関数型プログラミング

# 第3回　基本（2）型と高階関数

萩野　達也

hagino@sfc.keio.ac.jp

# 型と値

- 値は型ごとに分類されている
  - 型は値の集合
- Haskellは静的な型チェックを行う
  - コンパイル時に型をチェックしてくれる
  - 型が合わないとエラーになる
- Haskellは型推論を行う
  - 明示的に型を指定しなくとも推論して補ってくれる

# 基本的な型

| 型 | 意味 | 値の例 |
|---|---|---|
| `Int` | 整数値 | `-1, 0, 1, 777` |
| `Char` | 文字 | `'a', 'A', '¥n'` |
| `String` | 文字列 | `"abc", "string"` |
| `Bool` | 真偽値 | `True, False` |
| `[〇〇]` | 〇〇のリスト | `[1, 2, 3]` |

- 真偽値
  - 真が`True`，偽が`False`
- リスト
  - `[Int]` 整数のリスト型
  - `[Char]` 文字のリスト型（別名は`String`型）

# 関数の型

```
第1引数の型 -> 第2引数の型 -> ···· -> 返り値の型
```

- `lines`の型
  - `String -> [String]`
- `unlines`の型
  - `[String] -> String`
- `firstNLines n cs = unlines $ take n $ lines cs`
  - `Int -> String -> String`

# 型変数

- **length関数**
  - `length [1, 2, 3]`
  - `length ['a', 'b']`
  - `length ["abc", "def"]`
  - 色々な型のリストに適用可能
  - 多層(polymorphic)関数
  - **length関数の型**
    - `[a] -> Int`
    - **a**は型変数
- **take関数**
  - `take 3 [1, 2, 3]`
  - **take関数の型**
    - `Int -> [a] -> [a]`

ghciを使って型の調べ方

```
Prelude> :type take
take :: Int -> [a] -> [a]
```

# 型の宣言

```
変数名 :: 型
```

- 変数の型を指定する

```
関数名 :: 第1引数の型 -> 第2引数の型 -> ···· -> 返り値の型
```

- 関数の型を指定する
  - 関数の定義の時にチェックされる
  - 型推論がうまくいかないときに指定する

```
length  :: [a] -> Int
reverse :: [a] -> [a]
take    :: Int -> [a] -> [a]
words   :: String -> [String]
lines   :: String -> [String]
unlines :: [String] -> String
print   :: (Show a) => a -> IO()
putStr  :: String -> IO()
putStrln:: String -> IO()
```

# 高階関数

- 関数も値
  - 整数値などと同じように関数も値として扱うことができる
- 高階関数
  - 関数の値を引数として取る関数
  - 関数を値として返す関数
- **map関数**
  - `map :: (a -> b) -> [a] -> [b]`
  - `map square [1, 2, 3]`
    - → `[(sqaure 1), (square 2), (sequare 3)]`
    - → `[1, 4, 9]`
  - ここで `square n = n * n`
  - `map f xs`
    - リスト`xs`の各要素に`f`を適用したリストを返す

# expandコマンドの作成

expand.hs

```
main = do cs <- getContents
          putStr $ expand cs

expand :: String -> String
expand cs = map translate cs

translate :: Char -> Char
translate c = if c == '¥t' then '@' else c
```

- 上記のプログラムを入力して，実行してみなさい．
  - タブを@に置き換えるコマンド

```
C:¥> ghc expand.hs
...
C:¥> expand < USA-states.txt
AK@Alaska@Juneau
AL@Alabama@Montgomery
...
```

# if式

```
if 条件式 then 式1 else 式2
```

- C言語やjavaのif文とは少し違います
  - 3項演算子の「式1?式2:式3」と同じ
- if式の値
  - 条件式の値が**True**ならば式1の値になり，**False**ならば式2の値になる

```
translate c = if c == '¥t' then '@' else c
```

- **c**がタブだったら**'@'**で，そうでない場合には**c**
  - **translate '¥t' → '@'**
  - **translate 'a' → 'a'**
  - **translate '¥n' → '¥n'**

# expandの説明

```
expand :: String -> String
expand cs = map translate cs
```

- translateはタブ文字を@に変換する関数
  - translate :: Char -> Char
- map translate cs
  - csは文字列(Charの配列)
  - mapによってcsの各文字にtranslateを適用した結果の文字列になる
  - map translate "abc¥tdef¥n"

  → map translate ['a', 'b', 'c', '¥t', 'd', 'e', 'f', '¥n']

  → [(translate 'a'), (translate 'b'), (translate 'c'), (translate '¥t'), (translate 'd'), (translate 'e'), (translate 'f'), (translate '¥n')]

  → ['a', 'b', 'c', '@', 'd', 'e', 'f', '¥n']

  → "abc@def¥n"

# ==演算子

- ==は二項演算子

- (==) が関数名
  - `(==) :: a -> a -> Bool`
  - `x == y`
    - xとyが等しいときにTrueを返し
    - 等しくないときにFalseを返す

