

はじめる関数型プログラミング

- Haskellで感じる関数型のエッセンス -

泉雅彦 @mizmarine

who are you?

- 泉 雅彦(@mizmarine)
- 株式会社VOYAGE GROUP
 - そろそろ2年目終わります
 - Pythonで広告配信書いてます
- Like
 - 🖥️ Python / Haskell
 - 🏃 脱出ゲーム



伝えたいこと

- 小さいパーツで大きな処理を作る関数型プログラミング
- 「関数を値として扱う」ニュアンスを感じてください
- 日々のプログラミングに活きたら幸いです

agenda

1. 関数型プログラミングとは
2. Haskell入門
3. パターンマッチとガード
4. 再帰を用いた繰り返し
5. 高階関数を用いた集合演算

参考書籍



- 左: すごいHaskellたのしく学ぼう！
- 右: Scala関数型デザイン&プログラミング

関数型プログラミングとは

- Hello functional programming world. -

関数型プログラミングとは

- プログラミングパラダイムの1つ
- パラダイム: 考え方.捉え方
- 例.手続き型/オブジェクト指向/etc..
- 書き方、表現方法が異なるだけ
- 最終的にできること自体は変わりません

関数型プログラミングとは

- 関数型プログラミング
 - 「純粋関数」をベースとしたプログラミングのこと
- 小さな「関数」をパーツとし、組み合わせて様々な処理を作る
 - linuxコマンドをイメージしてください
 - cat, grep, sort, ...

例: テキストファイルの操作

- 「#」から始まるデータを取得する

```
cat slide.md | grep -e '^#' | sort | head -n 3
```

```
# Features
```

```
# Haskellとは
```

```
# Haskell入門
```

何が嬉しいの？

- 1つ1つのパーツ（関数）がコンパクトになる
 - テスト性, メンテナンス性が上がる
- 再利用性が高い
 - 例: grepした後にsedの変換処理はさもう
- ではそもそも「関数」とはなにか？

関数

- 「値を渡したら」「何か返ってくる処理」のこと

addOneは 受け取った数に1を足して返す関数

```
In [1]: def addOne(x):  
...:     return x + 1  
...:
```

```
In [2]: addOne(3)
```

```
Out[2]: 4
```


副作用

- 関数呼び出し前後で「状態」を書き換えること
 - 状態次第では、同じ引数でも返り値が異なることがある
- 例
 - オブジェクトの状態更新
 - DBへの書き込み

純粋関数

- 副作用がない関数のこと
 - 同じ引数なら、いつでも同じ値を返す
 - 数学における関数と同じもの

$$f(x) = x^2$$

- 詳しくは「参照等価性」などで調べてみてください

ここまでのまとめ

- 関数型プログラミング
 - 小さい純粋関数を組み合わせて処理を記述するスタイルのこと
- 純粋関数
 - 副作用のない関数

Haskell入門

- 5minで学ぶ基礎構文 -



- 純粋関数型言語
- 以降 Haskell を用い関数型プログラミングに触れていきます

how to isntall

- macの場合

```
brew cask install haskell-platform
```

- 他は公式サイトを参考
 - <https://www.haskell.org/downloads>

基本

- 数値計算
- 論理値 / 等価性
- リテラル

関数呼び出し

- 関数名のあと空白おくと関数適用になる

-- succ: 1引数を受けて, 「次にくもの」を返す

```
Prelude> succ 2  
3
```

```
Prelude> succ 'a'  
'b'
```

2引数関数

- 複数引数の場合も同じように空白をあける

-- max: 2引数を受け, 大きい方を返す

```
Prelude> max 3 5  
5
```

```
Prelude> max 'a' 'b'  
'b'
```


関数定義

- 型定義と実装を書きます

```
-- sample.hs
```

```
-- 型定義: IntをうけてIntを返す, の意味
```

```
doubleMe :: Int -> Int
```

```
-- 実装: 左辺が引数, 右辺が返り値
```

```
doubleMe x = x * 2
```

関数利用

```
Prelude> :l sample.hs  
[1 of 1] Compiling Main           ( sample.hs, interpreted )  
Ok, modules loaded: Main.
```

```
*Main> doubleMe 3  
6
```

