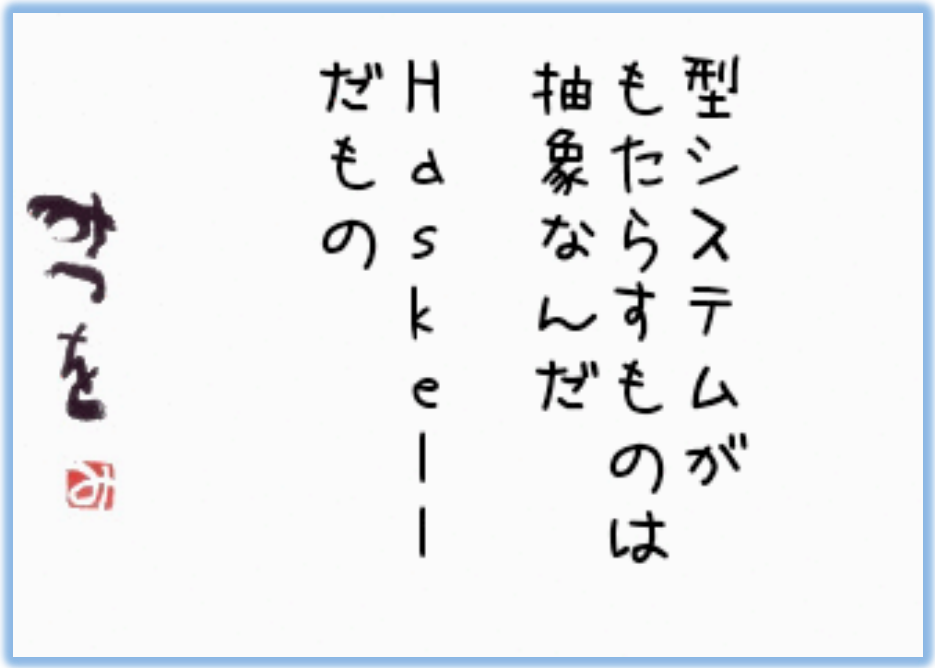
プログラミングHaskell
第十章 型とクラスの定義

kzfm

そこはかとなく悟ろう



内容

- typeによる型宣言
- dataによる型宣言
- 再帰型
- 恒真式検査
- 仮想マシン
- クラスとインスタンスの宣言

typeによる型宣言

- typeは既存の型に別名をつける
  - 新しい型の名前は大文字で始まらなければならない
    - 例 type String = [Char]
  - 入れ子にもできる
    - type Board = [Pos]
    - type Pos = (Int,Int)
  - 再帰にはできない
    - でも newtype ならできる（なぜならニュータイプだから☺）
  - 多相的にできる,複数の型変数もOK
    - type Parser a = String -> [(a, String)]
    - type Assoc k v = [(k,v)]

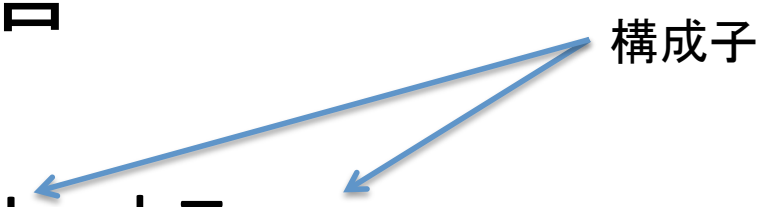
find

- Eqクラスである必要がある

```
type Assoc k v = [(k, v)]

find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k == k']
```

dataによる型宣言

- 新しい型を宣言
- data Bool = False | True



  - 値は引数として関数に渡せる
  - 関数からの結果として戻せる
  - リストのようなデータ構造の要素にすることが出来る
  - パターンにも利用出来る

move

```
data Move = Left | Right | Up | Down

move :: Move -> Pos -> Pos
move Left  (x,y) = (x-1,y)
move Right (x,y) = (x+1,y)
move Up    (x,y) = (x,y+1)
move Down  (x,y) = (x,y-1)

moves :: [Move]  -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)

flip :: Move -> Move
flip Left  = Right
flip Right = Left
flip Up    = Down
flip Down  = Up
```

shape

```
data Shape = Circle Float | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rect x y) = x * y
```

