

# 関数型プログラミング

## 第4回 リストと関数(2)

---

萩野 達也

hagino@sfc.keio.ac.jp

Slide URL

<https://vu5.sfc.keio.ac.jp/slide/>

# リスト結合

- `xs ++ ys`
  - リスト `xs` と `ys` の結合したリストを返す
  - $(++) :: [a] \rightarrow [a] \rightarrow [a]$
  - 左結合の演算子

append.hs

```
append xs ys = if null xs then ys
               else (head xs) : (append (tail xs) y)
```



```
xs ++ ys = if null xs then ys
           else (head xs) : (tail xs ++ ys)
```

# リストの反転

- **reverse xs**
  - リスト **xs** を反転したリストを返す
  - **reverse :: [a] -> [a]**

rev.hs

```
rev xs = if null xs then []  
        else append (rev(tail xs)) [head xs]
```



```
reverse xs = rs xs []  
  where rs xs ys = if null xs then ys  
                  else rs (tail xs) ((head xs):ys)
```

- **rs xs ys**
  - **xs** を反転させて **ys** と結合したリストを返す
  - **reverse xs == rs xs []**
  - **rs** を **xs** に関する分割統治で解く

# 多相的関数

- リストに関する関数はいろいろなリストに適用できる.
  - `[1,2,3] ++ [4,5,6] :: [Int]`
  - `[True,False] ++ [True] :: [Bool]`
- `(++) :: [a] -> [a] -> [a]`
  - `a` は型変数
  - `a` はいろいろな型に具体化 (instantiation) される
  - `a` は同じ型に具体化されなくてはならない
  - `[1,2,3] ++ [True,False]` はでは `++` の型の具体化で失敗する
- 多相的関数 (polymorphic function)
  - `a` は型変数を含む型を持つ関数
  - オーバーロードとは異なる
    - 負数の関数を同じ名前にして、状況に応じて選ぶ
    - 多相的関数は、同じ関数が複数の型の要素に適用可能

# その他のリスト関係の関数

- 前回は再帰を使って定義をしたが、もともとたくさん定義されている。
- **length xs**
  - リスト **xs** の要素数を返す.
  - `length :: [a] -> Int`

```
length xs = if null xs then 0
            else length(tail xs) + 1
```

- **sum xs**
  - リスト **xs** の要素の合計を返す.
  - `sum :: [Int] -> Int`

```
sum xs = if null xs then 0
          else (head xs) + sum(tail xs)
```

# リストの数列表記

- 一定の範囲の整数や文字などを含むリストを生成する特別な記法
  - $[1..7] == [1,2,3,4,5,6,7]$

```
[x .. y] = if x > y then [] else x:[(x + 1) .. y]
```

- 一定間隔で要素を並べることもできる
  - $[1,3..11] == [1,3,5,7,9,11]$

# 練習問題4-1

- `[1,3..11] == [1,3,5,7,9,11]` と同じ働きをする関数 `ap` を定義しなさい.
  - `[1,3..11] == ap 1 3 11`
  - `ap` は等差数列を生成する

```
ap.hs
```

```
ap x y z = if x > z then []  
           else ...
```

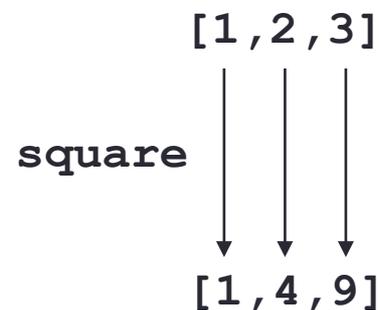
```
% stack ghci  
Prelude> :load ap.hs  
*Main> ap 1 3 11  
[1,3,4,7,9,11]  
*Main> ap 3 7 20  
[3,7,11,15,19]
```

# 高階関数

- 関数型プログラミング言語
  - 関数を通常のように扱う

- 高階関数

- 関数を引数とする関数
- 関数を返す関数



- `map f xs`

- 関数 `f` をリスト `xs` の要素それぞれに適用したリストを返す
- `map :: (a -> b) -> [a] -> [b]`

```
% stack ghci
Prelude> let square n = n * n
Prelude> map square [1,2,3]
[1,4,9]
Prelude> map square [1..10]
[1,4,9,16,25,36,49,64,81,100]
```

## 練習問題4-2

- `map` を自分で定義しなさい.
  - 名前の重なりを避けるために `maap` を定義

```
maap.hs
```

```
maap f xs = if null xs then []  
            else .....
```

- `xs` について分割統治を考える
  - `xs` が空リストのときには、何もしなくても良い
  - そうでないときには、`tail xs` に `maap` を再帰的に適用した結果と `head xs` に `f` を適用した結果をつなげる.

