Maybe Monad (3C)

Young Won Lim
1/16/18

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://www.schoolofhaskell.com/user/EFulmer/currying-and-partial-application

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Young Won Lim
1/16/18

Maybe Monad

a **Monad** is just a special **Functor** with <u>extra features</u>

Monads like **Maybe**

> map <u>types</u> a to a <u>new type</u> **Maybe** a
>
> that represent "computations that result in values"

- Wrap meaningful values by **Just x**
- All meaningless values by **Nothing**

Monads like **Maybe**, the **bind** (**>>=**) operation

> passes meaningful values through **Just**, while
> **Nothing** will force the result to always be **Nothing**.

https://wiki.haskell.org/Maybe

An immediate abort

Maybe is also a **Monad**

represents "computations that could *fail* to return a value"

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

enable a whole bunch of computations
without explicit checking for errors in each step

a computation on **Maybe** values *stops*
as soon as a **Nothing** is encountered

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

Maybe Monad

The **type constructor** is **m = Maybe**

general Monad type class

```
return :: a -> Maybe a

return x  = Just x
```

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

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

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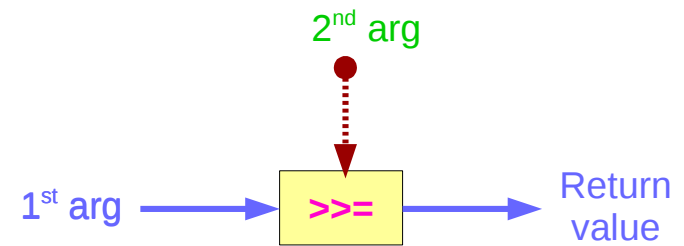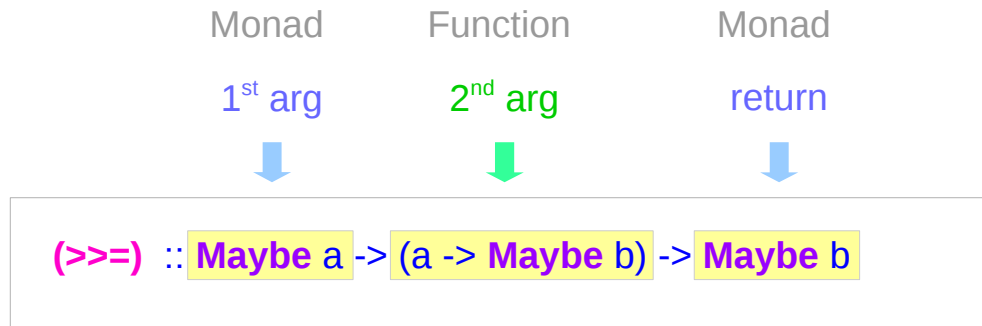
```
(>>=)  :: m a -> (a -> m b) -> m b
```

mx :: Maybe a
g :: a -> Maybe b

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

Maybe Monad

Monad   Function   Monad

1st arg   2nd arg   return

(>>=) :: **Maybe** a -> (a -> **Maybe** b) -> **Maybe** b

2nd arg

1st arg   **>>=**   Return value

mx >>= g   = case **mx** 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.

