

Monad Background (3A)

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.

This document was produced by using OpenOffice.

Based on

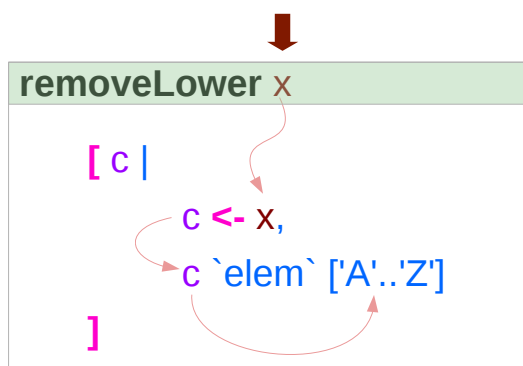
[Haskell in 5 steps](https://wiki.haskell.org/Haskell_in_5_steps)

https://wiki.haskell.org/Haskell_in_5_steps

A List Comprehension Function

```
let removeLower x = [c | c <- x, c `elem` ['A'..'Z']]
```

a list comprehension



“Hello”

```
[ c: 'H'  
  c: 'e'  
  c: 'l'  
  c: 'l'  
  c: 'o' ]
```

“H”

```
do { x1 <- action1  
    ; x2 <- action2  
    ; mk_action3 x1 x2 }
```

`x1` : Return value of action1

`x2`: Return value of action2

<https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell>

Pattern and Predicate

```
let removeLower x = [c | c <- x, c `elem` ['A'..'Z']]
```

a list comprehension

```
[c | c <- x, c `elem` ['A'..'Z']]
```

`c <- x` is a **generator**

(`x` : argument of the function `removeLower`)

`c` is a **pattern**

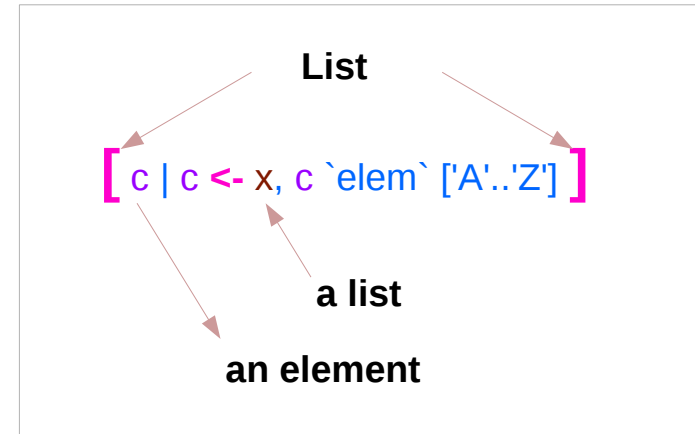
matching from the **elements** of the **list** `x`

successive binding of `c` to the **elements** of the **list** `x`

`c `elem` ['A'..'Z']`

is a **predicate** which is applied to each successive binding of `c`

Only `c` which passes this predicate will appear in the output list



<https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell>

Assignment in Haskell

Assignment in Haskell : declaration with initialization:

- no uninitialized variables,
- must declare with an initial value
- no mutation
- a variable keeps its initial value throughout its scope.

<https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell>

Generator

```
[c | c <- x, c `elem` ['A'..'Z']]
```

```
filter (`elem` ['A' .. 'Z']) x
```

```
[ c | c <- x ]
```

c: an element
x: a list

```
do c <- x  
  return c
```

```
x >>= (\c -> return c)
```

```
x >>= return
```

c: an element
x: an element

or

c: a list
x: a list

```
action1 >>= (\ x1 ->  
  action2 >>= (\ x2 ->  
    mk_action3 x1 x2 ))
```

