universität innsbruck

Functional Programming	WS 2020	LVA 703025
Exercise Sheet 7, 10 points	Deadline: Wednesday, November 25, 2020, 6am	

- Please write all the Haskell code into a single .hs-file and upload it in OLAT.
- You can use the template .hs-file that is provided on the proseminar page.
- Your .hs-file should be compilable with ghci.
- Don't forget to mark your completed exercises in OLAT.

Exercise 7.1 *Type Classes* **2 p.**

Types and type classes are useful to model data.
The Electricity generation of Europe in 2019 by sources is as follows:[1]

Resource	Type	Electricity in TWh
NaturalGas	Fuel	52
Oil	Fuel	768
Coal	Fuel	699
Nuclear	Fuel	928
Hydro	Fuel	632
Wind	Fuel	587
Renewables	FuelType	1469
Fussel	FuelType	1519
CarbonZero	Summary	2397
Resource	Summary	3916

1. Given the types:

   ```
   data Fuel = NaturalGas | Nuclear | Hydro | Wind
   data FuelType = Renewables | Fussel
   data Summary = CarbonZero | Resource
   ```

   Make `Fuel`, `FuelType` and `Summary` instances of the class `Electricity` (slides 15–18 of part 3), given by

   ```
   class Electricity a where
     generation :: a -> Integer
   ```

   (1 point)

2. Implement `generateMore :: (Electricity a, Electricity b) => a -> b -> Bool` comparing electricity generation. The function returns `True` if the first source of energy generates more electricity than the second one.

   Compare the type of `generateMore` with that of the following function:

   ```
   generateMore2 :: (Electricity a) => a -> a -> Bool
   ```

   What is the difference between these two types? Can you think of two concrete expressions `e_1` and `e_2` such that only one of `generateMore e_1 e_2` and `generateMore2 e_1 e_2` is accepted by `ghci`?

   Note that the notation `(Electricity a, Electricity b) =>` means that `a` may only be instantiated by a type that is an instance of type-class `Electricity`, likewise, `b` can only be instantiated by instances of `Electricity`. (1 point)

[1] Source: https://www.bp.com/content/dam/bp/business-sites/en/global/corporate/pdfs/energy-economics/statistical-review/bp-stats-review-2020-electricity.pdf

Exercise 7.2 *Error Handling* **3 p.**

Each complex number C consists of a real part a and an imaginary part b.

$$C = a + bi \tag{1}$$

where i indicates the imaginary part.
When we visualise a complex number in the complex plain, we are interested in the slope S of the complex number $a + bi$ which is undefined if $a = 0$ and otherwise it is defined as:

$$S = \frac{b}{a} \tag{2}$$

1. Define a type `Complex` for complex numbers and implement two functions

   ```
   slope :: Complex -> Double
   ```

   (should calculate the slope and raise an error if a equals 0) and

   ```
   slopeSum :: Complex -> Complex -> Complex -> Double
   ```

   (should calculate the sum of the slopes of three complex numbers).

   Hint: For storing the real and imaginary parts of a complex number you can use the Haskell type `Double`. Note that all arithmetic operations and comparisons are available for type `Double` using standard notation, i.e., you can use `+`, `-`, `*`, `/`, `==`, etc. Here, `/` is standard division on floating point numbers, and not the integral division with remainder that you already know (`div`). (1 point)

2. Whereas in the previous task error handling is done implicitly by `ghci`, the task is now to use the `Maybe`-type to perform explicit error handling. Here, `Nothing` represents an error and `Just x` a successful calculation with result `x`.

