03 Lab
More Functional Programming in Scala
Seqs, Streams, and more

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 03: Outline

Outline

- Continue with the practice of functional programming
- Exercise with lists and streams (these are key functional data structures!)

Getting started

- Fork repository `https://github.com/unibo-pps/pps-lab03`
- The repo contains code from lectures 02 and 03
- Solve the exercises of the following slides in a proper module and evaluate your solutions through a main program or test class

Tasks – part 1 (lists)

- Look at the file `u03/Sequence.scala`
- Provide explicit implementations for the missing methods in the `Sequence` module, ensuring each behavior aligns with the specifications in `SequenceTest.scala`.
- Follow the suggested order for the mandatory methods:
  - Mandatory: `skip`, `zip`, `concat`, `reverse`, `flatMap`
- **Optional** If you want to exercise more, you can also implement the following methods:
  - Optional: `min`, `evenIndices`, `contains`, `distinct`, `group`, `partition`
- Look at the corresponding tests in `u03/SequenceTest.scala` to observe the expected behavior.
- **Note!** It is recommended to first implement the mandatory methods. Once completed, feel free to continue with the optional methods as time permits.

Tasks – part 2 (more on lists)

1. Consider `Person` and `Sequence` as implemented in class slides. Create a function that takes a sequence of `Persons` and returns a sequence containing only the `courses` of `Teacher` in that list
   - Hint 1: you essentially need to combine `filter` and `map`
   - Hint 2: there is a very concise solution that reuses `flatMap`
2. (Hard) Implement `foldLeft` function that, starting from a default value, "fold over" sequences by "accumulating" elements via a binary operator.
   - Idea: given a list `[3, 7, 1, 5]` and a default value `0`, a left-fold (resp., right-fold) through e.g. operator `+` is given by `(((0 + 3) + 7) + 1) + 5`
   - Note: the accumulator type may differ from the element type

```
val lst = Cons(3,Cons(7,Cons(1,Cons(5, Nil()))))
foldLeft(lst)(0)(_ - _) // -16
```

3. Consider again `Person` and `Sequence`. Create a function that takes a sequence of `Persons` and returns the total number of courses taught by all `Teacher` in that list. Use a combination of `filter`, `map`, and `foldLeft`.
   - Hint: first filter for `Teacher`, then map each teacher to the length of their `courses` list, and finally apply `foldLeft` to sum up these counts.
4. **Note!** In this case no test are given, so you have to write your own tests!!!

Taks – part 3 (streams)

6. Consider the `Stream` type discussed in class. Define a function, called `takeWhile(s)(n)`, that returns the first `n` elements of the stream `s` that satisfy a given predicate.

```
val stream = Stream.iterate(0)(_ + 1)
Stream.toList(Stream.takeWhile(stream)(_ < 5))
// Cons(0, Cons(1, Cons(2, Cons(3, Cons(4, Nil())))))
```

7. Implement a generic function `fill(n)(k)` that creates a stream of `n` elements, each of which is `k`.

```
Stream.toList(Stream.fill(3)("a")) // Cons(a, Cons(a, Cons(a, Nil())))
```

8. Implement an infinite stream for the Fibonacci Numbers: `https://en.wikipedia.org/wiki/Fibonacci_number`

```
val fibonacci: Stream[Int] = ???
Stream.toList(Stream.take(fibonacci)(5)) // Cons(0, Cons(1, Cons(1, Cons(2, Cons(3, Nil())))))
```

9. (Optional) Implement a generic function `interleave(s1, s2)` that merges two streams by alternating elements from each.

```
val s1 = Stream.fromList(List(1, 3, 5))
val s2 = Stream.fromList(List(2, 4, 6, 8, 10))
Stream.toList(Stream.interleave(s1, s2))
// Expected output: Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Cons(8, Cons(10, Nil()))))))))
   )
```

10. (Optional) Implement a function `cycle(list)` that creates an infinite stream by cycling through the elements of a given finite list in an innovative way.

```
def cycle[A](lst: Sequence[A]): Stream[A] = ???
val repeat = cycle(Cons('a', Cons('b', Cons('c', Nil()))))
Stream.toList(Stream.take(innovativeStream)(5))
// Expected output: Cons(a, Cons(b, Cons(c, Cons(a, Cons(b, Nil())))))
```