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. 2023/2024

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

Tasks – part 1 (warm up)

1. Fork `https://github.com/unibo-pps/pps-23-24-lab02`
   - `File -> Open Folder` and select your cloned repo's root directory
   - The repo contains the code shown in lecture 02
   - Open in with Metals installed
   - 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 2a (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 `String`: $positive(x) = \begin{cases} \text{“positive”} & \text{if } x \geq 0 \\ \text{“negative”} & \text{if } x < 0 \end{cases}$
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 2b (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?

Tasks – part 3 (recursion)

6. Create a function to compute the greatest common divisor (GCD) of two integers a and b
   - The GCD is the largest positive integer that divides both a and b without leaving a remainder.
   - Signature: `gcd(a: Int, b: Int): Int`
   - Example: `(gcd(12, 8), gcd(14, 7)) // (4, 7)`
   - Hint: Use the Euclidean algorithm, which states that if a and b are two integers with $a > b$, then $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$.
   - Hint: For mod, use the same operator you would use in Java
   - Write a tail-recursive version of the function using the same approach as the tail-recursive Factorial function seen in lecture 02.

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

7. Define a set of geometric shapes and methods for calculating their perimeter and area
   - Define an enum `Shape`
   - Define concrete types `Rectangle`, `Circle`, and `Square`; these product types should exhibit the typical geometric properties you would expect to characterise the corresponding shapes
   - Define a module with two methods `perimeter(shape: Shape): Double` and `scale(shape: Shape, alpha: Double): Shape` for computing perimeter and scaling a shape, respectively
     - scaling means multiplying the dimensions of the shape by a factor α
   - You may want to address this exercise through a TDD process

Tasks – part 5 (more functional combinators)

8. 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.