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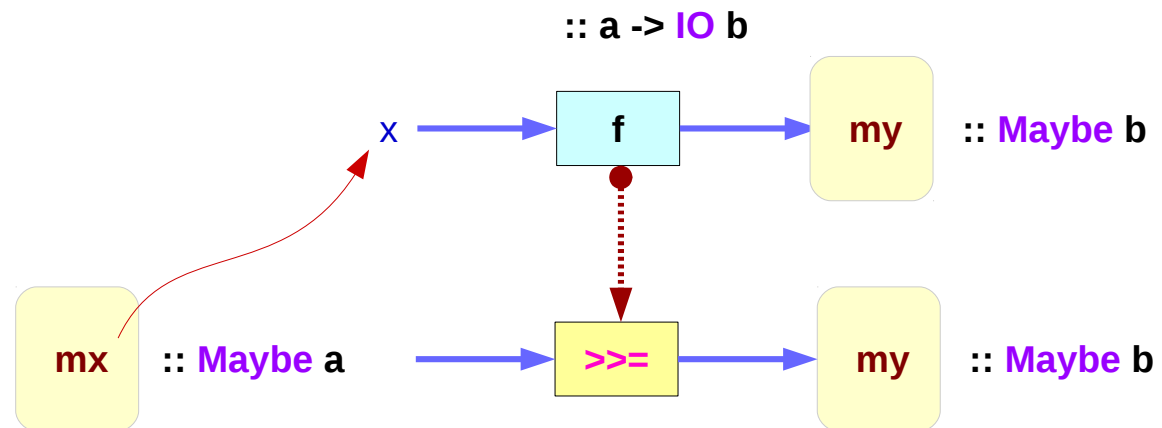
Maybe Monad

```
instance Monad Maybe where
  return x =  Just x


  mx >>= g       = case mx of
                        Nothing  -> Nothing
                        Just x     -> g x
```

ioX >>= f :: IO a -> (a -> IO b) -> IO b

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

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

Maybe Monad

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


mx >>= g        = case mx of
                        Nothing  -> Nothing
                        Just x    -> g x
```

mx >>= g   (a function with 2 args)

mx   :: Maybe a   (Maybe monad)
g   :: (a -> Maybe b)   (function)

x   :: a
g x   :: Maybe b

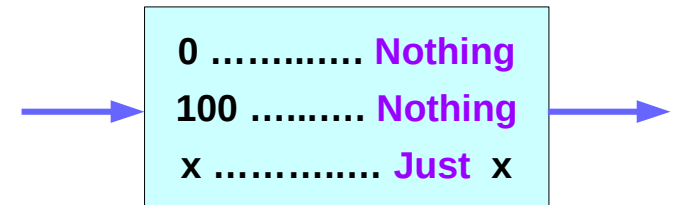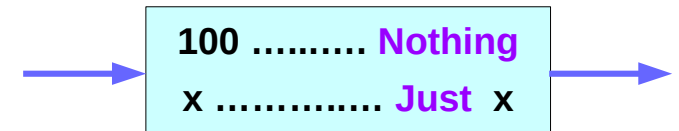https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Maybe as a Monad

f::Int -> Maybe Int
f 0 **= Nothing**
f x **= Just** x

if **x**==0 then **Nothing**
else **Just x**

0 …........ **Nothing**
x …........ **Just x**

g :: **Int -> Maybe Int**
g 100 **= Nothing**
g x   **= Just** x

if **x**==100 then **Nothing**
else **Just x**

100 …....... **Nothing**
x …........ **Just x**

0 …......... **Nothing**
100 …........ **Nothing**
x …........ **Just x**

https://wiki.haskell.org/Maybe

Maybe as a Monad

h ::Int -> **Maybe** Int   or   **h'** :: Int -> **Maybe** Int

f::Int -> **Maybe** Int

g :: Int -> **Maybe** Int

0 ……....... **Nothing**
x ……........ **Just x**

Just y …….............. **y**
Nothing ……**Nothing**

100 …...…. **Nothing**
x …….......…. **Just x**

Int   **Maybe** Int   Int   **Maybe** Int

https://wiki.haskell.org/Maybe
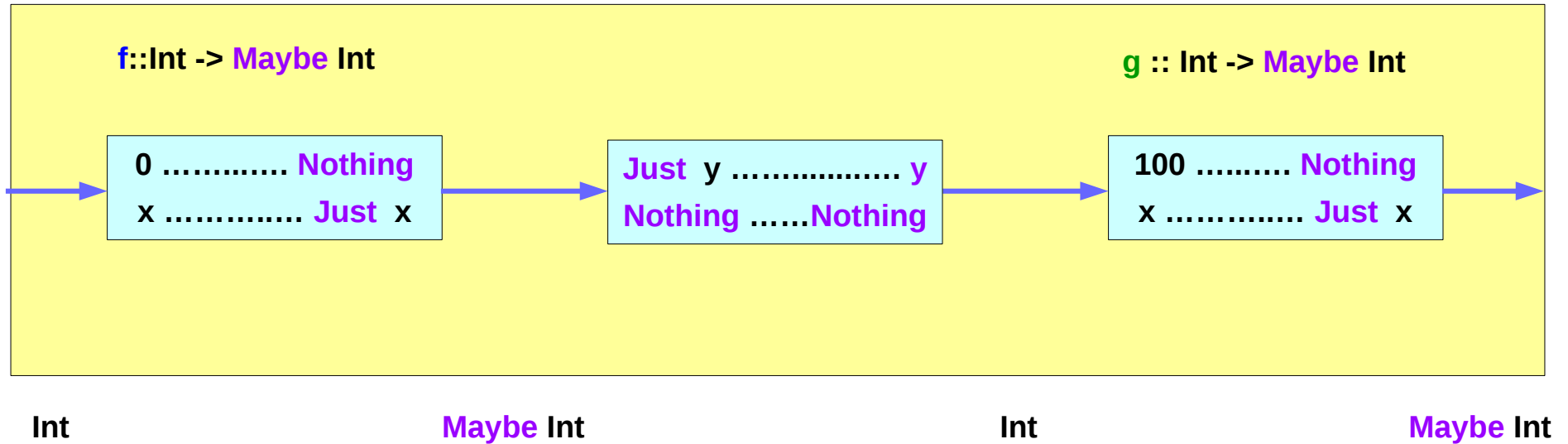
Maybe as a Monad

