Practical Sheet nº3

Content

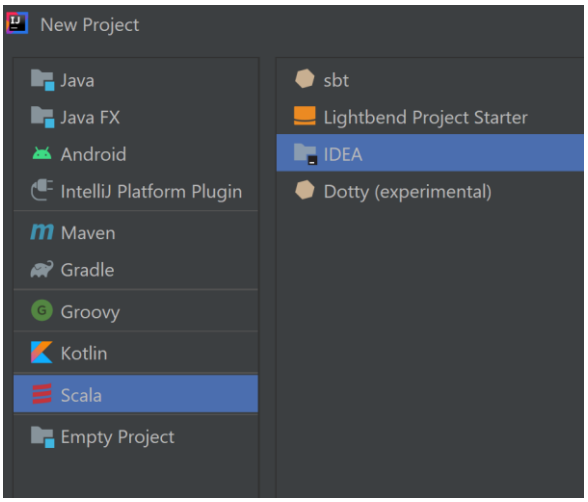
- Tail Recursion
- Lazy evaluation
- Recursive Function Patterns on lists (mapping, filtering and folding)
- Higher Order Functions (*map*, *filter*, introduction to *reduceLeft* and *fold*)

In this class, it is intended to analyze efficiency, work with recursive function patterns on lists (mapping, filtering and folding), and use higher order functions to more quickly define those same functions.

IDE IntelliJ IDEA – Tutorial

Create a new IDEA-based Scala project by doing:

- File/New Project…



- give it a name (e.g., Week3)
- add a new Scala class to the *src* folder (right mouse click New/Scala class/Object) with the name Sheet3
- add to it the following code:

```
object Sheet3 {

  def ex1(x: Int) = {

    println("TODO")

  }


  def main(args: Array[String]): Unit = {

        ex1(1)

  }
```

- run it by clicking on the green arrow ( ) next to the main method

Solve the following exercises by completing the `Sheet3 Object`.

Efficiency

Tail recursion and lazy evaluation are ways of functional programming to improve efficiency of the code.

Exercise 1

1.1. Define three versions of the `factorial` method:
   a) without using an if
   b) using an if
   c) tail recursive

1.2. Define two versions of the `remDup` **polymorphic/generic** method that eliminates consecutive duplicates of a list of elements. Use the `dropWhile` function.
```
remDup(List('a', 'a', 'b', 'c', 'c', 'a', 'a', 'd', 'e', 'e', 'e')
val res01:List[Char] = List(a, b, c, a, d, e)
```
   a) standard recursive
   b) tail recursive

Exercise 2

Consider the following methods (older Scala version might not recognize Lazy lists):

```
def lazyListRange(lo: Int, hi: Int): LazyList[Int] = {
        println(lo)
        if(lo >= hi) LazyList.empty
        else lo #:: lazyListRange(lo+1, hi)
}
def listRange(lo: Int, hi: Int): List[Int] = {
        println(lo)
        if(lo >= hi) List()
        else lo :: listRange(lo+1, hi)
}
```

When you write `listRange(1, 100).take(3).toList` what gets printed and returned? And when you write `lazyListRange(1, 100).take(3).toList` ? Why?

Higher-order functions

Like any other value, a function can be passed as a parameter and/or returned as a result of another function.

Exercise 3

a) Write a function that accepts two lists (with the same length) and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6)` become `List(5,7,9)`.

b) Generalize the function you just wrote so that it is not specific to integers or addition. Name your generalized function `zipWith`.

c) Implement isSorted, which checks whether a List[A] is sorted according to a given comparison function:

```
def isSorted[A](lst: List[A], ordered: (A,A) => Boolean): Boolean
```

d) Implement bubbleSort (recursive) that can be used for ascending/descending ordering.

```
def bubbleSort(data: List[Int], f: (Int, Int) => Boolean): List[Int]
```

Mapping, Filtering, and Folding Functions on Lists

Certain recursive functions on lists follow strict patterns, which allows to classify them in following three categories:

a. **Mapping**: apply the same function to all elements of the argument list, resulting in a list with the same dimension. For example:

```
def doubles(lst:List[Int]):List[Int] = {
	lst match {
		case Nil => Nil
		case x :: xs => (2 * x) :: ( doubles(xs))
	}
}
```

b. **Filtering**: calculate as a result a sub-sequence of the argument list, containing (by order) only the elements that satisfy a certain criterion. For example:

```
def oneOrTwoLetters(lst:List[String]):List[String] = {
	lst match {
		case Nil => Nil
		case x:: xs => {
			if(x.length <= 2) x :: oneOrTwoLetters(xs)
			else oneOrTwoLetters(xs)
```

```
            }
        }
    }
```

c. **Folding**: Combine all the elements in the list through a binary operation. More exactly, iterate a binary operation on a list, which corresponds to the well-known mathematical operations of "sum" or "product" on sets. To the empty list results in any constant value (typically the neutral element of the binary operation, but can be any other value). Examples:

```
def sumList(lst:List[Int]):Int = {
        lst match {
            case Nil => 0
            case x::xs => x + (sumList(xs))
        }
}
def multList(lst:List[Int]):Int = {
        lst match {
            case Nil => 1
            case x::xs => x * (multList(xs))
        }
}
```

