

# 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](mailto:youngwlim@hotmail.com).

This document was produced by using OpenOffice.

# Based on

---

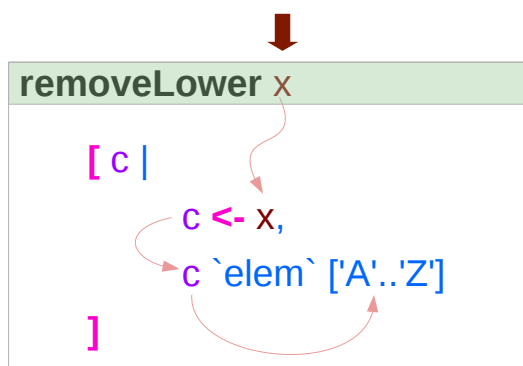
[Haskell in 5 steps](https://wiki.haskell.org/Haskell_in_5_steps)

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

# 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**

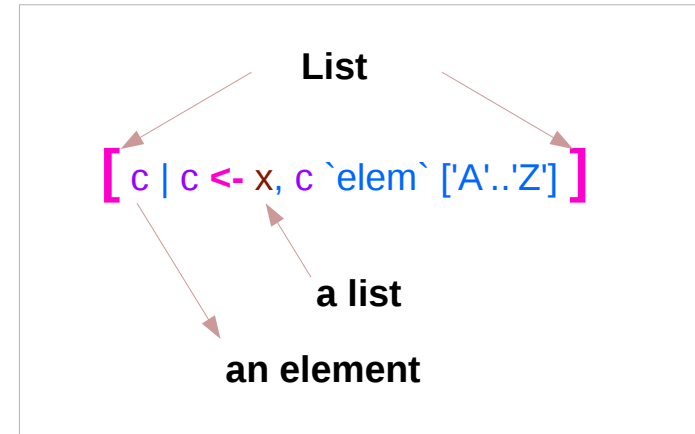
matching from the **elements** of the **list** `x`

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](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

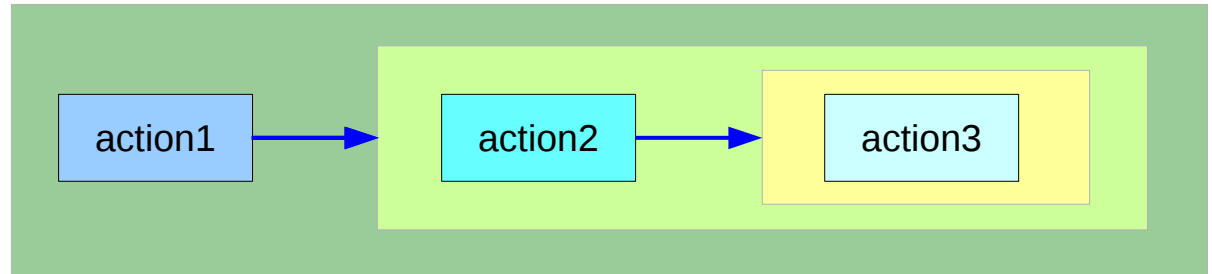
```
putStr "Hello" >>  
putStr " " >>  
putStr "world!" >>  
putStr "\n"
```

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

[https://en.wikibooks.org/wiki/Haskell/do\\_notation](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](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](https://en.wikibooks.org/wiki/Haskell/do_notation)

# Chaining `>>=` and `do` notations

`->`

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

```
action1
```

```
>>=
```

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

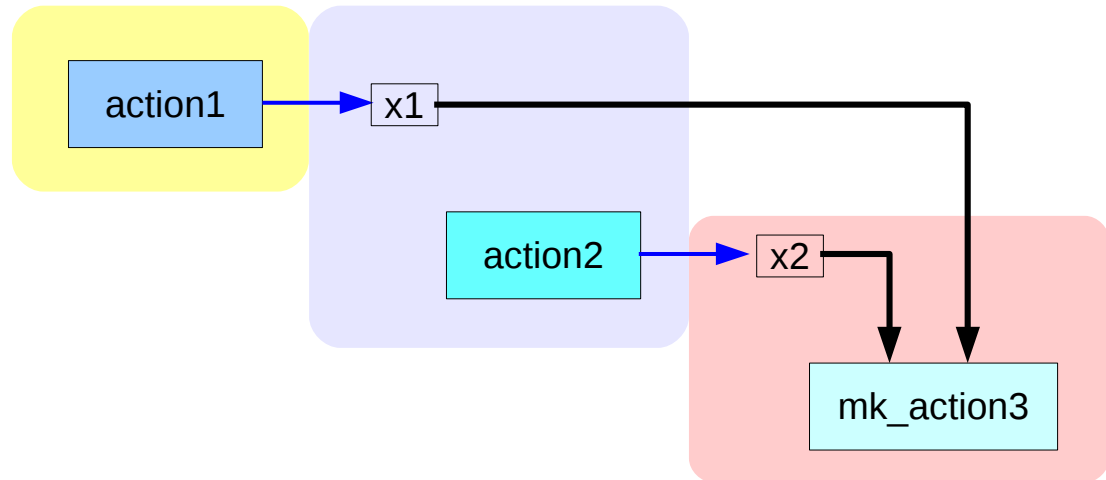
```
>>=
```

```
(\ 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](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](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](https://en.wikibooks.org/wiki/Haskell/do_notation)

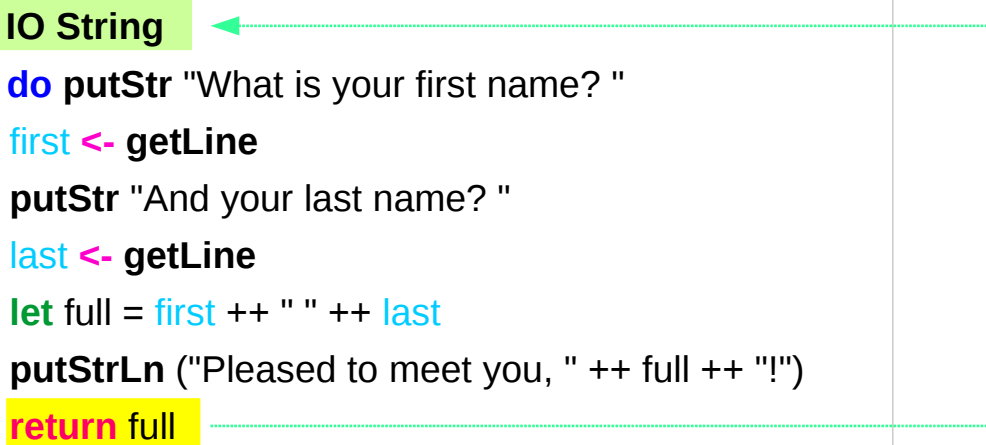
```
do { x1 <- action1
    ; x2 <- action2
    ; mk_action3 x1 x2 }
```

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 right side of the 'return full' line in the code block above, moves right, then up, then left, ending with an arrowhead pointing to the 'IO String' type signature in the line above.

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

[https://en.wikibooks.org/wiki/Haskell/do\\_notation](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 ++ "!") }
```

