

モナモナ言わない
モナド入門

2011.12.3

山本和彦

お品書き

- 統一理論と大統一理論
- コンテナの力の階層
- 内包表記

Q) なぜモナドが理解できないのか？

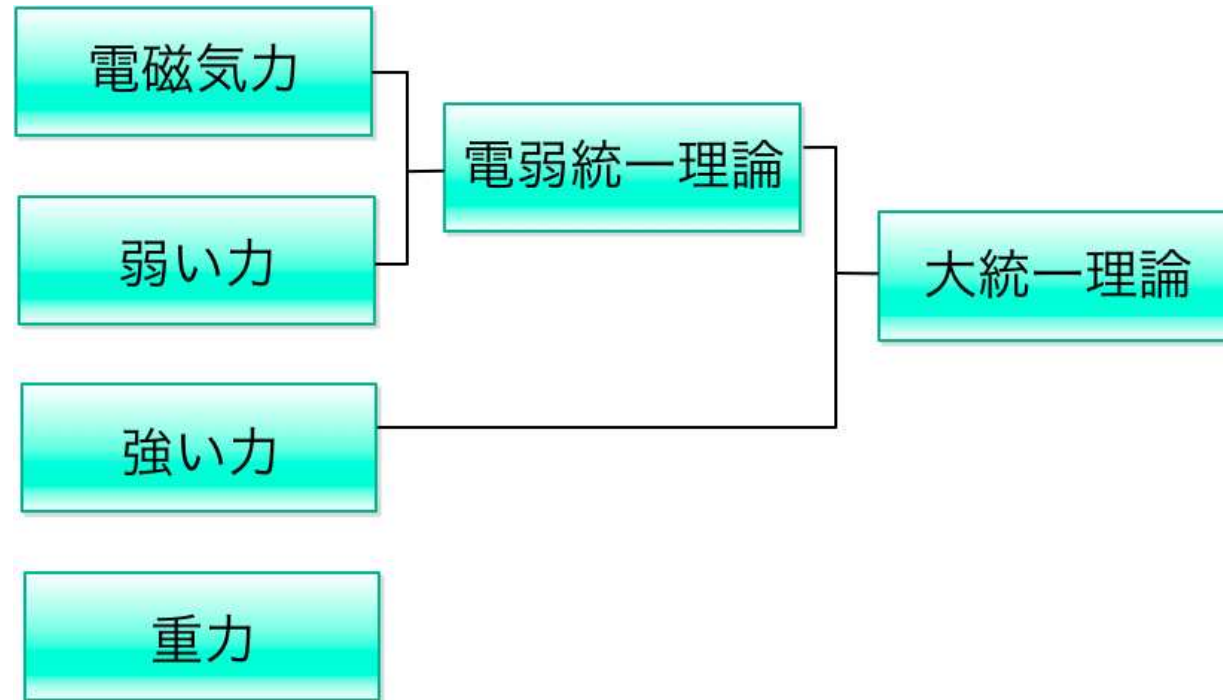
A) 抽象的な概念だから

Q) 抽象の壁を突破するには
どうすればいいのか？

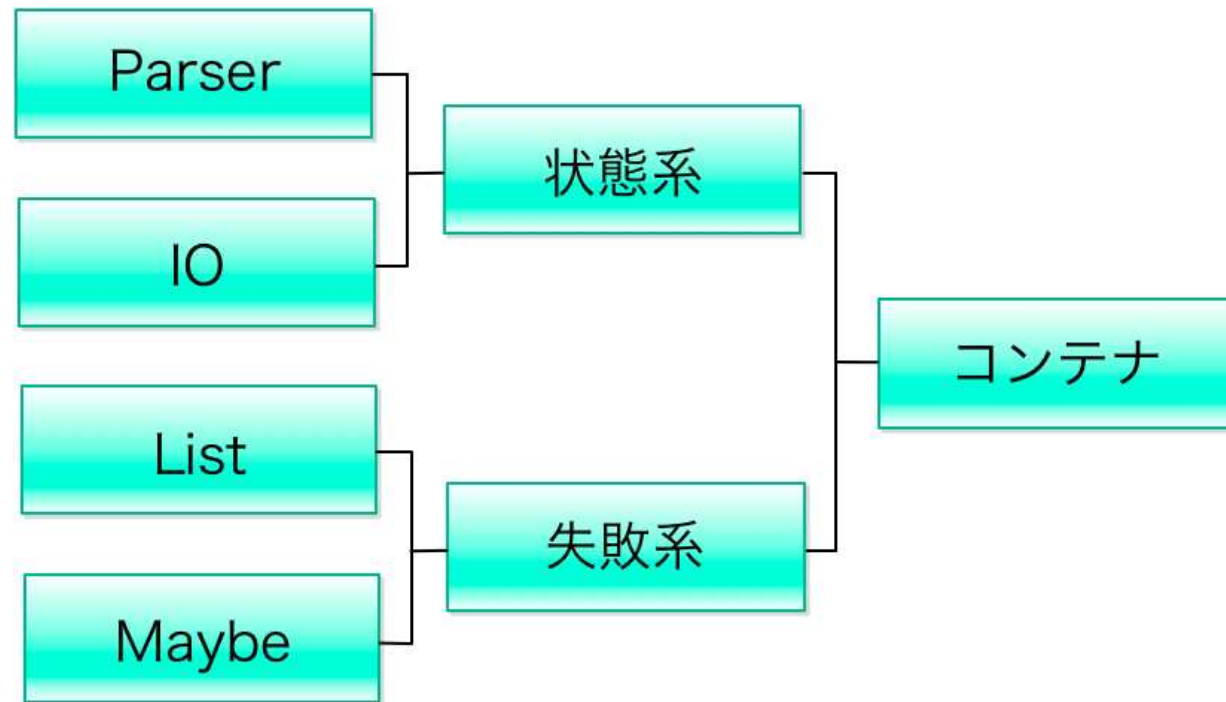
A) まず具体例を見てから
抽象化していくしかない

統一理論と大統一理論

物理学での抽象化



Haskell での抽象化



狀態系統—理論

Parser

■ データ定義

```
data Parser a = Parser (String -> [(a,String)])
```

■ Parser を生成する関数

```
pwrap :: a -> Parser a  
pwrap v    = Parser $ \inp -> [(v,inp)]
```

■ Parser を合成する関数

```
pbind :: Parser a -> (a -> Parser b) -> Parser b  
pbind p f = ...
```

■ 利用例

```
string :: String -> Parser String  
string []      = pwrap []  
string (x:xs) = char x      `pbind` \v ->  
                  string xs  `pbind` \vs ->  
                  pwrap (v:vs)
```

IO

- データ定義

```
data IO a = IO (World -> (a, World))
```

注：IO は命令ではなく命令書。すなわちオブジェクト！

- IO を生成する関数

```
iwrap :: a -> IO a  
iwrap v    = IO $ \world -> (v, world)
```

- IO を合成する関数

```
ibind :: IO a -> (a -> IO b) -> IO b  
