

# Side Effects (3B)

---

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

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

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).

This document was produced by using OpenOffice.

# Based on

---

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

[https://wiki.haskell.org/Haskell\\_in\\_5\\_steps](https://wiki.haskell.org/Haskell_in_5_steps)

# Variables

## 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](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Haskell Language Features

## Haskell Functional Programming

- **Immutability**
- **Recursive Definition**
- **No Data Dependency**

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Redefinition : not allowed

**imperative programming:**

**r = 5**

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

**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.

**r = 5**



~~**r = 2**~~

**No mutation  
In Haskell**

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Variables in a file

## Immutable:

they can change only based on  
*the data we enter to run the program.*

We cannot define `r` two ways in the same code,  
but we could change the value **by changing the file**

## Vars.hs

```
a = 100
```

```
r = 5
```

```
pi = 3.14159
```

```
e = 2.7818
```

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Loading a variable definition file

```
$ 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
```

:load **Var1.hs**

:load **Var1.hs**

definition with initialization

Var1.hs

r = 5

Var2.hs

r = 55

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)



# No Mutation

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

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

```
$ ghci
```

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

```
Prelude> r = 333
```

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

```
Prelude>
```

No mutation, Immutable

```
let r = 33
```

```
let r = 33
```

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Recursive Definition

**imperative programming:**

$r = r + 1$

incrementing the variable  $r$

(**updating** the value in memory)

**Haskell programming:**

a **recursive definition** of  $r$

(defining it in terms of itself)


Side effect, Stateful computation

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](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# No Data Dependence

$y = x * 2$   
 $x = 3$

A curved arrow points from the variable 'x' in the second line to the variable 'x' in the first line, indicating that the value of 'y' depends on the value of 'x'.

$x = 3$   
 $y = x * 3$

A curved arrow points from the variable 'x' in the second line to the variable 'x' in the first line, indicating that the value of 'y' depends on the value of 'x'.

## Haskell programming:

because the values of variables do not change

variables can be defined in any order

no mandatory : "x being declared before y"

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Evaluation

area 5

=> { replace the LHS `area r = ...` by the RHS `... = 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`

`area r = pi * r^2`

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Translation to instructions

## functional programming

- leaving the **compiler** to **translate** these to the step-by-step **instructions** that the computer can process.

**replace** each **function** and **variable** with its **definition**  
**repeatedly replace** the results **until a single value remains**.

to apply or call a function means

to **replace the LHS** of its **definition** by **its RHS**.

[https://en.wikibooks.org/wiki/Haskell/Variables\\_and\\_functions](https://en.wikibooks.org/wiki/Haskell/Variables_and_functions)

# Side Effects Definition

a **function** or **expression** is said to have a **side effect** if it modifies some state outside its scope or has an observable interaction with its calling functions or the outside world besides returning a value.

a particular function might

- modify a **global** variable or **static** variable,
- modify one of its **arguments**,
- raise an **exception**,
- write data to a **display** or **file**,
- read data from a **keyboard** or **file**, or
- call *other side-effecting functions*.

[https://en.wikipedia.org/wiki/Side\\_effect\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

# History, Order, and Context

In the presence of **side effects**,  
a program's behaviour may depend on **history**;

the **order** of **evaluation** matters.  
the **context** and **histories**

Imperative programming : frequent utilization of **side effects**.  
functional programming : **side effects** are rarely used.

The lack of side effects makes it easier  
to do **formal verifications** of a program

[https://en.wikipedia.org/wiki/Side\\_effect\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

# Side Effects Examples in C

```
int i, j;  
i = j = 3;  
  
i = (j = 3);    // j = 3 returns 3, which then gets assigned to i
```

```
// The assignment function returns 10  
// which automatically casts to "true"  
// so the loop conditional always evaluates to true  
  
while (b = 10) { }
```

[https://en.wikipedia.org/wiki/Side\\_effect\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))



# Pure Languages

Haskell is a **pure** language

programs are made of **functions**

that can't change

any global state or variables,

they can only do

some computations and return them results.

every variable's value does not change in time

However, some problems are inherently stateful

in that they rely on some state that changes over time.

