Monad Overview (2A)

Young Won Lim
12/4/18

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

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

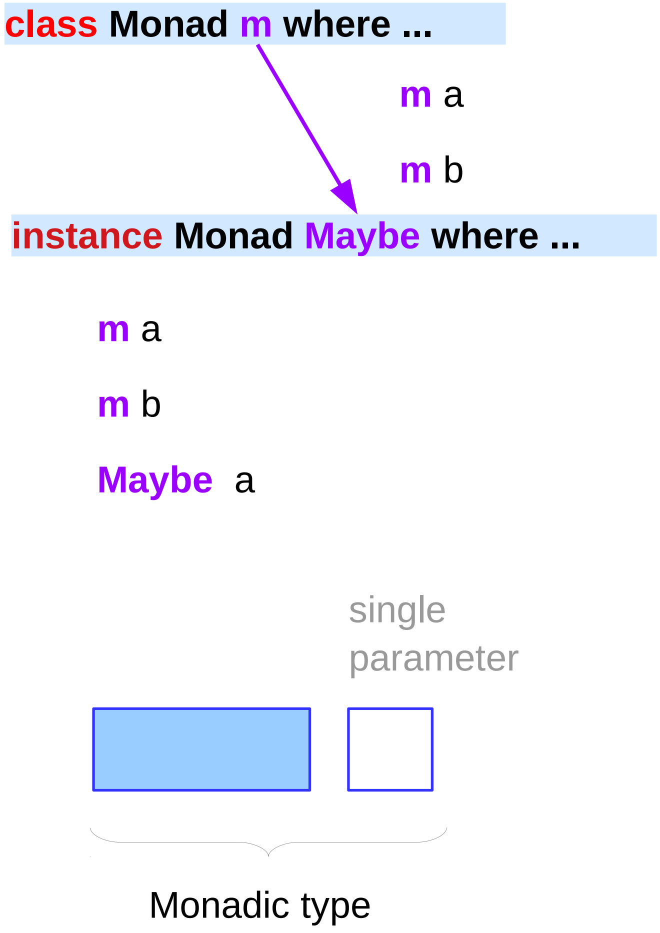
Monad, Monoid

monad (plural monads)

- An ultimate atom, or simple, unextended point; something ultimate and indivisible.
- (mathematics, computing) A monoid in the category of endofunctors.
- (botany) A single individual (such as a pollen grain) that is free from others, not united in a group.

monoid (plural monoids)

- (mathematics) A **set** which is closed under an associative binary operation, and which contains an element which is an identity for the operation.



class Monad m where ...

m a

m b

instance Monad Maybe where ...

m a

m b

Maybe a

https://en.wiktionary.org/wiki/monad, monoid

Maybe Monad Instance

```
class Monad m where

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

method type signatures

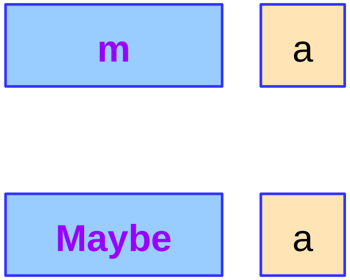
```
instance Monad Maybe where
   -- return      :: a -> Maybe a
   return x                 =  Just x

   -- (>>=)       :: Maybe a -> (a -> Maybe b) -> Maybe b
   Nothing  >>= _           =  Nothing
   (Just x)  >>= f          =  f x


   f :: a -> Maybe b
```

method definition

method definition



a parameterized type

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad – a parameterized type

a monad is a parameterized type **m**

that supports **return** and **>>=** functions of the specified types

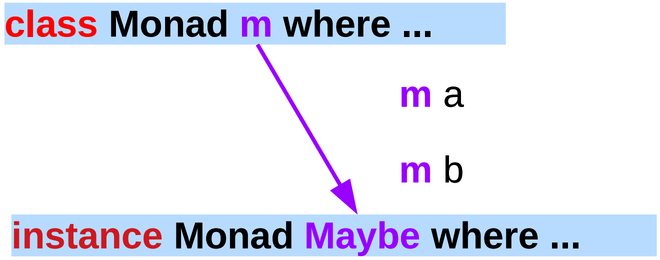
m must be a parameterized type,

rather than just a type (**Maybe** is not a concrete type)

(Just x) >>= f = f x

the **do** notation can be used to sequence **Maybe** values.

More generally, Haskell supports the use of this notation **(>>=)** with any monadic type.



class **Monad** **m** **where ...**

m a

m b

instance **Monad** **Maybe** **where ...**

(Just x) >>= f = f x

Assume

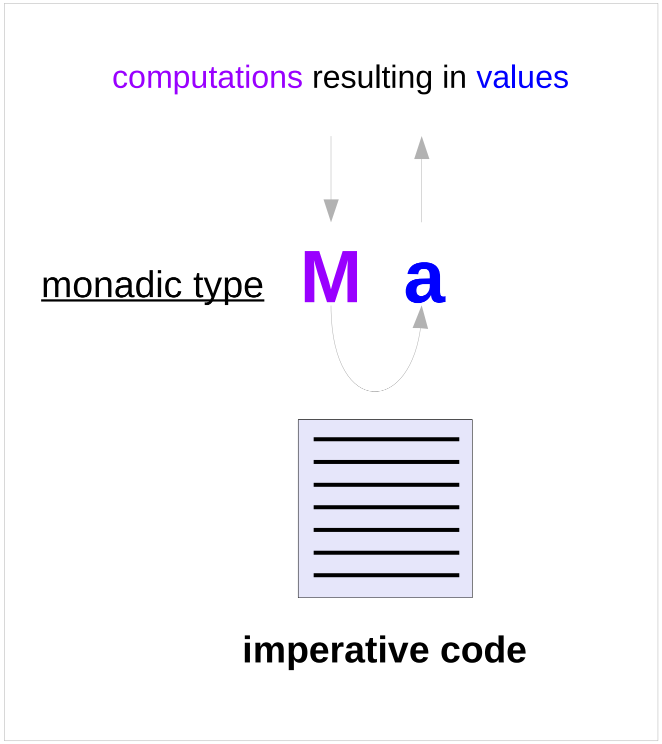
(Just 3) :: Maybe Int

f :: Int -> Maybe Int

>>= passes 3::Int to f as an argument

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

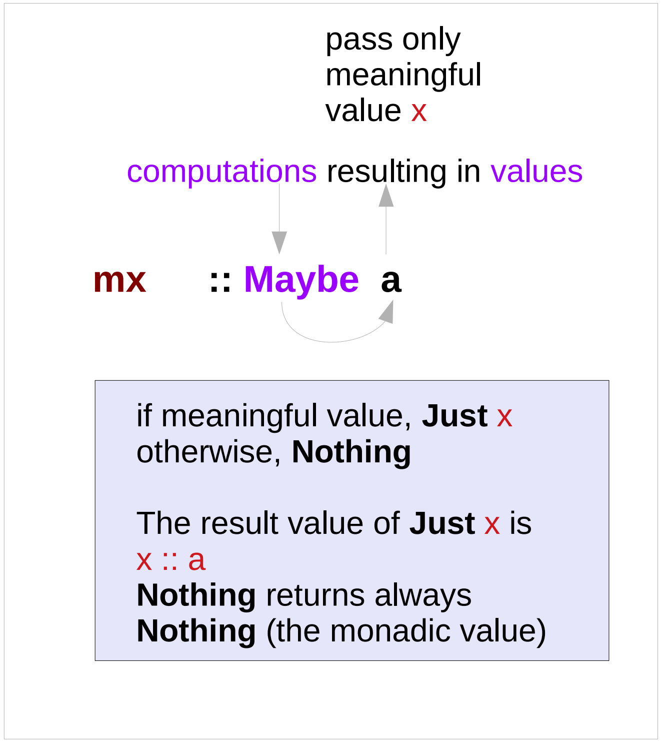
Maybe Monad – an action and its result



computations resulting in values

monadic type **M a**

imperative code



pass only
meaningful
value x

computations resulting in values

mx :: Maybe a

if meaningful value, **Just** x
otherwise, **Nothing**

The result value of **Just** x is
x :: a
Nothing returns always
Nothing (the monadic value)

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

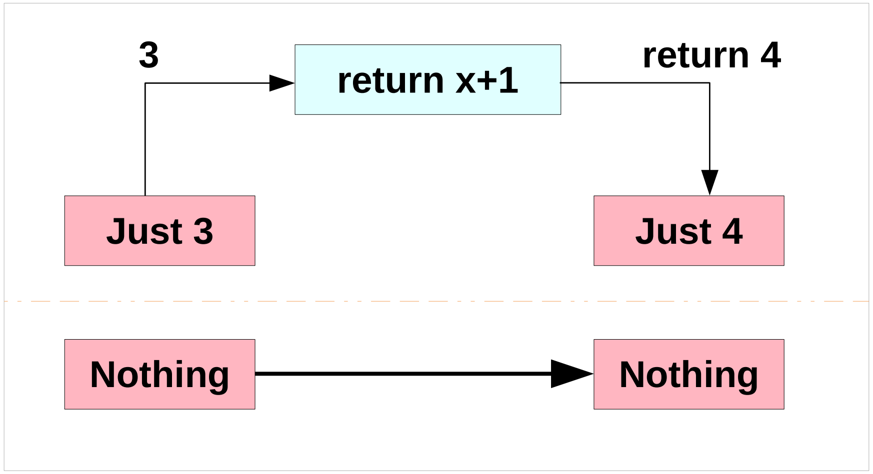
Maybe Monad – the bind operator (**>>=**)

pass only the meaningful value

Just 10 >>= g ⇒ g 10

Nothing >>= g ⇒ Nothing

computation stops immediately



3

return x+1

return 4

Just 3

Just 4

Nothing

Nothing

g x = reutrn x+1

g = \x -> reutrn x+1

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad – the assignment operator (<-)