ibind i f = ...
```

- 利用例

```
echoChar :: IO ()  
echoChar = getChar `ibind` \c -> putChar c
```

- あるいは

```
echoChar = getChar `ibind` putChar
```

Parser と IO の統一

- Parser と IO には共通点はないか？

- データ定義

```
data Parser a = Parser (String -> [(a,String)])
data IO a      = IO      (World -> (a, World))
```

- 生成する関数

```
pwrap :: a -> Parser a
iwrap :: a -> IO a
```

- 合成する関数

```
ibind :: IO a -> (a -> IO b) -> IO b
pbind :: Parser a -> (a -> Parser b) -> Parser b
```

- どちらも状態を表している！
→ 型クラスで抽象化

状態系

■ 状態を表すクラス

```
class Stateful m where
  swrap :: a -> m a
  sbind :: m a -> (a -> m b) -> m b
```

■ インスタンス

```
instance Stateful Parser where
  swrap = pwrap
  sbind = pbind

instance Stateful IO where
  swrap = iwrap
  sbind = ibind
```

状態系で統一して何が嬉しいの？

■ 構文糖衣 `do` を導入する

```
s1 `sbind` \v1 ->          do v1 <- s1
s2 `sbind` \v2 ->          v2 <- s2
...
sn `sbind` \vn ->         vn <- sn
swrap (f v1 v2 .. vn)     swrap (f v1 v2 .. vn)
```

