

関数型プログラミング

第8回 関数

萩野 達也

hagino@sfc.keio.ac.jp

関数定義

```
square n = n * n
```

- 与えられた数の2乗を計算するsquare関数を定義している.
- 変数squareに2乗を計算する関数を束縛(bind)したい.
 - `a = 10`
 - 変数aに定数10を束縛する.
 - `square = ...`

高階関数

- 関数も値の一つである.
 - 引数に渡すことができる.
 - 関数の戻り値として関数が返ってくる.

```
map square [1,2,3,4,5] ⇒ [1,4,9,16,25]
```

- `map`は関数を引数に取る.
- `map`は関数を返す.
 - `(map square)`はリストを引数に取る関数.

無名関数

```
∀パターン1 パターン2 …… -> 式
```

- 関数名を与えずに関数を作ることができる.
 - 関数定義 = 関数作成 + 変数束縛
- 使用例
 - 関数の値を作成する.
 - 一度しか使わない関数に名前を与える必要はない.

```
square = ∀n -> n * n
```

```
map (∀n -> n * n) [1, 2, 3, 4, 5]
```

無名関数(つづき)

```
add x y = x + y
```

```
add =  $\forall x y \rightarrow x + y$ 
```

```
( $\forall x y \rightarrow x + y$ ) 2 3  $\Rightarrow$  ( $\forall y \rightarrow 2 + y$ ) 3  $\Rightarrow$  2 + 3  $\Rightarrow$  5
```

- パターンマッチを利用することも可能
 - ただし一つのパターンしか書くことができない

```
add2 (x, y) = x + y
```

```
add2 =  $\forall (x, y) \rightarrow x + y$ 
```

```
map ( $\forall (x, y) \rightarrow x + y$ ) [(1,11), (2,12), (3,13)]
   $\Rightarrow$  [(1+11), (2+12), (3+13)]
   $\Rightarrow$  [12,14,16]
```

関数合成

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

凡例 `f.g`

- 2つの関数を合成して新しい関数を作る
 - $(f.g) \ x = f \ (g \ x)$
 - $f.g = \forall x \rightarrow f \ (g \ x)$

```
numberOfLines :: String -> Int
numberOfLines cs = length $ lines cs
```

```
numberOfLines :: String -> Int
numberOfLines = length . lines
```

- $(\$)$ との違い
 - $(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$
 - $f \ \$ \ x = f \ x$

関数合成(つづき)

```
sortLines :: String -> String
sortLines cs = unlines $ sort $ lines cs
```

- 関数合成で書くと

```
sortLines = unlines . (sort . lines)
```

- (.) は右結合

```
sortLines = unlines . sort . lines
```

- 他の例:

```
tac :: String -> String
tac cs = unlines $ reverse $ reverse $ reverse $ lines cs

tac = unlines . reverse . reverse . reverse . lines
```

部分適用

- 関数に引数は一度に渡す必要はない
 - `addThree i j k = i + j + k`
 - 「`addThree 5`」は`addThree`に最初の引数を与えた部分適用状態
 - 残り2つの引数を与えられるのを待っている
- 部分適用
 - 関数に一部の引数を与えた状態のこと

```
addThree i j k = i + j + k
```

```
addThree 5 =  $\forall j k \rightarrow 5 + j + k$ 
```

```
(addThree 5) 6 =  $\forall k \rightarrow 5 + 6 + k$ 
```

```
((addThree 5) 6) 7 = 5 + 6 + 7
```


セクション

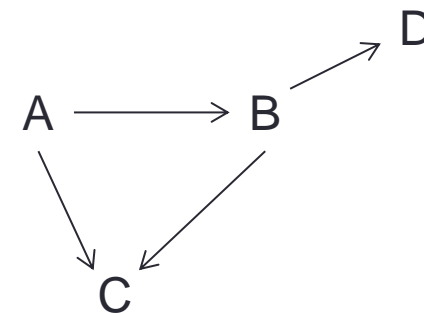
- 二項演算子の部分適用
- 例:
 - 「(+ 1)」は「+」の2つ目の引数を部分適用したもの
 - 「(1 +)」は「+」の1つ目の引数を部分適用したもの
 - (+ 1) 2 ⇒ 3
- 注意:
 - (-) 二項演算子でもあり単項演算子でもある
 - 「(- 1)」は単に「-1」を意味する
 - 「(subtract 1)」を使うこと

```
map (+ 7) [1,2,3,4,5]
      ⇒ [8,9,10,11,12]
```

```
filter (/= '¥r') "aaa¥r¥nbbb¥r¥ncccc¥r¥ndddd¥r¥neeee¥r¥n"
      ⇒ "aaa¥nbbb¥ncccc¥ndddd¥neeee¥n"
```

ポイント・フリー・スタイル

- 圏論(カテゴリー理論)
 - 対象(オブジェクト)と射(アロー)の理論
 - ポイント = 値
- ポイント・フリー・スタイル
 - 関数合成だけを使い, 値を直接参照しない



fgrep.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

ポイント・フリー・スタイルへの変換

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

- **where**句を使わない



```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter (match pattern) $ lines cs

match :: String -> String -> Bool
match pattern line = any (prefixp pattern) $ tails line

prefixp :: String -> String -> Bool
prefixp pattern line = pattern `isPrefixOf` line
```

