Practical Sheet nº4

Content

- Explicit Recursion (consolidation)
- Recurring Patterns on Lists (*foldLeft, foldRight*)
- (Implicit Parameters, Partially Applied Functions and Currying) - extra

The function foldRight

The `foldRight` function, like `map` and `filter`, allows you to write quickly, without explicit recursion, a large set of functions. The function of this function can be easily understood if we consider that the cons (::) and List() are simply replaced by the two `foldRight` parameters. For example, remembering that

```
List(1,2,3) => 1:: (2:: (3:: List()))
```

we have

```
foldRight (+) 0 List(1,2,3) => 1 + (2 + (3 + 0))

foldRight (*) 1 List(1,2,3) => 1 * (2 * (3 * 1))
```

This allows you to define:

```
def sum(xs: List[Int]) = (xs foldRight 0) (_ + _)

def product(xs: List[Int]) = (xs foldRight 1) (_ * _)
```

Exercise 1

1.1. The concatenation operator (++) can be applied to generic Lists and both logical conjunction (&&) and logical disjunction (||) operators could be applied to a boolean List.
Write methods to represent these behaviors using i) `foldRight` ii) `foldLeft`.

1.2. Define an additional version of the `remDup` polymorphic/generic method (Practical Sheet nº3 – ex 1.2) that eliminates consecutive duplicates of a list of elements using `foldRight` and `dropWhile`.

Exercise 2

It is intended to keep information about the results of the matches of a soccer championship day in the following data structure:

```
type Team = String

type Goals = Int

type Match = ((Team, Goals), (Team, Goals))

type Fixtures = List[Match]
```

Define the following methods using `foldLeft` or `foldRight`:

a) `noItself` which checks that no team plays with itself.
b) `withoutRep` which checks that no team plays more than one game.
c) `teams` which gives the list of teams participating in the Fixtures.
d) `draws` which gives lists of pairs of teams that tied for the day.
e) `points` which calculates the points that each team obtained in the Fixtures (won - 3 points; lost – 0 points; tied - 1 point). The function should return a value of type: `List[(Team, Int)]`

Exercise 3

One way to represent polynomials of a variable is to use lists of pairs (coefficient, exponent)

```
type Pol = List[(Float, Int)]
```

Note that the polynomial may not be simplified. For example,

```
List((3.4f, 3), (2.0f, 4), (1.5f, 3), (7.1f, 5))
```

represents the polynomial $3.4x^3 + 2x^4 + 1.5x^3 + 7.1x^5$

a) Define a method with explicit recursion to order a polynomial in ascending order of degree.
b) Define a method to normalize a polynomial (implement both versions i.e., with implicit and explicit recursion).
c) Define a method to add two polynomials in this representation (implement both versions i.e., with implicit and explicit recursion).
d) Define the method of calculating the value of a polynomial at a point (implement both versions i.e., with implicit and explicit recursion).
e) Define a method that, given a polynomial, calculates its degree.
f) Define a method that calculates the derivative of a polynomial (implement both versions i.e., with implicit and explicit recursion).

Exercises Extra

Exercise 4 - extra

Consider the following two functions `merge` and `insert`.

The first merges two lists ordered in ascending order and returns an ordered list; the second inserts an element in an ascending list:

```
> merge(List(1,4), List(2,3)) // List(1,2,3,4)

> insert(2, List(1, 3)) // [1, 2, 3]
```

A possible definition of `insert` is:

```
def insert(x: Int, lst:List[Int]): List[Int] = {

        lst match{

                case Nil => List(x)
```

```
                   case y::ys => if(x < y) x::y::ys else y :: insert(x, ys)
             }
      }
```

1. Write the `merge` function using `foldRight` and `insert`.
2. Recall the insertion sort ordering algorithm, implemented by the function isort (see theorical slides T2, page 35) and rewrite this function using foldRight to order a list of Int.
3. Transform `isort` into a polymorphic function with an implicit parameter for ordering. Use import scala.math.Ordering

Exercise 5 - extra

Curry the `merge` function (from the previous exercise) to use it in a mapping with a list of lists. Examples:

```
 scala> List(List(1,2),List(3,4)) map merge(List(0,0))
 val res0 = List(List(0, 0, 1, 2), List(0, 0, 3, 4))

 or

 scala> List(List(1,2),List(3,4)) map mergeWith00
 val res1 = List(List(0, 0, 1, 2), List(0, 0, 3, 4))
```

Exercise 6 - extra

Write a polymorphic function using `foldRight` that separates a list into two alternate parts and returns both parts in a tuple.

```
> separate(List(1,2,3,4)) //returns (List(1,3),List(2,4))
```