# if式の書き方

```
if c == '¥t' then '@' else c
```

- thenやelseの後の式が大きくなった場合には改行して書いてもかまわない.

```
if c == '¥t'
   then '@'
   else c
```

- あるいは

そろえる

```
if c == '¥t' then '@'
             else c
```

# 練習問題3－1

- 各行の先頭に'¥t'を追加するコマンドtab.hsを完成させなさい.

tab.hs

```
main = do cs <- getContents
          putStr $ unlines $ map addTab $ lines cs

addTab :: String -> String
addTab cs = ...
```

- addTab csはcsの先頭に'¥t'を追加する関数です.
  - リストを連結するには++演算子を使うことができます.
  - [1, 2, 3] ++ [4, 5] → [1,2,3,4,5]
  - "abc" ++ "def" → "abcdef"

```
C:¥> ghc tab.hs
...
C:¥> tab < USA-states.txt
        AK      Alaska   Juneau
        AL      Alabama  Montgomery
        AR      Arkansas Little Rock
...
```

# expand(バージョン2)コマンドの作成

expand2.hs

```
main = do cs <- getContents
          putStr $ expand cs

expand :: String -> String
expand cs = concat $ map expandTab cs

expandTab :: Char -> String
expandTab c = if c == '¥t' then "        " else [c]
```

- 最初のexpandではタブが1文字の@に置き換えられた
  - タブは8文字の空白と置き換えたい
  - 上記のプログラムを入力して実行してみなさい.

```
% ghc expand2.hs
...
% ./expand2 < USA-states.txt
AK        Alaska         Juneau
AL        Alabama         Montgomery
...
```

# concat

```
concat :: [[a]] -> [a]
```

- リストのリストが与えられたときに，それらのリストを連結して一つのリストにする．
  - 2重のリストを1重にする（flat化）
  - `concat [[1, 2], [3], [4, 5]]` → `[1, 2, 3, 4, 5]`
  - `concat [[1, 2], [], [3]]` → `[1, 2, 3]`
  - `concat ["ab", "c", "de"]` → `"abcde"`
  - `concat ["ab", "", "cd"]` → `"abcd"`

# expand(バージョン3)コマンドの作成

expand3.hs

```
tabStop = 8

main = do cs <- getContents
          putStr $ expand cs

expand :: String -> String
expand cs = concatMap expandTab cs

expandTab :: Char -> String
expandTab '¥t' = replicate tabStop ' '
expandTab c    = [c]
```

- expandTabをパターンマッチを使って書き直す.
- concatとmapを組み合わせたconcatMapを利用する.

```
% ghc expand3.hs
...
% ./expand3 < USA-states.txt
AK          Alaska          Juneau
AL          Alabama          Montgomery
...
```

# パターンマッチによる関数の定義

```
expandTab :: Char -> String
expandTab '¥t' = replicate tabStop ' '
expandTab c    = [c]
```

- 引数の値のパターンに応じて場合分けして書くことができる.
  - 引数の値が`'¥t'`の時には`replicate tabStop ' '`となる
  - それ以外の場合には`[c]`となる

```
関数名  第1引数のパターン  第2引数のパターン  ・・・・  =  定義1
関数名  第1引数のパターン  第2引数のパターン  ・・・・  =  定義2
関数名  第1引数のパターン  第2引数のパターン  ・・・・  =  定義3
  :          :                :                :     :
  :          :                :                :     :
```

# 練習問題3－2

lower.hs

```
main = do cs <- getContents
          putStr $ map lower cs

lower :: Char -> Char
lower 'A' = 'a'
...
lower c = ...
```

- ファイルの大文字をすべて小文字にするlowerコマンドを完成させなさい.

```
% ghc lower.hs
...
% ./lower < USA-states.txt
ak       alaska   juneau
al       alabama  montgomery
ar       arkansas little rock
...
```

# concatMap関数とreplicate関数

- **concatMap関数**
  - `concatMap :: (a -> [b]) -> [a] -> [b]`
  - concatとmapをまとめた関数
  - `concatMap f xs = concat $ map f xs`

- **replicate関数**
  - `replicate :: Int -> a -> [a]`
  - `replicate n x` = xをn個含むリストを返す
  - `replicate 3 True` → `[True, True, True]`
  - `replicate 3 77` → `[77, 77, 77]`
  - `replicate 3 'a'` → `"aaa"`
  - `replicate 0 True` → `[]`

# リストのパターンマッチ

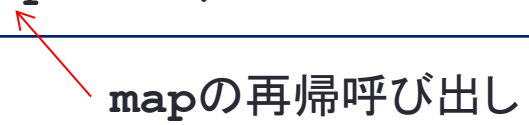
- リストは実は空リスト[]と(:)からできています.
  - `[] :: [a]`
  - `(:) :: a -> [a] -> [a]`
  - `[1, 2, 3] = 1:(2:(3:[]))`
  - `1:[2]` → `[1, 2]`
  - `5:[]` → `[5]`

- リストに対するパターンマッチでは[]の場合と(:)の場合を書きます.
  - `map :: (a -> b) -> [a] -> [b]`
  - `map f [] = []`
  - `map f (x:xs) = f x : map f xs`
  - 再帰呼び出しによって実装されています.

