EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer Term 2025
Prof. Torsten Grust, Björn Bamberg
WSI — Database Systems Research Group

Assignment 2

Hand in this assignment until **Tuesday, May 13, 12pm** at the latest.

🤔 Running out of ideas?
Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` 🌐 channel on our Discord server—maybe you'll find just the help you need.

Ⓔxam-style Exercises
Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Task 1: Patterns Ⓔ

Simplify the following definitions using Pattern Matching:

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

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

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

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

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

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

Task 2: List Processing

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

```
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.

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

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

> Note
> Do not use function `map` to implement `map'`.

B 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[1]:

```
mapEveryOther ((+) 42) [1,2,3,4] ≡ [43,2,45,4]
mapEveryOther ((+) 42) []        ≡ []
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.

C Implement function `filter' :: (a -> Bool) -> [a] -> [a]` that filters a list according to a predicate. The predicate is applied to each element of the list, and only those elements for which the predicate returns `True` are included in the result. Examples:

```
filter' even  [1,2,3,4] ≡ [2,4]
filter' odd   [1,2,3,4] ≡ [1,3]
filter' (> 2) [1,2,3,4] ≡ [3,4]
```

> Note
> Do not use function `filter` to implement `filter'`.

[1] Recall `((+) 42)` is a function of type `Integer -> Integer` (partial application)

Task 3: Matrices Ⓔ

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:

```
type Matrix = [[Integer]]

m1 :: Matrix
m1 = [ [1, 2, 3],
       [4, 5, 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:

$$\mathrm{trace}(a_{\mathrm{ij}})_{1 \leq i,j \leq n} = \sum_{k=1}^{n} a_{\mathrm{kk}}$$

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`.

Task 4: Powerset

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:

```
powerset []      ≡ [[]]
powerset [1,2,3] ≡ [[1,2,3],[1,2],[1,3],[2,3],[1],[2],[3],[]]
                 ≡ [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]
```

> Note
> The order of the subsets in the power set does not matter.

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:

(a) Compute the power set S' of `xs`.

(b) $S = S' \cup \{\{\mathtt{x}\} \cup s \mid s \in S'\}$.

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