

Applicative Sequencing (3C)

Copyright (c) 2016 - 2018 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.

This document was produced by using LibreOffice.

Based on

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

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

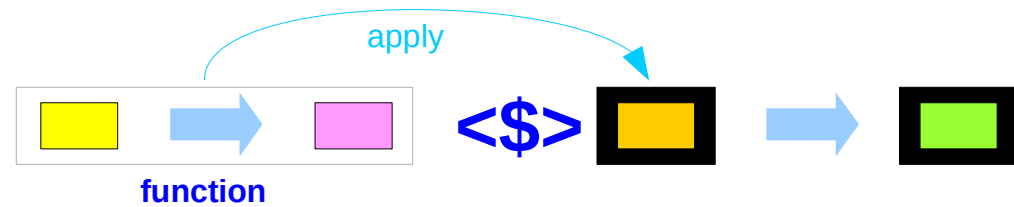
Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

<\$> and <*> operators

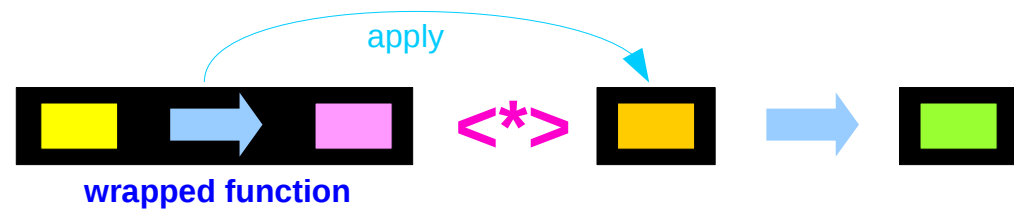
Prelude> (*2) <\$> (Just 3)

Just 6



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

Just 6



<\$> related operators

Functor map <\$>

$(\langle \$ \rangle) :: \text{Functor } f \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b$

$(\langle \$ \rangle) :: \text{Functor } f \Rightarrow a \rightarrow f b \rightarrow f a$

$(\$ \rangle) :: \text{Functor } f \Rightarrow f a \rightarrow b \rightarrow f b$

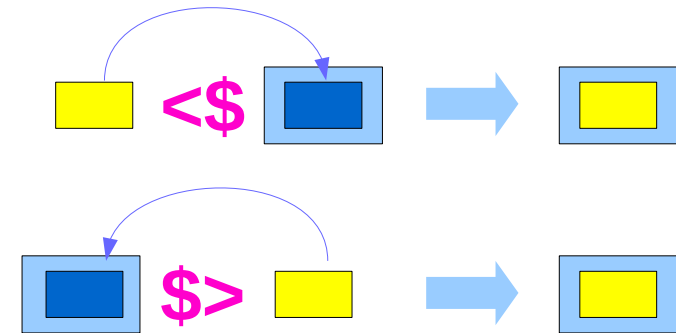
replace **b** in **f b** with **a** ... **f a**

replace **a** in **f a** with **b** ... **f b**

The $\langle \$ \rangle$ operator is just a synonym for the `fmap` function in the Functor typeclass.

`fmap` generalizes `map` for **lists** to other data types : **Maybe**, **IO**, **Map**.

Replacing the core



<https://haskell-lang.org/tutorial/operators>

<\$ / <\$> / \$> operators

there are two additional operators provided
which replace a **value** inside a **Functor**
instead of applying a **function**.

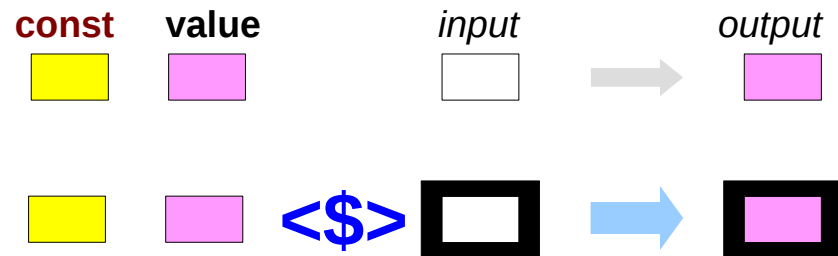
This can be both more convenient in some cases,
as well as for some Functors be more efficient.

value <\$ **functor** = **const value** <\$> **functor**

functor \$> **value** = **const value** <\$> **functor**

$x \text{ <\$ } y = y \text{ \$ } x$ $y :: \text{functor}$

$x \text{ \$ } y = y \text{ <\$ } x$ $x :: \text{functor}$



<https://haskell-lang.org/tutorial/operators>

<\$ / <\$> / \$> operators examples

```
import Data.Functor
```

```
Prelude> Just 1 $> 2
```

```
Just 2
```

```
Prelude> Just 2 $> 1
```

```
Just 1
```

```
Prelude> 1 <$ Just 3
```

```
Just 1
```

```
Prelude> 3 <$ Just 1
```

```
Just 3
```

```
Prelude> 1 <$ Just 3
```

```
Just 1
```

```
Prelude> 3 <$ Just 1
```

```
Just 3
```

```
import Data.Functor
```

```
Prelude> (+1) <$> Just 2
```

```
Just 3
```

```
Prelude> (+1) <$> Just 3
```

```
Just 4
```

```
Prelude> (+1) <$> Nothing
```

```
Nothing
```

```
Prelude> const 2 <$> Just 111
```

```
Just 2
```

<https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/Simple%20examples>

<\$> examples

```
#!/usr/bin/env stack
-- stack --resolver ghc-7.10.3 runghc
import Data.Monoid ((<>))

main :: IO ()
main = do
  putStrLn "Enter your year of birth"
  year <- read <$> getLine
  let age :: Int
      age = 2020 - year
  putStrLn $ "Age in 2020: " <> show age
```

getLine :: IO String

Input: read "12"::Double

Output: 12.0

-- this infix synonym for mappend is found in Data.Monoid
x <> y = mappend x y
infixr 6 <>

<https://haskell-lang.org/tutorial/operators>

<*> related operators

Applicative function application <*>

<*> :: Applicative f => f (a -> b) -> f a -> f b

(*>) :: Applicative f => f a -> f b -> f b

<*> :: Applicative f => f a -> f b -> f a

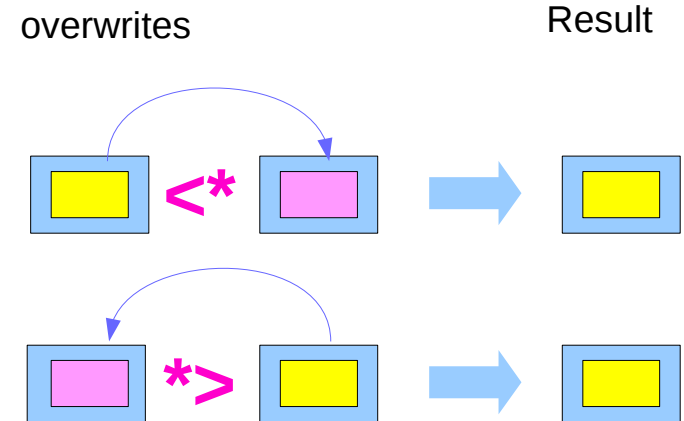
<*> is an operator that applies
a wrapped function to a wrapped value.