```
f::Int -> Maybe Int
f 0 = Nothing
f x = Just x
```
if x==0 then Nothing
else Just x

```
g :: Int -> Maybe Int
g 100 = Nothing
g x    = Just x
```
if x==100 then Nothing
else Just x

```
h ::Int -> Maybe Int
h x = case f x of
        Just n -> g n
        Nothing -> Nothing
```
if f x==Nothing then Nothing
else g n

Compact Codes

```
h' :: Int -> Maybe Int
h' x = do n <- f x
          g n
```
g (f x)

h & h' give the same results
h 0 = h' 0 = h 100 = h' 100 = Nothing;
h x = h' x = Just x

https://wiki.haskell.org/Maybe

Maybe as a Monad

```
f::Int -> Maybe Int
f 0 = Nothing
f x = Just x
```

```
g :: Int -> Maybe Int
g 100 = Nothing
g x    = Just x
```

```
h ::Int -> Maybe Int
h x = case f x of
        Just n -> g n
        Nothing -> Nothing
```

```
h' :: Int -> Maybe Int
h' x = do n <- f x
          g n
```

```
h'' ::Int -> Maybe Int
h'' x = f x >>= g
```

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

mx >>= g      = case mx of
                  Nothing -> Nothing
                  Just x  -> g x
```

https://wiki.haskell.org/Maybe

Maybe as a Library Function

When the module is imported   **import Data.Maybe**

maybe :: b->(a->b) -> Maybe a -> b

   Applies the second argument (a->b) to the third Maybe a,
   when it is Just x, otherwise returns the first argument (b).

isJust, **isNothing**
    Test the argument, returing a Bool based on the constructor.

ListToMaybe , **maybeToList**
   Convert to/from a one element or empty list.

mapMaybe
   A different way to filter a list.

