プログラミング Haskell

第5章 リスト内包表記

となか (id:ftnk)

2010-04-24

agenda

- 内包表記
  - リスト内包
    - ガード
  - 関数 zip
  - 文字列の内包表記
- 応用
  - シーザー暗号

まずはいつもの

- リスト内包の解析結果
  - 大阪のおいしい水: 51%
  - 祝福: 34%
  - 濃硫酸: 7%
  - ミスリル: 4%
  - 成功の鍵: 4%

内包

内包（comprehension）は，集合論の「内包公理」に由来.

内包公理では，ある性質を満たす値を選択することで集合を作成する.

数学の内包表現

既存の集合から新しい集合を生成する.

例

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

既存の集合{1,2,3,4,5}から，それぞれを2乗した集合{1,4,9,16,25}を生成する.

リスト内包とは

リストの内包表記

既存のリストから新しいリストを生成する.

$$[x \uparrow 2 \mid x \leftarrow [1..5]]$$

実際の入力

```
[x^2| x <- [1..5]]
```


生成器

```
[x^2| x <- [1..5]]
```

の

```
x <- [1..5]
```

を生成器という.

生成器の列挙

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

- x <- [1,2,3]
- y <- [4,5]

2つの生成器をカンマで区切って列挙

生成器の列挙における順番

生成器の順番によって，生成される要素の順番が変わる．

```
> [(x,y)| x <- [1..3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

> [(x,y)|y <- [4,5], x <- [1..3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```


前の生成器の変数を利用

複数の生成器を利用するとき，後ろの生成器は前の生成器が使う変数を利用できる．

利用例 1: 重複のない順列

(1,2), (2,1) などを同一視.

```
> [(x,y)|x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

- x = 1 のとき y <- [1..3]
- x = 2 のとき y <- [2..3]
- x = 3 のとき y <- [3..3]

利用例 2: **concat**

concat

リストを連結するライブラリ関数

```
> concat [[1,2],[3,4,5],[6,7,8,9]]
[1,2,3,4,5,6,7,8,9]
```


利用例 2: **concat**

```
concat      :: [(a,b)] -> [a]
concat xss = [x| xs <- xss, x <- xs]
```

1. リストのリスト xss からリストをとりだす
2. リスト xs から要素 x をとりだす

利用例 2: **concat**

```
concat xss = [x| xs <- xss, x <- xs]
> concat [[1,2],[3,4,5],[6,7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

1. xs として [1,2] をとりだす
2. [1.2] から要素 1 をとりだす

疑問

xs

リスト

x の set（集合）だから xs?

xss

リストのリスト

x の set の set で xss?

例題

2 つのサイコロを振ったときの出目の組み合わせの数を，リスト内包を使って求めよ．

(1,2) と (2,1) などは同一視しない．

答え

リスト内包で全ての組み合わせを求め，そのサイズを求めればよい．

```
> length [(x,y)|x <- [1..6], y <- [1..6]]
36
```


ワイルドカード

- リストの要素の一部を捨てる
- カウンタ

リストの要素の一部を捨てる

組のリスト [(1,2), (3,4), ...] から各組の先頭の要素を取り出す（先頭以外の要素を捨てる）.

```
firsts    :: [(a,b)] -> [a]
firsts ps :: [x | (x,_) <- ps]
```


例題

組のリスト [(1,2), (3,4), ...] から各組の2番目の要素を取り出す seconds を定義せよ.

答え

```
seconds     :: [(a,b)] -> [b]
seconds ps = [x | (_,x) <- ps]
```


カウンタ

ライブラリ関数 length

```
length     :: [a] -> Int
length xs = sum [1| _ <- xs]
```

既存のリストの要素を捨て，生成するリストの要素として常に 1 を返す．

練習問題

休憩 1

ガーード

ガード

「ガードは，前方の生成器で生成された値を間引く」

ガードを評価した値が

- True -> 残す
- False -> 捨てる

ガードの例：約数

```
factors   :: Int -> [Int]
factors n = [x| x <- [1..n], n `mod` x == 0]
```

生成器のうしろにある「n `mod` x == 0」がガード.

n = 3 のとき，[1,2,3] の各要素のうち，3 を割って余りのないもののリストを返す.

ガードの例：素数の判定

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

factors で返ってくる約数が，1 と 自分自身（n）のみを要素として持つリストかチェック．

遅延評価

```
> prime 2147483646   # 2^31 - 2
> prime 2147483647   # 2^31 - 1
```

factors を最後まで評価しきらない（遅延する）.

『factors n == [1,n]』にマッチしない時点で，prime の評価は終了.

ガードの例：素数

prime で素数の判別をして，与えられた数以下の素数をリストアップ.

```
primes   :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

prime x がガード.

ガードの例：**find**

- キーと値の組からなるリストを検索
- 引数とキーを比較し，一致したら値を返す

```
find      :: Eq a => a ->  [(a,b)] -> [b]
find k t = [v | (k',v) <- t, k == k']
```


関数 zip

特殊なリスト内包

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

- 2つのリストをとる
  - 1つだけや，3つ以上ではダメ
- 対応する要素を組にして1つのリストを生成

関数 **zip** とリスト内包

関数 zip とリスト内包を組み合わせると便利なことが多いらしい.

