Function Monad (10A)

Young Won Lim
8/6/18

Please send corrections (or suggestions) to youngwlim@hotmail.com.

Based on

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

Prefix vs Infix Functions

(+) 1 2 -- prefix function
(*) 3 4 -- prefix function

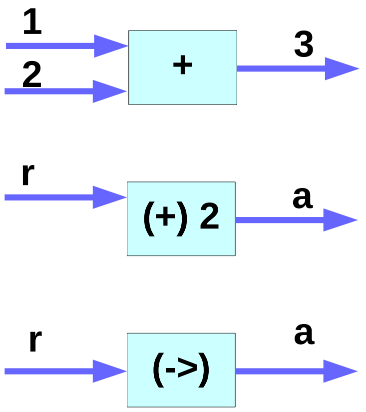
1 + 2 -- infix function
3 * 4 -- infix function

1, 2, 3, 4 : values

(->) r a -- prefix function

r -> a -- infix function

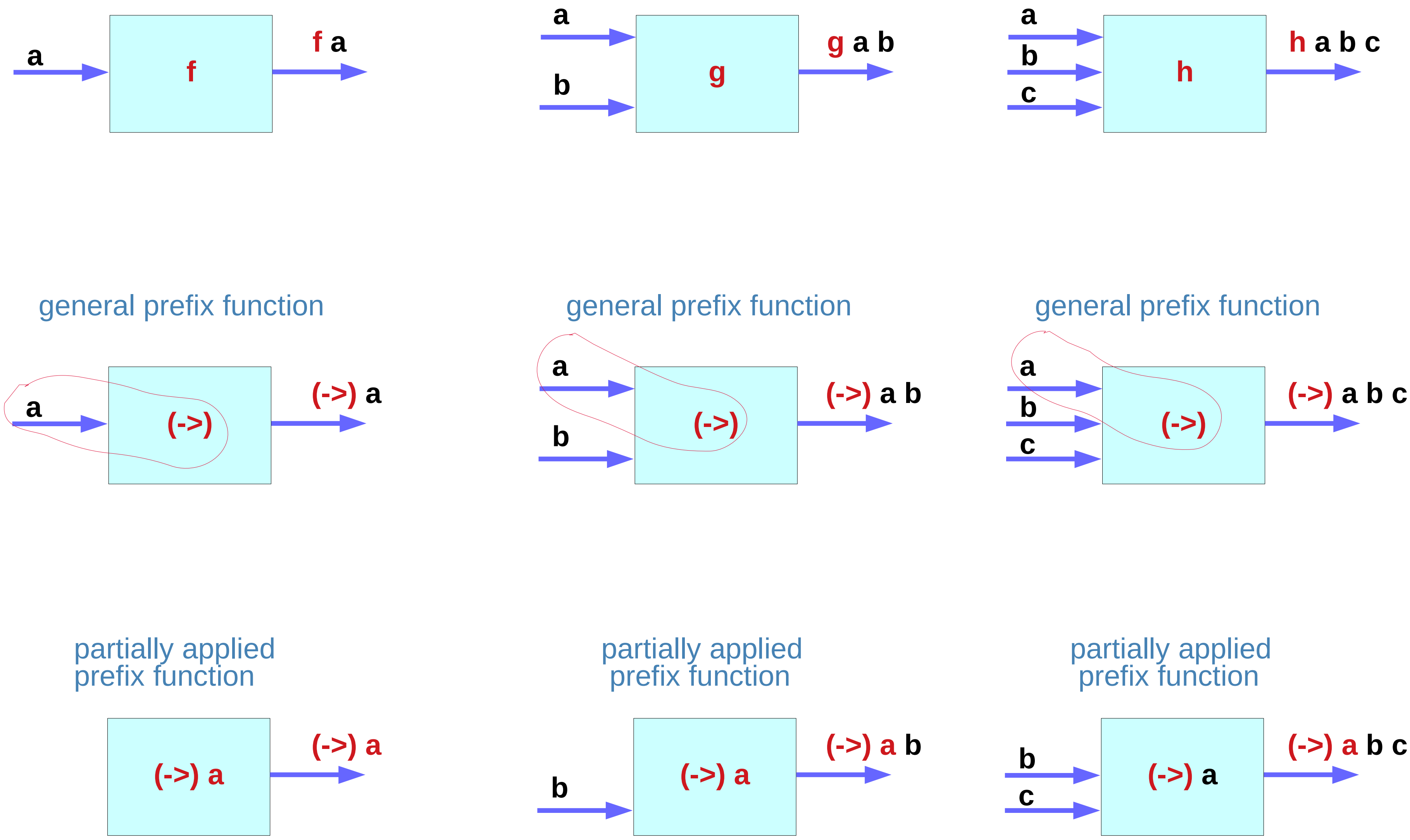
r, a : types



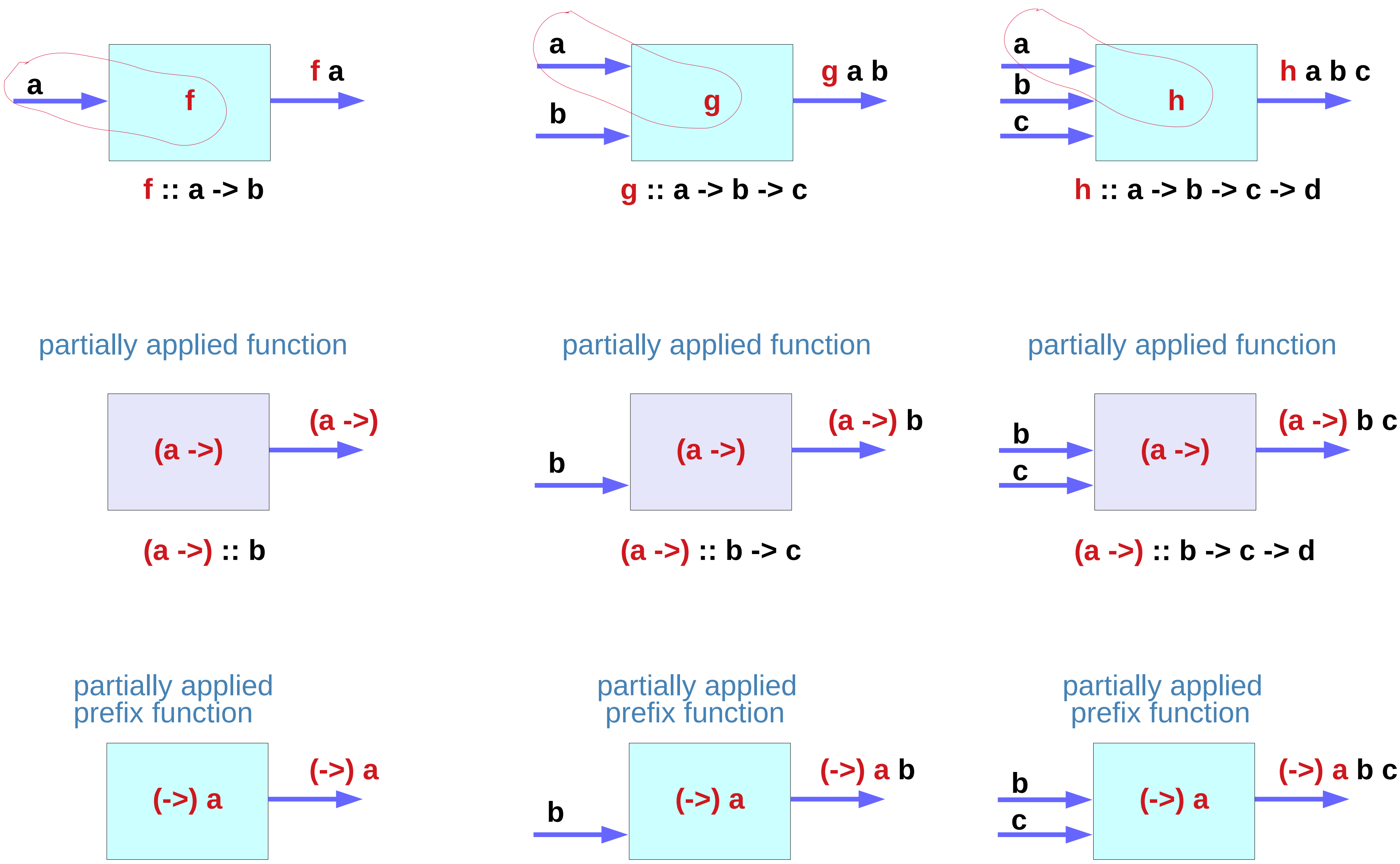
2 + 3 =	(2 +) 3 =	(+) 2 3
r -> a =	(r ->) a =	(->) r a
infix	partial app	prefix

