Haskell Lecture 2

kinokkory@shiatsumat

目次

- 関数と仲良くなろう！
- データ型と仲良くなろう！
- 型クラスと仲良くなろう！
- ファンクタと仲良くなろう！
- モナドと友達になろう！
- いろいろなモナドを見てみよう！
- もっとモナドを便利にしよう！

目次2

- 入出力であそぼう！
- モナド変換子とあそぼう！
- 状態遷移であそぼう！
- エラー処理であそぼう！
- 構文解析であそぼう！
- フリーモナドとあそぼう！
- モナドライフを始めよう！

関数と仲良くなろう！

関数のおさらい

- 全ての関数型a->bは一引数関数。
- 2引数関数の型はa->b->cとなる。
  a->b->c = a->(b->c)
  f x y = (f x) y
- 関数は部分適用ができる。

関数のライブラリ

```
($) :: (a->b) -> a -> b
f $ x = f x
-- $ の優先順位は低く、かつ右結合
-- f x $ g y $ h z = f x (g y (h z))
-- 通常の関数適用の優先順位は最高で左結合

(.) :: (b->c) -> (a->b) -> a -> c
(f.g) x = f (g x)
-- f.g は f と g の合成関数
```

関数のライブラリ2

```
id :: a -> a
id x = x
-- 恒等関数

const :: a -> b -> a
const x _ = x
-- const x は定数関数
```

関数のライブラリ3

```
flip :: (a->b->c) -> b-> a -> c
flip f x y = f y x
-- flip f は f の二引数を逆転したもの

on :: (b->b->c) -> (a->b) -> a -> a -> c
(f `on` g) x y = f (g x) (g y)
-- fとgの一種の関数合成
```

fix

```
fix :: (a->a) -> a
fix f = f (fix f)
-- fix f は f の最小不動点
-- つまり f x = x となる値x のうち
-- 最小の計算で終わるもの
```

fixと再帰

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

は

```
factorial :: Int -> Int
factorial = fix factorial'
factorial' :: (Int->Int) -> Int -> Int
factorial' _ 0 = 1
factorial' f n = n * f (n-1)
```

と同じ。

データ型と仲良くなろう！

リスト

```
data [a]          -- a がたくさん
  =  a : [a]      -- cons
   |  []          -- nil

[] a = [a]
```

タプル

```
data () = ()                  -- ユニット
data (a,b) = (a,b)            -- ペア（直積型）
data (a,b,c) = (a,b,c)        -- トリプル

(,) a b = ((,) a) b = (a,b)
(,) x y = ((,) x) y = (x,y)
```

Maybe

```
data Maybe a    -- a かもしれない
  =  Just a     -- 値がある
   | Nothing    -- 値がない
```

Either

```
data Either a b -- aまたはb（直和型）
  =  Left a     -- a
   |  Right b   -- b
```

列挙型

```
data Bool = True | False

data ArithException
  =  Overflow
   | Underflow
   | LossOfPrecision
   | DividedByZero
   | Denormal
   | RatioZeroDenominator
```

木構造

```
data BinTree a          -- 二分木
  =  Bin a (BinTree a) (BinTree a)
   |  Tip

data RoseTree a         -- 多分木
  =  Rose a [RoseTree a]
```

抽象構文木

```
data Expression          -- 数式
  =  ENumber Int
   |  EBinary Symbol Expression Expression
   |  EUnary Symbol Expression
```

Identity, Const, Tagged

```
newtype Identity a = Identity {runIdentity::a}

newtype Const a b = Const {getConst :: a}

newtype Tagged a b = Tagged {unTagged :: b}
```

代数的データ型は直積の直和

```
data D
   =  C1 T1a T2a
    | C2 T2a T2b
```

は

```
type D = Either (T1a,T2a) (T2b,T2b)
```

と大体同じ (同型)。

代数的データ型は直積の直和 2

```
data D
   =  C1
    |  C2 T2
    |  C3 T3a T3b T3c
```

は

```
type D = Either (Either () T2) (T3a,T3b,T3c)
```

と大体同じ (同型)。

型クラスと仲良くなろう！

Eq

```
class Eq a where                    -- 等しさがある
        (==) :: a -> a -> Bool      -- 等しい
        (/=) :: a -> a -> Bool      -- 等しくない
        x == y = not (x /= y)
        x /= y = not (x == y)
```

Ord

```
data Ordering = LT | EQ | GT
class Eq a => Ord a where  -- 比較可能
        compare :: a -> a -> Ordering
        (<), (>=), (>), (<=) :: a -> a -> Bool
        max, min :: a -> a -> a
        x<y = compare x y == LT
        x>=y = compare x y /= LT
        ...
```

Enum

```
class Enum a where   -- 列挙可能
        succ, pred :: a -> a
        toEnum :: Int -> a
        fromEnum :: a -> Int
        enumFrom :: a -> [a]
        enumFromThen :: a -> a -> [a]
        enumFromTo :: a -> a -> [a]
        enumFromThenTo :: a -> a -> a -> [a]
```