```
dt1 = do { x <- Just 3;
                   if x == 3 then return 33;
                                  else return 44;}
```

Just 33

After evaluating the monadic value,
only the result is assigned to x

```
dt2 = do { x <- Just 4;
                   if x == 3 then return 33;
                                  else return 44;}
```

Just 44;

If a meaningful number
is assigned to x

```
dt2 = do { x <- Nothing;
                   if x == 3 then return 33;
                                  else return 44;}
```

Nothing;

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad – the value for failure

The **Maybe** monad provides

a simple model of computations that can fail,

a value of type **Maybe a** is either **Nothing** (failure) or

the form **Just x** for some **x** of type **a** (success)

The **list** monad generalizes this notion,

by permitting multiple results in the case of success.

a value of **[a]** is

either the empty list **[ ]** (failure)

or the form of a non-empty list **[x1,x2,...,xn]** (success)

for some **xi** of type **a**

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Maybe Monad Examples

```
bothGrandfathers :: Person -> Maybe (Person, Person)

bothGrandfathers p =
   father p >>=
       (\dad -> father dad >>=
           (\gf1 -> mother p >>=
               (\mom -> father mom >>=
                   (\gf2 -> return (gf1,gf2) ))))
```

```
bothGrandfathers p = do {
        dad    <- father p;
        gf1    <- father dad;
        mom  <- mother p;
        gf2    <- father mom;
        return (gf1, gf2);
      }
```

p :: Person

father **p** :: Maybe Person

mother **q** :: Maybe Person

dad :: Person

gf1 :: Person

mom :: Person

gf2 :: Person

(gf1, gf2) :: Maybe (Person, Person)

gf1 is only used in the final return

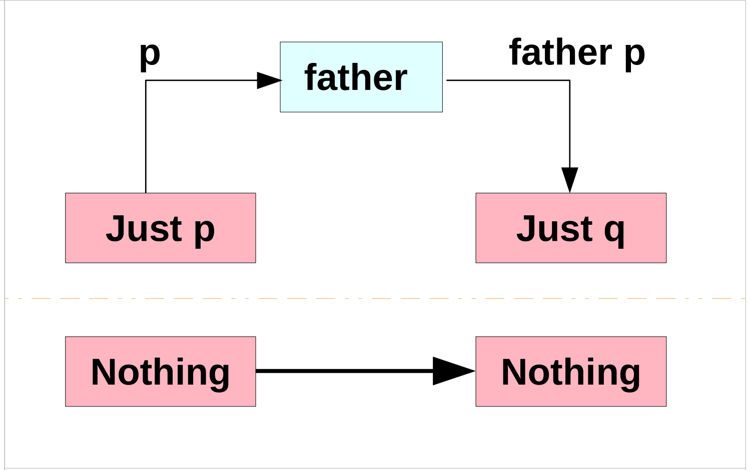
https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Fail to return result exception

Sequencing operator **>>=** and **do bock** look like
an **imperative** programming code
but they support **exceptions** :

father and **mother** are **functions**
that might **fail** to produce results,
raising an **exception** instead;

when any exception happens, the whole code will fail, i.e.
terminate with an exception (evaluate to **Nothing**).



p :: Person
father **p** :: Maybe Person

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x  =  [x]

   -- (>>=)  :: [a] -> (a -> [b]) -> [b]
   xs >>= f  =  concat (map f xs)
```

return converts a **value** into a ***successful* result** containing that value

>>= provides a means of *sequencing* computations
that may produce *multiple results*:

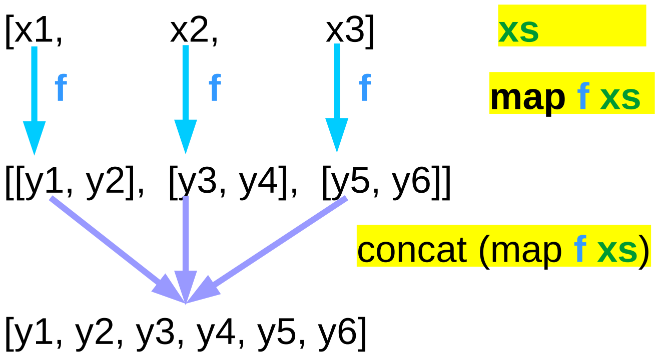
xs >>= **f** applies the function **f** to each of the *results* in the list **xs**
to give a *nested list* of *results*,
which is then concatenated to give a *single list* of *results*.

(Aside: in this context, [] denotes the list type [a] without its parameter.)

xs :: [a]

f :: a -> [b]

(>>=) :: [a] -> (a -> [b]) -> [b]



[x1, x2, x3]

f f f

[[y1, y2], [y3, y4], [y5, y6]]

[y1, y2, y3, y4, y5, y6]

xs

map f xs

concat (map **f xs**)

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

A Type **Monad**

Haskell does not have **states**

but it's type system is powerful enough

to construct the stateful program flow

defining a **Monad type** in Haskell

- similar to defining a **class**
  in an object oriented language (C++, Java)
- a **Monad** can do much more than a class:

A **Monad type** can be used for

- **exception handling**
- **parallel program workflow**
- **a parser generator**

Collection of method to be implemented

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Types: rules and data

Haskell **types** are the ***rules*** associated with the **data**,

not the actual **data** itself.

OOP (Object-Oriented Programming) enable us

to use **classes** / **interfaces**

to define **types**,

the **rules** (**methods**) that interacts with the actual **data**.

to use **templates**(c++) or **generics**(java)

to define more **abstracted rules** that are more reusable

Monad is pretty much like **templates / generic class**.

Rules + Data

Rules

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Monad Rules

A **type** is just **a set of rules**, or **methods**
in Object-Oriented terms

A **Monad** is just yet another type, and
the definition of this type is defined by **four rules**:

1) **bind (>>=)**
2) **then (>>)**
3) **return**
4) **fail**

Rules (methods)

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Monad Applications

1. **Exception Handling**

2. **Accumulate States**

3. **IO Monad**

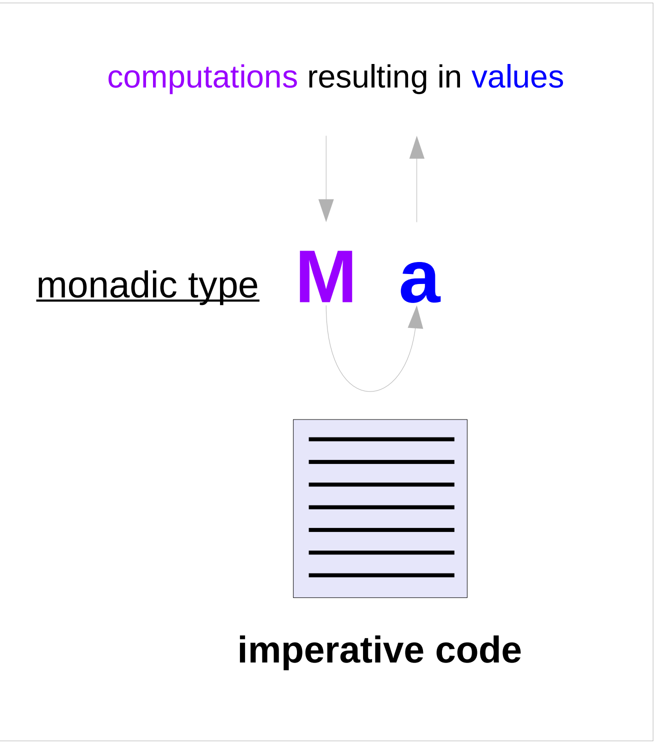
http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

A notion of computations

the expression **father p**, which has type **Maybe** **Person**,
is interpreted as a **statement** in an imperative language
that ***returns*** a **Person** as the **result**, **Just p**
or ***fails***. **Nothing**

a value of type M **a** is interpreted as a **statement**
in an imperative language M
that returns a value of type **a** as its **result**;

executing a **statement** returns the **result**
running a function



computations resulting in values

monadic type **M a**

imperative code

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Semantics of a language M

Semantics : what the language **M** *allows us to say*.

