Простейшая монада, обеспечивающая эффект отсутствующего значения (ошибки).

```
instance  Monad Maybe  where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just x) >>= k  =  k x
  Nothing  >>= _  =  Nothing

  (>>) :: Maybe a -> Maybe b -> Maybe b
  (Just _) >>  m  =  m
  Nothing  >>  _  =  Nothing

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

Примеры:

```haskell
type Name = String
type ParentsTable = [(Name, Name)]

fathers, mothers :: ParentsTable
fathers =
  [("Bill","John"),("Ann","John"),("John","Piter")]
mothers =
  [("Bill","Jane"),("Ann","Jane"),("John","Alice")
  ,("Jane","Dorothy"),("Alice","Mary")]

getM, getF :: Name -> Maybe Name
getM person = lookup person mothers
getF person = lookup person fathers
```

```haskell
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Ищем прабабушку Билла по материнской линии отца

```
GHCi> getF "Bill" >>= getM >>= getM
Just "Mary"
GHCi> do { f <- getF "Bill"; m <- getM f; getM m }
Just "Mary"
```

- Первая форма удобна только когда результат предыдущего действия должен передаваться непосредственно в следующее.
- В остальных случаях предпочтительна do-нотация.

```
granmas person = do
  m   <- getM person
  gmm <- getM m
  f   <- getF person
  gmf <- getM f
  return (gmm, gmf)
```

```
GHCi> granmas "Ann"
Just ("Dorothy","Alice")
GHCi> granmas "John"
Nothing
```

Хотя одна бабушка у Джона есть, но, как только результат одного действия стал `Nothing`, все дальнейшие действия игнорируются.

Монада списка представляет недетерминированное вычисление (с нулём или большим числом возможных результатов).

```
instance Monad [] where
  (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= k  =  concat (map k xs)
  return :: a -> [a]
  return x  =  [x]

instance MonadFail [] where
  fail :: String -> [a]
  fail _    =  []
```

Здесь `map k xs :: [[b]]` «уплощается» `concat`'ом.

```
GHCi> "abc" >>= replicate 4
"aaaabbbbcccc"
```

Следующие три списка — это одно и то же:

```
list1 = [ (x,y) | x <- [1,2,3], y <- [1,2], x /= y ]

list2 = do
  x <- [1,2,3]
  y <- [1,2]
  True <- return (x /= y)
  return (x,y)

list3 =
  [1,2,3]          >>= (\x ->
  [1,2]            >>= (\y ->
  return (x /= y) >>= (\r ->
  case r of True -> return (x,y)
            _    -> fail "Will be ignored :)")))
```

В монадах результат предыдущего вычисления может влиять на «структуру» последующих:

```
GHCi> do {a <- [1..3]; b <- [a..3]; return (a,b)}
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Для аппликативных функторов такое невозможно, структура результата полностью задана структурой аргументов

```
(<*>) :: f (a -> b) -> f a -> f b
```

```
GHCi> (,) <$> [1..3] <*> [3..3]
[(1,3),(2,3),(3,3)]
GHCi> (,) <$> [1..3] <*> [2..3]
[(1,2),(1,3),(2,2),(2,3),(3,2),(3,3)]
GHCi> (,) <$> [1..3] <*> [1..3]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```