構成子関数

- 構成子は関数
  - 引数をとるから

```
*Main> :t Circle
Circle :: Float -> Shape
```

- ただし、それ以上簡約できない
- 等式ももたない

Maybe（失敗するかも知れない）

```
data Maybe a = Nothing | Just a

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

8章のパーザーでは失敗を空リストで表現していたがMaybeでもいいはず

typeとdataの違い

- typeは直交座標系と極座標系の区別が出来ないが、dataで明確に定義しておけば、そういう自明なエラーはコンパイル時に検出できる
  - (1,2)が極座標系なのか直交座標系なのか判断できない

```
type Coodinate = (Int,Int)

data Cartesian = Cart (Int, Int) deriving (Show, Eq)
data Polar     = Polar (Int, Int) deriving (Show, Eq)
```

再帰型(dataは再帰が可能)

- 自然数
  - ZeroまたはSucc Nat
  - SuccがZeroに何回適用されているかで数を表す
    - Zero # 0
    - Succ Zero # 1
    - Succ (Succ Zero) # 2

Nat

```
data Nat = Zero | Succ Nat deriving Show

nat2int :: Nat -> Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0     = Zero
int2nat (n+1) = Succ (int2nat n)

add :: Nat -> Nat -> Nat
add m n = int2nat(nat2int m + nat2int n)

add Zero n = n
add (Succ m) n = Succ (add m n)
```

List

- data List a = Nil | Cons a (List a)
  - 1:(2:(3:(4:[]))

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

len :: List a -> Int
len Nil = 0
len (Cons _ xs) = 1 + len xs
```

二分木

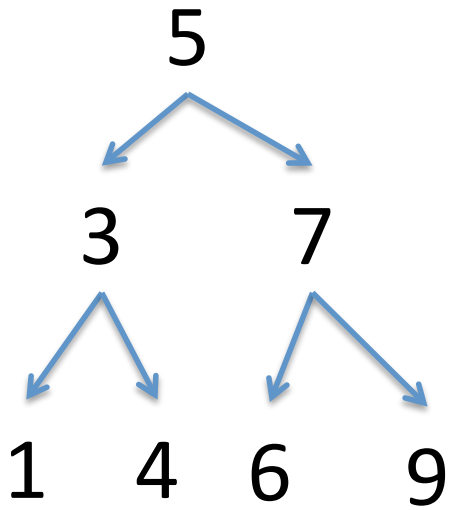
```
data Tree = Leaf Int | Node Tree Int Tree deriving Show

t :: Tree
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))

occurs :: Int -> Tree -> Bool
occurs m (Leaf n) = m == n
occurs m (Node l n r) = m == n || occurs m l || occurs m r

flatten :: Tree -> [Int]
flatten (Leaf n) = [n]
flatten (Node l n r) = flatten l ++ [n] ++ flatten r
```

flattenを適用した結果整列されたリストが得られる場合、その木は探索木と呼ばれる

二分木（続き）

- 探索の範囲を狭める

```
occurs :: Int -> Tree -> Bool
occurs m (Leaf n) = m == n
occurs m (Node l n r)
    | m == n    = True
    | m <  n    = occurs m l
    | otherwise = occurs m r
```



tautology

- 常に真となる命題のこと
  - トートロジーはつまりトートロジーである。
  - 主語と述語が等しい命題。また、特に記号論理学では、束縛されていない変数に何を代入しても真になってしまう命題。恒真命題。