■ `do` で書き直す

```
string []          = swrap []
string (x:xs)     = do v  <- char x
                   vs <- string xs
                   swrap (v:vs)

echoChar          = do c <- getChar
                   putChar c
```

■ 命令型言語の逐次実行を再発明しちゃった！

走らせる！

- 状態系の中身は `sbind` で合成された関数
- パターンマッチで関数を取り出して、引数(初期値)を与えないと走らない

■ `runParser`

```
runParser :: Parser a -> String -> (a, String)
runParser (Parser p) xs = p xs
```

- `data` と `getter` を同時に定義

```
data Parser a = Parser {
  runParser :: String -> (a, String)
}
```

■ `runIO`

```
runIO :: IO a -> World -> (a, World)
runIO (IO io) world = io world
```

- `IO` は抽象データ型で構成子 `IO` は公開されていない
- パターンマッチできないので関数を取り出せない
- ランタイムが `runIO main` を実行

失敗系統—理論

Maybe

- データ定義

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

- Nothing は失敗

- Just a は成功

- 連想Listの検索

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

```
lookup 0 [(1, 'a')]  
→ Nothing
```

```
lookup 1 [(1, 'a')]  
→ Just 'a'
```


おばあちゃんを探せ

- 親DB

子供をキーとして検索すると母親が出てくるDB

- おばあちゃんを探すコード

```
case lookup me db of
  Nothing -> Nothing
  Just mom -> case lookup mom db of
                Nothing -> Nothing
                Just gmom -> Just gmom
```

- どうにかならないの？

Maybe と Bool

- Bool の True に値を持たせたのが Maybe

```
data Bool      = False   | True
data Maybe a   = Nothing | Just a
```

- Bool の連鎖

```
x > 0 && x < 100
```

- Maybe も連鎖にできないか？

```
lookup ... ??? lookup ...
```

Maybe の演算子

- Maybe を生成する関数

```
mwrap :: a -> Maybe a  
mwrap v = Just v
```

- Maybe を合成する関数

```
mbind :: Maybe a -> (a -> Maybe b) -> Maybe b  
mbind Nothing _ = Nothing  
mbind (Just x) f = f x
```

おばあちゃんを探せ(再び)

■ case 版 (再掲)

```
case lookup me db of
  Nothing -> Nothing
  Just mom -> case lookup mom db of
                Nothing -> Nothing
                Just gmom -> Just gmom
```

■ 合成版

```
lookup me db `mbind` \mom ->
lookup mom db `mbind` \gmom ->
mwrap gmam
```

おばあちゃんを探せ(三度)

- 合成版(再掲)

```
lookup me db `mbind` \mom ->
lookup mom db `mbind` \gmom ->
mwrap gmam
```

- あるいは単に

```
lookup me db `mbind` \mom ->
lookup mom db
```

- もうちょっと綺麗に

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

lookup' db me `mbind` lookup' db
```

- あまり意味がないけど

```
mwrap me `mbind` lookup' db `mbind` lookup' db
```

List

- データ定義

```
data [] a = [] | a : [a]
```

- 型表記の統一

```
data [] a = [] | a : [] a
```

- 構成子を前に

```
data [] a = [] | (::) a ([] a)
```

- 記号を単語に

```
data List a = Nil | Cons a (List a)
```

Maybe と List

- データ定義

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
```

- Nil も失敗を表現していると考えられないか？

- Maybe は、失敗と一つの成功
- List は、失敗と複数の成功

- もっと抽象化して

- Maybe は、答えが 0 個か 1 個
- List は、答えが 0 個以上

- Maybe が合成できるなら、List も合成できるのでは？

Maybe の合成とは何なのか？

- 0 か 1 のかけ算と考えられる

`Nothing >>= \x -> Nothing >>= \y -> return (x,y)`
→ `Nothing`

`Nothing >>= \x -> Just 2 >>= \y -> return (x,y)`
→ `Nothing`

`Just 1 >>= \x -> Nothing >>= \y -> return (x,y)`
→ `Nothing`

`Just 1 >>= \x -> Just 2 >>= \y -> return (x,y)`
→ `Just (1,2)`

List の合成はどうあるべきか？

- 要素が0個か1個なら Maybe と同じはず

```
[ ] >>= \x -> [ ] >>= \y -> return (x,y)  
→ [ ]
```

```
[ ] >>= \x -> [2] >>= \y -> return (x,y)  
→ [ ]
```

```
[1] >>= \x -> [ ] >>= \y -> return (x,y)  
→ [ ]
```

```
[1] >>= \x -> [2] >>= \y -> return (x,y)  
→ [(1,2)]
```