Bounded

```
class Bounded a where -- 最小値と最大値がある
    minBound, maxBound :: a
```

Show

```
class Show a where    -- 文字列に変換可能
        show :: a -> String
```

Read

```
type ReadS a = String -> [(a,String)]
class Read a where    -- 文字列から変換可能
      readsPrec :: Int -> ReadS a
read :: Read a => String -> a
```

Ix

```
class Ord a => Ix a where  -- 区間を考えられる
        range :: (a,a) -> [a]
        index :: (a,a) -> a -> Int
        inRange :: (a,a) -> a -> Bool
        rangeSize :: (a,a) -> Int
```

deriving

- Eq, Ord, Enum, Bounded, Show, Read, Ix については、インスタンス宣言をしなくても、データの宣言の際にderivingを用いればインスタンスにできる。

```
data Bool = True | False
  deriving (Eq,Ord,Enum,Bounded,Show,Read,Ix)
```

Num

```
class Num a where
        (+),(*),(-) :: a -> a -> a
        negate :: a -> a
        abs :: a -> a
        signum :: a -> a
        fromInteger :: Integer -> a
```

Real

```
data Ratio a = !a :% !a
type Rational = Ratio Integer
class (Num a, Ord a) => Real a where
      toRational :: a -> Rational
```

Integral

```
class (Real a, Enum a) => Integral a where
        quot, rem, div, mod :: a -> a -> a
        quotRem, divMod :: a -> a -> (a,a)
        toInteger :: a -> Integer
```

Fractional, RealFrac

```
class (Num a) => Fractional a where
        (/) :: a -> a -> a
        recip :: a -> a
        fromRational :: Rational -> a

class (Real a, Fractional a) => RealFrac a where
        truncate,round,ceiling,floor
                :: (Integral b) => a -> b
        properFraction :: (Integral b) => a -> (b,a)
```

Floating, RealFloat

```
class (Fractional a) => Floating a where
        pi :: a
        exp,log,sqrt,sin,cos,tan,asin,acos,atan,
        sinh,cosh,tanh,asinh,aconh,atanh :: a -> a
        (**),logBase :: a -> a –> a

class (RealFrac a, Floating a) => RealFloat a
where
        floatRadix :: a -> Integer
        …
```

モノイド

```
class Monoid a where
        mempty :: a
        mappend :: a -> a -> a
        mconcat :: [a] -> a
        mconcat = foldr mappend mempty

(<>) = mappend
```

モノイド則

```
mempty <> x = x <> mempty = x
x <> (y <> z) = (x <> y) <> z
```

ファンクタと仲良くなろう！

ファンクタって何？

- データ間の変換をおこなうためのもの。
- 日本語の訳語は関手。
- 型引数をひとつ取るデータ型は、基本的には必ずファンクタにすることができる。

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

(<$>) = fmap
```

ファンクタ則

```
fmap id == id
fmap (f . g) == fmap f . fmap g
```

リストファンクタ

```
instance Functor [] where
        fmap = map

map f [] = []
map f (x:xs) = f x : map f xs

map f [a,b,c] = [f a, f b, f c]
```

Maybeファンクタ, Eitherファンクタ

```
instance Functor Maybe where
        fmap _ Nothing = Nothing
        fmap f (Just a) = Just (f a)

instance Functor (Either a) where
        fmap _ (Left x) = Left x
        fmap f (Right y) = Right (f y)
```

タプルファンクタ

```
instance Functor ((,) a) where
        fmap f (x,y) = (x,f y)
```

関数ファンクタ

```
instance Functor ((->) a) where
        fmap f x = f.x
        -- f :: b -> c, x :: a -> b, fmap f x :: a -> c
```

Identity/Const/Taggedファンクタ

```
instance Functor Identity where
        fmap f (Id x) = Id (f x)

instance Functor (Const a) where
        fmap _ x = x

instance Functor (Tagged a) where
        fmap f (Tagged x) = Tagged (f x)
```

ファンクタがいっぱい

- これまでの例から考えると、多くのデータ型はファンクタ！
- 再帰的なデータ型の場合、fmapも再帰的になる。
- 実は、ghcのDeriveFunctor拡張でファンクタを自動生成 (deriving Functor) できる。
  (deriving Functorの挙動を理解するのも重要)

ファンクタにしよう！

```
data RoseTree a          -- 多分木
  =  Rose a [RoseTree a]

instance Functor RoseTree where
      fmap f (Rose x rs) =
            Rose (f x) (fmap (fmap f) rs)
```

反変ファンクタって何？

- ファンクタと逆の対応付けをするもの。
- ふつうのファンクタは共変ファンクタともいう。
  共変と反変は双対の概念。
- contravariant package にある。

```
class Contravariant where
        contramap :: (a -> b) -> f b -> f a
(>$<) = contramap
(>$$<) = flip contramap
```

反変ファンクタ則

```
contramap id = id
contramap f . contramap g = contramap (g . f)
```

双対関数反変ファンクタ

