Monad P1 : Maybe Monad (4A)

Young Won Lim
6/6/19

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

This document was produced by using LibreOffice.

Young Won Lim
6/6/19

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/

Maybe Monad – Computation

a **Monad** is just a special **Functor** with extra features

Maybe Monad

maps types a to a new type **Maybe** a

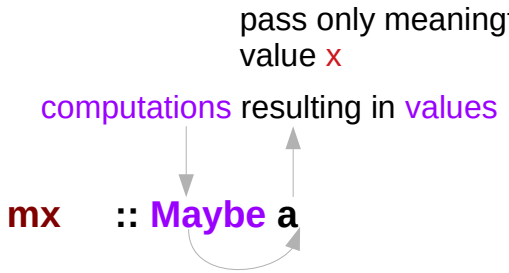
that represent "computations that result in values"

Maybe can be considered as

statements in an imperative language

to be executed

- meaningful value **x** by **Just x**
- all other meaningless values by **Nothing**



if x is meaningful value, **Just** x

otherwise, **Nothing**

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

Maybe Monad – Semantics

Maybe Monad

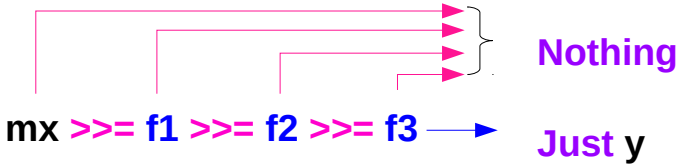
- represents “computations
  that could *fail* to return a value"
- enables an immediate abort
  by a valueless return in the middle of a computation.
- enable a whole bunch of computations
  without explicit checking for errors in each step
- a computation on **Maybe** values *stops*
  as soon as a **Nothing** is encountered

the **bind** (**>>=**) operation
passes meaningful values through **Just**, while
Nothing will force the result to always be **Nothing**.

context

semantics

effects




```
mx  ::     Maybe a
f1 :: a -> Maybe b
f2 :: b -> Maybe c
f3 :: c -> Maybe d
```

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

Maybe Monad – **Nothing** monadic value

- **meaningless value return**

  Maybe is also a **Monad** represents

  "computations that could

  fail to return a (meaningful) value"

- **no explicit check in each step**

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

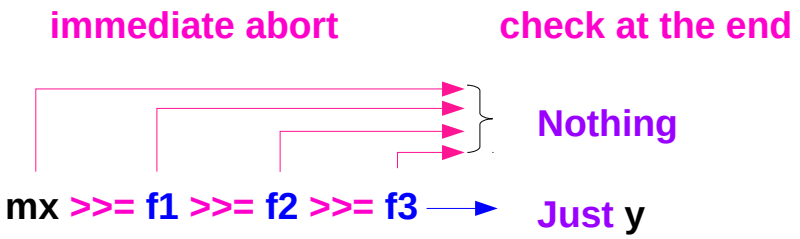
- **immediate abort**

  because of the way the **Monad** instance is constructed,

  a computation on **Maybe** values *stops*

  as soon as a **Nothing** is encountered,

return Nothing
A monadic value



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

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

Maybe Monad – an instance of a Monad class

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

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

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

   (>>) :: m a -> m b -> m b
   fail :: String -> m a
```

The type constructor is **m = Maybe**

general Monad **type class**

```
return :: a -> Maybe a

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

```
return :: a -> m a

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

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

Maybe Monad – type signatures & implementations

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

```
return :: a -> Maybe a
return x  = Just x
```

type signature

implementation

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

type signature

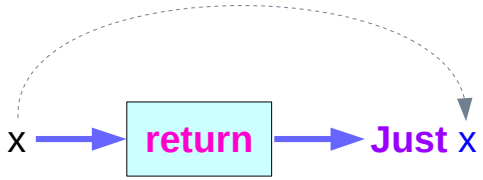
implementation

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

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

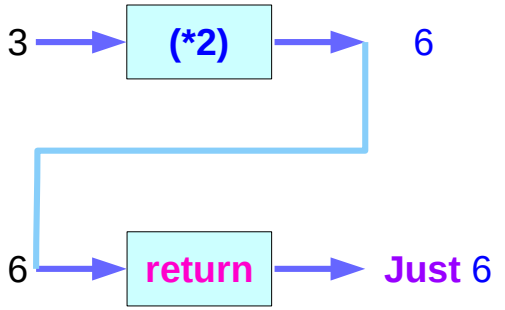
Maybe Monad – return method

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

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



only return a meaningful value x

encapsulate inside **Just**