~~st1 = 10~~

s -> (x,s)

st1 (v,10)

a bit tedious to model

Haskell has the **state monad** features

<http://learnyouahaskell.com/for-a-few-monads-more>

# Side Effects in Haskell

The functional language Haskell expresses side effects  
such as **I/O** and  
other **stateful computations**  
using **monadic actions**  
**state monad**

[https://en.wikipedia.org/wiki/Side\\_effect\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

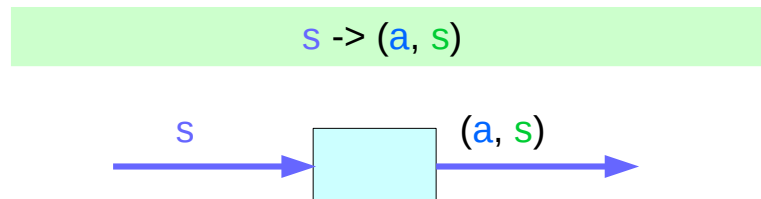
# Stateful Computation

a **stateful computation** is a **function** that  
takes some **state** and  
returns a **value** along with some **new state**.

That function would have the following type:

```
s -> (a,s)
```

**s** is the type of the **state** and  
**a** the **result** of the **stateful computation**.



<http://learnyouahaskell.com/for-a-few-monads-more>

# Assignment

**Assignment** in an imperative language :

will assign the value 5 to the variable x

will have the value 5 as an *expression*

**Assignment** in a functional language

as a **function** that

takes a **state** and

returns a **result** and a **new state**

**x = 5**

<http://learnyouahaskell.com/for-a-few-monads-more>

# Assignment as a stateful computation

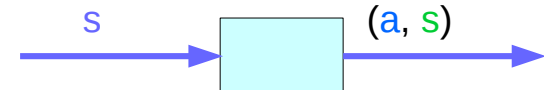
**Assignment** in a functional language  
as a **function** that  
takes a **state** and  
returns a **result** and a **new state**

an input **state** :  
all the variables that have been assigned previously  
a **result** : 5  
a **new state** :  
all the previous variable mappings plus  
the newly assigned variable.

<http://learnyouahaskell.com/for-a-few-monads-more>

**x = 5**

$s \rightarrow (a, s)$



all the variables  
that have been  
assigned  
previously

**a = 1**  
**b = 2**

all the previous  
variable mappings  
plus the newly  
assigned variable

**a result : 5**

**a = 1**  
**b = 2**  
**x = 5**

# A value with a context

The **stateful computation**:

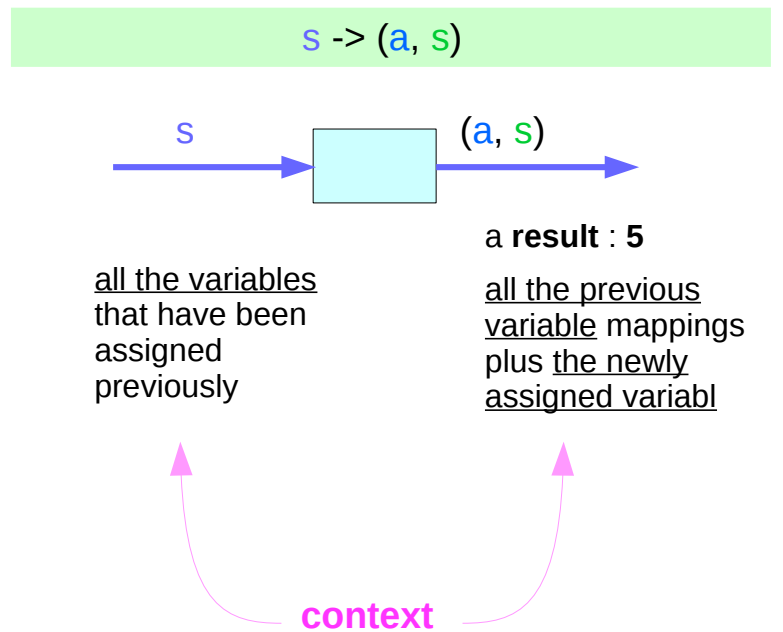
- a **function** that
  - takes a **state** and
  - returns a **result** and a **new state**
- can be considered as **a value with a context**

the actual **value** is

the **result**

the **context** is

that we have to provide an **initial state** to get the result and that apart from getting the result we also get a **new state**.



