Monad P1 : Several Monad Types (4A)

Young Won Lim
10/16/19

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

What is a monad

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

Pure functional programs

Why do you need a monad?

Pure functional languages are different from **imperative languages** like C, or Java in that,

- a pure functional program is not necessarily executed in a specific order, one step at a time.
- A Haskell program is more akin to a mathematical function, in which you may solve the "equation" in any number of potential orders.
- it eliminates the possibility of certain kinds of bugs (data dependency, and those related to things like **state**)

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

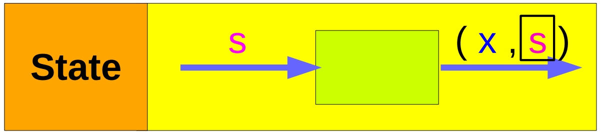
Execution orders

However, certain problems like
console programming, and **file i/o**,
need things to happen in a particular order, or
need to maintain **state**.

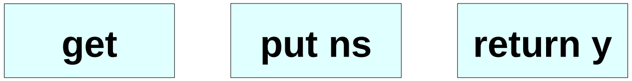
One way to deal with this problem is to create

- a kind of **object** that represents the **state** of a computation, and
- a set of **functions** that take a **state object** as input, and return *a new modified* **state object**.

state object



a set of functions



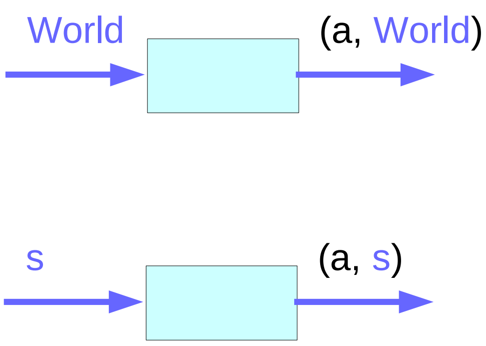
https://stackoverflow.com/questions/44965/what-is-a-monad

A hypothetical state value

a hypothetical **state value** can

represent the **state** of a console screen.

- exact value is not important,
- an array of byte length ascii characters
  that represents what is currently *visible* on the screen
- an array that represents
  the last line of *input* entered by the user, in pseudocode.
- create some **functions** that take console **state**,
  modify it, and return a new console **state**.



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

Nesting style for a particular execution order

consolestate MyConsole = new consolestate;

for a pure functional manner, a possible choice is to **nest** a lot of function calls inside each other.

```
consolestate FinalConsole =
    print( input( print( myconsole, "Hello, what's your name?" ) ),"hello, %inputbuffer%!" );
```

- this programming keeps the pure functional style
- while forcing changes to the console to happen in a particular order.

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

No-nesting style

- more than just a few operations at a time
- more than nesting functions
- a more convenient way to write it

```
consolestate FinalConsole = myconsole :
                 print("Hello, what's your name?") :
                 input() :
                 print("hello, %inputbuffer%!");
```

- **more than sequencing**
- **flexible function combining**

: (cons operator)

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

Monad, bind and lift operators

If you have a **type** (such as consolestate)

that you want to define along with a few **functions**

that are designed to operate on that type,

you can pack the **type** and related **function definitions**

into a **monad** by defining an **operator** like :

(**bind operator**) automatically feeds **return** values on its left,

into **function** parameters on its right,

(**lift operator**) turns normal functions, into functions

that work with that specific kind of **bind operator**.

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

```
liftM   :: a -> b -> m a -> m b
      f :: a -> b
liftM f :: m a -> m b
```

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

Bind operator >>=

```
putStrLn "What is your name?"
>>= (\_ -> getLine)
>>= (\name -> putStrLn ("Welcome, " ++ name ++ "!"))
```

The **>>=** operator takes a value (on the left side)

and combines it with a function (on the right side),

to produce a new value.

