universität innsbruck

Functional Programming	WS 2024/2025	LVA 703025
Exercise Sheet 4, 10 points		Deadline: Tuesday, November 05, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs file that is provided on the proseminar page.
- Upload your modified `Template_04.hs` file in OLAT.
- Your .hs file must be compilable with ghci.

Exercise 1 *Lists, Maybe and Either* **5 p.**

1. Recall the function `findY` from slide 19 of lecture 4. Define a function `find`, that is similar to `findY`, but allows to search for arbitrary keys. Additionally, instead of throwing an exception if the key is not found, we want to use the type `Either String b`. Then we can choose the return type `String` for an error message or `b` if the key was found. Particularly, the following identities should hold: (2 points)

   ```
   find 'c' [('a', "foo"), ('b', "bar")] = Left "cannot find 'c'"
   find 'c' [('c', "test"), ('d', "bar")] = Right "test"
   ```

   Hint: The auxiliary function `ite :: Bool -> a -> a -> a` might be useful.

2. Define a function `polymorphicFind`, that works on pairs of type `(a, b)` instead of `(Char, b)`. Additionally, instead of returning an error string if the key is not found, we want to use the datatype `Maybe b`. Particularly, the following identities should hold: (1 point)

   ```
   polymorphicFind 'c' [('a', "foo"), ('b', "bar")] = Nothing
   polymorphicFind 4 [(1, 'a'), (2, 'b'), (3, 'c')] = Nothing
   polymorphicFind 'c' [('c', "foobar"), ('d', "bar")] = Just "foobar"
   polymorphicFind 4 [(1, 'a'), (4, 'd'), (3, 'c')] = Just 'd'
   ```

   Hint: You might need to slightly change the type of `polymorphicFind` in the template.

3. Define a function `suffixes` that computes the list of all suffixes of a list. Particularly, the following identities should hold (note that `String` is the same as `[Char]`): (1 point)

   ```
   suffixes [1, 2, 3] = [[1,2,3], [2,3], [3], []]
   suffixes "hello" = ["hello", "ello", "llo", "lo", "o", ""]
   ```

4. Define a function `removeLast`, that removes the last element of a list. Particularly, the following identities should hold: (1 point)

   ```
   removeLast [1,2,3] = [1,2]
   removeLast "hello world" = "hello worl"
   ```

Exercise 2 *Polymorphism* **2 p.**

1. Write a polymorphic function `threeEqual` taking three arguments of the same, arbitrary type, which returns `True` if and only if all of them are equal. Also write down the type signature. (1 point)

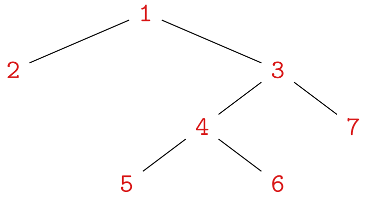
2. Write two different functions `foo` and `bar` of type `a -> a -> a`. Here, different means that for some input values `x` and `y` the result of `foo x y` is different from the result of `bar x y`. (1 point)

Exercise 3 *Polymorphism and Trees* **3 p.**

In this exercise we will consider a type for representing binary trees. To this end we consider the datatype `Tree` defined as

```
data Tree a = Node a (Tree a) (Tree a) | Leaf a
```

For instance, `exampleTree` represents the following tree:



```
exampleTree = Node 1 (Leaf 2) (Node 3 (Node 4 (Leaf 5) (Leaf 6)) (Leaf 7))
```

1. Write a function `height :: Tree a -> Integer` that calculates the height of a binary tree. The height is the number of edges on the longest path between the root and a leaf. (1 point)
   Hint: the Haskell function `max :: Ord a => a -> a -> a` *might be useful.*

   Examples:
   ```
   height (Leaf 'a') == 0
   height exampleTree == 3
   ```

2. Write a function `flatten :: Tree a -> [a]` which takes a tree as an argument and returns a list containing exactly the elements in the tree from left to right. In particular, each node element should appear in the list after the elements in its left subtree and before the elements in its right subtree. (1 point)
   Hint: `(++) :: [a] -> [a] -> [a]` *is Haskell's predefined append-function for lists.*

   Examples:
   ```
   flatten (Node 1 (Leaf 2) (Leaf 2)) == [2,1,2]
   flatten exampleTree == [2,1,5,4,6,3,7]
   ```

3. A binary tree `t` is said to be a binary search tree if `flatten t` is a list whose elements appear in strictly increasing order. Write a function `isSearchTree:: Ord a => Tree a -> Bool` that takes a tree as an argument and returns `True` if and only if the tree is a binary search tree. (1 point)
   Hint: you may assume that `flatten` *is available even if you did not solve question 2. It might be useful to define an auxiliary function* `isStrictlySorted :: Ord a => [a] -> Bool` *to determine whether the elements in a list are strictly increasing.*

   Examples:
   ```
   isSearchTree (Leaf "hello") == True
   isSearchTree exampleTree == False
   isSearchTree (Node 3 (Leaf 1) (Node 6 (Leaf 4) (Leaf 11))) == True
   ```