```
newtype Op a b = Op {getOp :: b -> a}

instance Contravariant (Op a) where
        contramap f g = Op (getOp g . f)
        -- f :: b -> c, g :: Op a c
        -- contramap f g :: Op a b
```

双ファンクタって何？

- ファンクタの2引数版。
- bifunctors package にある。

```
class Bifunctor p where
    bimap :: (a->b) -> (c->d) -> p a c -> p b d
    first :: (a->b) -> p a c -> p b c
    second :: (b->c) -> p a b -> p a c
    first f = bimap f id
    second f = bimap id f
```

双ファンクタ則

```
bimap id id = id
first id = id
second id = id
bimap f g  = first f . second g
bimap (f . g) (h . i) = bimap f h . bimap h i
first (f . g) = first f . first g
second (f . g) = second f . second g
```

タプル双ファンクタ

```
instance Bifunctor (,) where
        bimap f g (x,y) = (f x, g y)
```

Either双ファンクタ

```
instance Bifunctor Either where
        bimap f _ (Left x) = Left (f x)
        bimap _ g (Right y) = Right (g y)
```

Const/Tagged双ファンクタ

```
instance Bifunctor Const where
    bimap f _ (Const a) = Const (f a)

instance Bifunctor Tagged where
    bimap _ g (Tagged b) = Tagged (g b)
```

共変と反変

基本的な型構築子のそれぞれの引数について、
共変 (+) か反変 (−) かをあらわすと以下の通り。

- (+,+)
- Either + +（以降、+ | + と表記する。）
- − -> +

共変：covariant
反変：contravariant

組み合わせ

+×+=+, +×−=−, −×+=−, −×−=+

- (+,+,+)
- +|+|+
- − -> (− -> +)
- (+ -> −) -> +
- ((− -> +) -> −) -> +
- (+,+) | (+,+)
- (−,−) -> (+,+)
- (−|−) -> (+|+)

(? -> –) -> + の共変性

```
data Foo b c a = Foo ((a -> b) -> c)
instance Functor (Foo b c) where
        fmap f (Foo x) = Foo (¥y -> x (y . f))
        -- f :: a -> d
        -- x :: (a -> b) -> c
        -- y :: d -> b
        -- y . f :: a -> b
        -- ¥y -> x (y . f) :: (d -> b) -> c
```

自由変と固定変

- newtype Const a b = Const a の場合、
  bは右辺にあらわれていないので、+とも−ともとれる状態。このとき、自由変 (⊥) とする。
- data X a b = Fun (a -> b) | Nuf (b -> a) の場合、
  aもbも+と−の両方でなくてはならず、問題。
  このとき、固定変 (T) とする。

- Const + ⊥
- X T T

自由変：free-variant (or 非変：nonvariant)
固定変：fixed-variant (or 不変：invariant)

補足

- Identity +
- Tagged ⊥ +
- Maybe +
- [+]
- BinaryTree +
- RoseTree +

ファンクタはすごい！

- すべてのデータ型は直和・直積・関数型と再帰的定義によって書ける。
- よって、ほとんどの一引数のデータ型は、ファンクタか反変ファンクタのいずれかである。
- 双ファンクタはファンクタの二引数版である。同様にして、反変ファンクタとファンクタの二引数からなる版もつくれるし、n引数版のものもつくれる。
- ほとんどのデータ型は広い意味でのファンクタ！

モナドと友達になろう！

モナドって何？

- 手続き型的な計算を抽象化したもの。
- 計算を構造化して柔軟に扱えるようになる。
- さまざまなモナドに触れることが理解の近道。
- モナドはとっても楽しい！

モナドの定義

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
    m >> k = m >>= ¥_ -> k
    fail s = error s
```

モナドの意味

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

第一引数を計算した結果を、第二引数に渡して、あらたな結果を得る（バインドと読む）。

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

第一引数を計算し、その結果を無視して、第二引数の計算を引き続き行う。

return :: a -> m a

値を返す単純な計算を得る。

一方向なモナド

- モナドのメンバ関数のみをもちいるとき、「a -> m a」という操作をすることはできるが、「m a -> a」という操作はすることができない。
- IOモナドは基本的に、内部のデータ構造を隠蔽している。このおかげで、Haskellにおいて参照透過性が守られている。これがIOモナドの一方向性である。

一方向なモナドだけれど

- main関数の型は IO a 。だから、IOモナドから脱出できなくても構わない。

do記法

```
do 文1
   文2
   ...
   文n
   モナド
```

という形式で書く。
文はモナドか、モナドから値を取りだす操作 (“x <- exp”, ただしx :: a, exp :: m a) か、let式 (let x = exp) のいずれか。

do記法の意味

単なる糖衣構文。

```
do x <- e1
   y <- e2
   e3
```

は

```
e1 >>= (¥x -> e2 >>= (¥y -> e3))
```

と変換できる。

（Listモナドなどでの挙動を理解しよう。）

failはなんなのか

- あんまり気にしなくていいと思う。自分でfailを呼び出すことは少ない。
- ただdo記法中でパターンマッチに失敗すると呼ばれるので、その動作を変えたければ、failを変えればよい。