```
return . (*2) :: a -> Maybe a
```

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

Maybe Monad – >>= method

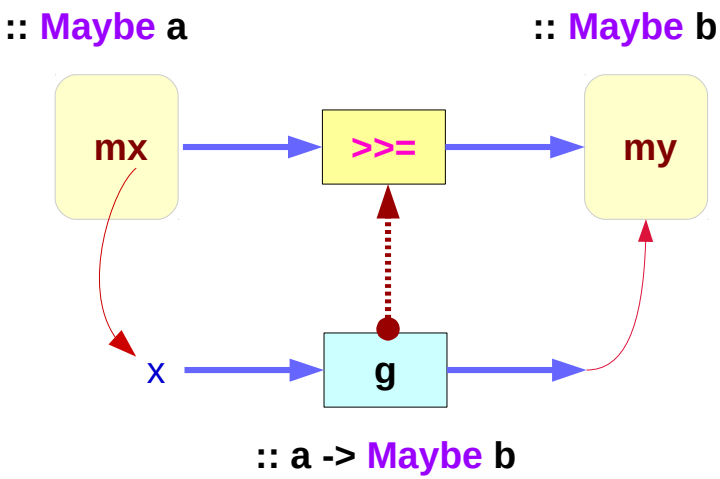
```
instance Monad Maybe where
  return x =  Just x

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

pass only meaningful value x

computations resulting in values

mx :: Maybe a

if meaningful value, **Just** x
otherwise, **Nothing**



pass only the meaningful value

Just 10 >>= g ➡ **g 10**

Nothing >>= g ➡ **Nothing**
computation stops immediately

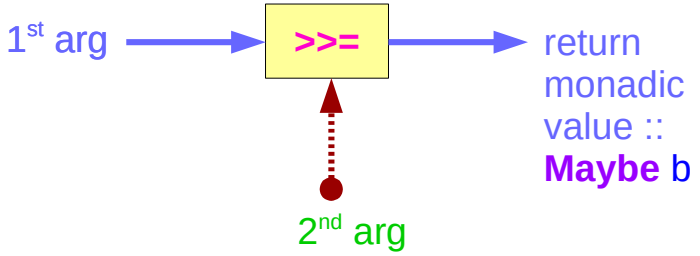
https://www.cs.hmc.edu/~adavidso/monads.pdf

Maybe Monad – >>= method type signature

Monad Function Monad

1st arg 2nd arg return

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




```
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 – types in >>= method

```
(>>=)  :: 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**

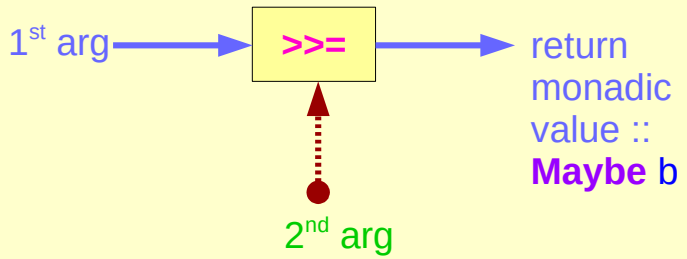
mx >>= g (a function with 2 args **mx** and **g**)

mx :: **Maybe** a (Maybe monad)

g :: (a -> **Maybe** b) (function returning Maybe monad)

x :: a

g x :: **Maybe** b (taking only the meaningful value returning a monadic value)



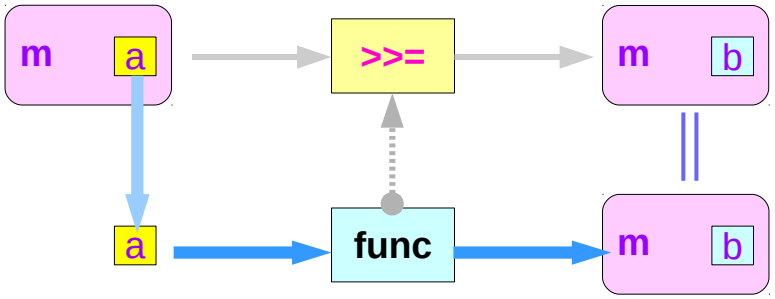
https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Maybe Monad – **Nothing** and **>>=** method

