Algebraic Data Types in Scala

Exercises marked **[–]** should be easy. Skip them if you already know Scala and functional programming. Exercises marked **[+]** are cognitively more demanding and show the more typical work done in the course later on. There is *extremely* many exercises this week, by design. The idea is to give students who have little functional programming experience a good material to catch up in the beginning. If you had studied functional programming before, you just need to recall it, and see how things are written in Scala—solve only some exercises. Skip those that you know how to solve. We will have much less exercises in the following weeks.

Do not use variables, side effects, exceptions or return statements. (I explicitly say in the exercise text, if you should use an exception this week. Exceptions are not referentially transparent, but we will eliminate the use of exceptions next week.)

If you solve all exercises, and your experience with functional programming is *average*, solving all exercises will take about 10 hours *plus* the time used for reading the textbook, making coffee, chit-chat and smoking. When skipping the simplest exercises, you can probably do in half of the time. If you have no experience, you are up for a tough and long ride . . . .

No Hand-in: There is no hand-in this week. Please rely on automatic tests and compiler errors to see whether you are doing fine. We give feedback in exercise sessions, on the forum and in teams. It is fine to post solutions of exercises for discussion on the forum this week, if you feel the need.

Exercise 1 [–]. What is the value of the following match expression?[1] Answer without running the code.

```
List (1,2,3,4,5) match {
  case Cons (x,Cons (2,Cons (4,_))) => x
  case Nil => 42
  case Cons (x,Cons (y,Cons (3,Cons (4,_)))) => x + y
  case Cons (h,t) => h + sum(t)
  case _ => 101
}
```

Remark. All exercises use our own implementation of lists, not the one in the Scala's standard library, so do not google for Scala Lib's API documentation to find the available functions. The implementation of lists is in the file `src/main/scala/ExercisesInterface.scala`, in the very top. This is the only API that is available (plus the functions from earlier exercises can be used in solving later exercises).

Exercise 2 [–]. Implement the function `tail` for removing the first element of a list. The function should run in constant time. Throw an exception if given an empty list (but a different exception than `NotImplementedError`).[2]

```
def tail[A] (as: List[A]) : List[A]
```

[1] Exercise 3.1 [Chiusano, Bjarnason 2014]
[2] Exercise 3.2 [Chiusano, Bjarnason 2014]

Exercise 3 [–]. Generalize `tail` to the function `drop`, which removes the first `n` elements from a list. Note that this function takes time proportional only to the number of elements being dropped—we do not need to make a copy of the entire list. Again, throw an exception if the list is too short.[3]

```
def drop[A] (l: List[A], n: Int): List[A]
```

Exercise 4 [–]. Implement `dropWhile`, which removes elements from the given list prefix as long as they match a predicate `f`. Do not use exceptions. If all list's elements match the predicate, then the natural value to return is the empty list.[4]

```
def dropWhile[A] (l: List[A], f: A =>Boolean): List[A]
```

Exercise 5 [–]. Implement a function, `init`, that returns a list consisting of all but the last element of the original list. So, given `List (1,2,3,4)`, `init` will return `List (1,2,3)`. Throw an exception if the list is empty.

```
def init[A] (l: List[A]): List[A]
```

Is this function constant time, like `tail`? Is it constant space? [5]

Exercise 6 [–]. Compute the length of a list using `foldRight`:[6]

```
def length[A] (as: List[A]): Int
```

Exercise 7 [+]. The function `foldRight` presented in the book is not tail-recursive and will result in a `StackOverflowError` for large lists. Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that *is* tail-recursive:

```
def foldLeft[A,B] (as: List[A], z: B)(f: (B, A) =>B) : B
```

For comparison consider that:

`foldLeft (List (1,2,3,4),0) (_ + _)` computes $(((0 + 1) + 2) + 3) + 4$ while
`foldRight (List (1,2,3,4),0) (_ + _)` computes $1 + (2 + (3 + (4 + 0)))$.

In this case the result is obviously the same, but not always so. The two functions also have different space usage.[7]

Exercise 8 [–]. Write `product` (computing a product of a list of integers) and a function to compute the `length` of a list using `foldLeft`.[8]

Exercise 9. Write a function that returns the reverse of a list (given `List (1,2,3)`, it returns `List (3,2,1)`). Use one of the fold functions (no recursion) [9]

Exercise 10. Write `foldRight` using `foldLeft` (**Hint:** use reverse). This version of `foldRight` is useful because it is tail-recursive, which means it works even for large lists without overflowing the stack. On the other hand, it is slower by a constant factor.