<*> is a part of the Applicative typeclass,

<*> is very often used as follows

foo <\$> bar <*> baz

faa <*> bar <*> baz



<https://haskell-lang.org/tutorial/operators>

*> operator

two helper operators

*> ignores the value from the first argument.

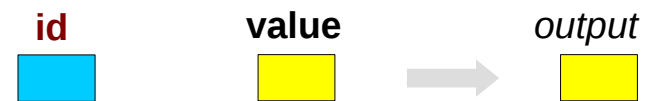
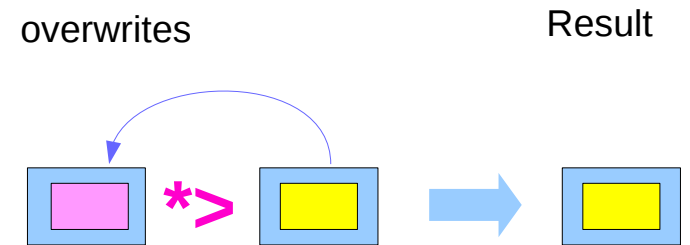
*> is completely equivalent to >> in Monad

$a1 *> a2 = (id \lt \$ a1) \lt * \gt a2$

$a1 *> a2 = do$

`_ <- a1`

`a2`



$(id \lt \$ a1)$



$(id \lt \$ a1) \lt * \gt a2$

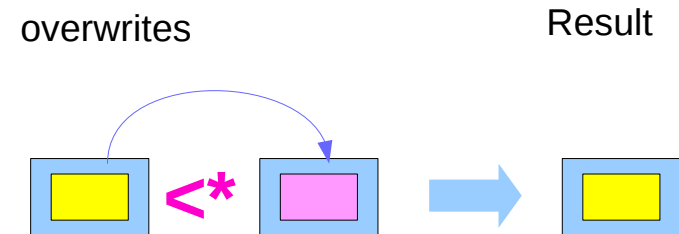


<https://haskell-lang.org/tutorial/operators>

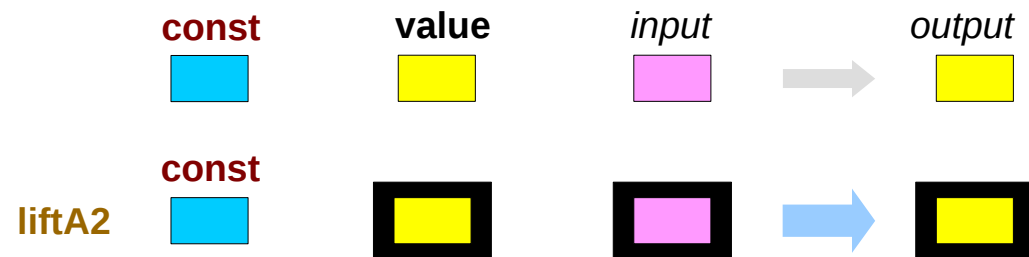
<* operator

<* is the same thing in reverse: perform the first action then the second, but only take the value from the first action.

(<*) = liftA2 const



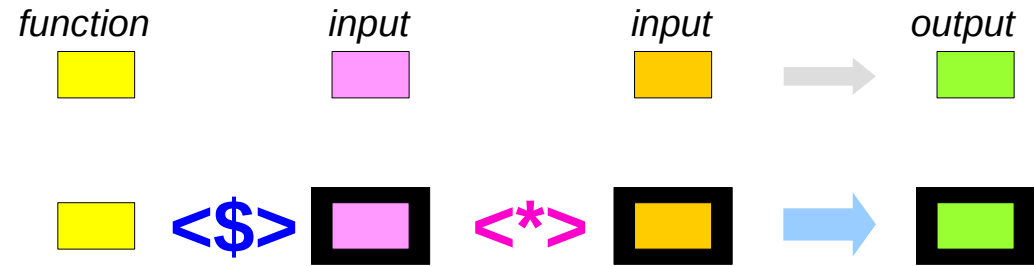
```
a1 <* a2 = do
  res <- a1
  _ <- a2
  return res
```



<https://haskell-lang.org/tutorial/operators>

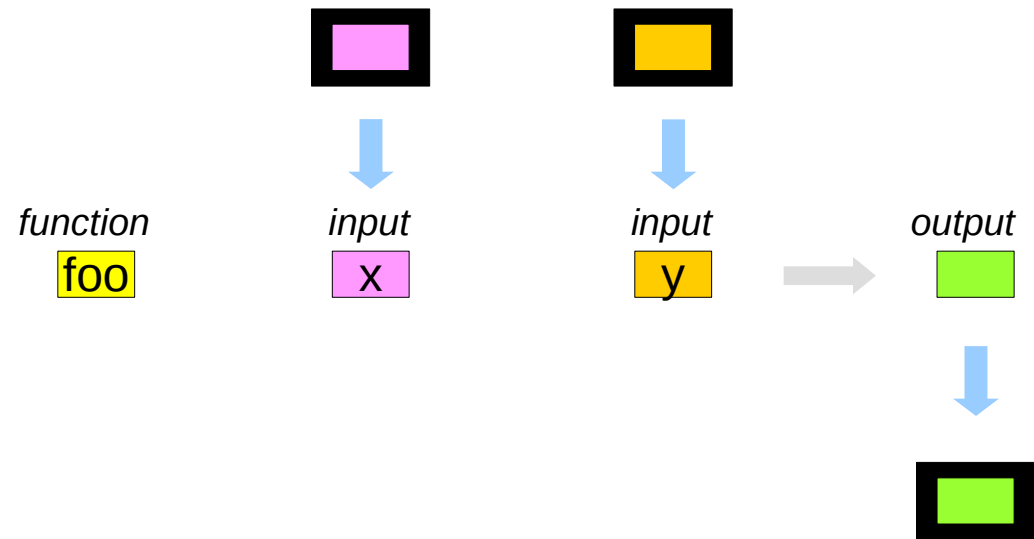
<*> examples

foo <\$> bar <*> baz



With a Monad, this is equivalent to:

```
do x <- bar
  y <- baz
  return (foo x y)
```



<https://haskell-lang.org/tutorial/operators>

<*> examples

examples including parsers and serialization libraries.

using the **aeson** package: (handling **JSON** data)

```
data Person = Person { name :: Text, age :: Int } deriving Show
```

```
-- We expect a JSON object, so we fail at any non-Object value.
```

```
instance FromJSON Person where
```