モナド則

- このモナド則はプログラムを書く上では
  重要ではありません。

結合法則 (m>>=f)>>= g = m>>=(¥x->f x>>=g)
単位元則 return x >>= f = f x
単位元則 m >>= return = m

いろいろなモナドを見てみよう！

Identityモナド

```
instance Monad Identity where
        return x                    = Identity x
        (Identity x) >>= f          = f x
```

ただ普通に計算するだけのモナド。

Maybeモナド

```
instance Monad Maybe where
        return x          = Just x
        (Just x) >>= k    = k x
        Nothing >>= _     = Nothing
        fail _            = Nothing
```

部分関数を実現しているモナド。

Listモナド

```
instance Monad [] where
        return x     = [x]
        m >>= k      = concatMap k m
        fail _       = []
```

多値関数を実現しているモナド。

関数モナド

```
instance Monad ((->) a) where
        return x     = ¥_ -> x
        f >>= k      = ¥r -> k (f r) r
        -- f :: a -> b, k :: b -> a -> c, r :: a
        -- f >>= k :: a -> c
```

一種の関数合成をしつづける不思議なモナド。
実はReaderモナドそのもの。

もっとモナドを便利にしよう！

モナドはファンクタ

```
instance Monad m => Functor m where
        fmap f x = x >>= (return.f)
```

このコードを動かすにはghc拡張が必要。
WrappedMonadを使うと同様のことができる。

アプリカティヴって何？

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

関数適用のできるファンクタ。

アプリカティヴ則

```
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

ちなみに、fmap f x = pure f <*> x

モナドはアプリカティヴ

```
instance Monad m => Applicative m where
        pure = return
        f <*> g = do{x <- f; y <- g; return (x y)}
```

このコードを動かすにはghc拡張が必要。
WrappedMonadを使うと同様のことができる。

モナドはアプリカティヴ2

do {x<-a; y<-b; z<-c; return (f x y z)} は
f <$> a <*> b <*> c と書ける。

モナドじゃないアプリカティヴ

```
instance Monoid a => Applicative ((,) a) where
    pure x = (mempty, x)
    (u, f) <*> (v, x) = (u <> v, f x)
```

オルターナティヴって何？

```
class Applicative f => Alternative f where
        empty :: f a
        (<|>) :: f a -> f a -> f a
        some :: f a -> f [a]
        many :: f a -> f [a]
        some v = (:) <$> v <*> many v
        many v = some v <|> pure []
```

モノイドのあるアプリカティヴ。

モナドプラスって何？

```
class Monad m => MonadPlus m where
        mzero :: m a
        mplus :: m a -> m a -> m a
```

失敗と選択のあるモナド。

Listモナドプラス

```
instance MonadPlus [] where
        mzero = []
        mplus = (++)
```

Maybeモナドプラス

```
instance MonadPlus Maybe where
        mzero = Nothing
        Nothing `mplus` ys      = ys
        xs `mplus` _            = xs
```

モナドプラスはオルターナティヴ

```
instance MonadPlus m => Alternative m where
        empty = mzero
        (<|>) = mplus
```

このコードを動かすにはghc拡張が必要。
WrappedMonadを使うと同様のことができる。

繰り返し

```
sequence :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
replicateM :: Monad m => Int -> m a -> m [a]
replicateM_ :: Monad m => Int -> m a -> m ()
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
forM = flip mapM
forM_ = flip mapM_
forever :: Monad m => m a -> m b
```

畳み込み

```
foldM :: Monad m=> (a->b->m a)->a->[b]->m a
foldM_::Monad m=>(a->b->m a)->a->[b]->m ()
msum :: MonadPlus m => [m a] -> m a
```

条件分岐

```
guard :: MonadPlus m => Bool -> m ()
when :: Monad m => Bool -> m () -> m ()
unless = when.not
mfilter :: MonadPlus m=> (a->Bool)->m a->m a
```

その他

```
(=<<) = flip (>>=)
(>=>)::Monad m=>(a->m b)->(b->m c)->a->m c
(<=<) = flip (>=>)
join :: Monad m => m (m a) -> m a
liftM :: Monad m => (a -> r) -> m a -> m r
liftM2 :: Monad m=>(a->b->r)->m a->m b->m r
...
liftM5 :: Monad m => ...
ap :: Monad m => m (a -> b) -> m a -> m b
```

モナド則 クライスリ圏版

```
(>=>)::Monad m=>(a->m b)->(b->m c)->(a->m c)
f >=> g = ¥x -> f x >>= g

-- >=>とfmapから>>=を定義することもできる
(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = (const x >=> f) x
```

モナド則 クライスリ圏版2

結合法則 (f >=> g) >=> h = f >=> (g >=> h)
単位元則 return >=>f = f
単位元則 f >=> return = f

モナド則 自然変換版

