Prof. Dr. Peter Thiemann

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 2

Exercise 1 (Map, Filter, and Fold)

In the lecture you saw three rather famous examples of higher-order functions[1]:

1. `map` applies a function to each element of a list, and returns a list of the results, e.g.

   ```
   >>> map (\x -> x * 2) [1, 2, 3, 4]
   [2, 4, 6, 8]
   ```

2. `filter` applies a function to each element of a list, and returns a list with those elements for which the function returned `True`, e.g.

   ```
   >>> filter (\x -> x > 2) [1, 2, 3, 4]
   [3, 4]
   ```

3. `foldr` "folds" a list with a binary function, e.g. we can sum up a list with

   ```
   >>> foldr (+) 0 [1, 2, 3, 4]
   10
   ```

   Intuitively, `foldr (+) 0 xs` replaces each `(:)` constructor with `(+)` and the `[]` constructor with `0`, e.g.

   ```
   foldr (+) 0 (1 : (2 : (3 : (4 : []))))
   ```

   is equivalent to

   ```
   (1 + (2 + (3 + (4 + 0))))
   ```

These functions can be used to express many recursive functions on lists in a more concise way, but to do that it is important to be able to spot when a recursive function definition fits the pattern of one of these functions. For example, we can define a function, which sums up a list of integers as a recursive function with pattern matching:

```
sum' :: [Int] -> Int
sum' []     = 0
sum' (x:xs) = x + sum' xs
```

or more concisely by using `foldr`:

```
sum' :: [Int] -> Int
sum' = foldr (+) 0
```

Your task is to first reimplement the `map`, `filter`, and `foldr` functions with the following type signatures

[1] functions which take functions as arguments or return functions as results

```
map'    :: (a -> b) -> [a] -> [b]
filter' :: (a -> Bool) -> [a] -> [a]
foldr'  :: (a -> b -> b) -> b -> [a] -> b
```

and then use them to implement the following functions first directly (using recursion and pattern matching) and then again by using a combination of `map'`, `filter'`, and `foldr'`, as we did for `sum'` in the example above.

For functions, which already exist in the standard library (either automatically imported or in `Data.List`), we append a single quote to the function name, e.g. we use `sum'` instead `sum`, as in the previous exercise. This helps to avoid name clashes and also tells you which of these functions you don't have to write yourself in subsequent programming tasks.

1. Write a function `inc`, which increments each element in a list of integers.

   Examples:

   ```
   >>> inc [1, 2, 3]
   [2, 3, 4]
   ```

2. Write a function `evenList`, which returns all even integers in a list.

   Examples:

   ```
   >>> evenList [1, 2, 3, 4]
   [2, 4]
   ```

3. Write a function `shortStrs`, which takes a list of integers and returns the `String` representations of those integers whose `String` representation is at most 2 characters long.

   Examples:

   ```
   >>> shortStrs [1, 10, 100, 1000, -1, -10]
   ["1", "10", "-1"]
   ```

4. Write two functions `and'` and `or'`, which take a list of booleans. `and'` returns whether all booleans are `True`, whereas `or'` returns whether at least one boolean in the list is `True`.

   Examples:

   ```
   >>> and' [True, True, True]
   True
   >>> and' [True, False, True]
   False
   >>> or' [False, False, False]
   False
   >>> or' [False, True, False]
   True
   ```

5. Write two functions `all'` and `any'`, which take a list and a predicate on the list elements (function from element type to `Bool`) and return whether all elements (`all'`) or at least one element (`any'`) satisfy the predicate.

   Examples:

   ```
   >>> all' (\x -> x < 10) [1, 20, 3]
   False
   >>> any' (\x -> x < 10) [1, 20, 3]
   True
   ```

6. Write a function `length'`, which takes a list and returns its length.

   Examples:

   ```
   >>> length' [10, 20, 30]
   3
   ```

7. Write three functions `idMap`, `idFilter`, and `idFold`, which take a list and return it unchanged.

   The functions should not just return their argument, but instead each function should apply `map'`, `filter'`, and `foldr'` to the argument list and choose the right arguments such that `map'`, `filter'`, and `foldr'` reproduce the input list.