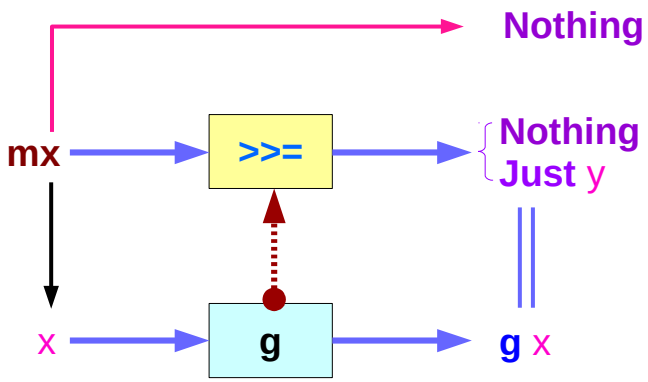
```
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```

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

m = Maybe



computation stops immediately



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

Maybe Monad – **Nothing** and the function argument

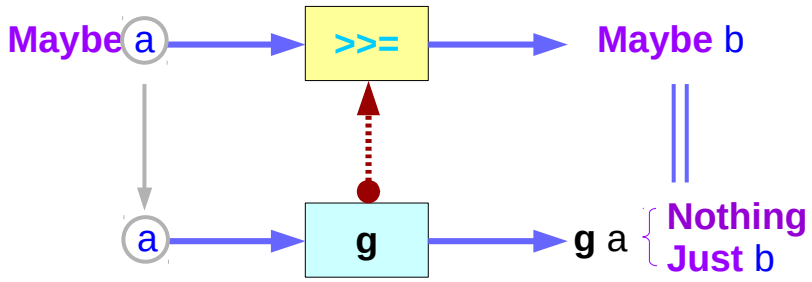
return brings a value into it
by wrapping it with **Just**

(>>=) takes
a value **mx** :: **Maybe** a
a function **g** :: a -> **Maybe** b

if mx 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

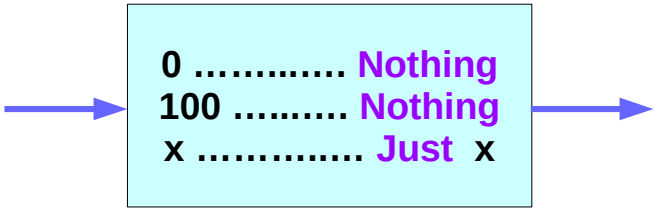
Maybe Monad Example

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

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

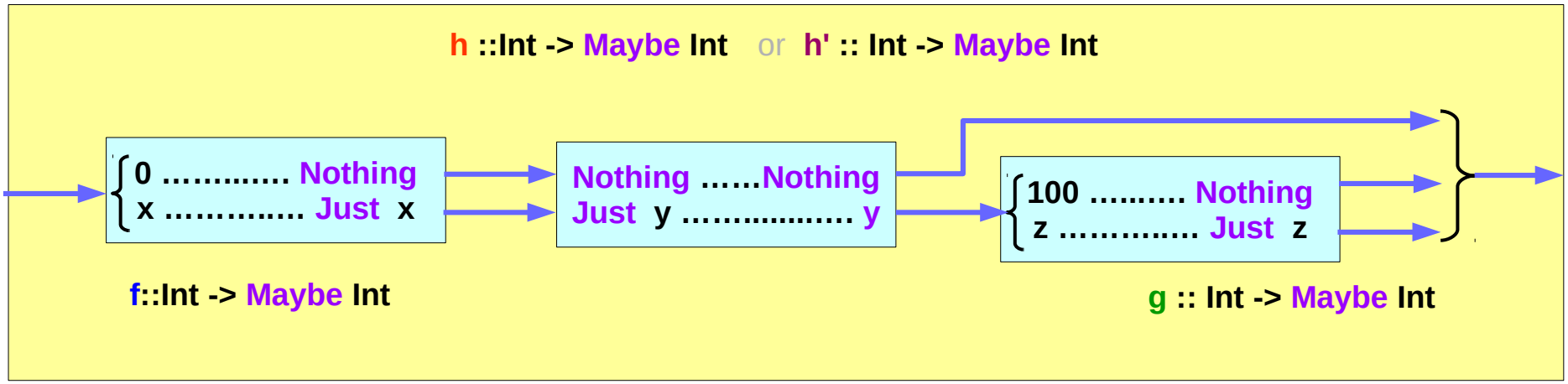

