universität innsbruck

Functional Programming	SS 2020	LVA 703025
Exercise Sheet 10, 10 points		Deadline: Monday, May 25, 2020, 10am

- Watch the lecture of week 10 and study part 6 and 7 of the slides[1].
- Please write all the Haskell code into a single .hs-file and upload it in OLAT.
- Add exercise 10.1 as comment to the .hs-file.
- You can use the template `.hs`-files that are provided on the proseminar page[2].
- Your .hs-files should be compilable with ghci.
- Don't forget to mark your completed exercises in OLAT.

Exercise 10.1 *Infinite Lists, Lazy Evaluation* **2 p.**

Consider the following function definitions:

```
index 0 (x:xs) = x
index n (x:xs) = index (n-1) xs

zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

ones = 1 : ones
nats = 0 : zipWith (+) nats ones

get n = index n nats
```

1. Evaluate the expression `get 2` step by step on paper. You can summarize multiple evaluation step in one as long as the evaluation order/strategy is still clear. (1 point)
2. Which of the following two Haskell expressions terminates and why[3]?

   ```
   foldr (\x acc -> if x == 1 then 1 else acc) 0 nats
   foldl (\acc x -> if x == 1 then 1 else acc) 0 nats
   ```

   Evaluate the two expressions on paper to explain the difference in behaviour. (1 point)

Exercise 10.2 *Tail Recursion* **3 p.**

1. Write a tail recursive function `greatest` which finds the greatest element in a non-empty list of integers. (1 point)
2. Write a tail recursive function `average` which computes the average of a non-empty list of doubles. (1 point)
3. Consider the function `to_upper_str` from slide-set 3, slide 54. Write a tail-recursive function equal to `to_upper_str`. You may use `toUpper :: Char -> Char` from `Data.Char`. (1 point)

[1] `http://cl-informatik.uibk.ac.at/teaching/ws19/fp/slides/07x1.pdf`
[2] `http://cl-informatik.uibk.ac.at/teaching/ss20/fp/index.php#exercises`
[3] See `https://wiki.haskell.org/Fold#Overview` for the definition of `foldl` and `foldr`.

Exercise 10.3 *Mutual recursion* **2 p.**

In the lecture, mutual recursion was only briefly mentioned. In fact, mutual recursion can always be translated into direct recursion. Therefore, in this exercise you should think about how such a translation could work.

1. Consider the mutually recursive functions `f` and `g`.

```
f :: Fractional t => t -> [t] -> t
f a [] = a
f a (x:xs) = g (a + x/3) xs

g :: Fractional t => t -> [t] -> t
g b [] = b
g b (x:xs) = f (b + x*x) (x:xs)
```

   Rewrite function `f` and `g` without mutual recursion. To this end you may define arbitrary auxiliary (non-mutual-recursive) functions.

   Hint: you don't have to understand what `f` and `g` actually compute. (1 point)

2. Similarly to the previous exercise, rewrite functions `c` and `d` so that no mutual recursion is used. (1 point)

```
import Data.Char (ord)

c :: String -> Int
c [] = 1
c xs = sum . map ord $ d xs

d :: String -> String
d [] = []
d (x:xs) = replicate (c xs) x ++ xs
```

Exercise 10.4 *Infinite Lists* **3 p.**

We want to create a function `fp_nums` that returns a list of integers that fulfills the following four conditions:

- The list begins with the number `1`.
- If a number `x` is an element of the list, the numbers `2*x`, `3*x` and `5*x` are also elements of the list.
- The list is in ascending order without duplicates.
- The list contains no other numbers.

1. Write a function `merge` that merges two list into one. `merge xs ys` should fulfill the following conditions:
   - All elements in `merge xs ys` are also elements from `xs` or `ys`.
   - If `xs` and `ys` are in ascending order and contain no duplicates, then `merge xs ys` is in ascending order and contains no duplicates.

   Example: `merge [1,18,200] [19,150,200,300] = [1,18,19,150,200,300]`

   (1 point)

2. Define the infinite list `fp_nums` using the function `merge` and functions like `map (3*)`. Also have a look at the definition of `fibs` in "Part 7 – Efficiency and Lazy Evaluation" of the lecture slides for ideas.

   Example: `take 7 fp_nums = [1,2,3,4,5,6,8]`

   (2 points)