

Chapter 2

The Basics of the Language

*Ordo autem qui in verbis attenditur est
illud per quod verba tam in loquente
quam in audientibus virtutem et
efficaciam sortiuntur.*
Ramon Llull, *Rethorica nova* [VII].

In this chapter we give a quick review of the most basic elements of the language. We begin reviewing the data types that the language provides by default. Then, we review the syntax for statements and declarations (e.g. conditional and loops). We also discuss definition of functions, higher-order functions and the use of pattern recognition in functions. Our discussion on functions include also recursion. The chapter includes also definitions of lists and other types of collections predefined in Scala which can be naturally processed using recursive functions.

2.1 Data Types

Java provides the usual types that implement integers, and real numbers, Boolean and characters. They are the basic types. Class names and precisions of the basic data types in Scala are given in Table 2.1. Usual functions are defined for objects in these classes.

Scala and Java. As explained above, Scala is based on Java. That is why we have in Scala the same types we have in Java with the same precision. Nevertheless, there are some differences due to a different structure in the system of classes. In Java, the primitive data types (byte, short, int, long, char, float, double and boolean) are not classes, and, thus, do not belong to the hierarchy of objects. In addition to these primitive data types, Java have the classes as e.g. Integer, Long which have some methods implemented. In Scala there is no such distinction. These types are classes, and methods are directly implemented on the classes. Therefore, there is also a difference on how some methods are called/applied in Scala and Java.

Table 2.1 Basic data types in Scala: Type names and precision. Byte, Short, Int, Long, Float, Double and Char are numeric types. All of them are signed except Char that is an unsigned integer. Boolean and Unit are non-numeric types. For details on the actual implementation see Scala documentation.

Type name	Precision
Byte	8 bit signed integer. [Byte.MinValue=-128, Byte.MaxValue=127]
Short	16 bit signed integer. [Short.MinValue=-32768, Short.MaxValue=32767]
Int	32 bit signed integer. [-2147483648, 2147483647]
Long	64 bit signed integer. [-9223372036854775808, 9223372036854775807]
Float	32 bit IEEE-754 floating point number
Double	64 bit IEEE-754 floating point number
Char	16 bit unsigned integer (Unicode char). Range [U+0000,U+FFFF]
Boolean	Values true and false
Unit	There is only one value of type Unit ()

The classes of these types are `value classes`. Value classes are a particular type of classes that are implemented in a more efficient way. Details on value classes as well as an explanation on how to define new ones are given in Sect. 4.3. At this point, the fact of being value classes or not is not relevant.

We review below some of the basic operations defined for basic data types.

- **Value class Int.** Usual operations are implemented in Scala. For example, the following ones.

- Arithmetic. Addition (+), subtraction (−), product (*), quotient (/), remainder (%)
- Comparison (<, >, ==, <=, >=, !=), shifts (>>, <<), bitwise operations (| for OR, & for AND, ^ for XOR), absolute value (abs as e.g. x.abs for an integer value x), maximum (x.max), minimum (x.min).
- Transformation to String. + (binary operator with a string in its first argument).

Examples of valid expressions:

```
2+2      3<4      5>=6
2|10     1|10     3|11
3.abs    1.max(3)  3.min(5)
```

- **Value class Double.** Similar operations exist for Double (including remainder). In addition we have the following (functionality is as expected):

- floor, ceil, isInfinite(), isNaN()

- **Value class Boolean.**

- Comparison (==).
- Logical connectives. And (&, &&), or (|, ||), xor (^), negation (!, this operator is prefix so used as !true). & and | evaluate the two arguments (eager evaluation). && and || do not necessarily evaluate both (i.e., they use lazy evaluation).

Scala Language Specification (Version 2.11): Section 6.12.1. Prefix operations

A prefix operation $op; e$ consists of a prefix operator op , which must be one of the identifiers $+$, $-$, $!$ or $.$. The expression $op; e$ is equivalent to the postfix method application $e.unary_op$.

Prefix operators are different from normal function applications in that their operand expression need not be atomic. For instance, the input sequence $-\sin(x)$ is read as $-(\sin(x))$, whereas the function application $\text{negate } \sin(x)$ would be parsed as the application of the infix operator \sin to the operands negate and (x) .

Fig. 2.1 Prefix operators in Scala according to Scala Language Specification (Version 2.11).

For a discussion of prefix and infix operators, and on precedence of operators see Figs. 2.1 and 2.2.

Prefix and infix. We have an infix operator when it goes between its arguments. In mathematics, addition and subtraction are usually expressed by the infix operators $+$ and $-$ as e.g. in $2 + 2$ and $2 - 1$. We have a prefix operator when it goes before the arguments. The expression -2 has a prefix operator $-$.

It is usual to mix prefix and infix operators, but there are languages where all operations are prefix. This is the case of LISP where we have the name of the function always first. So, an expression as $2 + 3 * 3 + \text{sqrt}(5 + 4)$ is expressed in LISP as follows:

```
(+ 2 (* 3 3) (sqrt (+ 5 4)))
```

2.1.1 Strings

Among the predefined types of Scala we find Strings. They are defined as in most languages by double quotes. We can determine the length of a string (with `length`), concatenate them (with `concat` and with `+`), compare them (with `==`), and select the element at a given position (with `charAt(position)`). Other methods from `java.lang.String` can also be used in Scala (e.g., `toUpperCase`, and `compareToIgnoreCase`). So, the following are valid operations with strings:

```
"a, b, c; alpha beta gamma; 1, 2, and 3"
"one" + "two" + "three"
"one".concat("two") == "one" + "two"
"one".compareToIgnoreCase("ONE")
"one".charAt(0) + "one".charAt(1) + "one".charAt(2)
```

Note that the last expression returns 322 because `charAt` returns a char that is a 16 bit unsigned integer (Unicode char)! Also note that the initial character of a string is at position zero.

Scala Language Specification (Version 2.11): 6.12.3 Infix operations

An infix operator can be an arbitrary identifier. Infix operators have precedence and associativity defined as follows:

The *precedence* of an infix operator is determined by the operator's first character. Characters are listed below in increasing order of precedence, with characters on the same line having the same precedence.

```
(all letters)
|
^
&
= !
< >
:
+ -
* / \%
(all other special characters)
```

That is, operators starting with a letter have lowest precedence, followed by operators starting with |, etc.

There's one exception to this rule, which concerns *assignment operators*. The precedence of an assignment operator is the same as the one of simple assignment (=). That is, it is lower than the precedence of any other operator.

The *associativity* of an operator is determined by the operator's last character. Operators ending in a colon ':' are right-associative. All other operators are left-associative.

Fig. 2.2 Infix operators in Scala according to Scala Language Specification (Version 2.11).

It is important to know that strings are immutable¹ objects.