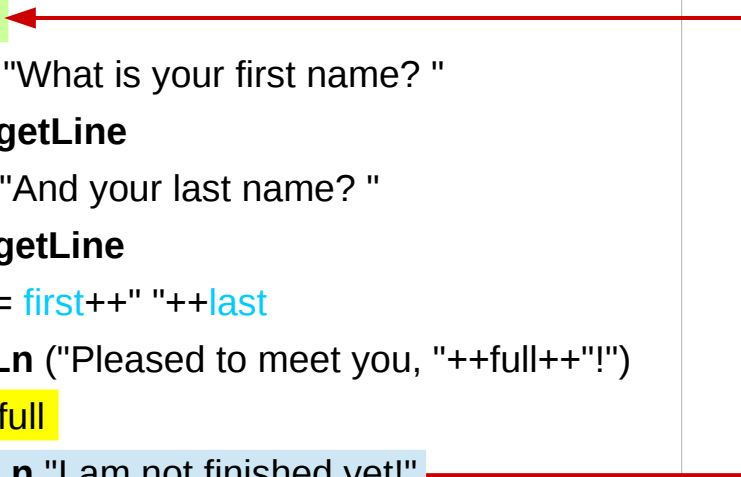
no return statement  
returns **empty IO** monad

[https://en.wikibooks.org/wiki/Haskell/do\\_notation](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](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

```
type SBTTree = BTree String
```

```
type BBTTree = BTree Bool
```

## 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

```
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>

# 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")  
  ]
```

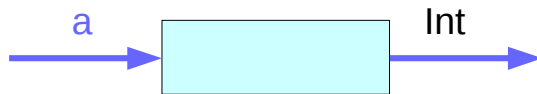
<http://learnyouahaskell.com/making-our-own-types-and-typeclasses>

# Type Synonyms for Functions

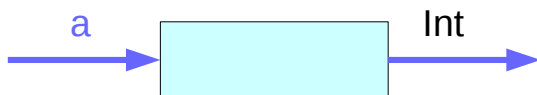
```
type Bag a = a -> Int
```

```
data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

**a -> Int**



**Bag a**



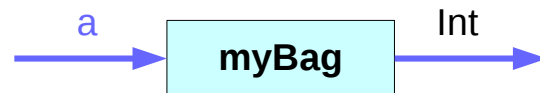
<https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions>

# Type Synonyms for Functions

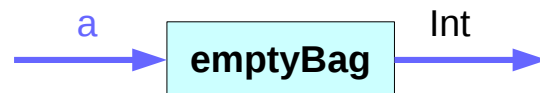
```
type Bag a = a -> Int
```

```
data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

`myBag :: Bag Gems`



`emptyBag :: Bag Gems`



<https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions>

# Type Synonyms for Functions

```
type Bag a = a -> Int
```

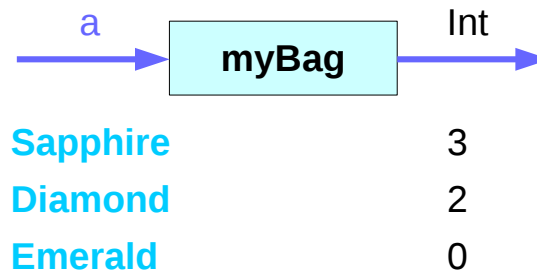
```
data Gems = Sapphire | Emerald | Diamond deriving (Show)
```

```
myBag :: Bag Gems
```

```
myBag Sapphire = 3
```

```
myBag Diamond = 2
```

```
myBag Emerald = 0
```

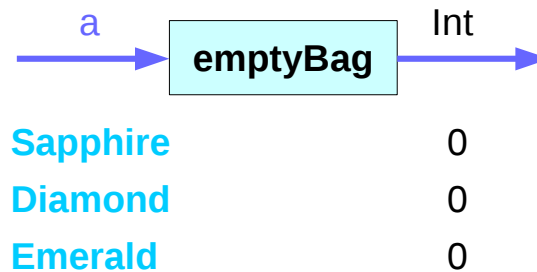


```
emptyBag :: Bag Gems
```

```
emptyBag Sapphire = 0
```

```
emptyBag Diamond = 0
```

```
emptyBag Emerald = 0
```



<https://stackoverflow.com/questions/14166641/haskell-type-synonyms-for-functions>

# 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](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](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](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>