universität innsbruck

Functional Programming | WS 2022 | LVA 703025

Exercise Sheet 5, 10 points | Deadline: Wednesday, November 9, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_05.hs` provided on the proseminar page.
- Upload your modified .hs file in OLAT.
- Your .hs file should be compilable with `ghci`.
- Try to define auxiliary functions within a `where` or `let ... in` construct.

Exercise 1 *Recursion on Lists* **4 p.**

1. The Haskell function `lookup k xs` of type `lookup :: Eq a => a -> [(a, b)] -> Maybe b` takes a key `k` and a list of key-value-pairs `xs :: [(a,b)]`. If it finds a pair `(k', v)` where `k' == k` it returns `Just v`, otherwise `Nothing` is returned.

   Implement a Haskell function `bidirectionalLookup` that takes a key `k :: Either a b`, and a list of pairs of type `[(a,b)]`. For keys of shape `Left l`, perform a lookup on the left half of the pairs, and for keys `Right r` on the right half of the pairs. In both cases, return the other half of the first matching pair. If no match is found, the function should return `Nothing`. (2 points)

   Examples:
   ```
   namesAges = [("Felix", 45), ("Grace", 25), ("Hans", 57), ("Ivy", 25)]
   bidirectionalLookup (Left "Grace") namesAges == Just (Right 25)
   bidirectionalLookup (Right 57) namesAges == Just (Left "Hans")
   bidirectionalLookup (Right 25) namesAges == Just (Left "Grace")
   bidirectionalLookup (Left "Bob") namesAges == Nothing
   ```

2. Implement a Haskell function `lengthSumMax :: (Num a, Ord a) => [a] -> (Int, a, a)` that, given a list of non-negative numbers, computes its length, the sum of all its elements and the maximum of all its elements and returns those three values as a triple. (2 points)
   Remark: Find a solution without using `length`, `sum`, and `maximum`.

   Examples:
   ```
   (case lengthSumMax [] of (l,s,_) -> (l,s)) == (0,0)
   lengthSumMax [0,1,0,2,0] == (5,3,2)
   ```

Exercise 2 *Recursion on Numbers* **3 p.**

1. Implement a function `slice :: Int -> Int -> [a] -> [a]`, where `slice n m xs` returns the elements of `xs` starting at index `n` and ending at index `m` (both inclusive). Make sure you find a reasonable treatment for edge cases, i.e. indices that are negative, or larger than the list length. (1 point)
   Remark: in your solution, do **not** use `take` or `drop`.
   Examples:
   ```
   slice 1 1 [0, 1, 2] == [1]
   slice 2 1 [0, 1, 2] == []
   slice 1 3 [0, 1, 2, 3, 4] == [1, 2, 3]
   slice 1 3 [0, 1, 2] == [1, 2]
   ```

2. Implement a function `dropEveryNth :: Int -> [a] -> [a]` which takes a list and eliminates every `n`-th element. For `n <= 0`, return the original list. (2 points)
   Remark: once again, find a solution that does **not** use `take` or `drop`.
   Examples:
   ```
   dropEveryNth 3 [1] == [1]
   dropEveryNth 3 [1, 2, 3, 4, 5, 6, 7] == [1, 2, 4, 5, 7]
   dropEveryNth 1 [1, 2] = []
   ```

Exercise 3 *Sequences and Series* — 3 p.

1. The Collatz conjecture is a famous unsolved problem in mathematics. It states that the sequence

$$a_0 = n, \quad a_{i+1} = \begin{cases} \frac{a_i}{2} & \text{if } a_i \text{ is even,} \\ 3a_i + 1 & \text{if } a_i \text{ is odd.} \end{cases}$$

   eventually reaches the cycle $4 \to 2 \to 1 \to 4 \to \dots$. As of 2020, all starting values up to 2^{68} have been tested and do reach the cycle. Implement a function `collatz :: Integer -> Integer` that counts the number of steps it takes for the input to reach 1 for the first time. (1 point)
   Remark: Note that the Haskell Prelude defines functions `even, odd :: Integer -> Bool`. Additionally, `(/)` is not defined for `Int` and `Integer`, use `(div)` instead (i.e. `div x y` or ``x `div` y``).
   Examples:
   ```
   collatz 1 == 0
   collatz 3 == 7
   collatz 16 == 4
   ```

2. The Mercator series is an infinite series to calculate the natural logarithm $\ln(1+x)$ for $-1 < x \leq 1$. The $n-$th partial sum y_n of the series for some value of x can be calculated recursively by

$$y_n = \begin{cases} x & \text{if } n = 1, \\ y_{n-1} + \frac{(-1)^{n+1}x^n}{n} & \text{if } n > 1. \end{cases}$$

   Mathematically, this sequence converges to $\ln(1+x)$ but never actually reaches it (aside from for $x = 0$), giving successively better and better approximations. However, due to the finite precision of the `Double` type, when doing this computation in Haskell, you will always find that at some point $y_{n+1} == y_n$[1].

   Your task is to write a function `mercator :: Double -> (Double, Integer)` that outputs a tuple $(y_n,\ n)$, where n is the smallest number such that $y_{n+1} == y_n$. For values $x \leq -1$ and $x > 1$, an error should be raised (use `error`). (2 points)
   Hint: you might need to convert between numbers of the two types `Double` and `Integer`. To this end you can use `fromInteger :: Num a => Integer -> a` or `round :: Double -> Integer`.
   Examples:
   ```
   mercator 0 == (0.0,1)
   mercator 0.12 == (0.1133286853070032,17)
   mercator (-3) -- *** Exception: ....
   ```

[1] Note that for values approaching 1, this series converges *very* slowly. If your function takes too long to converge, press `CTRL + C` to stop program execution.