Programmation Fonctionnelle
Compréhension sur les listes

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

12 septembre 2013

Plan

Plan

Compréhensions ensembliste

En mathématiques, la compréhension est utilisée pour construire des nouveaux ensembles à partir d'ensembles donnés.

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

> *L'ensemble* $\{1, 4, 9, 16, 25\}$ *des tous les nombres* x^2 *tels que* x *est un élément de* $\{1 \ldots 5\}$.

Compréhensions sur les listes

En Haskell, une notation similaire est utilisée pour construire des nouvelles listes à partir de listes données.

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

La liste `[1,4,9,16,25]` de tous les nombres x^2 tels que x est un élément de la liste `[1..5]`.

Les générateurs I

Remarques :

- L'expression `x <- [1..5]` est appelée « *générateur* », car elle explique comment engendrer les valeurs de `x`.
- La compréhension peut avoir des *générateurs multiples*, séparés par des virgules. Par exemple :

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

- Si on change l'ordre des générateurs, on change l'ordre des éléments dans la liste engendrée :

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

Les générateurs II

- Les générateurs multiples sont similaires à des boucles imbriquées ; les derniers générateurs correspondent à des boucles plus imbriquées, leurs variables changent plus fréquemment.

Par exemple :

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

> *`x <- [1,2,3]` est le dernier générateur, la valeur de `x` de chaque couple change plus fréquemment.*

Générateurs dépendants

Les générateurs qui suivent peuvent contenir des variables introduites par les générateurs précédents.

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

La liste `[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]` *de toutes les couples de nombres* `(x,y)` *tels que* `x,y` *sont des éléments de la liste* `[1..3]` *et* `x < y`.

Exemple avec les générateurs dépendants

En utilisant un générateur dépendant, nous pouvons définir une fonction qui concatène une liste de listes :

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

Par exemple :

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

Filtres (gardes)

La compréhension sur les listes peut utiliser des filtres pour restreindre les valeurs produits par les générateurs précédents.

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

> *La liste* `[2,4,6,8,10]` *de tous les nombres* x *tels que* x *est un élément de la liste* `[1..10]` *et* x *est pair.*

Example avec les filtres

En utilisant les filtres, on peut définir une fonction qui envoie un entier positif vers la liste des ses facteurs :

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

Par exemple :

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

Un entier positif est *premier* si ses uniques facteurs sont 1 et lui même.

En utilisant `factors`, nous pouvons définir une fonction qui décide si un nombre est premier :

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

Par exemple :

```
> prime 15
False
> prime 7
True
```

Avec les filtres, nous pouvons définir une fonction qui retourne la liste de tous les premiers jusqu'à une limite donnée :

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

Par exemple :

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

Plan

La fonction `zip`

Une fonction importante de `Prelude` est `zip` qui mappe deux listes vers une liste de couples d'éléments correspondant.

```
zip :: [a] -> [b] -> [(a,b)]
```

Par exemple :

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

Avec `zip` nous pouvons définir une fonction qui retourne la liste de toutes les paires d'éléments adjacents d'une liste :

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

Par exemple :

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

Avec `pairs`, nous pouvons définir une fonction qui décide si une liste est ordonnée :

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

Par exemple :

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

Avec `zip` nous pouvons définir une fonction qui retourne la liste de toutes les positions d'un valeur dans une liste :

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

Par exemple :

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

Compréhension sur les chaînes de caractères

Une chaîne de caractères est une séquence de caractères simples inclus entre doubles guillemets.

Par ailleurs, une chaîne est représentée, internement, comme une liste de caractères.

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

signifie

```
['a','b','c'] :: [Char]
```

Les chaînes étant un type spécial de listes, toute fonction polymorphe qui opère sur les listes peut de même opérer sur les chaînes.

Par exemple :

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

De façon semblable, la compréhension peut opérer sur les chaînes de caractères.

Par exemple : une fonction qui compte le nombre de lettres en minuscule dans une chaîne :

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

```
> lowers "Haskell"
6
```

Exercices I

1. Un triplet (x, y, z) d'entiers positifs est dit *de Pythagore* si $x^2 + y^2 = z^2$.
   Définissez, en utilisant la compréhension, une fonction

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

   qui envoie un entier n vers la listes de tous les triplets de Pythagore avec composantes dans `[1..n]`. Par exemple :

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

2. Un nombre positif est *parfait* s'il est la somme de tous ses facteurs, sauf lui même. En utilisant la compréhension, définissez une fonction

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

Exercices II

qui retourne la liste de tous les nombres parfaits, jusqu'à la limite passée en paramètre.
Par exemple :

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

3. Le produit scalaire de deux listes d'entiers *xs* et *ys* de longueur *n* est la somme des produits des entiers correspondants :

$$xs \cdot ys = \sum_{i=0}^{n-1} xs_i * ys_i \,.$$

Utilisant la compréhension, définissez une fonction qui retourne le produit scalaire de deux listes.