

02 Lab

Functional Programming

Mirko Viroli, Roberto Casadei, Gianluca Aguzzi
{mirko.viroli, roby.casadei, gianluca.aguzzi}@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2024/2025

Lab 02: Outline

Outline

- Practice with Functional Programming (FP)
- Get acquainted with Scala
- Get acquainted with REPL/VS Code

Premise

- Product quality as a consequence of **process quality**
 - ▶ Use the VCS (git) also to **document** your development process: i.e., use **commits** to properly highlight solutions or refactoring steps
- FP, as a **paradigm**, requires a mind shift
 - ▶ This may not be easy, at the beginning
 - ▶ Strive to fully understand the **concept** behind the exercises (don't be satisfied when something "just works")
- Scala, as a **language**, requires practice
 - ▶ Distinguish syntactic vs. semantic aspects
- REPL and Scala, as **tools**, have pros&cons depending on context
 - ▶ Use REPL for quick experiments (e.g., language-oriented)
 - ▶ Use Scala for development and to evaluate alternative designs

Recap: Scala 3 REPL (Read Eval Print Loop)

`$ scala`

- `:help`
- `:reset` to forget all expression results and named terms
- `:type <expression>` to just show the expression type
- `:q` to quit the REPL

additional commands..

- `:load file.scala` interpret lines in file.scala

Note!

- You can also use `sbt console` to start the REPL with the project dependencies

Tasks – part 1 (warm up)

1. Fork <https://github.com/unibo-pps/pps-lab02>

- ▶ File → Open and select your cloned repo's root directory
- ▶ The repo contains the code shown in lecture 02
- ▶ Write a “Hello, Scala” main program
- ▶ The main code is in the body of an `object` extending `App`

2. Experiment with REPL (10 minutes)

- ▶ Run on the REPL simple examples of code from the lecture 02 on Scala/FP (take code from the repository cloned)
- ▶ Try variations, explore autonomously, and ask in case of doubts
- ▶ Copy and past code as in any other terminal

Tasks – part 1 (warm up)

2. On “exploring autonomously” in the REPL:

- ▶ Run “sbt console” on the root of the project (or use the REPL directly)
- ▶ Explore the class definitions in u02 by experimenting with code examples
- ▶ For example, with Currying:
 - Copy `mult` and `multCurried` from the lecture code
 - Try them with different parameters: `mult(3,4)`, `multCurried(3)(4)`
 - Create variations like changing the operation or parameter types
 - Try partial application: `val multiplyBy3 = multCurried(3)`
- ▶ Document your experiments in a file
 - Try to implement similar functions like `divide` and `divideCurried`
 - Test them with different inputs and edge cases
 - Record your observations and code examples in your file
 - This will be shared with us as a part of the lab

Tasks – part 2 (functions)

3. Get familiar with first-class and higher order functions as well as with the different styles for expressing functions

a) Using match-cases, implement the following function from Int to

$$\text{String: } \textit{positive}(x) = \begin{cases} \text{"positive"} & \text{if } x \geq 0 \\ \text{"negative"} & \text{if } x < 0 \end{cases}$$

in both of the following styles: (i) val assigned to function literal (lambda) and (ii) method syntax.

b) Implement a *neg* function that accepts a **predicate** on strings (i.e., a function from strings to Booleans) and returns another predicate on strings, namely, one that does the exact opposite; write the type first, and then define the function both as a val lambda and with method syntax

```
val empty: String => Boolean = _ == "" // predicate on strings
```

```
val notEmpty = neg(empty) // which type of notEmpty?
```

```
notEmpty("foo") // true
```

```
notEmpty("") // false
```

```
notEmpty("foo") && !notEmpty("") // true.. a comprehensive test
```

c) Make neg work for generic predicates, and write tests to check it (therefore, neg will be generic: `def neg[X]...`).