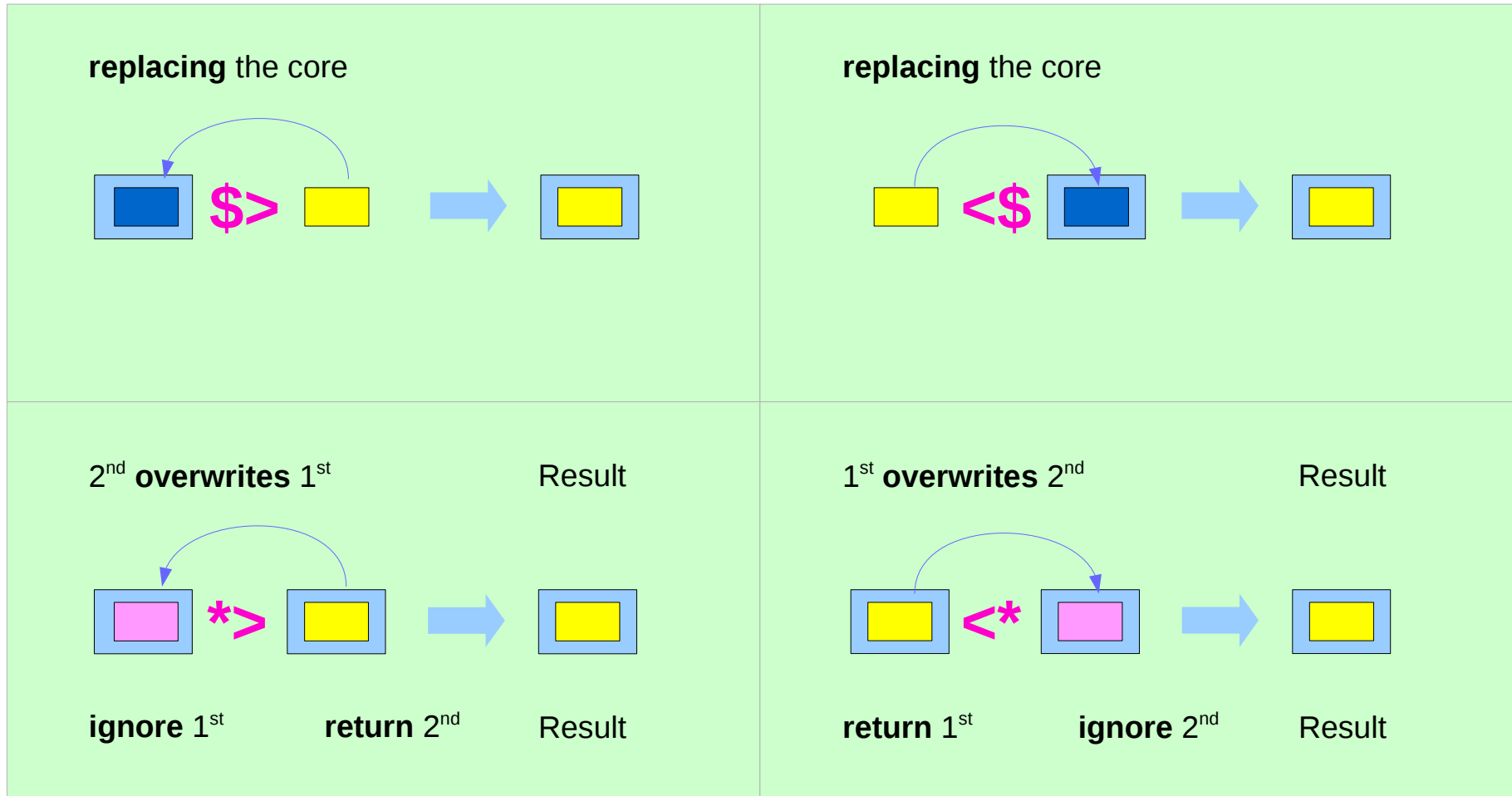
```
  parseJSON (Object v) = Person <$> v .: "name" <*> v .: "age"
```

```
  parseJSON _ = empty
```

- . append-head operator (cons)
- . function composition operators
- . name qualifier

<https://haskell-lang.org/tutorial/operators>

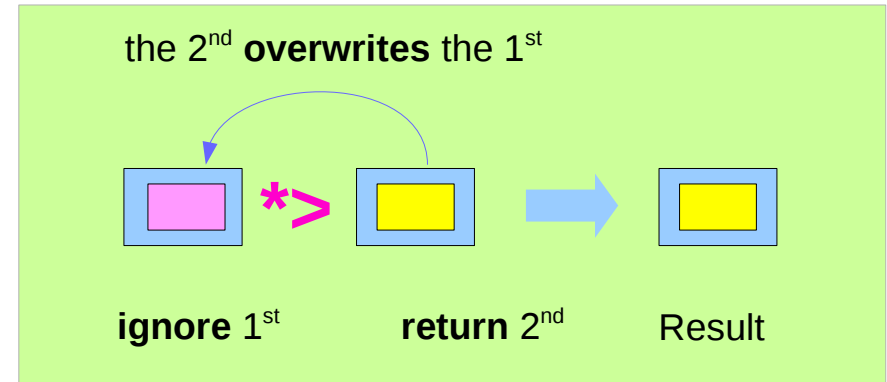
(\$> v.s. <\$) and (*> v.s. <*)



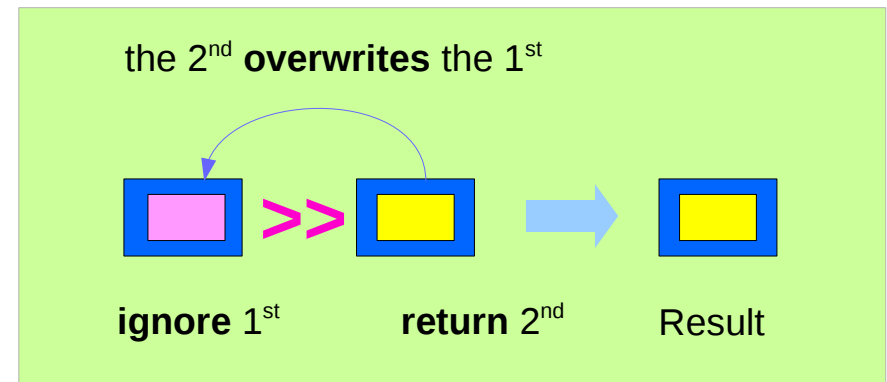
https://en.wikibooks.org/wiki/Haskell/Applicative_functors

(*>) V.S. (>>)

(*>) :: **Applicative** $f \Rightarrow f\ a \rightarrow f\ b \rightarrow f\ b$



(>>) :: **Monad** $m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$



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

(*> v.s. >>) and (pure v.s. return)

```
(*>) :: Applicative f => f a -> f b -> f b  
>> :: Monad m      => m a -> m b -> m b
```

```
pure :: Applicative f => a -> f a  
return :: Monad m    => a -> m a
```

the constraint changes from **Applicative** to **Monad**.