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

Using Prefix Function **(->)**



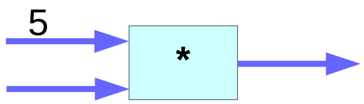
Using Partially Applied Function **(a ->)**

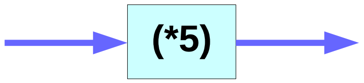


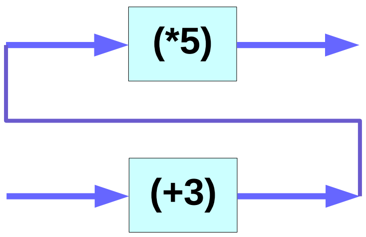
(->) r Prefix Function

```
let f = (*5)
let g = (+3)
(fmap f g) 8

(*5) ((+3) 8) = (*5) 11 = 55
```

```
let f = (+) <$> (*2) <*> (+10)
f 3

(*2) 3 = 6
(+10) 3 = 13
(+) 6 13 = 19
```







(fmap (*5) (+3))

http://learnyouahaskell.com/for-a-few-monads-more#reader

(->) r Monad – return type signature

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

The **monad instance**

return converts any **value** of type **a**
to a **lifted value** of type **r->a**
(application of type constructor **((->) r)** to **a**)

the **signature** of **return**: **return ::**

a -> (r -> a)
a -> r -> a

can remove the **parentheses**
(arrow is **right associative**)

a **function** that takes two **arguments** of types **a** and **r**,
and returns the **value** of type **a**.

https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

(->) r Monad – return definition

a **function** that takes two **arguments** of types **a** and **r**,
and returns the **value** of type **a**.

a -> r -> a

There's only one possibility implementation

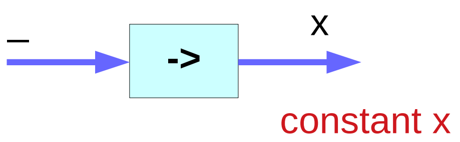
it must ignore the **second argument** — **r** in **a->r**
and **return** the **first**: — **a** in **a->r**

return x y = x — **x :: a**, **y :: r**

const is such a function in Prelude

return = const

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```



https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

(->) r Monad – **>>=** type signature

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

the **type signature** of **>>=**

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

replacing **m** with the type constructor (**r ->**):

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

We need to produce a **function** from **r** to **b** (**h>>=f :: r->b**)
using **functions h :: r -> a** and **f :: a -> r -> b**

https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

(->) r Monad – >>= return definition

```
h >>= f = \w -> f (h w) w

        h :: (r -> a)                    h r :: a
        f  :: (a -> r -> b)              f a r :: b
h >>= f  :: (r -> b)

(>>=) :: (r -> a) -> (a -> r -> b) -> (r -> b)
```

We can use **h** to produce the first **argument a** to **f**
and then provide the second **argument r** to **f**:

```
\r -> f (h r) r                  f (h r) r :: f a b
\w -> f (h w) w                  f (h w) w :: f a b
```

```
instance Monad ((->) r) where
   return x = \_ -> x
   h >>= f = \w -> f (h w) w
```

https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

(->) r Monad – >>= bind operator

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

use **>>=** to feed a monadic **value** **h** to a **function f**,
the **result** is always a monadic **value**

feed a **function** (a monadic value) to another **function f**,
the **result** is a **function** (a monadic value) as well

that's why the result starts off as a **lambda**
anonymous function : lambda abstraction