In the case of **Maybe**,

the **semantics** *allow us to express failures*

when a statement fails to produce a result,

allow statements that are following to be ***skipped***

the **semantics** of this language are determined by the **monad M**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

A value of type **M a**

a **value** of type **M a** is interpreted

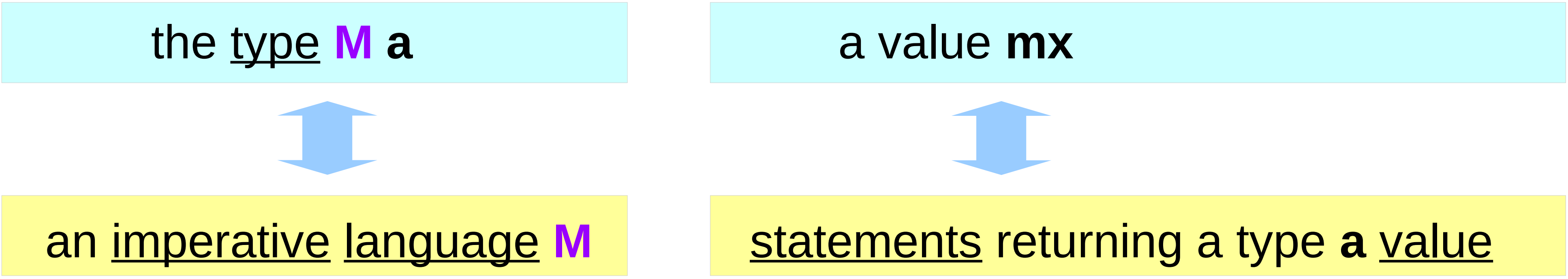
mx :: **M a**

as a **statement** in an imperative language **M**
that returns a **value** of type **a** as its **result**;

the **semantics** in an imperative language **M**
allow us to express **failure**
the **statements** that follow it being **skipped**

Maybe a
IO a
ST a
State s a

the type of result **a** : **IO** a


the type **M a**

an imperative language **M**

semantics

a value **mx**

statements returning a type **a** value

execution, a function, a return value

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

A value of type **M a** – computations and semantics

a **value** of type **M a** is interpreted

as a **statement** in an imperative language **M**
that returns a value of type **a** as its result;

and the **semantics** of this language
are determined by the **monad M**.

computations that result in values

an immediate abort
a valueless return in the middle of a computation.

types are the ***rules*** associated with the **data**,
not the actual **data** itself. *(classes in C++)*

Maybe a
IO a
ST a
State s a

an imperative language
statements
computations
rules
execution
a function
return

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Monad Minimal Definition

A minimal definition of **monad**

a **type constructor** **m**;
a function **return**;
an operator (>>=) “**bind**"

The function and operator are methods of the **Monad** type class
and have types (type signatures)

```
return :: a -> m a

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

are required to obey three laws

m a

- **Maybe a**
- **IO a**
- **ST a**
- **State s a**

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

Monad Laws

every **instance** of the Monad type class must obey

```
m >>= return     =  m                          -- right unit
return x >>= f   =  f x                        -- left unit
(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)    -- associativity
```

```
(m  >>=  return)    =  m
(return  x)  >>=  f   =  f x
((m >>= f) >>= g)   =  m >>= (\x -> f x >>= g)
```

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

m :: m a

f :: a -> m b
f x :: m b
f x >>= g :: m c

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

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

Monad Laws Examples

```
m >>= return     =  m                          -- right unit
return x >>= f   =  f x                        -- left unit
(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)    -- associativity
```

```
(m  >>=  return)     =  m
(Just 3 >>=  return)     =  Just 3

(return  x)  >>=  f   =  f x
(return  3)  >>=  (\x -> return (x+1))   =  return 4 = Just 4

((m >>= f) >>= g)   =  m >>= (\x -> f x >>= g)
((Just 3)  >>=  (\x -> return (x+1)) )  =  return 4 = Just 4
((Just 4)  >>=  (\x -> return (2*x)) )  =  return 8 = Just 8
```

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

Then (>>) and bind (>>=) operators

the **then** operator **(>>)**

an implementation of the **semicolon**

;

The **bind** operator **(>>=)**

an implementation of the **semicolon** (**;**) and
assignment (**<-**) of the result
of a previous computational step.

<-   ;

;   ->

```
x <- foo;    return (x + 3)

foo   >>=   (\x -> return (x + 3))
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

A function application and the bind operator

a **let** expression as a **function** application,

```
let x = foo in (x + 3)          foo  &  (\x -> id (x + 3))     -- v & f = f v
```

& and **id** are trivial;

id is the identity function

just returns its parameter unmodified

an **assignment** and **semicolon** as the **bind** operator:

```
x <- foo; return (x + 3)        foo >>= (\x -> return (x + 3))
```

>>= and **return** are substantial.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Reverse Function Application &

(&) :: a -> (a -> b) -> b

& is just like **$** only backwards.

foo $ bar $ baz bin

semantically equivalent to:

bin & baz & bar & foo

& is useful because the order in which functions are applied to their arguments read left to right instead of the **reverse** (which is the case for $).

This is closer to how English is read so it can improve code clarity.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

& and id

a **let** expression as a **function** application,

let x = foo in (x + 3)   **foo & (\x -> id (x + 3))**   **-- v & f = f $ v = fv**

The **&** operator combines together two *pure calculations*,

foo and **id (x + 3)**

while creating a new binding for the variable **x** to hold **foo**'s value,   x ← foo

making **x** available to the second computational step: **id (x + 3)**.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

>>= and return

an assignment and semicolon as the **bind** operator:

```
x <- foo; return (x + 3)          foo >>= (\x -> return (x + 3))
```

The bind operator >>= combines together two computational steps,

foo and return **(x + 3)**,

in a manner particular to the **Monad** M,

while creating a new binding for the variable **x** to hold foo's **result**,

making **x** available to the next computational step, return **(x + 3)**.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

>>= and return – Semantics of Maybe Monad

an **assignment** and **semicolon** as the **bind** operator:

x <- foo; return (x + 3)

foo >>= (\x -> return (x + 3))

In the particular case of **Maybe**,	***semantics***
if **foo** fails to produce a result,	**Nothing**
the second step will be skipped and	
the whole combined computation will also fail immediately.	**Nothing**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Contexts of >> & >>=

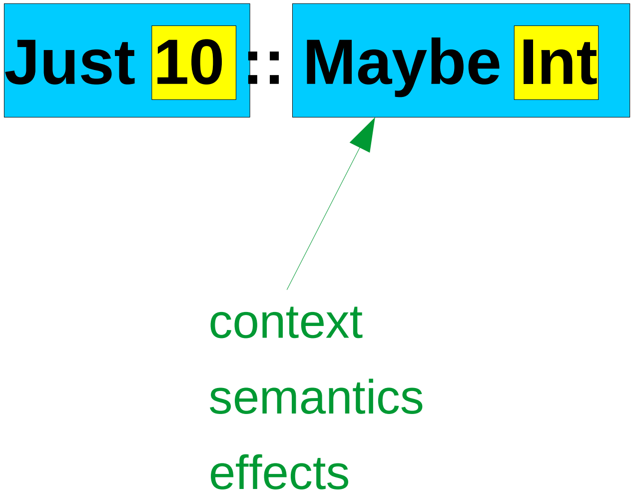
>>= and >> : methods from the Monad *class*

Monad Sequencing Operator

>> is used to **order** the **evaluation** of expressions
within some **context**;
it makes evaluation of the *right*
depend on the evaluation of the *left*

Monad Sequencing Operator with value passing

>>= **passes** the result of the expression on the *left*
as an argument to the expression on the *right*,
while preserving the **context** that the argument and function use