Scala includes three types of interpolators for strings. Interpolators permit us to include in a string expressions that need to be evaluated (e.g. variables to be replaced by their value). The three types are *s*, *t*, and *raw* interpolation. *s* permits to evaluate expressions, *t* is similar to `printf` in the C language, and *raw* does not escape the `\` characters. We do not go into details of this, but just consider the following examples that permit to replace an expression by its computation, and prints `\n` without replacing it by a new line.

```
println(s"The maximum between 1 and 8 is ${1.max(8)}")
println(raw"\n 1 \n 2 \n 3")
```

The output in Scala is as follows.

```
scala> println(s"The maximum between 1 and 8 is ${1.max(8)}")
The maximum between 1 and 8 is 8

scala> println(raw"\n 1 \n 2 \n 3")
\n 1 \n 2 \n 3
```

¹Immutable objects are discussed in Sect. 2.9.1

2.2 Statements and Expressions

We review in this section some of the basic constructions in Scala.

- **Conditional.** It is similar to conditional in most languages. We have the following

```
if(BooleanExpression) { Expression }

if(BooleanExpression) { ExpressionTrue }
else { ExpressionFalse }
```

We use here expressions for the then and the else branches. In fact, being Scala an object oriented language, we can use commands (and sequences of commands) in both then and else branches. When we have single expressions, we can remove the curly brackets.

- **Loops.** As we focus on functional programming, we will avoid loops as much as possible in our programs. Nevertheless, they are explained here for the sake of completeness. We have while and do-while loops that can be used as follows.

```
while (BooleanExpression) { Expression }

do { Expression } while (BooleanExpression)
```

There are also for loops in Scala. We will discuss them later in Sect. 2.9.3, but for the time being, they can be used as in the following example.

```
for (i <- 1 to 10) { statement }
```

Then, we will execute the statement ten times and the variable `i` will take values 1, 2, 3, ..., 10, as expected, in the 10 consecutive executions of the statement.

2.3 Statement Separator and Blocks

Newline separates statements in Scala. Alternatively we can use semicolon “;” to separate statements, when needed. Published code usually do not have much semicolons. So, the code

```
statement1
statement2
```

is equivalent to the following:

```
statement1; statement2
```

Blocks of statements use curly brackets. For example,

```
{ statement; statement }
```

2.4 Comments

We can include comments in the following two ways.

```
/* Multiple
   line
   comment */

// One line comment
```

2.5 Declarations

We can associate values to identifiers using `val` and `var`. We use them as follows

```
val nameConstant: Type = expression
var nameVariable: Type = expression
```

With `val` we are defining a constant, and we associate it with a value. This association can no longer be changed. With `var` we are defining a variable (in an imperative sense) and associate it to a value that can be later changed. We will revisit the difference between both `val` and `var` in Sect. 2.9 when discussing mutable and immutable data structures.

To change the value of a variable defined with `var`, we just assign another value to it. Observe the following.

```
val a1 = 2*5
var a3 = 4*6
a3 = 8484
```

In the interpreter, constants defined with `val` can be redeclared, but they cannot be really overwritten. Observe the following.

```
scala> val a1 = 2*5
a1: Int = 10
scala> a1 = 54
<console>:8: error: reassignment to val
      a1 = 54
      ^
scala> val a1 = 54
a1: Int = 54
```

Note that `a1` cannot be redefined, but we can declare the same name again. This is, in fact, as defining a new constant which can be seen as hiding the scope of the previous definition.

Declarations in functional programming. We will mainly use in this text `val` because we understand variables in a mathematical way (as in mathematical expressions *Let X be ...*). That is, as constants that do not change their values. We do not see them as positions of memory whose value can be changed.

2.5.1 *Composite Types: Cartesian Products*

We can use basic types and compose them. We can also define variables as the cartesian product of two or more types. For example, the following is a valid declaration in Scala.

```
val a1 = (2*5, "ten")
```

We can access the elements of the products by means of `._1`, `._2`, etc. So, `a1._1` will return 10.

2.5.2 *Nested Declarations*

We can include declarations inside other declarations. That is, nested declarations are possible in Scala. This can help on the calculation of a value. Note that the declarations inside another declarations are not accessible from outside.

In the following example, `a1` and `a2` are local to the definition of `a`.

```
val a = {  
  val a1 = 10  
  val a2 = "In text: "+a1+"is ten"  
  (a1, a2)  
}
```

Check that the values of `a1` and `a2` are not available once the declaration of `a` is completed. So, the scope of `a1` and `a2` is only within `a`.

2.6 Functions

The declaration of a function can be done by means of code with the following structure: list of arguments between parenthesis, the symbol “=>”, and the body of the function. The following are simple examples of functions.

```
(a:Int) => 2*a
(a:Int, b:Int) => a+b
(a:Int, f:Int=>Int) => f(a)
```

The first function has a single integer parameter (with name `a` and type `Int`) and multiplies it by two. The second function has two integer parameters (with names `a` and `b` and types `Int`) and adds them.

The third function has two parameters. The first parameter is an integer (parameter `a`) and the second one is a function (parameter `f`) that given an integer computes another integer. The body of the third function shows that applies `f` to `a`. Note that the type of the parameter `a` is `Int`. The type of the function `f` is `Int => Int` because it receives an `Int` and returns `=>` another `Int`.

In general, the type of a function with n arguments has the following structure.

```
Type1, Type2, ..., TypeN => OutputType
```

The functions we have seen are anonymous. That is, they have no name. Nevertheless, they can be applied and passed to other functions. For example, we can apply the first function to 3 as follows:

```
((a:Int)=>2*a)(3)
```

and we can pass the first anonymous function to the third anonymous functions as follows (together with the integer 3 as the latter needs two parameters. This is done as follows.

```
((a:Int, f:Int=>Int) => f(a))(3, ((a:Int)=>2*a))
```

Exercise 2.1. Given the three parameters of a 2nd degree equation

$$ax^2 + bx + c = 0$$

write an anonymous function that returns its two solutions. Use a nested declaration to compute the discriminant of the solutions only once. Apply the anonymous function to find the solution of $x^2 - 3 = 0$.

Anonymous functions are useful in functional programming, but it is of course also necessary to have functions with names.

In Scala, all functions are objects. Therefore, we can declare/assign them using `val`. For example, we can declare previous functions (i.e., give them a name!!) as follows².

```
val f1 = (a:Int) => 2*a
val f2 = (a:Int, b:Int) => a+b
val f3 = (a:Int, f:Int=>Int) => f(a)
```

² This is not the only way used in Scala to *define* functions. We can use `def`. Both ways are not exactly the same and `def` is not properly speaking a way to define functions. That is why we start defining functions with `val`. This is further explained in Sect. 6.3.