(*>) in Applicative



>>) in Monad

pure in Applicative



return in Monad

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

Commutativity

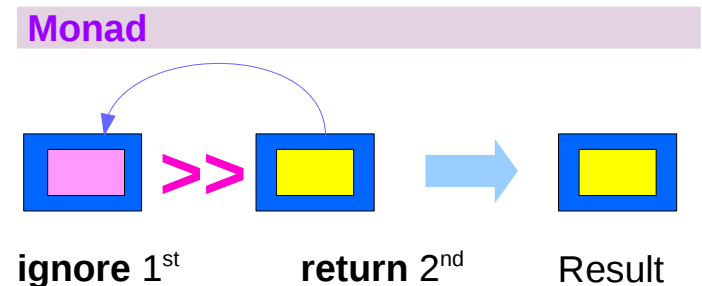
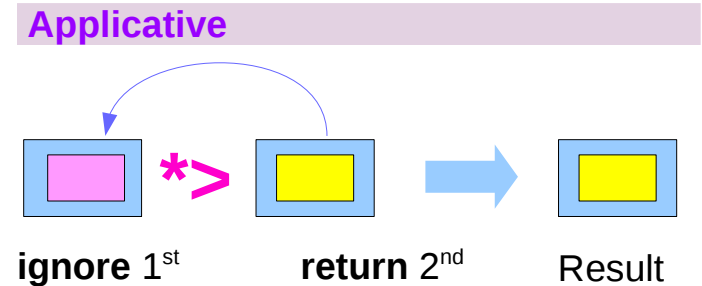
the concept involved in **commutative monads**,
is the same as the one in **commutative applicatives**,
only specialised to **Monad**.

commutativity (or the lack thereof) affects
other functions which are derived from **(<*>)** as well.

(<*>) is a clear example:

(<*>) :: Applicative f => f a -> f b -> f b

(<*>) combines effects
preserves only the **values** of
its second argument.
is equivalent to **(<*>)**, for **monads**



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

Commutativity examples (1)

Prelude> **Just 2** *> **Just 3**

Just 3 – with value

Prelude> **Just 3** *> **Just 2**

Just 2 – with value

Prelude> **Just 2** *> **Nothing**

Nothing – non-value

Prelude> **Nothing** *> **Just 2**

Nothing – non-value

Maybe is commutative

swapping the arguments does not affect the **effects** (the being and nothingness of wrapped values).

for **IO**, however, swapping the arguments does reorder the **effects**:

(*>) combines effects

preserves only the **values** of
its second argument.

is equivalent to **(>)**, for **monads**

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

Commutativity examples (2)

Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)

"foo"

"bar"

3

Prelude> (print "bar" *> pure 3) *> (print "foo" *> pure 2)

"bar"

"foo"

2

Prelude> (print "foo" *> pure 2) <*> (print "bar" *> pure 3)

"foo"

"bar"

2

IO is non-commutative

swapping the arguments
does reorder the effects:

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

Sequencing of Effects

```
Prelude> [(2*), (3*)] <*> [4,5]
```

- 1) [8,10,12,15] -- correct answer
- 2) [8,12,10,15]

The difference is that for the first (and correct) answer

the result is obtained

by taking the skeleton of the first list

and replacing each element

by all possible combinations

with elements of the second list,

```
[(2*), (3*)] <*> [4,5]
```

```
[(2*) <*> 4, (2*) <*> 5, (3*) <*> 4, (3*) <*> 5]
```

while for the other possibility

the starting point is the second list.

```
[(2*), (3*)] <*> [4,5]
```

```
[(2*) <*> 4, (3*) <*> 4, (2*) <*> 5, (3*) <*> 5]
```

sequencing of effects

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

Non-commutative Functors

by **effects** we mean the functorial **context**,
as opposed to the **values** within the functor

some **effects** examples:

the skeleton of a **list**,
actions performed in the real world in **IO**,
the existence of a value in **Maybe**

The existence of two legal implementations of $(\langle * \rangle)$ for lists
only differ in the **sequencing of effects**
 \square is a **non-commutative** applicative functor.

Prelude> $[(2^*), (3^*)] \langle * \rangle [4, 5]$

1) [8,10,12,15]

2) [8,12,10,15]

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

Commutative Functors

a commutative applicative functor is one for which the following holds:

```
liftA2 fn u v = liftA2 (flip fn) v u
```

-- Commutativity

Or, equivalently,

```
fn <$> u <*> v = flip fn <$> v <*> u
```

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

```
fn      :: (a -> b -> c)
```

```
flip fn :: (b -> a -> c)
```

```
u       :: f a
```

```
v       :: f b
```

swapping the arguments does not affect the **effects** as well as the **value**

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

Commutative Monads (1)

```
do
  a <- actA
  b <- actB
  return (a + b)
```

```
do
  b <- actB
  a <- actA
  return (a + b)
```

commutative if the order of **side effects** is not important.

there are many monads that commute (e.g. **Maybe**, **Random**).

If the monad is **commutative**,
then the operations captured within it
can be computed in parallel.

No good syntax for monads that commute
still an open research problem

<https://stackoverflow.com/questions/5897845/relax-ordering-constraints-in-monadic-computation>

Commutative Monads (2)

Commutative monads are monads
for which the order of actions makes no difference
(they commute), that is when following code:

```
do
  a <- actA
  b <- actB
  m a b
```

```
do
  b <- actB
  a <- actA
  m a b
```

commutative if the order of **side effects** is not important.

Examples of commutative include:

Reader monad

Maybe monad

https://wiki.haskell.org/Monad#Commutative_monads





<https://stackoverflow.com/questions/6089997/how-do-i-find-out-whether-a-monad-is-commutative>

Left-to-right sequencing

The convention in Haskell is to always implement `(<*>)` and other applicative operators using **left-to-right sequencing**.

Even though this convention helps reducing confusion, it also means appearances sometimes are misleading.