   Reimplement `slope` and `slopeSum` from above with the following types:

   ```
   slopeMaybe    :: Complex -> Maybe Double
   slopeSumMaybe :: Complex -> Complex -> Complex -> Maybe Double
   ```

   (1 point)

3. Compare the usage of `error` with the usage of the `Maybe`-type to represent errors. To this end, look at the previous two tasks and think of the following function: given three complex numbers it either produces the sum of their slopes as string, or the string `"one of the numbers has an undefined slope"`. (1 point)

Exercise 7.3 *Types, Guards, Cases, Patterns and Where* **5 p.**

In this exercises you will write an app that gives a user advice on his or her lifestyle. [2]

Based on a users *biological sex*, *weight* (kg), *height* (m), *age* and *exercise pattern* the app will give one of the following three advises:

- The user is *healthy* and does not have to change his/her lifestyle pattern
- The user is *underweight* and given his/her exercise pattern should consume more than X calories per day to gain weight
- The user is *overweight* and given his/her exercise pattern should consume less than X calories per day to lose weight

The app will use the *body mass index* (BMI)[3] to determine whether a user is *underweight*, *healthy* or *overweight*:

$$\text{BMI} = \frac{\text{mass}}{\text{height}^2} \tag{3}$$

The ministry of health in Austria defines the three categories as follows:[4]

[2] Disclaimer: we don't recommend to use this app. For health advise we recommend you contact your doctor.
[3] https://en.wikipedia.org/wiki/Body_mass_index
[4] https://www.gesundheit.gv.at/lexikon/k/bmi

Category	BMI Range
Underweight	BMI $<$ 18.5
Healthy	18.5 $\geq$ BMI $<$ 25.0
Overweight	25.0 $\leq$ BMI

The app will calculate the amount of calories a user burns per day using the *total energy expenditure* (TEE),[5] which is calculated by multiplying the *Basal Metabolic Rate* (BMR)[6] with the *physical activation level* (PAL).[7]

$$\text{TEE} = \text{BMR} \times \text{PAL} \tag{4}$$

The BMR is calculated using the Harris-Benedict equation:[8]

$$\begin{aligned} \text{Female} \rightarrow \text{BMR} &= 655 + 9.6 \times \text{weight} + 1.8 \times \text{height} - 4.7 \times \text{age} \\ \text{Male} \rightarrow \text{BMR} &= 66 + 13.7 \times \text{weight} + 5 \times \text{height} - 6.8 \times \text{age} \end{aligned} \tag{5}$$

For the PAL we use the following categories:

Category	PAL
Sedentary	1.53
Active	1.76
Extreme	2.25

While doing this exercise try to

- use *guards*, *cases* and *pattern matching* as much as possible;
- use the *where* construct for writing auxiliary functions;
- minimize duplicate code.

Most of the exercises have individual test functions available (see bottom of the `template_07.hs` file), *uncomment* the test cases in order to use them. All the tests can be executed at once using the function `test73`.

1. Define four data-types:
   - `Sex` is an enumeration that can assume the values male or female.
   - `Person` with a constructor `Person` that takes the following four arguments in this exact order: sex (`Sex`), weight (`Double`), height (`Double`) and age (`Integer`).
   - `HealthCategory` is an enumeration that can take the values underweight, healthy or overweight. Make `HealthCategory` an **`instance`** of the `Eq` type-class.
   - `Exercise` is an enumeration that can take the values sedentary, moderate or active. Implement a function `pal :: Exercise -> Double` that returns the PAL for a given exercise category.

   Available test functions: `testPerson`, `testHealthCategory` and `testPal`. (1 point)

2. Implement a function `health :: Person -> HealthCategory` that calculates the BMI of a person and returns his/her health category. Available test function: `testHealth`. (1 point)

3. Implement a function `bmr :: Person -> Double` that calculates the BMR of a person.

   (Note: age is defined as an `Integer`, it can be converted to a double using the `fromInteger` function.)

   Available test function: `testBmr`. (1 point)

[5] https://en.wikipedia.org/wiki/Energy_homeostasis#Expenditure
[6] https://en.wikipedia.org/wiki/Basal_metabolic_rate
[7] https://en.wikipedia.org/wiki/Physical_activity_level
[8] https://en.wikipedia.org/wiki/Harris-Benedict_equation

4. Implement a function `healthAdvice :: Person -> Exercise -> String` that takes a person and his/her exercise category as arguments and returns a string containing the health advice.

   You can come up with your own personalized advice and add any information you want (like BMI, BMR etc.). But in the case of an underweight or overweight person the health advice must contain advice on how many calories the person should consume (TEE).

   (Note: numbers (`Integer`, `Double` etc.) can be converted to a `String` using the function `show`)

   No test function available. (2 points)