



## Assignment 2 (05.11.2021)

Handin until: Friday, 12.11.2021, 00:00

### Exercise 1: Patterns

[4 Points]

Simplify the following definitions using Pattern Matching:

1. 

```
f :: Integer -> b
f x | x == 42 = ...
```
2. 

```
g :: [a] -> b
g xs | not $ null xs = ...
```
3. 

```
h :: (a, b) -> c
h (_, _) = ...
```
4. 

```
i :: [[a]] -> b
i xs | (not $ null xs) && (null $ head xs) = ...
```
5. 

```
j :: ((a, b), c) -> b
j t = snd . fst $ t
```
6. 

```
k :: (Integer, Integer, Integer, String) -> d
k (a,b,c,xs) | a == b && c == 42 && length xs == 2 && head xs == 'a' = ...
```

### Exercise 2: List Processing

[6 Points]

This exercise is concerned with list processing in Haskell, both via explicit recursion and using functions from the prelude and `Data.List`. You can import the module `Data.List` via

```
1 import Data.List
```

at the top of your Haskell source file or in GHCi. It contains a large number of functions on lists. You may freely use functions from this module unless we explicitly state differently.

API documentation for the prelude, `Data.List` and other modules included in the Haskell installation can be found online at <http://hackage.haskell.org/package/base-4.16.0.0/docs/Data-List.html>, Hoogle (<https://www.haskell.org/hoogle/>) or in your local Haskell documentation.

1. Implement function `map' :: (a -> b) -> [a] -> [b]` that applies a function to every element of a list.

**Examples:**

- ```
1 map' negate [1,2,3,4] == [-1,-2,-3,-4]
2 map' odd    [1,2,3,4] == [True,False,True,False]
```

**Note:** Do not use function `map` to implement `map'`.

2. Please write a similar function `mapEveryOther` that applies its functional argument only to every second element of the input list and leaves the other elements as-is.

**Examples<sup>1</sup>:**

- ```
1 mapEveryOther ((+) 42) [1,2,3,4] == [43,2,45,4]
2 mapEveryOther ((+) 42) []       == []
3 mapEveryOther ((+) 42) [1]      == [43]
```

Before you write down the definition, write down the function's polymorphic type. Compare the types of `map'` and `mapEveryOther` and explain the difference.

<sup>1</sup>Recall `((+) 42)` is a function of type `Integer -> Integer` (partial application)

### Exercise 3: Matrices

[5 Points]

Using lists, a *matrix* might be represented as a list of lists such that each inner list represents a row of the matrix. For example, the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

is represented as follows:

```
1 type Matrix = [[Integer]]
2
3 m1 :: Matrix
4 m1 = [ [1, 2, 3],
5         [4, 5, 6],
6         [7, 8, 9] ]
```

Write a function `trace :: Matrix -> Integer` that computes the *trace* of a *quadratic* matrix such as `m1`. The trace of a quadratic matrix is defined as the sum of the elements on the main diagonal:

$$\text{trace}(a_{ij})_{1 \leq i,j \leq n} = \sum_{l=1}^n a_{ll}$$

Example: `trace m1 ≡ 15`

**Note:** You may assume that the input of your function is a well-formed quadratic matrix. Function `map` might be helpful to implement `trace`.

### Exercise 4: Powerset

[5 Points]

A *set* is a collection of values with no particular order, and no repeated values. We can use lists in Haskell to represent such sets. The *power set* of a set  $S$  is defined as *all the subsets of a set* and always contains the empty set `[]`, as well as the set  $S$  itself:

```
1 powerset []      = [[]]
2 powerset [1,2,3] = [[1,2,3],[1,2],[1,3],[2,3],[1],[2],[3],[]]
3                  = [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]] -- No particular order
```

Write a function `powerset :: [a] -> [[a]]` that computes the power set of its argument list `xs`. You may assume that `xs` does not contain any duplicate elements. Hint: To compute the power set  $S$  of a non-empty list `x:xs`, follow this recipe:

1. Compute the power set  $S'$  of `xs`.
2.  $S = S' \cup \{ \{x\} \cup s \mid s \in S' \}$ .

**Note:** Your solution must only use functions and Haskell constructs that have been discussed in the lecture.