```
join :: Monad m => m (m a) -> m a
join x = x >>= id

-- joinとfmapから>>=を定義することもできる
(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

モナド則 自然変換版2

結合法則 join . fmap join = join . join
単位元則 join . return = id
単位元則 join . fmap return = id
自然性　 return . f = fmap f . return
自然性　 join . fmap (fmap f) = fmap f . join

入出力であそぼう！

IOモナド

```
newtype IO a = …
instance Monad IO where
        …
```

現実世界を裏で操作する計算。

ファイル処理

```
data Handle = ...
data IOMode = ReadMode | WriteMode |
              AppendMode | ReadWriteMode
type FilePath = String
openFile :: FilePath -> IOMode -> IO Handle
```

入出力

```
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hLookAhead :: Handle -> IO Char
hGetContents :: Handle -> IO String
hIsEOF :: Handle -> IO Bool
hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()
hPrint :: Show a => Handle -> a -> IO ()
```

ランダムアクセス

```
data SeekMode =
    AbsoluteSeek | RelativeSeek | SeekFromEnd
data HandlePosn = HandlePosn Handle Integer

hFileSize :: Handle -> IO Integer
hTell :: Handle -> IO Integer
hSeek :: Handle -> SeekMode -> Integer -> IO()
hGetPosn :: Handle -> IO HandlePosn
hSetPosn :: HandlePosn -> IO ()
```

コンソール処理

```
stdin, stdout, stderr :: Handle
getChar = hGetChar stdin
getLine = hGetLine stdin
getContents = hGetContents stdin
isEOF = hIsEOF stdin
putChar = hPutChar stdout
putStr = hPutStr stdout
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

便利な関数

```
readIO :: Read a => String -> IO a
readLn = getLine >>= readIO
interact :: (String -> String) -> IO ()
```

IOも遅延評価

- 遅延評価はHaskellの基本。
- IOも当然遅延評価される。
- うまく使いこなせると楽しい。

IORef

```
data IORef a = ...
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
modifyIORef' :: IORef a -> (a -> a) -> IO ()
```

正格評価版

再代入可能な変数への参照。
メモリの削減に役立つ。

性能比較

```
big = 1000000
go1 = f big >>= print
   where   f 0 = return  0
           f n = f (n-1) >>= return.(+1)
go2 = do {r <- newIORef 0; replicateM_ big
(modifyIORef r (+1)); readIORef r >>= print}
go3 = do {r <- newIORef 0; replicateM_ big
(modifyIORef' r (+1)); readIORef r >>= print}
```

go3はgo1, go2と比べてメモリを大きく削減する

乱数生成

- 乱数も基本的に参照透明性を壊してしまう。
- そこでIOを用いて乱数を生成する。
- 基本的に、次の使い方だけ覚えればよい。

```
getStdRandom (randomR (lo::a,hi::a)) :: IO a
-- lo以上hi以下の乱数の取得
getStdRandom random :: IO a
-- aの上限から下限までの範囲の乱数の取得
-- ただし実数型については0以上1未満
```

モナド変換子とあそぼう！

モナド変換子って何？

- 複数のモナドの機能をくっつけて新しいモナドを作りたいことは多い。
- そこでモナド変換子。
- モナドを直列につなぐのに使える。

モナド変換子の定義

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

モナドを一段上のモナドにできる。

（各 t に対し Monad m => Monad (t m) をインスタンス宣言する必要がある。）

モナド変換子則

```
lift . return = return
lift (m >>= f) = lift m >>= (lift . f)
-- m :: t m a, f :: a -> t m b, m >>= f :: t m b
-- lift (m >>= f) :: m b
-- lift m :: m a, lift . f :: a -> m b
-- lift m >>= (lift . f) :: m b
```

MaybeTモナド変換子

```
newtype MaybeT m a =
        MaybeT {runMaybeT :: m (Maybe a)}

instance MonadTrans MaybeT where
        lift m = MaybeT (Just <$> m)
```

MaybeTモナド変換子2

```
instance (Monad m)=>Monad (MaybeT m) where
        return = lift . return
        x >>= f = MaybeT $ do
                v <- runMaybeT x
                case v of
                        Nothing -> return Nothing
                        Just y -> runMaybeT (f y)
        fail _ = MaybeT (return Nothing)
```

ListTモナド変換子

```
newtype ListT m a = ListT {runListT :: m [a]}

instance MonadTrans ListT where
        lift m = ListT $ do {a <- m; return [a]}
```

ListTモナド変換子2

```
instance (Monad m) => Monad (ListT m) where
    return = lift . return
    m >>= k = ListT $ do
        a <- runListT m
        concat <$> mapM (runListT . k) a
    fail _ = ListT $ return []
```

MonadIO

```
class (Monad m) => MonadIO m where
        liftIO :: IO a -> m a
```

IOからの変換。最適化のためにある。

MonadIOのインスタンス宣言