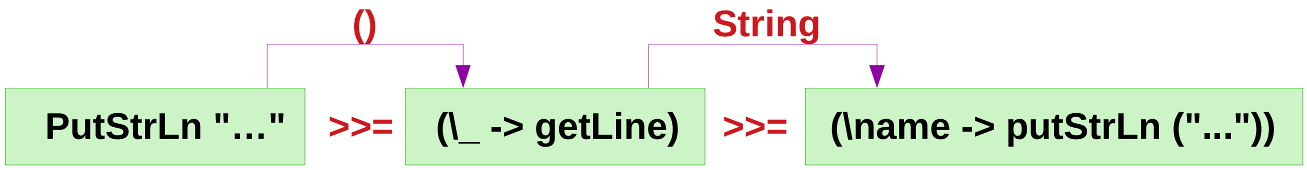
This new value is then taken by the next **>>=** operator

and again combined with a function to produce a new value.

>>= can be viewed as a mini-evaluator.

putStrLn :: String -> IO ()

getLine :: IO String



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

Monadic operation

a **monad**

- is a **parameterized type**
- is an **instance** of the **Monad type class**
- defines **>>=** along with a few other operators.
- just a **type** for which the **>>=** operation is defined.

In itself **>>=** is just a cumbersome way of **chaining functions**,
but with the presence of the **do-notation**
which hides the "**plumbing**",
the **monadic operations** turns out to be a very nice
and useful **abstraction**, useful many places in the language,
and useful for creating your own mini-languages in the language.

```
tick :: State Int Int
tick = do   n <- get
            put (n+1)
            return n
```

```
test  = do  tick
            tick
```

```
test  = tick  >> tick
```

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

>>= : an overloaded operator

Note that **>>=** is overloaded for different types,

so every monad has its own implementation of >>=.

(All the operations in the chain have to be of the type

of the same monad though, otherwise the >>= operator won't work.)

The simplest possible implementation of **>>=** just takes the value

on the left and applies it to the function on the right and

returns the result, but as said before,

what makes the whole pattern *useful* is

when there is something extra going on

in the monad's implementation of >>=.

every monad must implement >>=

only the same monad can be used in a chain

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

M :: m a
F :: a -> m b
G :: b -> m c
H :: c -> m d

M >>= F >>= G >>= H

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

Combining functions

in a **do-block**, every operation (basically every line) is wrapped in a separate anonymous function.

these functions are then combined using the **bind** operator

the **bind** operation combines **functions**,

it can execute them as it sees *fit*:

- sequentially,
- multiple times,
- in reverse,
- discard some,
- execute some on a separate thread and so on.

M >>= F >>= G

m a >>= a -> m b >>= b -> m c

```
M = do
      imperative
      codes …
F = do
      imperative
      codes …
G = do
      imperative
      codes …
```

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

Various Monad applications (1)

1) The **Failure Monad**:

If each step returns a success/failure indicator,

bind can execute the next step only if the previous one succeeded.

a failing step can abort the whole sequence "automatically",

without any conditional testing from you.

2) The **Error Monad** or **Exception Monad:**

Extending the Failure Monad, you can implement **exceptions**

By your own definition (not being a language feature),

you can customize how they work.

(e.g., can ignore the first two exceptions and

abort when a third exception is thrown.)

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

Various Monad applications (2)

3) The **List Monad:**

each step returns multiple results, and the bind function iterates over them, feeding each one into the next step

No need to write loops all over the place when dealing with multiple results.

4) The **Reader Monad**

As well as passing a result to the next step, the bind function pass extra data around as well

This extra data now doesn't appear in your source code, but it can be still accessed from anywhere, without a manual passing

environment

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

Various Monad applications (3)

5) The **State Monad** and the **Writer Monad**

the extra data can be replaced.

