Exercises for Lecture: Functional Programming
Exercise Sheet 4 (Combinators and Data Types)

Problem 1 (Reduction)

Reductions can be computed in a left associative way or a right associative way.
Implement both variants in Haskell (using a recursion over the given list). The types for the functions shall be as general as possible, i.e.:

```
reduceL :: (o -> i -> o) -> o -> [i] -> o
reduceR :: (i -> o -> o) -> o -> [i] -> o
```

`reduceL (+) 0 [1,2,3]` computes $\Big(\big((0+1)+2\big)+3\Big)$ and `reduceR (+) 0 [1,2,3]` computes $\Big(1+\big(2+(3+0)\big)\Big)$.

Note: Define both functions using a recursion along the list. Do not reverse the list (in particular for `reduceR`).

Problem 2 (List Combinators)

(a) Compute the sum of the elements and the length of a list in a single pass over the list using `foldl` and return the arithmetic mean of the list elements.
Note: the accumulator can be of an arbitrary type, e.g., a pair of values.

(b) Write a function `makeUpper :: String -> (Int, String)` which converts all lower case letters in the input into upper case letters and counts how many letters have been changed.

Note: You can use `isLower :: Char -> Bool` and `toUpper :: Char -> Char` from module `Data.Char`.

(c) Let a polynomial be given by the list of its cofficients in ascending order of the powers of the unknown, see also below. Define `evalPoly :: Num a => [a] -> a -> a` which evaluates a polynomial at a given value for the unknown using Horner's method. Use `foldr` to implement Horner's method.

`evalPoly [3,2,0,5] 10` should yield 5023.

Note: Horner's method factors out common powers of the unknown and applies one multiplication and one addition per coefficient. In the example, the polynomial $3+2\cdot x+0\cdot x^2+5\cdot x^3$ is evaluated as $3+x\cdot\Big(2+x\cdot\big(0+x\cdot(5)\big)\Big)$, i.e., for $x=10$, the expression computed using `foldr` is `3+10*(2+10*(0+10*(5+10*(0))))`.

(d) Given the same representation of polynomials as in the previous subexercise (c), write a function `derivePoly :: Num a => [a] -> [a]` that computes the derivative of the polynomial represented by its coefficient list. Use `zipWith` in your implementation.

`derivePoly [3,2,0,5]` should yield `[2,0,15]`.

Note: If you plan to use list enumerations in your implementation you also need the typeclass `Enum`, i.e., `derivePoly :: (Num a, Enum a) => [a] -> [a]`.

Problem 3 (Polynomials as a Data Type)

A polynomial in one unknown can be represented by the list of its coefficients, e.g., the polynomial $a_0 + a_1 \cdot X + a_2 \cdot X^2$ can be represented by the list $[a_0, a_1, a_2]$. Any numerical type is allowed as type for the coefficients.

Define

(a) a data type `Polynomial a` for polynomials with coefficients of type `a` which represents the polynomials internally by the list of coefficients. Also define a function `eval` to evaluate polynomials (similar to exercise 2(c)).

(b) a function to add two polynomials,

(c) a function to negate a polynomial,

(d) a function to multiply two polynomials.

Hint: Think about how to compute the coefficients of the product polynomial in the required order if you want to compute the coefficients of the product directly. Alternatively, use recursion to define multiplication and use the addition function you have defined already.

(e) a function to create a (constant) polynomial from a number,

(f) a (polymorphic) constant `x` representing the polynomial X,

(g) an instance of type class `Num a` for the data type with implementations for the functions

```
(+)         :: a -> a -> a
negate      :: a -> a
(*)         :: a -> a -> a
fromInteger :: Integer -> a
```

(functions `abs` and `signum` cannot be defined in a meaningful way); `(-)` works automatically when `negate` is defined.

Test your implementation in GHCi using `(3*x-x*(5-x))*(x*x-x)`. The result should have the coefficient list `[0,0,2,-3,1]`.

Due date: Tuesday, May 16, 2023 at 16:00

Upload your Haskell source codes (`*.hs`) to the `Submissions` folder for this exercise on Stud.IP.