Practical Sheet nº2

Content

- Definition (multi-clausal) of methods
- Local definitions
- Definition of recursive methods on Lists
- Polymorphic Methods
- Type aliases

A method that verifies if a list (of any type) is empty can be defined in the following form:

```
def empty[A]( lst: List[A]) = lst.length == 0
```

The lists are defined recursively in the following form:

- Nil the list is empty
- head :: tail (being head the first element ant tail the remaining of the List)

The method to calculate the length of a List might be implemented in the following way:

```
def length( lst: List[Int]): Int = {

        lst match {

                case Nil => 0

                case _ :: tail => 1 + length(tail)

        }

}
```

The method that receives a list of points on the Cartesian plane and calculates the distance from each point to the origin can be defined by:

```
def distance(lstPoint: List[(Double, Double)]): List[Double] = {

        lstPoint match {

                case Nil => Nil;

                case (x, y) :: tail => (Math.sqrt(x * x + y * y)) ::
        distance(tail);

        }

}
```

Exercise 1

a) Define the method `transf` that does the following transformation: receives a list and replaces the 1st with the 2nd element, and the 3rd with the 4th, until the end of the list. For example: transf [1,2,3,4,5,6] ⇒ [**2,1,4,3,6,5**].
b) Define a method that calculates the product of all the elements in a list of numbers.
c) Define a method that, given a list and an element, places it at the end of the list.
d) Define a method that, given two lists (of any type) concatenates them (without using the ++ operator), i.e., creates a list with the elements of the first list followed by those on the second list.
e) Define a sumEl recursive method that receives a list of pairs of Int and calculates the sum of the pairs with indexes 2 and 4.
Example: `sumEl(List((1,2),(3,4),(2,0),(5,6),(1,1)) // 2+0+1+1`
f) In the previous work sheet, we have seen a way to calculate the average of a list of numbers.

```
def average1(lst: List[Double]) = lst.sum / lst.length
```

However, this solution goes through the list two times!

i. Define a method that given a list, calculates a pair containing the length of the list and the sum of its elements – going through the list only once.
ii. Using the previous method, define a method that calculates the average of the elements of a list.

g) Define a method that, given a list (of Double) and a value (Double), returns a pair of lists where the first contains all the elements of the list below this value and the second list contains all other elements.
Example: `metH(List(1.0,2.0,4.0,5.0), 3.0) //(List(1.0, 2.0), List(4.0, 5.0))`
h) Define a method that, given a list of Double, returns the list with the elements that are superior or equal to the average.

Exercise 2

Consider that we want to define methods for manipulating a phone book. So we decided that the information for each entry in the phone book will contain the name, the phone number and e-mail address. We can then make the following definitions:

```
type Entry = (String, String, String)

type LTelef = List[Entry]
```

The method that calculates the known email addresses can be defined as:

```
def emails(lst : LTelef) : List[String] = {
 lst match {
      case Nil => Nil
```

```
            case (_ , _ , email):: tail => email :: (emails(tail))
            }
    }
```

i) Define a method that, given a phone book, produces the list of email addresses of the entries whose telephone numbers are from the fixed network (prefix '2').

j) Define a method that given a phone book and a name, returns the pair with the phone number and the email address associated with that name, in the phone book.

Exercise Extra:

There are methods in which it is more difficult to avoid multiple crossings of the list. Define a **polymorphic** method (without using the List methods *take* or *takeRight*) that, given a list and going through it only once, divides it into two (returning a pair of lists) with the same number of elements (this, of course, if the original list has an even number of elements; in the other case, one of the lists will have one more element).

Example:

```
divide(List(1,2,3,4)) // (List(1, 2),List(3, 4))

divide(List(1,2,3)) // (List(1, 2),List(3))
```