Compréhension

Nous explorons aujourd'hui la puissance des `for`-expressions, une notation concise pour le parcours et la génération de collections.

1 Requêtes dans une base de données

On suppose la définition d'une petite base de données de livres :

```
case class Book(title: String, authors: String*)

val books: List[Book] = List(
  Book("Principles of Compiler Design", "Aho, Alfred", "Ullman, Jeffrey"),
  Book("Programming in Modula-2", "Wirth, Niklaus"),
  Book("Elements of ML Programming", "Ullman, Jeffrey"),
  Book("The Java Language Specification", "Gosling, James",
       "Joy, Bill", "Steele, Guy", "Bracha, Gilad"),
  Book("Structure and Interpretation of Computer Programs",
       "Abelson, Harold", "Sussman, Gerald J.")
)
```

Exprimer chacune des requêtes suivantes sous la forme d'une `for`-expression :

1. Tous les titres de livre de la bibliothèque ;
2. Les livres dont le titre contient "Program" ;
3. Les titres des livres à un seul auteur ;
4. L'ensemble de toutes les paires (t, n) si le livre de titre t a n auteurs
5. L'ensemble des auteurs de la bibliothèque ;
6. L'ensemble des paires (t, a) telles que l'auteur a a écrit le livre de titre t.
7. Les auteurs qui ont écrit plusieurs livres ;
8. Les co-auteurs de Jeffrey Ullman ;

2 Le problème des n dames

On veut résoudre de façon concise grâce aux `for`-expressions le problème des N dames. Il consiste à trouver tous les placements possibles de N dames sur un échiquier de taille $N \times N$ sans que les dame ne puissent se menacer mutuellement. On rappelle qu'aux échecs une dame menace toute pièce située sur la même ligne, colonne ou diagonale qu'elle.

On représente un placement de k dames par une liste de k coordonnées où placer les dames sur l'échiquier :

```
case class Queen(x: Int, y: Int)  // x, y < N
type Placement = List[Queen]
```

Si $k = N$, alors le placement est une solution à notre problème.

1. Definir une fonction `def inCheck(q1: Queen, q2: Queen)` qui renvoit `true` si les dames passées en argument se menacent. Définir une fonction `def isSafe(queen: Queen, p: Placement)` qui renvoit `true` si `queen` ne sera pas menacé si ajoutée au le placement `p`.
2. Notez qu'une solution au problème aura forcément une dame par ligne de l'échiquier. Nous allons donc sans perte de généralité placer les dames sur chaque ligne k successivement. Pour chaque ligne nous cherchons à énumérer *tous* les placements valides. Raisonnons par induction sur k :

   Si $k = 0$ trivial (rien à faire) ;

   Si $k = k' + 1$ on a un placement correct `p` de k' dames sur les k' premières lignes ; on peut placer nouvelle dame sur la ligne $k' + 1$ seulement si sa position est sûre vis-à-vis de `p` (fonction `isSafe`).

   Définir une fonction `def placeQueens(n: Int, k: Int): List[Placement]` qui renvoit la liste de tous les placements possibles de `k` dames sur les `k` premières lignes d'un échiquier `n` × `n`. Son corps utilisera une `for`-expression.
3. Définir une fonction `def queens(n: Int): List[Placement]` qui renvoit l'ensemble de tous les placements valides de `n` dames sur un échiquier `n` × `n`.
4. Comment auriez-vous écrit 2. sans `for`-expressions ?

3 Traduction de `for`-expressions

1. Traduire sur papier les `for`-expressions suivantes en expressions d'ordre supérieur utilisant les méthodes `map`, `flatMap`, `filter` et `foreach` :
   (a) `for (x <- List(1,2,3) yield x+1`
   (b) `for (i <- 1 to 10) println(i)`
   (c) `for (i <- 1 to 10) yield (i, i/2)`
   (d) `for (x <- List(1,2); y <- List(3,4)) yield (x, y)`
   (e) `for (i <- 1 to 10; n=i/2 if n*2==i) yield n`
2. Inversement, implémenter à l'aide de `for`-expressions les fonctions :
   (a) `def map[A, B](xs: List[A], f: A => B): List[B]`
   (b) `def flatMap[A, B](xs: List[A], f: A => List[B]): List[B]`
   (c) `def filter[A](xs: List[A], p: A => Boolean): List[A]`
   (d) `def foreach[A](xs:List[A], b: A => Unit): Unit`

4 Compréhension et monades

Les `for`-expressions ne sont que du sucre syntaxique pour des expressions à base des combinateurs `map`, `flatMap`, `foreach`, etc. On peut donc les utiliser, en plus des listes, sur n'importe quel type qui contient ces méthodes, par exemple `Option` ou `Future`. On appelle ces types des *monades*.

1. Ré-implémentez les fonctions du TP 5 (`List` et `Option`) en utilisant une ou plusieurs `for`-expressions :

```
def combine[A,B](x: Option[A], y: Option[B]): Option[(A,B)]
def extract[A](z: List[Option[A]]): List[A]
def split[A,B](z: Option[(A,B)]): (Option[A],Option[B])
```

2. Ré-implémentez les fonctions du TP 6 (`Future`)) en utilisant une ou plusieurs `for`-expressions :

   ```
   def map [A,B](f: A => B, z: List[Future[A]]): List[Future[B]]
   def filter[A](p: A => Boolean, z: List[Future[A]]): List[Future[Option[A]]]
   def collect [A](z : List[Future[A]]): Future[List[A]]
   ```

3. Implémentez la monade identité, qui n'a aucun effet :

   ```
   class Id[A](val x:A) {
     def map[B](f:A=>B): Id[B] = new Id(f(this.x))
     def flatMap[B](f:A=>Id[B]): Id[B] = f(this.x)
   }
   ```

   et substituez `Id` à `Future` dans la question précédente. Notez qu'on se ramène alors au tri récursif.

4. Qu'obtient-on si on utilise la monade `List` à la place ?