计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2023年09～12月

Adapted from Graham's Lecture slides

第5章：List Comprehension

主要知识点：

Generators、Guards, String Comprehension

Set Comprehensions

In mathematics, the *set comprehension* notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1,2,3,4,5\}\}$$

List Comprehensions

In Haskell, a *similar comprehension* notation can be used to construct new lists from old lists.

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

=

```
[1, 4, 9, 16, 25]
```

List Comprehensions

- The expression `x <- [1..5]` is called a generator, as it states how to generate values for x.
- Comprehensions can have ***multiple*** generators, separated by commas. For example:

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

=

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

List Comprehensions

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

=

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

- Changing the ***order*** of the generators changes the order of the elements in the final list:

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

=

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

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

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

=

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

Dependant Generators

- Using a dependant generator we can define the library function that *concatenates* a list of lists:

```haskell
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

program — ghc-9.4.2 -B/Users/nrutas/.ghcu...

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

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
```

```
[2, 4, 6, 8, 10]
```

Guards

✳ Using a guard we can define a function that maps a positive integer to its list of factors:

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

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4.2/lib/ghc-9.4.2/lib --i...

```
ghci>
ghci>
ghci> factors 1000
[1,2,4,5,8,10,20,25,40,50,100,125,200,250,500,1000]
ghci>
```

Guards

* A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

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

program — ghc-9.4.2 -...

```
ghci>
ghci> prime 72
False
ghci> prime 71
True
ghci> prime 127
True
ghci>
```

Guards

- A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

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

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4.2/lib/ghc-9.4.2/lib --intera...

```
ghci>
ghci> primes 70
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67]
ghci>
ghci>
```

The Zip Function

- A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] -> [b] -> [(a,b)]
zip      []        _     = []
zip      _        []     = []
zip (a:as) (b:bs)   = (a,b) : zip as bs
```

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4....

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

The Zip Function

- Using zip, we can define a function returns the list of all pairs of adjacent elements from a list:

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

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4.2/lib/ghc-9.4.2/lib --interactive...

```
ghci>
ghci>
ghci> pairs [1..10]
[(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,10)]
ghci>
```

The Zip Function

- Using pairs, we can define a function that decides if the elements in a list are sorted:

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

program — ghc-9.4.2 -B...

```
ghci>
ghci> sorted [1..10]
True
ghci> sorted [1,3,2,4]
False
ghci>
```

The Zip Function

✳ Using zip, we can define a function that returns the list of all positions of a value in a list:

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

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9....

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

String Comprehensions

- A string literal is a sequence of characters enclosed in double quotes.
- Internally, strings are represented as lists of characters.

```
"abc" :: String
        =
['a','b','c','d',] :: [Char]
```

String Comprehensions

- Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings.

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4....

```
ghci>
ghci> length "abcde"
5
ghci>
ghci> length "abcde"
5
ghci> take 3 "abcde"
"abc"
ghci> zip "abcd" [1,2,3,4]
[('a',1),('b',2),('c',3),('d',4)]
ghci>
```

String Comprehensions

✳ Similarly, list comprehensions can also be used to define functions on strings, such counting how many times a character occurs in a string.

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

program — ghc-9.4.2 -B/Users/nrutas/.gh...

```
ghci>
ghci>
ghci> count 'g' "yanglegeyang"
3
ghci>
```

凯撒加密问题

- To encode a string, Caesar simply replaced each letter in the string by the letter three places further down in the alphabet, wrapping around at the end of the alphabet.

program — ghc-9.4.2 -B/Users/nrutas/.ghcup/g...

```
ghci>
ghci> :type encode
encode :: Int -> String -> String
ghci>
ghci> encode 3 "haskell is fun"
"kdvnhoo lv ixq"
ghci>
ghci> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
ghci>
ghci> :type crack
crack :: String -> String
ghci>
ghci> crack "kdvnhoo lv ixq"
"haskell is fun"
ghci>
```

加密 / encode

```haskell
import Data.Char(ord, chr, isLower)

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

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

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

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

ord 和 chr 是模块 Data.Char 中定义的函数

- ord :: Char -> Int 将字符转换为编码值
- chr :: Int -> Char 将编码值转换为字符

解密 / crack

- The key to cracking the Caesar cipher is the observation that some letters are used more frequently than others in English text.

```
table :: [Float]
table = [8.1, 1.5, 2.8, 4.2, 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, 9.0, 2.8, 1.0,  2.4, 0.2, 2.0, 0.1]
```

A standard method for comparing a list of observed frequencies *os* with a list of expected frequencies *es* is the *chi-square statistic*, defined by the following summation in which n denotes the length of the two lists, and xs_i denotes the ith element of a list xs counting from zero:

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

解密 / crack

```haskell
crack :: String -> String
crack xs = encode (-factor) xs
  where
    -- minimum: exported by Prelude
    factor = position (minimum chitab) chitab

    -- 计算每种加密偏移量下的chisqr
    chitab = [chisqr (rotate n table') table | n <- [0..25]]

    -- 计算密文中字母的出现频率
    table' = freqs xs

freqs :: String -> [Float]

chisqr :: [Float] -> [Float] -> Float
```

作业

作业

5-1 请给出凯撒解密函数的完整定义:

```
crack :: String -> String
```

(仅考虑“明文中仅包含小写字母和空格”的情况)

作业

5-2 A triple (x,y,z) of positive integers is called pythagorean, if $x^2 + y^2 = z^2$.
Using a list comprehension, define a function

```
pyths :: Int -> [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
ghci> pyths 5
[(3,4,5),(4,3,5)]
```

作业

5-3 A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself.

Using a list comprehension, define a function

```
perfects :: Int -> [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
ghci> perfects 500
[6,28,496]
```

作业

5-4 The scalar product of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

Using a list comprehension, define a function that returns the scalar product of two lists.

第5章：List Comprehension

就到这里吧