- 要素が複数の場合は？ → 決めの問題

```
[1,2] >>= \x -> [3,4,5] >>= \y -> return (x,y)  
→ ???
```

- かけ算だから組み合わせにする

```
→ [(1,3), (1,4), (1,5), (2,3), (2,4), (2,5)]
```

- 実際 zip を取る流儀もある

失敗系

- 失敗が含まれることを表すクラス

```
class Failable m where
  fwrap :: a -> m a
  fbind :: m a -> (a -> m b) -> m b
```

- インスタンス

```
instance Failable Maybe where
  fwrap = mwrap
  fbind = mbind
```

```
instance Failable List where
  fwrap = lwrap
  fbind = lbind
```

- クイズ) lwrap と lbind の実装を考えてみよう

失敗系で統一して何が嬉しいの？

- 木の探索を考える

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

- 探索木ではない

- 右と左のどちらに答えがあるか分からない
- 答えは複数あるかもしれない

- Haskell なら全探索するように書く

```
search :: Eq a => a -> Tree a -> [a]
search _ Leaf = []
search x (Node l v r)
  | x == v      = search x l ++ [v] ++ search x r
  | otherwise   = search x l ++ search x r
```

- 実行してみる

```
search 1 $ Node (Node Leaf 1 Leaf)
              2
              (Node Leaf 1 Leaf)
→ [1,1]
```

失敗系で型を入れ替える

■ 失敗系で抽象化する

```
search :: (Eq a, Failable m, Alternative m) =>
  a -> Tree a -> m a
search _ Leaf = empty
search x (Node l v r)
  | x == v      = search x l <|> fwrap v
                <|> search x r
  | otherwise = search x l <|> search x r
```

■ 動かしてみる

```
search 1 先ほどの木 :: [Int]
→ [1,1]

search 1 先ほどの木 :: Maybe Int
→ Just 1
```

大統一理論

お題

失敗系と状態系は統一できるか？

そのとき

Philip Wadler

の脳裏に閃光が走った！

コンテナ

- 中身が一つのコンテナ

```
data m a = ...
```

- コンテナを生成

```
wrap :: a -> m a  
wrap a = ...
```

- コンテナを合成

```
bind :: m a -> (a -> m b) -> m b  
bind m f = ...
```


プログラム可能コンテナ

■ プログラム可能なコンテナのクラス

```
class Programmable m where
  wrap :: a -> m a
  bind :: m a -> (a -> m b) -> m b
```

■ インスタンス

```
instance Programmable Parser where
  wrap = pwrap
  bind = pbind
```

```
instance Programmable IO where
  wrap = iwrap
  bind = ibind
```

```
instance Programmable Maybe where
  wrap = mwrap
  bind = mbind
```

```
instance Programmable List where
  wrap = lwrap
  bind = lbind
```

プログラム可能コンテナは何が嬉しいの？

- 失敗系でも `do` が使える
 - あまり使わないけれど。。。

```
do mom <- lookup me db
    lookup mom db
```

```
do x <- [1..5]
    y <- [2..5]
    return (x,y)
```

- 他の利点は後述

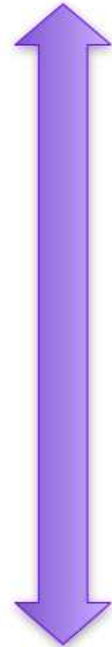
文脈としてのコンテナ

- **Parser**
 - パーサー
 - **IO**
 - 副作用の表現
 - **Maybe**
 - 失敗するかもしれない計算
 - **List**
 - 答えが0個以上の計算
 - 分岐する世界 = 非決定性
- 文脈が混ざらないことが大事！