Just 10 :: Maybe Int

context
semantics
effects

Just 10 >>= f

➡ **f 10**

10 is passed to the function
f as an argument

https://www.quora.com/What-do-the-symbols-and-mean-in-haskell

Monadic Effect

```
class Monad m where
   return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/IO

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

https://www.cs.hmc.edu/~adavidso/monads.pdf

Monadic Operations

Monadic operations tend to have types which look like

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

Inputs to monadic operations

a parameterized type

a monad type

effect-monad

statements in the imperative language

about a function

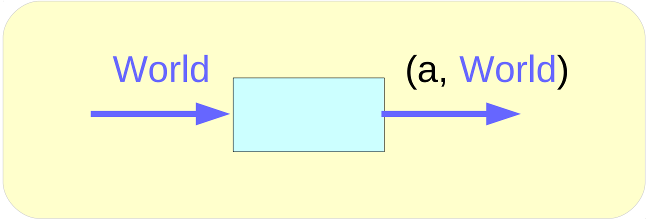
executable

execution result

val-out-type

effect-monad val-out-type

IO a



https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

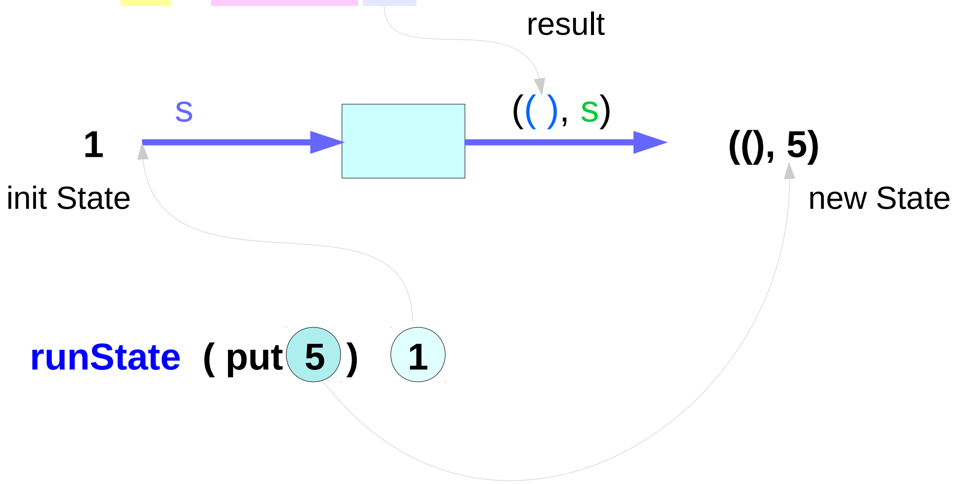
Monadic Operations – put example

Monadic operations tend to have types which look like

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

effect-monad	val-out-type
(**State s**)	**( )**

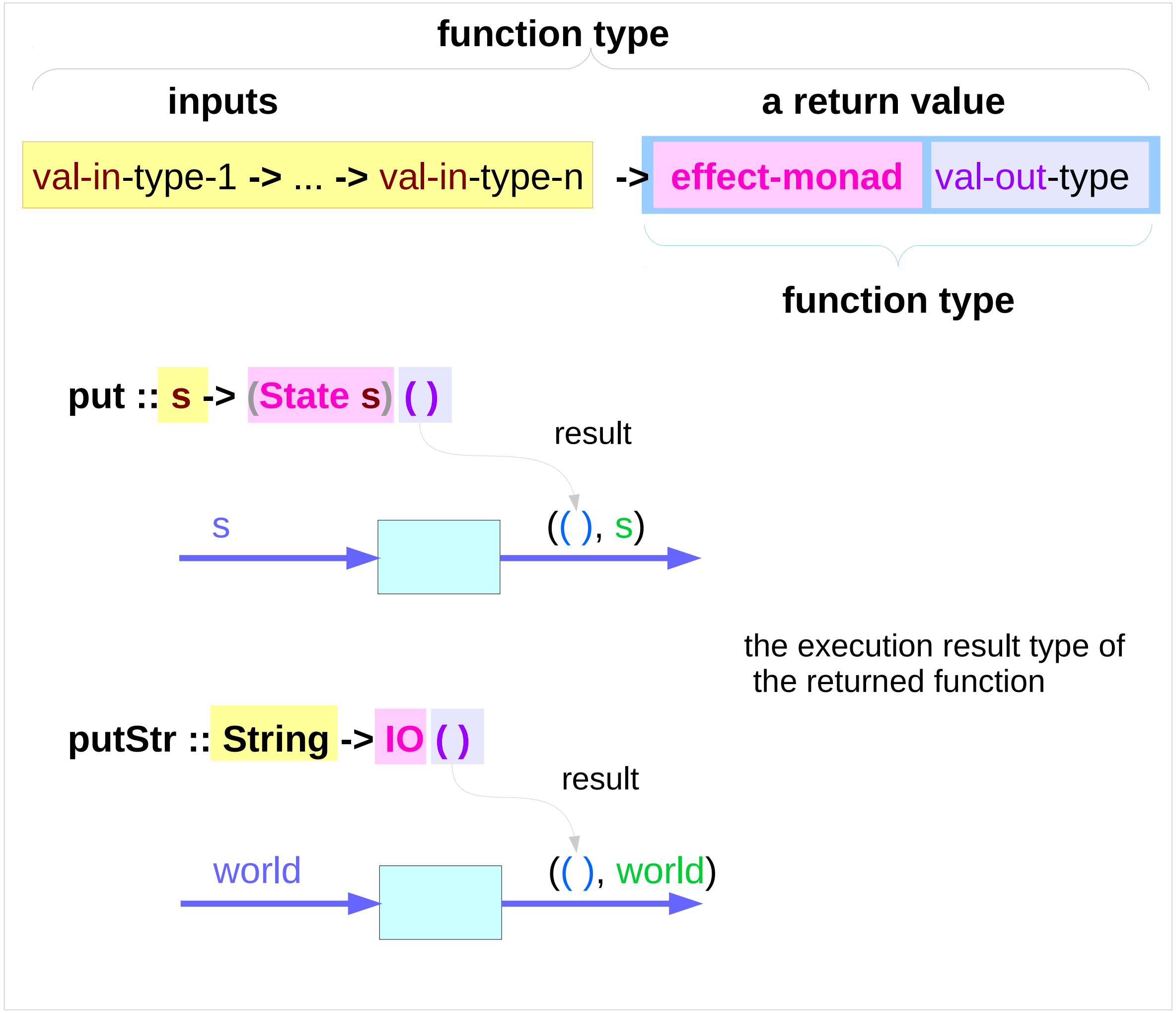
put :: s -> (State s) ( )



runState (put 5) 1

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operation Examples



function type

inputs **a return value**

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

function type

put :: s -> (State s) ( )

the execution result type of the returned function

putStr :: String -> IO ( )

A monadic operation

= A function

- **inputs**
- **a return value**

  which can be another **function**

  returning a **function** as a **value**

  executing this function

  produces a <u>result</u>

  → val-out-type

computations

statement

in the imperative language

effect-monad

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operation Examples

effect-monad val-out-type
parameter

`tick :: State Int Int`

where the **Int type** is a type application:

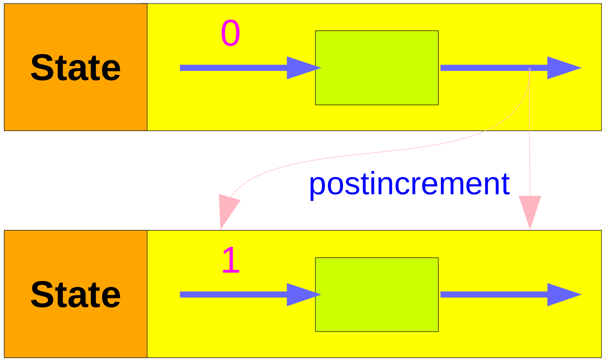
effect-monad — **State Int**
an executable function giving information
about which **effects** are possible **statement**
in the imperative language

Val-out-type — **Int**
the parameter of the executable function
the **Int** type **result** produced by the function
(the result of executing the function)

```
tick :: State Int Int
tick = do   n <- get
            put (n+1)
            return n


test  = do  tick                 -- (0,1)
            tick                 -- (1,2)


runState test 0                  -- (1,2)
```



https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations

val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

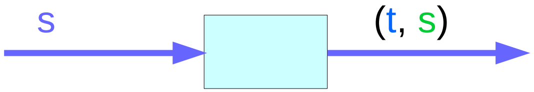
parameter

where the **return type** is a type application:
a **type** with a **parameter** type (a parameterized type)

effect-monad
an executable function
giving information about which **effects** are possible
statement in the imperative language

val-out-type
the parameter of the executable function
the type of the **result** produced by the function
(the result of executing the function)

returning a function as a value



put :: s -> (State s) ( )

putStr :: String -> IO ( )

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

Monadic Operations – **put**, **putStr**

put :: s -> State s ( )

put :: s -> (State s) ( )

one value input type	**s**
the **effect-monad**	**State s**
the value output type	**( )**

the operation is used *only for its effect*;

the *value* delivered is *uninteresting*

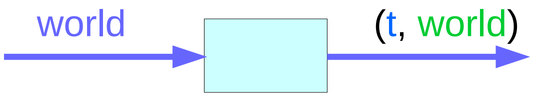
putStr :: String -> IO ( )

delivers a string to **stdout** but does not return anything meaningful

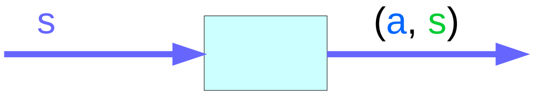
https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

IO t and State s a types

type **IO** t = World -> (t, World) type synonym



newtype **State** s a = **State {** **runState** :: s -> (a, s) **}**



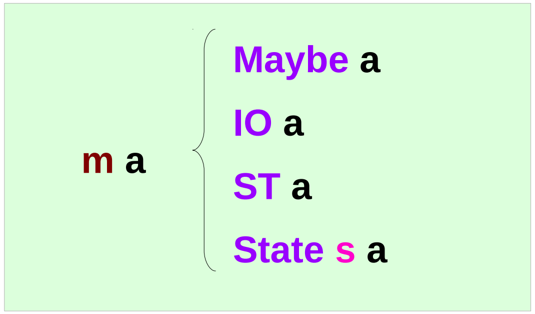
s : the type of the state,

a : the type of the produced result

s -> (a, s) : function type

Monad Definition

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
```