Now, we can apply these functions to objects in a more usual way. E.g., we can compute

```
f1(3)
f2(5, 8)
f3(10, f1)
```

As a summary, we have that the definition of a function follows this structure:

```
val name = <anonymous-function-definition>
```

2.6.1 *Alternative Ways to Define Types in Functions*

When we use a definition of the form above, the information on the types of involved parameters is in the anonymous function.

We can also give the information about the type on the name of the function. For example, function `f1` receives an `Int` and returns an `Int`. This is expressed in Scala as `(Int => Int)`. The following three definitions are all valid in Scala for `f1`. Note that the third one contains redundant information, as the type of parameter `a` is given twice.

```
val f1 = (a:Int) => 2*a
val f1:(Int => Int) = a => 2*a
val f1:(Int => Int) = (a:Int) => 2*a
```

Similarly, we can define functions `f2` and `f3` above as follows:

```
val f2:((Int,Int)=>Int) = (a:Int, b:Int) => a+b
val f2:((Int,Int)=>Int) = (a, b) => a+b
val f3:((Int,Int=>Int)=>Int) = (a:Int, f:Int=>Int) => f(a)
val f3:((Int,Int=>Int)=>Int) = (a, f) => f(a)
```

Type definition in functions. Scala permits different ways to express the type of a function. It is usually **more convenient** to associate types to functions than to their parameters. That is, among the alternatives seen, the most convenient way to define a function is to follow this pattern:

```
val name: FunctionType = <anonymous-function-definition>
```

2.6.2 *Type Inference in Scala*

A type inference system permits to conclude the types of objects and functions from their definition. There are languages as Standard ML where the type inference system is very advanced.

Scala has a limited type inference system. For example, the following definition without types is valid (because Scala infers the type of `f4` from the type of `f1`).

```
val f4 = a => f1(a)
```

Nevertheless, the following two definitions return an error because the type is not given.

```
val f5 = a => 3*a
val f6 = a => 2*f1(a)
```

Type inference in Standard ML. Standard ML (SML) has a more elaborated type inference system than Scala. It accepts the following two definitions for `f5` and `f6`, and infers correct types for them. If we type in the SML interpreter

```
fun f5(a) = 3*a;
fun f6(a,f1) = 2*f1(a);
```

We obtain:

```
val f5 = fn: int -> int
val f6 = fn: 'a * ('a -> int) -> int
```

where `'a` means that any type is valid.

Observe that Scala needs type declarations here.

2.6.3 Signature

The signature of a function corresponds to a description of the types involved in the inputs and output of the function. That is, the types of its arguments (inputs) and its result (output).

Signature of a function. In general, the type of a function is

```
Type1, Type2, ..., TypeN => OutputType
```

However, note that `TypeI` and `OutputType` can correspond to types of functions.

To illustrate that in the signature we may need to express that some parameters are functions, we can consider the following signature.

```
((Int,Int)=>(Int => Int) , (Int=>Int) , Int, Int, Int)=>Int
```

This is valid for a function type. For example, the following function `f7` has this type.

```
val f7: ((Int, Int) => (Int => Int), (Int => Int), Int, Int, Int) => Int =
(f: (Int, Int) => (Int => Int), g: (Int => Int), a: Int, b: Int, c: Int) =>
  f(g(a), g(b))(c)
```

Note that this function has five arguments, two of them are functions, and three are integers. We apply `f7` below to two functions (one defined with name `ff` and the other anonymous) and three integers.

```
val ff: ((Int, Int) => (Int => Int)) = (a, b) => (x: Int) => a + b + x
f7 (ff, (x: Int) => 2 * x, 1, 2, 3)
```

Note that in Scala we need to declare the types.

Inference in Standard ML. In Standard ML it suffices to define:

```
fun f7Then(f, g, a, b, c) = f(g(a), g(b))(c);
```

Standard ML infers for `f7` the following type:

```
val f7 = fn : ('a * 'a -> 'b -> 'c) * ('d -> 'a) *
          'd * 'd * 'b -> 'c
```

The computation of the expression above in Standard ML is:

```
fun ff(a, b) = (fn x => a + b + x);
f7 (ff, fn x => 2 * x, 1, 2, 3);
```

2.6.4 Referentially Transparent

Given an expression e , we say that e is *referentially transparent* when we can replace the expression e by its value in all occurrences of e in the program without affecting the result of the program.

A pure function is a function that given the same input values, the output value is always the same, and there are no side effects.

Side effect. An expression has side effects when in addition to its evaluation it modifies somehow the state of the machine (e.g., update global variables, print values in the screen or to a file).

Note that this is not always the case in imperative languages, as functions can have a state, and then the output of the function can change even if we do not change the input.

Random number generators are typical examples of non pure functions. For example, in Java, the functions `nextInt()` and `nextInt(int n)` of class `java.util.Random` are not pure. Note that different applications of these func-

tions may result into different results. Precisely, this is the goal of the random generator function, that different values are obtained. Functions and methods applied to objects with internal states are usually not pure (as the state is not explicitly stated in the function). An explicit random state as e.g. in the call `randomState.nextInt()` could return a new random state and the random number, being a pure function.

The expression `println` in Scala is also not pure. Although its result is always of type `Unit`³, it has a side effect. That is, the expression prints a string onto the screen.

Referentially transparent. An expression satisfies this property when it can be replaced by its evaluation without modifying the outcome of the program. Expressions with side effects are not referentially transparent.

2.6.5 Higher-Order Functions

We have a higher-order function when one of its parameter is another function. This is the case of function `f3` above. Recall that it requires a function with signature `(Int=>Int)` as a parameter.

For example, the arithmetic mean of two values a and b corresponds to

$$(a + b)/2$$

and the quasi-arithmetic mean is defined for a function f with inverse f^{-1} as

$$f^{-1}((f(a) + f(b))/2).$$

We can implement the quasi-arithmetic mean [18] as a higher-order function `qam` with parameters `a` and `b` and two functions f and f^{-1} . We call this later function `fm1` in the definition below.

```
val qam: (Double, Double, Double=>Double, Double=>Double) => Double =
  (a, b, f, fm1) => fm1((f(a) + f(b)) / 2)
```

Then, we can compute the quasi-arithmetic mean of 1 and 2 with $f(x) = x$ and $f^{-1}(x) = x$ as follows⁴.

```
qam(1, 2, (x:Double) => x, (x:Double) => x)
```

Similarly, the quasi-arithmetic mean of 1 and 2 with $f(x) = x^2$ and $f^{-1}(x) = \sqrt{x}$ is computed by⁵:

```
qam(1, 2, (x:Double) => x*x, (x:Double) => math.sqrt(x))
```

³Check the value of `pln` after declaring

```
val pln = println("println is pure?").
```

⁴The quasi-arithmetic mean with $f(x) = x$ is just the arithmetic mean.

