Functional Programming	WS 2022	LVA 703025
Exercise Sheet 11, 10 points		Deadline: Wednesday, January 11, 2023, 6am

- Mark your completed exercises in the OLAT course of the PS.
- Start from `template_11.hs` provided on the proseminar page.
- Your .hs-file should be compilable with `ghci` and be uploaded in OLAT.

Exercise 1 *Evaluation Strategies and Kinds of Recursion* — 4 p.

1. Given the four functions:

```haskell
double x = x * 2
square x = x * x
add2times x y = x + double y
func x y = square x + add2times y x
```

   Evaluate each of the following expressions step-by-step under the three evaluation strategies call-by-value, call-by-name, and call-by-need. (3 points)

   (a) `add2times (5+2) 8`

   (b) `double (square 5)`

   (c) `func (2+2) 4`

2. Implement two variants of a function that takes a string and produces an upper case version of it: `stringToUpperTail` using tail recursion and `stringToUpperGuarded` using guarded recursion. For example `stringToUpperTail "Hello" = stringToUpperGuarded "Hello" = "HELLO"`. (1 point)

Exercise 2 *Kinds of Recursion, Seq* — 3 p.

1. Complete the table below with which type(s) of recursion each of the following functions use:

   (a)
```haskell
pow2 0 = 1
pow2 n = 2 * pow2 (n-1)
```

   (b)
```haskell
factAux r i n
  | i <= n = factAux (i * r) (i + 1) n
  | otherwise = r
factorial = factAux 1 1
```

   (c)
```haskell
init [x] =  []
init (x:xs) =  x : init xs
```

   (d)
```haskell
binom n 0 = 1
binom n k
  | n == k = 1
  | otherwise = binom (n - 1) k + binom (n - 1) (k - 1)
```

   (e)
```haskell
negList [] = []
negList (x : xs) = if x > 0 then negList (-x : xs) else x : negList xs
```

   The first row of the table has been completed for you. (1.5 points)

Function	Recursion	Linear recursion	Tail recursion	Guarded recursion
`pow2`	✓	✓	×	×
`factAux`				
`factorial`				
`init`				
`binom`				
`negList`				

2. Consider again the provided functions of exercise 2 task 1. For which of them would it make sense to enforce strict evaluation via `seq` or bang-patterns? Provide a modified implementation for at least one of these functions. (1.5 points)

Exercise 3 *Laziness, Modularity, Infinite Lists* **3 p.**

1. Implement a recursive function `applyIndefinitely` that, given a function `f` and an initial value `x`, generates the infinite list of repeated applications of `f` to `x`, that is

   ```
   [x, f x, f (f x), f (f (f x)), ...]
   ```

   Indicate which kind of recursion you are using. (0.5 points)

2. Implement a recursive function `takeUntil` that, given a predicate `p` and a list `xs`, yields the initial segment of `xs` up to, and including, the first element that satisfies `p`. (0.5 points)

   Example: `takeUntil (> 5) [1..10] = [1,2,3,4,5,6]`

3. Implement the function `what :: ([a], [a]) -> ([a], [a])` such that the following implementation of `rev :: [a] -> [a]` yields the reverse of a given list:

   ```
   rev = fst . head . dropWhile (not . null . snd) . applyIndefinitely what . (,) []
   ```

   Moreover, explain what happens in the respective parts of `rev` that are plugged together by function composition (the infix operator "`.`" in Haskell). (1 point)

4. Just by composing existing functions and without explicit recursion, implement a function `prefixes` that yields the list of all prefixes of a given list. (1 point)

   Make sure that your function also works on infinite lists.

   Example: `prefixes [1..] = [[],[1],[1,2],[1,2,3], ...]`

   Hint: `applyIndefinitely`, `what`, `rev`, and `takeUntil` from above might be useful.