m a

Maybe a
IO a
ST a
State s a

1) return
2) bind (>>=)
3) then (>>)
4) fail

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

Maybe **Monad** Instance

```
instance Monad Maybe where
        return x = Just x
        Nothing >>= f = Nothing
        Just x >>= f  = f x
        fail _ = Nothing
```

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

State Monad Instance

```
instance Monad (State s) where


return :: a -> State s a
return x = state ( \s -> (x, s) )


(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p          -- p' :: s -> (a, s)
    k' = runState . k        -- k' :: a -> s -> (b, s)
    q' s0 = (y, s2) where    -- q' :: s -> (b, s)
        (x, s1) = p' s0      -- (x, s1) :: (a, s)
        (y, s2) = k' x s1    -- (y, s2) :: (b, s)
    q = State q'
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

IO Monad Instance

```
instance  Monad IO  where
    m >> k    = m >>= \ _ -> k
    return     = returnIO
    (>>=)      = bindIO
    fail s      = failIO s


returnIO :: a -> IO a
returnIO x = IO $ \s -> (# s, x #)


bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k
   = IO $ \s ->  case m s of (# new_s, a #)
                 -> unIO (k a) new_s
                m = new_s,
                s  =  a
                (k a) new_s
                (k s) m
```

```
case expression of
        pattern -> result
        pattern -> result
        pattern -> result
          ...
```

https://stackoverflow.com/questions/9244538/what-are-the-definitions-for-and-return-for-the-io-monad

IO Monad – return method

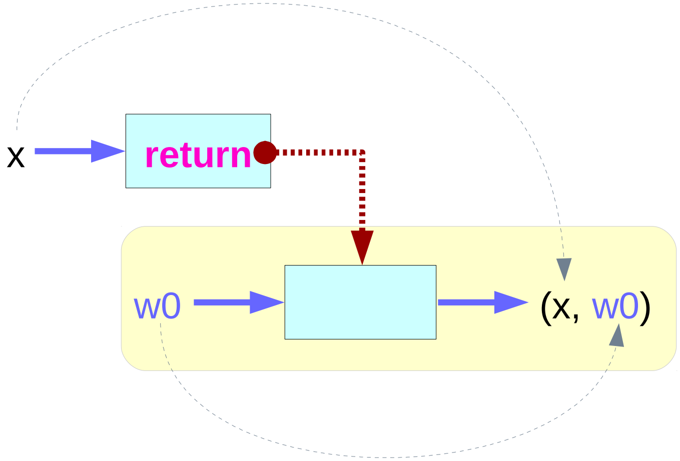
The **return** function takes x

and gives back a function

that takes a w0 :: World

and returns x along with the updated World,

but not modifying the given w0 :: World

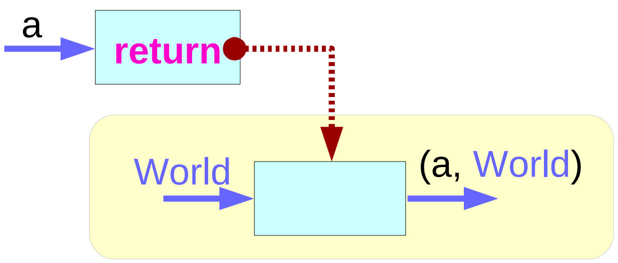


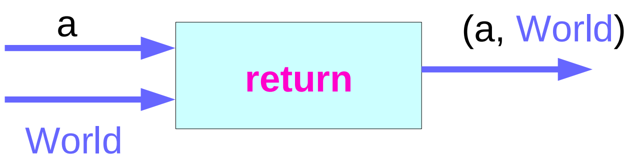
.

https://www.cs.hmc.edu/~adavidso/monads.pdf

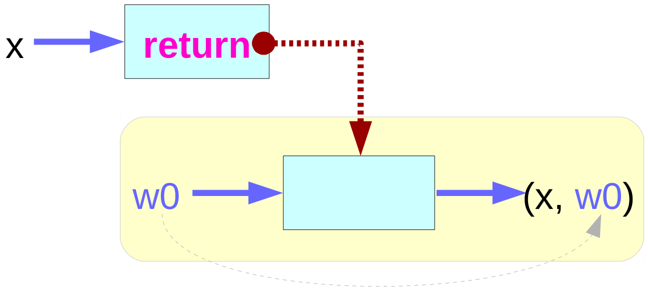
IO Monad – return method and partial application

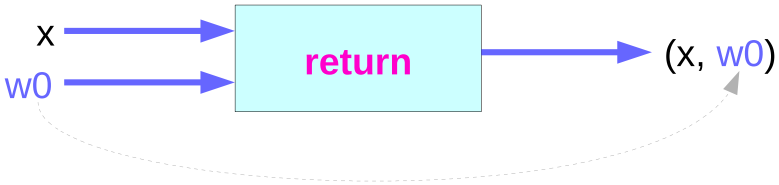
return a :: a -> IO a ⇦ **TYPEs** ⇨ **return a World :: (a, World)**





let (x, w0) = return x w0 ⇦ **Values** ⇨ **let (x, w0) = return x w0**





https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad – ioX >>= f (1. state update, 2. result)

the expression (**ioX** >>= **f**) has

type IO **a -> (a ->** IO **b) ->** IO **b**

the implementation of bind

first, execute the **action**

<u>execute</u> **ioX**