パターンマッチとガード

- ifを使わない場合分け -

場合分けの便利記法

- パターンマッチ
- ガード

を紹介します

パターンマッチ

- マッチしたケースにあわせて処理を行う
- イメージはこれ

$$f(x) = \begin{cases} 10 & (x = 0) \\ 50 & (x = 1) \\ x & (otherwise) \end{cases}$$

パターンマッチ

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
dayOfTheWeek otherwise = "unknown"
```

パターンマッチ

```
*Main> dayOfTheWeek 0
```

```
"Sunday"
```

```
*Main> dayOfTheWeek 4
```

```
"Thursday"
```

```
*Main> dayOfTheWeek 7
```

```
"unknown"
```

ガード

- マッチした引数の条件分けができる
- イメージはこれ

$$g(x) = \begin{cases} 2x & (x \geq 1) \\ x & (1 > x \geq 0) \\ 0 & (otherwise) \end{cases}$$

ガード

```
scoreCheck :: Int -> String
scoreCheck x
  | x == 100 = "Excellent!!"
  | x > 90   = "very good!"
  | x > 80   = "good."
  | x > 70   = "so so."
  | otherwise = "do your best >_<"
```

ガード

```
*Main> scoreCheck 100
```

```
"Excellent!!"
```

```
*Main> scoreCheck 85
```

```
"good."
```

```
*Main> scoreCheck 59
```

```
"do your best >_<"
```

例：うるう年判定してみよう

- 西暦を与えられて、うるう年かどうか判定する関数を考える
- うるう年
 - 西暦年が4で割り切れる年は閏年。
 - ただし、西暦年が100で割り切れる年は平年。
 - ただし、西暦年が400で割り切れる年は閏年。

例：うるう年判定してみよう

```
isLeapYear :: Int -> Bool
isLeapYear x
  | mod x 400 == 0 = True
  | mod x 100 == 0 = False
  | mod x 4 == 0 = True
  | otherwise = False
```

慣れない文法で疲れませんか？

一旦休憩しましょう 🍵

再帰を用いた繰り返し

- forを使わないループ処理 -

Haskellにはfor文やwhile文は存在しません

- 繰り返しを利用する場合, 再帰定義を使います
 - 高校でやった「数列」を思い出してください
 - 例: フィボナッチ数列

$$a_n = \begin{cases} a_{n-1} + a_{n-2} & (n > 2) \\ 1 & (n = 1, 2) \end{cases}$$

例：リストの合計を求めてみよう

- 配列 a_n の合計 S_n を求めてみよう

$$S_n = a_1 + a_2 + \dots + a_n$$

- 要素を一つずつ足していけばよさそう

例：リストの合計を求めてみよう

- Pythonで 手続き型っぽく書いてみます

```
def mysum(xs):  
    v = 0  
    for i in xs:  
        v += i  
    return v
```

再帰的に考えてみよう

- ポイントは基底部と再帰部を見極めること
- リストに要素がない時
 - 合計はゼロとして良さそう
- リストに要素がある時
 - 先頭要素を，先頭要素以外の合計に足せば良さそう

式にするとこんな感じ

- S_n はこう考える事ができる

$$S_n = \begin{cases} S_{n-1} + a_n & (n > 0) \\ 0 & (n = 0) \end{cases}$$

再帰的なリスト合計

```
mysum :: [Int] -> Int  
mysum [] = 0  
mysum (x:xs) = x + mysum xs
```

再帰を用いた繰り返し

- 手続き型の繰り返しで書く場合
 - ループごとの状態を意識する必要がある
- 再帰的な繰り返しで書く場合
 - 数学的な定義そのままに書けることが多い
 - 基底部と再帰部を記述するだけ

高階関数を用いた集合演算

- 同一パターンにおける処理の抽象化 -

高階関数

- 関数を引数として取る関数のこと
 - 関数型プログラミングの最初の一步！
- 様々な高階関数が関数型プログラマの武器
 - map
 - filter
 - fold

「関数を引数に取る」とは？

- こんな関数を考えてみよう
 - ある処理を引数に対し2回適用する
- $f(x)$ と x を引数として取り x に $f(x)$ を2回適用した値を返す
 - `twice` と名付ける

twice

-- 「受け取った関数」をxに2回適用する関数