<https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell>

Anonymous Functions

```
(\x -> x + 1) 4  
5 :: Integer
```

```
(\x y -> x + y) 3 5  
8 :: Integer
```

```
inc1 = \x -> x + 1
```

```
incListA lst = map inc2 lst  
  where inc2 x = x + 1
```

```
incListB lst = map (\x -> x + 1) lst
```

```
incListC = map (+1)
```

https://wiki.haskell.org/Anonymous_function

Then Operator (>>) and do Statements

a chain of actions

to sequence input / output operations

the (>>) (**then**) operator works almost identically in **do** notation

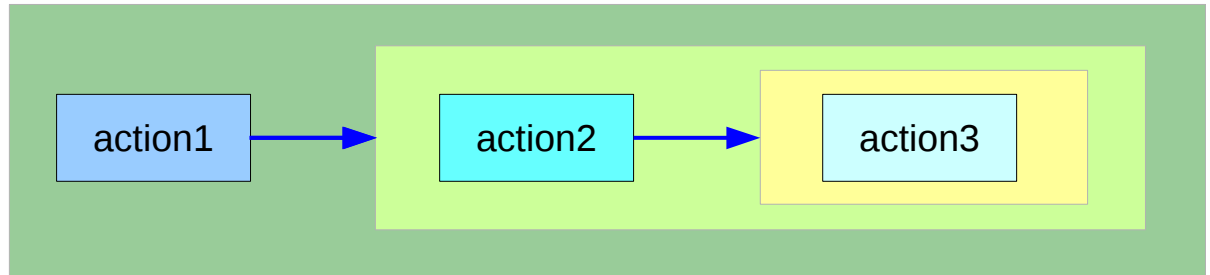
```
putStr "Hello" >>  
putStr " " >>  
putStr "world!" >>  
putStr "\n"
```

```
do { putStr "Hello"  
    ; putStr " "  
    ; putStr "world!"  
    ; putStr "\n" }
```

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

Chaining in `do` and `>>` notations

```
do { action1  
  ; action2  
  ; action3 }
```



```
do { action1  
  ; do { action2  
        ; action3 } }
```



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

can **chain** any actions
all of which are in **the same monad**

```
do { action1  
  ; do { action2  
        ; do { action3 } } }
```



```
action1 >>  
  action2 >>  
    action3
```

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

Bind Operator (`>=>`) and `do` statements

The bind operator (`>=>`)

passes a value `->`

(the result of an action or function),

downstream in the binding sequence.

```
action1 >=> (\ x1 ->
  action2 >=> (\ x2 ->
    mk_action3 x1 x2 ))
```

anonymous function
(lambda expression)
is used

`do` notation assigns a variable name

to the passed value using the `<-`

```
do { x1 <- action1
    ; x2 <- action2
    ; mk_action3 x1 x2 }
```

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

Chaining `>>=` and `do` notations

`->`

```
action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))
```

```
action1
```

```
>>=
```

```
(\ x1 -> action2
```

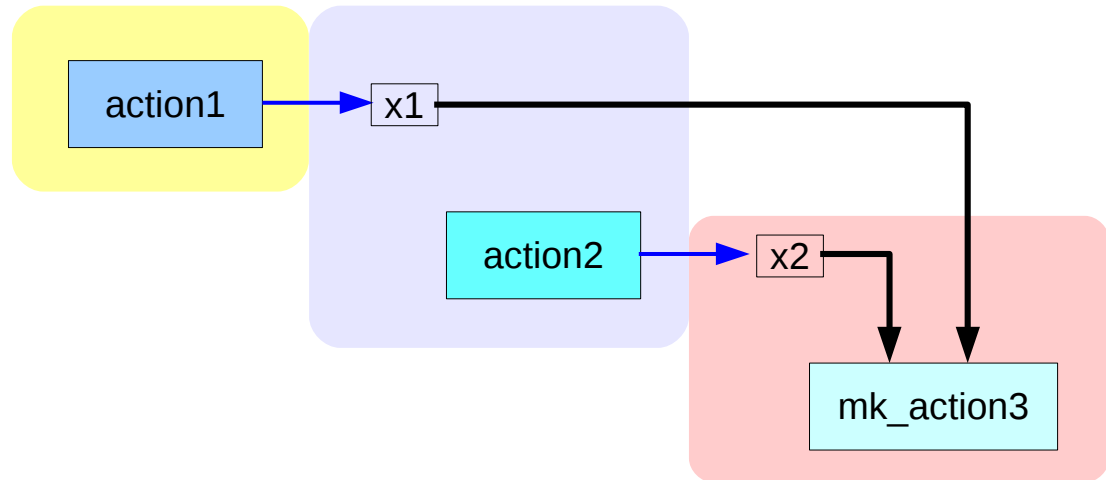
```
>>=
```

```
(\ x2 -> mk_action3 x1 x2 ))
```

```
action1 >>= (\ x1 ->  
  action2 >>= (\ x2 ->  
    mk_action3 x1 x2 ))
```