https://wiki.haskell.org/Maybe

Monad Definition

```
class Monad m where
    return :: a -> m a

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

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```
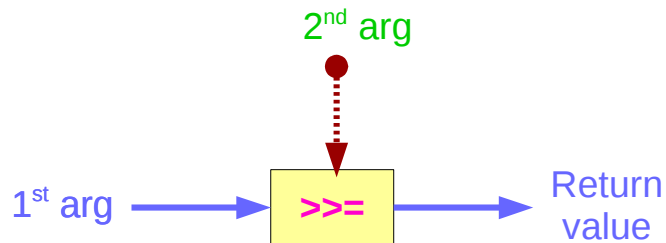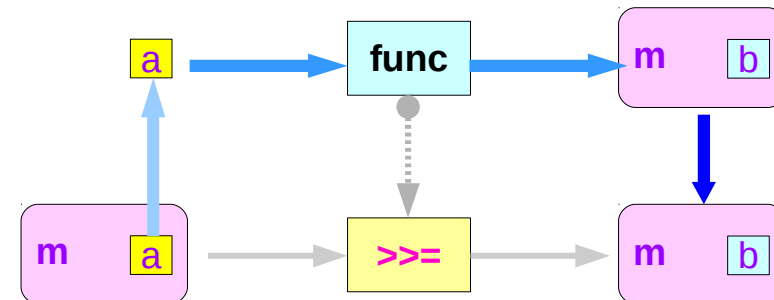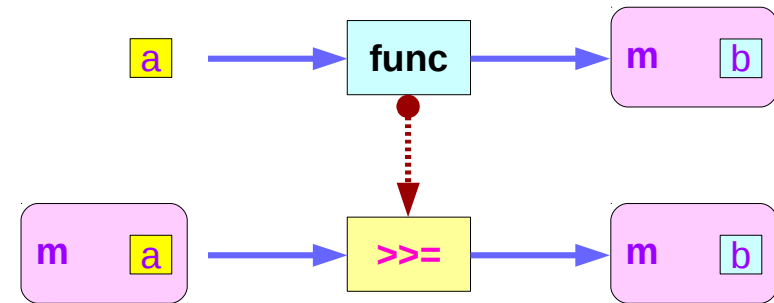
https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Monad – Bind Operation

class Monad m where
   (>>=) :: m a -> (a -> m b) -> m b

2nd arg

1st arg	Monad	**m a**
2nd arg	Function	**(a -> m b)**
return	Monad	**m b**

Maybe Monad

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

mx >>= g      = case mx of
                    Nothing  -> Nothing
                    Just x   -> g x
```

1st arg	Monad	m a
2nd arg	Function	(a -> m b)
return	Monad	m b

mx	:: **Maybe** a	(Maybe monad)
g	:: (a -> **Maybe** b)	(function)
mx >>= g		(a function with 2 args)
g	:: (a -> **Maybe** b)	
x	:: a	
g x	:: **Maybe** b	
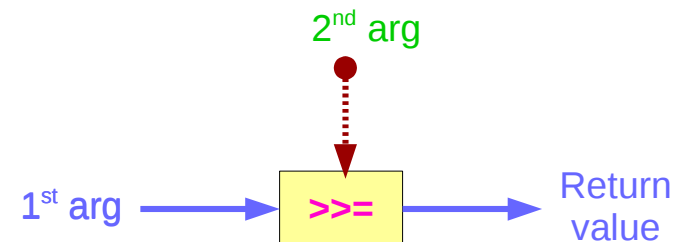
https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Monad Class Function **>>=** & **>>**

(>>=) :: **Maybe** a -> (a -> **Maybe** b) -> **Maybe** b

mx >>= g = case **mx** of
        Nothing -> **Nothing**
        Just x   -> **g** x
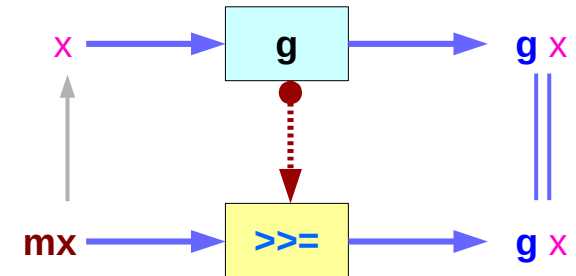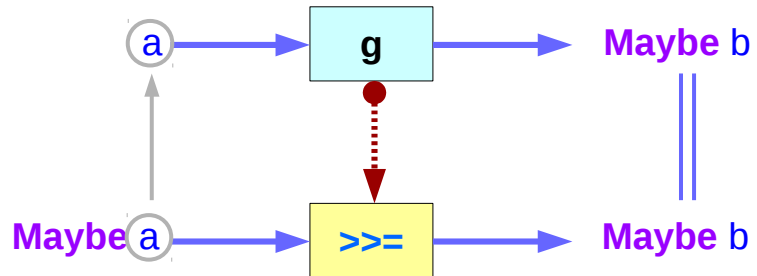
Just x   →   **g** x
Nothing   →   **Nothing**