   Examples:

   ```
   >>> idMap [1, 2, 3]
   [1, 2, 3]
   ```

8. Implement `map'` and `filter'` by using `foldr'` instead of implementing them as recursive functions with pattern matching.

Exercise 2 (3-Dimensional Vectors)

Higher-order functions are not only useful for lists but for all kinds of data. Consider the following data type to represent vectors in a 3-dimensional space:

```
data V3 a = V3 a a a deriving (Show, Eq)
```

Values of this data type can be created as e.g.

```
exampleV3 :: V3 Int
exampleV3 = V3 5 3 8
```

We want such a vector data type to support the usual arithmetic operations, such as addition or multiplication, by applying the operations on numbers pointwise to the coordinates of the vectors. For example, one might define the negation, addition, and multiplication operations as

```
negateV3 :: V3 Int -> V3 Int
negateV3 (V3 x y z)  = V3 (negate x) (negate y) (negate z)
addV3 :: V3 Int -> V3 Int -> V3 Int
addV3 (V3 x1 y1 z1) (V3 x2 y2 z2) = V3 (x1 + x2) (y1 + y2) (z1 + z2)
mulV3 :: V3 Int -> V3 Int -> V3 Int
mulV3 (V3 x1 y1 z1) (V3 x2 y2 z2) = V3 (x1 * x2) (y1 * y2) (z1 * z2)
```

which show the following behavior:

```
>>> negateV3 (V3 2 (-4) 8)
V3 (-2) 4 (-8)
>>> addV3 (V3 1 2 3) (V3 10 20 30)
V3 11 22 33
```

The above code clearly contains duplication: the concept of "lifting" a binary operator on numbers pointwise to a binary operator on vectors is repeated in `addV3` and `mulV3`. Similarly, it is the case for unary operators, like `negate`.

Write two functions `mapV3` and `liftV3` which lift a unary and binary operator on numbers to vectors, and use them to implement negation, addition, subtraction, multiplication, and division for `V3 Int`.

Their type signatures are:

```
mapV3  :: (a -> b) -> V3 a -> V3 b
liftV3 :: (a -> b -> c) -> V3 a -> V3 b -> V3 c
```

Exercise 3 (Tic Tac Toe)

In this exercise we are going to implement the classic game Tic-Tac-Toe.

As we did not learn about the `IO` monad yet, which is used in Haskell to represent side effects like printing to or reading from the terminal, we have already written the user interaction code for you. You can download the code template from the link on the website next to the link to this exercise sheet.

First, we need to decide what kind of data types we use to represent the game state. We propose the following representation:

```
data Token = X | O | E deriving (Show, Eq)

type Board = [[Token]]
```

A `Board` is a matrix of `Token`s, and a `Token` can be either `X` or `O`, for the corresponding players, or `E` (for empty), if there is no `Token` yet at this position.

As an example, the following defines a board where player `X` has won the game:

```
exampleBoard :: Board
exampleBoard =
  [ [ X, O, E ]
  , [ X, O, E ]
  , [ X, E, E ]
  ]
```

Once you have finished the exercise, you can run the game with `cabal run` as followed:

```
$ cabal run
_ _ _
_ _ _
_ _ _

Where to place X? (1, 0)

_ X _
_ _ _
_ _ _

Where to place O? [...]
```

Complete the Tic-Tac-Toe game by implementing the following functions:

1. The function

   ```
   emptyBoard :: Int -> Board
   ```

   takes a number n and returns an empty $n \times n$ board.

   Examples:

```
>>> emptyBoard 3
[[E,E,E],[E,E,E],[E,E,E]]
>>> emptyBoard 4
[[E,E,E,E],[E,E,E,E],[E,E,E,E],[E,E,E,E]]
```

Hint: The function `replicate` is imported by default and creates a list by repeating a single element, e.g.

```
>>> replicate 5 'a'
"aaaaa"
```

2. The function

   ```
   isFull :: Board -> Bool
   ```

   should return whether the `Board` contains only `X` or `O` tokens, but no `E` tokens.

