universität innsbruck

Functional Programming	WS 2020	LVA 703025
Exercise Sheet 6, 10 points		Deadline: Wednesday, November 18, 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 6.1 *Telling the time* 4 p.

Consider the following datatype representing the time of day:
`datatype Time = Time Integer Integer Integer`
The first integer tells the number of hours, the second integer the number of minutes and the third integer the number of seconds. For example, `Time 6 5 35` represents 5 minutes and 35 seconds past 6.

1. Implement the following three functions:

   ```
   hours :: Time -> Integer
   minutes :: Time -> Integer
   seconds :: Time -> Integer
   ```

   where `hours` extracts the first, `minutes` the second and `seconds` the third value from an input of type `Time`. (1 point)

2. Implement a function `well_formed` that takes a `Time t` as input and returns `True` if and only if `t` is well-formed. A period of time is well-formed, if if the value of hours is between 0 and 23 and the value of the minutes and the value of the seconds are each between 0 and 59.

   Examples:

   ```
   well_formed (Time 23 0 59) = True
   well_formed (Time 0 0 0) = True
   well_formed (Time 24 0 0) = False
   well_formed (Time 2 67 7) = False
   ```

   (1 point)

3. Implement a function `pretty :: Time -> String` that takes a value of type `Time` and returns a formatted `String`.

   Examples:

   ```
   pretty (Time 23 0 45) = "23:00:45"
   pretty (Time 1 1 1) = "01:01:01"
   ```

   Specification:

   - Hours, minutes and seconds should be seperated by colons.
   - The hours, minutes and seconds should be padded to two digits.
   - You are allowed to assume that the input to the `pretty` function is well-formed.

   An `Integer x` can be converted to a `String` by simply using `show x`. (1 point)

4. Implement a function `tick :: Time -> Time` that increases a value of type `Time` by one second. The function `tick` only needs to work for well-formed values of type `Time`. `tick x` must be well-formed, if `x` is well-formed. At midnight there should be a wrap around, see examples.

   Examples:

   ```
   tick (Time 1 2 3) = Time 1 2 4
   tick (Time 1 59 59) = Time 2 0 0
   tick (Time 23 59 59) = Time 0 0 0
   ```

   (1 point)

Exercise 6.2 *Data Types* 3 p.

This exercise should show you how types can be used to model data.

1. Create a data type `Region` that models different kinds of territories.

   Specification:

   - There are three kinds of regions: City, State and Country.
   - Every region has a name (use `String`) and the number of inhabitants (use `Integer`).

   Create three constants[1] `innsbruck :: Region`, `tyrol :: Region` and `austria :: Region` using the data below.

Location	Type	Population
Innsbruck	City	132110
Tyrol	State	754705
Austria	Country	8858775

   Hint: It could be useful to encode the three different kinds of regions in its own data type and reuse it in the definition of `Region`. (1 point)

2. Implement the following three functions:

   ```
   name :: Region -> String
   population :: Region -> Integer
   typeOfRegion :: Region -> String
   ```

   `name` should return the name of the `Region` and `population` the number of inhabitants. `typeOfRegion` should return a `String` based on the type of a `Region`. For example, `"city"` if it is a city.

   Examples:

   ```
   name innsbruck = "Innsbruck"
   population austria = 8858775
   typeOfRegion innsbruck = "city"
   typeOfRegion tyrol = "state"
   typeOfRegion austria = "country"
   ```

   (1 point)

3. Implement a function `info :: Region -> String` that returns a short summary about a `Region`.

   Examples:

   ```
   info innsbruck = "The city of Innsbruck has a population of 132110."
   info tyrol = "The state of Tyrol has a population of 754705."
   info austria = "The country of Austria has a population of 8858775."
   ```

   An `Integer x` can be converted to a `String` by using `show x`. (1 point)

Exercise 6.3 *Pattern Matching* 3 p.

This exercise asks you to program the key ingredients for a function that determines whether and where one string occurs in another string.

[1] A constant in Haskell is a function that takes no parameters. Therefore the function's return value cannot depend on any parameter and is therefore constant.

1. Implement the functions

   ```
   ev :: String -> String
   od :: String -> String
   merge :: String -> String -> String
   ```

   where `ev` extracts the string of characters at even positions in a string, and `od` the string of characters at odd positions. `merge` merges two strings by interleaving their characters, starting with the first character of the first string, then that of the second string, then the second character of the first string, etc.. In case of strings not having the same length, `merge` should result in an error.

   Examples:

   ```
   ev "ababacda" = "aaad"
   od "ababacda" = "bbca"
   merge "aaad" "bbca" = "ababacda"
   ```

   More generally, for any string `s` of even length, `merge (ev s) (od s) == s` should evaluate to `True`.

   Write these three functions using, as much as possible, pattern matching on strings. Note that matching a pattern `c:s` (with `:` as seen last week) with a non-empty string, results in `c` being the character at the head of the string and `s` the tail of the string. For instance, the `head` and `tail` functions could be defined by `head (c:s) = c` respectively `tail (c:s) = s`. Moreover, `""` is a pattern matching (only) with the empty string. (2 points)

2. Implement the functions

   ```
   lshift :: String -> String
   sand :: String -> String -> String
   ```

   where `lshift` takes a non-empty string, removes its first element and appends `"F"` to it. `sand` takes two strings of the same length and yields a string of that same length having a `T` at some position if both input strings have a `T` at that same position, and an `F` otherwise. That is, `sand` acts like a logical *and* on corresponding characters in a string, with `T` standing for true and `F` (and other characters) for false.

   You may choose how you implement these functions, using `head`, `tail`, `null` as seen before, or using pattern matching as in the previous item.

   Examples:

   ```
   lshift "ababacda" = "babacdaF"
   sand "aTbTTcF" "xTbcTcT" = "FTFFTFF"
   ```

   (1 point)

If you've implemented all 5 functions in this exercise correctly, the `match` function based on them and provided in the template, should work correctly.