

プログラミング言語論

第10回

関数型プログラミング
Haskellの文法

Main.hs

```
main = do  
    print "Hello, World!"
```

実行結果

```
"Hello, World!"
```

- 関数名 = 処理内容
- `main` には `do` を書きます。

上の例は `do` を付けなくても実行できます。 `--` はコメントです。

```
main =  
  print "Hello, World!" -- OK
```

実行結果

```
"Hello, World!"
```

連続して出力するには `do` が必要です。

`do` がない場合、連続して出力できません。

NG

```
main =  
  print "Hello,"  
  print "World!" -- エラー
```

```
main = do  
  print "Hello,"  
  print "World!" -- OK
```

実行結果

```
"Hello,"  
"World!"
```

□ 代入

- C, C++, Javaなどの言語で変数に値を入れること

□ 束縛（英語でバインド）

- Haskellなどの関数型言語で**値を定義**すること
- C++での, `const int a=10`などはある意味バインド

□ グローバル変数のこと

```
a = 1  
b = 2  
c = a + b  
  
main = do  
    print c
```

実行結果

3

これは代入ではなく
定義

mainの前にa = 3という文
を入れるとコンパイラに
Multiple declarations of 'a'
といって怒られる（代入は
できない）

□使用箇所の下で定義します。独特な書き方です。

```
main = do
  print c
  where
    a = 1
    b = 2
    c = a + b
```

実行結果

3

字下げですが、タブだと
ghcに怒られるようです。
スペースをつかいましょう。

英語で数式を書くときに、命題を
述べてから、「ここで、」と
補足説明をするときに利用するwhere の
意味です。

- 定義してから使用します.

doあり

```
main = do
  let a = 1
      b = 2
      c = a + b
  print c
```

実行結果

3

- doがなければ最後にinが必要になります.

doなし

```
main =
  let a = 1
      b = 2
      c = a + b in
  print c
```

実行結果

3

- 数学に登場する関数, $f(x)=x+1$ や $f(1)$ の括弧がない版だとイメージしてください. C言語のreturnに相当するキーワードは使いません.

直接呼び出す場合

```
f x = x + 1  
a = f 1
```

```
main = do  
  print a
```

実行結果

2

```
f x = x + 1  
  
main = do  
  print (f 1)
```

実行結果

2

- 閉じ括弧を省略するための\$という書式があります。
\$から行末までを括弧で囲むのと同じ効果があります。

```
main = do
  print (f 1)
  print $ f 1
```

- 2つの引数を取る関数を``で囲むと中置演算子として使用できます

```
add x y = x + y

main = do
  print $ add 1 2
  print $ 1 `add` 2
```

実行結果

3
3

□ さっきの逆. 中置演算子を `()` で囲むと関数に.

```
main = do
  print $ 1 + 2
  print $ (+) 1 2
```

実行結果

3
3

```
main = do
  print $ 5 + 2
  print $ 5 - 2
  print $ 5 * 2
  print $ 5 / 2
  print $ div 5 2
  print $ mod 5 2
  print $ 5 `div` 2
  print $ 5 `mod` 2
```

実行結果

```
7
3
10
2.5
2
1
2
1
```

- / は浮動小数点
- div は商, mod は剰余

- 変数と関数は名前を小文字で始める必要があります。大文字で始めるとエラーになります。

```
A = 1  
F x = x + 1
```

エラー内容

```
src\Main.hs:1:1: Not in scope: data constructor `A'  
src\Main.hs:2:1: Not in scope: data constructor `F'
```

- ifは文ではなく、**関数**. elseが必須に.
- ※ 「等しい」は==, 「等しくない」は/=です. 後者は記号≠に由来します.

```
a = 1

main = do
    if a == 1 then print "1" else print "?"
```

実行結果

"1"

- 複数行で書く場合，ifに対してthenやelseがぶら下がっていることを示すため，thenやelseのインデントをifより右にずらす必要があります。

```
a = 1

main = do
    if a == 1
        then print "1"
        else print "?"
```

実行結果

"1"

```
foo n =
    if n < 0
        then "negative"
    else if n > 0
        then "positive"
    else "zero"
```

else ifの例

```
a = 1

main = do
  print $ if a == 1 then "1" else "?"
```

実行結果

"1"

```
f a = if a == 1 then "1" else "?"
```

```
main = do
  print $ f 0
  print $ f 1
```

実行結果

"?"