```
g :: Int -> Maybe Int
g 100 = Nothing
g x   = Just x
```

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



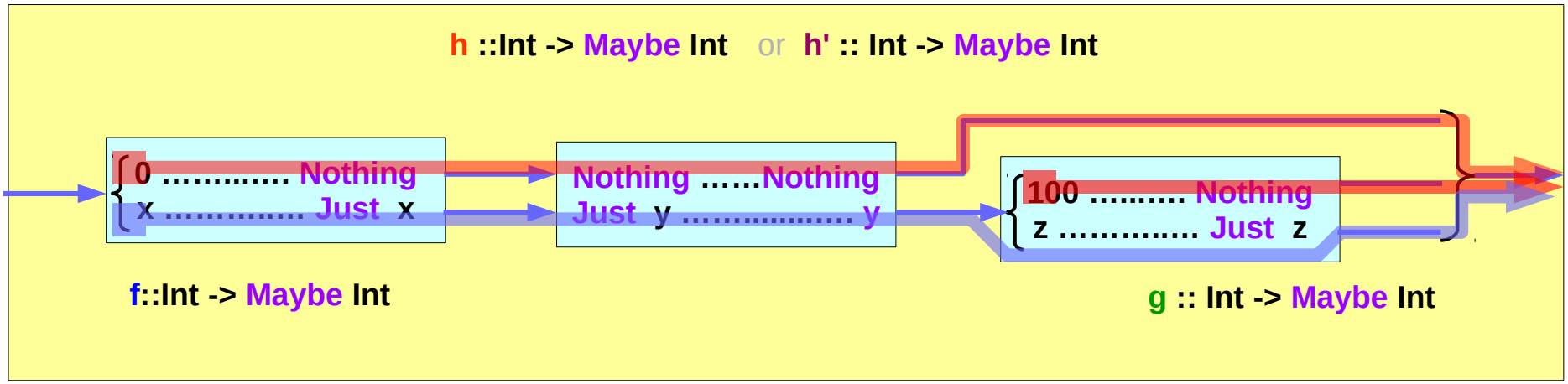
https://wiki.haskell.org/Maybe

Maybe Monad Example – composition



https://wiki.haskell.org/Maybe

Maybe Monad – immediate abort



- valueless return
- no explicit check in each step
- immediate abort

https://wiki.haskell.org/Maybe

Maybe Monad Example – implementing composition

```
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**==n then **g** n
else if **f x==Nothing** then **Nothing**

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

g (**f** x) **= g.f** x

Compact Codes

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 Monad – composition using >>=

```
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
```

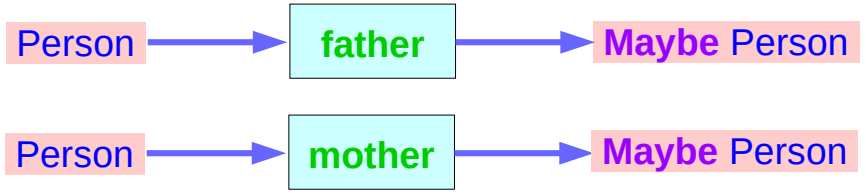
```
f :: Int -> Maybe Int

g :: Int -> Maybe Int

f x :: Maybe Int
```

https://wiki.haskell.org/Maybe

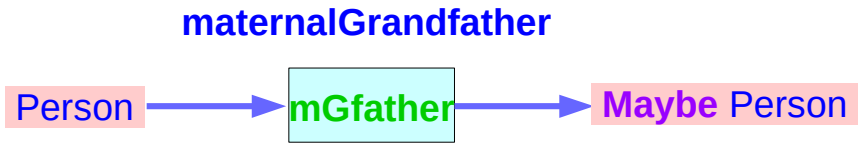
Maybe Person Examples

a family database that provides two functions:

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



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



Input the name of someone's father or mother.

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

Maybe Person Examples – Nothing

Maybe Person

- Database
- Query information

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

➡

Maybe is useful

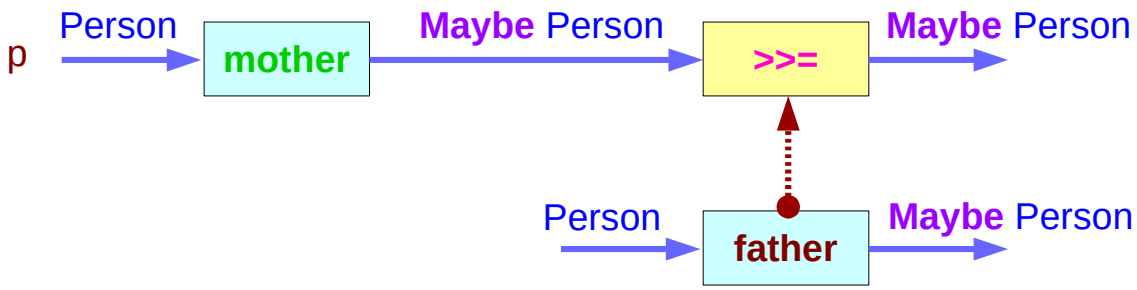
Maybe returns a **Nothing** value
to indicate that the lookup failed,
rather than crashing the program.