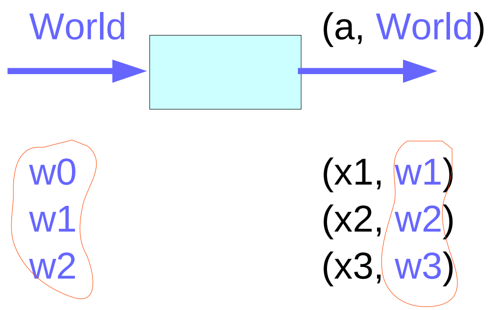
this allows you to simulate **destructive updates**

without actually doing destructive updates

you can trivially do things that would be impossible

with real destructive updates.

- **undo**
- **revert**
- **pause**
- **resume**



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

Various Monad applications (4)

for example, you can **undo** the last update,

or **revert** to an older version.

You can make a monad where calculations can be paused,

so you can **pause** your program,

go in and tinker with internal state data,

and then **resume** it.

You can implement **continuations** as a monad.

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

Various Monad applications (5)

6) The **Writer Monad**

computations produce a stream of data **log**

in addition to the computed values. **value**

It is often desirable for a computation

to generate **output** on the side.

Logging and **tracing** are the most common examples

data is generated during a computation

that we want to retain

but is not the primary result of the computation.

A **Writer monad value** is a
(**computation value**, **log value**) pair.

(**value**, **log**)

https://wiki.haskell.org/All_About_Monads#The_Writer_monad

List Monad Examples

```
[x*2 | x<-[1..4], odd x]

t = do  x <- [1..4]
        if odd x then [x*2]  else []


[1..4] >>= (\x -> if odd x then [x*2] else [])
          1              [2]
          2                        [ ]
          3              [6]
          4                        [ ]
```

Monads as computation builders

the monad chains operations

in some specific, useful way.

in the **list comprehension** example:

if an operation returns a list,

then the following operations are

performed on **every item** in the list.

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

IO Monad Examples

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

name :: String

getLine :: IO String

Read a line from the standard input device

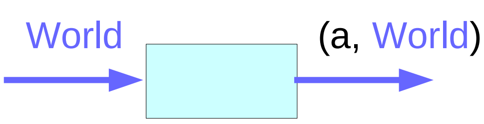
getChar :: IO Char

Read a character from the standard input device

Monads as computation builders
the monad chains operations
in some specific, useful way.

in the **IO monad** example

the operations are performed sequentially,
but a hidden variable is *passed* along,
which represents the **state** of the **world**,
allows us to write ***I/O code*** in a **pure**
functional manner.



w0	(x1, w1)
w1	(x2, w2)
w2	(x3, w3)

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

Input functions

getChar	:: IO Char	Char
getLine	:: IO String	Line
getContents	:: IO String	Contents
interact	:: (String -> String) -> IO ()	
readIO	:: Read a => String -> IO a	IO
readLn	:: Read a => IO a	Ln

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

Input functions – **getChar**, **getLine**, **getContents**

The **getChar** operation raises an **exception** on **end-of-file**;
a predicate **isEOFError** that identifies
this exception is defined in the IO library.

The **getLine** operation raises an **exception**
under the same circumstances as **hGetLine**,
defined the IO library.

The **getContents** operation returns
all user input as a single string,
which is read lazily as it is needed.

getChar :: IO Char
getLine :: IO String
getContents :: IO String

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

Input functions – **interact**

The **interact** function takes a function
of type **String->String** as its argument.
The entire input from the standard input device
is passed to this function as its argument,
and the resulting string is output on the standard output device.

interact :: (String -> String) -> IO ()

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html
https://wiki.haskell.org/Tutorials/Programming_Haskell/String_IO

interact examples

A.hs

```
main    = interact count                -- 24 characters
count s = show (length s) ++ "\n"       -- 33 characters
```

```
$ runhaskell A.hs < A.hs
57
```

The following program simply removes
all non-ASCII characters from its standard input
and echoes the result on its standard output.
(The **isAscii** function is defined in a library.)

```
main = interact (filter isAscii)
```

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

Input functions – **readIO**, **readLn**

Typically, the **read** operation from class **Read**
is used to convert the **string** to a **value**.

The **readIO** function is similar to **read**
except that it signals parse failure
to the **IO** monad instead of terminating the program.

