universität innsbruck

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 5, 10 points		Deadline: Wednesday, November 10, 2021, 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 .hs-file(s) of Exercises 1 and 2 in OLAT.
- Your .hs-file(s) should be compilable with ghci.

Exercise 1 *Lists and Tuples* **5 p.**

1. Implement a Haskell function `mergeLists` that takes two lists and combines them into a list of pairs. If the lists do not have the same length, the rest of the longer list is ignored. Make sure that the type of your function is as general as possible. (1 point)

   Example: `mergeLists [1,2,3,4] ['a','b','c'] == [(1,'a'),(2,'b'),(3,'c')]`

2. Implement a Haskell function `calculateAge :: (Int, Int, Int) -> Int` that, given a birthday as a triple of day, month and year, computes the current age in years (as of November 10, 2021). (1 point)

   Example: `calculateAge (10, 11, 2021) == 0`

3. Implement a Haskell function

   ```
   convertDatesToAges :: [(String, (Int, Int, Int))] ->  [(String, Int)]
   ```

   that converts a list of names and birthdays into a list of names and ages. That is, it takes a list of pairs with names and dates and computes a list of pairs with names and ages. (1 point)

4. Implement a Haskell function `getOtherPairValue` that takes a pair of name and age as well as a second argument that can be either a name *or* an age and returns the part of the input that wasn't the second argument. If the second argument does not match either element of the input-pair the function should return nothing. (2 points)

   Example: When invoking `getOtherPairValue` on the pair `("Hans", 45)` together with the second argument `45` (wrapped in an appropriate type), the result should be `"Hans"` (wrapped in an appropriate type).

Hint: You might find the `ite` (if-then-else) function of sheet 3 useful for some of the above exercises.

```
ite :: Bool -> a -> a -> a
ite True x y = x
ite False x y = y
```

Solution 1

1.
   ```
   mergeLists :: [a] -> [b] -> [(a, b)]
   mergeLists (x : xs) (y : ys) = (x, y) : mergeLists xs ys
   mergeLists _ _ = []
   ```

2.
   ```
   calculateAge :: (Int, Int, Int) -> Int
   calculateAge (d, m, y) = ite (m < 11 || m <= 11 && d <= 10) (2021 - y) (2020 - y)
   ```

3.
   ```
   convertDatesToAges :: [(String, (Int, Int, Int))] ->  [(String, Int)]
   convertDatesToAges ((x, y) : xs) = (x, calculateAge y) : convertDatesToAges xs
   convertDatesToAges [] = []
   ```

4.
   ```
   getOtherPairValue :: (String, Int) -> Either String Int -> Maybe (Either String Int)
   getOtherPairValue (x, y) (Right r) = ite (y == r) (Just (Left x)) Nothing
   getOtherPairValue (x, y) (Left l) = ite (x == l) (Just (Right y)) Nothing
   ```

Exercise 2 *Equational Reasoning, Lists, and Tuples* **5 p.**

Consider the following Haskell functions

```
addPair (x, y)    =  x + y
addList []        =  []
addList (x : xs) =  addPair x : addList xs
```

1. Write down the types of the functions above and explain how you were able to derive them. (1 point)
2. Evaluate the result of the following expression step-wise (that is, only using one of the above equations at a time) by hand. (2 points)

   `addList ((1,2):(2,1):[])`

   Remark: For clarification here is an example of another step-wise evaluation (this process is called *equational reasoning*; the definition of `lastElem` can be found on page 3 of the slides of week 4):

   ```
   lastElem (Cons 1 (Cons 2 Empty)) = lastElem (Cons 2 Empty)
                                    = 2
   ```

3. Implement a Haskell function `fstList :: [(a, b)] -> [a]` that collects all the first elements of a list of pairs. (1 point)

   Examples:
   ```
   fstList [(1,'a'),(2,'b'),(3,'c')] == [1,2,3]
   fstList [("hello","world")] == ["hello"]
   fstList [] == []
   ```

4. 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. (1 point)

   Examples:
   ```
   lengthSumMax [] == (0,0,0)
   lengthSumMax [0,1,0,2,0] == (5,3,2)
   ```

Solution 2

1. - The type of `addPair` is `Num a => (a, a) -> a`. Since addition `(+)` is used on `x` and `y` both have to be of the same type `a` that is an instance of `Num`. Moreover, the input of `addPair` is a pair.
   - The type of `addList` is `Num a => [(a, a)] -> [a]`. Since `addList` is defined by pattern-matching on lists on its input, its input has to be a list. Moreover, since to each element of the input list the function `addPair` is applied, the list-elements have to be pairs of type `(a, a)` where `Num a`.

2.
   ```
   addList ((1,2):(2,1):[]) = addPair (1,2) : addList ((2,1):[])
                            = (1 + 2) : addList ((2,1):[])
                            = 3 : addList ((2,1):[])
                            = 3 : addPair (2,1) : addList []
                            = 3 : (2 + 1) : addList []
                            = 3 : 3 : addList []
                            = 3 : 3 : []
   ```

3.
   ```haskell
   fstList :: [(a, b)] -> [a]
   fstList ((x, y) : xs) = x : fstList xs
   fstList [] = []
   ```

4.
   ```haskell
   lengthSumMax :: (Num a, Ord a) => [a] -> (Int, a, a)
   lengthSumMax [] = (0, 0, 0)
   lengthSumMax (x : xs) = lsmHelper x (lengthSumMax xs)

   lsmHelper x (l, s, m) = (l + 1, s + x, max m x)
   ```