

# すごいHaskellたのしく学ぼう！

## 読書会#2

山縣ひろか

# 本日の流れ

- ▶ 練習問題解答
  - 練習問題①
  - 練習問題②
  - 練習問題③
  - 練習問題④
- ▶ 第3章 関数の構文
- ▶ 第4章 Hello再帰!
- ▶ 練習問題

# 練習問題解答

➤➤ 練習問題解答①～④

# 練習問題①解答

- ▶ 論理学者であるハスケル・カリーに由来する。  
(by Wikipedia先生)  
→ <http://ja.wikipedia.org/wiki/Haskell>
- ▶ ハスケル・カリー  
アメリカの数学者、論理学者。  
→ <http://ja.wikipedia.org/wiki/%E3%83%8F%E3%82%B9%E3%82%B1%E3%83%AB%E3%83%BB%E3%82%AB%E3%83%AA%E3%83%BC>
- ▶ 組合せ論理/コンビネータ論理  
→ <http://ja.wikipedia.org/wiki/%E7%B5%84%E5%90%88%E3%81%9B%E8%AB%96%E7%90%86>

# 練習問題②解答

- ▶ cycle: リストを受け取り、その要素を無限に繰り返し、無限リストを生成する。  
cycle []  
take 10 (cycle [1,2,3,4,5])  
take 15 (cycle "Pika ")
- ▶ repeat: 1つの要素を受け取り、その要素のみが無限に繰り返される無限リストを生成する。  
要するに長さ1のリストをcycleにかけるとの同じ(cycle [8])  
repeat a  
take 5(repeat 8)
- ▶ replicate 単一の値からなるリストを作る。リストの長さと、複製する要素を与える。  
replicate 7 10  
replicate' :: Int -> a -> [a]  
replicate' len x = take len (repeat x) replicate' 7 10

# 練習問題③解答

## ▶ FizzBuzz

```
fizzbuzz.hs
```

```
-----
```

```
fizzBuzz :: Int -> String
```

```
fizzBuzz x
```

```
  | x `mod` 15 == 0 = "FizzBuzz"
```

```
  | x `mod`  5 == 0 = "Buzz"
```

```
  | x `mod`  3 == 0 = "Fizz"
```

```
  | otherwise = show x
```

```
-----
```

```
ghci>:l fizzbuzz
```

```
ghci>map fizzBuzz [1..20]
```

# 練習問題④解答

- ▶ `removeNonUppercase :: [Char] -> [Char]`  
実際に関数を書いてチェックしたところ、こっち。
- ▶ ちなみに、  
`removeNonUppercase :: String -> String`  
でもOKです。

# 第3章 関数の構文

» P.35～P.50



# パターンマッチ

- ▶ 手軽に値を分解したり、大きなif/elseの連鎖を避けたり、そんな場合に有効。
- ▶ パターンマッチは上から順に判定されます。

```
sayMe :: Int -> String  
sayMe 1 = "One!"  
sayMe 2 = "Two!"  
sayMe x = "Not 1 or 2"
```

```
sayMe match {  
  case 1 => "One!"  
  case 2 => "Two!"  
  case _ => "Not 1 or 2"  
}
```

# タプルのパターンマッチ

- ▶ パターンマッチはタプルでも使えます。
- ▶ ダブルの場合 `fst`, `snd` という関数で要素が分解できます。(Scalaにはない気がする)

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
addVectors (2, 3) (5, 9)
```

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)
```

# リストのパターンマッチ

## ▶ リスト内包表記のパターンマッチ

```
let xs = [(1, 3), (4, 3), (2, 4)]  
[a+b | (a, b) <- xs]  
[x*100+3 | (x, 3) <- xs]
```

## ▶ リストのパターンマッチ

```
tell :: (Show a) => [a] -> String  
tell [] = "empty"  
tell (x:[]) = "size = 1. element = " ++ show x  
tell (x:y:[]) = "size = 2. elements = " ++ show x ++ " and " ++ show y  
tell (x:y:_) = "This list is long!"
```

