Exercise 3

This exercise session consists of Scala programming exercises.

Higher Order Functions Basics

Implement the methods described in the file: higherOrderFunc.scala.

Exercises From Last Week OPTIONAL

Revisit the exercises from last week and think about how you could use higher-order functions to solve the problems.

Hint: map, flatmap, pack, span and foldLeft are relevant.

Binary Search Trees (Dictionaries)

Background Information

A binary tree is either empty or it is composed of a root element and two successors, which are binary trees themselves.

A basic initial implementation of a binary tree is available in this project. Go to the binarytree.scala file and look at the following description of the implementation:

An End is equivalent to an empty tree. A Branch has a value, and two descendant trees. The toString functions are relatively arbitrary, but they yield a more compact output than Scala's default. Putting a plus in front of the T makes the class covariant; it will be able to hold subtypes of whatever type it's created for. (This is important so that End can be a singleton object; as a singleton, it must have a specific type, so we give it type Nothing, which is a subtype of every other type.)

A tree with only a root node would be Node('a') and an empty tree would be End. An example tree can be made with:

```scala
val tree = Node('a',
    Node('b', Node('d'), Node('e')),
    Node('c', End, Node('f', Node('g'), End)))
```

The result of calling tree.toString should be:

```scala
val expected = "T(a T(b T(d . .) T(e . .)) T(c . T(f T(g . .) .)))"
```

Implementation: addNode

Write a function to add an element to a binary search tree.

```
scala> End.addNode(2)
res0: Node[Int] = T(2 . .)
```

```
scala> res0.addNode(3)
res1: Node[Int] = T(2 . T(3 . .))
```

```
scala> res1.addNode(0)
res2: Node[Int] = T(2 T(0 . .) T(3 . .))
```

Hint: The abstract definition of addNode in Tree should be:

```
def addNode(x:Int,tree:BinaryTree):BinaryTree
```

Implementation: isBinarySearchTree

Write a function to check if the given BinaryTree is a binary search tree. The function should have the following definition:

```
def isBinarySearchTree(bt:BinaryTree):Boolean
```

The exercise is adapted from P57 at http://aperiodic.net/phil/scala/s-99/.

Group the elements of a set into disjoint subsets

Create a new file and implement P27 from http://aperiodic.net/phil/scala/s-99/ as described in the following.

Hint: You need to solve P26 before doing P27.

a)

In how many ways can a group of 9 people work in 3 disjoint subgroups of 2, 3 and 4 persons? Write a function that generates all the possibilities. Example:

```
group3(List("Aldo", "Beat",
        "Carla", "David",
        "Evi", "Flip",
        "Gary", "Hugo",
        "Ida"))
res0: List[List[List[String]]] =
        List(List(
        List(Aldo, Beat),
```

```
        List(Carla, David),
        List(Evi, Flip, Gary, Hugo, Ida)), ...
```

b)

Generalize the above predicate in a way that we can specify a list of group sizes and the predicate will return a list of groups.

Example:

```
group(
    List(2, 2, 5),
    List("Aldo", "Beat",
        "Carla", "David",
        "Evi", "Flip",
        "Gary", "Hugo",
        "Ida"))
res0:
    List[List[List[String]]] =
        List(List(
        List(Aldo, Beat),
        List(Carla, David),
        List(Evi, Flip, Gary, Hugo, Ida)), ...
```

Note that we do not want permutations of the group members; i.e. ((Aldo, Beat), ...) is the same solution as ((Beat, Aldo), ...). However, we make a difference between ((Aldo, Beat), (Carla, David), ...) and ((Carla, David), (Aldo, Beat), ...).

You may find more about this combinatorial problem in a good book on discrete mathematics under the term "multinomial coefficients".

Gray Code

Create a new file and implement P49 from http://aperiodic.net/phil/scala/s-99/ as described in the following.

An n-bit Gray code is a sequence of n-bit strings constructed according to certain rules. For example, n = 1: C(1) = ("0", "1"). n = 2: C(2) = ("00", "01", "11", "10"). n = 3: C(3) = ("000", "001", "011", "010", "110", "111", "101", "100"). Find out the construction rules and write a function to generate Gray codes.

For instance:

```
gray(3)
res0 List[String] = List(000, 001, 011, 010, 110, 111, 101, 100)
```

See if you can use memoization to make the function more efficient.