```
twice :: (Int -> Int) -> Int -> Int
```

```
twice f x = f (f x)
```

```
*Main> twice succ 1
```

```
3
```

-- succ を 1に 2回適用する

twice

- succ 以外にも `Int -> Int` な関数ならなんでも渡せる

```
addThree :: Int -> Int
```

```
addThree x = x + 3
```

```
*Main> twice addThree 5
```

```
11
```

```
mulTwo :: Int -> Int
```

```
mulTwo x = x * 2
```

```
*Main> twice mulTwo 3
```

```
12
```

twice

- 共通パターン（2回繰り返す）に対し，具体的なロジックを切り替えるだけで 様々な処理が作れる
- ロジックを切り替えられる，というのが高階関数のメリット
- ロジックの抽象化
- オブジェクト指向でいう strategy パターン

無名関数(ラムダ式)

- twiceでしか利用しない関数をわざわざ定義するのが面倒？
- その場でしか利用しない関数を定義できる

```
*Main> a = \x -> x + 2
```

```
*Main> a 1
```

```
3
```

twice * 無名関数

- 無名関数を利用すると，高階関数がより便利になります

```
*Main> twice (\x -> x + 1) 1  
3
```

```
*Main> twice (\x -> x * 3) 4  
36
```

部分適用

- 部分適用
 - 複数引数の関数において、一部引数だけ適用した状態の関数
 - Haskellの関数はすべて部分適用が可能
- 詳しくは「カリー化」と合わせて調べてみてください

例: 部分適用

```
*Main> :t (+ 1)  -- 引数に1を足す関数. addOne
```

```
(+ 1) :: Num a => a -> a
```

```
*Main> (+ 1) 3
```

```
4
```

```
*Main> :t (* 3)  -- 引数を3倍する関数. mulThree
```

```
(* 3) :: Num a => a -> a
```

```
*Main> (* 3) 2
```

```
6
```


twice * 部分適用関数

- こちらも非常に高階関数と相性が良い

```
*Main> twice (+1) 1  
3
```

```
*Main> twice (*3) 4  
36
```

集合と高階関数

- x に対する処理 $f(x)$
- x の集合に対して $f(x)$ を一括で適用したい
- 「一括で適用とする」というパターンに $f(x)$ を渡す
- 高階関数として表現できる！

map

- 集合の各要素に対し, $f(x)$ による変換処理を行う高階関数
 - source

```
map :: (a -> b) -> [a] -> [b]
```

```
map _ [] = []
```

```
map f (x:xs) = f x : map f xs
```

例: map

```
*Main> a = [1,2,3,4,5]
```

```
*Main> map succ a  
[2,3,4,5,6]
```

```
*Main> map (\x -> x + 3) a  
[4,5,6,7,8]
```

```
*Main> map (*2) a  
[2,4,6,8,10]
```

filter

- 集合に対し, $f(x)$ がTrueになる要素のみにする高階関数
 - source

```
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []      = []
filter pred (x:xs)
  | pred x           = x : filter pred xs
  | otherwise        = filter pred xs
```

例: filter

```
*Main> a = [1,2,3,4,5]
```

```
*Main> filter even a -- even: 引数が偶数の時 Trueを返す関数  
[2,4]
```

```
*Main> filter (\x -> mod x 2 == 1) a -- mod: 余りを返す関数  
[1,3,5]
```

```
Main> filter (> 3) a  
[4,5]
```

mapやfilterのメリット

- ループで集合に演算する場合， 具体処理を読まないに変換なのか， フィルタなのか， そうでないのか わからない
- mapは変換， filterはフィルタと目的（パターン） がはっきりしている
- コードが読みやすくなる

fold

- リストの「畳み込み」
 - 2引数関数fと初期値acc, 走査するリストxsを受取る
 - リストを読みながらfをどんどん適用していく
- リストを左から読む場合「左畳み込み」, 逆の場合「右畳み込み」という

畳み込みのイメージ

- 関数: (+), 初期値: 0, リスト: [1, 2, 3, 4]
 - $0 + 1 = 1 \# [1, 2, 3, 4]$
 - $1 + 2 = 3 \# [2, 3, 4]$
 - $3 + 3 = 6 \# [3, 4]$
 - $6 + 4 = 10 \# [4]$
- result: 10

例: foldl

- リストの左の値から適用していく

`foldl :: (b -> a -> b) -> b -> [a] -> b`