h :: ((->)r) a
h>>=f :: ((->)r) b

h :: r->a
h>>=f :: r->b

http://learnyouahaskell.com/for-a-few-monads-more#reader

(->) r Monad – >>= bind operator example

((*2) >>= (+)) 3

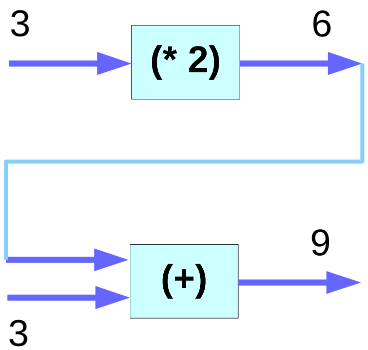
(+) ((*2) 3) 3

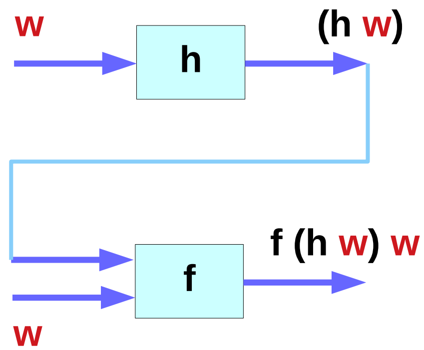
(+6) 3

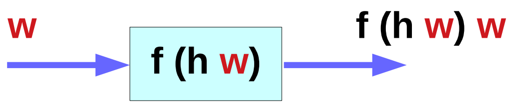
(h >>= f) w

f (h w) w

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
    h >>= f = \r -> f (h r) r
```







http://learnyouahaskell.com/for-a-few-monads-more#reader

(->) r Monad – >>= monadic values

(h >>= f) r

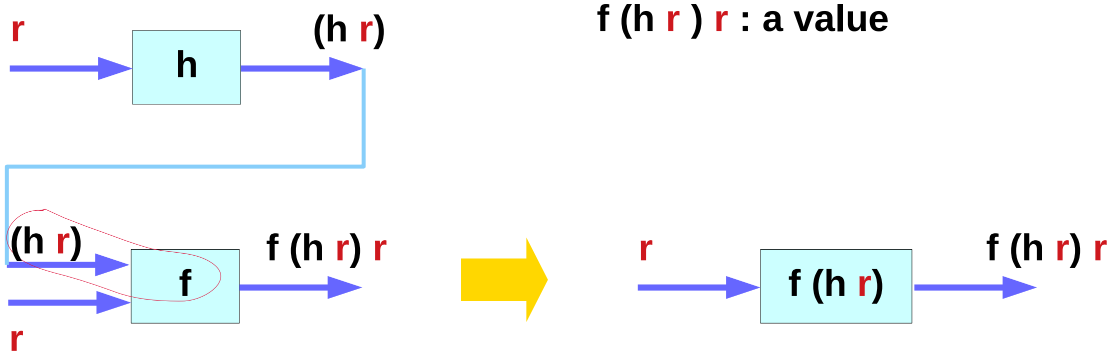
f (h r) r

(+6) 3

(h >>= f) r = f (h r) r

h : a monadic value
r : a value
h r : a value

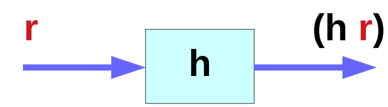
f : a function
f (h r) : a monadic value
f (h r) r : a value



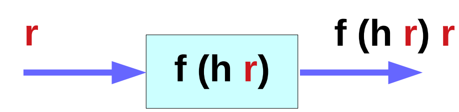
partial application

instance Monad ((->) r) where
    return x = _ -> x
    h >>= f = \w -> f (h w) w

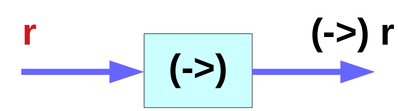
a function : a monadic value



a function : a monadic value



a function monad



http://learnyouahaskell.com/for-a-few-monads-more#reader

((->) r) Monad – >>= bind operation

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

>>= somehow isolates the **result** (**h w**) from the monadic **value h** and then applys the **function f** to that **result**. **f (h w)**

to get the **result (h w)** from a **function h**, we have to apply **h** to **w**

f returns a monadic **value**, which is a **function** in this case, so we apply the returned **function f (h w)** to **w** as well.

(h >>= f) r = f (h r) r

h : a monadic value
r : a value
h r : a value

f : a function
f (h r) : a monadic value
f (h r) r : a value

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

h :: r->a
f :: a -> r->b
h>>=f :: r->b

http://learnyouahaskell.com/for-a-few-monads-more#reader

Function Monad – bind operator examples