For instance, the `(<*>)` function is not flip `(*>)`, as it sequences effects from left to right just like `(*>)`:

| | | | | |
|--------------------------|-------------------------------------|--------------------------------------|--|------------------------|
| <code>(<*>)</code> | <code>:: Applicative f =></code> | <code>f (a -> b) -> f a</code> |  | <code>-> f b</code> |
| <code>(*>)</code> | <code>:: Applicative f =></code> | <code>f a -> f b</code> |  | <code>-> f b</code> |
| <code>(<*>)</code> | <code>:: Applicative f =></code> | <code>f a -> f b</code> |  | <code>-> f a</code> |
| <code>(<*>)</code> | <code>:: Applicative f =></code> | <code>f a -> f (a -> b)</code> |  | <code>-> f b</code> |

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

<*> operators

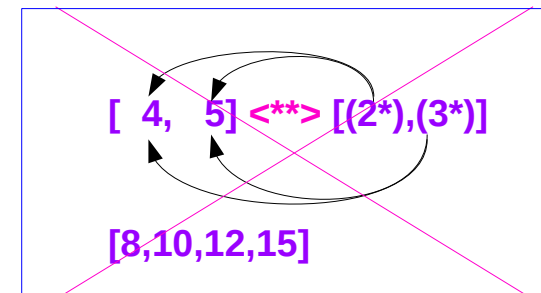
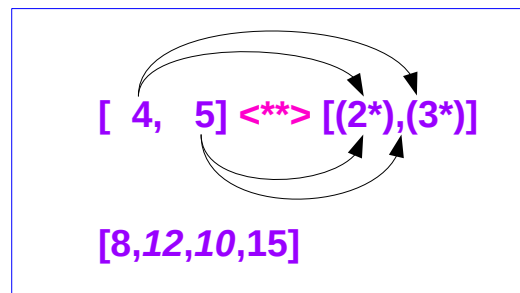
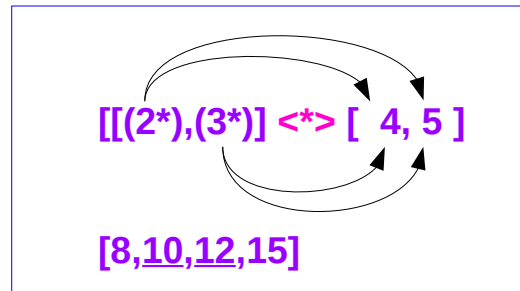
(<*>) :: Applicative f => f a -> f (a -> b) -> f b

(<*>) :: Applicative f => f (a -> b) -> f a -> f b

from **Control.Applicative**

not flip (<*>)

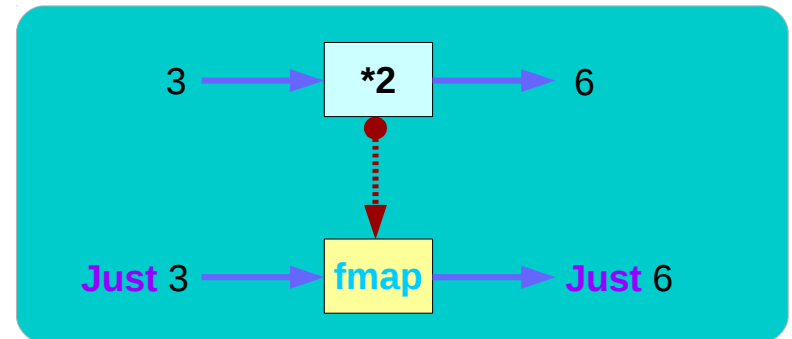
a way of inverting the sequencing



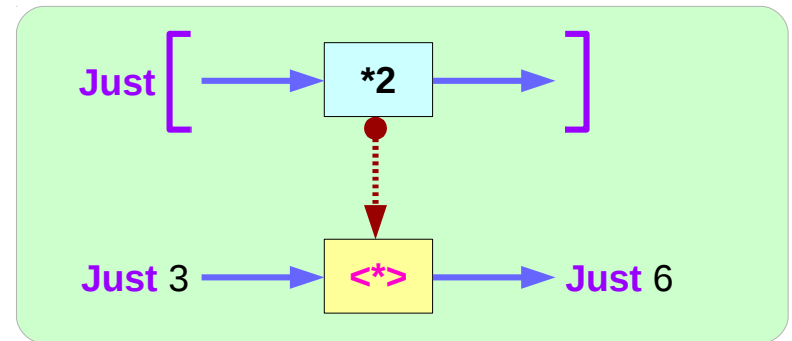
https://en.wikibooks.org/wiki/Haskell/Applicative_functors

Functors, Applicative, and Monad

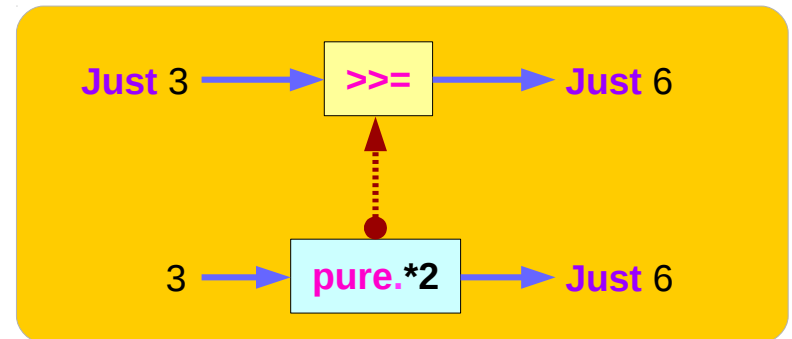
`fmap` :: Functor `f => (a -> b) -> f a -> f b`



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



`(>>=)` :: Monad `m => m a -> (a -> m b) -> m b`



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

Functors, Applicative, and Monad Examples

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

```
Prelude> fmap (*2) (Just 3)
Just 6
```

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

```
Prelude> (Just (*2)) <*> (Just 3)
Just 6
```

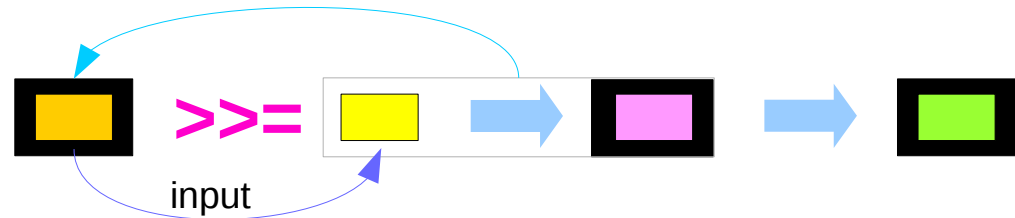
`(>>=)` :: Monad `m => m a -> (a -> m b) -> m b`

```
Prelude> (Just 3) >>= (pure . (*2))
Just 6
Prelude> (Just 3) >>= (return . (*2))
Just 6
```

(=<<) : the flipped version of (>>=)

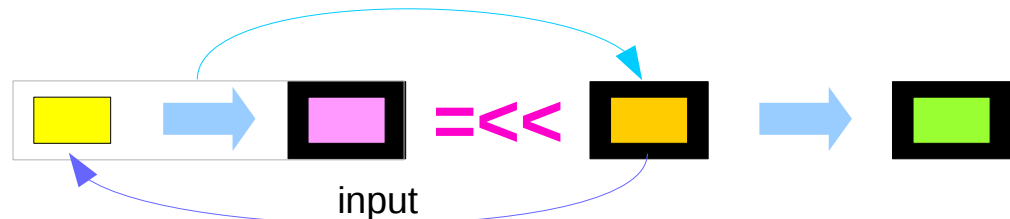
(>>=) :: Monad m => m a -> (a -> m b) -> m b

maps a -> m b function at the right
over monadic ma functors at the left



(=<<) :: Monad m => (a -> m b) -> m a -> m b

maps a -> m b function at the left
over monadic ma functors at the right



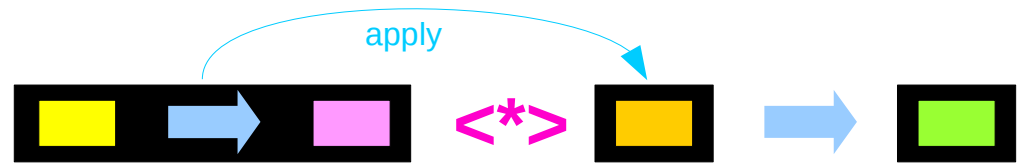
https://en.wikibooks.org/wiki/Haskell/Applicative_functors