```
Prelude> foldl (+) 0 [1,2,3,4]  
10
```

```
Prelude> foldl (*) 1 [1,2,3,4]  
24
```

例: foldr

- リストの右の値から適用していく

```
Prelude> :t foldr
```

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
Prelude> foldr (:) [] [1,2,3,4] -- (:) cons関数. リストを作成する  
[1,2,3,4]
```

関数合成

- 既存の関数を組み合わせて、新しい関数を作成できます

-- 1を足してから2倍する関数

```
addOneThenDouble' :: Int -> Int
```

```
addOneThenDouble' = doubleMe . addOne
```

-- もちろん部分適用とも併用できます

```
addOneThenDouble'' :: Int -> Int
```

```
addOneThenDouble'' = (*2) . (+1)
```

例: 冒頭のfilter処理

```
cat slide.md | grep -e '^#' | sort | head -n 3
```

- 最後にコレを今まで学んだ要素で作ってみましょう

例: 冒頭のfilter処理

```
Prelude Data.List> text <- readFile "slide.md"
Prelude Data.List> startWithSharp = (==) '#' . head
Prelude Data.List> take 3 . sort . filter startWithSharp . filter (/= "") $ text
["# title1", "# title2", "## subtitle1"]
```

高階関数を用いた集合演算

- 高階関数
 - 関数を値として扱う関数のこと
 - 共通パターンをくくり出し、処理の抽象化を行える
- 集合に対する演算として高階関数が便利
 - map, filter など 意図が明確になる

まとめ

まとめ

- Haskellを題材に関数型プログラミングの考え方を紹介した
 - 純粋関数を最小パーツとする
 - 小さなパーツを組み合わせて大きな処理を作る
- 関数をファーストクラスオブジェクトとして扱えるなら応用可
 - 日々の開発でも活貸してみてください

Haskellさらなる学びのためのキーワード

- 型コンストラクタ: List, Maybe, IO, etc..
- 独自データ型: data構文, class構文, instance構文
- 型クラス: Functor, Monoid, Monad, etc..

 Happy Functional Life 

補足

folding \Rightarrow map/filter

```
mymap :: (a -> b) -> [a] -> [b]
mymap f as = foldr step [] as
  where
    step a acc = f a : acc
```

folding \Rightarrow map/filter

```
myfilter :: (a -> Bool) -> [a] -> [a]
myfilter f as = foldr step [] as
  where
    step a acc
      | f a = a : acc
      | otherwise = acc
```

副作用なしでどうプログラム書くの？

- 副作用を持つ部分と副作用を持たない部分に切り分ける
- 副作用持つ部分をなるべく薄くしていく
- 例: データIOは副作用、処理は純粋関数として処理

数値計算

-- 演算子は優先度あり

```
Prelude> 3 + 5
```

```
8
```

```
Prelude> 50 * 100 - 4999
```

```
1
```

```
Prelude> 100 - 10 * 9
```

```
10
```

-- 負値は注意

```
Prelude> 5 * -3
```

```
interactive:4:1: error:
```

```
    Precedence parsing error
```

```
      cannot mix '*' [infixl 7] and prefix '-' [infixl 6] in the same infix expression
```

```
Prelude> 5 * (-3)
```

```
-15
```


論理演算

-- 論理演算

```
Prelude> True && False  
False
```

```
Prelude> True || False  
True
```

```
Prelude> not True  
False
```

-- 比較演算

```
Prelude> 1 == 1  
True
```

```
Prelude> 1 /= 1 -- not equal  
False
```

リテラル

-- 文字

```
Prelude> 'a'  
'a'
```

-- 文字列

```
Prelude> "hoge"  
"hoge"
```

リテラル

-- リスト

```
Prelude> []  
[]
```

```
Prelude> [1,2,3]  
[1,2,3]
```

-- listはconsの構文糖衣

```
Prelude> 1:[]  
[1]
```

```
Prelude> 1:2:3:[]  
[1,2,3]
```

-- 文字列は文字リストのsyntax sugar

```
Prelude> "masa" == ['m', 'a', 's', 'a']  
True
```

