universität innsbruck

Functional Programming	SS 2020	LVA 703025
Exercise Sheet 6, 10 points		Deadline: Monday, April 27, 2020, 10am

- Watch the lecture of week 6 and study part 4 of the slides.[1].
- Please write all the Haskell code into a single .hs-file and upload it in OLAT.
- Exercise 6.3 can be added as a comment to the .hs-file.
- You can use the template .hs-file that is provided on the proseminar page[2].
- Your .hs-file should be compilable with ghci.
- Don't forget to mark your completed exercises in OLAT.

Exercise 6.1 *Lists* 3 p.

1. Write a function `sh` which returns the second largest number in a list of `Integer`s. Return an error if the list has less than two elements. In this exercise, if the largest number appears twice, it is also the second largest number in the list. Examples:

```
sh [7,3,2,5,6] = 6
sh [0] = error "Not enough elements in list"
sh [1,1,1] = 1
sh [5,5,5,4] = 5
```

(1 point)

2. Write a function `removeAt n` which removes the n-th element in a list. Return the list unmodified if there is no n-th element or if the given index is less or equal to zero. Examples:

```
removeAt 3 ["Innsbruck", "Tirol", "Deutschland", "Wien"] = ["Innsbruck", "Tirol", "Wien"]
removeAt 66 [1,2,3] = [1,2,3]
```

(1 point)

3. Write a function `insertS x ys` which inserts an element `x` into the list `ys` after the first element in `ys` which is smaller or equal. If no such element exists in `ys`, append `x` at the end. The inserted element and the elements in the list must have a datatype which is an instance of `Ord`. Examples:

```
insertS 3 [7,7,7,1,7] = [7,7,7,1,3,7]
insertS 3 [] = [3]
insertS "A" ["C","B","A"] = ["C","B","A","A"]
insertS 0 [132,132] = [132,132,0]
```

(1 point)

[1] http://cl-informatik.uibk.ac.at/teaching/ws19/fp/slides/04x1.pdf
[2] http://cl-informatik.uibk.ac.at/teaching/ss20/fp/index.php#exercises

Exercise 6.2 *Higher-Order-Functions* **4 p.**

1. Consider Exercise 7.2 from the previous sheet. Starting from the solution in OLAT (also included in the template file), rewrite `encode` and `decode` to higher order functions, so that they no longer assume a function `code` to exist, but take one as argument, their type being `(Char -> String) -> String -> String`. Use this alternative code function `code2` to test them:

   ```
   code2 :: Char -> String
   code2 'a' = "11"
   code2 'b' = "101"
   code2 'c' = "01"
   code2 'd' = "0011"
   ```

   In particular it should hold that `decode code2 (encode code2 x) = x` for all `x` containing only `'a'`, `'b'`, `'c'` and `'d'`. (1 point)

2. Write a function `compareCodes` which takes two code functions and a `String`, and returns a triple containing a `Bool` that is true iff the first code is better than the second, and the two encodings. Here *better* means that the encoding of the same `String` is shorter, for instance the encoding of `"ab"` using `code` (results in `"011"`) is shorter than the encoding using `code2` (results in `"11101"`). Use your function `compareCodes` to find `String`s (of length at least 5) for which `code` and `code2` respectively generate shorter encodings. (2 points)

3. Write a higher order function `nTimes` which takes a function and an `Integer` and returns a function. Applying the returned function to some argument should be equal to applying the argument function n times to the same argument, for instance:
   `nTimes f 3 x = f (f (f x))` for all `x` and
   `nTimes (1+) 5 3 = 8`.

   Also write down the type signature.

   You can test your function by running `testNTimes`, but as usual you do not have to understand the test functions.

   Hint: You might want to make this function recursive and you might want to look up the Haskell function `id`. (1 point)

Exercise 6.3 *Type-Checking* **3 p.**

For each of the three functions given below, determine the result of the type-inference algorithm (slides 27–31 of part 3). Based on this inferred type, determine which of the given type declarations are valid. **You should solve this exercise without GHCi!** You can find the types of all built-in functions that are used in this exercise on slides 19–23 of part 3.

1. `func_1 x y = div x y + x` (1 point)
   - `func_1 :: Integer -> Integer -> Integer`
   - `func_1 :: Float -> Float -> Float`
   - `func_1 :: Integral a => a -> a -> a`
   - `func_1 :: Num a => a -> a -> a`

2. `func_2 x y z = if x then y == z else y /= z` (1 point)
   - `func_2 :: Eq a => a -> a -> a -> a`
   - `func_2 :: Eq a => Bool -> a -> a -> Bool`
   - `func_2 :: Bool -> Bool -> Bool -> Bool`
   - `func_2 :: Ord a => Bool -> a -> a -> Bool`

3. `func_3 x y z` (1 point)

   ```
   | x == z = (y, z)
   | y < 0 = (y + z, z)
   | otherwise = (y - z, z)
   ```

- `func_3 :: Float -> Float -> Float -> (Float, Float)`
- `func_3 :: Eq a => a -> a -> a -> (a, a)`
- `func_3 :: (Eq a, Ord b, Num b) => a -> b -> b -> (b, b)`
- `func_3 :: (Ord a, Num a) => a -> a -> a -> (a, a)`

Note: Guarded equations were not part of the type-checking algorithm in the lecture. Figure out how to extend the type-checking to guarded equations and use it to solve this exercise.