<http://learnyouahaskell.com/for-a-few-monads-more>

# 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://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 Operation

**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:

```
effect-monad val-out-type
```

the function tells you  
which **effects** are possible

the argument tells you (val-out-type)  
what sort of **value** is produced **by the operation**

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

```
putStr :: String -> IO ()
```

```
IO ()
```

function: effect-monad

argument: val-out-type

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



# Monadic Operation – 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 exciting.

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

# Side Effects of IO Monad

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
```

```
putStr :: String -> IO ()
```

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



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

# Type Synonym **IO t**

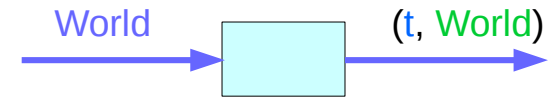
**IO t** is a **parameterized function**

*input* : a **World**

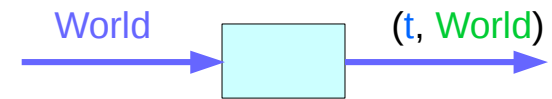
*output*: a value of the type **t** and a new **updated World**  
obtained by modifying the given **World**  
in the process of computing the value of the type **t**.

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

World -> (t, World)



IO t



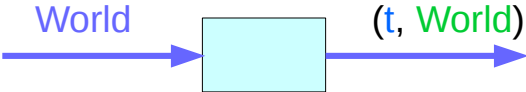
<https://www.cs.hmc.edu/~adavidso/monads.pdf>

# A Parameterized Function of IO Monad (1)

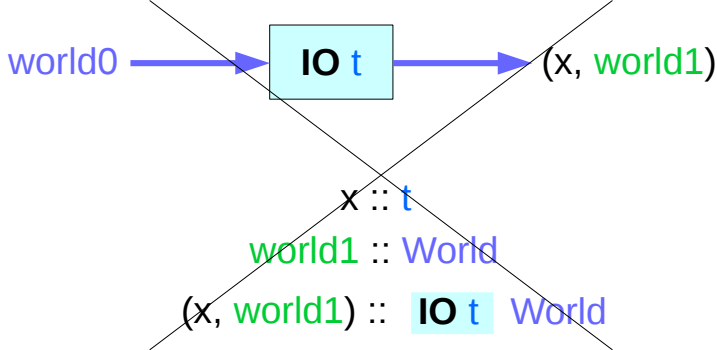
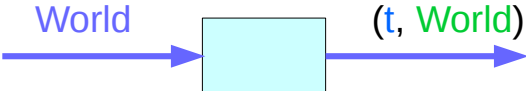
```
type IO t = World -> (t, World)
```

type synonym

```
World -> (t, World)
```



```
IO t
```



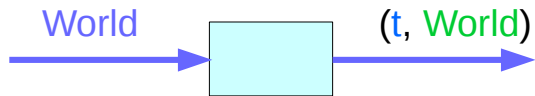
<https://www.cs.hmc.edu/~adavidso/monads.pdf>

# A Parameterized Function of IO Monad (2)

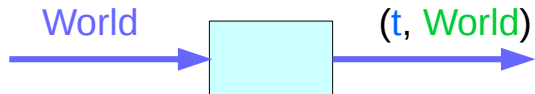
```
type IO t = World -> (t, World)
```

type synonym

```
World -> (t, World)
```



```
IO t
```



```
Func :: IO t
```

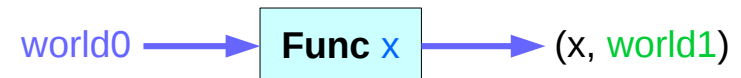
```
Func x world0 = (x, world1)
```