`<-`

```
do { x1 <- action1  
    ; x2 <- action2  
    ; mk_action3 x1 x2 }
```



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

fail method

```
do { Just x1 <- action1
    ; x2 <- action2
    ; mk_action3 x1 x2 }
```

```
do { x1 <- action1
    ; x2 <- action2
    ; mk_action3 x1 x2 }
```

O.K. when `action1` returns `Just x1`

when `action1` returns `Nothing`

crash with a non-exhaustive patterns error

Handling failure with `fail` method

```
action1 >>= f where
  f (Just x1) = do { x2 <- action2
                  ; mk_action3 x1 x2 }
  f _         = fail "..."
```

-- A compiler-generated message.

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

Example

```
nameDo :: IO ()
nameDo = do { putStr "What is your first name? "
             ; first <- getLine
             ; putStr "And your last name? "
             ; last <- getLine
             ; let full = first ++ " " ++ last
             ; putStrLn ("Pleased to meet you, " ++ full ++ "!") }
```

A possible translation into vanilla monadic code:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \ first ->
             putStr "And your last name? " >>
             getLine >>= \ last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

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

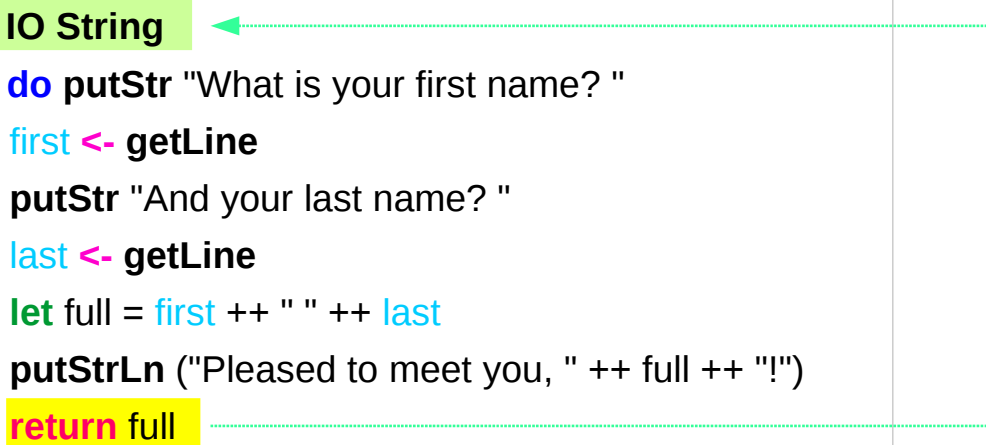
```
do { x1 <- action1
    ; x2 <- action2
    ; mk_action3 x1 x2 }
```

using the **do** statement

using **then (>>)** and **Bind (>>=)** operators

return method

```
nameReturn :: IO String  
nameReturn = do putStr "What is your first name? "  
               first <- getLine  
               putStr "And your last name? "  
               last <- getLine  
               let full = first ++ " " ++ last  
               putStrLn ("Pleased to meet you, " ++ full ++ "!")  
               return full
```

A diagram consisting of a green dotted line that starts from the `return full` line in the code block above, moves horizontally to the right, then vertically up, then horizontally left, ending with an arrowhead pointing to the `IO String` type signature in the first line of the code block.

```
greetAndSeeYou :: IO ()  
greetAndSeeYou = do name <- nameReturn  
                    putStrLn ("See you, " ++ name ++ "!")
```

