

La notation For

Les fonctions d'ordre supérieur telles que *map*, *flatMap* ou *filter* fournissent des constructions puissantes pour manipuler les listes.

Mais parfois le niveau d'abstraction requis par ces fonctions rend le programme difficile à comprendre.

Dans ce cas, la notation **for** de Scala peut être utile.

Exemple : Soit une liste *persons* de personnes, avec champs *name* et *age*. Pour imprimer les noms des personnes âgées de plus de 20 ans, on écrira :

```
for { val p ← persons; p.age > 20 } yield p.name
```

qui est équivalent à :

```
persons filter (p ⇒ p.age > 20) map (p ⇒ p.name)
```

L'expression *for* est similaire à la boucle *for* des langages impératifs, sauf qu'elle construit une liste des résultats de toutes les itérations.

Syntaxe du For

Une expression for est de la forme

for (*s*) *yield* *e*

(A la place des parenthèses, on peut utiliser les accolades)

Ici, *s* est une séquence de *générateurs* et de *filtres*.

- Un *générateur* est de la forme **val** *p* ← *e*, où *p* est un motif et *e* une expression dont la valeur est une liste. Il lie les variables dans le motif *p* aux valeurs successives de la liste.
- Un *filtre* est une expression *f* de type *boolean*. Il écarte toutes les liaisons pour lesquelles *f* vaut **false**.
- La séquence doit débuter par un générateur.
- S'il y a plusieurs générateurs dans la séquence, les derniers générateurs varient plus rapidement que les premiers.

Utilisation de *for*

Voici deux exemples qui étaient résolus précédemment grâce à des fonctions d'ordre supérieur :

Exemple : Étant donné un entier positif n , trouver tous les couples d'entiers positifs i, j tels que $1 \leq j < i < n$, et $i + j$ est premier.

```
for { val  $i \leftarrow List.range(1, n)$ ;  
      val  $j \leftarrow List.range(1, i)$ ;  
      isPrime( $i+j$ )  
} yield Pair( $i, j$ )
```

Exemple : On peut écrire le produit scalaire de deux vecteurs ainsi.

```
def scalarProduct( $xs: List[Double], ys: List[Double]$ ) : Double = {  
  sum (for { val Pair( $x, y$ )  $\leftarrow xs$  zip  $ys$  } yield  $x * y$ )  
}
```

Exemple : les n -reines

- Le problème des huit reines consiste à placer 8 reines sur un échiquier de telle manière qu'aucune reine ne soit en prise avec une autre.
- Autrement dit, il ne peut y avoir deux reines sur la même ligne, colonne ou diagonale.
- On développe maintenant une solution pour les échiquiers de taille quelconque, pas simplement 8.
- Une manière de résoudre le problème est de placer une reine sur chaque ligne.
- Une fois qu'on a placé $k - 1$ reines, on doit placer la k -ième reine dans une colonne où elle n'est en échec avec aucune autre reine sur le plateau.

- On peut résoudre ce problème grâce à un algorithme récursif :
 - Supposons qu'on ait déjà généré toutes les solutions consistant à placer $k-1$ reines sur un plateau de largeur n .
 - Chaque solution est représentée par une liste (de longueur $k-1$) contenant des numéros de colonne (entre 1 et n).
 - Le numéro de colonne de la reine dans la ligne $k-1$ vient en premier dans la liste, suivi par le numéro de colonne de la reine dans la ligne $k-2$, etc.
 - L'ensemble des solutions est alors représenté par une liste de listes, avec un élément pour chaque solution.
 - Maintenant, pour placer la k -ième reine, on génère toutes les extensions possibles de chaque solution précédente par une nouvelle reine :

```

def queens (n: int): List [List [int]] = {
  def placeQueens (k: int): List [List [int]] = {
    if (k == 0) List (List ())
    else {
      for { val queens ← placeQueens (k - 1);
            val col ← range (1, n + 1);
            isSafe (col, queens, 1) } yield col :: queens
    }
  }
  placeQueens (n);
}

```

Exercice : Écrire une fonction

```

def isSafe (col: int, queens: List [int], delta: int): boolean

```

qui teste si une reine dans la colonne indiquée *col* est en sûreté par rapport aux reines déjà placées. Ici, *delta* est la différence entre la ligne de la reine à placer et la ligne de la première reine dans la liste.

Requêtes avec *for*

La notation *for* est pour l'essentiel équivalente aux opérations communes des langages de requête des bases de données.

Exemple : Supposons que nous ayons une base de données de livres *books*, représentée comme une liste de livres.

```
class Book {
  val title: String;
  val authors: List[String];
}

val books: List[Book] = List(
  new Book {
    val title = "Structure and Interpretation of Computer Programs";
    val authors = List("Abelson, Harald", "Sussman, Gerald J.");
  },
```

```

    new Book {
        val title = "Introduction to Functional Programming";
        val authors = List("Bird, Richard");
    },
    new Book {
        val title = "Effective Java";
        val authors = List("Bloch, Joshua");
    }
)

```

Alors, pour trouver les titres des livres dont le nom de l'auteur est "Bird" :

```

for { val b ← books; val a ← b.authors; a startsWith "Bird"
    } yield b.title

```

(Ici, *startsWith* est une méthode de *java.lang.String*). Ou, pour trouver les titres de tous les livres qui ont le mot "Program" dans leur titre :

```

for { val b ← books; containsString(b.title, "Program")
    } yield b.title

```

(Ici, *containsString* est une méthode qu'il faut écrire, par ex. en utilisant la méthode *indexOf* de *java.lang.String*).

