universität innsbruck

Functional Programming	WS 2022	LVA 703025
Exercise Sheet 7, 10 points	Deadline: Wednesday, November 23, 2022, 6am	

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_07.hs` provided on the proseminar page.
- Your .hs-file should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Partial application* **2 p.**

Consider the following functions:

```
pow1 = (2 ^)
pow2 = (^ 2)
maxTuple f = (\ (x, y) -> max (f x) (f y))
maxTuple' f (x, y) = max (f x) (f y)
```

1. Explain what the functions `pow1` and `pow2` do and explain their most general type. You can either infer the types manually or obtain them via `:t pow1`, etc. Give an example that shows the difference between `pow1` and `pow2` 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 `maxTuple` and `maxTuple'` equal? Justify your answer. (1 point)

Exercise 2 *Lists* **3 p.**

Many of the following tasks can easily be solved with the help of the common functions on lists, e.g. `filter`, `map`, `reverse`, `sum`. You are **not** allowed to use list comprehensions which will be explained in lecture 8. Use higher-order functions and λ-abstractions if appropriate. Add type signatures to each function that are as general as possible.

You are given a dataset of website visits. Here is an example of how such a dataset might look:

```
webData :: [String]
webData = ["Youtube", "Google", "Facebook", "Youtube", "Facebook",
           "Youtube", "Facebook", "Google", "Youtube"]
```

Each entry in the list represents a visit to a website. The goal is to create a ranking based on the number of times a website has been visited. For example, Google has been visited two times.

1. First you have to determine which websites occur in the dataset. Write a function `uniqueWebsites` that takes a dataset of the same form as `webData` and returns a list with the websites that occur in the dataset. For the example dataset a result could be:

   ```
   uniqueWebsites webData = ["Google", "Facebook", "Youtube"]
   ```

   (1 point)
2. The next step is to write a function `count` that takes the name of a website and a dataset and returns how often that website occurs in the dataset. For the example dataset and `website = "Google"` the result is: `count "Google" webData = 2`. (1 point)

3. The functions `uniqueWebsites` and `count` can now be combined to create a function that gives us a ranking of website visits. Write a function `result` that takes the dataset `webData` and returns an *ordered* list of tuples `[(count, website)]`, where the first tuple in the list is the website with the highest count and the last tuple the website with the lowest count (If two websites have equal count the ordering does not matter). For the example dataset the result is:

```
result webData = [(4, "Youtube"), (3, "Facebook"), (2, "Google")]
```

Hint: have a look at the functions `sortBy` and `compare`. (1 point)

Exercise 3 *Higher-Order Functions* **5 p.**

Exercise 3 of week 5 was about the Collatz conjecture and the Mercator series. They will be revisited in this exercise.

1. A series is an infinite sum of numbers

$$\sum_{i=0}^{\infty} a_i$$

We cannot sum infinitely many numbers in Haskell, but we can approximate the result by stopping the summation once it is accurate enough. The goal of this exercise is to write a higher-order function `series :: (Integer -> Double) -> [Double]`. Its single argument is a function which returns a_i for a given index (i.e. `f i` $= a_i$).

The output of `series` should be a list where the $i-$th element corresponds to the sum up to (including) a_i. If `f` is the argument passed to `series`, the result should be `[f 0, f 0 + f 1, f 0 + f 1 + f 2, ...]`.

We stop our approximation the same way as it was done for the Mercator series, that is, as soon as the list element at some position is equal to the tentative next list element.

The template includes a modified version of the solution to the Mercator series exercise from week 5. Feel free to use it as the basis for your implementation of `series`.

Example:

```
f :: Integer -> Double
f i = 1 / 2^(i * i)

series f == [1.0, 1.5, 1.5625, 1.564453125, 1.5644683837890625,
 1.5644684135913849, 1.5644684136059368, 1.5644684136059386]
```

(1.5 points)

2. Use the function `series` from the previous task to implement the Mercator series. Do not define a named function for the argument of `series`, but write a λ-abstraction.

As a reminder, here is the formula for the Mercator series, on the left as the infinite sum, and on the right as the recursive definition for the partial sum (y_n) up to the $n-$th element

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}x^n}{n} \qquad\qquad y_n = \begin{cases} x & \text{if } n = 1 \\ y_{n-1} + \frac{(-1)^{n+1}x^n}{n} & \text{if } n > 1 \end{cases}$$

Remark: Be careful, to avoid division by 0, the Mercator series starts at 1 instead of 0.

Examples:

```
mercator 0.0 == [0.0]
mercator 0.01 == [0.0, 1.0e-2, 9.95e-3, 9.950333333333334e-3, 9.950330833333333e-3,
 9.950330853333333e-3, 9.950330853166666e-3, 9.950330853168095e-3, 9.950330853168083e-3]
```

(1 point)

3. In this part of the exercise we model certain sequences. We limit ourselves to sequences in which element i can be calculated from element $i-1$.

Implement the higher-order function `sequence :: (a -> a) -> (a -> Bool) -> a -> [a]`. The first argument of `sequence` is a function which takes an element of the sequence and calculates the next element from it. The second argument is a function which takes an element and indicates whether the sequence

should end with the current element. The third argument of `sequence` should be the first element of the sequence. Feel free to use the implementation of `mercatorList`, or your solution for `series` as a starting point.

Examples:

```
sequence (+ 1) (> 5) 0 == [0, 1, 2, 3, 4, 5, 6]
sequence succ (== 'z') 'a' == "abcdefghijklmnopqrstuvwxyz"
```

(1.5 points)

4. Define a function `collatz :: Integer -> [Integer]` which returns the Collatz sequence starting with the given integer. Use the function `sequence` from the previous task. Again, do not define named functions for the arguments of `sequence` but use λ-abstractions or sections.

   As a reminder, the Collatz sequence for some positive integer n is defined as

   $$a_0 = n, \quad a_{i+1} = \begin{cases} \frac{a_i}{2} & \text{if } a_i \text{ is even} \\ 3a_i + 1 & \text{if } a_i \text{ is odd} \end{cases}$$

   The sequence should end with the first occurrence of 1.

   Examples:

   ```
   collatz 1 = [1]
   collatz 3 = [3, 10, 5, 16, 8, 4, 2, 1]
   ```

   (1 point)