Tasks – part 2 (functions)

4. Currying

- ▶ Implement a predicate that checks whether its arguments x, y, z respect the relation $x \leq y = z$, in 4 variants (curried/non-curried \times val/def)
 - `val p1: <CurriedFunType> = ...`
 - `val p2: <NonCurriedFunType> = ...`
 - `def p3(...)(...)(...): ... = ...`
 - `def p4(...): ... = ...`
 - Notice: function types and function literals are syntactically similar

5. Create a function that implements functional compositions

$$(f \circ g)(x) = f(g(x))$$

- ▶ Signature: `compose(f: Int => Int, g: Int => Int): Int => Int`
- ▶ Example: `compose(_ - 1, _ * 2)(5) // 9`
- ▶ Create a generic version of `compose`
 - What signature? Is there any constraint?

6. Create a function that implements functional compositions with three arbitrary functions (i.e., $f \circ g \circ h$)

- ▶ Signature should support arbitrary types:
`composeThree[A,B,C,D](f: C => D, g: B => C, h: A => B): A => D`
- ▶ Example: `composeThree(_ + "!", _.toString, _ * 2)(3) // "6!"`
- ▶ Can you implement this by reusing your generic `compose` function?

Tasks – part 3 (recursion)

7. Create a recursive function to calculate the power of a number

- ▶ Signature: `power(base: Double, exponent: Int): Double`
- ▶ Example: `(power(2, 3), power(5, 2)) // (8.0, 25.0)`
- ▶ Hint: $base^{exponent} = base \times base^{exponent-1}$ and $base^0 = 1$
- ▶ **Note:** The function should just work for positive exponents
- ▶ Try to implement the same function using tail recursion

8. Create a function to reverse the digits of an integer using recursion.

- ▶ Signature: `reverseNumber(n: Int): Int`
- ▶ Example: `reverseNumber(12345) // 54321`
- ▶ **Hint:** Use tail recursion with an accumulator to iteratively build the reversed number. For instance, you could define a helper function that takes the remaining part of the number and the current reversed number as parameters.
- ▶ **Hint:** To extract individual digits from a number, remember:
 - `n / 10` gives the number without the last digit (integer division)
 - `n % 10` gives just the last digit
 - Example: for 123, `123 / 10 = 12` and `123 % 10 = 3`

Tasks – part 4 (sum types, product types, modules)

9. Define a sum type `Expr` to represent arithmetic expressions.

► Include the following variants:

- **Literal**: a product type representing a numeric constant.
- **Add**: a product type representing the addition of two sub-expressions.
- **Multiply**: a product type representing the multiplication of two sub-expressions.

► Define a module that provides operations on expressions:

- `evaluate(expr: Expr): Int` - Recursively compute the numerical result of the expression.
- `show(expr: Expr): String` - Recursively generate a string representation of the expression.
- **NB!** For string concatenation in the `show` function, you can use string interpolation or the `+` operator, e.g.,
`"(" + leftExprString + " + " + rightExprString + ")"` for formatting expressions.

► Consider using a TDD (Test-Driven Development) process to design and test your functions.

- Write tests using JUnit as shown in `src/main/test/task5`

Tasks – part 5 (more functional combinators)

10. Look at tasks5.Optionals:

- ▶ This follows the concept of Java `Optional` but with an ADT approach, therefore describing the `Optional` with two cases:
 - `Maybe[A] (value: A)`: the value is present
 - `Empty()`: the value is not present
- ▶ Look at the implementation and the tests
- ▶ Implement **map**: a function that transform the value (if present)– for more details look at the tests

```
map(Maybe(5))(_ > 2) // Maybe(true)
map(Empty())(_ > 2) // Empty
```
- ▶ **filter**: a function that keeps the value (if present, otherwise the output is `None`) only if it satisfies the given predicate.

```
filter(Maybe(5))(_ > 2) // Maybe(5)
filter(Maybe(5))(_ > 8) // Empty
filter(Empty())(_ > 2) // Empty
```

The signature can be straightforwardly guessed by the examples.