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

Without a **return** method

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
               first <- getLine
               putStr "And your last name? "
               last <- getLine
               let full = first ++ " " ++ last
               putStrLn ("Pleased to meet you, " ++ full ++ "!")
               return full
```

explicit return statement
returns **IO String** monad

```
nameDo :: IO ()
nameDo = do { putStr "What is your first name? "
             ; first <- getLine
             ; putStr "And your last name? "
             ; last <- getLine
             ; let full = first ++ " " ++ last
             ; putStrLn ("Pleased to meet you, " ++ full ++ "!") }
```

no return statement
returns **empty IO** monad

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

return method – not a final statement

```
nameReturnAndCarryOn :: IO ()  
nameReturnAndCarryOn = do putStr "What is your first name? "  
    first <- getLine  
    putStr "And your last name? "  
    last <- getLine  
    let full = first++" "+last  
    putStrLn ("Pleased to meet you, "+full+"!")  
    return full  
    putStrLn "I am not finished yet!"
```

the return statement does not interrupt the flow
the last statements of the sequence returns a value

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

Data Constructor

```
data Color = Red | Green | Blue
```

Color is a type

Red is a constructor that contains a value of type **Color**.

Green is a constructor that contains a value of type **Color**.

Blue is a constructor that contains a value of type **Color**.

<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

Data Constructor with Parameters

```
data Color = RGB Int Int Int
```

Color is a type

RGB is not a value but a *function* taking three Int's and *returning a value*

```
RGB :: Int -> Int -> Int -> Color
```

RGB is a **data constructor** that is a *function* taking three **Int** values as its arguments, and then uses them to construct a new value.

<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

Type Constructor

Consider a binary tree to store **Strings**

```
data SBTree = Leaf String | Branch String SBTree SBTree
```

a type

SBTree is a **type**

Leaf is a **data constructor** (a function)

Branch is a **data constructor** (a function)

Leaf :: **String** -> **SBTree**

Branch :: **String** -> **SBTree** -> **SBTree** -> **SBTree**

<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

Similar Type Constructors

Consider a binary tree to store **Strings**

```
data SBTree = Leaf String | Branch String SBTree SBTree
```

Consider a binary tree to store **Bool**

```
data BBTree = Leaf Bool | Branch Bool BBTree BBTree
```

Consider a binary tree to store **a parameter type**

```
data BTree a = Leaf a | Branch a (BTree a) (BTree a)
```

<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

Type Constructor with a Parameter

Type constructors

Both **SBTree** and **BBTree** are type constructors

```
data SBTree = Leaf String | Branch String SBTree SBTree
data BBTree = Leaf Bool | Branch Bool BBTree BBTree
```

```
data BTree a = Leaf a | Branch a (BTree a) (BTree a)
```

Now we introduce a type variable **a** as a parameter to the type constructor.

BTree has become a function.

It takes a type as its argument and it returns a new type.

<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

Type Constructors and Data Constructors

A type constructor

- a "function" that takes 0 or more types
- gives you back a new **type**.

Type constructors with parameters

allows slight variations in types

A data constructor

- a "function" that takes 0 or more values
- gives you back a new **value**.

Data constructors with parameters

allows slight variations in values

```
type SBTree = BTree String
```

```
type BBTree = BTree Bool
```

```
RGB 12 92 27
```

```
#0c5c1b
```

```
RGB 255 0 0
```

```
RGB 0 255 0
```

```
RGB 0 0 255
```

<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

()

() is both a **type** and a **value**.

() is a special **type**, pronounced “unit”,
has one **value** (), sometimes pronounced “void”

the **unit type** has only one **value** which is called **unit**.

() :: () Type :: Expression

It is the same as the **void type void** in Java or C/C++.

<https://stackoverflow.com/questions/20380465/what-do-parentheses-used-on-their-own-mean>

Unit Type

a **unit type** is a type that allows only one value (and thus can hold no information).

It is the same as the **void type** **void** in Java or C/C++.

