universität innsbruck

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 8, 10 points		Deadline: Wednesday, December 1, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_08.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Partial Application and Sections* **3 p.**

Consider the following functions:

```
div1 = (/)
div2 = (2 /)
div3 = (/ 2)
eqTuple f = (\(x, y) -> f x == f y)
eqTuple' f (x, y) = f x == f y
```

1. Explain what these functions do and give the most general type signature for each function (do not use GHCi to find the type signatures). Give an example that shows the difference between `div2` and `div3` and explain why they are different. (1 point)
2. We say that a Haskell function `f` is equal to a Haskell function `g`, whenever `f x1 .. xN = g x1 .. xN` for all inputs `x1`, ..., `xN`. Based on this definition, are the functions `eqTuple` and `eqTuple'` equal? Justify your answer. (1 point)
3. Which of the functions `foo1 x y = y / x` and `foo2 x y = (\u v -> v / u) y x` are equal to `div1` from above? Justify your answer. (1 point)

Solution 1

1.
```
-- takes two fractional numbers x and y and computes x / y
div1 :: Fractional a => a -> a -> a
div1 = (/)

-- takes a fractional number x and computes 2 / x
div2 :: Fractional a => a -> a
div2 = (2 /)

-- takes a fractional number x and computes x / 2
div3 :: Fractional a => a -> a
div3 = (/ 2)

-- takes a function f and yields a function that
-- takes a pair (x, y) and computes f x == f y
eqTuple :: Eq b => (a -> b) -> (a, a) -> Bool
eqTuple f = (\(x, y) -> f x == f y)

-- takes a function and a pair (x, y) and computes f x == f y
eqTuple' :: Eq b => (a -> b) -> (a, a) -> Bool
eqTuple' f (x, y) = f x == f y
```

The following example shows that `div2` and `div3` are not equal: `div2 5` $\neq$ `div3 5`. We use partial application to provide a single argument of `(/)`: In `div2` we provide `2` as the first argument (the numerator) and in `div3` we provide `2` as the second argument (the denominator).

2. The functions `eqTuple` and `eqTuple'` are indeed equal: on every input function `f` and pair `(x, y)` they return the same value `f x == f y`.

3. The function `foo1` is not equal to `div1`, since `foo1 1 2` = `2 / 1` $\neq$ `1 / 2` = `div1 1 2`.

   The function `foo2` is equal to function `div1`, since `foo2 x y` = `(\u v -> v / u) y x` = `(\v -> v / y) x` = `x / y` = `div1 x y` for all `x` and `y`.

Exercise 2 *Higher-Order Functions and Lambdas* **5 p.**

1. Implement a recursive higher-order function `fan :: (a -> Bool) -> [a] -> [[a]]` that takes a predicate[1] and a list, and "fans out" the list into a list of sublists such that for each sublist either *each* element satisfies the predicate or *none* does. Moreover, your implementation should satisfy the equation

   `concat (fan p xs) == xs` (3 points)

   that is, concatenating the result of a call to `fan` results in the original list.

   Examples:
   ```
   fan undefined [] == []
   fan even [1..5] == [[1],[2],[3],[4],[5]]
   fan (== 'T') "This is a Test" == ["T","his is a ","T","est"]
   ```

2. Use `fan` from exercise 1 together with some lambda expression to implement a function

   ```
   splitOnNumbers = fan (\x -> ... x ...)
   ```

   that splits a given text into numbers and non-numbers. (1 point)

   Example: `splitOnNumbers "8 out of 10 cats" == ["8"," out of ","10"," cats"]`

   Hint: Recall that `Char` is an instance of `Ord`.

3. Use `fan` from exercise 1 to implement a function `splitBy :: (a -> Bool) -> [a] -> [[a]]` that splits a given list into sublists such that only parts remain that do not satisfy the given predicate. (1 point)

   Example: `splitBy (== '\n') "Just\nsome\nlines\n" == ["Just","some","lines"]`

Hint: If you did not manage to implement `fan` of exercise 1, you may use the following implementation in exercises 2 and 3:

```
import Data.List
fan p = groupBy (\x y -> p x == p y)
```

Solution 2

1. ```
   fan :: (a -> Bool) -> [a] -> [[a]]
   fan p [] = []
   fan p [x] = [[x]]
   fan p (x:xs)
     | p x == p y = (x:y:ys) : yss
     | otherwise  = [x] : (y:ys) : yss
     where
       ((y:ys):yss) = fan p xs
   ```

2. `splitOnNumbers = fan (\x -> '0' <= x && x <= '9')`

3. `splitBy p = filter (not . p . head) . fan p`

[1] Functions that return `Bool`s are sometimes called *predicates*, since they "decide" whether their input satisfies some property. For example `even` from the `Prelude` is a predicate on integers.
   ```

Exercise 3 *The* `foldr` *Function* **2 p.**

Implement the following functions using `foldr` instead of recursion. In the process, you may find lambda expressions useful.

1. Consider a function that converts a list of digits (represented as `Integer`s) into an `Integer`:

   ```
   dig2int :: [Integer] -> Integer
   dig2int [] = 0
   dig2int (x:xs) = x + 10 * dig2int xs
   ```

   Examples: `dig2int [2,1,5] == 512`

   Implement a variant `dig2intFold` of `dig2int` using `foldr`. (1 point)

2. Consider a function `suffs` that computes all suffixes of a list, from longest to shortest:

   ```
   suffs :: [a] -> [[a]]
   suffs [] = [[]]
   suffs (y @ (_ : xs)) = y : suffs xs
   ```

   Examples:

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

   Implement a variant `suffsFold` of `suffs` using `foldr`. (1 point)

Solution 3

1. ```
   dig2intFold :: [Integer] -> Integer
   dig2intFold = foldr (\x acc -> x + 10 * acc) 0
   ```

2. ```
   suffsFold :: [a] -> [[a]]
   suffsFold = foldr (\x acc -> (x : head acc) : acc) [[]]
   ```