Week 6: The For Notation

Higher-order functions such as map, flatMap or filter provide powerful constructs for manipulating lists.

But sometimes the level of abstraction required by these functions make the program difficult to understand.

In this case, Scala’s for notation can be used.

Example: Let persons be a list of people, with fields name and age. To obtain the names of people over 20 years old, we write:

```
for ( p ← persons if p.age > 20 ) yield p.name
```

which is equivalent to:

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

The for expression is similar to loops in imperative languages, except that it builds a list of the results of all iterations.

Syntax of For

A for expression is of the form

```
for ( s ) yield e
```

Here, s is a sequence of generators and of filters.

- A generator is of the form \( p ← e' \), where \( p \) is a pattern and \( e' \) an expression whose value is a list.
- A filter is of the form \( if \) where \( f \) is an expression of type Boolean. It removes all bindings for which \( f \) is false.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

And e is an expression whose value is returned by an iteration.

Example: the n queens

- The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.
- In other words, there can’t be two queens in the same row, column, or diagonal.
- We now develop a solution for a chessboard of any size, not just 8.
- One way to solve the problem is to place a queen on each row.
- Once we have placed \( k - 1 \) queens, one must place the \( k \)th queen in a column where it’s not “in check” with any other queen on the board.
• We can solve this problem with a recursive algorithm:
  • Suppose that we have already generated all the solutions consisting of placing 
    \(k - 1\) queens on a board of size \(n\).
  • Each solution is represented by a list (of length \(k - 1\)) containing the 
    numbers of columns (between \(0\) and \(n - 1\)).
  • The column number of the queen in the \(k - 1\)th row comes first in
    the list, followed by the column number of the queen in row \(k - 2\).
  • The solution set is thus represented by a list of lists, with one 
    element for each solution.
  • Now, to place the \(k\)th queen, we generate all possible extensions
    of each solution preceded by a new queen:

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

**Exercise:** Write a function

\[
def isSafe(col: Int, queens: List[Int], delta: Int): Boolean
\]

which tests if a queen in an indicated column \(col\) is secure amongst the
other placed queens. Here, \(delta\) is the difference between the row of the
queen to be placed and the line of the first queen in the list.

---

**Queries with for**

The for notation is essentially equivalent to the common operations of
query languages for databases.

**Example:** Suppose that we have a database of books \(books\), represented
as a list of books.

```scala
class Book {
  val title: String
  val authors: List[String]
}
val books: List[Book] = List(
  new Book {
    val title = "Introduction to Functional Programming"
    val authors = List("Bird, Richard")
  },
  new Book {
    val title = "Effective Java"
    val authors = List("Bloch, Joshua")
  }
)
```

So to find the titles of books whose author’s name is “Bird”:

```scala
for (b ← books; a ← b.authors; if (a.startsWith "Bird")) yield b.title
```

(Here, `startsWith` is a method of `java.lang.String`. `java.lang.String`.

Or, to find all the books which have the word “Program” in the title:

```scala
for (b ← books if containsString(b.title, "Program")) yield b.title
```

(Here, `containsString` is a method that we have to write, for example, using
the method `indexOf` of `java.lang.String`).

---

...
Or, to find the names of all authors who have written at least two books present in the database.

\[
\text{for } ( b1 \leftarrow \text{books}; \\
b2 \leftarrow \text{books}; \\
\text{if } b1.\text{title}.\text{compareTo}(b2.\text{title}) < 0; \\
a1 \leftarrow b1.\text{authors}; \\
a2 \leftarrow b2.\text{authors}; \\
\text{if } a1 == a2 ) \text{ yield } a1
\]

Problem: What happens if an author has published three books?

Solution: We must remove duplicate authors who are in the results list twice.

This is achieved with the following function:

\[
\text{def removeDuplicates[A](xs : List[A]) : List[A] =} \\
\text{if } (xs.isEmpty) xs \\
\text{else xs.head :: removeDuplicates(xs.tail filter (x => x != xs.head))}
\]

It is equivalent to formulate the last expression as:

\[
\text{xs.head :: removeDuplicates(for (x ← xs.tail; if x != xs.head) yield x)}
\]

We can see such an expression as being equivalent to the definition of a local class and of a value of this class.

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

Parentheses: expressions of object creation

The previous example showed a new way to create objects:

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

Here, the name of the class is followed by a \textit{template} (patron en français). The template is composed of definitions for the object to be created. Typically, these definitions implement the abstract members of the class. This is similar to \textit{anonymous classes} in Java.

Translation of \textit{for}

The syntax of \textit{for} is closely related to the higher-order functions \textit{map}, \textit{flatMap} and \textit{filter}.

First of all, these functions can all be defined in terms of \textit{for}:

\[
\text{abstract class List[A] {} \\
\text{def map[B](f : A ⇒ B) : List[B] =} \\
\text{for } ( x ← this ) \text{ yield } f(x) \\
\text{def flatMap[B](f : A ⇒ List[B]) : List[B] =} \\
\text{for } ( x ← this ; y ← f(x)) \text{ yield } y \\
\text{def filter[p : A ⇒ Boolean] : List[A] =} \\
\text{for } ( x ← this ; if (p(x))) \text{ yield } x
\]

Then, the expressions for them can be expressed in terms of map, flatMap and filter.

Here is the translation scheme used by the compiler (we limit ourselves here to simple patterns)

- A simple for expression

\[
\text{for}\ (x \leftarrow e) \quad \text{yield}\ e'
\]

is translated into

\[
e.\text{map}(x \Rightarrow e')
\]

- A for expression

\[
\text{for}\ (x \leftarrow e; \text{if}\ f; \text{seq}) \quad \text{yield}\ e'
\]

where \(f\) is a filter and \(\text{seq}\) is a (potentially empty) sequence of generators and filters, is translated into

\[
\text{for}\ (x \leftarrow e.\text{filter}(x \Rightarrow f); \text{seq}) \quad \text{yield}\ e'
\]

(and the translation continues with the new expression)

Exercise: Define the following function in terms of for.

\[
def \text{concat}\ [A]\ ([xss : \text{List}[\text{List}[A]]]) : \text{List}[A] = \text{xss}.\text{foldRight}\ ([\text{List}[A]])((xs, ys) \Rightarrow xs :: ys)
\]

Exercise: Translate

\[
\text{for}\ (b \leftarrow \text{books}; a \leftarrow b.\text{authors}; \text{if}(a \text{startsWith "Bird")}) \quad \text{yield}\ b.\text{title}
\]
\[
\text{for}\ (b \leftarrow \text{books}; \text{if}(\text{containsString}(b.\text{title}, "Program")}) \quad \text{yield}\ b.\text{title}
\]

into higher-order functions.

Generalization of for

Interestingly, the translation of for is not limited to lists; it is based solely on the presence of the methods map, flatMap and filter.

This gives the programmer the possibility to have the for syntax for other types as well— we must only define map, flatMap and filter for these types.

There are many types for which this is useful: arrays, iterators, databases, XML data, optional values, parsers, etc.

For example, books might not be a list, but a database stored on some server.

As long as the client interface to the database defines the methods map, flatMap et filter, we can use the for syntax for querying the database.
Active research topic: What do we need to make the language *scalable* (dimensionnables en français), so it can subsume domain specific languages (including query languages like SQL and XQuery)?