```
((*2) >>= (+)) 3
9
(+) ((*2) 3) 3
(+6) 3

((+10) >>= (+)) 3
16
(+) ((+10) 3) 3
(+13) 3

((+10) >>= (*)) 3
39
(*) ((+10) 3)  3
(*13) 3
```

(h >>= f) r = f (h r) r

h : a monadic value
r : a value
h r : a value

f : a function
f (h r) : a monadic value
f (h r) r : a value

addStuff

```
import Control.Monad.Instances

addStuff :: Int -> Int
addStuff = do
   a <- (*2)
   b <- (+10)
   return (a+b)
```

A **do** expression always results in a **monadic value**

this **monadic value** is a **function (Int -> Int)**.

return has no other effect

but to convert the **(a+b)** into a **monadic value**

- takes a number
- **(*2)** gets applied to that number
- the result becomes a.
- **(+10)** is applied to the same number
- the result becomes b

addstuff 3

3

(*2) 3 = 6

(+10) 3 = 13

6 + 13 = 19

http://learnyouahaskell.com/for-a-few-monads-more#reader

Function Monad

both **(*2)** and **(+10)** are applied to the **number 3**
return (a+b) is applied to the **number 3** as well,
return (a+b) converts **(a+b)** into a **monadic value**
(applying the **number 3**)

the **function monad** is also called the **reader monad**
all the **functions** read from a **common source**. **3**

rewrite **addStuff**

```
addStuff' :: Int -> Int
addStuff' x = let
    a = (*2) x
    b = (+10) x
    in a+b
```

```
addStuff :: Int -> Int
addStuff = do
    a <- (*2)
    b <- (+10)
    return (a+b)
```

monadic values

a monadic value

function monad

= reader monad

http://learnyouahaskell.com/for-a-few-monads-more#reader

Reader Monad

the **reader monad** allows us to treat **functions**
as **values** with a **context**. (**monadic values**)

We can act as if *we already know what the functions will return*.
(a kind of functions **r->a**)

by gluing **functions** together into one **function**
and then giving that function's **parameter**
to all of the **functions** that it was glued from.

a lot of **functions** that are all just missing one **parameter**
and they'd eventually be applied to the same thing,

the **reader monad** to extract their future **results**
and the **>>=** implementation will make sure that it all works out.

(h >>= f) r = f (h r) r

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

http://learnyouahaskell.com/for-a-few-monads-more#reader

(->) r Monad

```
instance Monad ((->) r) where
    return = const
    x >>= f = \r -> f (x r) r

    (x >>= f) r = f (x r) r
```

```
x :: m a                    m ... r ->
f :: a -> m b               m ... r ->

x ::      r -> a
f :: a -> r -> b
```

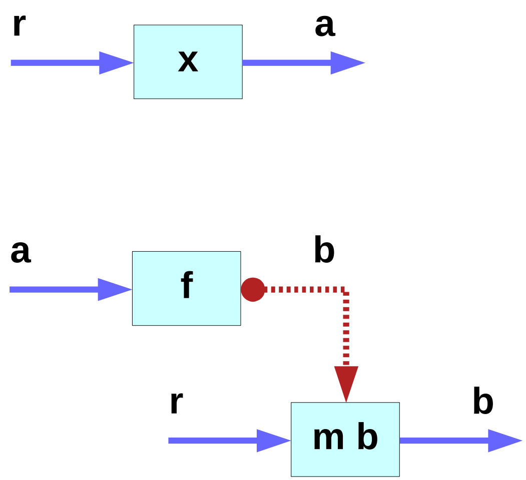
```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

```
    lift . return = return

    lift (m >>= f) = lift m >>= (lift . f)
```

