Lab 0 - Week 1. Haskell Basics & Monads

Aim

The aim of this assignment is to provide an introduction to coding with Haskell, using Monads and utilising QuickCheck to test Haskell Functions.

This assignment is not graded but you will receive feedback.

Prerequisites

Guidelines

- Please submit one Haskell file per exercise (only Haskell files will be admitted) per group (one set of answers per group).
- Name each file in the following format: **ExerciseX.hs** for exercises, where X is the exercise number eg. Exercise1.hs (note the capital), and the euler problems: EulerX.hs where X is the problem number. All your Haskell files need to have a capital first letter for the tests to pass.
- Add your answers in the form of comments in the respective exercise file
- Follow closely the naming conventions indicated by each exercise (some exercises go through automatic testing and it creates an overhead for the TAs if your files dont compile)
- **Please indicate the time spent on every exercise.**
- Codegrade: You will have to create a new group for every submission so make sure all your teammates are aware and have joined before the submission. We recommend you make the codegrade group in the beginning of each the assignment
- If you are using additional dependencies please indicate so in a comment on top of the file.

Imports

```
import Data.List
import Data.Char
import System.Random
import Test.QuickCheck
```

Useful Functions

```
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
```

Exercises

Exercise 1

Redo exercises 2 and 3 of Workshop 1 by writing QuickCheck tests for these statements.

Deliverables: Haskell program, indication of time spent.

Exercise 2

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.

Deliverables: Test, concise test report, indication of time spent

---

Exercise 3

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` if the three numbers cannot occur as the lengths of the sides of triangle,
- `Equilateral` if the three numbers are the lengths of the sides of an equilateral triangle,
- `Rectangular` if the three numbers are the lengths of the sides of a rectangular triangle,
- `Isosceles` if the three numbers are the lengths of the sides of an isosceles (but not equilateral) triangle,
- `Other` 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)
```

Use the declaration: `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.

Exercise 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. Follow this type declaration:

```
reversibleStream :: [Integer]
```

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

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

Implement **at least three** of the following properties and elaborate on each property's coverage. Can you think of any more properties? If so, briefly describe them.

1. Reversal correctness:
    a. This property ensures that the `reversal` function accurately produces the reversal of a number. If this property holds true, it implies that the reversal of a number is a reversible operation, as reversing it twice should yield the original number.
2. Prime reversibility
    a. This property ensures that the function correctly identifies prime numbers that are reversible. If this property holds true, it demonstrates that the combination of the prime-checking function and the reversal-checking function works as intended to identify reversible primes.
3. Prime Membership
    a. This property ensures that the `reversibleStream` function generates a list of prime numbers. If this property holds true for all generated numbers, it guarantees that the function is correctly identifying reversible prime numbers.

4. Reversible Prime Count
    a. This property helps ensure that the `reversibleStream` function is generating the correct number of reversible primes. By comparing the generated count with a pre-computed count, you can verify if the function's output matches expectations.
5. Reversal Symmetry
    a. This property confirms that the function's identification of reversible primes is bidirectional – if a prime number is in the list, its reversal should also be in the list. This property helps validate the correctness of the combination of prime checking and reversal checking.
6. Unique Values
    a. This property confirms that each element in the list is unique
7. Check Maximum Value
    a. This property checks whether the values in the list are less than the maximum value (10000)

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

Exercise 5

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:

```
rot13 :: [Char] -> [Char]
```

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

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

Exercise 6

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.

Follow this type declaration:

```
consecutive101Prime :: Integer
```

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

Deliverables: Haskell program, solution, answer to the questions, indication of time spent.

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. Make sure the functions have the following format, where X is the problem's number (eg. euler9):

```
eulerX :: Integer
```