⁵The quasi-arithmetic mean with $f(x) = x^2$ is the geometric mean.

We use higher-order functions when we need some generality in our definition. For example, if we want to order the elements of a database but we do not want to establish how to compare pairs of elements we can use a higher-order function where the comparison (e.g., `lessThan`) is one of the parameters of the ordering method. We will then be able to compare and order in different ways. For example, by surname, or by city.

2.6.6 *Currification*

Any function of n parameters can be seen as a function that has a single parameter, and given it, it returns a function with $n - 1$ parameters.

Currification is the technique for making this transformation.

As an example of Currification, observe that we can define the arithmetic mean as a function of two arguments as follows:

```
val am: ((Double, Double) => Double) = (a,b) => (a+b)/2
```

but also as a function of one argument that returns another function that given one argument it computes the mean of this argument with the previous one. That is,

```
val curryAm: (Double => (Double => Double)) =  
  (a) => { (b) => (a+b)/2 }
```

The way to call these functions will be different. We will use:

```
am(2,5)  
curryAm(2)(5)
```

The main and important difference between curried and non curried is that we can call the latter with only some of the first arguments. For example, the following call is valid:

```
curryAm(2)
```

This call returns a function that computes the mean of any number with 2. We can thus define

```
val meanWith2 = curryAm(2)
```

and then apply this function to any other number as e.g.

```
meanWith2(10)
```

Let us consider a function to calculate the compound interest of a sum. The expression of the total accumulated value when the initial amount was P (the principal sum) and the total is to be computed for t years at a i nominal interest rate compounded annually is the following:

$$P(1+i)^t$$

We can write this function in Scala as follows.

```
val compoundInterest: ((Double, Double, Double) => Double) =
  (i,t,p) => p*Math.pow(1+i,t)
```

Then, if we want to compute the balance after 5 years of 1000 Euros at 2.5 % of annual interest, we can call this function as:

```
compoundInterest(0.025, 5, 1000)
```

We give now the function in curried form and give also an example of its application.

```
val compoundInterest:
  (Double => (Double => (Double => Double))) =
  (i) => { (t) => { (p) => { p*Math.pow(1+i,t) } } }
compoundInterest(0.025)(5)(1000)
```

When the function is curried we can define a new function that computes the compound for any value when the interest and the number of years are known. For example, let us consider that today we have a 3 % interest and 4 years. Then, we can define a function `compound interest` to any principal sum. We give an example below and its application to 1000 euros. Naturally, once we have this function, we can apply it to any amount of money.

```
val ourBankInterestTodayAt4Years = compoundInterest(0.03)(4)
ourBankInterestTodayAt4Years(1000)
```

Curried functions are helpful to us because we can apply them only partially, which gives us additional flexibility.

2.6.7 Recursive Functions

In functional programming repetition is usually implemented by means of recursion. Recall that a function is recursive when it calls to itself.

Iteration and loops are avoided in functional programming because they rely on variables that change their state. Let us compare the definition of the factorial using loops and using recursion.

Recall that the factorial of zero is defined as 1, and then in general the factorial of an integer number $n > 0$ is defined as n multiplied by the factorial of $n - 1$. The mathematical expression for the factorial is, therefore.

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fact(n - 1) & \text{if } n > 0 \end{cases}$$

Using this definition, it is straightforward to define the recursive form of the factorial.

In Scala, when we define a function recursively we need to declare its type. Because of that, in the definition of factorial we express that this function receives an `Int` and computes another `Int`. Recall that this is expressed as `(Int => Int)`. Then, the body of the function distinguishes by means of an `if` the base case (i.e., when $n = 0$) that directly returns the value of the factorial (i.e., $fact(0) = 1$) and the recursive case that computes $n \cdot fact(n - 1)$. The corresponding code in Scala is, thus, as follows.

```
val fact: (Int=>Int) =
  (n: Int) => { if (n==0) {1} else {n*fact(n-1)}}}
```

or, equivalently, without giving the type of `n`:

```
val fact: (Int=>Int) =
  (n) => { if (n==0) {1} else {n*fact(n-1)}}}
```

It is well known that the factorial is equivalently defined by the following expression.

$$fact(n) = \prod_{i=1}^n i$$

That both expressions are equivalent is proven by induction but this is out of the scope of this text. See e.g. [17] and [14] for a reference on induction and proofs by induction.

This later expression is used in most imperative versions of the factorial function. In this case, we have a variable that takes values from 1 to n , and a variable that stores the partial results.

We can implement this version in Scala as follows.

```
val fact: (Int=>Int) = (n) => {
  var res = 1
  for (i <- 1 to n) {
    res = res*i
  }
  res
}
```

We can compare this definition with the iterative version studied in Chap. 1. Note that the function in Scala does not need a `return` statement as the last expression computed in the function is the one returned.

Recursive functions. They are functions that call themselves. Recall that recursive functions need to consider: (a) a base case, that is not recursive and that returns a value; (b) a recursive case, in which the function is applied recursively to an object that is simpler than the one received by the function. Simpler means that is more similar to the base case. For example, the factorial has 0 as its base case, and the recursive case applies the factorial function to a simpler object (the original value less one, naturally $n - 1$ is more similar to zero than n).

Other typical examples of recursive functions are the function Fibonacci, and the function to solve the problem of towers of Hanoi. The straightforward implementation of the Fibonacci function is quite inefficient but it is useful to illustrate recursion.

Exercise 2.2. Define recursively the function Fibonacci and the towers of Hanoi. Use a pure functional implementation for the former. You can use some imperative (as e.g. sequences of `println`) for the later.

Recall that the Fibonacci series are defined as follows. $F_0 = 0$, $F_1 = 1$ and $F_i = F_{i-1} + F_{i-2}$ (for $i > 1$).

2.6.8 Functions and Non Functional Programming

When we define a function, we can include in its body any valid Scala expression. This naturally includes loops and blocks (with sequences of statements). We have seen an example above of the iterative version of the factorial function. When we have a sequence of expressions, the function returns the last one (we do not need an explicit return statement).

2.7 Lists

Scala implements lists. In a list, all objects should be of the same type. So, if we have a list of integers, formally it will be of type `List[Int]`. Lists have two constructors. `Nil`, which establishes an empty list, and `::`, which adds an element to a list.

The following are valid expressions.

```
val exampleEmptyList = Nil
val exampleListOne = 1::Nil
val exampleListThree = 1::2::3::Nil
```


The constructor `::` is right associative, so, `1::2::3::Nil` is equivalent to `1::(2::(3::Nil))`. We can also use the function `List` that can receive an arbitrary number of arguments to define a list. This is used as follows:

```
List(objects between commas)
```

Consider for example the following list.

```
val anotherExampleListThree = List(1,2,3)
```

These examples defined lists of integers. Similarly, we can make lists of strings as follows.

```
"First"::"Second"::"Third"::"Fourth"::Nil
```

In fact we can even mix the type of the objects when we construct a list. We can define, for example,

```
"First"::2::"Third"::4::Nil
```

Nevertheless, as all the elements of the list should have the same type, in this case, the type of the list will be `List[Any]` because of the hierarchy of objects in Scala (see Sect. 4.1).