The **readLn** function combines **getLine** and **readIO**.

readIO :: Read a => String -> IO a
readLn :: Read a => IO a

convert the **string** to a **value**

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

readIO examples

```
main = do x <- rList "[1,3,5,7]"
          y <- rInt  "5"
          print (map (y*) x)

rList :: String -> IO [Int]
rList = readIO

rInt :: String -> IO Int
rInt = readIO

Output: [5,15,25,35]


rList "[1,3,5,7]"  ➡  [1,3,5,7] :: [Int]
rInt  "5"          ➡  ”5” :: Int
```

```
main = do x <- aaa "[1,3,5,7]"
          print x

aaa :: String -> IO (Int,Int,[Int])
aaa str = do x <- readIO str
             return (sum x, product x, x)

Output: (16,105,[1,3,5,7])


aaa "[1,3,5,7]"  ➡  [16, 105, [1,3,5,7]]
     [1,3,5,7]
     sum [1,3,5,7]      ➡  16
     product [1,3,5,7]  ➡  105
```

http://zvon.org/other/haskell/Outputprelude/readIO_f.html

readLn Examples

```
main = do x <- getDouble
          y <- getDouble
         print (x+y)

getDouble :: IO Double
getDouble = readLn
```

Input: 12 (return)
Input: 4.34 (return)
Output: 16.34

```
main = do x <- getList
          print (product x)

getList :: IO [Int]
getList = readLn
```

Input: [1,2,3,4] (return)
Output: 24

```
main = do x <- aaa
          print x

aaa :: IO (Int,Int,[Int])
aaa = do x <- readLn
         return (sum x, product x, x)
```

Input: [1,3,5] (return)
Output: (9,15,[1,3,5])

http://zvon.org/other/haskell/Outputprelude/readIO_f.html

Output functions

```
putChar    :: Char -> IO ()
putStr     :: String -> IO ()
putStrLn   :: String -> IO ()          -- adds a newline
print      :: Show a => a -> IO ()
```

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

Output functions – **print**

the **print** function outputs a **value** of any **printable type**
to the standard output device.

printable types are those that are **instances** of class **Show**;

print converts **values** to **strings** for output
using the **show** operation and adds a **newline**.

For example, a program to print the first 20 integers
and their powers of 2 could be written as:

```
main = print ( [ (n, 2^n) | n <- [0..19] ] )
```

```
putChar    :: Char -> IO ()
putStr     :: String -> IO ()
putStrLn   :: String -> IO ()
print      :: Show a => a -> IO ()
```

https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

Reader Monad Example (1)

Reader r a

where **r** is some **environment** and

a is some **value** you create from that environment

let r1 = return 5 :: Reader String Int

:t r1

r1 :: Reader String Int

a Reader that takes in a **String** and returns an **Int**.

The String is the **environment** of the Reader.

https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

Reader Monad Example (2)

```
Reader r a
let r1 = return 5 :: Reader String Int
r1 :: Reader String Int

(runReader r1) "this is your environment"
5
runReader :: Reader r a -> r -> a
```

So **runReader** takes in a **Reader** and an **environment** (**r**) and returns a **value** (**a**).

https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

Reader Monad Example (3)

```
import Control.Monad.Reader

tom :: Reader String String
tom = do
    env <- ask
    return (env ++ " This is Tom.")

jerry :: Reader String String
jerry = do
  env <- ask
  return (env ++ " This is Jerry.")
```

```
tomAndJerry :: Reader String String
tomAndJerry = do
    t <- tom
    j <- jerry
    return (t ++ "\n" ++ j)

runJerryRun :: String
runJerryRun = (runReader tomAndJerry) "Who is this?"
```

```
Who is this? This is Tom.
Who is this? This is Jerry.
```

https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

Writer Monad Example (1)

```
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = writer (x, ["Got number: " ++ show x])  -- here

-- or can use a do-block to do the same thing,
-- and clearly separate the logging from the value
logNumber2 :: Int -> Writer [String] Int
logNumber2 x = do
    tell ["Got number: " ++ show x]
    return x
```

