

# CHAPTER 12 - A FISTFUL OF MONADS

モナドがいっぱい

発表者: 池田 伸太郎 2019年5月14日

## Functor → Applicative Functor → Monad



モナドはアプリカティブファンクターの特徴も兼ねた強化版

## 復習

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

FunctorもApplicative Functorもある文脈(箱とも言うっている)を表現する型クラス。

Eq, Ord型クラスとは異なり、1つ分の型引数を取る型コンストラクタが利用対象。

[], Maybe, IOとかはApplicative FunctorだしMonadでもある。

## 復習 - You **Maybe** remember...

```
-- Maybeの定義 in Data.Maybe embedded module  
data Maybe a = Nothing | Just a
```

```
instance Applicative Maybe where  
  pure = Just  
  Nothing <*> _ = Nothing  
  (Just f) <*> something = fmap f something
```

```
ghci> Just (+3) <*> Just 9  
Just 12  
ghci> pure (+3) <*> Just 10  
Just 13  
ghci> Nothing <*> Just "woot"  
Nothing
```

ああ、確かにアプリカティブ則が成り立っている  
ようだな、と納得することができますね。

## 復習 - リスト []

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
ghci> [( *0), (+50), (^3)] <*> [1,2,3]
[0,0,0,51,52,53,1,8,27]
ghci> [(+),(*)] <*> [1,2] <*> [3,4] # アプリカティブ・スタイル
[4,5,5,6,3,4,6,8]
```

ああ、確かに(ry

## モナドくんの願い

普通の値 $a$ を取って文脈付きの値を返す関数に、文脈付きの値 $m\ a$ を渡したい

言い換えると以下の関数が欲しい

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

関数 $>>=$ はバインド(bind)と呼ばれる。

モナドは $>>=$ をサポートするアプリカティブファクターである。

# モナドくん



```
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
```

Haskellの歴史的背景によりMonad定義にはApplicativeの型クラス制約がない。

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
  fail _ = Nothing
```

```
ghci> Just 9 >>= \x -> return (x*10)
Just 90
```



## 【綱渡りの問題】

養魚場で働くピエールが休暇を取り綱渡りに挑戦。バランス棒に鳥が止まりに来るんです。鳥たちはちょっと休んでまたどこかへ飛んでいく。

棒の左右に止まった鳥の差が3以内ならば、ピエールはバランスが取れるがそうでないならバランスを崩して落ちてしまう。



```
type Birds = Int
type Pole = (Birds, Birds)

landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right)
  | abs ((n + left) - right) < 4 = Just (left + n, right)
  | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left, right)
  | abs (left - (right + n)) < 4 = Just (left, right + n)
  | otherwise                    = Nothing
```

```
ghci> landLeft 2 (0, 0)
Just (2,0)
ghci> landRight 10 (3, 0)
Nothing
```

でもlandLeft/Right 1はPole型しかとれない!

landLeft 1 (landRight 1 (0,0))と合成して書きたいときはどうすれば良い?

>>=はMaybeを文脈付きのまま扱うことができる。

```
ghci> return (0,0) >>= landRight 1 >>= landLeft 2 >>= landRight 3
Just (2,4)
ghci> return (0,0) >>= landRight 2 >>= landRight 2 >>= landLeft 3
Nothing
```

このように文脈の値どうしを相互作用させることはアプリカティブファンクターにはできないことでありモナドならば実現できる。

もしMaybeをモナドとして扱っていなかったら...

```
routine: Maybe Pole
routine = case landLeft 1 (0, 0) of
  Nothing -> Nothing
  Just pole1 -> case landRight 4 pole1 of
    Nothing -> Nothing
    Just pole2 -> case landLeft 2 pole2 of
      Nothing -> Nothing
      Just pole3 -> case landLeft 1 pole1 of pole3
```

このような巨大で汚いコードを>>=でモナド適用の連鎖で書き直すのは、Maybeモナド布教コードの定番。

## do記法

```
routine :: Maybe Pole
routine = do
  start <- return (0, 0)
  first  <- landLeft 2 start
  second <- landRight 2 first
  landLeft 1 second
```

```
ghci> routine
Just (3,2)
```

いつ>>=を使い、いつdo記法を使うか、選択はあなた次第です。



リスト(`[]`)はモナド。リストをモナドとしての側面を使うことで、非決定性を伴うコードをきれいに読みやすくすることができる。

```
# Applicative Functorもリストを非決定性の性質をもつものとして扱う。  
ghci> (*) <$> [0,1,2] <*> [10,100,1000]  
[0,0,0,10,100,1000,20,200,2000]
```

```
# リストモナドの定義  
instance Monad [] where  
  return x = [x]  
  xs >>= f = concat (map f xs)  
  fail _ = []
```

```
ghci> [3,-4,5] >>= \x -> [x,-x]  
[3,-3,-4,4,5,-5]
```

リストからすべてのパターンを尽くして非決定的値を最終的に返す。

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n, ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
# do記法で書く
listOfTuples :: [(Int, Char)]
listOfTuples = do
  n <- [1,2]
  ch <- ['a','b']
  return (n, ch)
```

```
ghci> [(n,ch) | n <- [1,2], ch <- ['a', 'b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

実は、リスト内包表記はリストモナドの糖衣構文であった！リスト内包表記もdo記法も、内部では>>=を使った非決定性計算に変換される。

## モナド則

### 【左恒等性と右恒等性】

=> いずれも`return`が通常値をモナド値に最小限な単位で返すことを保証する法則

### 【結合法則】

=> モナド値をモナド関数らに食わせるとき、入れ子の順序は関係なく関数自体の意味のみが結果に反映されるよ

詳しくは**WEB**で！