## 練習問題4-3

- 1 から  $n$  までの奇数の2乗の和を計算する関数 `os` を `map` などもとから定義されている関数(ここで紹介したもののみ)を使って定義しなさい。
  - $1^2 + 3^2 + 5^2 + \dots + n^2$

```
os3.hs
```

```
os n = ...  
  where square n = n * n
```

# filter 関数

- リストの中から条件に合うものだけを選ぶ.
  - `filter :: (a -> Bool) -> [a] -> [a]`
  - `filter p xs`
    - `p` は真偽値を返す関数
    - `p` が `True` となる `xs` の要素だけを選ぶ.
- `filter even [1,2,3,4,5] → [2,4]`
  - `even` は偶数かどうかを調べる関数
- `filter odd [1,2,3,4,5] → [1,3,5]`
  - `odd` は偶数かどうかを調べる関数
- 約数のリストを求める関数 `divisors`

```
divisors x = filter divisible [1..x]
  where divisible y = x `mod` y == 0
```

## 練習問題4-4

- `filter` を自分で定義しなさい.
  - 名前の重なりを避けるために `filter` を定義

```
filter.hs
```

```
filter p xs = if null xs then []  
              else ....
```

- `xs` について分割統治を考える
  - `xs` が空リストのときには、何もしなくても良い
  - そうでないときには、`tail xs` に `filter` を再帰的に適用した結果と `head xs` について `p` がどうなるかでリストを作る.

# 練習問題4-5

- $n$  以下の素数のリストを計算する関数 `primes` を定義しなさい.
  - `primes 10` → [2,3,5,7]
  - `primes 50` → [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]

```
primes.hs
```

```
primes n = filter isPrime [1..n]
  where isPrime n = ....
```

- [1..n] の中から素数であるものを選べばよい.
- 約数が2つの数が素数である.

# 文字型

- Char型
  - ユニコード文字
- 文字リテラル
  - 'a'
  - '\*'
  - ' '
- エスケープシーケンス

記述例	意味
'\t'	タブ
'\n'	改行
'\r'	復帰
'\v'	垂直タブ
'\f'	改ページ
'\a'	ベル
'\b'	バックスペース

記述例	意味
'\NNN'	10進数表記の数値NNNに対応する文字
'\oNN'	8進数表記の数値NNに対応する文字
'\xNN'	16進数表記の数値NNに対応する文字
'\^X'	コントロールX
'\''	シングルクォート
'\"'	ダブルクォート
'\\'	バックスラッシュ(あるいは\記号)

# 文字に関する関数

- 大文字・小文字の変換

関数	使用例	意味
<code>toLower</code> <code>:: Char -&gt; Char</code>	<code>toLower c</code>	文字 <code>c</code> がアルファベットの大文字のとき小文字を返す. それ以外の場合には文字 <code>c</code> 自身を返す.
<code>toUpper</code> <code>:: Char -&gt; Char</code>	<code>toUpper c</code>	文字 <code>c</code> がアルファベットの大文字のとき小文字を返す. それ以外の場合には文字 <code>c</code> 自身を返す.

- 文字と整数との変換

関数	使用例	意味
<code>ord</code> <code>:: Char -&gt; Int</code>	<code>ord c</code>	文字 <code>c</code> の文字コードを返す.
<code>chr</code> <code>:: Int -&gt; Char</code>	<code>chr n</code>	文字コードが <code>n</code> の文字を返す.

