Exercises — Scala (Week Four)

Functional functions

Spring term 2017

Overiew

This exercise assignment introduces several functions that are commonly used in functional programming. Again you are provided with a starter project where you will complete several Scala functions that exercise your understanding of functional programming.

Lists are the most common data structure in functional programming, and functions like `map` and `foldLeft` are the building blocks of programs that work with lists. Instead of recursion or looping, we define what transformations wed like to apply to the contents of the list and let our building blocks do the work.

You will be relying heavily on recursion and pattern matching, so please be familiar with these concepts, either from the notes or from other recommended texts. I particularly like this blog post[1] which can act as a quick pattern matching reference.

All of these functions will manipulate the Scala `List` type and the blog post referenced above goes into that a bit, but will we also provide some examples.

The functions you will write are based on concepts introduced in the notes and slides; further details can be found in the recommended supplementary texts. You will also have more practice using the `sbt` build system and running tests.

Examples

```
val list = List(1,2,3)
list match {
// Matching single element:
    case l :: _ => println(s"The head of the list is $l")
// Matching the rest of the list:
    case _ :: ls => println(s"The tail of the list is $ls")
// Match both parts:
    case l :: ls => println("Use l for something, then recurse with ls!")
// Checking if the list is empty:
    case Nil => println("Nil represents the empty list")
// Matching an element at the end of the list:
    case l :: Nil => println(s"$l is the last element of the list.")
// Matching a specific list:
    case List(1,2,3) => println("The lists are identical")
// Matching an unknown element:
    case List(1,2,n) => println(s"The last element is $n")
```

[1] `https://kerflyn.wordpress.com/2011/02/14/playing-with-scalas-pattern-matching/`

```
// Guards let you match on a condition (l > 2, in this case):
    case l :: ls if (l > 2) => println("The first element is greater than 0")
// A default case is often useful:
    case _ => println("The underscore matches everything.")
}
```

Scala will complain at compile time if you have too few cases or if you have too many. Too few cases means the match cant be completed successfully, and Scala will thrown an appropriate error.

```
list match {
    case l :: Nil => println("This only matches a list with 1 element.")
    // There's no other cases -- uh oh!
}
```

Learning Objectives

- To understand the methodology behind common functions like `map` and `fold`.
- To understand how these functions may be used in your programs.
- To understand how to combine these functions to elegantly program complex tasks.
- To practice using these functions in simple and complex ways.
- To reinforce your knowledge of the sbt tool.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate this to a member of the teaching team immediately.

Test files

In the `src/test/scala` directory, we provide several `ScalaTest` test suites that will help you keep on track while completing these exercises. We recommend you run the tests often and use them to help create a checklist of things to do next.

We recommend that you think about possible cases and add new test cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods handle edge cases such as integer arguments that may be positive, negative, or zero? Many methods only accept arguments that are in a particular range.
- Does your code handle unusual cases, such as empty or maximally sized data structures?

To build good test cases, think about ways to exercise the functions and methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case.

Project structure

Just a refresher on the project structure and what it should normally contain as root items:

src/main/scala : This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

src/instructor/scala : This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. If you make changes or add/remove anything it can lead to problems in your testing.

src/test/scala : The test folder where all of the public unit tests are available.

build.sbt : This is the build definition file. It provides build information to build your project. Do not remove.

.gitignore : This is a special file used by git source control to ignore certain files. You should not touch this file and please do not remove. This file may not be present if the assignment does not call for git usage.

Testing

As mentioned previously, you are provided a set of unit tests that will test various aspects of your implementation. You should get in the habit of running the tests frequently to see how you are doing and to understand where you might be going wrong.

Editors and IDEs

You are welcome to use any editor or IDE you choose. We recommend using either IntelliJ or a basic text editor such as Atom, SublimeText, Notepad++, emacs, or vim. If you use a text editor you should use sbt in a separate terminal window to compile, run, and test your code.

Implementing the functional functions

This exercise is split into four question:

1. list basics,
2. folding,
3. mapping, and
4. combining functions.

Each function has a comment in the `functions.scala` file, which explains the desired behaviour of the function and a `???` definition of the function signature. Further details and hints may be found there. You may have to write the signature for a function if we haven't specified it.

Every function must be written either recursively with *pattern matching* or by using functions you have already implemented.

You **must not** use explicit iteration, including *for loops* and *for comprehensions*, in any of these functions.

Additionally, you will rarely need to define variables or use `if` statements. If you find yourself doing so, try to see if theres another way to implement the function. If in doubt, ask a member of the teaching team.

1. List Basics

   These functions are simple enough that you wont need to call any other functions to implement them. All you need is pattern matching and recursion!

   (a) `tail`:
   takes a list and returns all but the first element. This isn't a recursive function, but you still need to use pattern matching to catch when the input is `Nil`. Calling `tail` on an empty list throws an `IllegalArgumentException`.
   Example: `tail(List(1,2,3)) = List(2,3)`

   (b) `setHead`:
   takes a list and replaces the first element. If the list is empty, it simply inserts the element into the empty list.
   Question: Can you use `tail` for this?
   Example: `setHead(List(1,2,3), 0) = List(0,2,3)`

   (c) `drop`:
   removes the first `N` elements from the given list. If the number of elements to remove exceeds the number of elements in the list, `drop` will return an empty list.
   Hint: You have multiple base cases here. How can you match the number of elements to remove and the list at the same time?
   Example: `drop(List(1,2,3,4), 3) = List(4)`.