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

Monad Class Function >>= & >>

Maybe is the monad

return brings a <u>value</u> into it
by wrapping it with **Just**

(>>=) takes
a <u>value</u>   **m** :: **Maybe** a
a <u>function</u> **g** :: a -> **Maybe** b

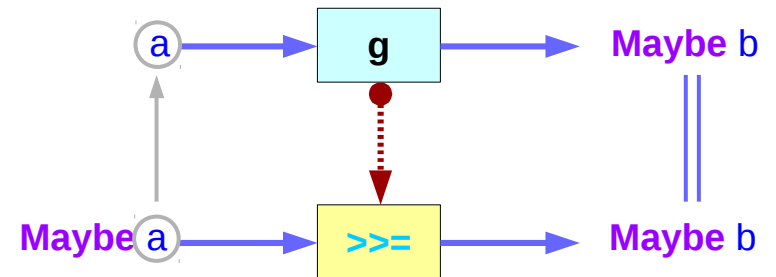if m is **Nothing**,
    there is nothing to do and the result is **Nothing**.
Otherwise, in the **Just** x case,
    the underlying value x is wrapped in **Just**
    g is applied to x, to give a **Maybe** b result.

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

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

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

Maybe Monad Examples

a family database that provides two functions:

father :: Person -> **Maybe** Person
mother :: Person -> **Maybe** Person

maternalGrandfather :: Person -> **Maybe** Person

Person ⟶ | **father** | ⟶ **Maybe** Person

Person ⟶ | **mother** | ⟶ **Maybe** Person

maternalGrandfather

Person ⟶ | **mGfather** | ⟶ **Maybe** Person
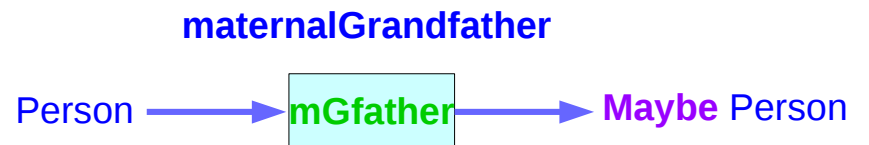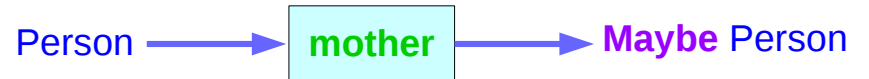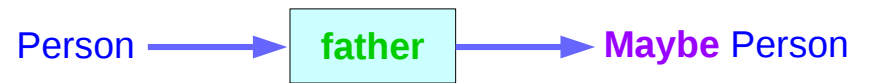
Input the name of someone's father or mother.

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

Maybe Monad Examples

Maybe Person

Database
Query information

When a query is failed (no relevant information in the database)

Maybe is useful

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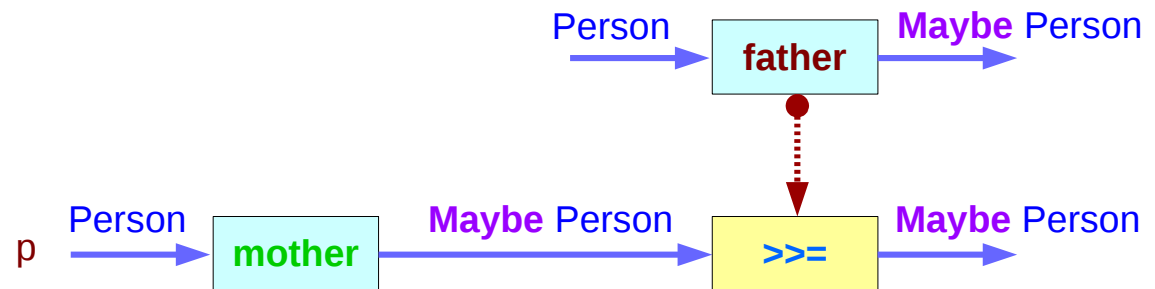to indicate that the lookup failed,
rather than crashing the program.

