

Monad (1A)

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

Generator

```
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**

`c` is a **pattern**

- to be matched from the elements of the list `x`

- to be successively bound to the elements of the input list `x`

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

is a **predicate** which is applied to each successive binding of `c` inside the comprehension
an element of the input only appears in the output list if it passes this predicate.

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

Assignment in Haskell

Assignment in Haskell : declaration with initialization:

You declare a variable;

Haskell doesn't allow uninitialized variables,

so an initial value must be supplied in the declaration

There's no mutation, so the value given in the declaration
will be the only value for that variable throughout its scope.

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

Assignment in Haskell

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

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

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

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

```
x >>= return
```

<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

Monad Class Function $>>=$ & $>>$

both $>>=$ and $>>$ are functions from the Monad class.

Monad Sequencing Operator with value passing

$>>=$ **passes** the result of the expression on the left
as an argument to the expression on the right,
in a way that respects the context the argument and function use

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

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

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**.

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

Color is a *type*
RGB is not a value but a *function* taking three Ints and *returning a value*

```
RGB :: Int -> Int -> Int -> Colour
```

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 (1)

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**

Consider a binary tree to store **Bool**

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

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

Type Constructor (2)

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 tUype.

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

Monad Definition

A **monad** is defined by

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

The function and operator are methods of the Monad type class and have types

```
return :: a -> m a
```

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

and are required to obey three laws

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

Maybe Monad

the Maybe monad.

The type constructor is $m = \text{Maybe}$,

$\text{return} :: a \rightarrow \text{Maybe } a$

$\text{return } x = \text{Just } x$

$(\gg=) :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b$

$m \gg= g = \text{case } m \text{ of}$
 Nothing \rightarrow Nothing
 Just $x \rightarrow g x$

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

Monad Class Function $>>=$ & $>>$

Maybe is the monad
return brings a value into it
by wrapping it with **Just**

$(>>=)$ takes
a value $m :: \text{Maybe } a$
a function $g :: a \rightarrow \text{Maybe } b$

if m is **Nothing**,
there is nothing to do and the result is **Nothing**.
Otherwise, in the **Just x** case,
the underlying value x is wrapped in **Just**
 g is applied to x , to give a **Maybe b** result.

Note that this result *may* or *may not* be **Nothing**,
depending on what g does to x .

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
  Nothing -> Nothing
  Just x   -> g x
```

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

Monad Class Function $\gg=$ & \gg

```
(\gg=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m \gg= g = case m of
  Nothing -> Nothing
  Just x   -> g x
```

if there is an underlying value of type **a** in **m**,
we apply **g** to it, which brings the underlying value back into the **Maybe** monad.

The key first step to understand how `return` and `(\gg=)` work is tracking
which values and arguments are monadic and
which ones aren't.

As in so many other cases, type signatures are our guide to the process.

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

Maybe Monad Examples

a family database that provides two functions:

```
father :: Person -> Maybe Person
```

```
mother :: Person -> Maybe Person
```

Input the name of someone's father or mother.

If some relevant information is missing in the database

Maybe returns a **Nothing** value
to indicate that the lookup failed,
rather than crashing the program.

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

Maybe Monad Examples

functions to query various grandparents.

the following function looks up the maternal grandfather (the father of one's mother):

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
  case mother p of
    Nothing -> Nothing
    Just mom -> father mom
```

```
maternalGrandfather p = mother p >>= father
```

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

Maybe Monad Examples

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

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

