Functional Programming
Lab session 1

Introduction

The first lab session of the course *Functional Programming* consists of 7 programming exercises. Exercise 7 is worth 20 points. The other exercises are worth 10 points each. The remaining 20 points are awarded (by manual inspection by the teaching assistants) for (a Haskellish) style of programming and efficiency. Hence, if you solved all problems, your grade is not automatically a 10. Lab sessions are made individually (so not inpairs/groups!). 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.

Very important: You submit your solutions to Themis (see `https://themis.housing.rug.n)`. If you are required to write some function `f :: a -> b` then the filename of your solution must be `f.hs` (case sensitive!) and the type of the arguments and return value of the function must match. If you fail this requirement, then Themis cannot compile your program. You should upload to Themis a file containing only the function that you were required to make and the implementation of helper functions (if needed). Do not write a main function/program. Also, do not import any library/module. The standard prelude file is sufficient.

Exercise 1: Divisible by 7

It is easy to test whether a positive integer `n` is divisible by 7 using the `mod` function. The answer is simply given by the Haskell expression `n `mod` 7==0`. However, there exists an interesting alternative method which we will study in this exercise.

Let n be a positive integer. If $n < 7$ then n is (clearly) not divisble by 7. If $n = 7$ or $n = 49$ then it clearly is divisible by 7. For any other n, split n in two numbers p and q, where p is the number which is obtained by removing the last digit of n and q is the last digit itself. Next, we compute $n' = p + 5 \cdot q$. If n' is divisible by 7, then so is n. Of course, we can repeat this process until $n \leq 7$ or $n = 49$. You do not have to prove that this algorithm works (especially termination is tricky), and may simply assume that it works (trust me, it does!).

Implement a Haskell function `div7 :: Integer -> Integer` such that the call `div7 n` returns the number of steps that this process needs to determine whether `n` is divisible by 7.

For example `div7 4` should immediately return 0, while `div7 17` should return 16 because

$$17 \to 36 \to 33 \to 18 \to 41 \to 9 \to 45 \to 29 \to 47 \to 39 \to 48 \to 44 \to 24 \to 22 \to 12 \to 11 \to 6$$

Maybe more impressive is the computation of `div7 (2^12345)` which should return 3717.

Exercise 2: Alternating Fibonacci Numbers

Of course, you know the standard recurrence for the Fibonacci numbers. In this exercise, however, we consider a Fibonacci like series which is defined by the following recurrence:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + (-1)^{n+1} F_{n-2} \end{aligned}$$

Make a function `altFib :: Integer -> Integer` such that the call `altFib n` returns the number F_n. Your program should be able to compute `altFib 10000` within 1 second. You can time code in `ghci` by typing `:set +s` at the command prompt. From that point on, the computation time is reported after each computation. However, be warned for caching! Often, doing a calculation after you did it before yields a cached answer with a computation time close to 0.0 seconds.

Exercise 3: Fermat Liars

Fermat's little theorem states that if p is a prime number and b is an integer such that $1 < b < p - 1$, then

$$b^{p-1} \textbf{ mod } p = 1$$

Note that this is an implication, and not an equivalence. The converse implication is in general not true. For example, for $b = 2$ the smallest value n for which the converse fails is $n = 341$. This number is composite ($341 = 11 \times 13$) while 2^{340} **mod** $341 = 1$.

Still, many so-called *probabilistic primality tests* are based on Fermat's little theorem. If one wants to test whether a number n is prime, then we can pick random integers b (where $1 < b < n - 1$) and test whether b^{n-1} **mod** $n = 1$. If the equality does not hold for some value of b, then n is composite (i.e. not prime). If it holds, then we can redo the test (as many times as we like) using another choice for the value of b. It is extremeley unlikely (and many primality tests are based on this) that a number n that passes (say) 100 times is composite. Threfore, we call the number a *probable prime*. Sometimes these number are called *commercial prime tests* because they are used for generating public/private key pairs in encryption methods (like RSA, and PGP).