<\$>, <*>, >>=, and =<< examples

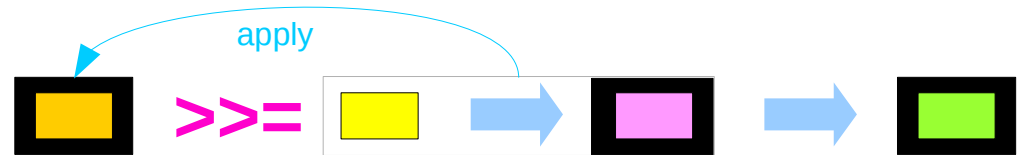
Prelude> (*2) <\$> (Just 3)
Just 6



Prelude> (Just (*2)) <*> (Just 3)
Just 6



Prelude> (Just 3) >>= (pure . (*2))
Just 6



Prelude> (pure . (*2)) =<< (Just 3)
Just 6



Comparing the three characteristic methods

replace `fmap` by its infix synonym, `(<$>)`

replace `(>=)` by its flipped version, `(=<<)`

| | | | | | | | |
|---|----------------|---------------|----------------------------|--------------------|----------------------------|--------------------|------------------|
| <code>fmap</code> :: Functor | <code>f</code> | \Rightarrow | <code>(a -> b)</code> | <code>-></code> | <code>f a</code> | <code>-></code> | <code>f b</code> |
| <code>(<*>)</code> :: Applicative | <code>f</code> | \Rightarrow | <code>f (a -> b)</code> | <code>-></code> | <code>f a</code> | <code>-></code> | <code>f b</code> |
| <code>(>=)</code> :: Monad | <code>m</code> | \Rightarrow | <code>m a</code> | <code>-></code> | <code>(a -> m b)</code> | <code>-></code> | <code>m b</code> |

| | | | | | |
|---|----------------|---------------|----------------------------|--------------------|------------------------------|
| <code>(<\$>)</code> :: Functor | <code>t</code> | \Rightarrow | <code>(a -> b)</code> | <code>-></code> | <code>(t a -> t b)</code> |
| <code>(<*>)</code> :: Applicative | <code>t</code> | \Rightarrow | <code>t (a -> b)</code> | <code>-></code> | <code>(t a -> t b)</code> |
| <code>(=<<)</code> :: Monad | <code>t</code> | \Rightarrow | <code>(a -> t b)</code> | <code>-></code> | <code>(t a -> t b)</code> |

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

All mapping functions over Functors

`fmap`, `<*>` and `=<<` are all **mapping functions** over **Functors**.

The differences between them are in what is being mapped (functions) over in each case:

| | | | | |
|---|----|------------|----|--------------|
| <code><\$></code> :: Functor t | => | (a -> b) | -> | (t a -> t b) |
| <code><*></code> :: Applicative t | => | t (a -> b) | -> | (t a -> t b) |
| <code>=<<</code> :: Monad t | => | (a -> t b) | -> | (t a -> t b) |

`fmap` maps (a -> b) arbitrary functions over functors.

`<*>` maps t (a -> b) morphisms over (applicative) functors.

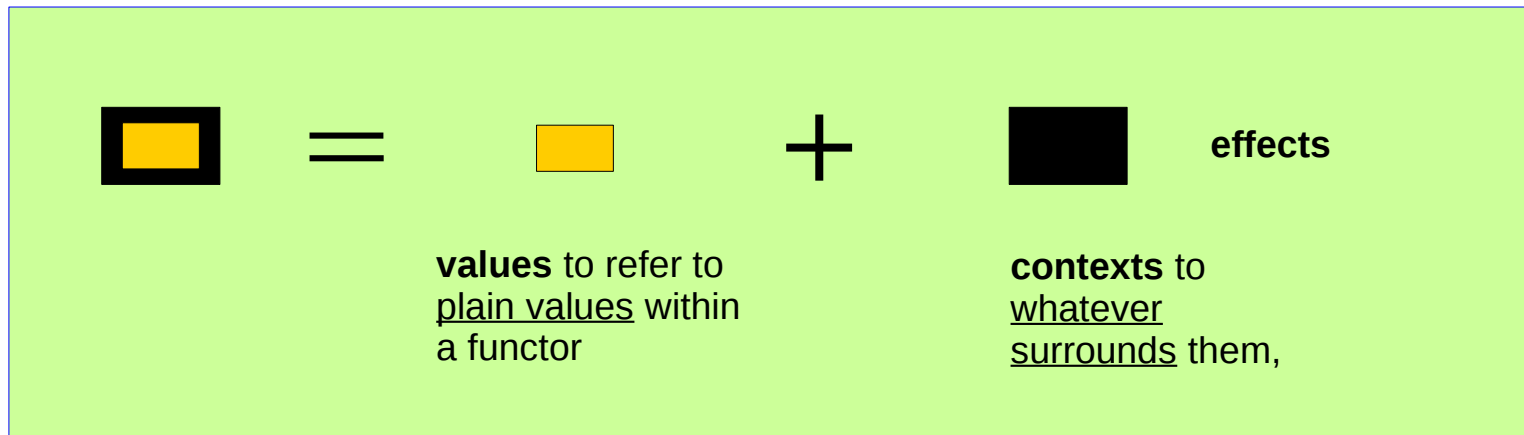
`=<<` maps a -> t b functions over (monadic) functors.

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

Power, Flexibility, Control

The differences of **Functor**, **Applicative** and **Monad** follow from what these three **mapping functions** allow you to do.

As you move from **fmap** to **(<*>)** and then to **(>>=)**, you gain in power, versatility and control, at the cost of guarantees about the results.



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

fmap does not change in the context

The type of `fmap` ensures that it is impossible to use it to **change the context**, no matter which function it is given.

In $(a \rightarrow b) \rightarrow t a \rightarrow t b$, the $(a \rightarrow b)$ function has nothing to do with the t context of the t a functorial value, and so applying it cannot affect the t context.

For that reason, if you do `fmap f xs` on some list `xs` the number of elements of the list will never change.

