

# Functional Programming

## Lab session 2a

### Introduction

The second lab session of the course *Functional Programming* consists of two parts (part 2a and part 2b). The reason for this split is that we did not cover enough material in the lectures yet to make the exercises of part 2b. This part will be published later.

The focus in part 2a is on the use of built-in Haskell functions, especially higher order functions. You are not allowed to use explicit recursion in your solutions of the exercises of part 2a.

Part 2a is worth 40 points in total (out of 100 points). It consists of 8 exercises, each one is worth 5 points. Note that passing all tests in Themis will not automatically mean that you scored all points. The TAs will manually check accepted solutions for forbidden usage of recursion (and subtract points if needed).

For some of the exercises, solutions can easily be found on the internet. Be warned, that copying those is considered plagiarism! Moreover, many of these published solutions are programmed in imperative languages.

### Exercise 1: Products of even and odd Integers

Write a Haskell function `productEvenOdd :: [Integer] -> (Integer, Integer)` which should return a pair consisting of the product of all even integers in the input list and the product of all odd integers in the input list. For example, `productEvenOdd [1,2,3,4,5]` should return `(8,15)`, since  $2 \cdot 4 = 8$  and  $1 \cdot 3 \cdot 5 = 15$ .

### Exercise 2: Function Composition

Let the functions `f` and `g` have the type `f, g :: a -> a`. The Haskell function composition operator `(.)` takes two functions and returns the function which is the composition of `f` and `g`. It is defined in the prelude as `f.g = \x -> f (g x)`. For example, if `f x = x*x` and `g x=x+1` then `f.g` is the function that maps `x` to `(x+1)*(x+1)`. Write a Haskell function `compose :: [a -> a] -> a -> a` that takes a finite list of functions of type `a -> a` and produces the function which is the composition of this list. For example, `compose [f, g, h]` should produce the function `f.(g.h)`. Your implementation should make use of (a) higher order function(s).

### Exercise 3: Run Remover

In this exercise the task is to remove *runs* (consecutive duplicates) of characters from a `String`. For example, the string "aaabbcaaabb" should be 'reduced' to the string "abcab".

Write a Haskell function `removeRuns :: String -> String` that removes all consecutive repeated characters from a string.

## Exercise 4: Horner's rule

Consider a polynomial of degree  $n$  written in the usual form

$$P(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n.$$

Evaluating this expression directly (term by term) requires many exponentiations and multiplications. A much more efficient method is due to Horner, who observed that the polynomial can be rewritten by repeatedly factoring out  $x$ :

$$P(x) = (((a_0x + a_1)x + a_2)x + \cdots)x + a_n.$$

This nested form has the advantage that it uses only  $n$  multiplications and  $n$  additions.

In this exercise we represent the coefficients  $a_0, \dots, a_n$  as a list of `Integers` in *decreasing* order of degree. For example, the list `[3,1,4,1,5]` represents the polynomial  $P(x) = 3x^4 + x^3 + 4x^2 + x + 5$ .

Write a function `evalPoly :: [Integer] -> Integer -> Integer` such that `evalPoly coeffs x` evaluates the given polynomial at the location `x`. For example, `evalPoly [3,1,4,1,5] 1` should return 14. Your implementation must make use of `foldl` to implement Horner's method.

## Exercise 5: Increasing numbers

Given an input list of integers, the aim is to make a filter that preserves those elements for which the value strictly increases compared to the previous one. For example, in the list `[3,1,4,1,5,9]` the value increases at  $1 \rightarrow 4$ ,  $1 \rightarrow 5$ , and  $5 \rightarrow 9$ , so the result should be `[4,5,9]`.

Write a Haskell function `increasing :: [Integer] -> [Integer]` that returns all elements (in the same order as the input) that are strictly larger than the element immediately preceding them.

## Exercise 6: Interleaving Strings

Given a list of strings, we merge into a single string by repeatedly taking the first remaining character of each non-empty string, in order. This process is called *round-robin interleaving*. For example, if the input strings are `["Hello", "el wrd", "l!"]` then the first characters we take are

`'H'`, `'e'`, `'l'`,

then the second characters

`'l'`, `'l'`, `'!'`,

and so on, skipping any strings that have already become empty. The resulting output string is `"Hello world!"`. Write a Haskell function `interleaveRR :: [String] -> String` that performs this round-robin interleaving of a list of input strings.

## Exercise 7: Parentheses

A string containing parentheses is said to be *balanced* if each opening parenthesis “`(`” is closed by a matching “`)`”, and at no position does a closing parenthesis occur without a preceding unmatched opening one. For example, `"((())()`” is balanced, while `"((())` and `"())(()"` are not.

Write a Haskell function `balancedParen :: String -> Bool` that determines whether its input string contains balanced parentheses. Note that only the characters `'(` and `')` are significant; any other characters should be ignored (skipped).

## Exercise 8: Most common character

In this exercise we consider the problem of finding the most frequently occurring character in a string. For example, in the string "Haskell is great" the character 'a' appears three times, which is more than any other character.

Write a Haskell function `mostCommon :: String -> Char` that returns the character that occurs most often in the input string. You may assume the input string is non-empty. If several characters are tied for the highest frequency, the largest one (in terms of ASCII table order) should be returned.