Exercise 4

Define the following three new functions and say if they match any of the patterns mentioned above.

a) `paresord` function that receives a list of pairs of numbers and returns only the pairs in that the first component is inferior to the second.

b) `myconcat` function that receives a list of strings and joins them (concatenated) in a single string.

c) `maximum` function that receives a list of pairs of numbers (of type `Double`) and calculates a list with just the largest element of each pair.

Example: List((2.0, 4.0), (3.2, 1,9)) => List(4.0, 3.2)

Functions `map, filter, reduceLeft and foldLeft`

To enable the easy definition of functions that follow the previously mentioned patterns, it is possible to capture these patterns in more abstract recursive functions. For the case of mapping and filtering, we have the following two predefined functions:

```
def map[B](f : (A) => B) : Iterable[B]

def filter(p : (A) => Boolean) : Iterable[A]
```

Note that these functions, said to be of higher order, receive another function as an argument, which specifies, in the case of `map`, which operation to apply to each element of the list, and in the case of filter, what is the filtering criterion.

The use of these functions allows the definition of implicitly recursive functions. For example, a new version of the `doubles` function can be defined as:

```
def doubles(lst: List[Int]) = lst map (x => x * 2)

    or

def doubles(lst :List[Int]) : List[Int] = lst.map( x => x*2)
```

A new version of the `paresord` function can be defined as:

```
def paresord(xs: List[(Int,Int)]): List[(Int,Int)] =

    xs.filter(x => x._1 < x._2)

    or

def paresord(xs: List[(Int,Int)]): List[(Int,Int)] =

    xs filter (x => x._1 < x._2)
```

Using `reduceLeft`, we can simplify the `sumList` and `multList` functions:

```
def sumList(xs: List[Int]) = (0 :: xs) reduceLeft ((x, y) => x + y)

def multList(xs: List[Int]) = (1 :: xs) reduceLeft ((x, y) => x * y)
```

Instead of `((x, y) => x * y))` one can also write shorter: `(_ * _)`

Every _ represents a new parameter, going from left to right. So, `sum` and `product` can also be expressed like this:

```
def sumList(xs: List[Int]) = (0 :: xs) reduceLeft (_ + _)

def multList(xs: List[Int]) = (1 :: xs) reduceLeft (_ * _)
```

The function `reduceLeft` is defined in terms of a more general function, `foldLeft`. `foldLeft` is like `reduceLeft` but takes an accumulator, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list. So, `sum` and `product` can also be defined as follows:

```
def sumList(xs: List[Int]) = (xs foldLeft 0) (_ + _)
```

```
def multList(xs: List[Int]) = (xs foldLeft 1) (_ * _)
```

(exercises with `fold` function will be performed next week)

Exercise 5

Evaluate manually each of the following expressions before executing them.

a) `List(1,2,3,4,5) map (x => x%2 !=0)`
b) `List(1,2,3,4,5) filter (x => x%2 !=0)`
c) `List(5, 6, 23, 3) map ( x => x%3 == 0)`
d) `List(5, 6, 23, 3) filter ( x => x%3 == 0)`
e) `List(1,3,7,8,12,15) filter ( x => x < 7)`
f) `List(List(2,3),List(1,5,3)) map ( x => 7::x)`
g) `List(1, 2, 3, 4, 5) map ( x => List(x))`
h) `List(1,2,3,4,5) filter ( x => x%2 !=0) map ( x => x + 1)`
i) `List(1,2,3,4,5) map ( x => x + 1) filter ( x => x%2 !=0)`

Exercise 6

Consider the following function:

```
def indicative(ind:String, telefs:List[String]) =
      telefs filter ( x => x.substring(0,ind.length).equals(ind))
```

which receives a list of digits with a callsign, a list of lists of digits representing telephone numbers, and selects the numbers that begin with the given callsign. For example:

```
indicative("253",List("253116787", "213448023", "253119905")
returns List("253116787", "253119905")
```

Redefine this function with explicit recursion, that is, avoiding the use of `filter`.

Exercise 7

Implement a function to convert a list of names to a list of abbreviations for those names, as follows: List("José Carlos Mendes", "André Carlos Oliveira") in List("J. Mendes ","A. Oliveira").

Note: use *map* and the *s* String Interpolator to create the final String (e.g. `s"${exp}. ${exp}"`).

Prepending *s* to any string literal allows the usage of variables/expressions directly in the string:

```
val name = "José"
println(s"Hello, $name")  // Hello, José
println(s"1 + 1 = ${1 + 1}") // 1 + 1 = 2
```