```
:t  
Expression :: Type
```

```
data Unit = Unit
```


```
Prelude> :t Unit  
Unit :: Unit
```

```
Prelude> :t ()  
() :: ()
```

<https://stackoverflow.com/questions/20380465/what-do-parentheses-used-on-their-own-mean>

Type Language and Expression Language

```
data Tconst Tvar ... Tvar = Vconst type ... type | ...  
                          Vconst type ... type
```



A new datatype declaration

Tconst (Type Constructor)

is added to *the type language*

Vconst (Value Constructor)

is added to *the expression language* and *its pattern sublanguage*
must not appear in *types*

Argument types in **Vconst** type ... type



are the types given to the arguments (**Tconst** Tvar ... Tvar)
are used in expressions

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

Datatype Declaration Examples

```
data Tree a = Leaf | Node (Tree a) (Tree a)
```

Tree (Type Constructor)

Leaf or **Node** (Value Constructor)

```
data Type = Value
```

```
data () = ()
```

() (Type Constructor)

() (Value Constructor)

the type (), often pronounced "Unit"

the value (), sometimes pronounced "void"

the type () containing only one value ()

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

IO ()

```
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 exciting.

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

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>

IO ()

Monadic operations tend to have types which look like

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

where the return type is a type application:

the function tells you which effects are possible

and the argument tells you what sort of value

is produced by the operation

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

Variable definition in a file

Var1.hs

```
r = 5
```

Var2.hs

```
r = 55
```

definition with initialization

```
young@Sys ~ $ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
Prelude> :load Var1.hs
[1 of 1] Compiling Main          ( var.hs, interpreted )
Ok, modules loaded: Main.
*Main> r
5
*Main> :t r
r :: Integer
*Main>
*Main> :load Var2.hs
[1 of 1] Compiling Main          ( var2.hs, interpreted )
Ok, modules loaded: Main.
*Main> r
55
```

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

No Mutation

Var1.hs

```
r = 5
```

Var2.hs

```
r = 55
```

No mutation

```
*Main> r = 33
```

```
<interactive>:12:3: parse error on input '='
```

```
young@Sys ~ $ ghci
```

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
```

```
Prelude> r = 333
```

```
<interactive>:2:3: parse error on input '='
```

```
Prelude>
```

```
let r = 33
```

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

Functional & Imperative Languages

Imperative programming:

- variables as **changeable locations** in a computer's memory
- imperative programs **explicitly commands** the computer what to do

functional programming

- a way to think in higher-level **mathematical terms**
- defining how variables **relate** to one another
- leaving the **compiler** to translate these
to the step-by-step instructions that the computer can process.

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

Redefinition : not allowed

```
r = 5
```

```
r = 2
```

imperative programming:

after setting $r = 5$ and then changing it to $r = 2$.

Haskell programming:

an error: "multiple declarations of r".

Within a given scope, a variable in Haskell gets defined only once and cannot change.

like variables in mathematics.

Immutable: They vary only based on the data we enter into a program.

We can't define r two ways in the same code,

but we could change the value **by changing the file**

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

Recursion

```
r = r + 1
```

imperative programming:

incrementing the variable r
(updating the value in memory)

Haskell programming:

a recursive definition of r
(defining it in terms of itself)

if r had been defined with any value beforehand,
then $r = r + 1$ in Haskell would bring an error message.

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

Data Dependence

$$y = x * 2$$

$$x = 3$$

$$x = 3$$

$$y = x * 3$$

Haskell programming:

because their values of variables do not change within a program

variables can be defined in any order

there is no notion of "x being declared before y" or the other way around.

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

Evaluation

```
area 5
=> { replace the left-hand side area r = ... by the right-hand side ... = pi * r^2 }
    pi * 5 ^ 2
=> { replace pi by its numerical value }
    3.141592653589793 * 5 ^ 2
=> { apply exponentiation (^) }
    3.141592653589793 * 25
=> { apply multiplication (*) }
    78.53981633974483
```

replace each function with its definition
calculate the results until a single value remains.
to apply or call a function means
to replace the left-hand side of its definition by its right-hand side.

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

Type Synonyms