恒真式検査

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop deriving Show

p1 :: Prop
p1 = And (Var 'A') (Not (Var 'A'))

p2 :: Prop
p2 = Imply (And (Var 'A') (Var 'B')) (Var 'A')

p3 :: Prop
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))

p4 :: Prop
p4 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

恒真式検査（続き）

```
type Subst = Assoc Char Bool

eval :: Subst -> Prop -> Bool
eval _ (Const b)   = b
eval s (Var x)     = find x s
eval s (Not p)     = not (eval s p)
eval s (And p q)   = eval s p && eval s q
eval s (Imply p q) = eval s p <= eval s q

vars :: Prop -> [Char]
vars (Const _)     = []
vars (Var x)     = [x]
vars (Not p)     = vars p
vars (And p q)   = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q

substs :: Prop -> [Subst]
substs p = map (zip vs) (bools (length vs))
        where vs = rmdups (vars p)

isTaut :: Prop -> Bool
isTaut p = and [eval s p | s <- substs p]
```

仮想マシン

```
data Expr = Val Int | Add Expr Expr

value :: Expr -> Int
value (Val n) = n
value (Add x y) = value x + value y
```

- 整数と加算演算子からなる単純な数式を解くための仮想マシンを定義する
  - 処理の順番を指示できる

```
type Cont = [Op]
data Op = EVAL Expr | ADD Int
```

スタックの型

eval

- 式が整数であれば、制御スタックの命令実行
  - 加算なら最初の引数を評価し、命令EVAL y を制御スタックの上に置く。
  - 二番目の引数は最初の引数が終わった後に評価

```
eval :: Expr -> Cont -> Int
eval (Val  n) c  = exec c n
eval (Add x y) c = eval x (EVAL y:c)
```

exec

- 制御スタックが空であれば引数である整数を返す
- 制御スタックの一番上にEVAL yがあればADD nを制御スタックの上に置く
- 制御スタックの一番上がADD nであれば二つの値を足しあわせた値の下で残りの制御スタックを実行

```
exec :: Cont -> Int -> Int
exec [] n       = n
exec (EVAL y:c) n = eval y (ADD n:c)
exec (ADD n:c) m  = exec c (n + m)
```

仮想マシンの評価の流れ

- value (Add (Add (Val 2) (Val 3)) (Val 4)) # value
- eval (Add (Add (Val2) (Val 3)) (Val 4)) [] # eval
- eval (Add (Val2) (Val 3)) [EVAL (Val 4)] # eval
- eval (Val 2) [EVAL (Val 3), EVAL (Val 4)] # eval
- exec [EVAL (Val 3), EVAL (Val 4)] 2 #exec
- eval (Val 3) [Add 2, EVAL (Val 4] #eval
- exec [ADD 2,EVAL (Val 4)] 3 #exec
- exec [EVAL (Val 4)] 5 #exec
- eval (Val 4) [ADD 5] #eval
- exec [ADD 5] 4 #exec
- exec [] 9 #exec
- 9

クラスとインスタンスの宣言

- 新しいクラスはclassを用いて宣言

```
class Eq a where
    (==),(\=) :: a -> a -> Bool
    x \= y  = not (x == y)
```

- ある型aがクラスEqのインスタンスになるためには、同等と不等のメソッドを実装する必要があるという宣言

クラスの拡張

- Eqを拡張してOrdを定義

```
class Eq a => Ord a where
  (<),(<=),(>),(>=) :: a -> a -> Bool
  min,max
  min x y | x <= y    = x
          | otherwise = y
  max x y | x <= y    = y
          | otherwise = x
```

- 同等クラスを順序クラスのインスタンスに

```
instance Ord Bool where
  False < True = True
  _ < _        = False
  b <= c       = (b < c) && (b == c)
  b > c        = c < b
  b >= c       = c <= b
```

インスタンスの自動導出

- いくつかのクラスはderivingを使うと自動的にインスタンスにできる。
  - Show とか
  - data Bool = False | True deriving (Eq, Ord, Show, Read)
- 構成子の順序は宣言中の位置できまる
- readは型を決定するために::が必要な場合がある

モナド型

- モナドは値およびその値を使う計算の並びという観点からいえば、計算を構造化 する方法です
  - http://www.sampou.org/haskell/a-a-monads/html/introduction.html#what

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

- モナドとはメソッドreturnと(>>=)を実装し、且つ型変数aをとる型m

do記法

- IOモナド
- リストモナド
  - リスト内包表記の<-とか
- Maybeモナド

練習問題