The Basics of Functional Programming

In a wider sense, functional programming means focusing on the functions and a functional programming language enables the definition of programs that focus on functions. In a functional language, a function is a value of the same status as, for instance, an integer or a string. In a restricted sense … it will be seen later.

1 Installing tools

Install the following tools on your machine:

- JDK, the Java Development Kit, **version 8 or 11**
- The Intellij IDEA Community (https://www.jetbrains.com/idea/download/). To start working with Scala in IntelliJ IDEA you need to download and enable the Scala plugin. If you run IntelliJ IDEA for the first time, you can install the Scala plugin when IntelliJ IDEA suggests you to download featured plugins.
- The SBT interactive build tool (https://www.scala-sbt.org/download.html)

2 Testing some expressions

Scala worksheets are normal Scala files, except that they end in .sc instead of .scala. The results appear next to the expressions in your object.

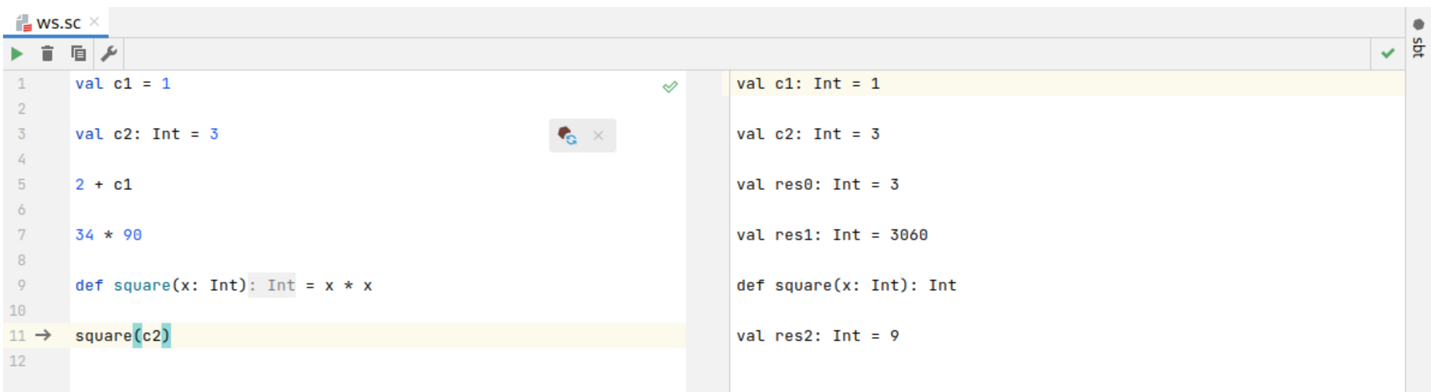

Figure 1. A Scala worksheet

Besides the calculation of simple expressions, values and functions can be defined (see Figure 1). Values are immutable variables and start with the reserved word **val**. Function definitions start with the reserved word **def**. Function parameters come with their type, which is given after a colon. If a return type is given, it follows the parameter list or it can be inferred by the compiler.

```
def square(x: Double) : Double = x * x
```

or

```
def square(x: Double) = x * x
```

Primitive types are as in Java, but are written capitalized, e.g.:

- Int - 32-bit integers
- Double - 64-bit floating point numbers
- Boolean - Boolean values true and false

Scala Worksheet

Clone `emptyScala3Project` project (link on moodle) and open it in Intellij IDEA:

1. Change the project name in `build.sbt` to "lab01"
2. Open the src/main as in Figure.Right-click on the scala folder in main and select "New" > "Scala Worksheet"
3. Choose a name for your worksheet.
4. Now you can type some Scala code into the worksheet. Test the expressions shown in Figure 1.

Conditional Expressions: Exercise 1

Scala has a conditional expression if-else, which is used for expressions, not statements. Example:

```
def abs(x: Int) = if (x >= 0) x else -x
```

`x >= 0` is a predicate, which is a function of type Boolean. An if/else has a value, namely the value of the expression that follows the if or else.

Boolean expressions b can be composed of constants (`true` and `false`), negation (`!`b_1), conjunction (b_1 `&&` b_2), disjunction (b_1 `||` b_2) and of the usual comparison operations: e_1 `<=` e_2, e_1 `>=` e_2, e_1 `<` e_2, e_1 `>` e_2, e_1 `==` e_2, e_1 `!=` e_2, where b_i and e_i are, respectively, Boolean expressions or general expressions.

The else part is not mandatory. Thus, the following sequence of expressions is valid:

```
val x=2
if (x > 0) 1 // Equivalent to "if (x > 0) 1 else ()"
```

Write a function `lessThan` with two arguments that returns `true` with the first argument is less than the second one, otherwise it should return `false`. Is it necessary to use a conditional expression if-else?

Conditional Expressions: Exercise 2

Without using `||` and `&&`, write functions `and` and `or` such that for all argument expressions x and y, `and(x, y) == x && y` and `or(x, y) == x || y`.