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

Maybe Person Examples – (1)

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

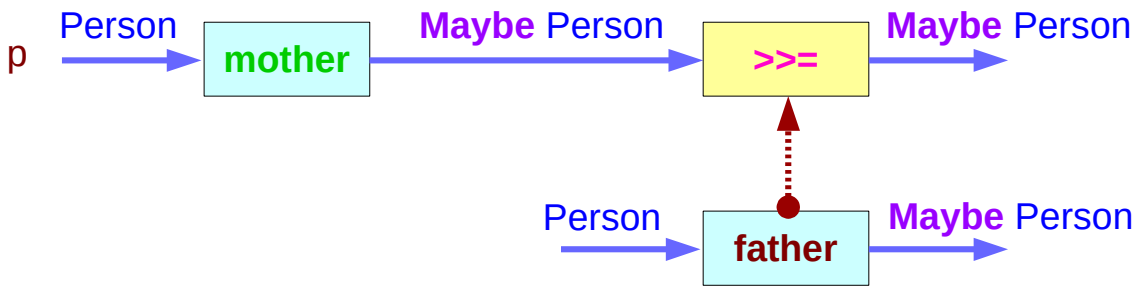
```
maternalGrandfather p = mother p >>= father
```

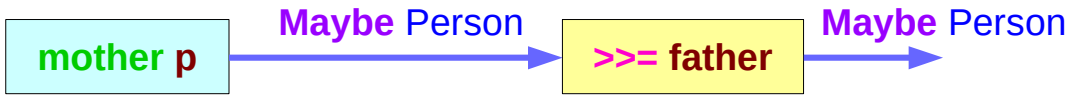


https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Maybe Person Examples – (2)

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





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

Maybe Person Examples – (3)

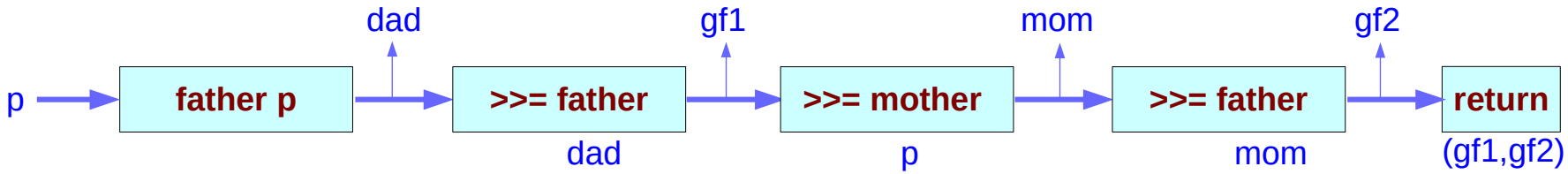
https://en.wikibooks.org/wiki/Haskell/Understanding_monads

```
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)
```

Maybe Person Examples – (4)

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

```
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 Person Examples – (5)

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

bothGrandfathers p =
  father p >>=
    (\dad -> father dad >>=
      (\gf1 -> mother p >>=
        (\mom -> father mom >>=
          (\gf2 -> return (gf1,gf2) ))))
```

```
bothGrandfathers p = do {
      dad  <- father p;
      gf1  <- father dad;
      mom  <- mother p;
      gf2  <- father mom;
      return (gf1, gf2);
    }
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Maybe as a data type definition

```
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

Two Data Constructors

```
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.

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

Just and Nothing Data Constructors

```
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

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

Pattern Matching in Data Constructors

the data constructors of a **type**
build *a* **value** of that type;

when using that **value**, **case analysis of values**
pattern matching can be applied

- Unlike functions, *constructors* can be used in *pattern binding expressions*
- **case analysis** of **values** that have more than one data constructor
- must provide **a pattern** for each constructor

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

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

Pattern Matching in Maybe Monad

```
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

References

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