Ou, pour trouver les noms de tous les auteurs qui ont écrit au moins deux livres présents dans la base de données.

```
for { val b1 ← books;  
      val b2 ← books;  
      b1.title.compareTo(b2.title) < 0;  
      val a1 ← b1.authors;  
      val a2 ← b2.authors;  
      a1 == a2 } yield a1
```

Problème : Que se passe-t-il si un auteur a publié 3 livres ?

Solution : On doit effacer les auteurs en double dans la liste des résultats.

On y arrive avec la fonction suivante.

```
def removeDuplicates [a] (xs: List [a]): List [a] =  
  if (xs.isEmpty) xs  
  else xs.head :: removeDuplicates (xs.tail filter (x ⇒ x != xs.head));
```

Il est équivalent de formuler la dernière expression ainsi

```
xs.head :: removeDuplicates (for (val x ← xs.tail; x != xs.head) yield x)
```

Parenthèse : expressions de création d'objet

L'exemple précédent a montré une nouvelle façon de créer des objets :

```
new Book {  
    val title = "Structure and Interpretation of Computer Programs";  
    val authors = List("Abelson, Harald", "Sussman, Gerald.J");  
}
```

Ici, le nom de la classe est suivi d'un *patron* (template).

Le template est composé de définitions pour l'objet à créer.

Typiquement, ces définitions implantent les membres abstraits de la classe.

C'est similaire aux *classes anonymes* de Java.

On peut voir une telle expression comme étant équivalente à la définition d'une classe locale et d'une valeur de cette classe :

```
{
  class Book' extends Book {
    val title = "Structure and Interpretation of Computer Programs";
    val authors = List("Abelson, Harald", "Sussman, Gerald.J");
  }
  (new Book'): Book
}
```

Traduction de *for*

La syntaxe du *for* est étroitement liée aux fonctions d'ordre supérieur *map*, *flatMap* et *filter*.

Tout d'abord, ces fonctions peuvent toutes être définies en terme de *for*:

```
abstract class List[a] {  
  ...  
  def map[b] (f: a ⇒ b): List[b] =  
    for { val x ← this } yield f(x)  
  
  def flatMap[b] (f: a ⇒ List[b]): List[b] =  
    for { val x ← this; val y ← f(x) } yield y  
  
  def filter (p: a ⇒ boolean): List[a] =  
    for { val x ← this; p(x) } yield x  
}
```

Ensuite, les expressions `for` peuvent elles-même être exprimées en termes de `map`, `flatMap` et `filter`.

Voici le schéma de traduction utilisé par le compilateur (on se limite ici aux motifs simples).

- Une expression `for` simple

`for (val x ← e) yield e'`

est traduite en

`e.map(x ⇒ e')`

- Une expression `for`

`for (val x ← e; f; s) yield e'`

où `f` est un filtre et `s` est une séquence (potentiellement vide) de générateurs et de filtres, est traduite en

`for (val x ← e.filter(x ⇒ f); s) yield e'`

(et la traduction continue avec la nouvelle expression).

- Une expression `for`

`for (val x ← e; val y ← e'; s) yield e''`

où `s` est une séquence (potentiellement vide) de générateurs et de filtres, est traduite en

`e.flatMap(x ⇒ for (val y ← e'; s) yield e'')`

(et la traduction continue avec la nouvelle expression).

Exemple : Si on reprend notre exemple de couples de somme paire :

```
for { val i ← range(1, n);  
      val j ← range(1, i);  
      isPrime(i+j)  
} yield Pair(i, j)
```

voici ce qu'on obtient quand on traduit cette expression :

```
range(1, n)  
  .flatMap(  
    i ⇒ range(1, i)  
      .filter(j ⇒ isPrime(i+j))  
      .map(j ⇒ Pair(i, j)) )
```

Exercice : Définir la fonction suivante en terme de *for*.

```
def concat [a] (xss: List [List [a]]): List [a] =  
  xss.foldRight (xs: List [a], ys: List [a] ⇒ xs ::: ys) (List ())
```

Exercice : Traduire

```
for { val b ← books; val a ← b.authors; a startsWith "Bird" } yield b.title  
for { val b ← books; containsString (b.title, "Program") } yield b.title
```

en fonctions d'ordre supérieur.

Généralisation de *for*

De façon intéressante la traduction de *for* ne se limite pas aux listes ; elle repose uniquement sur la présence des méthodes *map*, *flatMap* et *filter*.

Cela donne au programmeur la possibilité d'avoir la syntaxe *for* pour d'autres types également – on doit seulement définir *map*, *flatMap* et *filter* pour ces types.

Il y a de nombreux types pour lesquels ceci est utile : les tableaux, les itérateurs, les bases de données, les données XML, les valeurs optionnels, les analyseurs syntaxiques, etc.

Par exemple, *books* pourrait ne pas être une liste, mais une base de donnée stockée sur un serveur quelconque.

Du moment que l'interface client de la base de données définit les méthodes *map*, *flatMap* et *filter*, on peut utiliser la syntaxe *for* pour exprimer des requêtes sur cette base de données.

Sujet de recherche active : De quoi a-t-on besoin pour rendre les langages *dimensionnables* (scalable), de telle manière qu'ils puissent subsumer les langages spécifiques à un domaine (parmi lesquels les langages de requête pour les bases de données tels que SQL ou XQuery) ?