There are a few functions defined for lists. Some of them follow.

- `head` returns the first element of the list. E.g., `exampleListThree.head` returns 1.
- `tail` returns the tail of the list (the list without the first element). E.g., `exampleListThree.tail` returns the list `2::3::Nil`
- `isEmpty` returns true if the list is Nil. E.g., `exampleListThree.isEmpty` returns false.
- `==` compares two lists. E.g.,

```
scala> anotherExampleListThree == exampleListThree
res24: Boolean = true
```

```
scala> anotherExampleListThree == exampleListOne
res25: Boolean = false
```

Note that in these examples we call the function `functionName` for a list `aList` using `aList.functionName`. This is because `aList` is an object of the type `List` and we are calling the method `functionName` for this object (i.e., sending a message to the object using object oriented terminology). In the last case, the notation

```
anotherExampleListThree.==(exampleListOne)
```

is also correct. In Sect. 4.2.1 we discuss with some details different alternative notations in Scala.

- Other functions include: `reverse`, `length`, `:::` (that concatenates two lists), `last`, and `sorted`.

2.7.1 Recursion on Lists

It is usual to process the elements of a list to find one (or all) that satisfies a property, to count them, etc. We have seen some of these functions above. We will show how to implement them here.

Most algorithms can be classified as either as a traversal or as a search on a data structure. We have search when we are looking for an object with a certain property. Once the object is found, the search is stopped. We have traversal, when we need to visit all the objects in the data structure. The same applies to lists.

- **Examples traversing lists.** We give below a few examples that need to traverse a list. They are the functions `length`, `sum`, and `prod`. The first one computes the length of the list. Then, `sum` and `prod` compute the sum and the product of the elements of the list. In all cases we need to check all the elements either to count them or to operate them. All of them are defined by means of recursion.

As we need to traverse the whole list, the base case is always the empty list (`Nil`), and the general recursive case is applied to the list without the head. Note that when we remove the head, the list contains one element less and, thus, it is simpler and more similar to the empty list.

The definitions follow.

```
val length: (List[Int] => Int) = (l) => {
  if (l==Nil) { 0 } else { 1+length(l.tail) } }

val sum: (List[Int] => Int) = (l) => {
  if (l==Nil) { 0 } else { l.head+sum(l.tail) } }

val prod: (List[Int] => Int) = (l) => {
  if (l==Nil) { 1 } else { l.head*prod(l.tail) } }
```

Check that these functions work properly testing e.g.

```
sum(exampleListThree)
prod(exampleListThree)
```

- **Examples searching in lists.** Let us consider two examples of searching. One that looks for a particular integer in a list of integers, and another that given a test function returns the first integer that satisfies the test function. For simplicity, this latter function will return -1 if the object is not found. We call these functions, respectively, `thereIs` and `thereIsOneSatisfyingP`.

The signature of the first function is `((Int, List[Int]) => Boolean)` as it receives an integer and the list and returns a `Boolean`. The signature of the second function is `((Int => Boolean, List[Int]) => Int)`. In this case the function requires the function `p` and the list, and returns the integer found (or -1). In order to test the function `thereIsOneSatisfyingP`, we define two additional functions. They are the predicates `is2` and `is3multiple` that receive an integer and return `true` when it is 2, or a multiple of 3, respectively.

```

val thereIs: ((Int,List[Int]) => Boolean) = (e, l) => {
  if (l==Nil) { false } else {
    if (e==l.head) { true } else {
      thereIs(e, l.tail) }}}

val thereIsOneSatisfyingP: ((Int => Boolean, List[Int])=> Int) =
  (p, l) => {
    if (l==Nil) { -1 } else {
      if (p(l.head)) { l.head } else {
        thereIsOneSatisfyingP(p,l.tail) }}}
val is2: (Int => Boolean) = (x) => { x==2 }
val is3multiple: (Int => Boolean) = (x) => { x % 3 == 0 }

```

We illustrate now the application of these functions with the following calls.

```

thereIs(2,exampleListThree)
thereIs(5,exampleListThree)
thereIsOneSatisfyingP(is2, 1::2::3::4::Nil)
thereIsOneSatisfyingP(is3multiple, 2::5::8::9::Nil)
thereIsOneSatisfyingP(is3multiple, 2::5::Nil)

```

Predicate. We use the term predicate in this book as equivalent to a function that given an object returns a Boolean.

In a search problem, one base case typically corresponds to finding the element we are looking for. This is the element e in the first function (condition $e==l.head$) and an element that makes the test function p true in the second function (condition $p(l.head)$). In the case that the condition is true we return true in the first function and the element in the second.

The general case typically consists on a recursive application of the function to the tail of the list. When we are not sure to find the element, these functions have an extra base case to finish the traversal of the list. Usually, this is to check whether the list is empty. The first function returns false for this base case (there is no such element e) and the second function returns -1 .

We give below another version of the function product. This function traverses the list multiplying the elements but at the same time searches for a zero, and if the zero is found it returns zero directly.

```

val prodV2: (List[Int] => Int) = (l) => {
  if (l==Nil) { 1 } else {
    if (l.head==0) { 0 }
    else { l.head*prodV2(l.tail) } } }
// Test of function prodV2
prodV2(1::2::0::4::Nil)

```

Recursive functions on lists. There are mainly two types of functions: traversal and search.

- In traversal, it is usual that the base case corresponds to the empty list, and the general case applies recursively the function to the tail of the list.
- In search, it is usual that the base case corresponds to the case of finding the element (and also to the empty list if it may happen that the element is not found), and the general case applies recursively the function to the tail of the list.

2.8 Pattern Matching

Functional programming languages often include pattern matching. Pattern matching permits us to differentiate easily among different cases of a given structure. In addition, it permits us to associate some of the elements of the structure to variables. This is obtained making two structures equal. In Scala, we use `match` for pattern matching. The general structure is as follows:

```
variable match {
  case FirstCaseExpression => FirstExpression
  case SecondCaseExpression => SecondExpression
}
```

It is important to underline that in Scala, the variable should be instantiated. This means that it should be linked to a value.

When `variable` matches a case, the corresponding expression is evaluated. Pattern matching permits us to use variables in the case conditions. Then, if matching takes place, variables are bounded to subexpressions. These variables can then be used in the corresponding expression on the right hand side of the case.

