Scala évolue...

- Le retour sur les exercices a montré que la règle d'indentation en Scala était la source de beaucoup d'erreurs.
- Nous l'avons donc enlevée.
- A partir de maintenant, toutes les définitions et les instructions doivent se terminer par un point-virgule.
- La seule exception concerne l'expression de retour d'un bloc, pour laquelle le point-virgule est optionnel.
- En outre, un point-virgule est implicitement inséré après une accolade fermante '}', à condition que l'accolade fermante soit suivie d'un caractère de retour à la ligne, puis d'un token qui peut débuter une définition ou une expression.

Exemple:

Avant on écrivait :

```
def foo(x: Int) {
    val y = x * x
    y + 1
}
```

Maintenant on écrit :

```
def foo(x: Int) {
    val y = x * x;
    y + 1
}
```

Avant on écrivait :

```
def reverse: List[a] =
    if (isEmpty) []
    else tail.reverse ::: [head]
```

Maintenant on écrit :

```
def reverse: List[a]
    if (isEmpty) []
    else tail.reverse ::: [head];
```

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 écrit :

$$\textbf{for}\ \{\ \textbf{val}\ p \leftarrow persons;\ p.age > 20\ \}\ yield\ p.name$$

qui est équivalent à :

$$persons\ filter\ (p \Rightarrow p.age > 20)\ map\ (p \Rightarrow p.name)$$

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*** $x \leftarrow e$, où e est une expression dont la valeur est une liste. Il lie x 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.

Utiliser *for*

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

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

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

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

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

Exemple : Les n-reines

En utilisant ***for***, l'exemple des "n-reines" peut s'écrire comme suit.

```
def queens(n: Int): List[List[Int]] = {
  def placeQueens(row: Int): List[List[Int]] = {
    if (row == 0) [[]]
    else {
      def isSafe(col: Int, p: List[Int], delta: Int): Boolean =
        p.isEmpty ||
        (col != p.head && abs(col - p.head) != delta &&
         isSafe(col, p.tail, delta + 1));

      for { val placement ← placeQueens(row - 1);
            val col ← range(1, n);
            isSafe(col, placement, 1)
      } yield col :: placement
    }
  }
  placeQueens(n);
}
```

Requêtes avec *for*

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

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

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

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

```
    new Book with {
      val title = "Introduction to Functional Programming";
      val authors = ["Bird, Richard"];
    },
    new Book with {
      val title = "Effective Java";
      val authors = ["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 que nous devons écrire, en utilisant la méthode *indexOf* de *java.lang.String* par exemple).

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 comme suit.

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

A côté : Expressions de création d'objet

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

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

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

Le template est composé de définitions pour l'object créé.

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 with {
        val title = "Structure and Interpretation of Computer Programs";
        val authors = ["Abelson, Harald", "Sussman, Gerald.J"];
    }
    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] with {
  ...
  def map[b](f: (a)b) =
    for { val x ← this } yield f(x)

  def flatMap[b](f: (a)List[b]) =
    for { val x ← this; val y ← f(x) } yield y

  def filter(p: (a)Boolean) =
    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.

- Une expression for simple

  $$\textbf{for}\ (\textbf{val}\ x \leftarrow e)\ yield\ e'$$

  se traduit en

  $$e.map\,(x \Rightarrow e')$$

- Une expression for

  $$\textbf{for}\ (\textbf{val}\ x \leftarrow e;\ f;\ s)\ yield\ e'$$

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

  $$\textbf{for}\ (\textbf{val}\ x \leftarrow e.filter\,(x \Rightarrow f);\ s)\ yield\ e'$$

  (et la traduction continue avec la nouvelle expression).

- Une expression for

  for $(\textbf{val}\ x \leftarrow e;\ \textbf{val}\ y \leftarrow e';\ s)\ yield\ e''$

  où s est une séquence (potentiellement vide) de générateurs et de filtres, se traduit en

  $e.flatMap(x \Rightarrow \textbf{for}\ (\textbf{val}\ y \leftarrow e';\ s)\ yield\ e'')$

  (et la traduction continue avec la nouvelle expression).

Exemple : A partir de notre exemple "couples dont la somme est paire" :

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

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

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

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

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

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 seulement 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) ?