Algebraic Data Types in Scala

Do not use variables, side effects, exceptions or return statements (unless explicitly asked for). There is no hand-in this week. Please rely on automatic tests and compiler errors to see whether you are doing fine. Also remember to ask TAs or Andrzej to get feedback.

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

```
import adpro.adt.List.*
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 + t
  case _ => 101
```

Which function from `adpro.adt` is called in this example?

Remark. For pedagogical reasons, all exercises below use our own implementation of lists, not the Scala standard library. Do not use API docs online to find the available functions, as they will be different. Find our implementation in `Exercises.scala`, in the very top. This is the only API that is available. Additionally, the functions from earlier exercises can be used in solving the later ones.

Exercise 2. Implement the function `tail` for removing the first element of a list. The function should run in constant time. Throw the `NoSuchElementException` exception if given an empty list.[2]

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

Exercise 3. Generalize `tail` to `drop`, a function that removes the first `n` elements from a list. The running time should be proportional to `n`—no need to make a copy of the list. Throw `NoSuchElementException` if the list is too short.[3] For non-positive `n` the list is unchanged.

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

Exercise 4. Implement `dropWhile`, which removes elements starting the head of the list `l`, as long as they satisfy a predicate `p`. Do not use exceptions: if all elements satisfy `p` then return the empty list.[4]

```
def dropWhile[A](l: List[A], p: 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. Given `List(1, 2, 3, 4)`, the function returns `List(1, 2, 3)`. Throw `NoSuchElementException` 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]

[1]Exercise 3.1 [Pilquist, Chiusano, Bjarnason, 2022]
[2]Exercise 3.2 [Pilquist, Chiusano, Bjarnason, 2022]
[3]Exercise 3.4 [Pilquist, Chiusano, Bjarnason, 2022]
[4]Exercise 3.5 [Pilquist, Chiusano, Bjarnason, 2022]
[5]Exercise 3.6 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 6. Compute the length of a list using `foldRight`.[6] Remember that `foldRight` has been presented briefly in the lecture slides, in the text book; it can also be found in the top of the file `Exercises.scala`. Also, the next exercise has an example demonstrating the essence of `foldRight`.

```
def length[A](l: 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](l: 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.[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.[9]

Exercise 10. Write `foldRight` using `foldLeft` and `reverse`. The left fold performs the dual operation to the right one, so if you reverse the list you should be able to simulate one with the other.

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.

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).

Hint: Synthesize a function that computes the run of `foldLeft`, and then invoke this function. To implement `foldLeft[A, B]` you will be calling `foldRight` with the following type parameters:

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

This will compute a new function, which then needs to be called.[10]

Note: From now on, use of recursion is bad-smell for us. Recursion should only be used, when dealing with a non-standard iteration. Otherwise a suitable HOF should be used. Similarly, a fold should only be used if any of the other simpler HOFs cannot be.

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` that concatenates two lists (find it in the book and in the source file).[11]

[6] Exercise 3.9 [Pilquist, Chiusano, Bjarnason, 2022]
[7] Exercise 3.10 [Pilquist, Chiusano, Bjarnason, 2022]
[8] Exercise 3.11 [Pilquist, Chiusano, Bjarnason, 2022]
[9] Exercise 3.12 [Pilquist, Chiusano, Bjarnason, 2022]
[10] Exercise 3.13 [Pilquist, Chiusano, Bjarnason, 2022]
[11] Exercise 3.15 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 13. Implement `filter` that removes from a list the elements that do not satisfy `p`.[12]

```
def filter[A] (l: List[A]) (p: A =>Boolean): List[A]
```

Exercise 14. Write a function `flatMap` that works like map except that `f`, the function mapped, returns a list instead of a single value, and the result is automatically flattened to a list like with `concat`:

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

For instance, `flatMap(List(1, 2, 3)) (i =>List(i, i))` results in `List(1, 1, 2, 2, 3, 3)`. Together with `map`, (`flatMap`) will be key in the rest of the course. Understand this well.[13]

Exercise 15. Use `flatMap` to re-implement `filter`.[14]

Exercise 16. Write a recursive function that accepts two lists of integers and constructs a new list by adding elements at the same positions. Trailing elements of either list are dropped if the lists are not of the same length. For example, the lists `List(1,2,3)` and `List(4,5,6,7)` become `List(5,7,9)`.[15]

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]

[12] Exercise 3.19 [Pilquist, Chiusano, Bjarnason, 2022]
[13] Exercise 3.20 [Pilquist, Chiusano, Bjarnason, 2022]
[14] Exercise 3.21 [Pilquist, Chiusano, Bjarnason, 2022]
[15] Exercise 3.22 [Pilquist, Chiusano, Bjarnason, 2022]
[16] Exercise 3.23 [Pilquist, Chiusano, Bjarnason, 2022]
[17] Exercise 3.24 [Pilquist, Chiusano, Bjarnason, 2022]