State updated

w0 → **w1**

result extracted

x is the **result**

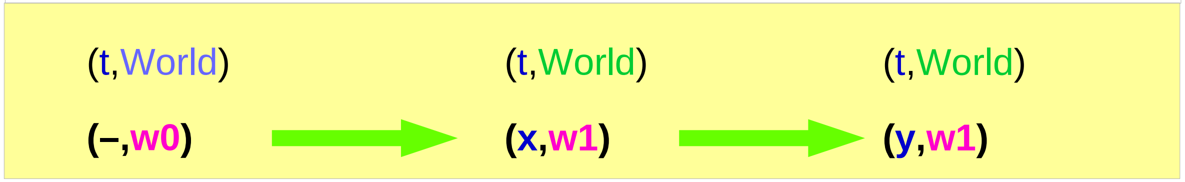
then, compute the new **result**

using **f x**

no **State transition**

w1 is remained

result computed **y = (f x)**

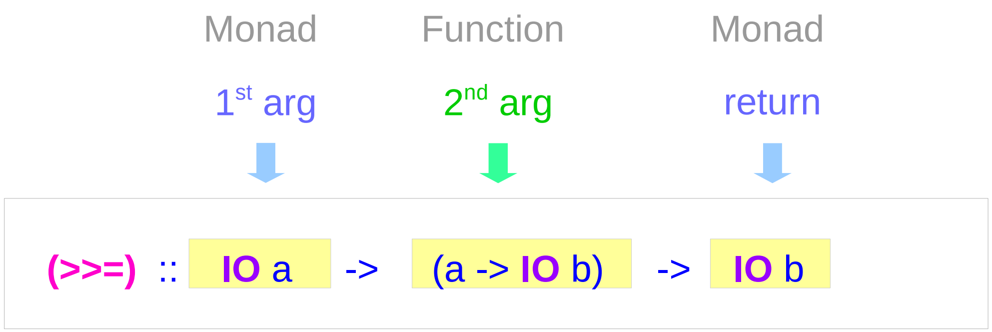


(t,World) (t,World) (t,World)

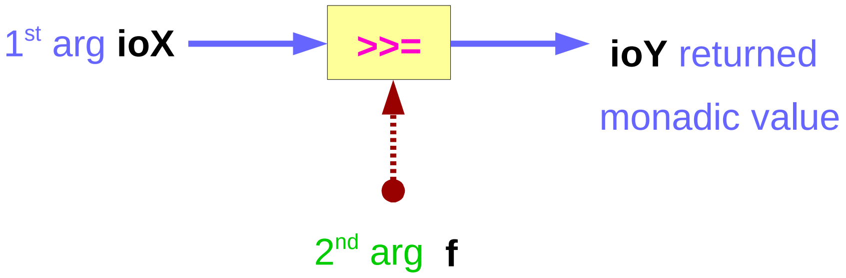
(–,w0) **(x,w1)** **(y,w1)**

https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad - (>>=) operator type signature



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



ioX :: IO a

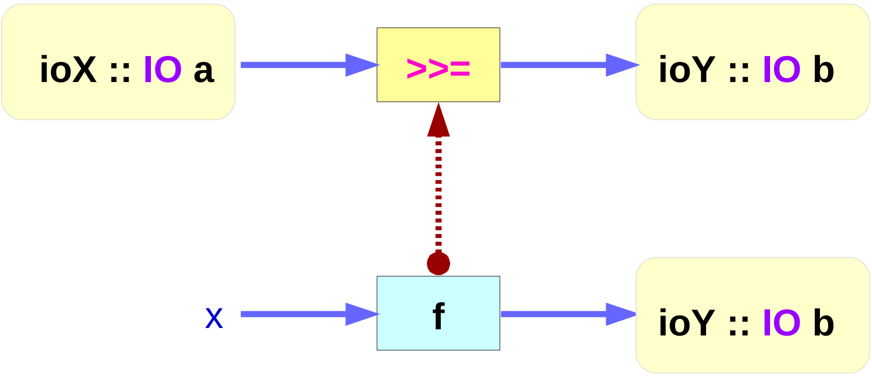
ioY :: IO b

IO Monad - (>>=) operator type diagram

Monad — Function — Monad

1st arg — 2nd arg — return

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



ioX :: IO a

>>=

ioY :: IO b

x

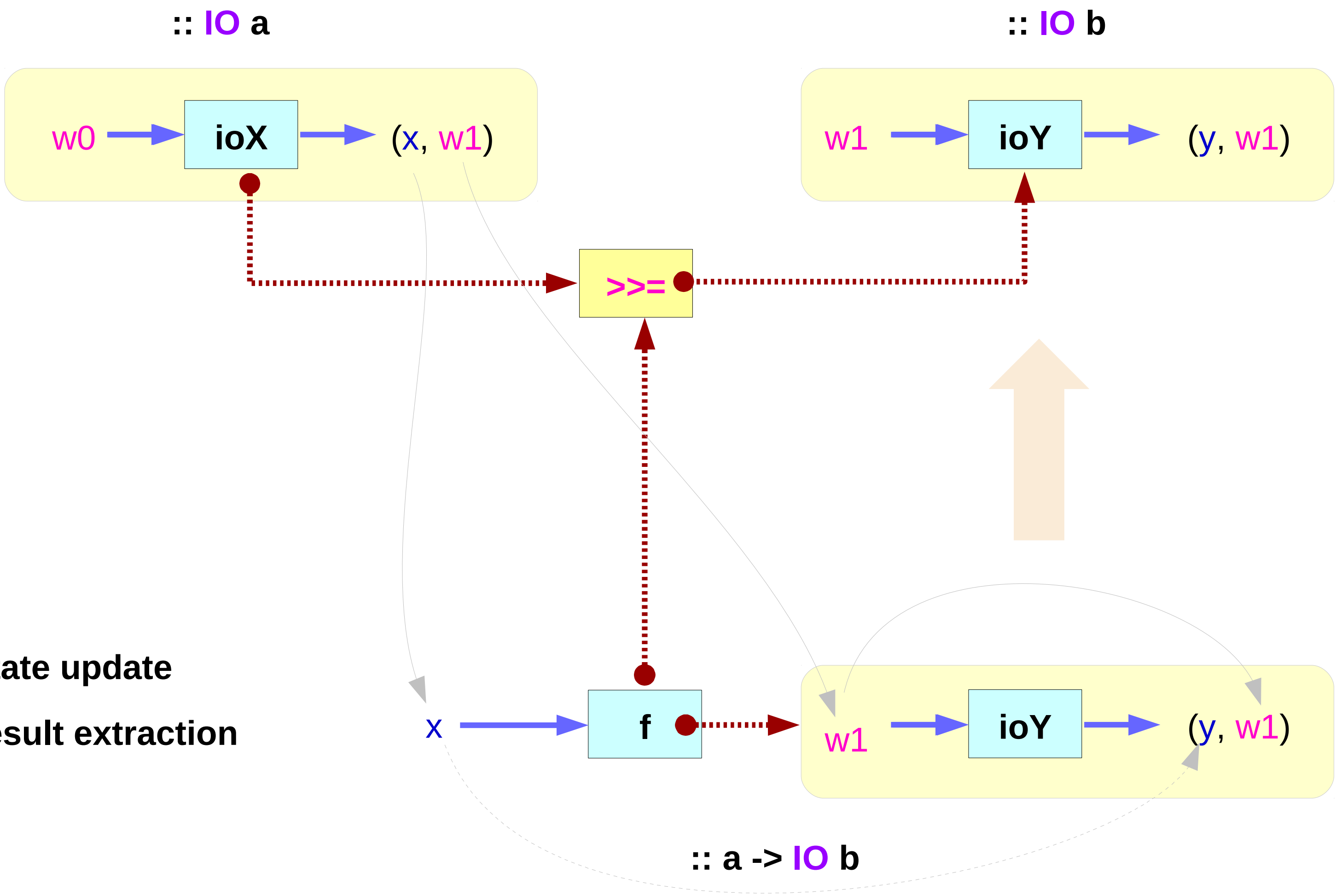
f

ioY :: IO b

IoX – state update

IoY – result extraction

IO Monad - (>>=) execution of ioX & ioY



ioX – state update

ioY – result extraction

IO Monad – >>= operator summary

the expression (**ioX** >>= **f**) has

type IO **a -> (a -> IO b) -> IO b**

ioX :: IO **a** has a **function** type of World **-> (a, World)**

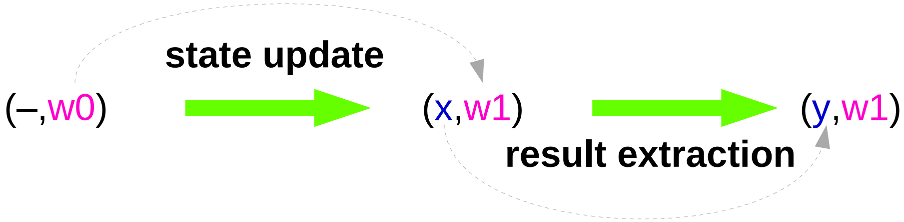
a **function** that takes **w0 :: World**,

returns **x :: a** and the new, updated **w1 :: World**

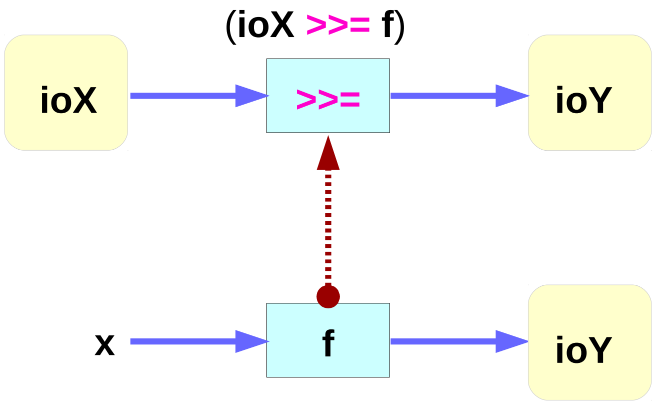
x and **w1** get passed to **f**, resulting in another IO monad,

which again is a **function** that takes **w1 :: World**

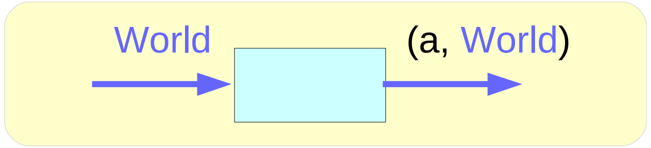
and returns **y** computed from **x** and the same **w1 :: World**



https://www.cs.hmc.edu/~adavidso/monads.pdf



ioX :: IO a **f :: a -> IO a** **ioY :: IO b**



ioX :: IO a

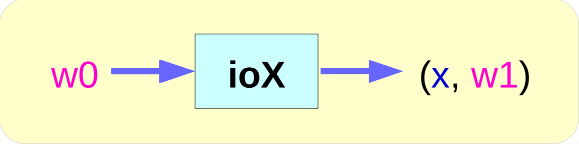
ioY :: IO b

ioX – state update

ioY – result extraction

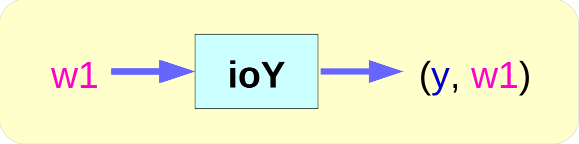
IO Monad – >>= operator binding

We give the **IOx** the **w0**	**w0 :: World**
we got back the updated **w1**	**w1 :: World**
and **x** out of its monad	**x :: a**
the **f** is given with	
the **x** with	**x :: a**
the updated **w1**	**w1 :: World**
.	
The final **IO** Monad	
takes **w1**	**w1 :: World**
returns **w1**	**w1 :: World**
and **y** out of its monad	**y :: a**
the expression (**ioX >>= f**) has	
type **IO a -> (a -> IO b) -> IO b**	



let (x, w1) = ioX w0

bind variables



let (y, w1) = ioY w0

bind variables

https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad Instance – implementation of **return** and **>>=**