If for some composite value (i.e. non-prime) n we have b^{n-1} **mod** $n = 1$ then we call the base b a *Fermat-liar* for n.

Each integer $b > 1$ is a Fermat liar for some composite n (you may accept this statement to be true without proof). Write a function `fermatLiar :: Integer -> Integer` that accepts an integer `b` and outputs the smallest integer `n` (where `n>b`) such that `b` is a liar for `n`. For example, `fermatLiar 2` should return 341.

Exercise 4: Prefix Strings

A string `p` is a *prefix* of a string `s` if there exists a string `t` such that `s = p++t`. For example, the string `"101"` is a prefix of the string `"101110"`, because `"101110"="101" ++ "110"`.

In this exercise we consider strings consisting solely of zeroes and ones. The string `"00110100101"` is special in the sense that for each of its prefixes the number of ones is less or equal to the number of zeroes. Write a function `numPref :: Int -> Int` that accepts an `Int n` and outputs the number of strings of length `n` that satisfy the above property. You may assume that the number `n` is at most 20. The calculation of `numPref 20` should not exceed one second.

Exercise 5: Last n digits

Write a function `lastDigits :: Integer -> Int -> [Integer]` such that the call `lastDigits n d` returns the list of the last `d` digits of the number $\sum_{k=1}^{n} k^k$. For example, the call `lastDigits 10 10` should

yield the list `[0,4,0,5,0,7,1,3,1,7]`, since $1^1 + 2^2 + .. + 10^{10} = 10405071317$. The time to compute `lastDigits (10^5) 10` using `ghci` should not exceed five seconds.

Exercise 6: Polynomial Multiplication

In this exercise we consider the *Polynomial multiplication* of two polynomials in the variable x. The coefficients of the polynmials are integers, and the exponents are non-negative integers. We represent a polynomial by a list of coefficients. For example, the polynomial $6x^3 - 2x^2 + 3$ is represented by the list `[6,-2,0,3]`. The multiplication of the polynomials $6x^3 - 2x^2 + 3$ and $x^2 + 1$ yields $6x^5 - 2x^4 + 6x^3 + x^2 + 3$.

Write a Haskell function `polMult` that performs polynomial multiplication given two lists of coeffients. So, `polMult [6,-2,0,3] [1,0,1]` should return `[6,-2,6,1,0,3]`. You may assume that both arguments of `polMult` are non-empty lists. Moreover, you may assume that the trivial polynomial which is zero everywhere is represented by the list `[0]` (so `[0,0]` is not valid). For any other polynomial, the first element of the list (i.e. the `head`) is non-zero, so `[0,1,2]` is not a valid representation of a polynomial while `[1,2]` is.

Exercise 7: Minimal Steps

Consider the following iterative process. Given two positive integers n, and g (where $0 < n \leq g$), we try to reach g starting from n, where we are allowed to iteratively use the operations `triple` ($n \to 3 \times n$), `double` ($n \to 2 \times n$), and `increment` ($n \to n + 1$).

For example, let $n = 1$, and $g = 42$. We can reach 42 in 5 steps in the following way:

$$1 \to_{+1} 2 \to_{\times 3} 6 \to_{+1} 7 \to_{\times 3} 21 \to_{\times 2} 42$$

Clearly, several other possibilities exist (e.g. $1 \to_{+1} 2 \to_{+1} 3 \to_{\times 2} 6 \to_{+1} 7 \to_{\times 3} 21 \to_{\times 2} 42$). However, there is no way to reach 42 (starting in 1) with fewer than 5 steps.

Write a Haskell function `minSteps :: Int -> Int -> Int` such that the call `minSteps n g` returns the minimum number of computational steps to get from `n` to `g`. So, `minSteps 1 42` returns `5`. You may assume that `1<n<=g<10000`.