TC2006 PROGRAMMING LANGUAGES
Assignment 5: Functional Programming in Haskell

In this assignment you will put into practice your knowledge of programming in Haskell. Solve each of the following problems. Only one of the team members must upload the code of your functions to Canvas. Remember to add comments with your team data and the functionality of each programmed functions. Respect the stablished function names and add the declaration of the functions (their types signature) in addition to their definitions. **Additionally**, add the **main** function to test all the cases of the programmed functions (see how is done in the solutions of the class exercises published in Canvas).

PROBLEMS: Basic and recursive programming WITHOUT USING LISTS (20 points)

1. Program the **mode** function which obtains the statistical mode of a group of 5 arguments. The mode is the value with the highest frequency in a data distribution. In case of a tie (multimodal) it returns the first value in the mode.
Test cases:
```
> mode 8 1 5 4 3        => 8
> mode 4 2 3 2 1        => 2
> mode 2 3 3 2 3        => 3
```

2. Program the recursive function **table** which receives a positive integer, so that from this it "displays" its multiplication table.
Test case:
```
> table 4  =>    4 x 1 = 4
                 4 x 2 = 8
                 4 x 3 = 12
                     …
                 4 x 10 = 40
```

PROBLEMS: Recursive Programming with Lists (30 points)

3. Program the RECURSIVE function **smallers** which receives 2 lists of integers and returns a list of sublists containing the integers of the second list that are less than the elements of the first; a sublist for each element of the first list.
Test cases:
```
> smallers [1,2,3] [4,5,1]        => [[],[1],[1]]
> smallers [2,4,0] [1,2,3,4,5]    => [[1],[1,2,3],[]]
```

4. Implement the RECURSIVE function **jumps** which receives 4 non-negative integers (n, i, s1, s2) and returns a list with n elements, starting at i and where the following are obtained by interspersedly jumping s1 steps and s2 steps.
Test cases:
```
> jumps 0 1 1 2         => []
> jumps 4 1 1 2         => [1,2,4,5]
> jumps 7 5 2 5         => [5,7,12,14,19,21,26]
```

5. Implement the RECURSIVE function **shift** which receives a non-negative integer and a list of sublists and returns a list with the same structure as the original (same number of sublists and sublists of the same size), but where its elements are shifted N positions

to the right. Scrolling causes elements of one sublist to be passed to subsequent sublists and must work in a circular fashion, so that elements of the last sublist would be passed to the first (if not empty).
Test cases:

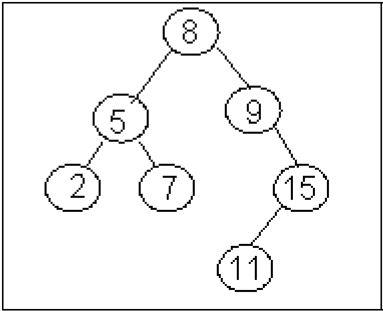
```
> shift 1 [[1,2,3],[4,5,6]]       => [[1,2,3],[4,5,6]]
> shift 1 [[1,2,3],[4,5,6]]       => [[6,1,2],[3,4,5]]
> shift 3 [[],[1],[2,3],[4,5,6]] => [[],[4],[5,6],[1,2,3]]
```

PROBLEMS: Data Structures in Haskell (20 points)

A Binary Tree (BT) can be represented in Haskell, by means of the following statement:

```
data BT t = N (BT t) t (BT t) | E derived (Show)
```

For example, if you have the following BT:



```
bt = N (N (N E 2 E)
          5
          (N E 7 E))
       8
       (N E
          9
          (N (N E 11 E)
             15
             E))
```

6. Program the **list_in_order** function which given a binary tree and a traversal type description, returns a list with the contents of the tree in the requested order. The possible orders will be prefix, infix and postfix.
Test cases:

```
> list_in_order bt "prefix"     => [8,5,2,7,9,15,11]
> list_in_order bt "infix"      => [2,5,7,8,9,11,15]
> list_in_order bt "postfix"    => [2,7,5,11,15,9,8]
```

7. A binary search tree (BST) is a binary tree where the root of the tree and of each subtree is greater than all the values in its left subtree and smaller or equal than all the values in its right subtree. Program the **add_bst** function which given a BST and a value, returns a new BST with the value inserted appropriately.
Test cases:

```
> add_bst E 5                  => N E 5 E
> add_bst (N E 5 E) 8          => N E 5 (N E 8 E)
> add_bst (N E 5 (N E 8 E)) 2  => N (N E 2 E) 5 (N E 8 E)
```

PROBLEMS: Higher Order Functions and other facilities (30 points)

8. Program the RECURSIVE function **g_disjoint** which using "guards" implement a predicate that checks if two plain lists passed as their arguments do not have elements in common.
Test cases:

```
> g_disjoint [1,2,3] [4,5,1]                  => False
> g_disjoint ['a','b','c'] ['d','e','f']      => True
```

9. Program the NON-RECURSIVE function **c_evens** which using "list comprehension" obtains an ordered list with the square of the even numbers from 2 to n.

Test cases:

```
> c_evens 1       => []
> c_evens 10      => [4,16,36,64,100]
```

10. Program the NON-RECURSIVE function **f_count** which using the higher-order functions seen in class, counts the number of odd numbers of the sublists of a list of sublists.

Test cases:

```
> f_count [[1,3,5],[1,2,3]]                  => [3,2]
> f_count [[],[1],[1,2,3],[5,4,3,2,1]]       => [0,1,2,3]
```