action3 **} } }**

| action1 | → | action2 | → | 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 **}**

Translation using the bind operator (**>>=**)
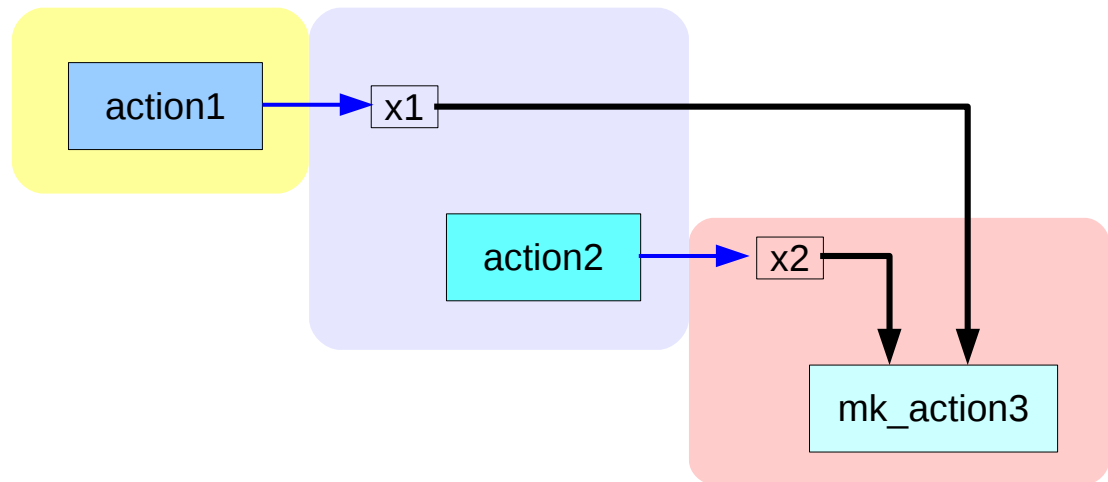
```
do { x1 <- action1
    ; x2 <- action2
    ; mk_action3 x1 x2 }
```

```
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

Functor Typeclass

instance Functor IO where
    fmap f action = **do**
        result **<- action**
        return (**f** result)
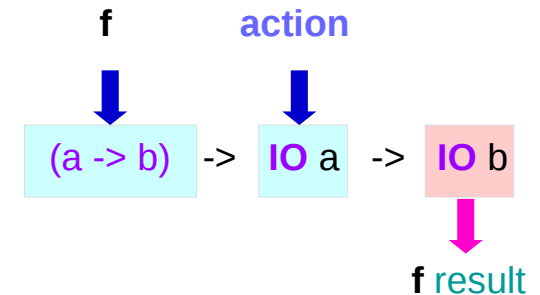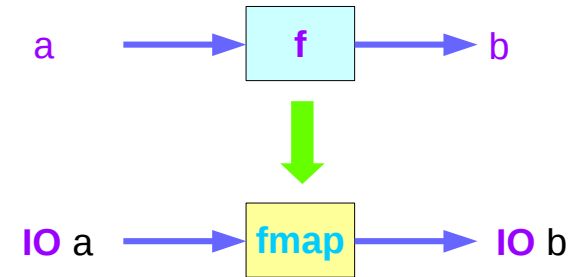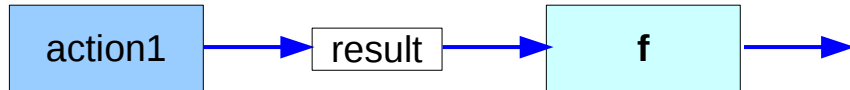
instance Functor Maybe where
    fmap func (**Just** x) = **Just** (**func** x)
    fmap func Nothing = **Nothing**

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 reverse getLine = do
    fmap f action = do                                  result <- getLine
        result <- action                                return (reverse result)
        return (f result)
```

$ 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

Young Won Lim
11/27/17

. 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

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))