To illustrate how this works, let us redefine the factorial function using `match`.

```
val fact:(Int=>Int) = (n) => { n match {
  case 0 => 1
  case m => m*fact(m-1)
}}
// Test
fact(5)
```

When we compute `fact(5)`, first, `n` is associated to 5. Then, as we have `n match` the value of `n` i.e. 5 is compared with each of the case expressions. That is, first, we compare `n=5` and the case `0`. As the two values are different, this case fails. Then, we compare `n=5` and the case `m`. As `m` is a variable, both expressions can be made equal when `m` is 5. Thus, we apply the right hand side of this case with `m=5`. That is, we compute `5*fact(4)`. In this way, we will obtain the result of the function.

2.8.1 Pattern Matching on Lists

Pattern matching is usually applied to structures. For example, to define functions for lists. In this case it is usual to distinguish between the empty list and the list with at least one element. In the examples given in the previous section, we used the conditional to distinguish between these two cases. We can use pattern matching for the same purpose. In this case, we can directly associate variables to the appropriate elements of the list.

For example, the following definition computes the length of a list using pattern matching. As in the previous section, we distinguish between two cases: the empty list and the case of at least one element. The first case checks whether the list `l` can be made equal to `Nil`. This is only possible if `l` is empty. The second case checks whether `l` can be made equal to a list `hd::tl`. Here, `hd` and `tl` are two variables and we will have that `hd` will be associated with the head of the list and `tl` to the tail. This association will only be possible if `l` has at least one element. The names of variables `l`, `hd`, and `tl` are all arbitrary.

```
val lengthMatching: (List[Int]=>Int) = (l) => l match {  
  case Nil => 0  
  case hd::tl => 1+lengthMatching(tl)  
}
```

Note that in this definition `hd` is not used. Because of that we can just replace `hd` by the symbol `_` which corresponds to an unnamed variable. The alternative definition is as follows.

```
val lengthMatching: (List[Int] => Int) = (l) => l match {  
  case Nil => 0  
  case _::tl => 1+lengthMatching(tl)  
}
```

We redefine below the examples of `sum` and `prod`.

```
val sumMatching: (List[Int] => Int) = (l) => l match {  
  case Nil => 0  
  case hd::tl => hd+sumMatching(tl)  
}
```

```
val prodMatching: (List[Int] => Int) = (l) => l match {  
  case Nil => 1  
  case 0::_ => 0  
  case hd::tl => hd*prodMatching(tl)  
}
```

Exercise 2.3. Use lists to implement a multiset. A multiset is similar to a set but in which elements can appear more than once. For example, $\{a, a, b, b, b\}$ is a multiset. Implement the functions `union`, `intersection`, and `count` for multisets. Given a multiset and an element, the function `count` returns how many times this element is in the multiset.

Pattern matching in Prolog. In Scala when we consider variable match `{ list of cases }` we need that the variable is instantiated (i.e., its value is known). This is not the case in all languages. Prolog is an example of a language in which variables to be matched do not need to be instantiated. For example, we can build a predicate that delivers a list of N elements. This predicate compares the variable `List` which is not instantiated with the empty list `[]` when N is zero, and with a list with at least one element `[_|Tail]` when N is larger than zero.

```
listOfN(List,0):-List=[].
```

```
listOfN(List,N):-N>0, List=[_|Tail], N1 is N-1, listOfN(Tail,N1).
```

We can test this code writing the following

```
listOfN(AListWith5Elems,5).
```

The execution of this code will return a list of 5 arbitrary elements (see below). That is, the five elements are in fact not instantiated and can be *associated* to any value later. In Prolog, variables starting with `_` denote that they have no value associated.

```
?- listOfN(AListWith5Elems,5).
```

```
AListWith5Elems = [_G2984, _G2987, _G2990, _G2993, _G2996] ;  
true.
```

In Scala's pattern matching the variables within `match` are considered as *new*. Because of that, if we are using `hd` in the `match` part and we have a `hd` as one of the parameters of the function, they are considered as different variables. In the solution of the following example we illustrate that this may cause problems if used incorrectly.

Exercise 2.4. Define a recursive version of the function `from(n,m)` with n and m integers. The function returns the list of integers from n to m . Assume $n \leq m$. Consider the use of pattern matching in the definition.

Exercise 2.5. Define the function `quicksort` that given a list of integers, returns the list of integers ordered (from lower to large). Give a recursive version using pattern matching.

2.9 Collections and Their Higher Order Functions

We have studied lists in Sect. 2.7. There are other types of collections in Scala. Some of them are mutable and some of them are immutable. We will review some of them here. We start, however, reviewing what mutable and immutable data structures are.

2.9.1 *Mutable and Immutable Data Structures*

We have an immutable data structure when we cannot modify its values. *Modification* is achieved by means of constructing a new data structure. This is the case of lists in Scala. In Scala, we cannot access the i th element of the list and change its value. We proceed defining a *new* list with a different value in the i th position.

For example, if we want to change the 2nd position of the list (1,2,3,4) by 20, we do as follows (check values of x and y after the following code):

```
val x = 1::2::3::4::Nil
val y = x.head::20::x.tail.tail
```

In this case x is the original list and y is the new list. We define y as the elements of x from the 3rd element and adding to the front the first one of x and the number 20.

Note that the same type of definition is valid if we use `var` instead of `val` (we can check the values of variables x and y after the following code):

```
var x = 1::2::3::4::Nil
var y = x
x = x.head::20::x.tail.tail
```

We have mutable data collections when we can access and modify the original structure. In Scala `Array` is a mutable data structure. That is, we can access a given position and modify its value. The following code permits to visualize that the array is mutable.

```
val x = Array(1,2,3,4)
val y = x
x(1)=20
```

If we check the values of Arrays x and y after the execution of this code, we see that both contain `Array[Int] = Array(1, 20, 3, 4)`. Note that this was not the case with Lists.

In functional programming we prefer immutable objects, for the same reasons that we prefer constants to variables, and that we avoid loops. We consider that they are safer for programming and of a higher-level. We will discuss in Sect. 6.4 efficiency of immutable objects. Although at a first glance it seems that we need to copy the whole structure in order that a function returns an immutable object, this is not always so.

Note also that the way we modify a list and an array in the previous examples is different. The modification of an array was *an assignment* (and thus, imperative-like) while to *update* the list what we did was to build one with the first element, the number 20, and the remaining part. If `x` is a list, it is not allowed to do `x(1)=20`

Mutable and immutable objects. An object is mutable when we can modify (via assignment) one of its components. An object is immutable if no modifications are allowed.

2.9.2 Mutable and Immutable Collections

We list below some of the collections that exist in Scala. We will review them together with some of their methods. The language offers a large number of other collections and functions that we will not describe here. Aspects related to collections are defined and implemented in different places (including classes, traits⁶ and modules). We will not go into these details. See Scala documentation for details.