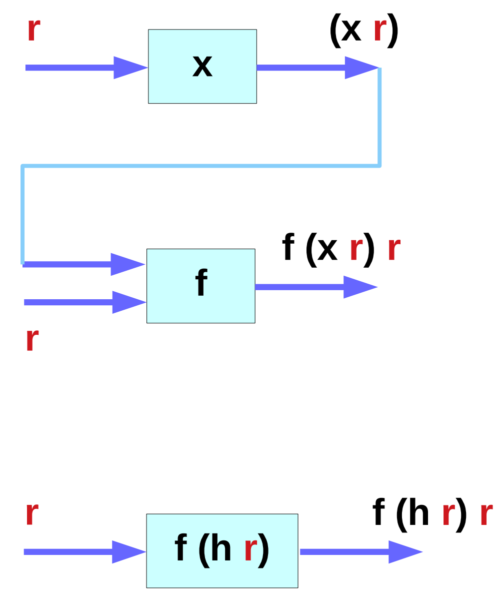
https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

(->) r Monad

```
x :: m a            x :: r -> a
f :: a -> m b       f :: a -> r -> b
```



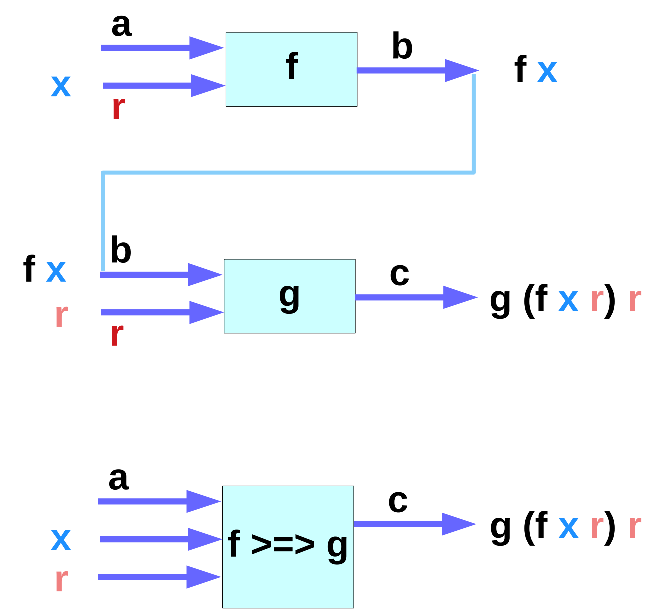

```
instance Monad ((->) r) where
  return = const
  x >>= f = \r -> f (x r) r
```



https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

(->) r Monad – >=> composition operator

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


(f >=> g)    = \x -> f x >>= g
             = \x -> (\r -> g (f x r) r)


(f >=> g) x r = g (f x r) r

(x >>= f) r = f (x r) r

return x r = x
```



https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

(->) r Monad – fmap

```
fmap f x = x >>= return . f

fmap f x r = (x >>= (const . f)) r            (m >>= n) r = n (m r) r
           = (const . f) (x r) r              (s . f) a = s (f a)
           = const (f (x r)) r                const c d = c
           = f (x r)                          s (f a) = (s . f) a
           = (f . x) r

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

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

Join Function

join flattens any nested **monadic value**
: a property unique to monads.

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

when the **result** of one **monadic value** is **another monadic value**
i.e. if one **monadic value** is nested inside the other,
it is possible to flatten them to just a single normal monadic value

Just (Just 9) ➡ **Just 9**

join [[1,2,3],[4,5,6]] ➡ **[1,2,3,4,5,6]**

flattening lists
for **lists**, **join** is just **concat**.

http://learnyouahaskell.com/for-a-few-monads-more#reader

Join Function

a successful computation as a result of a successful computation,
so they're both just joined into one big successful computation.

```
ghci> join (Just (Just 9))
Just 9
```

a **Nothing** as a result of a **Just value**.
Whenever dealing with **Maybe values** and
combining several of them into one with **<*>** or **>>=**
All these values have to be **Just values**
for the **result** to be a **Just value**.
any **failure** makes the **result** a **failure**

```
ghci> join (Just Nothing)
Nothing
```

flatten what is from the onset a **failure**,
so the **result** is a **failure** as well.

```
ghci> join Nothing
Nothing
```

http://learnyouahaskell.com/for-a-few-monads-more#reader

Join Function

To flatten a **Writer** value whose **result** is a **Writer value** itself,
we have to **mappend** the **monoid value**.