力の階層

高階関数と再帰の力の階層

強い



弱い

再帰

(自然な)再帰
末尾再帰

畳み込み

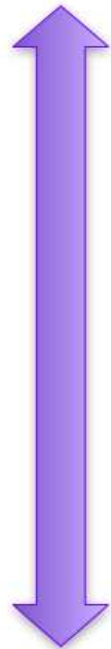
foldr
foldl

単純な高階関数

map
filter

コンテナの力の階層

強い



弱い

プログラム可能
コンテナ

>>=

逐次コンテナ

<*>
return

マップ可能
コンテナ

<\$>

チューリング完全

構造化定理

逐次

分岐

反復

この三つがあれば、計算可能なすべての
アルゴリズムを実現できる

まず map より始めよ

- だれでも分かる map

```
map (+1) [1,2,3,4]  
→ [2,3,4,5]
```

- 二項演算子にしてみる

```
(+1) `map` [1,2,3,4]  
→ [2,3,4,5]
```

- map を一般化した <\$> という演算子を考える

```
(+1) <$> [1,2,3,4]  
→ [2,3,4,5]
```

```
(+1) <$> Just 1  
→ Just 2
```


マップ可能コンテナ

- `<$>` を持つコンテナのクラス

```
class Mappable m where
  (<$>) :: (a -> b) -> m a -> m b
```

- インスタンス

```
class Mappable List where
  (<$>) = map
```

```
class Mappable Maybe where
  (<$>) = mmap
```

- クイズ) `mmap` の実装を考えてみよう

マップ可能の意味

- $\langle \$ \rangle$ の型

$(\langle \$ \rangle) :: (a \rightarrow b) \rightarrow m a \rightarrow m b$

- f へ適用

$f :: a \rightarrow b$

$(f \langle \$ \rangle) :: m a \rightarrow m b$

- $\langle \$ \rangle$ は関数を文脈用に変えている！

- 文脈に持ち上げる(lift)するという

純粋な関数を文脈に入れる

- 二引数の関数を考える

$f :: a \rightarrow b \rightarrow c$

- \rightarrow は右結合

$f :: a \rightarrow (b \rightarrow c)$

- $\langle \$ \rangle$ で文脈に持ち上げると

$(f \langle \$ \rangle) :: m a \rightarrow m (b \rightarrow c)$

- 文脈に関数が入る

$(+) \langle \$ \rangle [1, 2, 3, 4]$

$\rightarrow [(1+), (2+), (3+), (4+)]$

- 文脈の中でも関数適用したい！

文脈の中での関数適用

- 文脈の中で関数適用する演算子 $\langle * \rangle$ の型は？

$m (a \rightarrow b) \rightarrow m a \rightarrow m b$

- 使ってみる

(+) $\langle \$ \rangle$ Just 1 $\langle * \rangle$ Just 2
→ Just 3

(+) $\langle \$ \rangle$ [1,2] $\langle * \rangle$ [3,4]
→ [4,5,5,6]

- 心眼で $\langle \$ \rangle$ と $\langle * \rangle$ を消せ

- 単なる関数適用に見える！

(+) (Just 1) (Just 2)

(+) [1,2] [3,4]

逐次コンテナ

■ <*> を持つコンテナのクラス

```
class Mappable m => Sequential m where
  return :: a -> m a
  (<*>) :: m (a -> b) -> m a -> m b
```

■ return

- 今までの wrap のこと
- return は本当はプログラム可能コンテナのメソッド
- 逐次コンテナの本当のメソッドは pure
- しかし pure と return を同一視しても問題ない
- ここに return が入る理由は次のスライドで分かる

<*> のどこが逐次の？

■ 逐次の do

```
string :: String -> Parser String
string []      = wrap []
string (x:xs) = do v  <- char x
                  vs <- string xs
                  wrap (v:vs) -- (:) v vs
```

■ <\$> と <*> で書く

```
string :: String -> Parser String
string []      = return []
string (x:xs) = (:) <$> char x <*> string xs
```

