universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 7, 10 points		Deadline: Tuesday, November 28, 2023, 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 .hs file in OLAT.
- Your .hs file must be compilable with ghci.

Exercise 1 *Partial Application* **5 p.**

Consider the following functions:

```
div1      = (/ 2)
div2      = (2 /)
div3      = (. (/ 2))
div4      = ((/ 2) .)

div5 f    = f . div1
div6 x    = \ f -> f (2 / x)
div7 (f, x) = div3 f x
div8      = \ (f, x) -> f (x / 2)
```

1. Explain what the functions `div1` and `div2` do and write down their most general type. Give an example that shows the difference between `div1` and `div2`. (0.5 points)
2. Explain what the functions `div3` and `div4` do and write down their most general type. You can either infer the types manually or obtain them by calling `:t div3`, etc. in GHCI. Give an example that shows the difference between `div3` and `div4`.

   Hint: the type of the composition operation `(.)` is explained in slide 07/17. (1.5 points)
3. We say that a Haskell function `f` with `N` input arguments is equal to a Haskell function `g`, whenever `f x1 .. xN = g x1 .. xN` for all inputs `x1`, ..., `xN`. Based on this definition, which of the following pairs of functions are equal? Justify your answers.

   (i) `div3` and `div5` (1 point)

   (ii) `div5` and `flip div6` (1 point)

   (iii) `div7` and `div8` (1 point)

Solution 1

1.
```
div1 :: Fractional a => a -> a
div1 = (/ 2)
div2 :: Fractional a => a -> a
div2 = (2 /)
div3 :: Fractional a => (a -> b) -> a -> b
```
   The function `div1` takes a Fractional argument and returns the argument divided by 2. The function `div2` takes a Fractional argument and returns 2 divided by this argument. For example, `div1 1 = 1/2 = 0.5` and `div2 1 = 2/1 = 2.0`.

2. ```
   div3 :: Fractional a => (a -> b) -> a -> b
   div3 = (. (/ 2))
   div4 :: Fractional b => (a -> b) -> a -> b
   div4 = ((/ 2) .)
   ```
   The function `div3` takes a function `f` of type `a -> b` and a Fractional argument `x` of type `a` and returns `f (x/2)`. The function `div4` takes a function `f` of type `a -> b` and a Fractional argument `x` of type `a` and returns `(f x)/2`. For example, `div3 (+1) 3 = (3/2) + 1 = 2.5`, whereas `div4 (+1) 3 = (3 + 1)/2 = 2.0`.

3. (i) `div3` and `div5` are equal as `div5 f` is the same as `f . (/ 2)` by the definition of `div1`, which is equivalent to `(. (/ 2)) f` by the definition of the composition operator. Therefore, `div5 f` is equal to `div3 f` by the definition of `div3`. It follows that `div5` and `div3` are equal. (1 point)

   (ii) `div5` and `flip div6` are not equal. To see this, consider a counterexample with input arguments `(+1)` and `3`. Then `(flip div6) (+ 1) 4` is equivalent to `(2/4) + 1` which equals `1.5`. This is not equal to `div5 (+ 1) 3` which equals `2.5`.

   (iii) `div7` and `div8` are equal. This follows as `div7 (f,x)` equals `(. (/2)) f x` which is equal to `(f . (/ 2)) x` by applying the composition operator to `f`, which equals `f (x/2)` by function application. Then `div8 (f,x)` equals `f (x/2)` by unwrapping the lambda function.

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

In this exercise, we consider a simple `Employee` datatype.

```
type Name = String
type Age = Integer
type Salary = Double
data Employee = Employee Name Age Salary deriving Show
```

1. Write a function `mapEmployee :: (Name -> Name) -> (Age -> Age) -> (Salary -> Salary) -> Employee -> Employee` that takes three functions to update the name, the age, and the salary of an employee, respectively. (1 point)

2. The ages and salaries of the employees are updated every year. Write a function `nextYear` which takes a list of employees and increases their ages by 1 and their salaries by 20 %. You may not use explicit recursion on lists or list comprehensions (which will be explained in lecture 8).

   Hint: `map`, `mapEmployee`, sections, and the identify function `id` might be useful. (1 point)

3. Use the built-in functions `map` and `filter` as well as function composition and λ-abstractions to write a function that returns a list of the pairs of names and salaries of the employees with a salary $< 60\,000$. You may not use explicit recursion on lists or list comprehensions.

   Hint: look at the example on slide 07/19. (1 point)

4. Extend the quicksort implementation `qsort` from the lecture to a function

   ```
   qsortBy :: (a -> a -> Bool) -> [a] -> [a]
   ```

   where the first argument of `qsortBy` is a parametric less-or-equal function. (1 point)

5. Define a function that takes a list of employees and produces a sorted list of employee names, sorted by salary in decreasing order. You may assume that `qsortBy` is available but should not use explicit recursion on lists or list comprehensions. (1 point)
   ```

Solution 2

```haskell
-- 1
mapEmployee :: (Name -> Name) -> (Age -> Age) -> (Salary -> Salary) ->
  Employee -> Employee
mapEmployee n a s (Employee name age salary) = Employee (n name) (a age) (s salary)

-- 2
nextYear :: [Employee] -> [Employee]
nextYear = map (mapEmployee id (+1) (* 1.2))

-- 3
getName :: Employee -> Name
getName (Employee n _ _) = n

getSalary :: Employee -> Salary
getSalary (Employee _ _ s) = s

lowIncomeEmployees :: [Employee] -> [(Name,Salary)]
lowIncomeEmployees = map (\e -> (getName e, getSalary e)) . filter ((< 60000) . getSalary)

-- 4
qsortBy :: (a -> a -> Bool) -> [a] -> [a]
qsortBy _ [] = []
qsortBy le (x : xs) = qsortBy le (filter (`le` x) xs)
    ++ [x] ++ qsortBy le (filter (not . (`le` x)) xs)

-- 5
employeesByIncome :: [Employee] -> [Name]
employeesByIncome = map getName . qsortBy (\ e1 e2 -> getSalary e1 >= getSalary e2)
```