- List. As we have seen above, we create them with `Nil`, `::`, `List`. Recall from the discussion above that they are immutable.
- Array. We can have arrays of any type. Recall from the above description that they are mutable. The following code illustrates this type.

```
var myArray:Array[String] = new Array[String](10)
var my2ndArray = Array("Reus","Paris","London")
myArray(0)="This is the first position"
myArray(1)="This is the second position"
```

- Range. It represents a sequence of integers from a given number to another one (not included) with a given positive integer step. The construction `n to m` for integers `n` and `m` generates the sequence of integers between `n` and `m` both included. The construction `n until m` for integers `n` and `m` generate a similar sequence but with `m` not included. We illustrate below the answer of the system for `to` and `until`⁷

```
scala> val oneToTen = 1 to 10
oneToTen: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> val oneToNine = 1 until 10
oneToNine: scala.collection.immutable.Range =
  Range(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

⁶Traits are discussed in Sect. 4.8.

⁷Properly speaking, `to` and `until` are methods of the class `integer` that receive a single parameter. So, `4.to(16)` is also a valid construction. See Sect. 4.2.1 for details. There are functions `to` and `until` with two parameters, the first one is the limit and the second the step. Thus, `4.to(16,2)` returns the range with values (4, 6, 8, 10, 12, 14, 16).

- **Set.** We can define sets of any type of objects. See e.g. `Set(1, 2, 3)`. Elements can only be once in a set. We can add elements to the set by means of `+` and remove elements by means of `-`. In addition we have `union` and `intersect` to compute, respectively the union and intersection of two sets.

```
val s1 = Set(1, 2, 3, 4)
val s2 = Set(3, 4, 5, 6)
s1.intersect(s2)
s2.union(s2)
```

- **Map.** It defines a mapping from values of type `A` to values of type `B`. That is, a map returns an object of type `B` given one of type `A`. The following operations are defined `get` (which corresponds to the application of the mapping), `+` (to add an additional element to the map) and `-` (to remove one of the elements). Consider the following examples.

```
val map1:Map[String,Int]= Map("one"->1, "two"->2,
                              "three"->3, "four"->4, "five"->5)
map1.get("one")
map1 + ("six" -> 6)
map1 - ("one")
```

For these collections we can compute their `head`, `tail` and whether they are empty using `isEmpty`. For example, we can compute `map1.head`.

For each collection with name `NameOfCollection` we can create an object of this type from a set of elements by means of calling `NameOfCollection` with the elements as their parameters. We have already seen some examples above for lists, arrays, sets and maps. Details on why the code `List(1, 2, 3)`, `Set(1, 2, 3)`, or even `Iterator(1, 2, 3)` works correctly can be found in [22].

Exercise 2.6. Define a curried version of the function `curryF` that returns

$$a * (b + c)^2$$

The definition should be done so that the following two expressions work correctly.

```
curryF (2) (3) (1)
List(1.0, 2.0, 3.0).map(curryF(2) (3))
```

2.9.3 Some Imperative Construction on Collections

Given a collection, there are ways to process their elements in an imperative way. For example, `foreach` applies a function to each of the elements. The function is expected to have side effects, as `foreach` always returns an object of type `Unit`. For example, check that

```
val result = list1 foreach ((a)=>1)
```

is a Unit.

Test also the following, and observe that after the execution variable `r1` is Unit while `r2` is 6 as the block returns `total` instead of returning the output of `foreach`.

```
val list1 = List(1,2,3)
val array1 = Array(1,2,3)
val range1 = 1 to 3
val set1 = Set(1,2,3)
val map1:Map[String,Int]= Map("one"->1,"two"->2,
    "three"->3,"four"->4,"five"->5)
val r1 = {var total = 0; list1 foreach {
    (i)=>{ total = total + i }}}
array1 foreach {(i)=>{ println(i+2) }}
val r2 = {var total = 0;
    range1 foreach {(i)=>{ total = total + i }}; total}
set1 foreach println
map1 foreach println
```

We have seen in Sect. 2.2 that we can do a `for` in Scala using the following code.

```
for (i <- 1 to 10) { statement }
```

Again, we need here that the statement has some side effects.

In fact, we can use `for (i <- sequence) {statement}` to iterate on the elements of any collection. In the expression `i` is a variable that takes values in all the elements of the sequence. We use this construction with different types of collections in the following expressions.

```
for (i <- List("alpha","beta","gamma")) { println(i) }
for (i <- Array(("one",1),("two",2))) { println(i) }
for (i <- Set(1,4,2,4).union(Set(4,3))) { println(i) }
for (i <- Map("one"->1,"two"->2,"three"->3,
    "four"->4,"five"->5)) {
    println(i) }
for (i <- 1 to 5 by 2) {
    print(Map(1->"one",2->"two",3->"three",
        4->"four",5->"five")(i)) }
```

2.9.4 Higher-Order Functions for Collections

There exist a few important functions defined on collections that help us to apply functions on their elements. We review some of them below. For simplicity, we will focus on lists, and examples are mainly on lists and ranges, but they exist for the other types of collections.

- **map.** This higher-order function permits to apply a function to all the elements of the list. Given the list `l` and the function `f` we apply the function to all the elements of the list using `l.map(f)`. For example, we can compute the factorial of all the elements of a list of integers as follows.

```
val fact: (Int=>Int) =
  (n) => { if (n==0) { 1 } else { n * fact(n-1) }}
(0::1::2::3::4::Nil).map(fact)
```

If the function has type `A => B`, the given list needs elements of type `A` and `map` returns a list of elements of type `B`.

We can also use `map` to apply a map to a list. For example, using the map `map1` seen in Sect. 2.9.3 we can compute:

```
val map1: Map[String, Int] = Map("one"->1, "two"->2,
  "three"->3, "four"->4, "five"->5)
List("one", "one", "two", "five", "two", "one").map(map1)
```

- **filter.** It selects the elements of a list that satisfy a given function. Therefore, one of the parameters is a function that returns a Boolean. Let us denote the original list by L and this function by f , then `filter` returns a list with the elements that make f true. I.e., $\{l \in L | f(l) = \text{true}\}$.

```
val notMultiplesOf3: (Int=>Boolean) = (n) => {
  n % 3 != 0 }
(1::2::3::4::5::6::7::8::9::10::Nil).filter(notMultiplesOf3)
```

This returns `List(1, 2, 4, 5, 7, 8, 10)`.

Note that we can use in `filter` any anonymous function. For example, to select the multiples of 5 in the range 1 .. 300 that are also multiple of 3 we can proceed as follows.

```
(1 to 300).filter((n) => { n % 3 == 0 & n % 5 == 0 })
```

- **take.** Given n , it selects the first n elements of the list.