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

(<*>) changes the context

fmap cannot change the **context**

- the **(a -> b)** function has no relation with the **t context**
- the application of this function does not affect the **context t**
- the number of elements of the list will never change

```
Prelude> fmap (2*) [2,5,6]           a list with 3 elements  
[4,10,12]                          a list with 3 elements
```

That could be a safety guarantee or an unfortunate restriction depending on your purpose

(<*>) is clearly able to change the **context**:

```
Prelude> [(2*), (3*)] <*> [2,5,6]   two lists each with 3 elements  
[4,10,12,6,15,18]                  a list with 6 elements
```

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

(**<*>**) carries a context

The **t (a -> b)** morphism carries a **context** of its own, which is combined (applied) with the **context** of the **t** a **functorial value (a -> b)**.

(**<*>**), however, is subject to a more subtle restriction

while **t (a -> b)** morphisms carry context, within them there are plain (a -> b), which are still unable to modify the **context**.

this means the changes to the **context** (**<*>**) performs are fully determined by the **context** of its **arguments**, and the **values** have no influence over the resulting context.

t (a->b) or **t b**
(a->b) or **a**

```
Prelude> [(2*), (3*)] <*> [2,5,6]  
[4,10,12,6,15,18]
```

two lists each with 3 elements

a list with 6 elements

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

Carrying a context examples

```
Prelude> (print "foo" *> pure (2*)) <*> (print "bar" *> pure 3)      (pure (2*)) <*> (pure 3)
```

```
"foo"
```

```
"bar"
```

```
6
```

```
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)          (pure 2) *> (pure 3)
```

```
"foo"
```

```
"bar"
```

```
3
```

```
Prelude> (print "foo" *> pure undefined) *> (print "bar" *> pure 3)
```

```
"foo"                                (pure undefined) *> (pure 3)
```

```
"bar"
```

```
3
```

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

(>>=) creates a context

```
Prelude> [(2*), (3*)] <*> [2,5,6]  
[4,10,12,6,15,18]
```

*two lists each with 3 elements
a list with 6 elements*

with **list (<*>)** you know that the length of the resulting list will be the product of the lengths of the original lists,

with **IO (<*>)** you know that all real world effect will happen as long as the evaluation terminates, and so forth.

with **Monad**, however, it is very different

(>>=) takes a **(a -> t b)** function,
and so it is able to create context
from **values**
which means a lot of flexibility:

creating context t
a -> t b

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

Creating a context examples

```
Prelude> [1,2,5] >>= \x -> replicate x x      [ replicate 1 1, replicate 2 2, replicate 5,5 ]  
[1,2,2,5,5,5,5,5]
```

```
Prelude> [0,0,0] >>= \x -> replicate x x      [ replicate 0 0, replicate 0 0, replicate 0,0 ]  
[]
```

```
Prelude> return 3 >>= \x -> print $ if x < 10 then "Too small" else "OK"  
"Too small"
```

```
Prelude> return 42 >>= \x -> print $ if x < 10 then "Too small" else "OK"  
"OK"
```

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

Deciding context (1)

$(\langle * \rangle) :: m (a \rightarrow b) \rightarrow (m a \rightarrow m b)$

$(= \langle \langle \rangle) :: (a \rightarrow m b) \rightarrow (m a \rightarrow m b)$

In both cases there is $m a$, but only in the second case $m a$ can decide whether the function $(a \rightarrow m b)$ gets applied.

In its turn, the function $(a \rightarrow m b)$ can "decide" whether the function bound next gets applied by producing such $m b$ that does not "contain" b (like \square , **Nothing** or **Left**).

In **Applicative** there is no way for functions "inside" $m (a \rightarrow b)$ to make such "decisions" - they always produce a **value** of type b .

<https://stackoverflow.com/questions/23342184/difference-between-monad-and-applicative-in-haskell>

Deciding context (2)

f 1 = Nothing

-- here f "decides" to produce Nothing

f x = Just x

-- if the argument is 1, then Nothing

Just 1 >>= f >>= g

-- g doesn't get applied, because f decided so.

-- f gets 1 and returns Nothing

In **Applicative** this is not possible, no example can be shown.

The closest is:

f 1 = 0

f x = x

g <\$> f <\$> Just 1

-- f gets 1 and produces Just 0, g

-- but f can't stop from getting applied

<https://stackoverflow.com/questions/23342184/difference-between-monad-and-applicative-in-haskell>

Flexibility

the extra flexibility

the less guarantees about

- whether your functions are able to unexpectedly erase parts of a data structure for pathological inputs
- whether the control flow in your application remains intelligible

performance implications

- the complex data dependencies of monadic codes might prevent refactoring and **optimizations**.

use only as much power as needed

it is often good to check

whether **Applicative** or **Functor** are sufficient

just before using **Monad**.

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

Monadic binding / composition operators

$(>=>)$:: Monad $m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
 $(=<<)$:: Monad $m \Rightarrow (a \rightarrow m\ b) \rightarrow m\ a \rightarrow m\ b$
 $(>>)$:: Monad $m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$
 $(>=>)$:: Monad $m \Rightarrow (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c)$
 $(<=<)$:: Monad $m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c)$

<https://haskell-lang.org/tutorial/operators>

Monadic binding operators (1)

```
(>>=) :: Monad m => m a      -> (a -> m b) -> m b
(=<<=) :: Monad m => (a -> m b) -> m a      -> m b
(>>)  :: Monad m => m a      -> m b      -> m b
```

monadic binding operators

The two most basic are `>>=` and `>>`

`>>=`, `>>`, `=<<=` can be expressed in **do-notation**

`>>` is just a synonym for `*>` from the **Applicative** class

`=<<=` is just `>>=` with the arguments reversed

<https://haskell-lang.org/tutorial/operators>

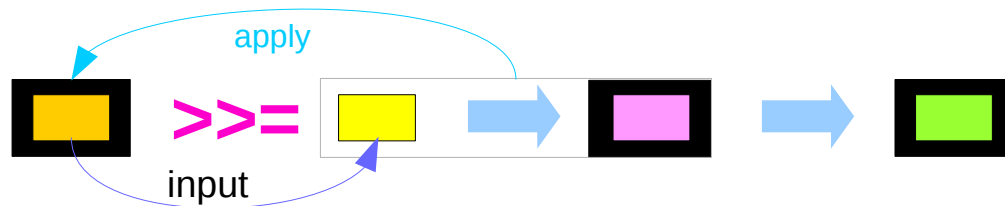
Monadic binding operators (2)

$(>>=)$:: Monad m => m a -> (a -> m b) -> m b

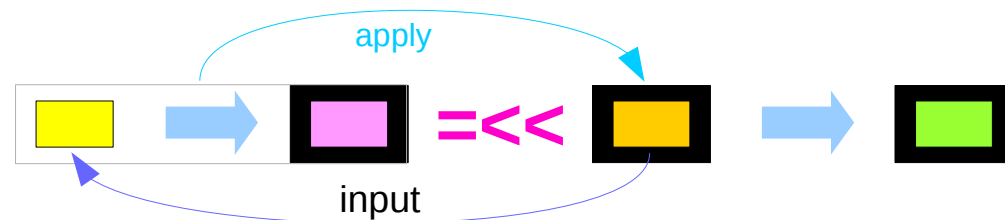
$(=<<)$:: Monad m => (a -> m b) -> m a -> m b

