

# Applicative (2A)

---

Copyright (c) 2016 - 2017 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto: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](https://wiki.haskell.org/Haskell_in_5_steps)

# Currying

Currying recursively transforms  
a function that takes multiple arguments  
into a function that takes just a single argument and  
returns another function if any arguments are still needed.

$f :: a \rightarrow b \rightarrow c$

$f\ x\ y$

$f :: a \rightarrow b \rightarrow c$

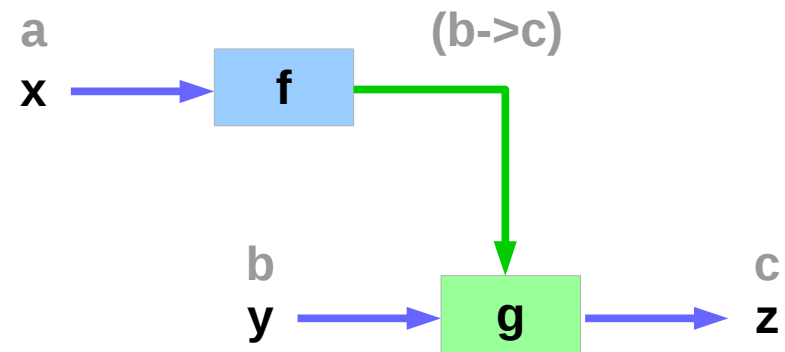
$(f\ x)\ y$

$f :: a \rightarrow (b \rightarrow c)$

$g\ y$

$g :: b \rightarrow c$

$f :: a \rightarrow b \rightarrow c$



<https://wiki.haskell.org/Currying>

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

# Curry & Uncurry

$f :: a \rightarrow b \rightarrow c$  the curried form of  $g :: (a, b) \rightarrow c$

$f = \text{curry } g$   
 $g = \text{uncurry } f$

$f \ x \ y = g \ (x,y)$

the curried form is usually more convenient because it allows **partial application**.

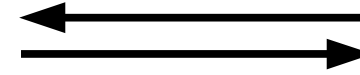
all functions are considered **curried**

all functions take **just one argument**

*the curried form*

$f :: a \rightarrow b \rightarrow c$

currying



$g :: (a, b) \rightarrow c$

uncurrying

$f \ x \ y$

$g \ (x,y)$

<https://wiki.haskell.org/Currying>

# Mapping functions over the Functor [ ] (1)

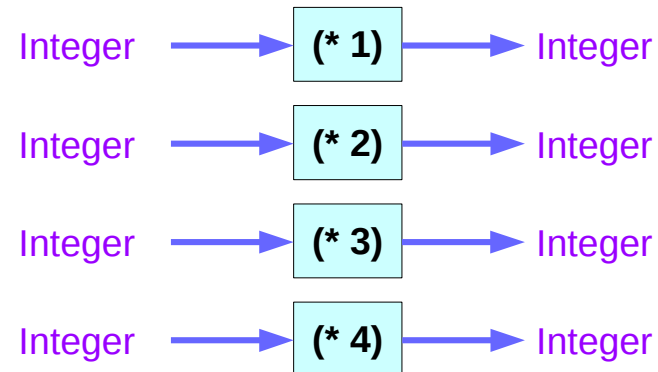
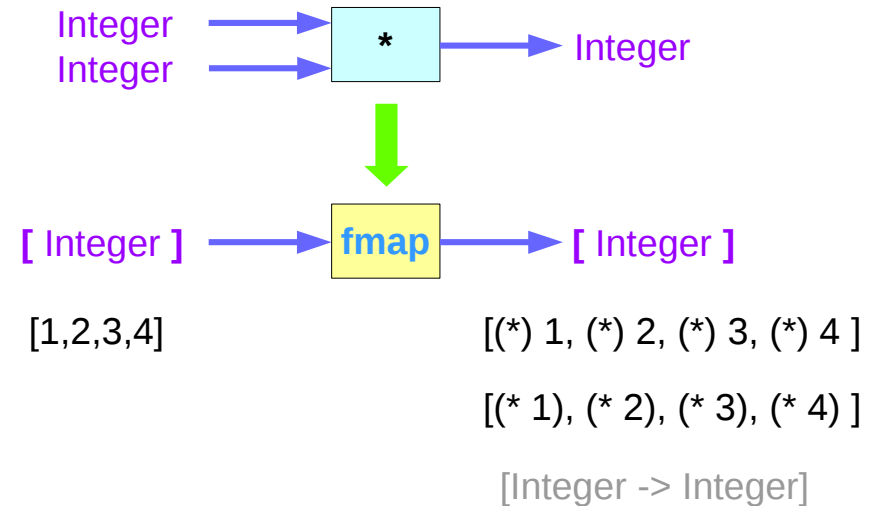
```
ghci> let a = fmap (*) [1,2,3,4]
```

```
ghci> :t a
```

```
a :: [Integer -> Integer]
```

```
ghci> fmap (\f -> f 9) a
```

```
[9,18,27,36]
```