```
scala> (1 to 10).take(5)
res8: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5)
```

- **zip.** It combines the values of two lists building a list of pairs or tuples (each pair contains an element of each list). For example,

```
(1::2::3::4::5::Nil).zip(0::1::2::3::4::Nil)
```

This expression returns the following:

```
List[(Int, Int)] = List((1,0), (2,1), (3,2), (4,3), (5,4))
```

When the two lists are of different length, the resulting function has the length of the shortest one.

- **zipped.** It is similar to **zip**. In this case, given a pair of two lists, returns a list of pairs. For example, the following expression combines two lists.

```
(1::2::3::4::5::Nil, 0::1::2::3::4::Nil).zipped
```

Compare this expression with the one above for **zip**.

We cannot visualize the whole structure, but we can see individual elements with **head** and **tail** and we can see the following. With

```
(1::2::3::4::5::Nil, 0::1::2::3::4::Nil).zipped.head
```

we obtain

```
res81: (Int, Int) = (1, 0)
```

and with

```
(1::2::3::4::5::Nil, 0::1::2::3::4::Nil).zipped.tail.head
```

we obtain

```
res82: (Int, Int) = (2, 1)
```

- **partition.** This function returns a pair of two lists on the basis of a predicate. The first list contains all elements that make true the predicate, and the second list contains the elements that make false the predicate. For example, the following code

```
val notMultiplesOf3: (Int => Boolean) = (n) => { n % 3 != 0 }
(1::2::3::4::5::6::7::8::9::10::Nil).partition(notMultiplesOf3)
```

returns

```
(List(1, 2, 4, 5, 7, 8, 10), List(3, 6, 9))
```

- **find.** It returns the first element of the list that satisfies a given predicate. For example, the following code returns 1 (as it is the first element of the list which satisfies **notMultiplesOf3**).

```
val notMultiplesOf3: (Int => Boolean) = (n) => { n % 3 != 0 }
(1::2::3::4::5::6::7::8::9::10::Nil).find(notMultiplesOf3)
```

The following example finds the first multiple of 7 and 5 among the integers below 100. It returns, of course, 35.

```
(1 to 100).find((n) => { n % 7 == 0 && n % 5 == 0 })
```

- **drop.** This function returns the collection without the first *n* elements. For example,

```
scala> (1 to 10).drop(5)
res87: scala.collection.immutable.Range = Range(6, 7, 8, 9, 10)
```

- **dropWhile.** This function is similar to the previous one, but with a Boolean function. Elements are removed until the function becomes false.

```
scala> (1 to 10).dropWhile((a)=> a%5 != 0)
res89: scala.collection.immutable.Range =
  Range(5, 6, 7, 8, 9, 10)
```

- **foldLeft**. This function is used to combine the values of a list according to a given function. Let $[x_1, x_2, x_3, x_4 \dots, x_n]$ be the list, then given a value e_0 and the function f the function `foldLeft` computes

$$f(\dots f(f(f(f(e_0, x_1), x_2), x_3), x_4) \dots, x_n).$$

In the following example we use `foldLeft` to define a function to add all the elements of a list.

```
val sum: (List[Int] => Int) = (l) => l.foldLeft(0)((a,b) => a+b)
```

we call this function with `List(0, 1, 2, 3, 4).sum`. Note that we can just use `foldLeft` to compute the sum of elements of any list without defining `sum` explicitly. This is done as follows.

```
List(0, 1, 2, 3, 4).foldLeft(0)((a,b) => a+b)
```

Let us consider the signature of the elements involved in `foldLeft`. Let us first assume that the elements of the list are of type A . Then, we have that f is a function whose second element is of type A . Then, note that the result of the function does not necessarily require to be also of type A . Let us use B for the type of the output of the function. Then, note that the first argument of f should also be of type B . Therefore $f: (B, A) => B$. Because of this, e_0 is also of type B and the output of `foldLeft` is also of type B .

Taking into account this fact we use `foldLeft` to transform a list of integers into a string with the numbers in characters:

```
List(1,2,3).foldLeft("")((a,b)=>a+
  Map(1->"one",2->"two",3->"three",4->"four",5->"five",
    6->"six",7->"seven",8->"eight",9->"nine",0->"zero")(b)+"")
```

- **foldRight**. Similar to `foldLeft` but elements are associated on the right. In the case of the sum, both `foldLeft` and `foldRight` are equally valid to define the sum.

```
val sum: (List[Int] => Int) = (l) => l.foldRight(0)((a,b) => a+b)
```

- **reduceLeft**. It is similar to `foldLeft` but without the initial value e_0 . So, it computes:

$$f(\dots f(f(f(x_1, x_2), x_3), x_4) \dots, x_n).$$

For example, we can compute the factorial of 10 as follows.

```
(1 to 10).reduceLeft((a,b)=>a*b)
```

- **reduceRight**. This is also similar to `foldRight` without e_0 .

Scala permits to use wildcards in the anonymous function we pass to these higher-order functions. For example, we can replace $(a) \Rightarrow (x \% 5 \neq 0)$ by $_ \% 5 \neq 0$, and $(a, b) \Rightarrow a + b$ by $_ + _$. Note that in the first case we replace one variable, and in the second case two. The following expressions are valid in Scala and are equivalent to two expressions above.

```
val sum: (List[Int] => Int) = (l) => l.foldLeft(0) (_ + _)

(1 to 10).dropWhile(_ % 5 != 0)
```

Observe that the use of wildcards here is different to their use in pattern matching, where the value associated to a wildcard was not considered.

Exercise 2.7. Given a list of natural numbers, add the positive ones (and ignore the negative ones).

Exercise 2.8. Implement the internal product of two vectors using some of these higher-order functions.

Exercise 2.9. Implement recursively first without pattern matching and later using it the higher-order functions discussed in this section.

2.10 List Comprehension

List comprehensions are expressions that roughly correspond to the description of sets in terms of the properties of their elements. For example, we write in mathematics:

$$\{y | x \in N, x > 0, x < 10, y = x^2\}.$$

The general expression for comprehensions in Scala is:

```
for (enums) yield { expressionOnEnums }
```

In `enums` we describe which elements are considered, and then we generate them with `expressionOnEnums`. For example, the following expression generates the elements in the previous set.

```
scala> for (i <- 1 to 9) yield { i*i }
res9: scala.collection.immutable.IndexedSeq[Int] =
      Vector(1, 4, 9, 16, 25, 36, 49, 64, 81)
```

In a comprehension, we can have multiple generators. The following example adapted from [25] generates pairs.

```
for (i <- Iterator.range(0, 20);
     j <- Iterator.range(i + 1, 20) if i + j == 32)
  println("(" + i + ", " + j + ")")
```

Scala: From a Functional Programming Perspective
An Introduction to the Programming Language

Torra, V.

2016, XIII, 124 p. 7 illus., Softcover

ISBN: 978-3-319-46480-0