Lab 1: Functional Programming and (Some) Logic

Note

Do not get discouraged if you find some of the exercises difficult. It does not matter if you are not able to finish all exercises.

Just submit what you have before the deadline. Please indicate the time spent on every exercise.

```
> module Lab1 where
> import Data.List
> import Test.QuickCheck
```

Some stuff from the lecture slides:

```
> prime :: Integer -> Bool
> prime n = n > 1 && all (\ x -> rem n x /= 0) xs
>   where xs = takeWhile (\ y -> y^2 <= n) primes

> primes :: [Integer]
> primes = 2 : filter prime [3..]
```

Useful logic notation

```
> infix 1 -->
>
> (-->) :: Bool -> Bool -> Bool
> p --> q = (not p) || q
>
> forall :: [a] -> (a -> Bool) -> Bool
> forall = flip all
```

1.

Redo exercises 2 and 3 of Workshop 1 by writing QuickCheck tests for these statements. See the end of Lecture 1 for how this can be done.

2.

Redo exercise 4 of Workshop 1 by replacing sets by lists, and testing the property for integer lists of the form `[1..n]`. You can use `subsequences :: [a] -> [[a]]` for the list of all subsequences of a given list.

```
*Lab1> subsequences [1..3]
[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

Use `length` for the size of a list.

```
*Lab1> length (subsequences [1..3])
8
*Lab1> length (subsequences [1..4])
16
*Lab1> length (subsequences [1..5])
32
```

Is the property hard to test? If you find that it is, can you given a reason why?

Give your thoughts on the following issue: when you perform the test for exercise 4, what are you testing actually? Are you checking a mathematical fact? Or are you testing whether `subsequences` satisfies a part of its specification? Or are you testing something else still?

3.

Redo exercise 5 of Workshop 1 by replacing sets by lists, and testing the property for integer lists of the form `[1..n]`.

Is the property hard to test? If you find that it is, can you given a reason why?

Again, give your thoughts on the following issue: when you perform the test for exercise 5, what are you testing actually? Are you checking a mathematical fact? Or are you testing whether `perms` satisfies a part of its specification? Or are you testing something else still?

4.

The natural number 13 has the property that it is prime and its reversal, the number 31, is also prime. Write a function that finds all primes < 10000 with this property.

To get you started, here is a function for finding the reversal of a natural number:

```
> reversal :: Integer -> Integer
> reversal = read . reverse . show
```

How would you test this function, by the way?

5.

The number 101 is a prime, and it is also the sum of five consecutive primes, namely 13 + 17 + 19 + 23 + 29. Find the smallest prime number that is a sum of 101 consecutive primes.

Do you have to test that your answer is correct? How could this be checked?

6.

Using Haskell to refute a conjecture. Write a Haskell function that can be used to refute the following conjecture. "If $p_1, \ldots, p_n$ is a list of consecutive primes starting from 2, then $(p_1 \times \cdots \times p_n) + 1$ is also prime." This can be refuted by means of a counterexample, so your Haskell program should generate counterexamples. What is the smallest counterexample?

7.

Implement and test the Luhn Algorithm

The Luhn algorithm is a formula for validating credit card numbers.

Give an implementation in Haskell. The type declaration should run:

```
luhn :: Integer -> Bool
```

This function should check whether an input number satisfies the Luhn formula.

Next, use `luhn` to write functions

```
isAmericanExpress, isMaster, isVisa :: Integer -> Bool
```

for checking whether an input number is a valid American Express Card, Master Card, or Visa Card number. Consult Wikipedia for the relevant properties.

Finally, design and implement a test for correctness of your implementation.

8.

Crime Scene Investigation

A group of five school children is caught in a crime. One of them has stolen something from some kid they all dislike. The headmistress has to find out who did it. She questions the children, and this is what they say:

Matthew: Carl didn't do it, and neither did I.

Peter It was Matthew or it was Jack.

Jack Matthew and Peter are both lying.

Arnold Matthew or Peter is speaking the truth, but not both.

Carl What Arnold says is not true.

Their class teacher now comes in. She says: three of these boys always tell the truth, and two always lie. You can assume that what the class teacher says is true. Use Haskell to write a function that computes who was the thief, and a function that computes which boys made honest declarations. Here are some definitions to get you started.

```
> data Boy = Matthew | Peter | Jack | Arnold | Carl
>            deriving (Eq,Show)
>
```

```
> boys = [Matthew, Peter, Jack, Arnold, Carl]
```

You should first define a function

```
  accuses :: Boy -> Boy -> Bool
```

for encoding whether a boy accuses another boy.

Next, define

```
  accusers :: Boy -> [Boy]
```

giving the list of accusers of each boy.

Finally, define

```
 guilty, honest :: [Boy]
```

to give the list of guilty boys, plus the list of boys who made honest (true) statements.

If the puzzle is well-designed, then `guilty` should give a singleton list.

Bonus

If this was all easy for you, you should next try some of the problems of Project Euler. Try problems 9, 10 and 49.

Submission deadline is Sunday evening, September 10th, at 6 pm.