```
type String = [Char]
```

```
phoneBook :: [(String,String)]
```

```
type PhoneBook = [(String,String)]
```

```
phoneBook :: PhoneBook
```

```
type PhoneNumber = String
```

```
type Name = String
```

```
type PhoneBook = [(Name,PhoneNumber)]
```

```
phoneBook :: PhoneBook
```

```
phoneBook =
```

```
  [("betty","555-2938")  
   ,("bonnie","452-2928")  
   ,("patsy","493-2928")  
   ,("lucille","205-2928")  
   ,("wendy","939-8282")  
   ,("penny","853-2492")  
  ]
```

<http://learnyouahaskell.com/making-our-own-types-and-typeclasses>

Side Effects in Haskell

Generally, a monad cannot perform side effects in Haskell.
there is one exception: **IO monad**

Suppose there is a type called **World**,
which contains all the state of the external universe

A way of thinking what IO monad does

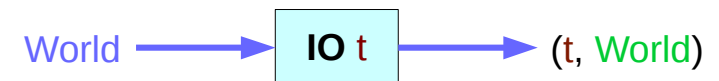
```
type IO t = World -> (t, World) type synonym
```

IO t is a function

input : a **World**

output: the **t** it's supposed to contain,
a new, updated **World** obtained
by modifying the given **World**
in the process of computing the **t**.

World -> (t, **World**)

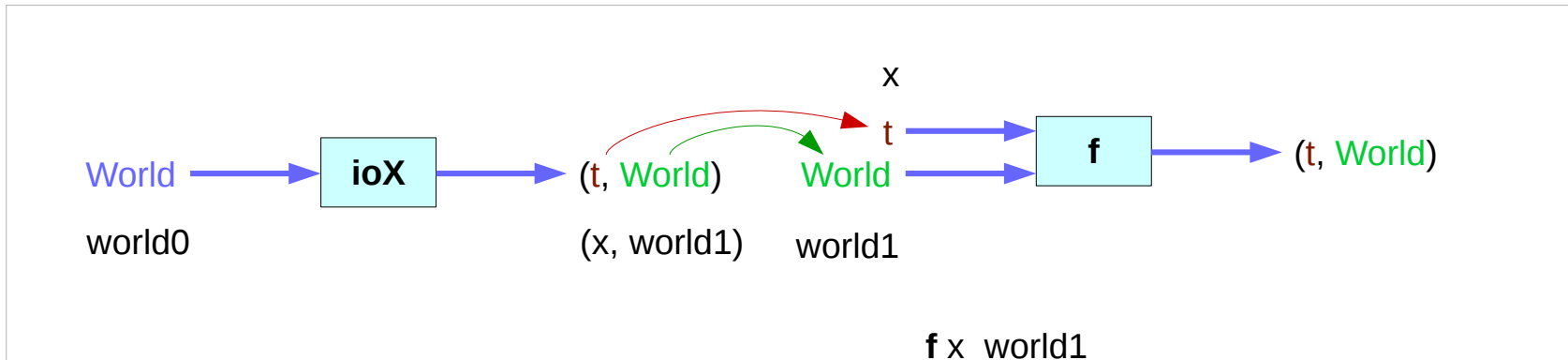


IO x world0 (x, world1)

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

Side Effects in Haskell