[3]Exercise 3.4 [Chiusano, Bjarnason 2014]
[4]Exercise 3.5 [Chiusano, Bjarnason 2014]
[5]Exercise 3.6 [Chiusano, Bjarnason 2014]
[6]Exercise 3.9 [Chiusano, Bjarnason 2014]
[7]Exercise 3.10 [Chiusano, Bjarnason 2014]
[8]Exercise 3.11 [Chiusano, Bjarnason 2014]
[9]Exercise 3.12 [Chiusano, Bjarnason 2014]

Exercise 11 [+]. Write `foldLeft` in terms of `foldRight`. Do not use `reverse` here (reverse is a special case of `foldLeft` so a solution based on `reverse` is cheating a bit).

Hint: To do this you will need to synthesize a function that computes the run of `foldLeft`, and then invoke this function. To implement `foldLeft[A,B]` you will be calling

```
foldRight[A,B=>B] (... , ...) (...)
```

that shall compute a new function, which then needs to be called. To our best knowledge this implementation of `foldLeft` has no practical use, but it is an interesting mind twister. It also demonstrates how to use anonymous functions to synthesize and delay computations. This technique is used for many things. We shall use it to implement lazy streams in several weeks.[10]

Note: *From now on we consider use of recursion a bad-smell if a piece of code could be written more or less directly using a fold function, or other higher order functions (map, forall, exists, etc.). Recursion should only be used if we are dealing with a non-standard structure of iteration. A fold should only be used if any of the other simpler HOFs cannot be used. This also means that in the course exam, if you use recursion, where a simpler solution with standard HOFs is possible, you will not receive top points.*

Exercise 12 [+]. Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists. Use `append`, which concatenates the two lists (described in the book).[11]

Exercise 13. Write a function filter that removes elements from a list unless they satisfy a given predicate `f`.[12]

```
def filter[A] (as: List[A])(f: A =>Boolean) : List[A]
```

Exercise 14 [+]. Write a function `flatMap` that works like map except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
def flatMap[A,B] (as: List[A]) (f: A =>List[B]): List[B]
```

For instance, `flatMap (List (1,2,3)) (i =>List (i,i))` should result in `List (1,1,2,2,3,3)`. `flatMap` will be key in the rest of the course (together with `map`), so this and the following exercise are the most important in this set.[13]

Exercise 15. Use `flatMap` to implement `filter`, a standard HOF in Scala's libraries. It was introduced in the lecture slides and in the book.[14]

Exercise 16 [–]. Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6,7)` become `List(5,7,9)`.Trailing elements of either list are dropped.[15]

[10]Exercise 3.13 [Chiusano, Bjarnason 2014]
[11]Exercise 3.15 [Chiusano, Bjarnason 2014]
[12]Exercise 3.19 [Chiusano, Bjarnason 2014]
[13]Exercise 3.20 [Chiusano, Bjarnason 2014]
[14]Exercise 3.21 [Chiusano, Bjarnason 2014]
[15]Exercise 3.22 [Chiusano, Bjarnason 2014]

Exercise 17 [–]. Generalize the function you just wrote so that it is not specific to integers or addition. It should work with arbitrary binary operations. Name the new function `zipWith`.[16]

Exercise 18 [+]. Implement a function `hasSubsequence` for checking whether a `List` contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)` , `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function that comes most naturally, but is not necessarily efficient (efficiency is often overrated). Note: Any two values `x` and `y` can be compared for equality in Scala using the expression `x ==y`. Here is the suggested type:

```
def hasSubsequence[A] (sup: List[A], sub: List[A]): Boolean
```

Recall that an empty sequence is a subsequence of any other sequence.[17]

Exercise 19 [+]. Recall the structure of Pascal's triangle structure (this animation summarizes the key information needed: https://upload.wikimedia.org/wikipedia/commons/0/0d/PascalTriangleAnimated2.gif). Write a recursive function `pascal (n :Int) :List[Int]` that generates the nth row of Pascal's triangle. For example, `pascal(1)` should generate `List(1)`, `pascal(2)` should generate `List(1,1)`, `pascal(3)` should generate `List(1,2,1)` and `pascal(4)` should generate `List(1,3,3,1)`.

[16]Exercise 3.23 [Chiusano, Bjarnason 2014]
[17]Exercise 3.24 [Chiusano, Bjarnason 2014]