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

Maybe Monad Examples

```
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

```
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)
```

Maybe Monad Examples

```
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) ))))
```
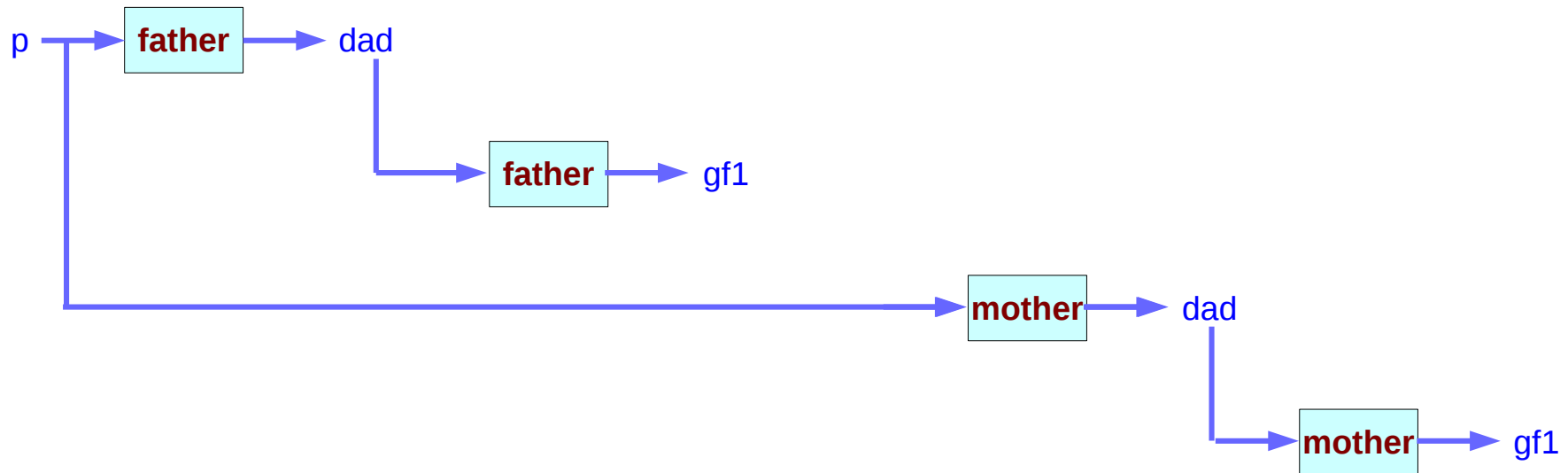
Maybe Monad Examples

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

a type definition: **Maybe** a

a parameter of a type variable a,

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.

Maybe Monad Examples

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

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

Maybe Monad Examples

the (data) <u>constructors</u> 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

Maybe Monad Examples

case maybeVal **of**

    Nothing   -> "There is nothing!"

    Just val   -> "There is a value, and it is " ++ (show val)

a pattern for each constructor

Maybe as a monad

the type signature **IO a** looks remarkably similar to **Maybe a**.
- IO doesn't expose its <u>constructors</u>
- only be "<u>run</u>" by the Haskell runtime system
- a <u>Functor</u>
- a <u>Monad</u>

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

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

lifting function
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

valueless return

Maybe is also a **Monad**

represents "computations that could *fail to return a value*"

no explicit check in each step

don't have to check explicitly for errors after each step.

immediate abort

Because of the way the Monad instance is constructed,

a computation on Maybe values *stops as soon as* a **Nothing** is encountered,

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 [])

Monad – I/O Examples

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

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))
```

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