# EX 5 - solution sheet

## Introduction

Please note that there can be other solutions than those listed in this document.

This is a literate Haskell file which is available as PDF, as well as literate Haskell source code (.lhs extension). If you are looking at the PDF version, you should download the .lhs file, which can be loaded into ghci or compiled with ghc. This document uses ***Markdown*** syntax, compiled to PDF using *Pandoc*.

Everything that is not enclosed in a code block like below is treated as text.

```
-- This is code
module Ex5 where
import Data.List
main :: IO ()
main = undefined
```

# 1 - Data.List

## *a* - Description and complexity

The function ***nub*** removes duplicates from a list, running in complexity $O(n^2)$, the actual maximum number of comparisons in worst case is $\frac{n(n-1)}{2}$, as we are removing at least 1 element in each recursive call, starting with n-1 comparisons.

***delete*** removes **all** occurences of a given element from a list. It runs in linear time $O(n)$, as it checks each element of the list once.

(`\\`) Is the list difference function, which returns a list of all elements in the first lists, which are not in the second. Fold itself is linear, but calls ***delete*** n times. Flip can be considered constant time. The complexity is: $O(n^2)$.

***union*** in a lists union, where the result is the first list + all elements from the second list which is not in the first. The `++` operator iterates through the first list, after that, appending the second list is a constant operation. In the end, the complexity is dominated by the list difference function. So we have quadratic $O(n^2)$.

***intersect*** Checks for each element in $l_1$, if it exists in $l_2$. If it does, it is added to the resulting list. As ***intersect*** uses elem, which is linear in worst case. Worst case comparisons: $O(mn)$

### *b* - nub using *filter*

Nothing too fancy, should have seen everything syntax-related in previous exercices

```
nubF :: Eq a => [a] -> [a]
nubF []     = []
nubF (x:xs) = x : nubF (filter (/= x) xs)
```

### *c* - *delete* using foldr

We want a function which will take a list and an element, and return either only the list, or the element added to the list.

```
deleteF :: Eq a => a -> [a] -> [a]
deleteF y xs = foldr check [] xs
  where
    check x ys
        | y == x = ys        -- y is from deleteF
        | otherwise = x : ys
```

### *d* - *delete* by filter or list comprehension

Yes to both.

With use of filter:

```
deleteFi :: Eq a => a -> [a] -> [a]
deleteFi y = filter (/=y)
```

As list comprehension:

```
deleteLc :: Eq a => a -> [a] -> [a]
deleteLc y xs = [x | x <- xs, x /= y]
```

### *e* - List difference using explicit recursion

We can still use delete, as in the original implementation.

```
difference :: Eq a => [a] -> [a] -> [a]
difference xs []     = xs
difference xs (y:ys) = difference (delete y xs) ys
```

### *f* - Intersection using filter

We use `elem` just as in the original implementation

```
intersectF :: Eq a => [a] -> [a] -> [a]
intersectF xs ys = filter (`elem` ys) xs
```

## 2 - foldTree

```
data Tree a = Nil | Node a (Tree a) (Tree a)

foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
foldTree _ z Nil = z
foldTree f z (Node a l r) = f a (foldTree f z l) (foldTree f z r)
```

### *a* - type of *foldTree*

The type of `foldTree` is quite similar to that of the original `fold`.

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
```

### *b* - Functions on trees

Note that the function `f` is given the result from folding left and right subtrees as parameters. This lets us define `height` as below.

```
height :: Tree a -> Int
height = foldTree f 0
  where
    f _ l r = 1 + max l r
```

And with a slight modification, we can get the full tree size.

```
size :: Tree t -> Int
size = foldTree f 0
  where
    f _ l r = 1 + l + r
```

We can also get the sum of a tree if all nodes are numbers.

```
treesum :: Num a => Tree a -> a
treesum = foldTree f 0
  where
    f a l r = a + l + r
```

Flatten can be done in a few different ways, depending on the order it is desired to get the nodes. These are known as *pre-order*, *in-order* and *post-order*.

```
flatten :: Tree a -> [a]
flatten = foldTree f []
  where
    f a l r = a : l ++ r     -- Pre-order
    -- f a l r = l ++ a : r     -- In-order
    -- f a l r = l ++ r ++ [a] -- Post-order
```

# 3 - Functors

Firstly, we want to show that with `fmap = (.)`, it still holds that `fmap id = id`. For this, we use the eta-converted version.

```
fmap id h x
= (id . h) x -- fmap = (.)
= id (h x)   -- definition of (.)
= h x        -- definition of id
```

Next, we want to show that `fmap (f . g) = fmap f . fmap g`, again, we use the eta-converted version.

```
fmap ((f . g) h) x
= ((f . g) . h) x       -- fmap = (.)
= (f . g) (h x)         -- definition of (.)
= f (g (h x))           -- definition of (.)
= f ((g . h) x)         -- definition of (.)
= (f . (g . h)) x       -- definition of (.)
= (fmap f (fmap g h)) x -- fmap = (.)
```

## 4 - List comprehensions

$$[(x, y)|x \leftarrow [1..n], \text{odd } x, y \leftarrow [1..n]] \tag{1}$$
$$[(x, y)|x \leftarrow [1..n], y \leftarrow [1..n], \text{odd } x] \tag{2}$$

Firstly, while the result of the two will be equal, the second potentially wastes alot of time assigning values to y's, where the first one only assigns values to y's if they are going to be used.

```
v1 :: Int -> [(Int, Int)]
v1 n = concat $ map (\x -> myZip x [1..n])
                  $ filter odd [1..n]
  where
    myZip :: a -> [b] -> [(a, b)]
    myZip _ []     = []
    myZip a (x:xs) = (a, x) : myZip a xs
```

Alternatively, we can aviod creating any helper functions.

```
v1' :: Int -> [(Int, Int)]
v1' n = concat $ map (\x -> zip (repeat x) [1..n])
                           $ filter odd [1..n]
```

If we for some reason wanted the program to waste a bunch of time zipping with values that will be discarded, as in list comprehension 2, we could do that.

```
v2 :: Int -> [(Int, Int)]
v2 n = filter (odd . fst)
        $ concat
        $ map (\x -> zip (repeat x) [1..n]) [1..n]
```

## 5 - Transpose without list comprehensions

```
transpose' :: [[a]] -> [[a]]
transpose' []               = []
transpose' ([] : xss)       = transpose' xss
transpose' ((x : xs) : xss) = (x : map head xss)
                            : transpose' (xs : map tail xss)
```

## 6 - remdups

[1,1,2,2,2,3,3,1] -> [1,2,3,1]

```
remdups :: Eq a => [a] -> [a]
remdups = foldr f []
  where
    f x [] = [x]
    f x (y:ys)
      | x == y = x : ys
      | otherwise = x : y : ys
```