Monad (1A)

Young Won Lim
6/14/17

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

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 = **Maybe**,

```
return :: a -> Maybe a

return x  = Just 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 >>= & >>

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

(**>>=**) takes
a value m :: **Maybe** a
a function g :: a -> **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 >>= & >>

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

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
```

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

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