```
x :: t
```

```
world0 :: World
```

```
world1 :: World
```

```
(x, world1) :: IO t World
```



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

# Implementation of IO t

It is impossible

to store the extra copies of the contents of your hard drive  
that each of the Worlds contains

given World → updated World

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

# IO Monad

We give **IO** the **World**

we got back the **World**

from getting **x** out of its monad,

and the thing **IO** gives back to us is

the **y** with

a final version of the **World**

**world0** :: **World**

**world1** :: **World**

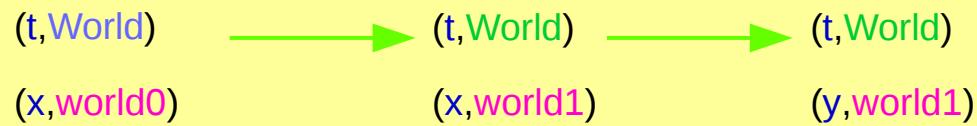
**x** :: **t**

**y** :: **t**

**world1** :: **World**

.

**the implementation of bind**



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

# IO Monad in GHC

Which World was given initially?

Which World was updated?

In **GHC**, a **main** must be defined somewhere with type **IO ()**

a program execution starts from the **main**

the **initial World** is contained in the **main** to start everything off

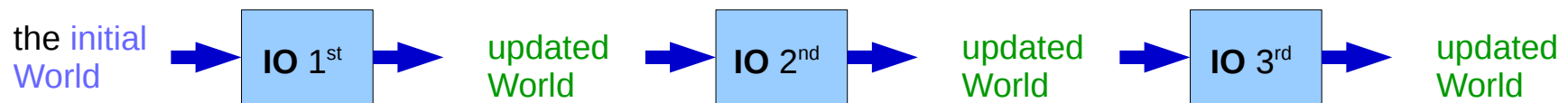
the **main** passes the **updated World** from each **IO**

to the next **IO** as its **initial World**

an **IO** that is not reachable from **main** will never be executed

an **initial / updated World** is not passed to such an **IO**

## The modification of the World



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

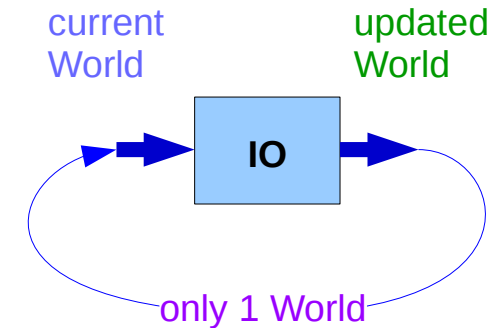


# IO Monad in GHCi

when using **GHCI**,  
everything is wrapped in **an implicit IO**,  
since the results get printed out to the screen.

there's **only 1 World** in existence at any given moment.  
Each IO takes that **one and only World**, consumes it,  
and gives back a single new World.  
Consequently, there's no way to accidentally run out of Worlds,  
or have multiple ones running around.

**the implementation of bind**



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

Every time a new command is given to **GHCI**,  
**GHCI** passes the current World to **IO**,  
**GHCI** gets the result of the command back,  
**GHCI** request to display the *result*

(which updates the World by modifying

- the contents of the screen or
- the list of defined variables or
- the list of loaded modules or whatever),

**GHCI** saves the new World to process the next command.

**the implementation of bind**

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

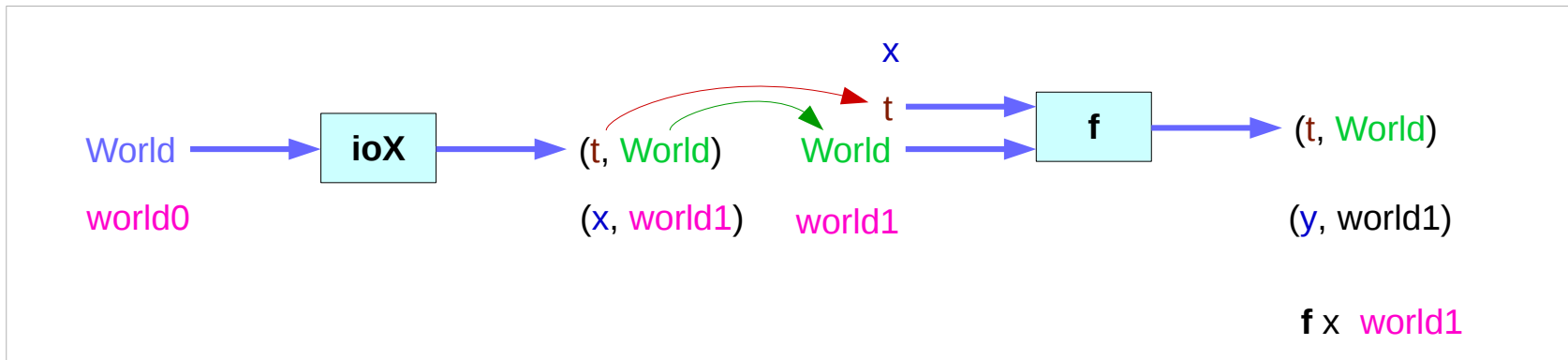
# IO Monad Implementation