ghci> runWriter $ join (Writer (Writer (1,"aaa"),"bbb"))
(1,"bbbaaa")

The **outer** monoid value "bbb" comes first
and then to it "aaa" is appended.

when you want to examine what the **result** of a **Writer value** is,
you have to write its **monoid value** to the log first
and only then can you examine what it has inside

http://learnyouahaskell.com/for-a-few-monads-more#reader

Join Function

Flattening Either values is very similar to flattening Maybe values:

ghci> join (Right (Right 9)) :: Either String Int
Right 9
ghci> join (Right (Left "error")) :: Either String Int
Left "error"
ghci> join (Left "error") :: Either String Int
Left "error"

http://learnyouahaskell.com/for-a-few-monads-more#reader

Join Function

join to a **stateful computation**
whose **result** is a **stateful computation**,
the **result** is a **stateful computation**
that first runs the outer **stateful computation**
and then the resulting one.

ghci> runState (join (State $ \s -> (push 10,1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])

The lambda here takes a **state** and puts 2 and 1 onto the **stack**
and presents push 10 as its **result**.
So when this whole thing is flattened with join and then run,
it first puts 2 and 1 onto the stack
and then push 10 gets carried out,
pushing a 10 on to the top.

http://learnyouahaskell.com/for-a-few-monads-more#reader

Join Function Implementation

```
join :: (Monad m) => m (m a) -> m a
join mm = do
    m <- mm
    m
```

Because the **result** of **mm** is a **monadic value**,
we get that **result** and then just put it on a line of its own
because it's a **monadic value**.

http://learnyouahaskell.com/for-a-few-monads-more#reader

Join Function Implementation

The trick here is that when we do **m <- mm**,
the **context** of the **monad** in which we are in gets taken care of.

That's why, for instance, **Maybe** values result in **Just values**
only if the **outer** and **inner values** are both **Just values**.

Here's what this would look like
if the **mm value** was set in advance to **Just (Just 8)**:

```
joinedMaybes :: Maybe Int
joinedMaybes = do
    m <- Just (Just 8)
    m
```

http://learnyouahaskell.com/for-a-few-monads-more#reader

((->) r) Monad

```
join x r = x r r


join x r = (x >>= id) r
         = (id (x r) r)
         = (x r) r
         = x r r



(x >>= f) r = join (fmap f x) r
            = join (f.x) r
            = (f.x) r r
            = f (x r) r
```

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

((->) r) Monad

```
incN :: Enum a => a -> Int -> a
incN c n = toEnum $ n + fromEnum c

decN c n = incN c (-n)

inc2N c n = incN c (2 * n)

*Main> incN ’a’ 3
’d’
*Main> inc2N ’a’ 3
’g’
*Main> decN ’m’ 3
’j’
```

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

((->) r) Monad

We can compose these functions easily with the Kleisli arrow:

```
*Main> (incN >=> inc2N >=> decN) 'a' 0
'a'
*Main> (incN >=> inc2N >=> decN) 'a' 1
'c'
*Main> (incN >=> inc2N >=> decN) 'a' 2
'e'

munge c = do
        x <- incN  c
        y <- inc2N x
        z <- decN  y
        return z
```

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

((->) r) Monad

join (+) 2

join (+) 2 = (+) 2 2 = 2 + 2 = 4

join (-) x = (-) x x = 0

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

((->) r) Monad

```
join (-) ≈ const 0


Prelude Control.Monad> :t join (-)
join (-) :: Num a => a -> a


Prelude Control.Monad> :t const 0
const 0 :: Num a => b -> a


join div 42 = 1
```

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

((->) r) Monad

```
join (-) ≈ const 0


Prelude Control.Monad> :t join (-)
join (-) :: Num a => a -> a

but

Prelude Control.Monad> :t const 0
const 0 :: Num a => b -> a


join div 42 = 1
```

https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

References

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