lab session 1: Functional Programming

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 last 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. 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: Happy numbers

A *happy number* is defined by the following process:

> *S*tarting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers, while those that do not end in 1 are unhappy numbers.

Make a function `countHappyNumbers` such that the call `countHappyNumbers a b` returns the number of happy numbers in the interval `[a,b]`. Your program should be able to compute `countHappyNumbers 1 100000` within 5 seconds. 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 caluclation after you did it before yields a cached answer, i.e. a computation time close to 0.0 seconds.

Exercise 2: Takuzu Bitstrings

A *Takuzu* is a logic-based number placement puzzle[1]. The objective is to fill a $n \times m$ grid with 1s and 0s, where there is an equal number of 1s and 0s in each row and column (hence n and m are even numbers) and no more than two of either number adjacent to each other. Additionally, there can be no identical rows, nor can there be identical columns.

In this exercise, you are asked to make a Haskell function `takuzuStrings n` that generates the list of all bitstrings with length `n` that satisfy the property that no more than two 0s or 1s are adjacent to each other. Moreover, the list should be ordered in lexicographic increasing order. Note that in this exercise, we ignore the rule that a string must consist of as many 0s as 1s. However, later in exercise 7 we do need to impose this rule. As an example, `takuzuStrings 4` should produce the output:

```
["0010","0011","0100","0101","0110","1001","1010","1011","1100","1101"]
```

[1] In Dutch newspapers, this type of puzzles is often called a *Binaire* or *Binairo*.

Exercise 3: RPN evaluator

In *reverse Polish notation (RPN)* arithmetic expressions are written using post-fix notation, i.e. the operators follow their operands; for instance, to add 3 and 4, one would write `3 4 +` rather than `3 + 4`. Another example, using multiple operations is `3 4 - 5 +`, which in conventional notation is `3 - 4 + 5`. A nice property of RPN is that there is no need for parentheses, e.g. in conventional notation we need parentheses in `(7 - 2)*5`, but in RPN this is simply written as `7 2 - 5 *`.

Write a Haskell function `rpnEval` that evaluates a string in RPN into an `Integer` value. You may assume that the input string consists of non-negative integers, and the operators `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (integer division, i.e. `div`). Moreover, the input contains spaces. As an example, `rpnEval "17 2 - 2 * 5 /"` should produce the answer `6`.

Exercise 4: Last n digits

Write a Haskell function `lastDigits n d` that computes the list of the last `d` digits of the number $\sum_{k=0}^{n} n^k$. For example, $\sum_{k=0}^{5} 5^k = 5^0+5^1+5^2+5^3+5^4+5^5 = 1+5+25+125+625+3125 = 3906$, so `lastDigits 5 3` should yield the list `[9,0,6]`. In case the sum has less digits than `d`, then only the digits of the sum should be returned (i.e. a list which has a length less than `d`). The time to compute `lastDigits (10^5) 10` using `ghci` should not exceed five seconds.

Exercise 5: Polynomial long division

Polynomial long division is an algorithm for dividing a polynomial by another polynomial of the same or lower degree, a generalised version of the familiar arithmetic technique called *long division*. For example, the polynomial term $6x^3 - 2x^2 + x + 3$ divied by the term $x^2 - x + 1$ yields $6x + 4$ with the remainder term $-x - 1$. This is depicted graphically in the following calculation:

$$\begin{array}{r} \left(\quad 6x^3 - 2x^2 \;\; + x + 3\right) \div \left(x^2 - x + 1\right) = 6x + 4 + \dfrac{-x-1}{x^2-x+1} \\ \underline{-\,6x^3 + 6x^2 - 6x} \\ \qquad\quad 4x^2 - 5x + 3 \\ \qquad\quad \underline{-\,4x^2 + 4x - 4} \\ \qquad\qquad\quad -x - 1 \end{array}$$

A detailed explanation be found at `https://en.wikipedia.org/wiki/Polynomial_long_division`.

We can represent a polynomial by a list of coefficients. For example, we can represent the polynomial $6x^3 - 2x^2 + x + 3$ by the list `[6.0,-2.0,1.0,3.0]`.

Write a Haskell function `polDivision` that performs polynomial division. Its first argument is the dividend, the second the divisor. The function should return a pair of polynomials, the quotient and the remainder.

For example `polDivision [6.0,-2.0,1.0,3.0] [1.0 -1.0 1.0]` should produce the pair of ploynimials `([6.0,4.0],[-1.0,-1.0])`. In case there is no remainder, the second list must be `[]`. Moreover, a polynomial has no leading coefficients that are zero (i.e. `[0.0,6.0,-2.0,1.0,3.0]` is not considered valid).

Exercise 6: Improved Caesar cipher

A Caesar cipher is one of the simplest and most widely known encryption techniques. The method is named after Julius Caesar, who used it in his private correspondence. It is a type of substitution cipher in which each letter in a text is replaced by a letter some fixed number of positions down the alphabet. For example, with a right rotation of 3 positions, A would be replaced by X, B by Y, C by

Z, D by A, etc. Coding (and decoding) is easy if the alphabet and the rotated alphabet are placed in alignment:

```
Plain:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:   XYZABCDEFGHIJKLMNOPQRSTUVW
```

In the above example, the *key* of the coder is 3, being the number of positions over which the alphabet is shifted to the right during the coding process. This key is used in the following example.

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

Using statistical knowledege about the number of occurrences of letters in a given language, the Caesar cipher is pretty easy to crack. In the *improved Caesar cipher* this is a lot harder. In this scheme the alphabet is rotated by `key` positions again after having used a letter from the alphabet. For example, using the key 3, the text `AAA` is now encoded as follows: the first `A` is encoded as `X`, next the alphabet is rotated again by 3 positions yielding the mapping of the second `A` onto `U`. After another rotation, the third `A` is mapped on `R`, yielding the encoded text `XUR`.
Write the Haskell functions `cipherEncode` and `cipherDecode` that accept two arguments: a small integer (the key) and a string. The functions should yield the coding (or decoding) of the second argument, given the first argument as the key. You may assume that the input consists only of the uppercase letters (A..Z), and spaces (which are not replaced).
So, `cipherEncode 3 "AAA"` should return `"XUR"`, while `cipherDecode 3 "XUR"` should return `"AAA"`.

Exercise 7: Takuzu solver [Hard]

In the second exercise, the rules are given for a Takuzu puzzle. An example of such a puzzle is given in the following figure.

1		1		1		0	0
		1				0	
	0				1		
0	0					1	1
0			1				1
	1						
			1			0	
1	1			0	0		0

A *correct* Takuzu should have a unique solution. An incorrect takuzu has no solution at all, or it has more than one solution. Write a Haskell function `isCorrectTakuzu` that takes as its input a takuzu, and produces the output `True` if the takuzu is a correct takuzu, or `False` otherwise. For example, the call

```
isCorrectTakuzu ["1.1.1.00",
                 "..1...0.",
                 ".0...1..",
                 "00....11",
                 "0..1...1",
                 ".1......",
                 "...1..0.",
                 "11..00.0"]
```

should yield the answer `True`. Note that the input format is a list of lists, one list per row of the grid. A '.' represents an empty cell. In fact the example input corresponds with the takuzu in the above figure (and is downloadable from Nestor).