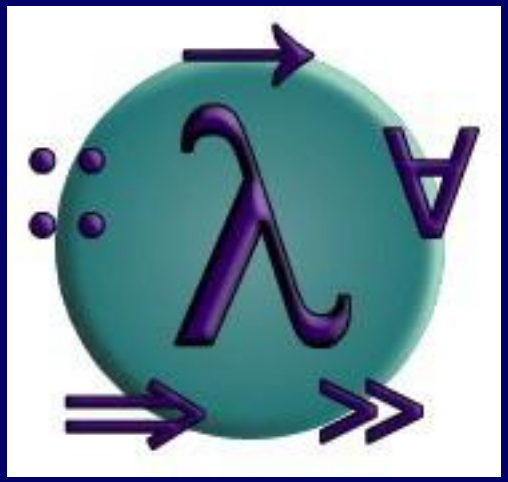
PROGRAMMING IN HASKELL
プログラミングHaskell



Chapter 5 - List Comprehensions
リスト内包表記

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

集合の内包表記(数学)

数学では、内包表記によって既存の集合から新しい集合を生成する

$$\{x^2 \mid x \in \{1...5\}\}$$

集合 {1,..., 5} の要素を x としたときの、
x^2 から成る集合 {1, 4, 9, 16, 25}

リストの内包表記(Haskell)

Haskell では、リスト内包表記によって既存のリストから新しいリストを生成する

```
[x^2 | x ← [1..5]]
```

リスト [1..5] の要素を x としたときの、
x↑2 から成るリスト [1, 4, 9, 16, 25]

注意:

- 式 x ← [1..5] を生成器といい、x の値をどのように生成するかを示す
- カンマで区切って、複数の生成器を利用できる。例:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

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

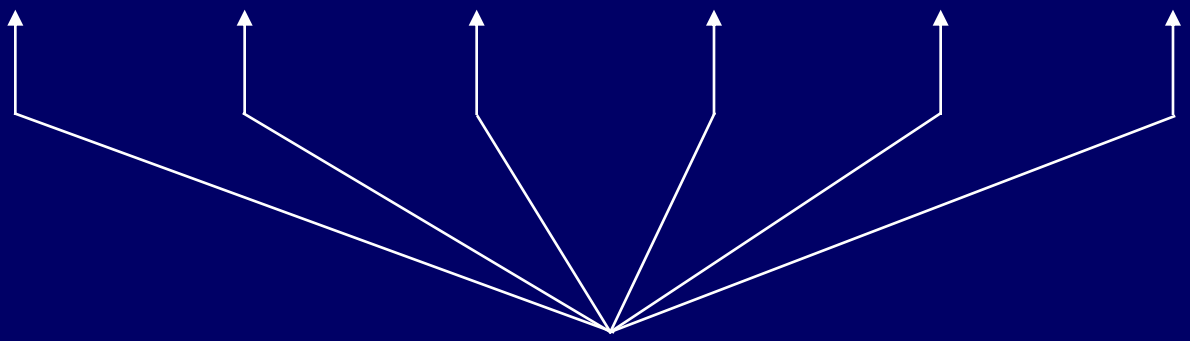
- 生成器の順番を入れ替えると、同じ要素から成るが、要素の順序が異なったリストが生成される:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

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

- 複数の生成器はループの入れ子と似ている。後方の生成器ほど、入れ子の奥にあるループのように、頻繁に値が変化する。

- For example:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

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



生成器 x ← [1,2,3] は後方なので、
タプルの x 要素がより頻繁に変化する

文脈依存の生成器

後方の生成器は、前方の生成器が設定する変数を参照できる

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

リスト [1..3] の要素を x とし、
リスト [x..3] の要素を y としたときの、
(x,y) から成るリスト
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

この方法で、リストのリストを引数とし、各要素を連結するライブラリ関数 concat を定義する:

```
concat     :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

ガード(Guards)

リスト内包表記では、ガードと呼ばれる論理式を用いて、前方の生成器が生成する値を制限する

```
[x | x ← [1..10], even x]
```

リスト [1..10] の要素を x としたときの、
偶数(even) x だけから成るリスト [2,4,6,8,10]

ガードを用いて、正の整数をその約数のリストに写像する関数を定義する:

```
factors  :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

For example:

```
> factors 15

[1,3,5,15]
```

1 と自分自身のみが約数である正の整数を素数という。関数 factors を用いて、数が素数であるか否かを判定する関数を定義する:

```
prime  :: Int → Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

ガードを用いて、指定された上限までの素数のリストを返す関数を定義する:

```
primes  :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

関数 zip

重要: Haskell で多用される

zip は、2 つのリストを引数とし、それぞれの要素を対にしたリストを返す有用な関数

```
zip :: [a] → [b] → [(a,b)]
```

For example:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

zip を用いて、リストに対して、
隣接した要素の対のリストを返す関数を定義する:

```
pairs    :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

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

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

pairs を用いて、リストが整列されているか否かを判定する関数を定義する:

```
sorted    :: Ord a ⇒ [a] → Bool
sorted xs =
   and [x ≤ y | (x,y) ← pairs xs]
```

For example:

```
and :: [Bool] -> Bool
要素全ての論理積
```

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

```
and [True,False,True]
```

zip を用いて、リスト xs において値 x が出現している全ての位置(0 始まり)をリストで返す関数を定義する:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x',i) ← zip xs [0..n], x == x']
   where n = length xs - 1
```

zip [1, 0, 0 1] [0..3]
= [(1,0), (0, 1), (0, 2), (1, 3)]

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

文字列の内包表記

文字列はダブルクオートで囲われた文字の並びであるが、内部的には文字のリストとして表現される

```
"abc" :: String
```

['a', 'b', 'c'] :: [Char] と同じ
"abc" == ['a', 'b', 'c'] は True

文字列は単なるリストの一種なので、リストを扱う多相関数は文字列も処理できる。例:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

同様に、文字列を扱う関数を定義するのにリスト内包表記が使える。文字列中の小文字を数える例:

```
lowers    :: String → Int
lowers xs =
   length [x | x ← xs, isLower x]
```

For example:

import Data.Char をファイルの先頭に追加すること

```
> lowers "Haskell"

6
```

まとめ(5章)

- リスト内包表記: 既存のリストから新しいリストを生成
  - 生成器(値の生成方法を示す): x ← [1..5]
  - [x^2 | x ← [1..5]] はリスト [1, 4, 9, 16, 25]
- 文脈依存の生成器
  - 後方の生成器が、前方の生成器が設定する変数を参照
- ガード: 前方の生成器が生成する値を制限
  - [x | x ← [1..10], even x] はリスト [2, 4, 6, 8, 10]
- 重要な役割を果たす zip :: [a] → [b] → [(a, b)]

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

- 文字列 String は文字のリスト [Char]
  - リストを扱う多相関数で文字列処理: length, take, drop, ...

問題

- 5.5節のシーザー暗号に関するプログラムの全体構造と動作を簡潔に説明せよ。カイ二乗検定は既知として良い。分量は A4 1枚以内とする。