"1"

- 関数内の全体をifで切り分ける代わりに、特定の引数を指定して関数を分割定義できます。このような書き方をパターンマッチと呼びます。
- C++だとテンプレートの特殊化みたいな定義

```
f 1 = "1"  
f a = "?"  
  
main = do  
  print $ f 0  
  print $ f 1
```

実行結果

```
"?"  
"1"
```

□再帰呼び出しで作ります.

```
fact 0 = 1
fact n = n * fact (n - 1)

main = do
    print $ fact 5
```

実行結果

120

- Haskell ではループがないので、反復処理には、再帰を良く使う
- 再帰呼出しで書くんだったらC++,python や他の手続き型でも普通にできるよね？
- 手続き型言語→再帰呼出しはスタックを消費→繰り返し返して呼ぶと、いつかスタックを使い果たしてstack overflow
- Haskell → 再帰呼出しをコンパイル時にループに自動変換 (スタックを消費しない)

- 1つの関数定義に引数の条件を列挙するガードという書き方があります.

```
fact n
  | n == 0      = 1
  | otherwise = n * fact (n - 1)

main = do
  print $ fact 5
```

実行結果

120

- Haskellのリストは一様なデータ構造です.
 - 同じ型の要素を複数個持つことができます.
 - 整数のリストや文字列のリストなど
 - 複数の型からなるリストは作れません.
 - 整数と文字列の混在リストなど
- リストの定義
 - 例: [0,1,2,3,4,5]

□ リストから要素を取り出すには!!を使います。先頭の要素は0番目です。

```
main = do  
  print $ [1, 2, 3, 4, 5] !! 3
```

実行結果

4

□ 連番リストを生成する専用の書き方があります

```
main = do  
  print [1..5]
```

実行結果

```
[1,2,3,4,5]
```

□ ++によりリスト同士を結合できます。

```
main = do  
  print $ [1, 2, 3] ++ [4, 5]
```

実行結果

```
[1,2,3,4,5]
```

- :によりリストの先頭に要素を挿入できます。
- 複数の先頭要素を連ねることもできます。
- :では末尾に追加できないため++を使用します。

```
main = do  
  print $ 1:[2..5]
```

実行結果

```
[1,2,3,4,5]
```

```
main = do  
  print $ 1:2:[3..5]
```

実行結果

```
[1,2,3,4,5]
```

```
main = do  
  print $ [1..4] ++ [5]
```

実行結果

```
[1,2,3,4,5]
```

□ head

- 先頭を取り出す

□ tail

- 先頭以外の残りを取り出す

□ last

- 最後を取り出す

□ init

- 最後以外の残りを取り出す

□ 文字列は文字のリストとして扱われます。

```
main = do
  print $ "abcde"
  print $ ['a', 'b', 'c', 'd', 'e']
  print $ ['a'..'e']
  print $ 'a':"bcde"
  print $ 'a':'b':"cde"
  print $ "abc" ++ "de"
  print $ "abcde" !! 3
```

実行結果

```
"abcde"
"abcde"
"abcde"
"abcde"
"abcde"
"abcde"
'd'
```

- 引数でリストの先頭要素を取り出せます。
- $(x:xs)$ で先頭 x とその後ろ xs に分割して受け取ります。
 - xs は x が複数あるという意味で慣例的に使う
- 先頭要素は複数を連ねることもできます。
 - $_$ は後にも先にも使わない変数という意味を慣例的に表す

```
first (x:xs) = x

main = do
  print $ first [1..5]
  print $ first "abcdef"
```

実行結果

```
1
'a'
```

```
second (_,x:_) = x

main = do
  print $ second [1..5]
  print $ second "abcdef"
```

実行結果

```
2
'b'
```

- length : 長さ
- sum : 総和
- product : 総積

```
main = do  
  print $ length [1, 2, 3]
```

実行結果

3

```
main = do  
  print $ sum [1..5]
```

実行結果

15

```
main = do  
  print $ product [1..5]
```

実行結果

120

- take : 先頭のn個を抽出します
- drop : 先頭のn個を落とします。
- reverse:逆順

```
main = do
  print $ take 2 [1, 2, 3]
```

