Combinatorial Search and For-Expressions

Handling Nested Sequences

- In imperative programming, when you search for something, you often do that by means of a loop or a series of nested loops
- In functional programming, you don't have loops at your disposal, but you do have higher order functions
- Higher order functions on sequences are a very good toolbox to achieve the same objective as combinatorial search
- Usually the code gets clearer and shorter when using higher order functions on sequences than when using loops

Example: *Given a positive integer n, find all pairs of positive integers I and j, with 1 <= j < i < n such that i + j is prime.*

A natural way to do this is to:

- *Generate the sequence of all pairs of integers (i, j) such that 1 <= j < i < n*
- *Filter the pairs for which i + j is prime*

One natural way to generate the sequence of pairs is to:

- *Generate all the integers i between 1 and n(excluded)*
- *For each integer i, generate the list of pairs (i, 1), … (i, i - 1)*

This can be achieved by combining until and map:

```
val xss = (1 until n).map(i => (1 until i).map(j => (i, j)))
```

We can combine all the sub-sequences using foldRight with ++:

```
xss.foldRight(Seq[Int]())( _ ++ _ )
```

Or, equivalently, we use the buit-in method flatten: ***xss.flatten***

Here is a useful law: ***xs.flatMap(f) = xs.map(f).flatten***

So the expression above, can be simplified to:

```
(1 until n).flatMap(i => (1 until i).map(j => (i, j)))
```

By assembling the pieces, we obtain the following expression:

```
(1 until n)
   .flatMap(i => (1 until i).map(j => (i, j)))
   .filter((x, y) => isPrime(x + y))
```

For-Expressions

- Higher-order functions such as ***map***, ***flatMap*** or ***filter*** provide powerful constructors 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 expression*** can help

Example: *Let persons be a list of elements of class Person, with fields name and age.*

```
case class Person(name: String, age: Int)
```

To obtain the names of persons over 20 years old, you can 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***, where s is a sequence of *generators* and *filters* and e is an expression whose value is returned by an iteration
- A **generator** is of the form ***p <- e***, where p is a pattern and e an expression whose value is a collection
- A **filter** is of the form ***if f*** where f is a Boolean expression
- The sequence must start with a generator
- If there are several generators in the sequence, the last generators vary faster than the first
- The example above can be rewritten in the following manner:

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)
```

Exercise: *Write a version of* ***scalarProduct*** *that makes use of for.*

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

Question: *What will the following code produce?*

```
(for x <- xs; y <- ys yield x * y).sum
```

It would multiply every element of xs with every element of ys and sum up the results.