universität innsbruck

Functional Programming	WS 2019/2020	LVA 703024+703025
Exercises 8		Deadline: November 27, 2019, 6am

- Please write all your Haskell functions from this exercise sheet into a single .hs-file and upload it in OLAT.
- You can use a template .hs-file that is provided on the proseminar page.
- The file should compile with ghci.
- Once the file has been uploaded, it cannot be changed or resubmitted!

Exercise 8.1 *Higher-Order-Functions* **3 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. (1 point)

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 8.2 *Partial Application* **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 of 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.

   Are the functions `eqTuple` and `eqTuple'` equal? Why or why not? Consider the same notion of equality as in exercise 5.2. (2 points)

2. Which of the functions `foo1 x y = y / x` and `foo2 x y = (\u v -> v / u) y x` are equal to `div1`?

   Justify your answer and use the notion of equality from exercise 5.2. (1 point)

Exercise 8.3 *Function Composition and Function Application* **4 p.**

1. Assume we have functions `f, g, h, i :: Int -> Int` and a value `x :: Int`. Which of the following expressions are type-correct?

   (a) `i $ h $ g $ f $ x`

   (b) `f $ g $ h $ i $ x`

   (c) `f . g . h . i $ x`

   (d) `f . g . h . i . x`

   (e) `\ x -> i . h . g . f`

   (1 point)

2. Which of the type-correct expressions from the previous exercise are equivalent to the Haskell expression `f (g (h (i (x))))`? (1 point)

3. Consider the following functions:

```
append :: [a] -> [a] -> [a]
append = (++)

not0 :: (Eq a, Num a) => a -> Bool
not0 = (/=0)

foo x = not0 (head (tail (tail x)))

foo1 x (y, z) = not0 (head (append y (append (tail x) z)))
```

   Rewrite function `foo` so that you can drop all parentheses using `(.)` and `$`.

   By using `(.)` and `($)`, is there also a solution without parentheses for `foo1`? If so give the solution, if not remove as much parentheses as you can (by using `(.)` and `($)`).

   The only allowed modifications in this exercise are dropping parentheses, adding parentheses, drop variables, adding `(.)` and adding `($)`. Add the missing type signatures to `foo` and `foo1` (do not use GHCi). (2 points)