   This function is used by `main` to determine whether the game is a draw (game over but neither player has won) in conjunction with other functions.

3. The function

   ```
   showBoard :: Board -> String
   ```

   takes a board and converts it to a string suitable for displaying it on the terminal for the user.

   It should display an empty token `E` as an underscore.

   Examples:

   ```
   >>> showBoard exampleBoard
   "X O _\nX O _\nX _ _"
   >>> putStrLn (showBoard exampleBoard)
   X O _
   X O _
   X _ _
   ```

   Hint: You might find the `intercalate` function from `Data.List` useful.

4. The function

   ```
   setToken :: (Int, Int) -> Token -> Board -> Maybe Board
   ```

   should update the board by placing a token at a position on the board.

   In a pure functional language, like Haskell, updating the board means returning a new board, which is similar to the old board.

   In Tic-Tac-Toe, updating the board in this way is only allowed, if the token at the given position is still an empty token `E`, i.e. if no player placed an `X` or `O` at this position yet. This is why the `setToken` function has `Maybe Board` as a return type: It either returns `Just` an updated board, or `Nothing` if the update failed because the token at the position is not an empty token `E` or because the position is not a valid index for the board.

   Examples:

```
>>> setToken (1, 0) X (emptyBoard 3)
Just [[E,X,E],[E,E,E],[E,E,E]]
>>> setToken (1, 0) O [[E,X,E],[E,E,E],[E,E,E]]
Nothing
>>> setToken (55, 32) X (emptyBoard 3)
Nothing
```

Hint: Implementing this function is much easier, if you first define a function to update regular lists. Consider the following function type:

```
mapAt :: Int -> (a -> a) -> [a] -> [a]
```

The idea is that this function is like the `map` function, but only applies its function argument to a single element at a specific index. For example:

```
>>> map (\x -> x * 100) [1, 2, 3, 4]
[100, 200, 300, 400]
>>> mapAt 1 (\x -> x * 100) [1, 2, 3, 4]
[1, 200, 3, 4]
```

What we need for `setToken` is a little bit more complex, because updating the board might also fail, e.g. if the board is not empty at the given position. To allow the update function to fail, we can slightly generalize `mapAt` as follows:

```
mapAtM :: Int -> (a -> Maybe a) -> [a] -> Maybe [a]
```

The idea is that, if we call `mapAtM i f xs`, the function `f` can either return `Just` a new element, or `Nothing` to signal that the update failed. If `f` returns `Nothing`, then `mapAtM i f xs` should also return `Nothing` to propagte the failure to the outside.

Examples:

```
>>> f x = if x > 1 then Just (x * 100) else Nothing
>>> mapAtM 1 f [1, 2, 3, 4]
Just [1, 200, 3, 4]
>>> mapAtM 0 f [1, 2, 3, 4]
Nothing
```

The `setToken` function can be implemented very concisely by using `mapAtM` twice.

Hint: To implement `mapAtM` you might find it useful to map over values of type `Maybe [a]`. You can use the `fmap` function for that, which behaves like `map` for lists, but also works on `Maybe` values. (Think of `Maybe` as a list which always contains exactly 0 or 1 elements.)

5. The function

   ```
   winner :: Board -> Token
   ```

   should return `X` or `O`, if `X` or `O` has won the game, and otherwise `E`.

   You do not need to check for draws, because the `isFull` function already covers that case.

   Examples:

```
>>> winner (emptyBoard 3)
E
>>> winner [ [X,X,X]
           , [E,E,E]
           , [O,O,E] ]
X
>>> winner [ [O,E,E]
           , [E,O,E]
           , [X,X,O] ]
O
```

Hint: If you defined a function to check if a player got three tokens in one *row*, then you can easily define how to check if a player got three tokens in one *column* by using the `transpose` function from `Data.List`.

Hint: The `all` function from exercise 1 can be helpful in this exercise.