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

Mirko Viroli, Roberto Casadei
{mirko.viroli,roby.casadei}@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2020/2021

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-20-21-lab03`
- The repo contains code from lecture 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
- Suggested code organisation
  - Do not touch the original sources
  - Write your solutions in a separate module (object)
  - Write a test suite (class) to test your solutions; use one or more test cases (methods) for each exercise (use the provided examples as a starting point)

Tasks – part 1 (lists)

1. Consider the `List` type introduced in class. Analogously to `sum` and `append`, create the following functions:
   a) `def drop[A](l: List[A], n: Int): List[A]`

```
val lst = Cons(10, Cons(20, Cons(30, Nil())))
drop(lst,1) // Cons(20,Cons(30, Nil()))
drop(lst,2) // Cons(30, Nil())
drop(lst,5) // Nil()
```

   b) `def flatMap[A,B](l: List[A])(f: A => List[B]): List[B]`

```
flatMap(lst)(v => Cons(v+1, Nil())) // Cons(11,Cons(21,Cons(31,Nil())))
flatMap(lst)(v => Cons(v+1, Cons(v+2, Nil())))
// Cons(11,Cons(12,Cons(21,Cons(22,Cons(31,Cons(32,Nil()))))))
```

   c) Write `map` in terms of `flatMap`
   d) Write `filter` in terms of `flatMap`
2. Considering both `List` and `Option`, create the following:

   `def max(l: List[Int]): Option[Int]`

```
max(Cons(10, Cons(25, Cons(20, Nil())))) // Some(25)
max(Nil()) // None()
```

Tasks – part 2 (more on lists)

3. Consider `Person` and `List` as implemented in class slides. Create a function that takes a list of `Person`s and returns a list containing only the `courses` of `Teacher`s in that list
   - Hint 1: you essentially need to combine `filter` and `map`
   - Hint 2: there is a very concise solution that reuses `flatMap`
4. (Hard) Implement two fold functions (`foldLeft`, `foldRight`) that "fold over" lists by "accumulating" elements via a binary operator.
   - Idea: given a list `[3,7,1,5]` a left-fold (resp., right-fold) through e.g. operator + is given by `(((0+3)+7)+1)+5` (resp., `3+(7+(1+(5+0)))`).
   - Note: dealing with empty lists requires providing a default or initial value for the accumulation, to be used on left or on right (the function corresponding to `fold` which fails on empty lists is called *reduce*).
   - Note: the type of the accumulator may be different wrt the type of the elements that are aggregated (two generic variables should be used)
   - Hint: do `foldLeft(l)`, it is easier. Then, `foldRight(l)` $\approx$ `foldLeft(reverse(l))`. Any efficient solution for `foldRight(l)`?

```
val lst = Cons(3,Cons(7,Cons(1,Cons(5, Nil()))))
foldLeft(lst)(0)(_-_) // -16
foldRight(lst)(0)(_-_) // -8
```

Tasks – part 3 (streams)

5. Consider the `Stream` type discussed in class. Define a function, dual to `take(s)(n)`, called `drop(s)(n)`, which drops the first `n` elements of the stream `s`.

```
val s = Stream.take(Stream.iterate(0)(_+1))(10)
Stream.toList(Stream.drop(s)(6))
// => Cons(6,Cons(7,Cons(8,Cons(9,Nil()))))
```

6. Implement a generic function `constant(k)` which generates an infinite stream of value `k`.

```
Stream.toList(Stream.take(constant("x"))(5))
// => Cons(x,Cons(x,Cons(x,Cons(x,Cons(x,Nil())))))
```

7. Implement an infinite stream for the Fibonacci series

```
val fibs: Stream[Int] = ???
Stream.toList(Stream.take(fibs)(8))
// =>  Cons(0,Cons(1,Cons(1,Cons(2,Cons(3,Cons(5,Cons(8,
    Cons(13,Nil()))))))))
```