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 function 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 ($p \leftarrow$ *persons* ***if*** *p.age* $>$ *20*) ***yield*** *p.name*

which is equivalent to:

persons filter ($p \Rightarrow$ *p.age* $>$ *20*) *map* ($p \Rightarrow$ *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 $p \leftarrow e'$, where *p* is a pattern and *e'* an expression whose value is a list.
- A *filter* is of the form ***if*** *f* 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.

Use of *for*

Here are two examples which were previously resolved with higher-order functions:

Example: Given a positive integer n, find all the pairs of positive integers (i, j) such that $1 \leq j < i < n$, and $i + j$ is first.

```
for ( i ← List.range(1, n);
      j ← List.range(1, i);
      if isPrime(i+j)
) yield (i, j)
```

Example: We can write the scalar product of two vectors as well.

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


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 kth 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 $k-2$, etc.
  - The solution set is thus represented by a list of lists, with one element for each solution.
  - Now, to place the kth 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)
}
```

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.

```
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")
  }
)
```

So to find the titles of books whose author's name is " Bird":

```
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:

```
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.

```
for ( b1 ← books;
      b2 ← books;
      if b1.title.compareTo(b2.title) < 0;
      a1 ← b1.authors;
      a2 ← b2.authors;
      if a1 == a2 ) 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:

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

It is equivalent to formulate the last expression as:

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


Parentheses: expressions of object creation

The previous example showed a new way to create objects:

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

Here, the name of the class is followed by a *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 *anonymous classes* in Java.

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

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


Translation of *for*

The syntax of for is closely related to the higher-order functions *map*, *flatMap* and *filter*.

First of all, these functions can all be defined in terms of ***for***:

```
abstract class List[A] {
  ...
  def map[B](f: A ⇒ B): List[B] =
    for ( x ← this ) yield f(x)

  def flatMap[B](f: A ⇒ List[B]): List[B] =
    for ( x ← this; y ← f(x) ) yield y

  def filter(p: A ⇒ Boolean): List[A] =
    for ( x ← this; if (p(x)) ) 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

  for (x ← e) ***yield*** e'

  is translated into

  e.map(x ⇒ e')

- A for expression

  for (x ← e; ***if*** f; s) ***yield*** e'

  where *f* is a filter and *s* is a (potentially empty) sequence of generators and filters, is translated into

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

  (and the translation continues with the new expression)

- A for expression

  for (x ← e; y ← e'; s) ***yield*** e"

  where *s* is a (potentially empty) sequence of generators and filters, is translated into

  e.flatMap(x ⇒ **for** (y ← e'; s) ***yield*** e")

  (and the translation continues with the new expression)

Example: If we take our example of pairs of the first sum:

```
for ( i ← List.range(1, n);
      j ← List.range(1, i);
      if isPrime(i+j)
) yield (i, j)
```

this is what you get when you translate this expression:

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

Exercise: Define the following function in terms of ***for***.

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

Exercise: Translate

for ($b \leftarrow books; a \leftarrow b.authors;$ ***if*** $(a\ startsWith\ "Bird")$) ***yield*** $b.title$
for ($b \leftarrow books;$ ***if*** $(containsString(b.title, "Program"))$) ***yield*** $b.title$

into higher-order functions.

Generalization of *for*

Interestingly, the translation of for is not limited to lists; it is based soley 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)?