```
instance MonadIO IO where
        liftIO = id
instance (Monad m)=>MonadIO (ListT m) where
        liftIO = lift . liftIO
        -- ほぼすべてのモナド変換子で
        -- このように宣言する
```

状態遷移であそぼう！

StateTとState

```
newtype StateT s m a =
        StateT {runStateT :: s -> m (a,s)}

type State s = StateT s Identity
```

古い状態から値と新しい状態を得る。

StateTモナド変換子

```
instance MonadTrans StateT where
    lift m = StateT $ (¥s -> return (m,s))
instance Monad m => Monad(StateT s m) where
    return = lift . return
    (StateT x) >>= f = StateT $ ¥s -> do
        (v,s') <- x s
        runStateT (f v) s'
    fail = StateT $ ¥_ -> fail msg
```

モナドステート

```
class Monad m => MonadState s m | m->s where
    get :: m s
    put :: s -> m ()
    state :: (s -> (a,s)) -> m a
    get = state (¥s -> (s,s))
    put s = state (¥_ -> ((),s))
    state f = do ~(a,s') <- f <$> get
                 put s'
                 return a
```

s は m に応じて一意に決まる
(関数従属)

状態の取得と更新。

StateTモナドステート

```
instance Monad m=> MonadState s (StateT s m)
where
        get = StateT $ ¥s -> return (s,s)
        put s = StateT $ ¥_ -> return ((),s)
```

ツッコミ

- Stateモナドは新しい状態にする際、古い状態は残したままで、取り替える方式にしてある。
- 古い状態を上書きしていく形式にすればいいじゃないか！
- そこでSTモナド。

STモナド

```
newtype ST s a =
        ST (State# s -> (# State# s, a #))
instance Monad (ST s) where
        return x = ST (¥s -> (#s,x#))
        (ST m) >>= k = ST (¥s ->
                let{(#new_s,r#) = m s;ST k2 = k r} in
                (k2 new_s)
runST :: (forall s. ST s a) -> a
```

黒魔術によってStateモナドを効率化したもの。

STRef

```
data STRef s a = STRef (MutVar# s a)
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
modifySTRef :: STRef s a -> (a -> a) -> ST ()
modifySTRef' :: STRef s a -> (a -> a) -> ST ()
```

再代入可能な変数への参照。

IOはST RealWorld

```
data RealWorld = ??? -- Haskellの闇
newtype ST s a = ST (State# s ->
                     (# State# s, a #))
newtype IO a = IO (State# RealWorld ->
                     (# State# RealWorld, a #)

stToIO :: ST RealWorld a -> IO a
stToIO (ST m) = IO m
ioToST :: IO a -> ST RealWorld a
ioToST (IO m) = ST m
```

IORefはSTRef RealWorld

```
newtype IORef a = IORef (STRef RealWorld a)
```

ReaderTとReader

```
newtype ReaderT r m a =
            ReaderT{runReaderT :: r -> m a}

type Reader = ReaderT Identity
```

環境を参照できる計算。
Readerは関数モナドと同じ。

ReaderTモナド変換子

```
instance MonadTrans (ReaderT r) where
    lift m = ReaderT $ ¥_ -> m
instance Monad m=>Monad(ReaderT r m) where
    return  = lift . return
    m >>= k = ReaderT $ ¥r -> do
        a <- runReaderT m r
        runReaderT (k a) r
    fail msg = ReaderT $ ¥_ -> fail msg
```

モナドリーダー

```
class Monad m=>MonadReader r m | m->r where
        ask :: m r
        local :: (r -> r) -> m a -> m a
        reader :: (r -> a) -> m a
        reader f = f <$> ask
```

環境の取得と局所実行。

ReaderTモナドリーダー

```
instance Monad m =>
MonadReader r (ReaderT r m) where
        ask = ReaderT return
        local f m = ReaderT $ ¥r ->
                                    runReaderT m (f r)
```

WriterTとWriter

```
newtype WriterT w m a =
        WriterT {runWriterT :: m (a,w)}

type Writer = WriterT Identity
```

ログを蓄積する計算。
ログは必ずモノイドにする。

WriterTモナド変換子

```
instance Monoid w => MonadTrans (WriterT w)
where
        lift m=WriterT$
                        do {a<-m; return (a,mempty)}
```

WriterTモナド変換子2

```
instance (Monoid w, Monad m) =>
Monad (WriterT w m) where
        return = lift . return
        m >>= k = WriterT $ do
                ~(a, w) <- runWriterT m
                ~(b, w') <- runWriterT (k a)
                return (b, w<>w')
        fail = WriterT . fail
```

モナドライター

```
class (Monoid w, Monad m)=>MonadWriter w m
where
        writer :: (a, w) -> m a
        tell :: w -> m ()
        listen :: m a -> m (a, w)
        pass :: m (a, w -> w) -> m a
        writer ~(a,w) = tell w >> return a
        tell w = writer ((), w)
```

ログへの書き足しとログの取得。

WriterTモナドライター

