lab session 2b: Functional Programming

Part 2b is worth 60 grade points in total. It consists of 8 exercises (exercise 9..16). Exercise 14 is worth 15 points. The exercise 13 and 16 are worth 10 points. The other exercises are worth 5 points (totalling 60 points). Together with the 40 points of lab 2a you can score 100 points (= grade 10) for lab 2.

Exercise 9: 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 :: (Integer -> Integer) -> Integer` which accepts a function `f` 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 10: 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:

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

Define the infinite list `trailingZeros :: [Integer]` such that `trailingZeros!!(n-1)` yields $Z($`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 numbers in between the now all-even numbers, you end up with the list of natural numbers again.]

Exercise 11: 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

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

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 12: Easy as pi

The first 51 digits of π (including the number 3 before the decimal dot) are:

3.14159265358979323846264338327950288419716939937510

These digits were computed using a so-called *spigot* algorithm (see wikipedia if you want to know more about it). From Themis you can download the file `spigotPi.hs`. It contains the following code:

```
-- Infinite list of decimal digits of pi: 3,1,4,1,5,9,...
piDigits :: [Integer]
piDigits = digits 1 0 1 1 3 3
  where
    digits q r t k n l
      | 4*q + r - t < n*t =
          -- We can safely emit digit n
          n : digits (10*q) (10*(r - n*t)) t k n' l
      | otherwise =
          -- Need to refine the approximation first
          digits (q*k) ((2*q + r)*l) (t*l) (k+1) n'' (l+2)
      where
        n'  = (10*(3*q + r)) `div` t - 10*n
        n'' = (q*(7*k + 2) + r*l) `div` (t*l)

neededPiDigits :: Int -> Int
neededPiDigits n = -- implement yourself
```

The function call `neededPiDigits n` should return the number of digits of π that are needed such that it contains all numbers from 0 to `n`. For example `neededPiDigits 0` returns 33, because the digit 0 is the 33th digit. However, all other digits occur already before this 0, so `neededPidigits 9` also returns 33. The call `neededPiDigits 10` returns 51.

Exercise 13: Prime Sieve

Apart from the sieve of Eratosthenes, there exist several other sieves that produce the infinite list of prime numbers. The following sieve generates, given an positive integer n, the list of odd primes upto $2n + 1$ 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]`.

Exercise 14: Parsing and Evaluating Integer Expressions

We consider integer valued arithmetic expressions that can be constructed using the following grammar, which is an extension of the grammar that was discussed in the lecture:

```
E  ->  T E'
E' ->  + T E'
E' ->  - T E'
E' ->  <empty string>
T  ->  F T'
T' ->  * F T'
T' ->  / F T'
T' ->  % F T'
T' ->  <empty string>
F  ->  ( E )
F  ->  <integer>
```

Note that the operator '`/`' is actually the integer `div` operator, and that '`%`' is the `mod` operator. The notation `<integer>` denotes an integer literal (like `42`).

The structure of the above grammar is such that operator precedence is automatically correct. So, an expression like `1+5*3` will be parsed the same as the expression `1+(5*3)` (and not like `(1+5)*3`).

On Themis, you can find the file `parseExpr.hs` which contains the parser for the minimal expression parser that was discussed in the lecture. In this parser the parsing routine produce pairs of the type `(String, [String])`. In this exercise you need to extend the parser such that it parses the above (complete) expression grammar. Moreover, you need to change the type of the parsing routines such that the parsed expression is evaluated on-the-fly. The main parsing routine must be `evalParser :: String -> Integer`. For example, `evalParser "80/2+2"` should return `42`.

Exercise 15: Unary minus

The expression grammar of the previous exercise does not allow numbers like `-21`. Hence, it is not possible to parse strings like `41 - -1` and `-2*(-21)"`. Extend the parser such that the unary minus is also supported.

Exercise 16: Exponentiation

Extend the parser of the previous exercise with exponentiation. We will use the operator `^` to denote exponentiation. Note that the exponetiation operator is right-associative, so `2^3^4` is equal to `2^(3^4)`.

For example, `evalParser "2^3^4"` should return `2417851639229258349412352`. On the other hand, `evalParser "(2^3)^4"` should return `4096`. Note that the unary minus binds stronger than the exponentiation operator. So, `evalParser "5 - -2^2"` should return `1`, and `evalParser "5 + -2^3"` should return `-3`.