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 infinite lists and higher order functions. Part 2a is worth 50 points in total. It consists of 5 exercises, each worth 10 points. Note that passing all tests in Themis will not automatically mean that you scored 50 points. The TAs will manually check accepted solutions for a Haskellish style of programming using infinite lists and higher order functions. If you solve an exercise without making proper use of these techniques, then you will not receive points for that exercise.

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: 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` that takes a finite list of funtions 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 2: Smallest Fix Point

We call `x` a *fix point* of a function `f` if `f x==x` (i.e. `f` maps `x` to itself). Write a function `fixpoint` which accepts a function `f :: Integer -> Integer` and returns the smallest non-negative integer `x` which is a fix point of `f`. You may assume that the test functions in Themis are chosen such that a fix point exists.

Exercise 3: Random Generator

In many computer programs we need a *random number generator*. The numbers produced by these generators are not really random, but they appear to be random. This explains why these generators are sometimes called called pseudo-random generators. One of the simplest generators of this type are the so-called *linear congruential generator*s. These generators produce a series of numbers X_i using the recurrence

$$X_{i+1} = (a \cdot X_i + c) \mathbf{\ mod\ } m.$$

The values for the parameters a, c, and m must be chosen wisely. If you are interested in how to choose these values, then you may want to read `https://en.wikipedia.org/wiki/Linear_congruential_generator`.

In this exercise, we choose $X_0 = 2773639$, $a = 25214903917$, $c = 11$, and $m = 2^{45}$. These numbers (apart from X_0) correspond with the values that are specified in the POSIX standard for the C-function `drand48`.

Give a Haskell implementation of the infinite list `random :: [Integer]` that returns the infinite list of random numbers generated by this set of parameters. The time complesxity of `take n random` should be linear in `n`. The expression `take 6 random` should result in:
`[2773639,25693544934790,35087648281,25863180521136,928172761339,19643099434218]`

Exercise 4: Trailing Zeros

For an integer $n > 0$, the function $Z(n)$ is defined as the number of zeros on the very right of the binary representation of n (i.e. the number of trailing zeros). Clearly, for any odd number this function is zero. The first fifty values of the series $Z(n)$ (starting with $n = 2$) are:

$$1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0$$

Implement the infinite list `trailingZeros :: [Integer]` such that `trailingZeros!!(n-1)` yields $Z(\texttt{n})$. You are not allowed to implement `trailingZeros` in the style `trailingZeros = [z n | n <- [2..]]`, so $Z(n)$ should not be computed for each n separately.

[Hint: if you take the list of natural numbers, multiply them by two, and then go through them and insert the correct odd number in between the now all-even numbers, you end up with the list of natural numbers again.]

Exercise 5: Prime Sieve

Apart from the sieve of Eratosthenes, there exist several other sieves that produce the infinite list of prime numbers. One of these sieves generates the list of odd primes upto some upperbound $2n + 1$ as follows:

- Start with the list of integers from 1 to n.
- From this list, remove all integers of the form $i + j + 2ij$ where $1 \leq i \leq j$.
- The remaining numbers are doubled and incremented by one. The result is the list of all the odd prime numbers less than $2n + 2$.

Write a Haskell definition of the lazy infinite list `primes::[Integer]`, which is the list of all primes that is generated with an adapted unbounded version of this sieve. Of course, `take 5 primes` must produce `[2,3,5,7,11]`.