https://gist.github.com/davidallsopp/b7ecf8789efa584971c1

Writer Monad Example (2)

```
multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["multiplying " ++ show a ++ " and " ++ show b ]
    return (a*b)

main :: IO ()
main = print $ runWriter multWithLog

-- (15,["Got number: 3","Got number: 5","multiplying 3 and 5"])
```

https://gist.github.com/davidallsopp/b7ecf8789efa584971c1

Writer Monad Example (3)

```
multWithLog :: Writer [String] Int
multWithLog = do

  a <- logNumber 3

  b <- logNumber 5

  tell ["multiplying " ++ show a ++ " and " ++ show b ]
  return (a*b)
```

```
logNumber :: Int -> Writer [String] Int
logNumber x = do
  tell ["Got number: " ++ show x]
  return x
```

```
logNumber :: Int -> Writer [String] Int
logNumber x = do
  tell ["Got number: " ++ show x]
  return x
```

```
-- ( 15, ["Got number: 3", "Got number: 5", "multiplying 3 and 5"] )
```

https://gist.github.com/davidallsopp/b7ecf8789efa584971c1

Writer Monad Instance

```
instance (Monoid w) => Monad (Writer w) where

  return :: a -> Writer w a
  return a  = writer (a,mempty)

  (>>=) :: Writer w a -> (a -> Writer w b) -> Writer w b
  (writer (a,w)) >>= f =
      let (a',w') = runWriter $ f a
        in writer (a',w `mappend` w')
```

binding replaces the **computation value a** with the result **a'** of applying the **bound function** to the **previous value**

(a',w') = runWriter $ f a

and appends any **log data** of application to the existing **log data**.

w `mappend` w'

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

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))
```

The operations (**char**, **digit**, etc)
either match or not

the monad manages the **control flow**:

The operations are performed sequentially until a match fails, in which case the monad backtracks to the latest **<|>** and tries the next option.

Again, a way of chaining operations with some additional, useful semantics.

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

Parser – **char**, **digit**

char :: Stream s m Char => Char -> ParsecT s u m Char

char c parses a single character c.

Returns the parsed character (i.e. c).

semiColon = **char** ';'

digit :: Stream s m Char => ParsecT s u m Char

Parses a digit.

Returns the parsed character.

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

Parser – **many**, **many1**, **noneOf**

many :: ReadP a -> ReadP [a] **.***

Parses **zero or more** occurrences of the given parser.

many1 :: ReadP a -> ReadP [a] **.+**

Parses **one or more** occurrences of the given parser.

noneOf :: Stream s m Char => [Char] -> ParsecT s u m Char

As the dual of **oneOf**, **noneOf** cs succeeds

if the current character not in the supplied list of characters cs.

Returns the parsed character.

consonant = **noneOf** "aeiou"

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

Parser – **<|>** combinator

(<|>) :: (ParsecT s u m a) -> (ParsecT s u m a) -> (ParsecT s u m a)

This combinator implements **choice**.

The parser **p <|> q** first applies **p**.

If it succeeds, the value of **p** is returned.

If **p** fails without consuming any input, parser **q** is tried.

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

ReadP

parser generator library:

Text.ParserCombinators.ReadP.

Whenever you need to write your own parser
to consume some kind of data

a library of parser combinators

It parses all alternatives in parallel,
so it never keeps hold of the beginning of the input string,
a common source of space leaks with other parsers

https://two-wrongs.com/parser-combinators-parsing-for-haskell-beginners.html#readp

ReadP

```
import Text.ParserCombinators.ReadP
isVowel :: Char -> Bool
isVowel char =
   any (char ==) "aouei"

vowel :: ReadP Char
vowel =
   satisfy isVowel


satisfy :: (Char -> Bool) -> ReadP Char
```

the **helper function isVowel** which simply returns True for any character that is a vowel.

checking if the **argument character** is equal to any character in "aouei".

isVowel is then used in the **parser vowel**, through the **satisfy** function from the **ReadP** library

