Note: The code itself is not commented very well as I found that most functions are rather simple and short and the entire file would be rather unreadable with many comments. I therefore chose to instead write more detailed comments here in the PDF.
The main function only tests the requested functions from question 1 and 5. However you can easily check all functions in the ghci environment.
Even if not requested, all functions are in the same file. (main.hs)

Exercise 3.1 Bite-sized Haskell Tasks

a) `Swap the values of the first and last element in a list`

```
2  swapFirstAndLast :: [a] -> [a]
3  swapFirstAndLast [] = []
4  swapFirstAndLast [x] = [x]
5  swapFirstAndLast xs = last xs : tail (init xs) ++ [head xs]
```

First check for the case that the list is empty, if so, return it. Then check the case if the list contains only a single item, if so, again, return it. Only in the third case, when the list contains 2 or more elements, swap the first and last.
`last xs` = last element of xs
`tail (init xs)` = all elements of xs without the first and last
`head xs` = first element of xs

b) `Return, as a Boolean, whether two consecutive elements in a list are the same`

```
8  containsEqualConsec :: Eq a => [a] -> Bool
9  containsEqualConsec [] = False
10 containsEqualConsec [_] = False
11 containsEqualConsec (x:y:xs) = x == y || containsEqualConsec(y:xs)
```

Same principle as in a). In the third case however we check if the first two elements are equal and if not, make a recursive call with the second and the third item. So we iterate recursively over the list until we find two equal elements next to each other. If so we return true. If not we will at some point end up with list with only one element and return false in the second case.

c) `Return a list containing all divisors of a given positive integer.`

```
14 getDivisors :: Int -> [Int]
15 getDivisors n = [x | x <- [1..n], n `mod` x == 0]
```

We use a generator to generate a list with numbers from 1 to n where n is our given Integer. We then constrain the generator to only generate x when n mod x equals 0.

d) `Remove all odd numbers from a list.`

```
18 removeOdd :: [Int] -> [Int]
19 removeOdd = filter even
```

Using the in-built filter function with the also in-built even function. We can easily filter our list for only even ones. which is the same as dropping the odd ones.

e) `Calculate the mean of the even numbers in a list`

```
22 calcMean :: [Int] -> Int
23 calcMean xs = sum (filter even xs) `div` length(filter even xs)
```

Divide the sum of all even numbers by the number of even numbers. We make use of the filter function just as in 1d) and also use the length function to get the number of even numbers.

f) `Generate a list of tuples (n, s) where 0 <= n <= 30 and where s = n2`

```
26 generateTupleList :: () -> [(Int, Int)]
27 generateTupleList () = [(x,x^2) | x <- [1..30]]
```

Use a list generator from 1 to 30. Code is trivial.

g) `Take two positive integers and return a list containing their shared prime factors.`

```
30 sharedPrimes :: Int -> Int -> [Int]
31 sharedPrimes n t = [ y | y <- [2..max n t], n `mod` y == 0, t `mod` y ==
       0, [x | x <- [1..y], y `mod` x == 0] == [1, y]]
```

The y generator generates numbers from two to the larger of the two inputs. This generator then has three constraints:

- y needs to be a divider of the first number n
- y needs to be a divider of the second number t
- y needs to be a prime number. Which is checked by comparing the list of all divisors of y with the list of only 1 and the number itself in it.

Exercise 3.2 Lazy Evaluation (Call by need / Infinite List)

a) `Write a function that returns an infinite list of natural even numbers`

```
34 infEvenN :: [Int]
35 infEvenN = [2,4..]
```

How the title already mentions, Haskell uses lazy evaluation which means that values are only calculated when needed. So when defining an infinite list, Haskell doesn't actually calculate an infinite amount of numbers. Only when working with the list, for example to retrieve some values from it, then those values are calculated.

b) `Take a list and return tuples with the result and the result times two.`

```
38 tupleList :: [Int] -> [(Int, Int)]
39 tupleList n = [(x, x*2) | x <- n]
```

Because Haskell uses lazy evaluation, this function works with finite lists just as well as with infinite lists.

c) `return elements where n*2 < given Integer`

```
42 anotherList :: [(Int, Int)] -> Int -> [(Int, Int)]
43 anotherList n s = takeWhile (\(_, y) -> y < s) n
```

Note that this implementation only works when using the lists generated from a) and b). If the input list is not ordered in ascending order, it might not return all values that match the constraint.

Exercise 3.3 Pattern Matching and Guards

1. `Three-argument ackermann function`

```
46 ackermann :: Int -> Int -> Int -> Int
47 ackermann m n 0 = m + n
48 ackermann m 0 1 = 0
49 ackermann m 0 2 = 1
50 ackermann m 0 p | p > 2 = m
51 ackermann m n p | n > 0, p > 0 = ackermann m (ackermann m (n-1) p) (p
       -1)
```

2. `Ackermann list function`

```
54 ackermannList :: [Int] -> [Int]
55 ackermannList [] = [13]
56 ackermannList [x] = [ackermann x 0 3]
57 ackermannList [x,y] = [ackermann x y 0]
58 ackermannList [x,y,z] = [ackermann x y z]
59 ackermannList (x:y:z:xs) = ackermann x y z : ackermannList xs
```

I am unsure on how to comment on this code. It's a simple function that takes three arguments (or a list) as input and then has different cases that it goes through. Depending on which case (or pattern) matches, it behaves differently.

