For Notation

Higher-order functions such as *map*, *flatMap*, *filter* provide powerful constructions for dealing with lists.

But sometimes the level of abstraction required by these functions makes a program hard to understand.

Here, Scala's ***for*** notation can help.

Example: Say we are given a list *persons* of persons with *name* and *age* fields. To print the names of all persons in the list aged over 20, one writes:

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

which is equivalent to:

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

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

For Syntax

A for-expression is of the form

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

(Instead of parentheses, braces may also be used.)

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

- A *generator* is of the form ***val*** $p \leftarrow e$, where p is a pattern and e is a list-valued expression. It binds the variables in pattern p to successive values in the list.
- A *filter* is an expression f of type *boolean*. It omits from consideration all bindings for which f is ***false***.
- The sequence must start with a generator.
- If there are several generators in a sequence, later generators vary more rapidly than earlier ones.

Using *for*

Here is an example which was solved previously with higher-order functions:

Example: Given a positive integer n, find all pairs of positive integers i, j, where $1 \leq j < i < n$ such that $i + j$ is prime.

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

Example: The scalar product of two vectors can be written as follows.

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

Example: n-Queens

- The eight-queens puzzle asks to place 8 queens on a chessboard so that no queen is in check from any other.
- That is, no two queens may be on the same row, column, or diagonal.
- We now develop a solution for chessboards of arbitrary size, not just 8.
- One way to solve the puzzle is to place a queen in each row.
- Once we have placed $k - 1$ queens, we must place the k'th queen in a column where it does not check any of the queens on the board.

- We can solve this puzzle by a recursive algorithm.
  - Assume that we have already generated all solutions of placing $k-1$ queens on a board of size n.
  - Each solution is represented by a list (of length $k-1$) of column numbers (between 1 and n).
  - The column number of the queen in row $k-1$ comes first in the list, followed by the column number of the queen in row $k-2$, etc.
  - All solutions together are then represented as a list of lists, one element for each solution.
  - Now, to place the k'th queen, generate all possible extensions of each previous solution by one more queen:

```
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);
}
```

Exercise: Write a function

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

which tests whether a queen in the given column *col* is safe with respect to the *queens* already placed. Here, *delta* is the difference between the row of the queen to be placed and the row of the first queen in the list.

Querying with *for*

The for-notation is essentially equivalent to common operations of database query languages.

Example: Say we are given a book database *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");
        }
    )
```

Then, to find the titles of all books whose author's last name is "Bird":

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

(Here, *startsWith* is a method in *java.lang.String*). Or, to find the titles of all books that have the word "Program" in their title:

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

(Here, *containsString* is a method we have to write, by using method *indexOf* in *java.lang.String* for example).

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

```
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
```

Problem: What happens if an author has published 3 books?

Solution: Need to remove duplicate authors from result lists.

This can be 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));
```

The last expression can be equivalently expressed as follows.

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

Aside: Object Creation Expressions

The previous example has shown a new way of creating objects:

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

Here, the class name was followed by a *template*.

The template consists of definitions for the created object.

Typically, these definitions override abstract members of the class.

This is similar to *anonymous classes* in Java.

One can think of such an expression as being equivalent to a definition of a local class and a value of that 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*

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

First, these functions can all be defined in terms of ***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
}
```

Second, for-expressions can themselves be expressed in terms of *map*, *flatMap* and *filter*.

Here is the translation scheme used by the Scala compiler.

- A simple for-expression

  for (**val** $x \leftarrow e$) ***yield*** *e'*

  is translated to

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

- A for-expression

  for (**val** $x \leftarrow e$; *f*; *s*) ***yield*** *e'*

  where *f* is a filter and *s* is a (possibly empty) sequence of generators or filters is translated to

  for (**val** $x \leftarrow$ *e.filter* ($x \Rightarrow f$); *s*) ***yield*** *e'*

  (and then translation continues with the latter expression).

- A for-expression

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

  where s is a (possibly empty) sequence of generators or filters is translated to

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

  (and then translation continues with the latter expression).

Example: Taking our "pairs of integers whose sum is prime" example:

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

Here is what we get when we translate this expression:

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

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

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

Exercise: Translate

$$\textbf{\textit{for}}\ \{\ \textbf{val}\ b \leftarrow books;\ \textbf{val}\ a \leftarrow b.authors;\ a\ startsWith\ \text{"}Bird\text{"}\ \}\ \textbf{\textit{yield}}\ b.title$$
$$\textbf{\textit{for}}\ \{\ \textbf{val}\ b \leftarrow books;\ containsString\,(b.title,\ \text{"}Program\text{"}\,)\ \}\ \textbf{\textit{yield}}\ b.title$$

to higher-order functions.

Generalizing *for*

Interestingly, the for-translation is not restricted to lists at all; it only relies on the presence of methods *map*, *flatMap*, and *filter*.

This gives programmers the possibility to have for-syntax for other types as well – one only needs to 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 instance, *books* might be not a list but a database stored on some server.

As long as the client interface to the database defines methods *map*, *flatMap* and *filter*, we can use for-syntax to query the database.

Topic of active research: What is needed to make languages *scalable*, so that they can subsume domain-specific languages (in this database query languages such as SQL or XQuery)?