実行結果

[1,2]

```
main = do
  print $ drop 2 [1, 2, 3]
```

実行結果

[3]

```
main = do
  print $ reverse [1..5]
```

実行結果

[5,4,3,2,1]

□例

```
[x**2 | x <- [1..1000]]
```

出力

```
[1.0,4.0,9.0,16.0,25.0,36.0,49.0,64.0,81.0,100.0]
```

□フィルタリングをすることもできる

```
[x | x <- [1..1000], x `mod` 2 == 1]
```

フィルタ条件

□ リストに関数を適用する

□ 引数に**関数**とリストを取る

□ `map (+1) [1..10]`

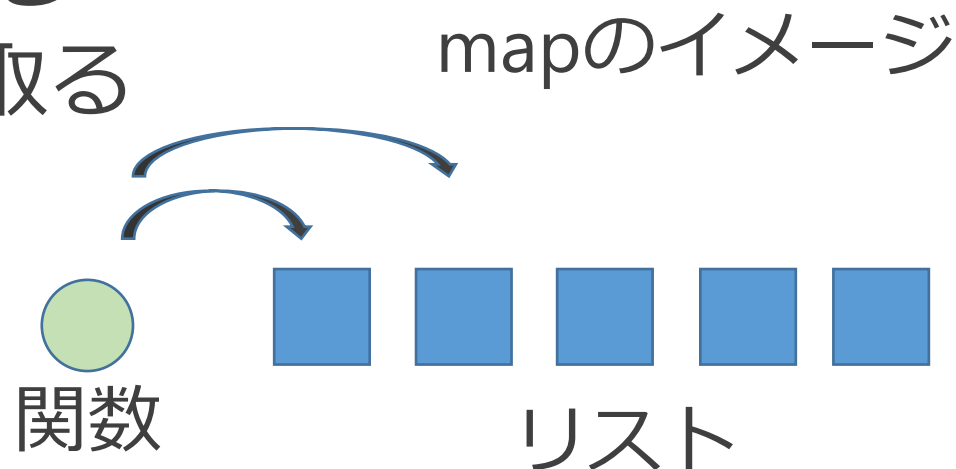
- 1 を足す

□ `map (*2) [1..10]`

- 2倍する

□ `map (\x -> x*x) [1..10]`

- 二乗する. (中はラムダ式, 無名関数)
- 環境によってはバックスラッシュの代わりに `¥` を使う



- λ 引数 \rightarrow 式

- 例 : $(\lambda x y z \rightarrow x + y + z) 1 2 3$

- mapなどの高階関数と組み合わせ使うことが多い

- 他の高階関数の例 : filter (述語が真となる要素のみを選び出す)

`filter ($\lambda x \rightarrow x \text{ `mod` } 2 == 0$) [1..10]`

出力 : [2,4,6,8,10]

- 合成関数 $f(g(x))$ は, $(f . g) x$ と書くことができる
- 関数を順に適用していくケースをスッキリ書くことができる

□ 例 :

```
f = sum.(map (¥ x->x**2))
```

```
main = do
```

```
  print $ f [1..10]
```

□ リスト[2,3,4]の先頭に1を追加し, printせよ.

レポート10- 1 （再帰とパターンマッチング）36

- フィボナッチ数列を計算するプログラムを書け.
 - 参考にならない参考. 限界を超えて書くところなるという例
 - https://qiita.com/mod_poppo/items/4f78d135bb43b7fd1743
- lengthと同じ動作をする関数my_lengthを再帰とパターンマッチングを利用して実装せよ.
- sum関数も同様にして, 再帰とパターンマッチングを利用して実装せよ.
- reverse関数も同様にせよ.

- 1 から 1 0 0 の間の 3 の倍数のみを含むリストを作成せよ(リストの内包的表記を利用のこと) .
- $1^2 + 2^2 + 3^2 + \dots + 1000^2$ を求めよ（リストの内包的表記を利用のこと） .

- （１） map関数を自分で実装したmy_mapを作成せよ。
- （２） filter関数を自分で実装したmy_filterを作成せよ。