```
bothGrandfathers p =
```

```
  case father p of
```

```
    Nothing -> Nothing
```

```
    Just dad ->
```

```
      case father dad of
```

```
        Nothing -> Nothing
```

```
        Just gf1 ->          -- found first grandfather
```

```
      case mother p of
```

```
        Nothing -> Nothing
```

```
        Just mom ->
```

```
          case father mom of
```

```
            Nothing -> Nothing
```

```
            Just gf2 ->      -- found second grandfather
```

```
            Just (gf1, gf2)
```

```
bothGrandfathers p =
```

```
  father p >>=
```

```
    (\dad -> father dad >>=
```

```
      (\gf1 -> mother p >>= -- gf1 is only used in the final return
```

```
        (\mom -> father mom >>=
```

```
          (\gf2 -> return (gf1, gf2) ))))
```

Maybe Monad Examples

```
data Maybe a = Just a  
             | Nothing
```

a type definition: **Maybe a**
a parameter of a type variable a,

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

Maybe Monad Examples

```
data Maybe a = Just a
              | Nothing
```

two constructors: **Just a** and **Nothing**

a value of **Maybe a** type must be constructed via either **Just** or **Nothing**
there are no other (non-error) possibilities.

Nothing has no parameter type,
names a constant value that is a member of type **Maybe a** for all types **a**.

Just constructor has a type parameter,
acts like a function from type **a** to **Maybe a**,
i.e. it has the type **a -> Maybe a**

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

Maybe Monad Examples

the (data) constructors of a type build a value of that type;

when using that value,
pattern matching can be applied

- Unlike functions, constructors can be used in pattern binding expressions
- **case analysis** of values that belong to types with **more than one constructor**.
- need to provide **a pattern** for each constructor

case maybeVal **of**

Nothing	-> "There is nothing!"
Just val	-> "There is a value, and it is " ++ (show val)

a pattern for each
constructor

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

Maybe

Maybe : Algebraic Data Type (ADT)

Widely used because it effectively extends a type

Integer into a new context in which it has an extra value (Nothing)
that represents a lack of value

Check for that extra value before accessing the possible Integer

Good for debugging

Many other languages have this sort of "no-value" value via NULL references.

The Haskell Maybe type handle this no-value more effectively.

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

Maybe as a functor

Functor type class:

- transforming one type to another
- transforming operations of one type to those of another

Maybe has a useful instance of a **functor** type class

Functor provides **fmap** method

maps functions of the *base type* (such as *Integer*)
to *functions* of the *lifted type* (such as *Maybe Integer*).

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

Maybe as a functor

A *function* `f` transformed with `fmap`
can work on a Maybe value

case maybeVal of

```
Nothing -> Nothing    -- there is nothing, so just return Nothing
Just val -> Just (f val) -- there is a value, so apply the function to it
```

```
father :: Person -> Maybe Person
mother :: Person -> Maybe Person
```

```
f :: Int -> Int
fmap f :: Maybe Integer -> Maybe Integer
```

a **Maybe Integer** value: `m_x`

```
fmap f m_x
```

In fact, you could apply a whole chain of

lifted Integer -> Integer functions to **Maybe Integer** values

and only have to worry about explicitly checking for **Nothing** once when you're finished.
<https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell>

Maybe as a functor

In fact, you could apply a whole chain of **lifted Integer -> Integer** functions to **Maybe Integer** values and only have to worry about explicitly checking for **Nothing** once when you're finished.

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

Maybe as a monad

the type signature **IO a** looks remarkably similar to **Maybe a**.

- IO doesn't expose its constructors
- only be "run" by the Haskell runtime system
- a Functor
- a Monad

a Monad is just a special kind of Functor with some extra features

Monads like **IO** *map* types to new types
that represent "computations that result in values"

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."

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

Maybe as a monad

Maybe is also a Monad
represents "computations that could fail to return a value"

Just like with the fmap example,
this lets you do a whole bunch of computations
without having to explicitly check for errors after each step.

And in fact, the way the Monad instance is constructed,
a computation on Maybe values stops as soon as a Nothing is encountered,

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

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

Monad – List Comprehension Examples

```
[x*2 | x<-[1..10], odd x]
```

```
do
  x <- [1..10]
  if odd x
    then [x*2]
    else []
```

```
[1..10] >>= (\x -> if odd x then [x*2] else [])
```

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

Monad – I/O Examples

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

<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 – 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>

References

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