# 再帰呼び出し

- 関数型プログラミング言語ではfor文やwhile文のような繰り返しがないので，再帰呼び出しを使って繰り返しを実現しています．

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




- 再帰呼び出し
  - 関数の定義において自分自身を直接・間接的に呼び出すこと

# 再帰呼び出し(例)

- リストの長さを返す関数**length**
  - `length [] = 0`
  - `length (x:xs) = 1 + length xs`
- リストの数の合計を返す関数**sum**
  - `sum [] = 0`
  - `sum (x:xs) = x + sum xs`
- 2つのリストをつなげる関数**(++)**
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) [] ys = ys`
  - `(++) (x:xs) ys = x : ((++) xs ys)`
- 2重のリストを結合して1重にする関数**concat**
  - `concat :: [[a]] -> [a]`
  - `concat [] = []`
  - `concat (x:xs) = x ++ concat xs`

# 練習問題3－3

fact.hs

```
main = print $ fact 10

fact 0 = 1
fact n = ...
```

- 階乗は1からその数までの数字をかけたものである.
  - 1! = 1
  - 2! = 1×2
  - 3! = 1×2×3
  - 4! = 1×2×3×4
  - 5! = 1×2×3×4×5
- 階乗は再帰的に定義することができる.
  - 5! = 5×4!
  - 4! = 4×3!
- 上記のHaskellのプログラムは階乗を計算する関数**fact**を定義し，10!を計算するものです．完成させなさい.

# 練習問題3－4

fact2.hs

```
main = print $ ...

fact 0 = 1
fact n = ...
```

- 1から10までの階乗の結果をリストにして出力するfact2.hsを完成させなさい.

```
% ghc fact2.hs
...
% ./fact2
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

# 練習問題3－5

fact3.hs

```
main = print $ fact 10

fact n = prod [1..n]

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

- [1..n]
  - 1からnまでの数を並べたリスト
  - [1..5] → [1, 2, 3, 4, 5]
  - [1..1] → [1]
- リストの数をすべて掛け合わせるprodを完成させて10の階乗を出力しなさい.
  - prod [2, 3, 4] → 2 * 3 * 4 → 24
  - sumの定義を参考にするとよいでしょう.

# 練習問題3－6

oddline.hs

```
main = do cs <- getContents
          putStr $ unlines $ oddList $ lines cs

oddList :: [a] -> [a]
oddList [] = []
oddList (x:xs) = ...

evenList :: [a] -> [a]
evenList [] = []
evenList (x:xs) = ...
```

- 入力の奇数行のみを出力するoddline.hsを完成させなさい.
  - oddList [1, 2, 3, 4, 5] → [1, 3, 5]
  - evenList [1, 2, 3, 4, 5] → [2, 4]

```
% ghc oddline.hs
...
% ./oddline < USA-states.txt
AK      Alaska   Juneau
AR      Arkansas Little Rock
CA      California       Sacramento
...
```

# 練習問題3－7

numberline.hs

```
main = do cs <- getContents
          putStr $ unlines $ numberLine 1 $ lines cs

numberList n [] = []
numberList n (ss:xs) = ... : (numberList (n+1) xs)
```

- 行番号をつけて出力するnumberline.hsを完成させなさい.
  - numberListがそれぞれの行に番号を付与します.
    - numberLine 1 ["aaa", "bbb"] → ["1 aaa", "2 bbb"]
  - 数字を文字列にするにはshowを用います
    - show 123 → "123"

```
% ghc numberline.hs
...
% ./numberline < USA-states.txt
1 AK     Alaska   Juneau
2 AL     Alabama  Montgomery
3 AR     Arkansas Little Rock
...
```

# 関数のまとめ

| 関数名 | 適用例 | 意味 |
|---|---|---|
| `map` | `map f xs` | リスト`xs`の各要素に関数`f`を適用したリストを返す |
| `concat` | `concat xs` | リストのリスト`xs`の要素を連結して一重のリストにする |
| `concatMap` | `concatMap f xs` | `concat (map f xs)`と同じ |
| `replicate` | `replicate n x` | `x`を`n`個含むリストを返す |
| `(==)` | `x == y` | `x`と`y`が等しいときに`True`を返す |
| `(++)` | `xs ++ ys` | リスト`xs`とリスト`ys`を連結する |
| `sum` | `sum xs` | リスト`xs`の要素を合計する |
| `show` | `show x` | `x`の値も文字列にする |