universität innsbruck

Functional Programming	WS 2024/2025	LVA 703025
Exercise Sheet 5, 10 points		Deadline: Tuesday, November 12, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You should use a template .hs file that is provided on the proseminar page.
- Upload your modified `Template_05.hs` file in OLAT.
- Do not change the first lines of `Template_05.hs`, in particular do not add any `import` instructions.
- Your .hs file must be compilable with ghci.

Exercise 1 *Expressions, Guarded Equations, Locally Defined Functions* **5 p.**

1. We consider a list of pairs of type `[(a,b)]` and think of the following bidirectional lookup problem.

   Given `x :: a` we would like to find the first pair of shape `(x,y)` in the list and return `y`. Alternatively, given `y :: b` we would like to find the first pair of shape `(x,y)` in the list and return `x`. In both cases, it might also happen that no such pair exists, which should also be reported.

   Implement two Haskell functions that both implement the bidirectional lookup *in a single function definition without local function definitions* and without using predefined functions such as `lookup`. In particular, in both Haskell functions you should not implement two separate lookup functions, e.g., one for `x :: a` and one for `y :: b`, but somehow handle both directions in one function definition.

   (a) In the first Haskell function `biLookupIte` you should use `if .. then .. else ..`, but are not allowed to use guarded equations. (1.5 points)

   (b) In the second Haskell function `biLookupGuarded` you should use guarded equations, but are not allowed to use if-then-else or any user-defined `ite` function. (1.5 points)

   You will have to adjust the type of `biLookupGuarded` and `biLookupIte` in the template. Try to be as general as possible.

   Examples (where `biLookup` may be either `biLookupGuarded` or `biLookupIte`):
   ```
   favoriteNumbers = [("Felix", 45), ("Grace", 25), ("Hans", 57), ("Ivy", 25)]
   biLookup (Left "Grace") favoriteNumbers == Just (Right 25)
   biLookup (Right 57) favoriteNumbers == Just (Left "Hans")
   biLookup (Right 25) favoriteNumbers == Just (Left "Grace")
   biLookup (Left "Bob") favoriteNumbers == Nothing
   ```

2. We consider the following definition of large numbers, where it is assumed that the input is a natural number, i.e., `n >= 0`.
   ```
   large :: Integer -> Integer
   large n
     | n <= 2 = 3 * n - 5
     | otherwise = 2 * large (n - 3) + 5 * large (n - 1) + 7
   ```
   For instance, `large 30 = 32938956746479142061`, `large 40 = 37520848028199450311964405 3`, and the computation of `large 60` is not obtained in reasonable time.

   In order to speed up the computation process, we like to ask you to implement a function `largeFast` to compute `large n` in a bottom up way using the following idea: given `large i`, `large (i+1)`, `large (i+2)`, it is easy to compute the upcoming three values if you would replace `i` by `i+1`. Hence, you just have to start with `i = 0` and then increase `i` until you reach `n`.

Your implementation should be able to compute values of `largeFast 1000` within a fraction of a second. The program should be designed in a way that you only define `largeFast` globally, and all auxiliary functions are defined locally. (2 points)

Exercise 2 *Factorial and Eulers Number* **5 p.**

In this exercise we will consider the factorial function as defined on slide 13 of lecture 5. Using factorials, it is possible to calculate the Euler number $e \approx 2.718$ with the following infinite sum, called the Taylor series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

When using a finite sum, we can define the approximation of e, where e_n with $n \in \mathbb{N}$ represents the approximation with upper bound n, recursively as:

$$e_n = \begin{cases} 1 & \text{if } n = 0 \\ \frac{1}{n!} + e_{n-1} & \text{otherwise} \end{cases}$$

1. Write a function `euler :: Integer -> Double` that takes a number `n` as input and computes e_n. For negative inputs, throw an error with an appropriate error message.

   Hint: The function `fromIntegral` transforms an `Integer` to a `Double`. (1 point)

2. In the template you can find a function `factorialListNaive :: Integer -> [Integer]`, that given as input `n`, returns the list `[factorial n, ..., factorial 0]`. It is not very efficient, as no intermediate values are reused. Implement the function `factorialList :: Integer -> [Integer]` that has the same result as `factorialListNaive`, but does not compute any intermediate values twice. For negative inputs, throw an error with an appropriate error message. (1.5 points)

3. Use the function `factorialList` (or `factorialListNaive` if you did not implement it), and complete the function `eulerListAux :: [Integer] -> Double` which takes as input the list of factorials and computes the Euler number using only the factorials from the list.

   Hint: Consider in which order the factorials appear in the list. (1 point)

4. Mathematically, the Taylor series converges to e, but reaches it only for $\lim_{n \to \infty}$. However, due to the finite precision of the `Double` type, doing this computation in Haskell, you will always find that at some point $e_{n+1} == e_n$. Write a function `eulerConvergence :: [Double]` that outputs the sequence of numbers $[e_0, ..., e_n]$, where n is the smallest number such that $e_n == e_{n+1}$. Do **not** figure out n manually and hard code this in the function. (1.5 points)