```
instance Monad IO where
  return x world = (x, world)
  (ioX >>= f) world0 =
  let
    (x, world1) = ioX world0
  in
    f x world1           -- Has type (t, World)
```

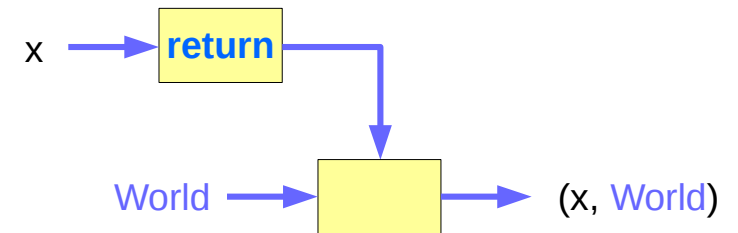


<https://www.cs.hmc.edu/~adavidso/monads.pdf>

Side Effects in Haskell

The return function takes x
and gives back a function
that takes a World
and returns x along with the “new, updated” World
formed by not modifying the World it was given

`return x world = (x, world)`



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

Side Effects in Haskell

the expression $(\text{ioX} \gg= \text{f})$ has type $\text{World} \rightarrow (\text{t}, \text{World})$

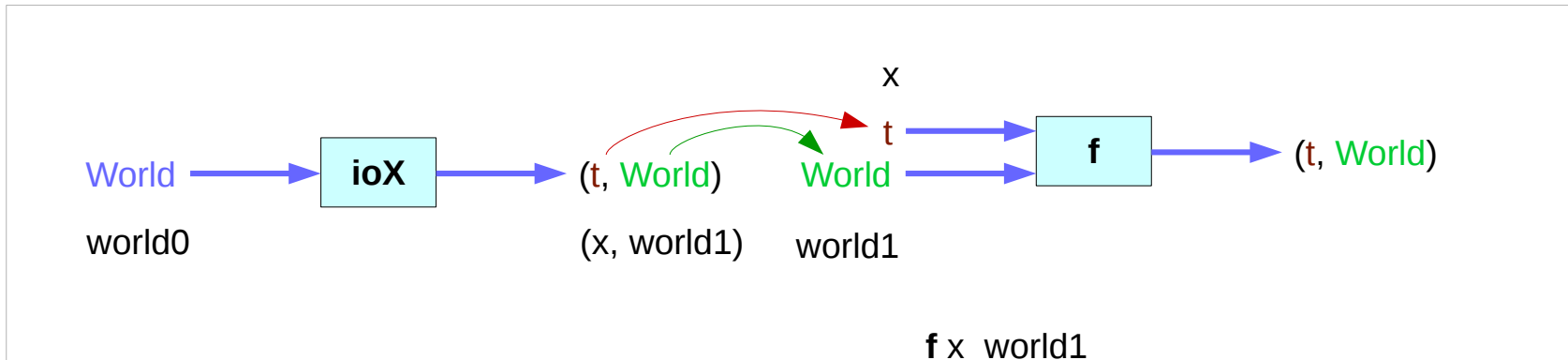
a function that takes a World , called world0 , which is used to extract x from its IO monad.

This gets passed to f , resulting in another IO monad,

which again is a function that takes a World and returns a x and a new, updated World .

We give it the World we got back from getting x out of its monad, and the thing it gives back to us is the t with a final version of the World

the implementation of bind



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

Record Syntax (named field)

```
data Configuration = Configuration
  { username      :: String
  , localhost     :: String
  , currentDir    :: String
  , homeDir       :: String
  , timeConnected :: Integer
  }
```

```
username :: Configuration -> String
```

```
-- accessor function (automatic)
```

```
localhost :: Configuration -> String
```

```
-- etc.
```

```
changeDir :: Configuration -> String -> Configuration
```

```
-- update function
```

```
changeDir cfg newDir =
```

```
  if directoryExists newDir      -- make sure the directory exists
```

```
    then cfg { currentDir = newDir }
```

```
    else error "Directory does not exist"
```

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

newtype and data

data  **newtype**

Data can only be replaced with newtype
if the type has exactly *one constructor* with exactly *one field* inside it.

It ensures that the trivial **wrapping** and **unwrapping**
of **the single field** is eliminated by the **compiler**.

simple wrapper types such as **State** are usually defined with **newtype**.

type : used for type synonyms

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

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

newtype examples

```
newtype Fd = Fd CInt
-- data Fd = Fd CInt would also be valid

-- newtypes can have deriving clauses just like normal types
newtype Identity a = Identity a
  deriving (Eq, Ord, Read, Show)

-- record syntax is still allowed, but only for one field
newtype State s a = State { runState :: s -> (s, a) }

-- this is not allowed:
-- newtype Pair a b = Pair { pairFst :: a, pairSnd :: b }
-- but this is:
data Pair a b = Pair { pairFst :: a, pairSnd :: b }
-- and so is this:
newtype NPair a b = NPair (a, b)
```

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

References

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