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

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 System.Random
import Test.QuickCheck
-- import Lecture1
```

Exercises

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 exercise 4 of Workshop 1 and test 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
```

Questions

1. Is the property hard to test? If you find that it is, can you given a reason why?
2. 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?

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), indication of time spent.

Exercise 4

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.

Exercise 5

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 themselves.

Answer the following:

1. Give a Haskell implementation of a property `isDerangement :: Eq a => [a] -> [a] -> Bool` that checks whether one list is a derangement of another one.
2. Give a Haskell implementation of a function `deran:: Int -> [[Int]]` that generates a list of all derangements of the list `[0..n-1]`.
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.

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

Exercise 6

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

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

Deliverables: for 5.1: answer to the question, quickCheck of two properties, indication of time spent; for 5.2: implementation of nsub, quickCheck properties, indication of time spent

---

Exercise 7

In SAT solving, one common technique is resolution style theorem proving. For that, it is usual to represent a formula in CNF as a list of clauses, were a clause is a list of literals, and where a literal is represented as an integer, with negative sign indicating negation. Here are the appropriate type declarations.

```
type Clause  = [Int]
type Clauses = [Clause]
```

Clauses should be read disjunctively, and clause lists conjunctively. So 5 represents the atom p5, −5 represents the literal ¬p5, the clause [5,−6] represents p5∨¬p6, and the clause list [[4],[5,−6]] represents the formula p4∧(p5∨¬p6)

Write a program for converting formulas to clause form. You may assume that your formulas are already in CNF. Here is the appropriate declaration:

```
cnf2cls :: Form -> Clauses
```

If you combine your conversion function from an earlier exercise with `cnf2cls` you have a function that can convert any formula to clause form.

Use automated testing to check whether your translation is correct, employing some appropriate properties to check.

Deliverables: Conversion program, test generator, test properties, documentation of the automated testing process. Also, give an indication of time spent.

Exercise 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. 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 9

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.

Bonus

Solve three problems of your choice from Project Euler in Haskell. Solving harder problems will account for more bonus points up to 2. 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.