- 反復と逐次ができるのは明らか！
- 残るは分岐！

プログラム可能コンテナ

- コンテナの中身を見て分岐したい！

- それを実現するのが `>>=`

- これまでの `bind` のこと

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

- `>>=` を持つコンテナのクラス

```
class Sequential m => Programmable m where  
  (>>=) :: m a -> (a -> m b) -> m b
```

分岐の例

- ファイルがあれば削除

```
import System.Directory

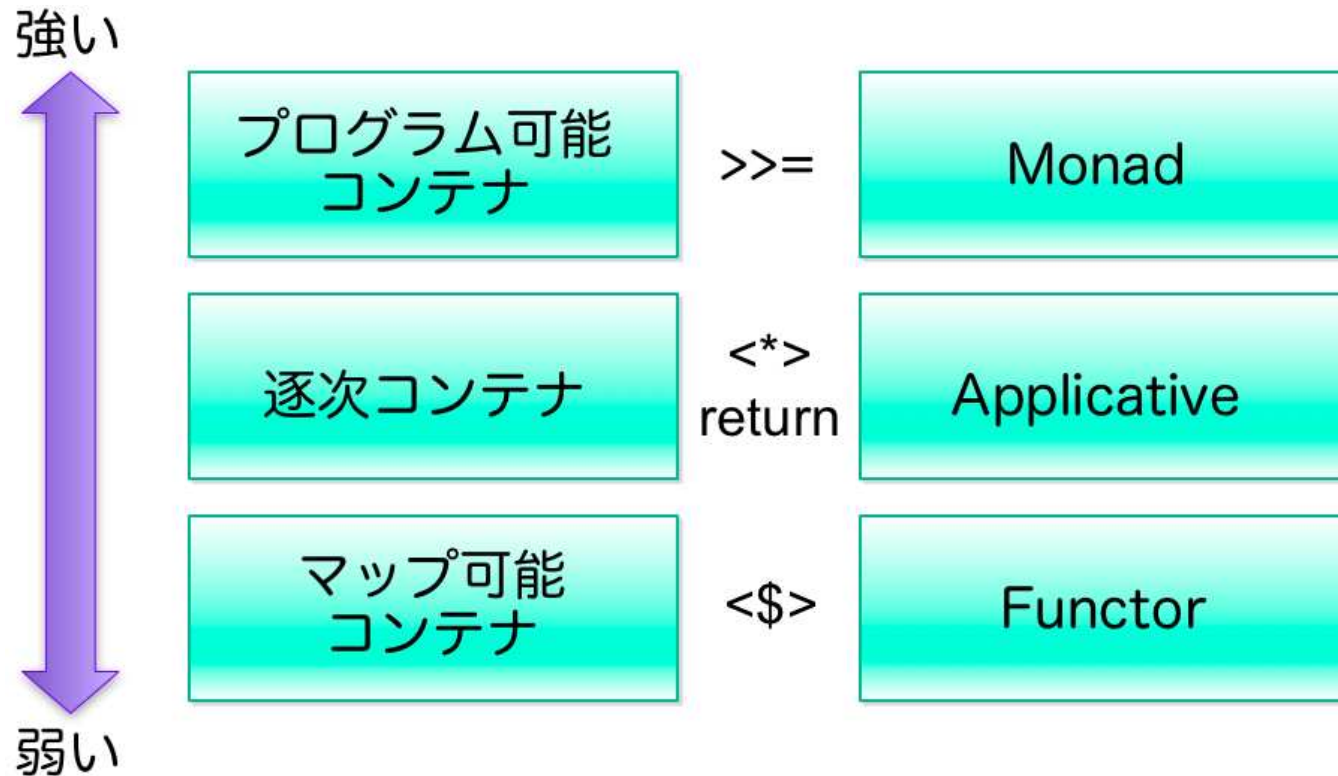
removeFileIfExist file = do
  exist <- doesFileExist file
  if exist
    then removeFile file
    else return ()
```

- あるいは

```
import Control.Monad
import System.Directory

removeFileIfExist file = do
  exist <- doesFileExist file
  when exist $ removeFile file
```


結局モナドとは何？



注：Control.Applicative を import すること！

メタパラダイムとしてのモナド

- モナドはチューリング完全
 - さまざまなパラダイム(文脈)を統一された API で実現可能
- モナドはメタパラダイム
 - 2011 (C) @tanakh
- Parser
 - パーサー
- IO
 - 命令型
- Maybe
 - 失敗するかもしれない計算
- List
 - 非決定性

