Advanced Programming II

Introduction to Scala and Functional programming

Exercises for lessons 1 and 2:

1. Implement a program that determines if a given year is a leap year.
2. Write a program that simulates a simple ATM (single account). It should allow the user to check his/her balance, deposit money, and withdraw money while handling insufficient funds.
3. Implement a function that checks if a given string is a palindrome (ignoring case and spaces).
4. Create a program that prints the first `N` prime numbers.
5. Implement a program that calculates the greatest common divisor (GCD) and least common multiple (LCM) of two numbers.
6. Implement a function that finds the second-largest element in a list.
7. Write a program that removes all duplicates from an array **without** using built-in functions like `distinct`.
8. Implement a function that rotates a list `k` positions to the right (e.g., `[1, 2, 3, 4, 5]` rotated by `2` becomes `[4, 5, 1, 2, 3]`).
9. Write a function that merges two sorted arrays into a single sorted array without using `sort()`.
10. Implement a program that checks if two arrays are permutations of each other.
11. Implement a function that compresses a string using run-length encoding (e.g., `"aaabbc"` → `"a3b2c1"`).
12. Implement a program that finds the most frequently occurring character in a string.
13. Implement a `BankAccount` class that supports deposit, withdrawal, and balance check operations, ensuring that withdrawal cannot exceed balance.
14. Create a `Triangle` class with methods to determine if it is equilateral, isosceles, or scalene.
15. Implement a `Student` class with attributes `name`, `age`, and `grades`. Add a method to calculate the student's average grade.
16. Create a `Fraction` class that supports addition, subtraction, multiplication, and division.
17. Implement a `Book` class with attributes `title`, `author`, and `yearPublished`. Add a method to check if a book is older than another.
18. Write a function that reads an integer from the user and retries until a valid integer is entered.
19. Implement a function that reads a file and prints its contents.
20. Implement a recursive function to compute the nth Fibonacci number. Convert it to tail-recursive.
21. Implement a recursive function that returns the sum of digits of a given integer. Convert it to tail-recursive.

22. Implement a recursive function to generate all subsets of a given set. Convert it to tail-recursive.
23. Write a function that finds the longest increasing subsequence in a list of integers.
24. Implement a program that reads a text file and counts the number of occurrences of each word.
25. Implement a program that finds the longest word in a file.
26. Write a program that copies the contents of one file into another.
27. Define a function last that returns an Option object with the last element in the list received as an argument, if such an element exists, or None otherwise. For example,

```
last(List("a","b","c","d")) == Some("d")
last(Nil) == None
```

28. Define a function nth that returns an Option object with the element of the list at position i (the list and the position i are arguments of nth), if such an element exists, or None otherwise. For example,

```
nth(List("a","b","c","d","e"), 2) == Some("c")
nth(List("a"), 2 ) == None
nth(Nil, 0) == None
```

29. Define a tail-recursive function `addSquares(List[Int])` to calculate the sum of the squares of the elements of the integer list received as argument.
30. Define a tail-recursive generic function (a list of items of any type) that receives a list and returns the same list but with its items in reverse order.
31. Define a tail-recursive function that receives as arguments a function *f* from interger to integer (Int => Int) and two integers, x and y, and calculates $\sum_{x \le i \le y} f(i)$.
32. Define a tail-recursive function that receives an integer n ≥ 0 and returns a list with the natural numbers from 0 to n.
33. Define a tail-recursive generic function `unzip` that receives a list of 2-item tuples and returns a tuple with two lists: one with the first components and the other with the second ones. For example,

```
unzip(List((10,'a'),(20,'b'),(10, 'c'))
      == (List(10,20,30), List('a','b','c'))
```

34. Define a tail-recursive generic function `zip` that receives two lists (they may have different lengths) and returns a list of 2-item tuples where their first components are taken from the first list and the second components are taken from the second list. For example,

```
zip(List(10,20,30), List('a','b','c'))
      == List((10,'a'),(20,'b'),(10,'c'))
zip(List(10,20,30), List('a','b'))
```

```
      == List((10,'a'),(20,'b'))
```

35. Develop a function `flatten` que transforme una estructura de listas anidadas en una lista aplanada. Por ejemplo,

```
flatten(List("a", List("b","c"), List("d", "e")))
      == List("a", "b", "c", "d", "e")
flatten(List("a", List("b","c",List("d", "e"))))
      == List("a", "b", "c", "d", "e")
```

36. Develop a function `compress` that removes consecutive repeated items from a list. For example,

```
compress(List("a","a","a","b","c","c","d",”a”,"e","e","e"))
      == List("a","b","c","d",“a”,"e")
```

37. Develop a function `pack` that groups consecutive repeated items into sublists. For example,

```
pack(List("a","a","a","b","c","c","d","e","e","e"))
      == List(List("a","a","a"), List("b"), List("c","c"),
            List("d"), List("e","e","e"))
```

38. Develop a function `replicate` that builds a list in which the items of the received list are replicated as many times as indicated in a second argument. For example,

```
replicate(List("a", "b", "c", "d"), 3)
      == List("a","a","a","b","b","b","c","c","c","d","d","d")
```

39. Develop a function `range` that returns a list with the integer values in between its two arguments (either in ascending or descending order, as requested). For example,

```
range(4, 9) == List(4,5,6,7,8,9)
range(9, 4) == List(9,8,7,6,5,4)
range(5, 5) == List()
```