Exercise 8 - Functional Collections

Objective

The objective of these exercises is to look at how to use collections with functional programming language constructs. We look at filter and flatMap to manipulate lists. We also look at how we can implement our own List class in scala allowing us to apply many of the techniques we have learned to this point

References:

Chapter 8 of the slides: Pages 174-197

Overview

1. Define a toInt function that tries to convert a String to an Int. It should return an Option, returning “Some[Int]” with the value if it can be converted, and “None” if it can't. Test your function with suitable inputs to make sure it works

Hint: A try / catch could be useful here!

1. Declare a list of strings, some of which are numbers (e.g. "12")
2. Use map to apply toInt to your list of strings, and use foreach to print the result. Do the same using flatMap, and make sure you appreciate the difference.
3. Given a map of cities to countries (e.g. "London" -> "UK"), can you write a statement to produce a map that swaps the keys and values (e.g. "UK" -> "London")?

Hint: This is not as tricky as you think it may be, think about how for/yield loops work

1. Create a List of numbers. Use foldLeft to calculate the sum of the list.
2. Re-implement the Luhn algorithm seen in exercise 5 to use Lists and any functional methods that may be useful (for example using fold to sum the numbers)
3. Create a class Employee. An employee has a name, and if they are a manager, a list of people who reports to them (remember the \* symbol allows you to enter zero or more values as parameters!)

class Employee(val name: String, val reports: Employee\*)

Next, create a few Employees, making sure that at least one is a manager and add them to a list

1. Use filter to produce a list of only those Employees who are managers (hint: you can use isEmpty to see whether there are any reports for an Employee)
2. Use a map (or flatmap) on this the list created in part 8 to produce tuples of the manager’s name and employee names, for example, for if Jane was a Manager for Bill and Dave the output would be ("Jane", "Bill") and ("Jane", "Dave")

Implementing a Simple List

The rest of this exercise is about implementing a simple linked list. Create a new worksheet for this exercise. If you include it in the same worksheet or class you are using the standard scala lists then the compiler will not be able to compile the code correctly.

A list consists of a series of linked items, called “cons cells”, terminated by a “null entry” called Nil. We can model this as follows:

sealed trait List[+A]

case object Nil extends List[Nothing]

case class Cons[+A](item: A, tail: List[A]) extends List[A]

What does the +A denote, and is it necessary?

Notes:

A sealed trait or class means that it cannot be extended by any code outside this file

Using case classes and objects means we can use these in pattern matching later. (It also means that you get a nice toString implementation, but that isn’t the point)

Since there is only one empty list, we make it an object

A Cons cell is created from a head and a tail

Try creating some Lists using nested constructor calls to Cons.

Now create a companion object to List, and implement an apply() method so that you can create lists like this:

val lst = List(1, 2, 3, 4)

Apply() will need to be a generic function that takes a variable number of arguments, so it will look like this:

def apply[A](items: A\*)

If ‘items’ is empty, then the result is an empty list (Nil). If it isn’t, return a Cons constructed like this:

Cons(items.head, apply(items.tail: \_\*))

(The \_\* is needed when passing a collection through as a variable argument list)

Try creating some lists with your factory.