```
instance Monad IO where
    return x w0 = (x, w0)

    (ioX >>= f) w0  =
        let  (x, w1) = ioX w0
        in   f x w1          -- has type (t, World)
```

```
type    IO t    =    World    ->    (t, World)
```

type synonym

https://www.cs.hmc.edu/~adavidso/monads.pdf

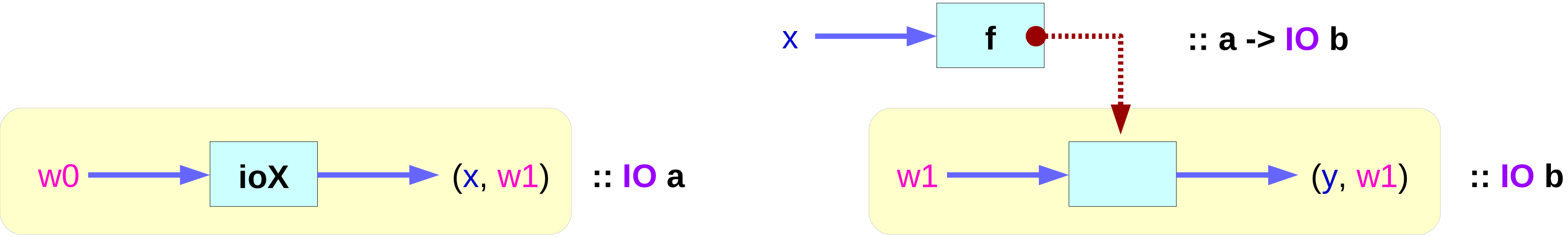
IO Monad – IO a, IO b types

```
instance Monad IO where
   return x w0 = (x, w0)

   (ioX >>= f) w0  =
      let  (x, w1) = ioX w0
      in   f x w1          -- has type (t, World)
```

ioX >>= f :: IO a -> (a -> IO b) -> IO b

`type   IO t   =   World   ->   (t, World)` type synonym



https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad – (a -> IO b) type function f

ioX >>= f :: IO a -> (a -> IO b) -> IO b

ioX :: IO a   w0 :: World   x :: a

f :: a -> IO b   w1 :: World

ioX w0 :: IO a World ➡ (x, w1)

f x :: IO b

f x w1 :: IO b World ➡ (y, w1)

f :: a -> IO b

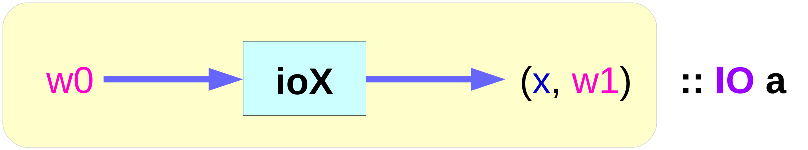
f x :: IO b

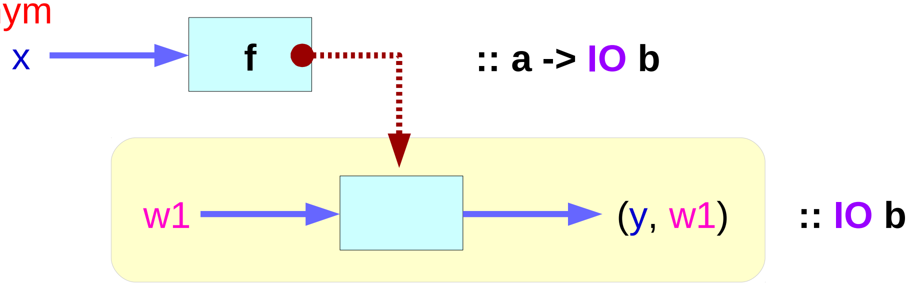
f :: a -> World -> (b World)

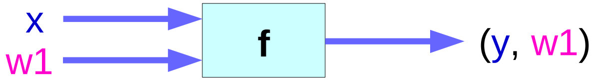
f x w1 :: IO b World

f x w1 :: (b World)

type IO t = World -> (t, World)   type synonym







https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad – binding variables

ioX >>= f :: IO a -> (a -> IO b) -> IO b

ioX :: IO a

f :: a -> IO b w0 :: World

x **:: a**

w1 **::** World

internal variables

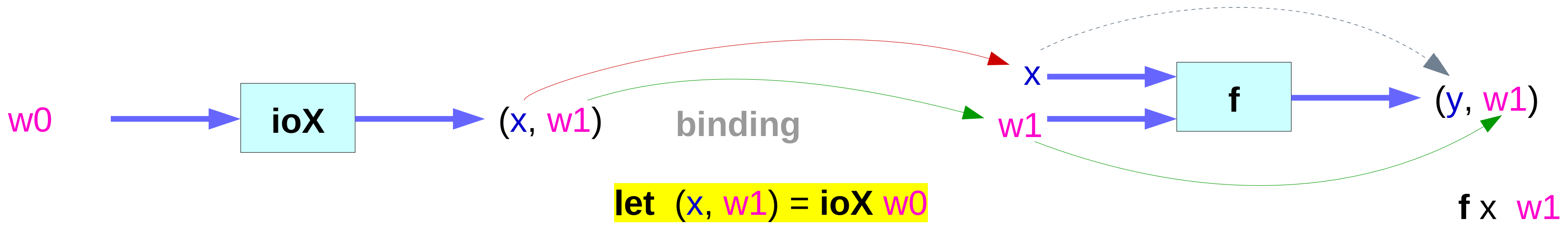
ioX w0 **:: IO a** World → (x, w1)

f x :: a -> a -> IO b

f x w1 **:: IO b** World → (y, w1)

ioX – monad execution

f – monad returning



let (x, w1) = **ioX** w0

https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad and ST Monad

```
instance Monad IO where
    return x w0 = (x, w0)

    (ioX >>= f) w0  =
        let    (x, w1) = ioX w0
        in     f x w1                -- has type (t, World)
```

```
instance Monad ST where
    return x  =  \s -> (x,s)

    st >>= f  =  \s -> let (x,s') = st s
                       in f x s'
    -- return :: a -> ST a
    -- (>>=)  :: ST a -> (a -> ST b) -> ST b
```

```
type   IO t   =   World   ->   (t, World)
```

type synonym

```
st >>= f  =  \w0 ->  let (x,w1) = st w0
                     in f x w1
```

https://www.cs.hmc.edu/~adavidso/monads.pdf

State Transformers ST Monad

```
instance Monad ST where
   -- return :: a -> ST a
   return x  =  \s -> (x,s)

   -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

>>= provides a means of sequencing **state transformers**:

st >>= f applies the **state transformer st** to an <u>initial</u> <u>state</u> s,

then applies the **function f** to the <u>resulting value</u> x

to give a second **state transformer** (f x),

which is then applied to the <u>modified</u> <u>state</u> s' to give the final result:

```
st >>= f  =  \s -> f x s'
       where (x,s') = st s
```

```
st >>= f  =  \s -> (y,s')
       where (x,s') = st s
             (y,s') = f x s'
```

(x,s') = **st** s

f x s'

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

Functors as containers

```
fmap   :: (a -> b) -> M a -> M b  -- functor

return :: a -> M a

join   :: M (M a) -> M a
```

the **functors-as-containers** metaphor

a **functor M** – a **container**

M a *contains* a value of type **a**

fmap allows functions to be applied to values in the **container**

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Function application, Packaging, Flattening

fmap applies a **function** to a **value** in a container

return packages a **value** in a container

join flattens a container in containers

applies

fmap :: (a -> b) -> M a -> M b -- functor

return :: a -> M a packaging

flatten

join :: M (M a) -> M a

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

>>= vs. fmap & join

(>>=) in terms of join and fmap

```
m >>= g = join (fmap g m)
```

fmap and join in terms of (>>=) and return

```
fmap f x = x >>= (return . f)
join x   = x >>= id
```

```
import Control.Monad
join (Just (Just 10))
   Just 10
join (Just (Just (Just 10)))
   Just (Just 10)
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

