



# Combinatorial Search and For-Expressions

Principles of Functional Programming

## Handling Nested Sequences

We can extend the usage of higher order functions on sequences to many calculations which are usually expressed using nested loops.

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

For example, if  $n = 7$ , the sought pairs are

$i$	2	3	4	4	5	6	6
$j$	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

## Algorithm

A natural way to do this is to:

- ▶ Generate the sequence of all pairs of integers  $(i, j)$  such that  $1 \leq j < i < n$ .
- ▶ 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  $(i, 1), \dots, (i, i-1)$ .

This can be achieved by combining `until` and `map`:

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

## Generate Pairs

The previous step gave a sequence of sequences, let's call it `xss`.

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

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

Or, equivalently, we use the built-in method `flatten`

```
xss.flatten
```

This gives:

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

## Generate Pairs (2)

Here's a useful law:

```
xs.flatMap(f) = xs.map(f).flatten
```

Hence, the above expression can be simplified to

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

## Assembling the pieces

By reassembling 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))
```

This works, but makes most people's head hurt.

Is there a simpler way?

## For-Expressions

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` expression notation can help.

## For-Expression 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  $p \leftarrow e$ , 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.

## Use of For

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

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 prime.

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

## Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =
```

## Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

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