準備として，隣り合う要素の組をリストにして返す pairs を定義する.

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

リストの要素がソートされているか

```
sorted    :: Ord a => [a] -> Bool
sorted xs = and [x <= y|(x,y) <- pairs xs]
```

リスト内包で，各隣り合う要素の大小を比較.

ここでも遅延評価.

リスト上での位置

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

生成器で各要素に index をつける.
あとは find と同じ.

文字列の内包表記

on GHC

```
> :type "string"
"string" :: [Char]
```

on Hugs

```
> :type "string"
"string" :: String
```

文字列の内包表記

文字列は Char のリスト.

ということは,

リスト内包が使える!

以上!!!!!

例

lowers

```
lowers     :: String -> Int
lowers xs = length [x | x <- xs, isLower x]
```

count

```
count  :: Char -> String -> Int
count x xs = length [x'|x' <- xs, x == x']
```


練習問題
3～6

休憩 2

シーザー暗号

文字列の暗号.

方法

アルファベットを 3 文字ずらすだけ.

```
"abc" -> "def"
"haskell is fun" -> "kdvnhoo lv ixq"
```


ROT13

ずらす文字数は 3 以外でもよい.

13 文字ずらす ROT13 が有名.

(ROTate by 13 places)

今回の仕様

- 変換する文字は小文字のみ
- ずらす文字数は任意

暗号化

手順

1. 文字を数値（0～25）に変換
2. 数値を ｎ ずらす
3. 数値を文字に変換

文字を数値に変換 **let2int**

```
let2int   :: Char -> Int
let2int c = ord c - ord 'a'
```

ord

文字を Unicode のコードポイントへ変換

数値を文字に変換 **int2let**

```
int2let   :: Int -> Char
int2let n = chr (ord 'a' + n)
```

chr

Unicode のコードポイントを文字へ変換

ord と char

ord と char は Data.Char に含まれるので，「import Data.Char」が必要

- Hugs の shell 上では import が使えない(？)
  - :load /path/to/Data/Char.hs
    - /usr/lib64/hugs/packages/base/Data/Char.hs

n 文字ずらす shift

```
shift                   :: Int -> Char -> Char
shift n c | isLower c = int2let ((let2int c + n) `mod` 26)
          | otherwise = c
```

4章の復習．ガード付きの等式．

暗号化 **encode**

```
encode       :: Int -> String -> String
encode n xs = [shift n x | x <- xs]
```

- 一文字ずつとりだして shift
- 復号化は (-n) 文字ずらせばいいので，復号化用の関数を定義する必要はない

ずらした文字数がわからない場合はどうする？

crack!!

必要なもの

- 文字の出現頻度表
  - 英文における出現頻度表
    - 期待頻度
  - 対象となる分における出現頻度表
    - 観測頻度
- カイ二乗検定
  - 期待頻度と観測頻度の比較

英文における出現頻度表

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0,
         6.1, 7.0, 0.2, 0.8, 4.0, 2.4, 6.7,
         7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8,
         1.0, 2.4, 0.2, 2.0, 0.1]
```


観測頻度の準備

```
percent      :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

fromIntegral は，整数を浮動小数点数に変換するライブラリ関数.

観測頻度

```
freqs    :: String -> [Float]
freqs xs = [percent (count x xs) n| x <- ['a'..'z']]
           where n = lowers xs
```

'a'から'z'の各文字について，「出現数 / 文字列中の小文字数」で頻度を計算.

カイ二乗検定

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

- 値が小さいほど2つのリストは似ている
  - n: リストの長さ
  - os: 観測頻度
  - es: 期待頻度

カイ二乗検定の定義

```
chisqr        :: [Float] -> [Float] -> Float
chisqt os es = sum [((o - e)^2) / e|(o,e) <- zip os es]
```


rotate

リストの要素を n だけ左に回転させる関数.

- リストの先頭は末尾に接続していると考える
- n は 0 以上で，リストの長さより小さい

```
rotate      :: Int -> [a] -> [a]
rotate n xs = drop n xs ++
```

以上で道具は揃ったので **crack**

方法

- 文字の出現頻度を求める
- 出現頻度を回転させながら，出現頻度を検定
- 検定の結果が最も小さい時のシフト数 n を返す
- 復号

出現頻度の回転

```
#       a     b     c     d     e     f     g
es = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0]
os = [2.2, 2.0, 8.2, 1.5, 2.8, 4.3, 12.7]
```

rotate 2

```
#       a     b     c     d     e     f     g
es = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0]
os = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0]
```


crack

```
crack    :: String -> String
crack xs = encode (-factor) xs
  where
    factor = head (posisions (minimum chitab) chitab)
    chitab = [chisqr (rotate n table') table | n <- [0..25]]
    table' = freqs xs
```


練習問題

以上，お疲れ様でした．