```
satisfy :: Stream s m Char => (Char -> Bool) -> ParsecT s u m Char
```

https://two-wrongs.com/parser-combinators-parsing-for-haskell-beginners.html#readp

Combinator (1)

A function or definition with no free variables.

a pure lambda-expression that refers only to its arguments, like

\a -> a	**id**
\a -> \b -> a	**const**
\f -> \a -> \b -> f b a	**flip**

https://wiki.haskell.org/Combinator

Combinator (2)

The second meaning of "combinator" is
a more informal sense
referring to the **combinator pattern**,
a style of organizing libraries centered
around the idea of combining things.

Usually there is some **type T**,
some **functions** for constructing "**primitive**" **values** of **type T**,
and some "**combinators**" which can combine **values** of **type T**
in various ways to build up more **complex values** of **type T**.

https://wiki.haskell.org/Combinator

Parse Combinator

ParsecT **s u m a**

a parser (a monad transformer)

stream type **s**,

user state type **u**,

underlying monad **m**,

return type **a**.

type Parsec s u = ParsecT s u Identity

type Parser = Parsec String ()

This means that a function returning **Parser a**

parses from a **String** with **()** as the initial state.

https://wiki.haskell.org/Combinator

Async Monad

to run **IO operations asynchronously** and wait for their results.

wait for the **return value** of a **thread**

The basic type is **Async a**

represents an **asynchronous IO action**

that will return a **value** of type **a**,

or die with an **exception**.

An **Async** corresponds to a **thread**,

and its **ThreadId** can be obtained with **asyncThreadId**

http://hackage.haskell.org/package/async-2.2.1/docs/Control-Concurrent-Async.html#v:async

Async Monad Example

to fetch two web pages at the same time,

we could do this (assuming a suitable getURL function):

```
do a1 <- async (getURL url1)
   a2 <- async (getURL url2)
   page1 <- wait a1
   page2 <- wait a2
   ...
```

async starts the operation in a separate **thread**,

and **wait** waits for and returns the result.

If the operation throws an **exception**,

then that **exception** is re-thrown by **wait**.

safety: it is harder to accidentally forget about

exceptions thrown in child threads.

http://hackage.haskell.org/package/async-2.2.1/docs/Control-Concurrent-Async.html#v:async

Async Monad – async and wait method

async :: IO a -> IO (Async a)

Spawn an **asynchronous action** in a separate thread.

wait :: Async a -> IO a

Wait for an asynchronous action to complete, and return its value.
If the asynchronous action threw an exception,
then the exception is re-thrown by wait.

http://hackage.haskell.org/package/async-2.2.1/docs/Control-Concurrent-Async.html#v:async

Async Monad F# 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() }
```

GetResponseAsync actually waits for the response on a separate thread, while the main thread returns from the function.

The last three lines are executed on the spawned thread when the response have been received.

F# code (not Haskell)

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

Async Monad F# Exampes

```
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() }
```

In most other languages you would have to explicitly create a separate function for the lines that handle the response.

The **async monad** is able to "split" the block on its own and postpone the execution of the latter half.

F# code (not Haskell)

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

References

[1] https://stackoverflow.com/questions/44965/what-is-a-monad

[2] https://wiki.haskell.org/All_About_Monads#The_Writer_monad

[3] https://www.haskell.org/onlinereport/haskell2010/haskellch7.html

[4] https://wiki.haskell.org/Tutorials/Programming_Haskell/String_IO

[5] https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

[6] https://gist.github.com/davidallsopp/b7ecf8789efa584971c1

[7] https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

[8] https://two-wrongs.com/parser-combinators-parsing-for-haskell-beginners.html#readp

[9] https://wiki.haskell.org/Combinator

[10] http://hackage.haskell.org/package/async-2.2.1/docs/Control-Concurrent-Async.html#v:async