   (d) `init`:
   is the opposite of `tail`: It removes the last element of the list. Remember that youre not modifying the input list, but building a new list. Calling `init` on an empty list throws an `IllegalArgumentException`.
   Example: `init(List(1,2,3)) = List(1,2)`.

2. Folding Lists

   To *fold* a list means to reduce it down to a single value. Youve done this before; for example, `mkString` takes a list (or array, vector, etc.) and turns it into a single String. For this question we are generalising that into a generic `foldLeft` function and seeing just what we can do with it. After you implement `foldLeft` yourself, you must use it to implement the remaining functions for this question.

   (a) `foldLeft`:
   One of the real *bread-and-butter* functions. takes a list, an initial value, and a function, then computes a value by iteratively applying the function to the elements of the list and carrying the result along. `foldLeft` and `foldRight` are very similar functions; they just perform the operations in a different order. The difference is subtle; the key takeaway is that `foldLeft` is tail recursive. You

should implement your own version of `foldLeft` using pattern matching and recursion. The function that `foldLeft` takes has two arguments: the *accumulative result* and the *current element* of the list.

(b) `sum`:
takes a list of doubles and computes the sum. Thats it! Very straightforward.
Example: `sum(List(1.0, 2.0, 3.0)) = 6.0`

(c) `product`:
takes a list of doubles and computes the product. This should be very similar to `sum`.
Example: `product(List(1.0, 2.0, 3.0)) = 6.0`

(d) `length`:
computes the length of the input. This is a case where the result of `foldLeft` isnt directly related to the input data! `foldLeft` doesnt have to use the data, it just iterates over the list.
Example: `length(List(1,2,3,4,5,6)) = 6`.

(e) `reverse`:
reverses a list. Yes, you can do this with `foldLeft`, too!
Example: `reverse(List(1,2,3)) = List(3,2,1)`

(f) `flatten`:
turns a nested list into a single list. This is more straightforward than it sounds.
Example: `flatten(List(List(1,2,3), List(4,5,6))) = List(1,2,3,4,5,6)`

At this point, you may be wondering about the difference between `foldLeft` and `foldRight`. Try this out in the Scala REPL: (using Scalas own `fold` functions, not ours.)

```
val h = List("H","e","l","l","o")
h.foldLeft("")(_ + _)
h.foldRight("")(_ + _)
```

Those should be the same. Now try these:

```
h.foldLeft("!")(_ + _)
h.foldRight("!")(_ + _)
```

3. Map and Filter
`map` and `filter` are two more building blocks. `map` is incredibly useful: it applies a function to each element of a list, producing a new list of the same length containing the results. `filter` lets you remove unwanted elements from a list using a function.

Question: Why are `map` (the function) and `Map[A,B]` (the data structure) named the same thing? This isnt a graded question.

Reminder: *recursion* and *pattern matching* are your friends!

(a) `map`:
takes a list and a function, then builds a new list by applying the function to each element of the list. This is the third or fourth time youve read that description so far in this exercise. Still unsure what it does? Ask!
Unlike `foldLeft`, `map`s function takes only a single argument.

(b) `filter`:
applies a predicate (a function that returns a `Boolean` value) to the elements of the list, and discards the ones for which the predicate returns `False`.
Example: `filter(List(1,2,3,4,5))( _ % 2 == 0) = List(2,4)`

(c) `flatMap`:
is a cross between `map` and `flatten`. Instead of a single value, `flatMap`s function returns a list of values. `flatMap` collects all of these values into a single list.
Hint: Dont overthink it.

4. Combining Functional Functions — bringing it all together
Youll notice there werent any neat small functions for the *Map and Filter* section like there were for the *Folding Lists* section. Instead, were going to jump right into solving some interesting and complex problems using the functions youve written so far.

Note: These functions are all independent. You must use the functions you implemented in the previous parts, but you wont be using these functions to implement each other unless specifically noted otherwise.

(a) `maxAverage`:
takes as input a list of tuples of doubles. It then finds the average of the largest values in each tuple.
Hint: You need to transform the input list into a list of the max values, then find the average of that list. What function(s) can you use for that?

(b) `variance`:
calculates the statistical variance of a population, or the squared deviation from the mean. Statistics!
In more detail: *variance* is the average of the squared difference of each element from the average of the whole list. So if the average of the whole list is M, the variance is

$$sum((element - M)^2)/length$$

(Sometimes, you will see the sum divided by $length - 1$. Dont worry about that – were treating the list as a whole population instead of a random sample.)

As you implement these functions you should run the tests frequently as described above. You should also take advantage of the REPL (read eval print) loop and test out your functions by cut and pasting them from your source code into the console to test. (This presumes you are not using an IDE; if you are then just *right click* the test file and select `run` from the resulting menu.)