```
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)
```

$(x, s) \longrightarrow (x, s') \longrightarrow (y, s')$

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



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

# Monad IO and Monad ST

**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)
```

**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'
```

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

type synonym

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

# State Transformers ST

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 initial state **s**,

then applies the function **f** to the resulting value **x**

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

which is then applied to the modified state **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
```

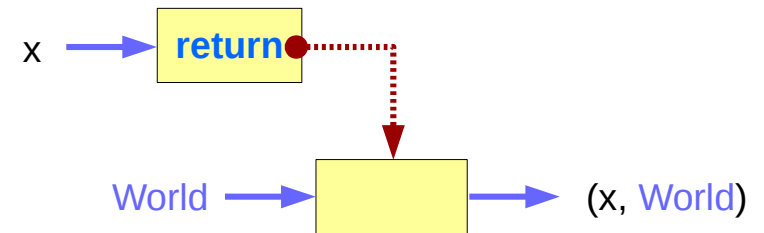
```
f x s'
```

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

# Monad IO - return

The **return** function takes  $x$   
and gives back a function  
that takes a **World**  
and returns  $x$  along with the new, **updated World (=World)**  
formed by not modifying the **World** it was given

**return**  $x$  world = (x, world)



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

# Monad IO - >>=

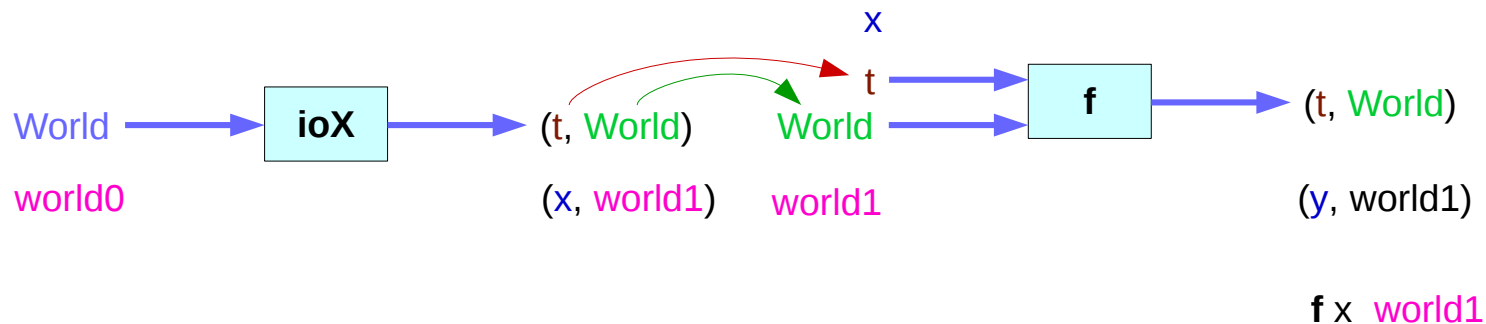
the expression  $(\text{ioX} \gg= \text{f})$  has

type  $\text{World} \rightarrow (\text{t}, \text{World})$

a function **ioX** that takes **world0** of the type **World**,  
which is used to extract **x** from its **IO** monad.

**x** gets passed to **f**, resulting in another **IO** monad,  
which again is a function that takes **world1** of the type **World**  
and returns a **y** and a new, updated **World**.

the implementation of bind



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

## References

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