ASSIGNMENT 6 – ANAGRAMS MINI-PROJECT

Advanced programming paradigms

1.1 Objectives

In this mini-project[1], you will work with anagrams and apply all that you have learned so far on a slightly larger project. Before we begin, let's start with some useful definitions:

1. An *anagram* is a rearrangement of the letters of a word to create another meaningful word. For instance *pea* is an anagram of *ape* and *listen* is an anagram of *silent*. This is the topic of the first part of this mini-project.
2. It is also possible to consider *sentence anagrams*, where we rearrange all the letters of a sentence to make a new sentence. For instance, those two sentence are anagrams of each other:

   ```
   eleven plus two ↔ twelve plus one
   ```

   Another example is for instance given by the sentences:

   ```
   debit card ↔ bad credit
   ```

   For sentence anagrams, the number of words in the new sentence might be different than from the original sentence. Sentence anagrams will be treated in the second part of this mini-project.
3. To produce such anagrams, we simply ignore character casing as well as punctuation characters.

1.2 Installing the project and running test suites

For this mini-project you have to import an Eclipse project that has been prepared for you on *Moodle*. To import the project, navigate to *File* → *Import* → *Existing project into workspace*. Select then *Select archive file* and point to the archive file you have downloaded from *Moodle*. The project should now be opened.

In addition to some source files, this template also includes Scala unit tests to check your code. You can easily execute the test suites directly inside Eclipse by navigating to source file of the test suite in `src/test/AnagramsTests.scala`, right-click on it and select *Run As* → *Scala JUnit Test*. The tests will then be executed and a window will display the result of each individual test.

1.3 Structure of the code and types definition

In the project you received, two main types aliases have been defined:

```
1 type Word = String
2 type Occurrences = List[(Char, Int)]
```

The `type` keyword introduces a *type alias* which means that `Word` is an alias for the `String` type. Similarly, the `Occurrences` type is defined as a list containing a pair made of a character and an integer value.

[1] This exercise is adapted from Martin Odersky's online course *Functional programming in Scala* which is available online on the *Coursera* web site http://https://www.coursera.org/course/progfun.

This said, the internal data representation you will be using is as follows:

- *Word* – words in anagrams contains only lower case letters no punctuation at all;
- *Occurrences* – This type will be used throughout the mini-project to store how often a letter appears in a word. Such a type will be very helpful because two anagrams have the same occurrences of letter per word. If you consider for instance the word `tea`, every letter (t,e,a) appears once. The occurrences for this word is then `List(('t', 1), ('e', 1), ('a', 1))`. If you now consider the word `eat`, it has the same occurrences, however appearing in a different order. You will use this property to look for anagrams.

Part 1 - Word anagrams

Question 1 – *Generating occurrences of word letters*

You first have to devise a function which gets a word as a parameter and builds its occurrence list as explained earlier. For this, you have to complete the implementation of the `wordOccurrences` function. For instance, calling

```
wordOccurences("abcd") returns List(('a', 1), ('b', 1), ('c', 1), ('d', 1))
wordOccurences("aabcddd") returns List(('a', 2), ('b', 1), ('c', 1), ('d', 3))
wordOccurrences("Robert") returns List(('b', 1), ('e', 1), ('o', 1), ('r', 2), ('t', 1))
```

Note that – as depicted above – the various letters extracted in the result list must be **sorted alphabetically** (this is important). In addition, please note that if a character is not present in a word, it should not be present in the result. Test your code with the test suite.

Hint: to simplify your code, have a look at the `groupBy` method provided by the `List` class. This method takes a function mapping an element of a collection to a key of some other type. It produces a `Map` of keys and collections of elements which are mapped to the same key. Applying this method groups the elements of the collection with respect to a given function (hence its name). For instance,

```
1 List("Programming", "is", "fun", "and", "great").groupBy(s => s.length)
```

returns the following map:

```
1 Map(2 -> List(is), 11 -> List(Programming), 5 -> List(great), 3 -> List(fun, and))
```

The `Map` collection

A map collection[2] provides a very efficient lookup method for elements. Each element of the map, which is written as `Map((a -> b), (c -> d), .....)`, maps a key to a value. The value on the left-hand side of the arrow is the *key* and the *value* is written on the right-hand side of the arrow. In addition to the `->` notation, each element of the map can also be accessed as a pair.

Here is an example of the possible operations on maps:

```
1 val colors = Map(
2   "red" -> "#FF0000",
3   "azure" -> "#F0FFFF",
4   "peru" -> "#CD853F")
```

This codes defines `colors` as a `Map[String, String]`. To access an element of the map, you can simply use the `apply` function, which can also be shortened by using parentheses. Thus, writing `colors("red")`

[2]Similar to an hash-map in Java

returns the string `"#FF0000"`. If you try to access a value not contained in the map, an exception is raised. For more information, you can have a look online[3] or on the Scala API[4].

Question 2 – *Valid words by occurrences*

To check what are the possible anagrams for a word, you will be provided an English dictionary file that contains all the valid words considered for this assignment. This file is automatically loaded in the code with the line

```
1 val dictionary : List[Word] = loadDictionary()
```

Starting from this and the fact that that you have now a function that returns the occurrences list of each character in a word, you have to devise a function that generates the occurrences list for the whole dictionary. This will be used to extract all the valid anagrams of words.

To do so, you have to complete the definition of the `val` called `dictionaryByOccurrences`, using the following steps:

(a) The word `eat` has the following occurrence list:

```
List(('a', 1), ('e', 1), ('t', 1))
```

Incidentally, so do the word `ate` and `tea`. Because we have a full dictionary, we can now generate the occurrence list for each word and group the words that share a similar list of occurrences together. In this example, this means that the `dictionaryByOccurrences` Map should contain (among others) an entry which looks similar to :

```
List(('a', 1), ('e', 1), ('t', 1)) -> Seq("ate", "eat", "tea")
```

(b) Test your function by running the test suite.