# asパターン

- ▶ 値をパターンに分解しつつ、パターンマッチの対象になった値自体も参照したいときに使います。

```
firstLetter :: String -> String
```

```
firstLetter "" = "Empty..."
```

```
firstLetter all@(x:xs) = "first letter of " ++ all ++ " is " ++ [x]
```

```
twoLetter :: String -> String
```

```
twoLetter "" = "Empty..."
```

```
twoLetter all@(x:y:xs) = "first letter is " ++ [x] ++ ", second letter is " ++ [y]
```

# 場合分けして、きっちりガード！

## パターン

これまでは“パターン”という構文

- ▶ 引数の構造で場合分け
- ▶ まさにパターンマッチ

## ガード

ここから“ガード”という構文

- ▶ 引数が満たす性質で場合分け
- ▶ どちらかというとif式

- ▶ 性質での場合分けの構文を“ガード”と呼ぶのは、条件を満たす場合にしか先の処理に進ませてもらえない、衛兵さんみたいだからだそうですよ。
- ▶ なにやらインデントがとっても大事だとのこと。

# ガード

- ▶ というわけでif/elseが深く長くなるのってやっぱりカッコ悪いのでガードどうですか。

```
bmiTell :: Double -> String
```

```
bmiTell bmi
```

```
| bmi <= 18.5 = “やせすぎです。”
```

```
| bmi <= 25.0 = “やったね、標準だよ！”
```

```
| bmi <= 30.0 = “ぽっちゃりって言われたい？”
```

```
| otherwise = “きみってもしかしてクジラ？あ、トドかな！？”
```

# ガード+α

- ▶ さっきの関数、やっぱりBMIの計算もしてほしいよね。ということでwhere！
- ▶ インデントをそろえればブロックはきちんと判断してくれる。

```
bmiTell :: Double -> Double -> String
```

```
bmiTell weight height
```

```
  | bmi <= 18.5 = “やせすぎです。” ++ st ++ “ kgまで太って平気！”
```

```
  | bmi <= 25.0 = “やったね、標準だよ！”
```

```
  | bmi <= 30.0 = “ぽっちゃりって言われたい？” ++ st ++ “ kgまで痩せよう”
```

```
  | otherwise = “きみってもしかしてクジラ？あ、トドかな！？”
```

```
  where bmi = weight / height ^ 2
```

```
        st = 25.0 * height ^ 2
```

# let

- ▶ let式、ここでようやくおまじないの正体が判明。
- ▶ whereと似ていますが……  
letは自身が式であり、where(節)はそうではない。
- ▶ let *binding in expression*
- ▶ “式”は“値”を必ず持つので、2つの大きな違いと言えばlet式はどこにでも書けるということ。



# let

- ▶ リスト内包表記でも使えます。

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

- ▶ ちなみにghciで `let ~ in` を1行で書くと、もう`let`の役目は終わっているということで以降参照は出来ない。

```
>let zoot a b = a + b
>zoot 1 2
```

```
>let boot a b = a + b in boot 1 2
>boot 2 3
```

# case

- ▶ さて最後はcase式。変数の指定した値に対するコードブロックを評価できます。
- ▶ 要するにコード中のどこでもパターンマッチが使える構文。
- ▶ *case expression of pattern -> result*  
*pattern -> result*  
*pattern -> result*

```
describeList :: [a] -> String
describeList ls = "The list is"
  ++ case ls of [] -> "Empty!"
             [a] -> "a singleton list."
             xs -> "a longer list."
```

# 第4章 Hello再帰!

»» P.51~P.60

# Sorry....

- ▶ 次回をお楽しみに！！
- ▶ 第3回は 12月5日 17:00～18:00 に開催します。

# 練習問題

➤➤ 練習問題①～②  
次回(12/5) ✕ 切

# 練習問題①

- ▶ パターンマッチを使って head' を実装してみてください。

## 練習問題②

- ▶ うるう年判定プログラムを作成してください。  
型シグネチャを宣言し、年数を引数とするようなプログラムとしてください。
- ▶ Ruby版のうるう年判定プログラムもさらしてください。