Lab 2 Software Specification and Testing

```
> module Lab2 where
>
> import Data.List
> import Data.Char
> import System.Random
> import Test.QuickCheck
```

Useful logic notation

```
> infix 1 -->
>
> (-->) :: Bool -> Bool -> Bool
> p --> q = (not p) || q
```

Your programmer Red Curry has written the following function for generating lists of floating point numbers.

```
> probs :: Int -> IO [Float]
> probs 0 = return []
> probs n = do
>              p <- getStdRandom random
>              ps <- probs (n-1)
>              return (p:ps)
```

He claims that these numbers are random in the open interval $(0..1)$. Your task is to test whether this claim is correct, by counting the numbers in the quartiles

$(0..0.25), [0.25..0.5), [0.5..0.75), [0.75..1)$

and checking whether the proportions between these are as expected.

E.g., if you generate 10000 numbers, then roughly 2500 of them should be in each quartile.

Implement this test, and report on the test results.

Recognizing triangles

Write a program (in Haskell) that takes a triple of integer values as arguments and gives as output one of the following statements:

- `Not a triangle` (`Geen driehoek`) if the three numbers cannot occur as the lengths of the sides of triangle,
- `Equilateral` (`Gelijkzijdig`) if the three numbers are the lengths of the sides of an equilateral triangle,
- `Rectangular` (`Rechthoekig`) if the three numbers are the lengths of the sides of a rectangular triangle,
- `Isosceles` (`Gelijkbenig`) if the three numbers are the lengths of the sides of an isosceles (but not equilateral) triangle,
- `Other` (`Anders`) if the three numbers are the lengths of the sides of a triangle that is not equilateral, not rectangular, and not isosceles.

Here is a useful datatype definition:

```
> data Shape = NoTriangle | Equilateral
>            | Isosceles  | Rectangular | Other deriving (Eq,Show)
```

Now define a function `triangle :: Integer -> Integer -> Integer -> Shape` with the right properties.

You may wish to consult wikipedia. Indicate how you *tested* or *checked* the correctness of the program.

Deliverables: Haskell program, concise test report, indication of time spent.

Testing properties strength

Considering the following predicate on test properties:

```
> stronger, weaker :: [a] -> (a -> Bool) -> (a -> Bool) -> Bool
> stronger xs p q = forall xs (\ x -> p x --> q x)
> weaker   xs p q = stronger xs q p
```

a) Implement all properties from the Exercise 3 from Workshop 2 as Haskell functions of type `Int -> Bool`. Consider a small domain like $[(-10)..10]$.

b) Provide a descending strength list of all the implemented properties.

Recognizing Permutations

A permutation of a finite list is another finite list with the same elements, but possibly in a different order. For example, [3,2,1] is a permutation of [1,2,3], but [2,2,0] is not. Write a function

```
isPermutation :: Eq a => [a] -> [a] -> Bool
```

that returns `True` if its arguments are permutations of each other.

Next, define some testable properties for this function, and use a number of well-chosen lists to test `isPermutation`. You may assume that your input lists do not contain duplicates. What does this mean for your testing procedure?

Provide an ordered list of properties by strength using the weakear and stronger definitions.

Can you automate the test process? Use the techniques presented in this week's lecture. Also use QuickCheck.

Deliverables: Haskell program, concise test report, indication of time spent.

Recognizing and generating derangements

A derangement of the list `[0..n-1]` of natural numbers is a permutation π of the list with the property that for no x in the list $\pi(x) = x$. This is what you need if you prepare for *Sinterklaas* with a group of friends, where you want to avoid the situation that someone has to buy a surprise gift for him- or herself.

Give a Haskell implementation of a property `isDerangement` that checks whether one list is a derangement of another one.

Give a Haskell implementation of a function `deran` that generates a list of all derangements of the list `[0..n-1]`.

Note You may wish to use the `permutations` function from Data.List, or the `perms` function from workshop 1.

Next, define some testable properties for the `isDerangement` function, and use some well-chosen integer lists to test `isDerangement`.

Provide an ordered list of properties by strength using the weakear and stronger definitions.

Can you automate the test process?

Deliverables: Haskell program, concise test report, indication of time spent.

Implementing and testing ROT13 encoding

ROT13 is a single letter substitution cipher that is used in online forums for hiding spoilers.

See also www.rot13.com.

First, give a *specification* of ROT13.

Next, give a *simple implementation* of ROT13.

Finally, turn the specification into a *series of QuickCheck testable properties*, and use these to test your implementation.

Implementing and testing IBAN validation

The International Bank Account Number (IBAN) was designed to facility international money transfer, to uniquely identify bank accounts worldwide. It is described here, including a procedure for validating IBAN codes. Write a function

```
iban :: String -> Bool
```

that implements this validation procedure.

Next, test your implementation using some suitable list of examples.

Note It is not enough to test only with *correct* examples. You should invent a way to test with *incorrect* examples also.

Can you automate the test process?

Deliverables: Haskell program, concise test report, indication of time spent.

Bonus

Take your pick from [Project Euler].

Submission deadline is Sunday evening, September 17th, at 6 pm.