Exercise 3.4 Sorting

1. `Return true or false whether the given list is sorted`

```
62 sorted :: (Ord a) => [a] -> Bool
63 sorted [] = True
64 sorted [_] = True
65 sorted (x:y:ys) = x <= y && sorted (y:ys)
```

Compares the first two items in a list, if sorted recursively call itself with the rest of the list except for the first item. If at any point it is not sorted, it returns false. If it reaches the end (list with one item) it will return true.

2. `Mergesort`

```
68 mergesort' :: (Ord a) => [a] -> [a]
69 mergesort' [] = []
70 mergesort' [x] = [x]
71 mergesort' xs = merge (mergesort' left) (mergesort' right)
72     where
73         (left, right) = splitAt(length xs `div` 2) xs
74
75         merge :: (Ord a) => [a] -> [a] -> [a]
76         merge [] r = r
77         merge l [] = l
78         merge (l:ls) (r:rs)
79             | last (l:ls) <= r = (l:ls)++(r:rs)
80             | last (r:rs) <= l = (r:rs)++(l:ls)
81             | l <= r    = l : merge ls (r:rs)
82             | otherwise = r : merge (l:ls) rs
83
84 mergesort :: (Ord a) => [a] -> [a]
85 mergesort xs
86     | sorted xs = xs
87     | otherwise = mergesort' xs
```

On the recursive "way down" we split our list in smaller lists until they are either empty or contain a single element. On the "way up" we then merge those lists together.

Since we only merge "sorted" lists, we can skip the merge part if the smallest value in one list is bigger then the biggest value in the other list. If so we can just append the two lists instead of merging. (Line 79-80)

To prevent unnecessary calculations we first check with the help of our sorted function if the given list is already sorted. If not, we continue with merge sort.

3. `Bubblesort`

```
89  bubbleSort :: (Ord a) => [a] -> [a]
90  bubbleSort xs = go xs
91    where
92      go :: (Ord a) => [a] -> [a]
93      go xs
94        | swapped   = go passResult
95        | otherwise = xs
96        where (swapped, passResult) = onePass xs
97
98      onePass :: (Ord a) => [a] -> (Bool, [a])
99      onePass [] = (False, [])
100     onePass [x] = (False, [x])
101     onePass (x:y:ys)
102       | x <= y    = (swappedTail, x : sortedTail)
103       | otherwise = (True, y : sortedTail')
104       where
105         (swappedTail, sortedTail) = onePass (y : ys)
106         (swappedTail', sortedTail') = onePass (x : ys)
```

The idea is to iterate over the list and compare the current value with the next one. If not sorted, we swap them. If we have swapped anything in one pass, we have to do it again until we are able to iterate over the list without swapping anything.

The onePass function returns a boolean next to the list in order to determine whether a swap happend or not. The code is very confusing and I was not able to write this entirely by myself without using the help of the internet.

Exercise 3.5, Recursion, Map and Fold

1. `filter`

```
myfilter :: (a -> Bool) -> [a] -> [a]
myfilter _ [] = []
myfilter p (x:xs)
  | p x = x : myfilter p xs -- if true, include and go next
  | otherwise = myfilter p xs -- if false, ignore and go next
```

`map`

```
mymap :: (a -> b) -> [a] -> [b]
mymap _ [] = [] -- if empty return empty
mymap f (x:xs) = f x : mymap f xs -- apply f and go next
```

While we're not allowed to use the in-built filter and map functions for our own implementation, I believe it is still allowed to just look up how those functions are implemented and then implement my own functions exactly the same way.

The Map function uses the given function on the first element of the list and then calls itself again with the rest of the list.

The Filter function checks if the first element returns true with the given function, if yes it adds it to the list, if not, it just continues.

2. `BMI-Lists`

```
bmi :: Double -> Double -> Double
bmi m l = m / (l*l) * 10000

isBad :: Double -> Bool
isBad x = x > 25 || x < 18.5

roundToOneDec :: Double -> Double
roundToOneDec x = fromIntegral (round (x*10))/10

bmilists :: [Double] -> [Double] -> ([Double], [Int])
bmilists m l = (bmis,badindex)
  where
  bmis = mymap roundToOneDec (zipWith bmi m l)
  badindex = [x | x <- [0..length bmis-1], isBad (bmis !! x)]
```

Sadly I did not find a way to use the myfilter function in a meaningful way. Since I already constrain the list when generating it, I don't have to filter it afterwards.

Exercise 3.6,7,8,9 Sorting

Sadly no crazy tictactoe implementation this time. Actually no implementation at all for those remaining exercises. There is just no time to learn an entire new language concept and language in just a week while also staying up to date in the other lectures... Sorry. (First Haskell lecture was on Friday the 5th and the main part was this last Friday.)