```
instance (Monoid w, Monad m) =>
MonadWriter w (WriterT w m) where
        tell w = WriterT $ return ((),w)
        listen m = WriterT $ do
                ~(a,w) <- runWriterT m
                return ((a, w), w)
        pass m = WriterT $ do
                ~((a,f), w) <- runWriterT m
                return (a, f w)
```

RWSTとRWS

```
newtype RWST m r w s a =
        RWST {runRWST :: r -> s -> m (a,s,w)}

type RWS = RWST Identity
```

ReaderTとWriterTとStateTを全部合成したもの。
かなり汎用的に使えて便利。

RWSTモナド変換子

```
instance Monoid w =>
MonadTrans (RWST r w s) where
        lift m = RWST $ ¥_ s ->
                do {a<-m; return (a,s,mempty)}
```

RWSTモナド変換子2

```
instance (Monoid w, Monad m) =>
Monad (RWST r w s m) where
        return = lift . return
        m >>= k = RWST $ ¥r s -> do
                ~(a,s',w) <- runRWST m r s
                ~(b,s'',w') <- runRWST (k a) r s'
                return (b, s'', w<>w')
        fail msg = RWST $ ¥_ _ -> fail msg
```

RWSTモナドステート

```
instance (Monoid w, Monad m) =>
MonadState s (RWST r w s m) where
        get = RWST $ ¥_ s -> return (s,s,mempty)
        put s = RWST $ ¥_ _ -> return ((),s,mempty)
```

RWSTモナドリーダー

```
instance (Monoid w, Monad m) =>
MonadReader r (RWST r w s m) where
        ask = RWST $ ¥r s -> return (r,s,mempty)
        local f m = RWST $ ¥r s -> runRWST m (f r) s
```

RWSTモナドライター

```
instance (Monoid w, Monad m) =>
MonadWriter w (RWST r w s m) where
        tell w = RWST $ ¥_ s -> return ((),s,w)
        listen m = RWST $ ¥r s -> do
                ~(a,s',w) <- runRWST m r s
                return ((a,w),s',w)
        pass m = RWST $ ¥r s -> do
                ~((a,f),s',w) <- runRWST m r s
                return (a,s',f,w)
```

エラー処理であそぼう！

ErrorT

```
newtype ErrorT e m a =
        ErrorT {runErrorT :: m (Either e a)}
```

エラーになる可能性のあるモナド。
Maybeモナドの改善版。
eは必ずエラーのインスタンス。

エラー

```
class Error e where
        noMsg :: e
        strMsg :: String -> e
        noMsg = strMsg “”
        strMsg _ = noMsg
```

エラーのインスタンス

```
data IOException = IOError ...
instance Error IOException where
        ...
class ErrorList a where
        ...
instance ErrorList Char where
        ...
instance ErrorList a => Error [a] where
        ...
```

もちろん自分でもインスタンス宣言できる。

ErrorTモナド変換子

```
instance Error e=> MonadTrans (ErrorT e) where
        lift m = ErrorT $ Right <$> m
```

ErrorTモナド変換子2

```
instance (Monad m, Error e) =>
Monad (ErrorT e m) where
        return = lift . return
        m >>= k = ErrorT $ do
                a <- runErrorT m
                case a of
                        Left _ -> return a
                        Right _ -> runErrorT (k a)
        fail msg = ErrorT$return(Left (strMsg msg))
```

モナドエラー

```
class Monad m=> MonadError e m | m->e where
        throwError :: e -> m a
        catchError :: m a -> (e -> m a) -> m a
```

エラーのスローとキャッチ。

ErrorTモナドエラー

```
instance (Monad m, Error e) =>
MonadError e (ErrorT e m) where
        throwError l = ErrorT $ return (Left l)
        m `catchError` h = ErrorT $ do
                a <- runErrorT m
                case a of
                        Left l -> runErrorT (h l)
                        Right _ -> return a
```

ErrorTモナドプラス

```
instance (Monad m, Error e) =>
MonadPlus (ErrorT e m) where
        mzero = ErrorT $ return (Left noMsg)
        m `mplus` n = ErrorT $ do
                a <- runErrorT m
                case a of
                        Left _ -> runErrorT n
                        Right _ -> return a
```

構文解析であそぼう！

モナディックパーサー

- モナディックパーサーはすごく実用的。
- しかもモナドの総まとめとして最適。

いろいろなモナディックパーサー

- Parsec — いちばん有名
- Attoparsec — Parsecより機能が貧弱だが高速
- Pappy — パックラットパーサー

どれも使い方は似ている。
ここではParsecを紹介する。

文献

- Parsec, a fast combinator parser
- Programming Haskell Chapter8 関数型パーサ
- Write Yourself a Scheme in 48 Hours

ParsecTとParsec

