TE 1 - 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.

```haskell
-- This is code
module PolyList where
import Data.List(foldr1)
main :: IO ()
main = undefined
```

1 - canonical

We can implement this using the functions `reverse` and `dropWhile`. Additionally, try to think about why this function does only makes sense if the input is a finite list (which is what allows the use of reverse).

```haskell
canonical :: (Num a,Eq a) => [a] -> [a]
canonical poly = reverse (dropWhile (==0) (reverse poly))
```

Alternatively, we can define canonical as a composition of these three functions using the dot operator.

```haskell
canonical' :: (Num a,Eq a) => [a] -> [a]
canonical' = reverse . dropWhile (==0) . reverse
```

2 - polynomial degree

Following the definition *degree of polynomial* from wikipedia. Since we can assume the polynomial to be in canonical form, the largest degree will always be the last, and hence the degree will be the length of the list *-1*. E.g. *[5, 2, 3] ==>* $5 + 2x + 3x^2$ Has a length of 3, but the degree is 2.

```haskell
deg :: [a] -> Int
deg poly = length poly - 1
```

3 - lead

Using the notation learned so far:

```
lead :: Num a => [a] -> a
lead []     = 0
lead [x]    = x
lead (_:xs) = lead xs
```

Alternatively, the function `last` gives the last element of a list (does not support empty lists)

```
lead' :: Num a => [a] -> a
lead' []   = 0
lead' poly = last poly
```

4 - neg

Again, we can define this recursively.

```
neg :: (Num a,Eq a) => [a] -> [a]
neg []     = []
neg (x:xs) = (-x) : neg xs
```

Or with some builtin-functions.

```
neg' :: (Num a,Eq a) => [a] -> [a]
neg' poly = map negate poly
```

When we dont really need to pattern match (only one case), we can avoid doing so. The following code will get the exact same parameters when called, as the previous i.e. negate will get the polynomial.

```
neg'' :: (Num a,Eq a) => [a] -> [a]
neg'' = map negate
```

5 - add

The function `zipWith` allows us to combine two lists to one, using any valid function, including *(+)*. Unfortunately, this will only work as long as the lists are of same length. Instead, we defined our own version of `zipWith`, `zipWithLossless`. Canonical is applied to the result, as mentioned in the exercise.

```
add :: (Num a,Eq a) => [a] -> [a] -> [a]
add poly1 poly2 = canonical (zipWithLl (+) poly1 poly2)
  where
    zipWithLl _ [] ys          = ys
    zipWithLl _ xs []          = xs
    zipWithLl f (x:xs) (y:ys) = f x y : zipWithLl f xs ys
```

6 - sub

Using `add` and `neg`, as suggested.

```
sub :: (Num a,Eq a) => [a] -> [a] -> [a]
sub poly1 poly2 = add poly1 $ neg poly2 -- add poly1 (neg poly2)
```

7 - addMany

We can use explicit recursion.

```
addMany :: (Num a,Eq a) => [[a]] -> [a]
addMany []           = []
addMany (poly:polys) = add poly $ addMany polys
```

Or use the function `foldr :: (a -> b -> b) -> b -> [a] -> b`, which is useful for reducing lists to single elements.

```
addMany' :: (Num a,Eq a) => [[a]] -> [a]
addMany' []           = []
addMany' (poly:polys) = foldr add poly polys -- foldr (add) (poly) (rest)
```

We can actually further reduce this. The following code will start by adding the first polynomial with the empty list, then the result from that with the next polynomial, and so on.

```
addMany'' :: (Num a,Eq a) => [[a]] -> [a]
addMany'' = foldr add []
```

```
addMany''' :: (Num a,Eq a) => [[a]] -> [a]
addMany''' = foldr1 add
```

8 - multconstant

Generally, if we want to modify every element of a list, we should consider `map`.

```
mulconstant :: (Num a,Eq a) => a -> [a] -> [a]
mulconstant a poly = map (*a) poly
```

Again, this can be done without writing poly explicitly

```
mulconstant' :: (Num a,Eq a) => a -> [a] -> [a]
mulconstant' a = map (*a)
```

9 - mulpower

Recall replicate and `fromElemCounts` from wednesday .

```
mulpower :: (Num a,Eq a) => Int -> [a] -> [a]
mulpower _ []    = []
mulpower i poly = replicate i 0 ++ poly
```

10 - diff & int

One way to implement `diff` would be by utilizing the fact that the original `zipWith` is right-lazy, to zip the polynomial with an infinite list.

```
diff :: (Num a,Eq a) => [a] -> [a]
diff poly = drop 1 $
            zipWith (*) poly idx
  where
    idx = iterate (+1) 0
```

Similarly, we can define `int`. We prepend 0, as seen in the example, as well as remove the original first element to avoid division by zero.

```
int :: (Fractional a,Eq a) => [a] -> [a]
int poly = 0 : zipWith (/) (drop 1 poly) (drop 1 idx)
  where
    idx = iterate (+1) 0
```

11 - mul

Again, we can make use of `zipWith`, to perform the operations described.

```
mul :: (Num a,Eq a) => [a] -> [a] -> [a]
mul as bs = addMany $
            zipWith mulpower idx [mulconstant a bs | a <- as]
  where
    idx = iterate (+1) 0
```

12 - eval

Following Horner's Rule, we can define `eval` recursively.

```
eval :: (Num a,Eq a) => [a] -> a -> a
eval [] _ = 0
eval (a:as) x = a + x * eval as x
```

13 - compose

We can define composition recursively by $p \circ q(x) = p(q(x)) = a_0 + q(x)p'(q(x))$, where p' is p without first element (a_0).

```
compose :: (Num a,Eq a) => [a] -> [a] -> [a]
compose [] _ = []
compose (a:as) bs = [a] `add` (bs `mul` compose as bs)
```

14 - polydiv

Rewrite of the pseudocode, to a recursive version. Smart use of *Haskell* functions could provide a cleaner implementation.

```
polydiv :: (Show a,Fractional a,Eq a) => [a] -> ([a], [a]) -> ([a], [a])
polydiv [] _ = ([],[]) -- d /= 0
polydiv d (zero, p) = polydiv' zero p
  where
    t r = canonical $ mulpower (deg r - deg d) [1] -- r > d - always
    polydiv' q []                = (q, [])
    polydiv' q r | deg r < deg d = (q, r)
                 | otherwise     = polydiv' (q `add` t r) (r `sub` mul (t r) d)
```

Another solution, with some reductions from previous.

```
polydiv'' :: (Show a,Fractional a,Eq a) => [a] -> ([a], [a]) -> ([a], [a])
polydiv'' d (q,r)
 | d /= [] && r /= [] && degree >= 0 =
    polydiv'' d (q `add` t,r `sub` (t `mul` d))
 | otherwise = (q,r)
 where
   degree = deg r - deg d
   t = mulpower degree [lead r / lead d]
```