Lab 1 - Week 2-3. Pre/Post Conditions, Hoare Logic and Propositional Logic

---

Aim

Test your knowledge on properly testing functions with QuickCheck and applying the theory from the lectures.

Prerequisites

Submission Guidelines

- Submit one Haskell file per exercise (only Haskell files allowed).

- Name each file as `ExerciseX.hs` for exercises and `EulerX.hs` for Euler problems.

- Ensure that the respective module of each file has the same naming format (`module ExerciseX where`).

- Follow the exercise naming conventions closely. Some exercises go through automated testing, so it is important not to change the indicated declarations.

- Do not include any personally identifiable information in the submissions.

- If using additional dependencies, indicate so in a comment at the top of the file.

- Indicate the time spent on each exercise like so `Time Spent: X min`. Make sure it is **in minutes.**

- Codegrade: **For each assignment**, you need to create **a new group with all teammates**. Please note that once you submit, you cannot change the team structure, so be cautious.

Imports

```
import Data.List
import System.Random
import Test.QuickCheck
-- import Lecture1
-- import Lecture2
-- import Lecture3
```

Useful logic notation

```
infix 1 -->

(-->) :: Bool -> Bool -> Bool
p --> q = (not p) || q

forall :: [a] -> (a -> Bool) -> Bool
forall = flip all
```

Exercise 1

Write a Haskell function that finds the factorial of an integer input, the specification of the function should be as follows :

```
factorial :: Integer -> Integer
```

The function should be written by utilising recursion.

Create at least two props to test the factorial function. The prop should use a random number generator using QuickCheck Gen construct.

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

Exercise 2

Redo the following example from the Workshop:

> Prove by induction that if A is a finite set with $|A| = n$ then $|P(A)| = 2^n$.

and test the property for integer lists of the form `[1..n]`.

Give a Haskell implementation of the property `subsequence` for the list of all subsequences of a given list

```
subsequences :: [a] -> [[a]]
```

This should produce:

```
*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
```

Questions

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

2. Give your thoughts on the following issue: When you perform these tests, 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?

Deliverables: Haskell program, concise test report, answers to the questions, indication of time spent.

Exercise 3

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
```

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

2. Provide a descending strength list of all the implemented properties.

Deliverables: Implementation of properties, descending strength list of said implementation(figure out how to print them, and provide your answer in a comment too), an indication of time spent.

Exercise 4

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 where someone has to buy a surprise gift for themselves.

Answer the following:

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

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

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

```
deran:: Int -> [[Int]]
```

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

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

5. Can you automate the test process?

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

Note: Completing **Bonus 1** can help you complete this task

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

Exercise 5

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**. Make sure you comment on **how you**

converted natural languages into the specifications.

Here are some definitions to get you started. Describe the process you followed. Code without explanation will be deemed insufficient.

```
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.

Deliverables: Haskell program, explanation, indication of time spent.

Exercise 6

Week 3's lectures discusses the conversion of Boolean formulas (formulas of propositional logic) into CNF form. The lecture notes also give a definition of a Haskell datatype for formulas of

propositional logic, using lists for conjunctions and disjunctions. Your task is to write a Haskell program for converting formulas into CNF.

Use the following declaration:

```
cnf :: Form -> Form
```

Deliverables: conversion program with documentation, indication of time spent.

Exercise 7

Consider the following implementation of a function `sub::Form->Set Form` that finds all the sub-formulae of a given formula. Import Lecture3 for the following code.

```
import SetOrd

type Name = Int

data Form = Prop Name
| Neg  Form
| Cnj [Form]
| Dsj [Form]
| Impl Form Form
| Equiv Form Form
deriving (Eq,Ord)

sub :: Form -> Set Form
sub (Prop x) = Set [Prop x]
sub (Neg f) = unionSet (Set [Neg f]) (sub f)
sub f@(Cnj [f1,f2]) = unionSet ( unionSet (Set [f]) (sub f1))
(sub f2)
sub f@(Dsj [f1,f2]) = unionSet ( unionSet (Set [f]) (sub f1))
(sub f2)
sub f@(Impl f1 f2) = unionSet ( unionSet (Set [f]) (sub f1))
(sub f2)
```

```
sub f@(Equiv f1 f2) = unionSet ( unionSet (Set [f]) (sub f1))
(sub f2)
```

1. How can you prove that the sub implementation is correct? Test the implementation with two QuickCheck properties. (Hint: *Chapter 7.7 Haskell Road to Logic)*

2. Write a recursive implementation of the function such that nsub f computes the exact number of sub-formulae of the formula f. Test your implementation using QuickCheck.

```
nsub :: Form -> Int
```

Deliverables: for 1: answer to the question, quickCheck of two properties, indication of time spent; for 2: implementation of nsub, quickCheck properties, indication of time spent

Exercise 8

Using Haskell to refute a conjecture. Write a Haskell function that can be used to refute the following conjecture. "If p1,...,pn is a list of consecutive primes starting from 2, then (p1×…×pn)+1 is also prime." This can be refuted by means of a counterexample, so your Haskell program should generate counterexamples.

Follow this type declaration:

```
counterexamples ::  [([Integer], Integer)]
```

where the `[Integer]` is the list of consecutive primes and the `Integer` is their product + 1.

Deliverables: Haskell program, indication of time spent.

Exercise 9

Note: Potential exam question

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?

Can you automate the test process? Also, use QuickCheck.

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

Euler (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
```

You can also attempt to solve more problems of your choice from Project Euler in Haskell. Solving harder problems will account for more bonus points up to 1. Also, make sure you describe how you would test the functions you implement for the Euler exercise, if no description is provided, no points will be awarded.