$(>>)$:: Monad m => m a -> m b -> m b

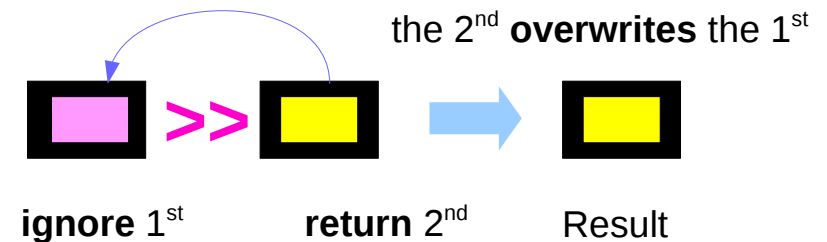
Prelude> (Just 3) >>= (pure . (*2))
Just 6



Prelude> (pure . (*2)) =<< (Just 3)
Just 6



Prelude> (Just 2) >> (Just 3)
Just 3



<https://haskell-lang.org/tutorial/operators>

Monadic binding operators (3)

`(>>=)` :: Monad m => m a -> (a -> m b) -> m b

`(<<=)` :: Monad m => (a -> m b) -> m a -> m b

`(>>)` :: Monad m => m a -> m b -> m b

`m1 >>= func = do`

`x <- m1` -- extract the value

`func x`

`m1 :: m a`

`x :: a`

`func :: a -> m b`

`m1 >> m2 = do`

`_ <- m1` -- side effect only, ignore the value

`m2`

`m1 :: m a`

`m2 :: m b`

`func <<= m1 = do`

`x <- m1` -- extract the value

`func x`

`m1 :: m a`

`x :: a`

`func :: a -> m b`

<https://haskell-lang.org/tutorial/operators>

Monadic composition operators (1)

`(>=>)` :: Monad `m` => (a -> m b) -> (b -> m c) -> (a -> m c)

`(<=<)` :: Monad `m` => (b -> m c) -> (a -> m b) -> (a -> m c)

composition operators for two monadic functions

`>=>=`, `<=<` can be expressed in **do-notation**

`>=>` **pipes** the **result** from the left side to the right side

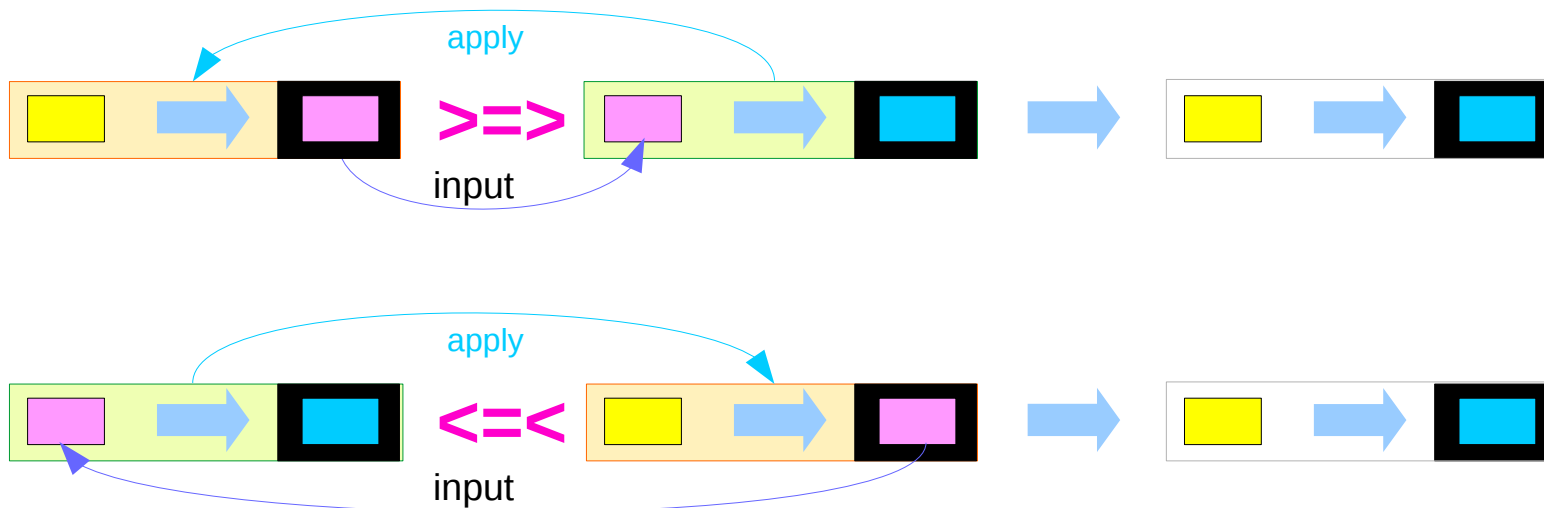
`<=<` **pipes** the **result** from the right side to the left side

<https://haskell-lang.org/tutorial/operators>

Monadic binding operators (2)

$(>=>)$:: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)

$(<=<)$:: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)



```
Prelude Control.Monad> ( (\x -> Just (x+0.5)) >=> (\x -> Just (x*2)) ) 3
```

Just 7.0

```
Prelude Control.Monad> ( (\x -> Just (x+0.5)) <=< (\x -> Just (x*2)) ) 3
```

Just 6.5

<https://haskell-lang.org/tutorial/operators>

Monadic composition operators (3)

$(>=>)$:: Monad $m \Rightarrow (a \rightarrow m b) \rightarrow (b \rightarrow m c) \rightarrow (a \rightarrow m c)$

$(<=<)$:: Monad $m \Rightarrow (b \rightarrow m c) \rightarrow (a \rightarrow m b) \rightarrow (a \rightarrow m c)$

| | | | |
|--|---------------------------|-----------|--------------|
| $f >=> g = \backslash x \rightarrow \text{do}$ | $f :: a \rightarrow m b,$ | $x :: a,$ | $f x :: m b$ |
| $y <- f x$ | $g :: b \rightarrow m c,$ | $y :: b,$ | $g y :: m c$ |
| $g y$ | | | |

| | | | |
|--|---------------------------|-----------|--------------|
| $g <=< f = \backslash x \rightarrow \text{do}$ | $f :: a \rightarrow m b,$ | $x :: a,$ | $f x :: m b$ |
| $y <- f x$ | $g :: b \rightarrow m c,$ | $y :: b,$ | $g y :: m c$ |
| $g y$ | | | |

| | |
|---------------------|-----------|
| $f >=> g = g <=< f$ | First f |
| $g >=> f = f <=< g$ | Then g |

<https://haskell-lang.org/tutorial/operators>

References

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