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

# Mapping functions over the Functor [ ] (2)

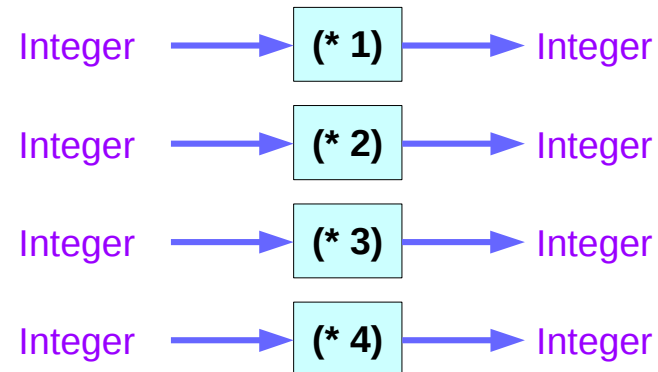
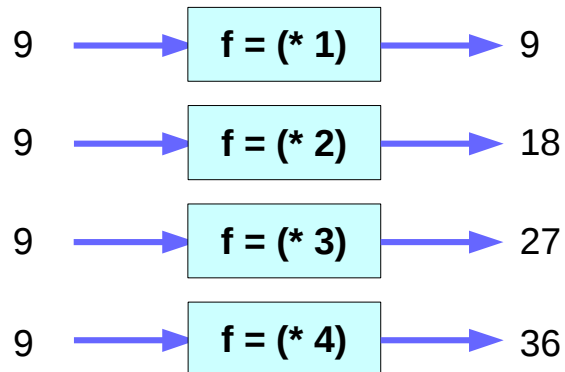
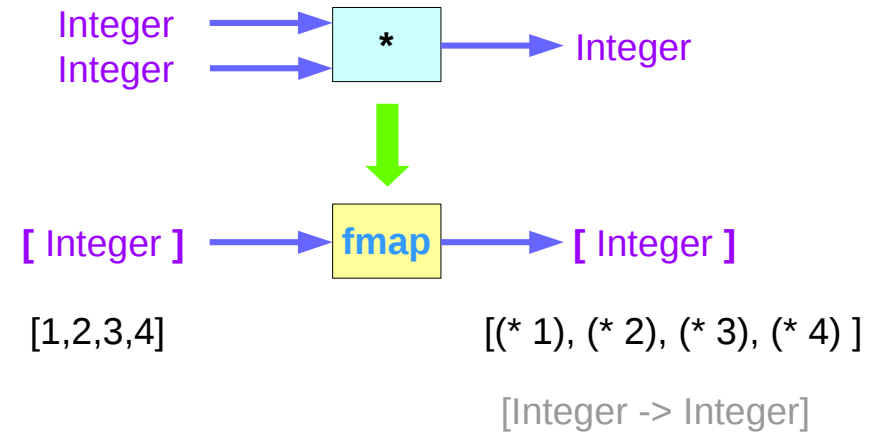
```
ghci> let a = fmap (*) [1,2,3,4]
```

```
ghci> :t a
```

```
a :: [Integer -> Integer]
```

```
ghci> fmap (\f -> f 9) a
```

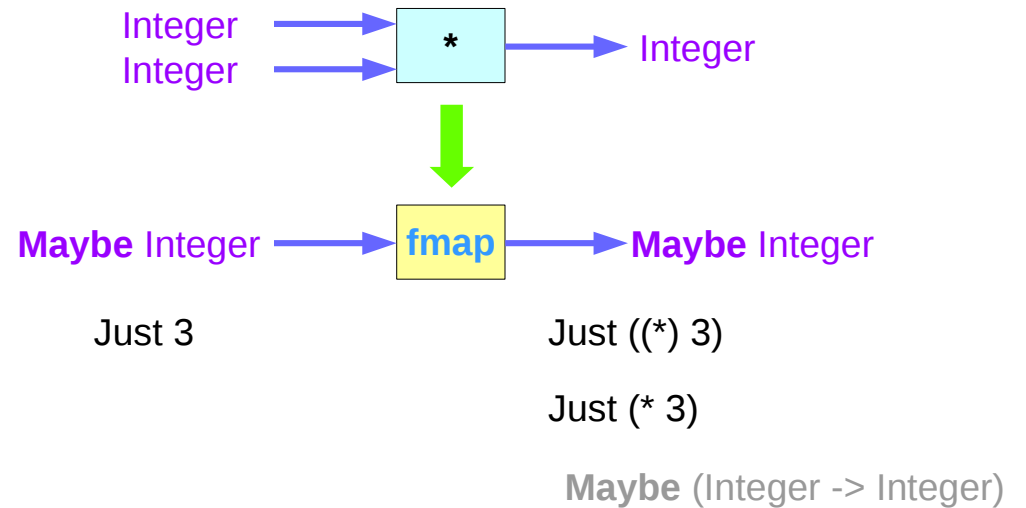
```
[9,18,27,36]
```



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

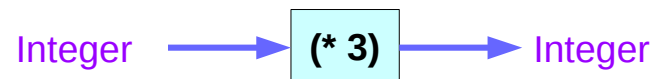
# Mapping functions over the Functor Maybe (1)

fmap (\*) (Just 3)



function wrapped in a **Just**

**Just** (\* 3)



integer wrapped in a **Just**

**Just** 2

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



# Extracting and Mapping a function

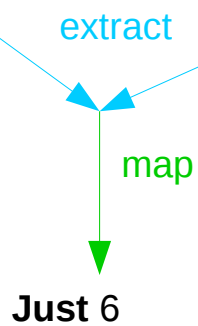
fmap (\*) (Just 3)

function wrapped in a **Just**

**Just (\* 3)**

integer wrapped in a **Just**

**Just 2**



**Just (\* 3) <\*> Just 2**

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

# The Applicative Typeclass

```
class (Functor f) => Applicative f where
```

```
  pure :: a -> f a
```

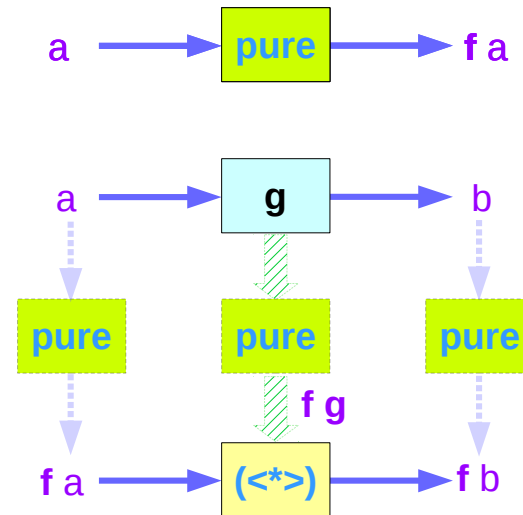
```
  (<*>) :: f (a -> b) -> f a -> f b
```

```
instance Applicative Maybe where
```

```
  pure = Just
```

```
  Nothing <*> _ = Nothing
```

```
  (Just f) <*> something = fmap f something
```



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

# Applicative Functor Examples

ghci> Just (+3) <\*> Just 9

Just 12

ghci> pure (+3) <\*> Just 10

Just 13

ghci> pure (+3) <\*> Just 9

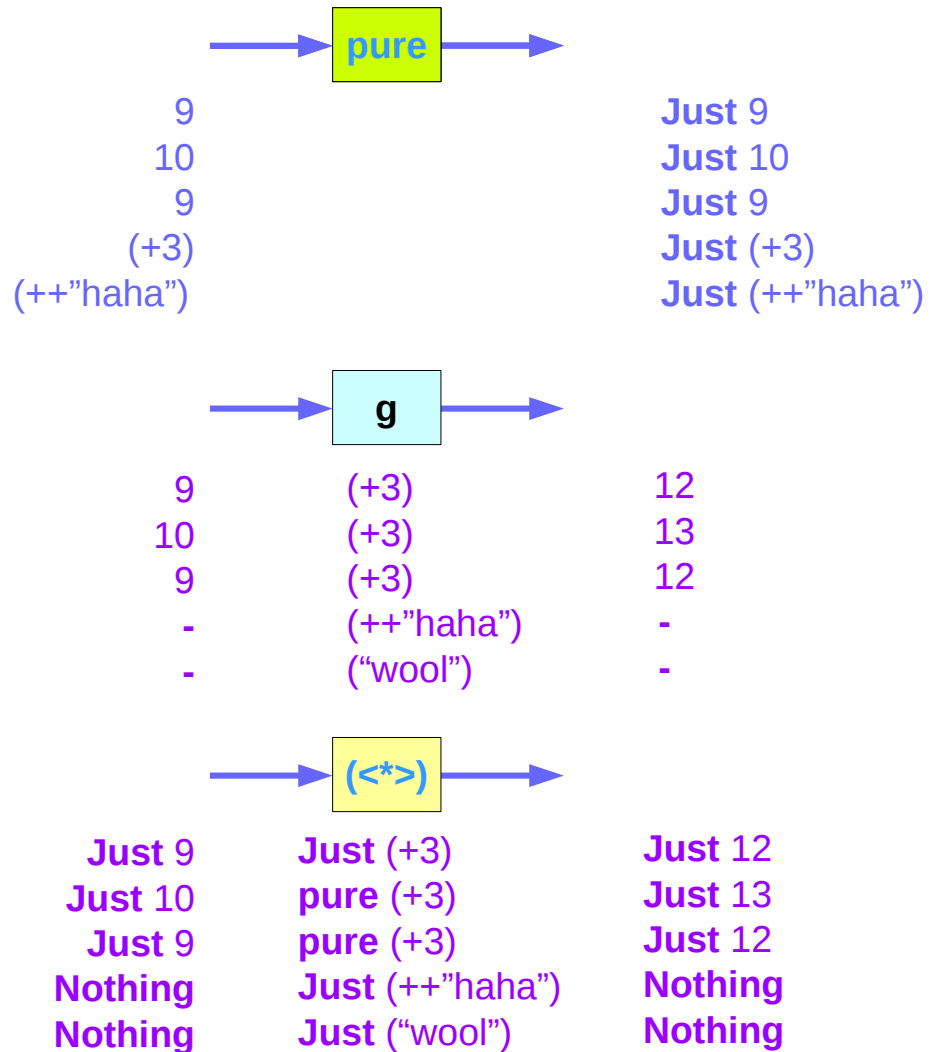
Just 12

ghci> Just (++"hahah") <\*> Nothing

Nothing

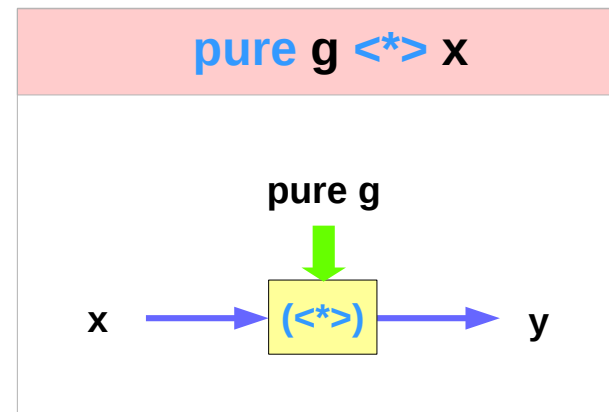
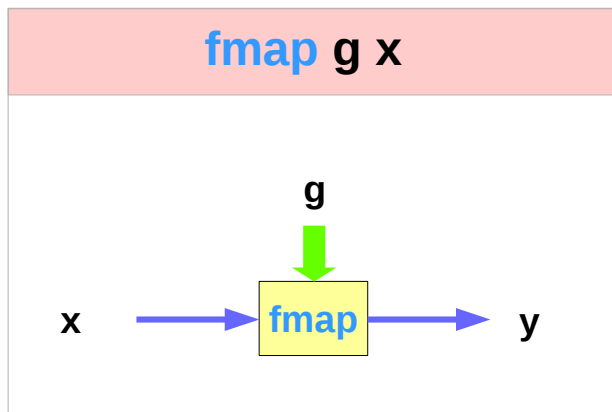
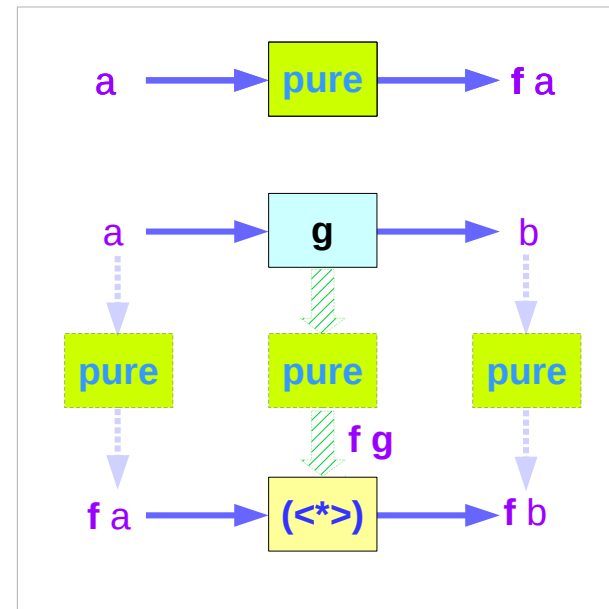
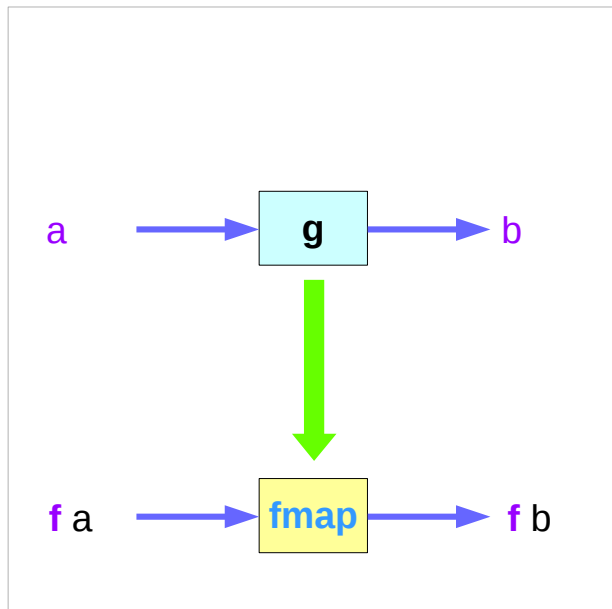
ghci> Nothing <\*> Just "woot"

Nothing



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

$$\text{fmap } g \ x = \text{pure } g \ \langle * \rangle \ x$$



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

# Left Associative <\*>

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

**pure (+) <\*> Just 3 <\*> Just 5**

**pure (+3) <\*> Just 5**

**Just 8**

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

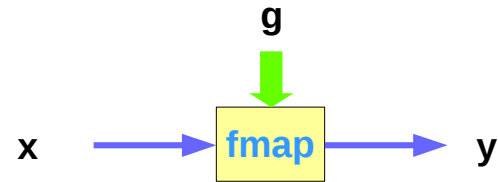
# Infix Operator $\langle \$ \rangle$

`pure f`  $\langle * \rangle$  `x`  $\langle * \rangle$  `y`  $\langle * \rangle$  `z`

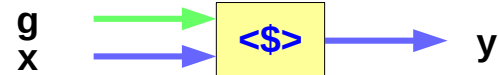
`fmap f` `x`  $\langle * \rangle$  `y`  $\langle * \rangle$  `z`

`f`  $\langle \$ \rangle$  `x`  $\langle * \rangle$  `y`  $\langle * \rangle$  `z`

`fmap g x`



`g`  $\langle \$ \rangle$  `x`



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

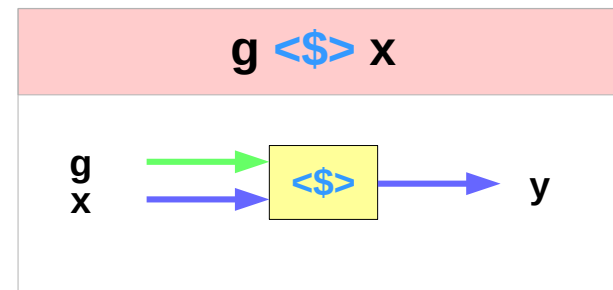
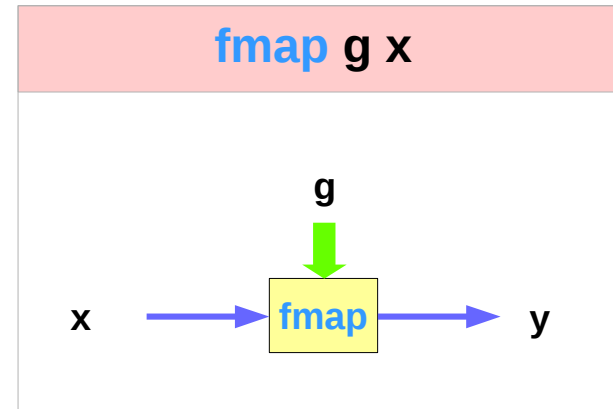
# Infix Operator $\langle \$ \rangle$

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

*not a class method*

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

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



<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>