モナドの嬉しさのまとめ

- do 表記
 - 失敗系ではあまり使わないけど。。。
- 文脈の交換
 - Maybe と List の交換
- 実装の差し替え
 - Parser の実装はモナドのインスタンスであることが多いので、実装の差し替えはあまり苦労しない
 - 例：binary を attoparsec へ変更
- モナド変換子
 - モナド自体を合成できる
 - 状態のない attoparsec へ状態を足す

モナドの神話

- モナドが実行順序を制御する
 - 違います
 - $m \gg= f$ で、 f を評価するには正格な m を必要とするので記述順になるだけです
 - たとえば $x > 0 \ \&\& \ x < 100$ と書けば、この式は記述順に評価されるのと同じことです
- IO モナドは副作用をカプセル化する
 - 意味が分かりません
 - "A History of Haskell" でもこの表現が使われていました orz
- IO モナドが参照透明性を守る
 - 参照透明になるのは「IOは命令書」というアイデアです
 - IO がモナドであることとは、ほとんど関係ありません
- 圏論が分からないとHaskellのモナドは使えない
 - 僕は圏論を理解していません

Monad とは単なる型クラスの一つ

構文糖衣 `do` の存在が Monad を
特殊にしている

Real World Monad

- 階層構造を反映できてない

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

- 多くの Monad のインスタンスは、
Functor と Applicative のインスタンスにもなっている
 - 例外) Parsec 2 は Monad のインスタンスだが、
Applicative のインスタンスではない
- なので実用上は、階層構造があると思って問題ない

Real World do

■ 実際の構文糖衣 do

```
m1 >>= \v1 ->          do v1 <- m1
m2 >>= \v2 ->          v2 <- m2
...                      ...
mn >>= \vn ->          vn <- mn
return (f v1 v2 .. vn)  return (f v1 v2 .. vn)
```

■ 演算子 >>

```
m1 >> m2 = m1 >>= \_ -> m2
```

内包表記

List 内包表記 と do

- List 内包表記

```
pairs n = [ (x,y)
            | x <- [1..n]
            , y <- [1..n]
            , x + y == n
            ]
```

- List の do 表記

```
pairs n = do
  x <- [1..n]
  y <- [1..n]
  guard (x + y == n)
  return (x,y)
```

- 内包表記は do の別表現

- 現在は List に限る

- Scala の for はモナド内包表記の一種

モナドと内包表記の歴史

- 1990: Haskell 1.0
 - Miranda 譲りの List 内包表記
- 1996: Haskell 1.3
 - モナドが登場
- 1997: Haskell 1.4
 - List 内包表記がモナド内包表記へと一般化された
 - do の採用
- 1999: Haskell 98
 - モナド内包表記が List 内包表記へと再び限定された
 - エラーメッセージ構文糖衣を解いた後のだったので意味不明だった
- 2011
 - List 内包表記を再びモナド内包表記へ戻す活動が活発化
 - エラーメッセージは構文糖衣を解く前のものを表示
- 参考文献: "A History of Haskell"

よく使われる表記

- Parser
 - Applicative スタイル : $f \langle \$ \rangle m1 \langle * \rangle m2 \langle * \rangle m3$
 - Monad より弱いのでより安全
- IO
 - do 表記
- Maybe
 - $>>=$
 - 値の連鎖なので演算子の方がしっくりくる
- List
 - List 内包表記
- 所感) do 表記台なし。。。

おめでとうございます

モナドって一体何？



抽象の壁を突破



どうして、
それモナドにしないの？

参考文献

- Typeclassopedia の日本語訳
- あどけない話
 - Applicative のススメ
 - QAで学ぶMonad
 - リストは非決定性のモナド
 - Monadとして抽象化すると何が嬉しいの？
 - Applicative よりも Monad の方が力が強い理由
 - 例題で比較する状態系のモナド
 - 無名関数と>>=に関する質問
 - Haskellには副作用がないのか？
 - Maybe モナドの秘密
- Real World Haskell
 - 18章：モナド変換子