```
instance Monad [] where
  -- return :: a -> [a]
  return m  =  [m]
  -- (>>=)  :: [a] -> (a -> [b]) -> [b]
  m >>= g  =  concat (map g m)
```

```
  m >>= g  =  join (fmap g m)
```

```
fmap (*3) (Just 10)
Just 10 >>= return . (* 3)
   Just 30
```

```
join (Just (Just 10))
Just (Just 10)) >>= id
   Just 10
```

Monad’s lifting capability

a **Monad** is just a special **Functor** with extra features

Monads

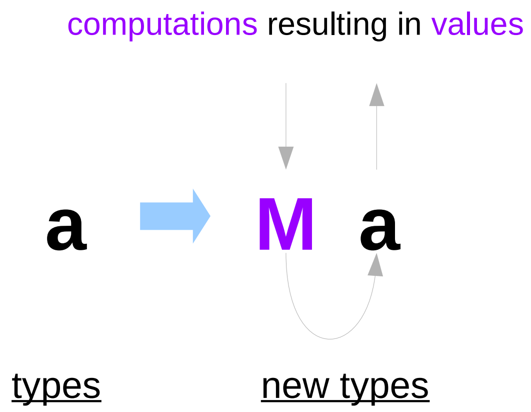
map types to new types

that represent "computations that result in values"

liftM (like **fmap**)

can **lift** regular functions into **Monad** types

(a -> b) **(m a -> m b)**



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

liftM Function

Control.Monad defines **liftM**

liftM transform a regular function

into a "computations that results in the value

obtained by evaluating the function."

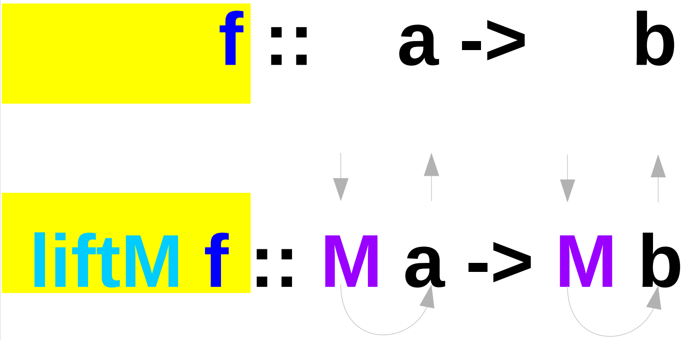
liftM :: (Monad m) => (a -> b) -> m a -> m b

liftM is merely

fmap implemented with (**>>=**) and **return**

fmap f x = x >>= (return . f)

liftM and **fmap** are therefore interchangeable.



f :: a -> b

liftM f :: M a -> M b

computations that results in the value obtained by evaluating the function

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Monad – mapping a type and lifting a function

mapping a new type

Monads map types to new types

that represent "computations that result in values"

The function **return** lifts a plain *value* **a** to **M a**

lifting function

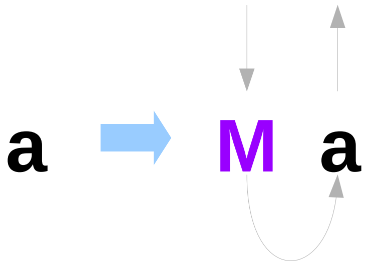
can lift **functions** into **Monad types**

via a very fmap-like function called **liftM**

that turns a regular function into a

"computation that results in the value

obtained by evaluating the function."

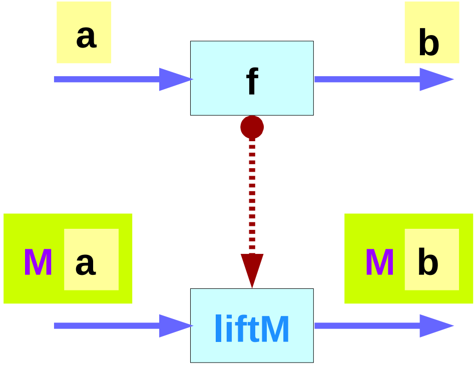


f :: a -> b

liftM f :: M a -> M b

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

liftM – function lifting



type lifting

a → M a

function lifting

f :: a -> b

liftM f :: M a -> M b

return – type lifting

The function **return** lifts a plain *value* **a** to **M a**

The *statements* in the imperative language **M** when executed, will result in the value **a** *without* any additional effects particular to **M**.

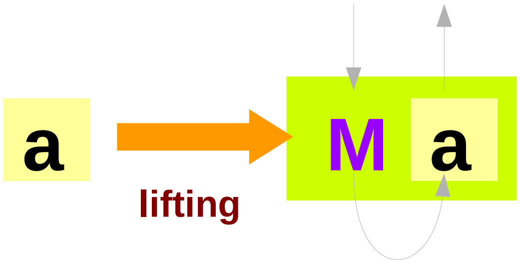
This is ensured by **Monad Laws**,

foo >>= return === foo

return x >>= k === k x;

foo >>= return
foo

return x >>= k
k x;



https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

ap Function

Control.Monad defines **ap** function

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

Analogously to the other cases,

ap is a monad-only version of (**<*>**).

M f :: M (a -> b)

ap M f :: M a -> M b

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

liftM vs fmap and ap vs <*>

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



ap :: Monad m => m (a -> b) -> m a -> m 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/Understanding_monads#cite_note-3

Monad – List Comprehension Examples

```
[x*2 | x<-[1..4], odd x]


do
   x <- [1..4]
   if odd x then [x*2]  else []



[1..4] >>= (\x -> if odd x then [x*2] else [])
                  1                        [2]
                  2                        [ ]
                  3                        [6]
                  4                        [ ]
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – I/O Examples

```
do
   putStrLn "What is your name?"
   name <- getLine
   putStrLn ("Welcome, " ++ name ++ "!")
```

getChar :: IO Char

Read a character from the standard input device

getLine :: IO String

Read a line from the standard input device

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – I/O Examples

Monads can be thought as computation builders.
the monad chains operations in some specific, useful way.

the **list comprehension** example:
if an operation returns a list,
then the following operations are performed
on every item in the list.

The **IO monad** example
the operations are performed sequentially,
but a hidden variable is passed along,
which represents the state of the world,
allows us to write I/O code in a pure functional manner.

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – A Parser Example

```
parseExpr = parseString <|> parseNumber

parseString = do
        char '"'
        x <- many (noneOf "\"")
        char '"'
        return (StringValue x)

parseNumber = do
    num <- many1 digit
    return (NumberValue (read num))
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – A Parser Example

The operations either match or don't match.
the monad manages the control flow:
The operations are performed sequentially
until a match fails, in which case the monad
backtracks to the latest <|> and tries the next option.
Again, a way of chaining operations
with some additional, useful semantics.

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – Asynchronous Examples

```
let AsyncHttp(url:string) =
   async {  let req = WebRequest.Create(url)
            let! rsp = req.GetResponseAsync()
            use stream = rsp.GetResponseStream()
            use reader = new System.IO.StreamReader(stream)
            return reader.ReadToEnd() }
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Three Orthogonal Functions

Thinking of extraction : a slightly misleading intuition.

Nothing is being "extracted" from a monad.

The more *fundamental* definition of a monad
can be stated by three orthogonal functions:

```
fmap :: (a -> b) -> (m a -> m b)
return :: a -> m a
join :: m (m a) -> m a
```

m is a monad.

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

Three Orthogonal Functions and >>=

```
fmap :: (a -> b) -> (m a -> m b)
return :: a -> m a
join :: m (m a) -> m a
```

```
(a ->     b) -> (m a -> m     b  )
(a -> m b) -> (m a -> m (m b))
(a -> m b) -> (m a -> m b)
```

how to implement (>>=) with these:

starting with arguments of type **m a** and **a -> m b**,

your only option is using **fmap** to get something of type **m (m b)**,

```
(a ->     b) -> (m a -> m     b  )
(a -> m b) -> (m a -> m (m b))
```

join to *flatten* the nested "layers" to get just **m b**.

```
(a -> m b) -> (m a ->m b)
```

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

Monad Law

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

nothing is being taken "out" of the monad

as the computation going *deeper* into the monad,

with successive steps being *collapsed*

into a single layer of the monad.

when **join** (m (m a) -> m a) is applied, it doesn't matter

as long as *the nesting order is preserved* (a form of *associativity*)

that the *monadic* *layer* introduced by **return**

does *nothing* (an *identity* value for **join**).

Left identity	**return a >>= f**	**f a**
Right identity	**m >>= return**	**m**
Associativity	**(m >>= f) >>= g**	**m >>= (\x -> f x >> g)**

```
 (a ->   b) -> (m a -> m b)
(a -> m b) -> (m a -> m (m b))
(a -> m b) -> (m a -> m b)
```

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf