lab session 2: Functional Programming

The second lab session of the course *Functional Programming* consists of 8 programming exercises. The first six exercises are worth one grade point each. The last two exercises are worth two points each. No partial points are given for passing some but not all test cases of an exercise. You submit your solutions (per exercise a single ASCII file containing Haskell functions) to Themis (`https://themis.housing.rug.nl`). Note that it is essential that the types of your solutions matches exactly the types that are given in the exercises, otherwise judgment by Themis will fail.

Note that in the first lab your grade was completely determined by Themis. However, your grade for lab sessions 2 and 3 will be computed on the basis of points awarded by Themis followed by a possible subtraction of points through manual inspection for (a Haskellish) style of programming and efficiency (by the teaching assistants). Hence, for those lab sessions your grade is not automatically a 10 in the case that Themis accepted all your submissions.

For some of the exercises, solutions can easily be found on the internet. Be warned, that copying those is considered plagiarism (and will be forwarded to the board of examiners)! Moreover, many of these published solutions are programmed in an imperative style.

Exercise 1: Smallest multiple

Write a Haskell function `smallestMultiple :: Integer -> Integer` such that `smallestMultiple n` returns the smallest number than can be divided by each of the numbers from 1 up to and including `n` without any remainder. For example, `smallestMultiple 10` should yield the answer `2520`. Your solution must make use of the standard Haskell function `foldr`, and should compute `smallestMultiple 10000` within one second (using `ghci`).

Exercise 2: Factorization of composites

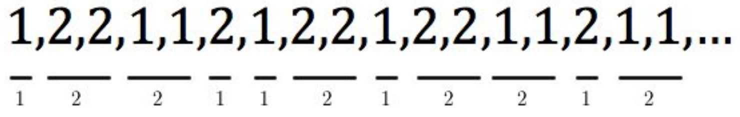
In the lecture it was shown that the infinite list of prime numbers can be defined as:

```
primes :: [Integer]
primes  =  sieve [2..]
  where
    sieve :: [Integer] -> [Integer]
    sieve (p:xs)  =  p : sieve [x | x <- xs, x `mod` p /= 0]
```

Using this code, write a Haskell definition of the infinite list `composites::[(Integer,[Integer])]`. Each element of this list is a pair, of which the first element is a composite number, and the second element is its prime factorization (an ascending list of prime factors). For example, `take 5 composites` must return `[(4,[2,2]),(6,[2,3]),(8,[2,2,2]),(9,[3,3]),(10,[2,5])]`.

Exercise 3: Run-length sequence

Consider the following sequence: $1, 2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2, 1, 1, ...$
The sequence only contains 1s and 2s. They appear either in a run of one, or in a run of two. If we start at the beginning of the sequence, then the lengths of the runs recreate the original sequence: there is **1** one, followed by **2** twos, followed by **2** ones, followed by **1** two, followed by **1** one, etc. If you put the bold-face runlengths in a sequence, then we arrive at the original sequence.



Write a Haskell definition of the list `selfrle::[Int]` that returns this infinite list. For example, `take 29 selfrle` must yield `[1,2,2,1,1,2,1,2,2,1,2,2,1,1,2,1,1,2,2,1,2,1,1,2,1,2,2,1,1]`.

Exercise 4: Fibonacci and Catalan numbers

Of course, you know the recurrence that defines the *Fibonacci sequence*:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

The first 10 terms of the Fibonacci sequence are: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34$.
Another famous recurrence generates the *Catalan sequence*:

$$C_0 = 1, \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

The first 10 terms of the Catalan sequence are: $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862$.

Write a Haskell definition of the infinite list `fibcat::[Integer]` that returns the infinite ascending list of positive integers that are either a Fibonacci number or a Catalan number (possibly both). For example, `take 11 fibcat` should yield the answer `[0,1,2,3,5,8,13,14,21,34,42]`.

Exercise 5: Decimal expansion

The decimal representation of $\frac{43}{42}$ is 1.023809523809523808... The number of digits in this decimal expansion is infinite, but it can be written as the finite prefix 1.0 followed by an infinite repetition of the finite digit string "238095". So, we could write $\frac{1}{42}$ as 0.0(238095)*, where $*$ means infinite repetition.

In general, the number $\frac{n}{d}$ can be written as $\alpha(\beta)*$, where α is a non-repeating prefix, and β is the repetitive part of the decimal representation. Of course, we want β to be the shortest possible digit string: so in the above example $\alpha = 1.0$ and $\beta = 238095$ (and not 238095238095). Also note, that β can be the empty string, since simple fraction like $\frac{3}{2}$ have a finite expansion.

Write a Haskell function `rationalToDecimalString :: Int -> Int -> String` that, given two positive arguments `n` and `d` returns the decimal expansion of $\frac{\mathtt{n}}{\mathtt{d}}$. So, `rationalToDecimalString 3 2` should return `"1.5"`, while `rationalToDecimalString 43 42` should return `"1.0(238095)*"`.

Exercise 6: Evaluation of multiplicative expressions

In the lecture we have discussed a parser for integer expressions that only make use of the operators `*` and `-`. The presented parser has the type `parser :: String -> (String,Tokens)`, and takes as its input a string containing an expression, and outputs the accepted part of the expression (a string) and the remaining input tokens. For example, `parser "21*8/2/2 7*2" = ("21*8/2/2",["7","*","2"])`. On Themis (and also on Brightspace) you can find the source code of this parser (files `parser.hs` and `Debug.hs`).

The aim of this exercise is to evaulate integer expressions instead of just parsing them. Change the parser such that its type becomes `parser :: String -> (Integer,Tokens)`. The parser should return a pair of which the first element is the evaluation of the accepted input expression, an the second element is the list of remaining tokens. So, `parser "21*8/2/2 7*2" = (42,["7","*","2"])`.

Note that division is integer division (so `17/5` is 3). You must also introduce the `%` operator (modulus) in the parser. Moreover, we will introduce expressions enclosed by parentheses and negative numbers such that `-(42%(5*3))=-12`, while `-(42%5*3)=-((42%5)*3)=-6`.

In conclusion, you should adapt the parser such that it evaluates integer expressions that can be constructed with the following grammar.

```
S  -> F S'
S' -> * F S'
S' -> / F S'
S' -> % F S'
S' -> <empty string>
F  -> <digits>
F -> ( S )
F -> - F
```

Exercise 7: Evaluation of basic arithmetic expressions

In this exercise, you are expected to extend the parser from the previous exercise such that addition and subtraction is also allowed. Note that `+` and `-` have a lower priority than the multiplicative operators. This hierarchy is reflected in the following adapted grammar:

```
S  -> E
E  -> T E'
E' -> + T E'
E' -> - T E'
E' -> <empty string>
T  -> F T'
T' -> * F T' | / F T' | % F T' | epsilon
F  -> <digits>
F -> ( S )
F -> - F
```

As an example, `parser "7*(1+1)*(2+1) 13"` should now return `(42,["13"])`.

Exercise 8: Evaluation of arithmetic expressions

Extend the parser of exercise 7 such that the exponentiation operator `^` is supported. You may assume that for all test cases in Themis, the exponent is a non-negative integer. Note that, in contrast with the other operators, exponentiation associates to the right. So `2^2^3=2^(2^3)=2^8=256` while `(2^2)^3=4^3=64`. So, `parser "2^2^3"` should return `(256,[])`.