型の確認

-- :type x もしくは :t x で xの型を調べられる

```
Prelude> :type 'a'
```

```
'a' :: Char
```

```
Prelude> :t "masa"
```

```
"masa" :: [Char]
```

-- 関数の型をみることもできる

```
Prelude> :t lines
```

```
lines :: String -> [String]
```

```
Prelude> import Data.Char
```

```
Prelude Data.Char> :t toUpper
```

```
toUpper :: Char -> Char -- Char型を受けてChar型を返す関数
```

中置関数

- $f\ x\ y$ の形で適用する関数を前置関数という
- $x\ op\ y$ の形で適用する中置関数を中置関数という

-- (+) 関数

```
Prelude> 1 + 2  
3
```

型クラス

- 同じような処理の関数も、型が違ったら適用できない

```
equalInt :: Int -> Int -> Bool
equalInt x y = x == y
```

```
*Main> equalInt 1 1
True
```

```
-- 'a', 'b' は Int ではないのでerror
```

```
*Main> equalInt 'a' 'b'
```

```
interactive:4:10: error:
```

- Couldn't match expected type 'Int' with actual type 'Char'
- In the first argument of 'equalInt', namely 'a'
- In the expression: equalInt 'a' 'b'
- In an equation for 'it': it = equalInt 'a' 'b'

型クラス

- 具体的な型(Int, Char,..)が どのような性質をもつか, を示すもの
 - 同値比較ができるか, 順序比較ができるか..など
 - java や go lang における interface に近い
- 例: Eq型クラス
 - == 演算子で比較演算ができるか

型クラス

- `equalInt`を `Eq`型クラスに属する型を対象としてする

```
equalEq :: (Eq a) => a -> a -> Bool
```

```
equalEq x y = x == y
```

```
*Main> equalEq 1 1
```

```
True
```

```
*Main> equalEq 'a' 'b'
```

```
False
```


その他型クラスの例

- Ord型クラス
 - 順序比較計算($>$)ができるもの
- Enum型クラス
 - 値を列挙できるもの
- Num型クラス

例2：フィボナッチ数を求めよう

- 定義

$$a_n = \begin{cases} a_{n-1} + a_{n-2} & (n > 2) \\ 1 & (n = 1, 2) \end{cases}$$

例2：フィボナッチ数を求めよう

- Pythonで 手続き型っぽく書いてみます

```
def fib(x):  
    a_1, a_2 = 1, 1  
  
    if x == 1:  
        return a_1  
    if x == 2:  
        return a_2  
  
    v = 0  
    i = 2  
    while i < x:  
        i += 1  
        v = a_1 + a_2  
        a_1 = a_2  
        a_2 = v  
    return v
```

例2：フィボナッチ数を求めよう

- Haskellで再帰的に書いてみます

```
fib :: Int -> Int
```

```
fib 1 = 1
```

```
fib 2 = 1
```

```
fib x = fib (x-1) + fib (x-2)
```

例: 関数合成

```
-- f(x)
addOne :: Int -> Int
addOne x = x + 1

-- g(x)
doubleMe :: Int -> Int
doubleMe x = x * 2

-- h(x) = g(f(x))
addOneThenDouble :: Int -> Int
addOneThenDouble x = doubleMe (addOne x)

*Main> addOneThenDouble 4
10
*Main> 2 * (1 + 4)
10
```

関数合成とポイントフリースタイル

- 関数合成演算子を使うと よりシンプルに書けます
- ポイントフリースタイルという

```
addOneThenDouble' :: Int -> Int
addOneThenDouble' = doubleMe . addOne
```

- もちろん部分適用とも併用できます

```
addOneThenDouble'' :: Int -> Int
addOneThenDouble'' = (*2) . (+1)
```

ポイントフリースタイルで利用するfold

-- listを走査する系の処理は foldingで記述できる

```
Prelude> sum = foldl (+) 0
```

```
Prelude> sum [1,2,3,4]
```

```
10
```

```
Prelude> reverse = foldl (flip (:)) []
```

```
Prelude> reverse [1,2,3,4]
```

```
[4,3,2,1]
```

-- challenge: map / filter を foldingで書いてみよう

-- hint: foldrをつかおう