ポイント・フリー・スタイルへの変換(つづき)

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter (match pattern) $ lines cs
```



```
fgrep pattern = unlines . filter (match pattern) . lines
```

```
prefixp :: String -> String -> Bool
prefixp pattern line = pattern `isPrefixOf` line
```



```
prefixp pattern = (pattern `isPrefixOf`)
```

```
match :: String -> String -> Bool
match pattern line = any (prefixp pattern) $ tails line
```



```
match pattern = any (prefixp pattern) . tails
```



```
match pattern = any (pattern `isPrefixOf`) . tails
```

ポイント・フリー・スタイルへの変換(つづき)

fgrep.hs

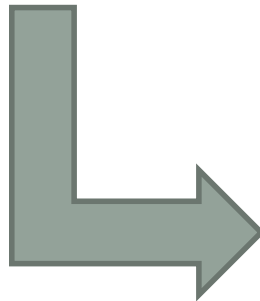
```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

ポイント・フリー・スタイル
に近づける
(まだ完全ではない)



fgrep.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern = unlines . filter (match pattern) . lines

match :: String -> String -> Bool
match pattern = any (pattern `isPrefixOf`) . tails
```

さらにポイント・フリー・スタイルへ

- `match`の2つ目の引数の値を参照しないためには、関数適用の順番を変える必要がある。

`f . g`  `(. g) $ f`

```
match pattern = any (pattern `isPrefixOf`) . tails
```

```
match pattern = any (isPrefixOf pattern) . tails
```

```
match pattern = ((any . isPrefixOf) pattern) . tails
```

```
match pattern = (. tails) $ (any . isPrefixOf) pattern
```

```
match = (. tails) . any . isPrefixOf
```

練習問題8-1

- fgrepを完全ポイントフリーに書き直さない。

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter (match pattern) $ lines cs
```

```
fgrep pattern = unlines . filter (match pattern) . lines
```

fgrep2.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep = ...
```

畳み込み関数

- 引数にリストを取る関数は、リストの再帰を使って次のように定義されることが多い。

```
f [] = v
f (x:xs) = x `op` f xs
```

- 空リストの値は「 v 」であり、そうでない場合には、先頭の要素「 x 」と残りの要素に自分自身を適用した結果を「 op 」で演算する。
- 例えば、これまでに使ってきた関数がこの形で書かれている。

```
sum [] = 0
sum (x:xs) = x + sum xs

prod [] = 1
prod (x:xs) = x * prod xs
```

- 「 v 」と「 op 」を変えるとこと様々な処理が可能になるかもしれない。
- 「 v 」と「 op 」を引数に取る汎用の関数を用意すればよい。

右結合畳み込み関数foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- 畳み込み関数foldrを使うとリストに関する関数を簡潔に書き表すことができる。

```
sum = foldr (+) 0
prod = foldr (*) 1
```

- さらに, lengthのような関数もfをうまく工夫することでfoldrで表すことができる。

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```



```
length = foldr (∀x n -> 1 + n) 0
```

練習問題8-2

- 2つのリストを結合する(++)をfoldrを使って表しなさい。

append.hs

```
(++) :: [a] -> [a] -> [a]
(++) [] ys = ys
(++) (x:xs) ys = x : ((++) xs ys)

append xs ys = foldr ....
```

- map関数をfoldrを使って表しなさい。

map2.hs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)

map2 f = foldr ....
```

- リストを反転させるreverseをfoldrを使って表しなさい。

rev.hs

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

rev = foldr ....
```

左結合の畳み込み関数foldl

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl (f v x) xs
```

- foldrは右結合の演算子に関するものであったが、同じように左結合に関する畳み込み関数foldlもある。
- 再帰しながら値を蓄積する形を取っている。
- (+) や (*) は左結合の二項演算子なのでfoldlを使う方が自然かもしれない。

```
sum = foldl (+) 0
prod = foldl (*) 1
```

- さらに, reverseについても次のように考えることができる。

```
reverse :: [a] -> [a]
reverse xs = reverse2 [] xs

reverse2 :: [a] -> [a] -> [a]
reverse2 ys [] = ys
reverse2 ys (x:xs) = reverse2 (x:ys) xs
```



```
reverse = foldl (λxs x -> x:xs) []
```

練習問題8-3

- 引数に与えられた文字列を、二進数の文字列であるとみなし、その二進数を十進数に変換して表示するプログラムを、畳み込み関数を使って書きなさい。

```
b2d.hs
```

```
import System.Environment

main = do args <- getArgs
         print $ b2d $ head args

bit::Char -> Int
bit '0' = 0
bit '1' = 1

b2d::String -> Int
b2d = ...
```

```
% ./b2d 1010
12
% ./b2d 11111100000
2016
%
```