Scala Evolves...

- Feedback from exercises showed that the indentation rule in Scala is very error-prone.
- So we have dropped it.
- From now on, all definitions and statements need to end in a semicolon.
- Excepted from this is only the return expression in a block, where the semicolon is optional.
- Furthermore, a semicolon is implicitly inserted after a closing brace '}', provided the closing brace is followed by a newline character, and then followed by a token that can start a definition or expression.

Example:

We used to write:

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

Now we need to write:

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

We used to write:

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

Now we need to write:

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

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** $x \leftarrow e$, where e is a list-valued expression. It binds x 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.

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:

for { **val** *p* ← *persons; 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 a for-loop in imperative languages, except that it constructs a list of the results of all iterations.

Example: n-Queens

Using ***for***, the "n-queens" example can be written as follows.

```
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);
}
```


Using *for*

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

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

```
for { val i ← range(1, n);
      val j ← range(1, i−1);
      isPrime(i+j)
} yield (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 (x, y) ← xs zip ys } yield x * y)
}
```

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.

```
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"];
    }
]
```

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).

Aside: Object Creation Expressions

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

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

Here, the class name was followed by ***with*** and 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.

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)
```

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] 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
}
```


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

- A for-expression

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

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

  e.flatMap (x ⇒ **for** (**val** y ← e'; s) *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–1);
        isPrime(i+j)
} yield (i, j)
```

Here is what we get when we translate this expression:

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


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 ← e) *yield* e'

  is translated to

  e.map (x ⇒ e')

- A for-expression

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

  (and then translation continues with the latter expression).

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)?

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

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

Exercise: Translate

```
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
```

to higher-order functions.