```
data ParsecT s u m a =
      ParsecT {unParser :: ...}
-- s は入力を、uはユーザー定義状態を、
-- m はモナドを、a はパース結果を表す

type Parsec s u = ParsecT s u Identity

type Parser = Parsec String ()
```

さまざまなインスタンス宣言

```
instance MonadTrans (ParsecT s u) where ...
instance Monad (ParsecT s u m) where ...
instance MonadPlus (ParsecT s u m) where ...
instance Functor (ParsecT s u m) where ...
instance Applicative (ParsecT s u m) where ...
instance Alternative (ParsecT s u m) where ...
```

実行

- parse :: Stream s Identity t =>
  Parsec s () a -> -- パーサー
  SourceName -> -- ファイル名 (デバッグ用)
  s -> -- 文字列
  Either ParseError a
- parseTest :: (Stream s Identity t, Show a) =>
  Parsec s () a -> -- パーサー
  s -> -- 文字列
  IO () -- 出力する

基本操作

- p >>= f, p >> q, do{...}
  順番に実行する。
- p <|> q
  パーサの選択肢を作る。
  p でパースし、p が入力を消費せず失敗すれば q でパースする。
  p で入力を一文字でも消費してから失敗すれば、p <|> q 全体が失敗する。

先読み

- try p
  p が入力を消費して失敗しても、入力を消費せずに失敗したことにしてくれる。
- notFollowedBy p
  pが失敗したときのみ成功する。入力は消費しない。

繰り返し

- many/many1 p ― 0/1回以上の繰り返し
- count n p ― n回の繰り返し
- sepBy/sepBy1 p sep
  ― sepで区切られたpの0/1回以上の繰り返し

中置演算子

- chainl/chainr p op x
  — pを左/右結合演算子opでつないだもの。xはpが1個もない場合の値。
- chainl1/chainr1 p op
  — pを左/右結合演算子opで1個以上つないだもの。

中置/前置/後置演算子

```
data Assoc = AssocNone|AssocLeft|AssocRight
data Operator s u m a
   =  Infix (ParsecT s u m (a -> a -> a)) Assoc
    | Prefix (ParsecT s u m (a -> a))
    | Postfix (ParsecT s u m (a -> a))
type OperatorTable s u m a = [[Operator s u m a]]
buildExpressionParser :: Stream s m t =>
        OperatorTable s u m a ->
        ParsecT s u m a ->
        ParsecT s u m a -- 式のパーサー
```

字句解析

- anyChar — 任意の文字
- space/newline/tab/eof — 空白/改行/タブ/EOF
- upper/lower/letter — 大文字/小文字/両方
- digit — 数字
- char c — 文字 c
- satisfy cond — condを満たす文字
- oneOf/noneOf cs
  — csのうちのいずれか/いずれでもないもの
- string s — 文字列 s

そのほかの処理

- choice [p1,...,pn] — 順に<|>でつなぐ
- option x p — なくてもよいpで、xがデフォルト値
- optionMaybe p — 値がなければNothing
- optional p — 結果を返さない
- between open close p — pをopenとcloseで挟む

エラー処理

- mzero — 失敗
- fail msg — メッセージ付きの失敗
- p <?> name — p にデバッグ用の名前を付ける

状態遷移

- getInput — 入力の取得
- setInput input — 入力の設定
- getPosition — 走査位置の取得
- setPosition pos — 走査位置の設定
- getState — ユーザー定義状態の取得
- putState p — ユーザー定義状態の設定

モナドライフを始めよう！

さあ、旅立とう！

- モナドを使っているソースコードを読もう。
- モナドを使ってソースコードを書こう。
- 内部実装をより詳しく見てみよう。
- モナドで作曲しよう。

演習問題

- https://github.com/Kinokkory/HaskellLecture
  に提出してください。

Freeモナド

- 再帰的なデータ型からモナドを簡単につくれる。
- IOに変換して実行したり、実行を制御したり、DSLとして構文をプリントしたり、などいろいろな使い方がある。
- さまざまなインスタンス宣言を省けてうれしい。
- とにかくたのしい！

Pipes

- モナドよりもカッコいい何か。
- ストリーミングをうまく行える。
- 設計がとにかく美しい。

GUI — GLUT

- OpenGLをHaskellで使うためのもの。
- モナドを使うと手続き型にしか見えない。
- Haskellで簡単にGUIがつくれて感動！

GUI — Free Game

- Freeモナドでゲームを作ろうという試み。
- クロスプラットフォーム。
- fumieval氏が開発中。

圏論

- そろそろファンクタ則とかモナド則とか気になってきたはず。
- Haskellをつかうのには圏論は要らないけれど、プログラミングでの法則を説明したり、新しい概念を考えたりするのに圏論はつかえる。
- まずはソースコードを書くのが先決だけれど、なんとなく理解しておいてもいいかもしれない。

アロー

- モナドよりも一般化された計算機構。
- 僕も正直ほとんど理解していない。
- トレース圏とかが書けてうれしいみたい。

コモナド

- モナドの双対。再帰の一般化のために使える。

文献

- モナドのすべて
- 本物のプログラマはHaskellを使う
- Typeclassopedia (日本語訳)
- モナモナ言わないモナド入門
- あどけない話 (QAで学ぶMonadなど)

Haskell Lecture 3 へつづく