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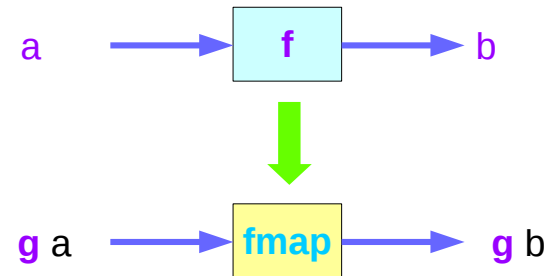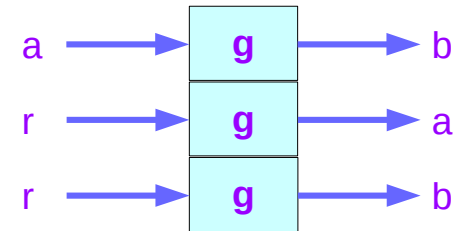
    g :: **a -> b**

    g :: **r -> a**

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

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

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

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

a ——→ **g** ——→ b

r ——→ **g** ——→ a

r ——→ **g** ——→ b

a ——→ **f** ——→ b

g a ——→ **fmap** ——→ **g** 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 <u>**f**</u> (**Just** x) = **Just** (<u>**f**</u> x)
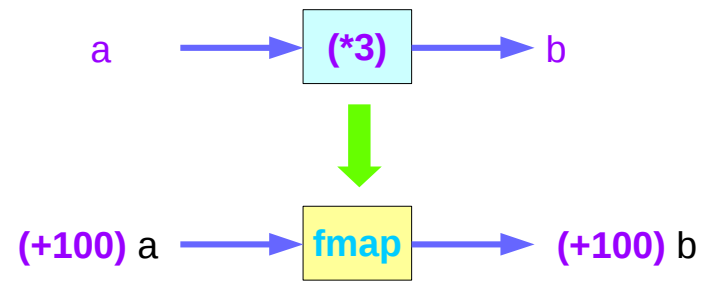   **fmap** <u>**f**</u> **Nothing** = **Nothing**

```
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

```
a ─────────▶ │  (*2)  │ ─────────▶ a
             ║
             ▼
f a ────────▶ │ fmap │ ─────────▶ f a
```

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

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

