Functional and Logic Programming Fall 2017

S03 : Haskell (Lists and lexical analysis)

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

Exercise 1

insert'

```
insert' :: Ord a => a -> [a] -> [a]
insert' elt [] = [elt]
insert' elt (x:xs)
  | elt <= x = (elt:x:xs)
  | otherwise = x : (insert' elt xs)
```

insertionSort

```
insertionSort :: Ord a => [a] -> [a]
insertionSort [] = []
insertionSort (x:xs) = insert' x (insertionSort xs)
```

takeWhile'

```
takeWhile' :: (a -> Bool) -> [a] -> [a]
```

zipWith'

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = (f x y) : (zipWith' f xs ys)
```

intersect'

```
intersect',intersect'',intersect''2,intersect''' :: Eq a => [a] -> [a] -> [a]
intersect' _ [] = []
intersect' [] _ = []
intersect' (x:xs) ys
  | elem x ys = x : intersect' xs ys
  | otherwise = intersect' xs ys

intersect'' xs ys = inner elem xs ys where
  inner :: (a -> [a] -> Bool) -> [a] -> [a] -> [a]
  inner _ _ [] = []
  inner _ [] _ = []
  inner f (x:xs) ys = if (f x ys)
    then x : (inner f xs ys)
    else (inner f xs ys)
```

```
intersect''2 xs ys = filter (\x -> elem x ys) xs

intersect''' xs ys = [x | x <- xs, y <- ys, x == y]
```

divisorList

```
divisorList,divisorList',divisorList''  :: Int -> [Int]

divisorList v
  | v < 1 = []
  | otherwise = inner 1 where
    inner :: Int -> [Int]
    inner n
      | n == v = []
      | otherwise = if v `rem` n == 0
        then n : (inner (n+1))
        else inner (n+1)

divisorList' v
  | v < 1 = []
  | otherwise = filter (\x -> v `rem` x == 0) [1..(v-1)]

divisorList'' v
  | v < 1 = []
  | otherwise = inner v where
    inner :: Int -> [Int]
    inner v = [ n | n <- [1..(v-1)], v `mod` n == 0]
```

perfectNumber

```
perfectNumber :: Int -> Bool
perfectNumber n = sum (divisorList n) == n
```

perfectNumbers

```
perfectNumbers :: Int -> [Int]
perfectNumbers n = take n perfectNumbersList where
  perfectNumbersList = filter perfectNumber [1..]
```

Exercise 2 : calculatePolynomial

```
type Polynom = [(Double,Int)]

-- PRE : polynom is not empty
calculatePolynomial,calculatePolynomial',calculatePolynomial''  :: Polynom -> Double -> Double

calculatePolynomial ((c,d):[]) x = computeCD c d x
calculatePolynomial ((c,d):cds) x = (computeCD c d x) + calculatePolynomial cds x

computeCD :: Double -> Int -> Double -> Double
computeCD c d x = c * (x ^ d)

calculatePolynomial' poly x = sum (map (\(c,d) -> c * (x ^ d) ) poly)

calculatePolynomial'' poly x = sum [v | (c,d) <- poly, let v = c * (x ^ d)]
```