(c) Why did we defined `dictionaryByOccurrences` as a `val` and not as a `def`?

.....

.....

.....

.....

(d) Can you find why we declared the `val` as `lazy`?

.....

.....

.....

(e) Can you see the effect of the `lazy` declaration in the test runner (when you execute `AnagramsTests` as a *Scala JUnit Test*)?

.....

.....

.....

[3] For instance here http://www.tutorialspoint.com/scala/scala_maps.htm

[4] http://www.scala-lang.org/api/current/#scala.collection.immutable.Map

Question 3 – *Generating word anagrams*

You now have to complete the definition of the `wordAnagrams` method whose objective is to return all the anagrams of a given word. Test you function by running the tests again.

Hint: Implementing this function is relatively straightforward if you think carefully of what is the content of the modified dictionary you created in the previous exercise contains (it is a map and looking into a map is simple).

Part 2 - Sentence anagrams

You will now generate sentence anagrams, which is a rearrangement of all the characters in the sentence such that a new sentence is formed.

As before, the new sentence consists of meaningful words, the number of which may or may not correspond to the number of words in the original sentence. For example, the three following sentences are sentence anagrams:

```
i love you
↔ you olive
↔ you i love
```

As you can see, permutations of words are considered here as different sentences.

Finding an anagram of a sentence is slightly more difficult. You will first transform the sentence into its occurrence list, then try to extract any subset of characters from it to see if you can form any meaningful words. From the remaining characters, you will solve the problem recursively and then combine all the meaningful words you have found with the recursive solution.

Let's apply this idea to our example, the sentence *you olive*. Lets represent this sentence as an occurrence list of characters *eiloouvy*. We start by subtracting some subset of the characters, say *i*. We are left with the characters *eloouvy*.

Looking into the dictionary we see that *i* corresponds to word *I* in the English language, so we found one meaningful word. We now solve the problem recursively for the rest of the characters *eloouvy* and obtain a list of solutions `List(List(love, you), List(you, love))`. We can combine *I* with that list to obtain sentences *I love you* and *I you love*, which are both valid anagrams.

Question 4 – *Generating all subsets of a set*

To compute all the anagrams of a sentence, we need an helper method that, given an occurrence list, produces all the subsets of that list using *for comprehensions*. For example, given the occurrence list:

```
List(('a', 2), ('b', 2))
```

the function should return the list of all the possible subsets:

```
List(
  List(),
  List(('a', 1)),
  List(('a', 2)),
  List(('b', 1)),
  List(('a', 1), ('b', 1)),
  List(('a', 2), ('b', 1)),
  List(('b', 2)),
  List(('a', 1), ('b', 2)),
  List(('a', 2), ('b', 2))
)
```

The order in which you generate the subsets is not important. Note that there is only one subset for the empty occurrence list and it is the empty occurrence list itself. The method you have to complete has the following prototype:

```
1 def combinations(occurrences: Occurrences): List[Occurrences]
```

Hints: To help you design the content of this function, you might use the following intermediary steps (it is not compulsory to do this, but it can help if you don't know how to start):

(a) Consider first the occurrence list `List(('a', 2))`. If we don't take into account the empty list that must be present, your function should yield:

```
List(
  List(('a',1)),
  List(('a',2))
)
```

Start by creating a *for comprehension* for achieving this.

(b) Add a level of complexity in your function by considering the occurrence list `List(('a',2), ('b',4))`. Modify your *for comprehension* to generate

```
List(
  List(('a',1)),
  List(('a',2)),
  List(('b',1)),
  List(('b',2)),
  List(('b',3)),
  List(('b',4))
)
```

(c) You now generate some of the required couples but the combinations such as `('a', 1), ('b', 1)` and `('a', 2), ('b', 1)` are still missing. Complete your function to make that happen.

(d) Add the empty list to your result.

Question 5 – *Remove an occurrence from an occurrence list*

We now implement another helper method called `subtract` which, given two occurrence lists `x` and `y`, subtracts the frequencies of the occurrence list `y` from the frequencies of the occurrence list `x`:

```
1 def subtract(x: Occurrences, y: Occurrences): Occurrences
```

For example, given two occurrence lists for words lard and r:

```
1 val x = List(('a', 1), ('d', 1), ('l', 1), ('r', 1))
2 val y = List(('r', 1))
```

the `subtract(x, y)` is `List(('a', 1), ('d', 1), ('l', 1))`.

The precondition for the subtract method is that the occurrence list `y` is a subset of the occurrence list `x`. If the list `y` has some character, then the frequency of that character in `x` must be greater or equal than the frequency of that character in `y`. When implementing `subtract`, you can safely assume that `y` is a subset of `x`.

Hint: you can use the operations `foldLeft`, `-`, `apply` and `updated` operations on `Map`.

Question 6 – *Generating sentence anagrams*

Now you can finally implement the `sentenceAnagrams` method for sequences. Its prototype is as follows:

```
1 def sentenceAnagrams(sentence: Sentence): List[Sentence]
```

Note that the anagram of the empty sentence is the empty sentence itself.

Hint: First of all, think about the recursive structure of the problem: what is the base case, and how should the result of a recursive invocation be integrated in each iteration? Also, using *for-comprehensions* helps in finding an elegant implementation for this method.

Test your method on short sentences, having no more than 10 characters. The combinations space gets huge very quickly as your sentence gets longer, so the program may run for a very long time. However for sentences such as *Linux rulez*, *I love you* or *Mickey Mouse* the program should end fairly quickly as there are not many other ways to say these things.