```
a ─────────▶ │ (replicate 3) │ ─────────▶ [ a ]
             ║
             ▼
f a ────────▶ │ fmap │ ─────────▶ f [ a ]
```
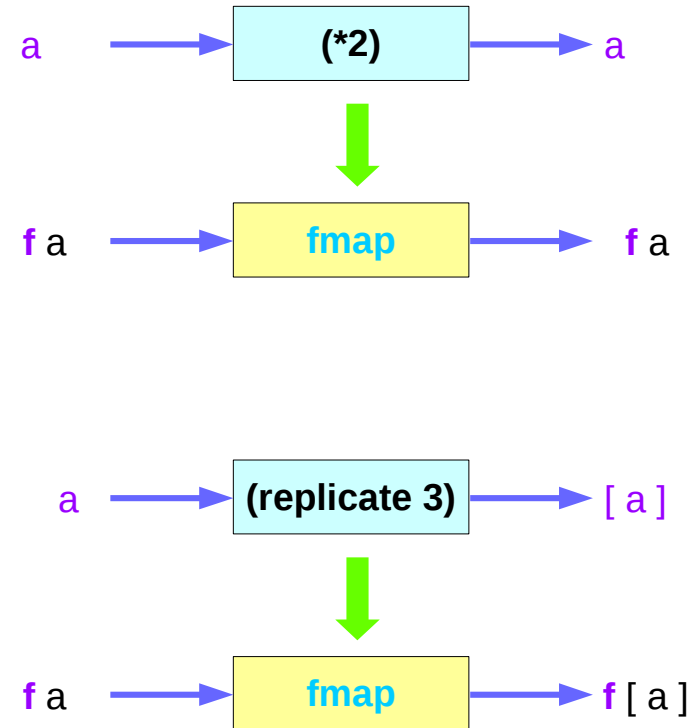
Functor Typeclass

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
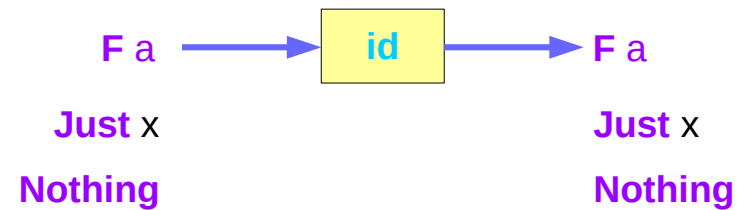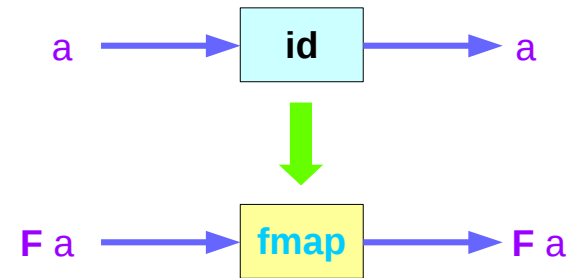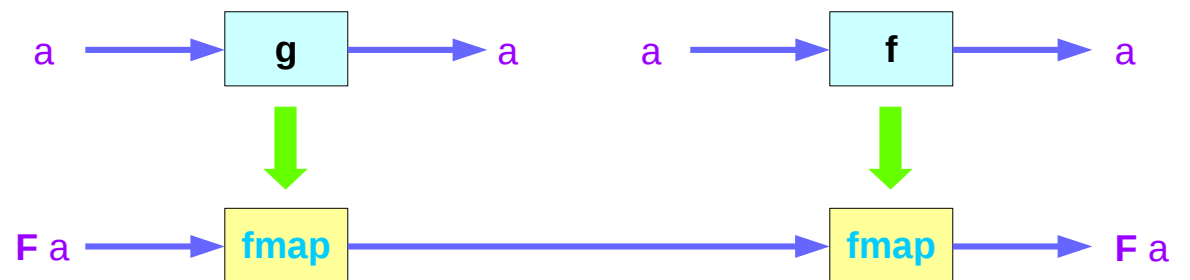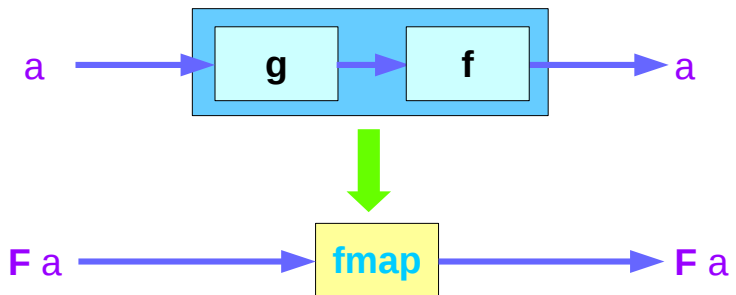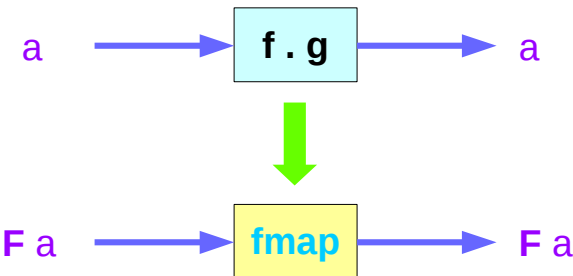
Functor Typeclass

```
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**)

a → **f . g** → a

F a → fmap → **F** a

a → **g** → **f** → a

F a → fmap → **F** a

a → **g** → a   a → **f** → a

F a → fmap → fmap → **F** a

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 **f̲** (**Just** x) = **Just** (**f̲** x)
      fmap **f̲** **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