Try to use only one conditional expression if-else in your implementation and test with the following expressions:

```
assert(and(true, true)==true)
assert(and(true, false)==false)
assert(and(false, true)==false)
assert(and(false, false)==false)
assert(or(false, false)==false)
assert(or(true, true)==true)
assert(or(true, false)==true)
assert(or(false, true)==true)
println("tests passed")
```

3 Recursion

A recursive function has a right-hand side that calls itself. The return type is optional for non-recursive functions, but the recursive ones need an explicit return type in Scala. Recursive functions are considered the bedrock of functional programming and they should be used instead of loops.

Example of a recursive factorial function:

```
def factorial(n: Long): Long =
  if (n == 0) 1 else n * factorial(n - 1)
```

When it is necessary a type of integer that can become really large, the type `BigInt` can be used instead. Integer literals and operators such as `*` and – can be used with values of that type, but it is not a built-in type.

Basic exercises

1 By completing the code bellow, write a recursive function `sumDown` with two arguments of type Int sum all values between the value received and zero.

```
def sumDown(x: Int, sum: Int) : Int = ???
// Test
assert(sumDown(5,0) == 15)
```

2 Write a recursive function `nSymbol` with three arguments: one indicates the number of times that the symbol (the second argument) should be returned.

```
def nSymbol(i: Int, c: Char, s: String) : String = ???
// Test
assert(nSymbol(5,'*',"") == "*****")
```

3 Write a recursive function `mult` with two arguments that returns the multiplication of the two values. The multiplication is to be computed using sums. For instance, 4 * 3 = 4 + 4 +4 = 3 + 3 + 3 + 3. Test with the following expressions:

```
assert(mult(4,3) == 12)
assert(mult(0,0) == 0)
assert(mult(0,1) == 0)
assert(mult(1,0) == 0)
assert(mult(-3,-3) == 9)
assert(mult(-3,4) == -12)
assert(mult(3, -4) == -12)
```

4 The greatest common divisor (GCD) of two integers `a` and `b` is defined to be the largest integer that divides both `a` and `b` with no remainder. For example, the GCD of 16 and 28 is 4.

The idea of the Euclid's algorithm is based on the observation that, if `r` is the remainder when `a` is divided by `b`, then the common divisors of `a` and `b` are precisely the same as the common divisors of `b` and `r`. Thus, we can use the equation to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example, GCD(206,40) = GDC(40,6) = GDC(6,4)=GDC(4,2)= GCD(2,0), which is 2. It is possible to show that starting with any two positive integers and performing repeated reductions will always eventually produce a pair where the second number is 0. Then the GCD is the other number in the pair.

Define a recursive function based on the Euclid's Algorithm.

Note: The operator `%` provides the remainder of the integer division between two numbers.

5 The following pattern of numbers is called *Pascal's triangle*:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
   ...
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. By completing the code bellow, write a recursive function that computes elements of Pascal's triangle, considering each column and row.

```
def pascal(row: Int, col: Int): Int = ???

def pascal_row(row : Int, maxCols: Int): String = {
  val srow =
    for {
      col <- (1 to row)
      v = pascal(row, col)
    } yield s"$v "
  " " * (maxCols - row) + srow.mkString
}

def pascal_triangle(n : Int): String = {
  val stri =
    for {
      row <- (1 to n)
    } yield pascal_row(row, n)
  "\n" + stri.mkString("\n")
}
```

4 Tail recursion

If a function only calls itself as its last action, the function's stack frame can be reused. This is called tail recursion. In the example of the factorial function, presented before, tail recursion was not used, although there is a call to factorial, it is not the last action. Aftar the call there is a multiplication:

```
def factorial(n: Long): Long =
  if (n == 0) 1 else n * factorial(n - 1)
```

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called tail-calls.

In Scala, only directly recursive calls to the current function are optimized. One can require that a function is tail-recursive using a `@tailrec` annotation:

```
@tailrec
def recursiveFunction(a: Int): Int = ...
```

If the annotation is given, and the implementation of `recursiveFunction` was not tail recursive, an error would be issued or nothing will happen, depending on the IDE in use. In addition, Scala only optimizes directly recursive calls back to the same function making the call. If the recursion is indirect, as in the following example of two mutually recursive functions, no optimization is possible:

```
def isEven(x: Int): Boolean = if (x == 0) true else isOdd(x - 1)

def isOdd(x: Int): Boolean = if (x == 0) false else isEven(x - 1)
```

Here is an implementation of the factorial function with tail recursion:

```
import scala.annotation.tailrec

def factorialTailRec(n: BigInt): BigInt = {

  @tailrec
  def fact_aux(acc:BigInt, n:BigInt):BigInt={
    if (n<=0) acc else fact_aux(acc*n,n-1)
  }

  fact_aux(1,n)
}
```

The previous implementation of factorial, even with BigInt, will probably result in an error java.lang.StackOverflowError if a big value is passed as an argument. This last implementation solves the stack overflow problem.

Exercise

Provide a solution for the third exercise proposed in the third section (recursive *mult*) using tail recursion.