- 利用する前に `Data.Char` モジュールを `import` する必要がある.

```
import Data.Char
```

# 文字列型

- **String**型
  - 文字のリスト
  - **[Char]**型
- 文字列リテラル
  - `"string"`
    - `['s','t','r','i','n','g']`
  - `"Keio"`
    - `['K','e','i','o']`
  - `"abc\ndef\n\"hello\65\tend"`

# 文字列は文字のリスト

- リストに関する関数は文字列に適用することができる。
- 文字列の長さを求める
  - `length "Keio" → 4`
- 文字列を結合する
  - `"Keio" ++ " " ++ "SFC" → "Keio SFC"`
- 文字列を反転させる
  - `reverse "Keio" → "oieK"`
- 文字列を文字コードのリストにする
  - `map ord "Keio" → [75,101,105,111]`

## 練習問題4-6

- 与えられた文字列のアルファベット(A~Z, a~z)を1文字ずつ次の文字にずらすことで暗号化する関数 `caesar` を作成しなさい。
  - `caesar "Keio" → "Lfjp"`
  - `caesar "XYZ123" → "YZA123"`

```
caesar.hs
```

```
carsar s = ...
```

- 文字列のそれぞれを変換すればよい。
- 文字コードに変換して考えるとずらしやすいかもしれない。

```
conv ch = chr(ord ch + 1)
```

文字	コード
A	65
B	66
Z	90
a	97
b	97
z	122

- すべての文字をずらしてしまっている。
  - アルファベットだけに限定したい。

## 練習問題4-7

- 与えられた文字列が回文であるかどうかを判定する関数 `palindrome` を定義しなさい.
  - `palindrome "山本山" → True`
  - `palindrome "みがかぬかがみ" → True`

```
palindrome.hs
```

```
palindrome s = ...
```

- 文字列を反転させて, 同じ文字列かどうかを調べればよい.
- 2つのリストが等しいかどうかを調べる.

```
listEq xs ys = if null xs then null ys  
              else ...
```

- `xs` の分割統治を考える.

## 練習問題4-8

- $n$  までの Fizz Buzz の数列を作る `fizzBuzz` 関数を作りなさい。
  - $1 \sim n$  までのリストですが, 3の倍数のときは数字ではなく `Fizz`, 5の倍数のときは `Buzz`, 両方のときは `Fizz Buzz` に置き換えたリストを作りなさい。
  - `fizzBuzz 7`  $\rightarrow$  `["1", "2", "Fizz", "4", "Buzz", "Fizz", "7"]`

```
fizzBuzz.hs
```

```
fizzBuzz n = ...
```

```
main = purStr (unlines (fizzBuzz 100))
```

100までの Fizz Buzz を表示

- $1 \sim n$  までのリストのそれぞれの数字を文字列に置きかえる。
- 数字と文字列の混在のリストは作れないので, 数字も文字列に変換する。
  - `show :: Int -> String`
  - `show 123`  $\rightarrow$  `"123"`
  - `map show [1..5]`  $\rightarrow$  `["1", "2", "3", "4", "5"]`

# リストに関する関数

関数名	適用例	意味
<code>head :: [a] -&gt; a</code>	<code>head xs</code>	リスト <code>xs</code> の先頭の要素を返す
<code>tail :: [a] -&gt; [a]</code>	<code>tail xs</code>	リスト <code>xs</code> から先頭の要素を取り除いたリストを返す
<code>null :: [a] -&gt; Bool</code>	<code>null xs</code>	リスト <code>xs</code> が空リストなら <code>True</code> を返す
<code>length :: [a] -&gt; Int</code>	<code>length xs</code>	リスト <code>xs</code> の長さを返す
<code>reverse :: [a] -&gt; [a]</code>	<code>reverse xs</code>	リスト <code>xs</code> を反転させたリストを返す
<code>(++) :: [a] -&gt; [a] -&gt; [a]</code>	<code>xs ++ ys</code>	リスト <code>xs</code> とリスト <code>ys</code> を連結したリストを返す
<code>map :: (a -&gt; b) -&gt; [a] -&gt; [b]</code>	<code>map f xs</code>	リスト <code>xs</code> の要素に <code>f</code> を適用した結果のリストを返す
<code>filter :: (a -&gt; Bool) -&gt; [a] -&gt; [a]</code>	<code>filter f xs</code>	リスト <code>xs</code> の要素で <code>f</code> を適用して <code>True</code> のものだけを抽出したリストを返す

# まとめ

- リスト
  - もともと定義されている関数
  - ++
  - `reverse`
  - `length`
  - `sum`
- 高階関数
  - `map`
  - `filter`
- 文字と文字列
  - 文字列は文字のリスト