Recursive function definitions

3.1 Write your own recursive definitions for the following Prelude functions. Use different names in order to avoid clashes, e.g. define a function `myand` instead of `and`.

(a) `and :: [Bool] -> Bool` — test if all values are true;

(b) `or :: [Bool] -> Bool` — test if some value is true;

(c) `concat :: [[a]] -> [a]` — concatenate a list of lists;

(d) `replicate :: Int -> a -> [a]` — produce a list with repetead values;

(e) `(!!) :: [a] -> Int -> a` — index the n-th value in a list (starting from zero);

(f) `elem :: Eq a => a -> [a] -> Bool` — check if a value occurs in a list.

3.2 In this exercise we want to implement a prime number test that is more efficient than the one in Worksheet 2.

(a) Write a function `leastDiv :: Integer -> Integer` that computes the smallest divisor greater than 1 of a given number. We need only try candidate divisors d, i.e. numbers d such that $n = d \times k$. However, if $d \geq \sqrt{n}$, then $k \leq \sqrt{n}$ is also a divisor. Hence the smallest divisor will always be less than or equal to $\sqrt{n}$.

(b) Use `leastDiv` to define a function `isPrimeFast :: Integer -> Bool` that checks primality: n is prime if $n > 1$ and the least divisor of n is n itself.

Test that the fast version gives the same results as the original slow one with some examples, e.g. numbers from 1 to 10.

3.3 The function `nub :: Eq a => [a] -> [a]` from the `Data.List` module eliminates repeated ocorrences of values in a list.

For example: `nub "banana" = "ban"`.

Write a recursive definition for this function; because the function is not in the Prelude, you don't need to use a different name.

3.4 Write a definition of the function `intersperse :: a -> [a] -> [a]` from the `Data.List` module that intercalates a value between elements of a list.

Examples:

```
intersperse 0 [1,2,3] = [1,0,2,0,3]
intersperse 0 [1] = [1]
intersperse 0 [] = []
```

Hint: use recursion with pattern matching; you need only consider 3 distinct cases.

3.5 Sorting a list using the **insertion sort** algorithm.

(a) Write a recursive definition of the function `insert :: Ord a => a -> [a] -> [a]` that inserts a value into an ordered list the maintaining the ascending order.
Example: `insert 2 [0,1,3,5] = [0,1,2,3,5]`.

(b) Using insert, escreva a recursive definition of the function `isort :: Ord a => [a] -> [a]` that sorts the list using the insertion method:

- the empty list `[]` is trivially sorted;
- to sort a non-empty list, we first recursively sort the tail and then insert the head into the correct position.

3.6 Sorting a list using the **merge sort** algorithm.

(a) Write a recursive definition of a function `merge :: Ord a => [a] -> [a] -> [a]` that joins two sorted lists into a single sorted list. Example: `merge [3,5,7] [1,2,4,6] = [1,2,3,4,5,6,7]`. *Because the arguments lists are already sorted, you can do this with a single pass over both lists.*

(b) Using the `merge`, write a recursive definition of the function `msort :: Ord a => [a] -> [a]` that implements the *merge sort* algorithm:

- an empty list or with a single value is trivally sorted;
- to sort a list with dois or more elements, we first split the list into two halves, recursively sort both halves and join the result using `merge`.

Because we always split the list in half in each step, this algorithm has better complexity that insert sort: it runs in $O(n \log n)$ steps rather of $O(n^2)$.

3.7 Write a definition a of the function `toBits :: Int -> [Int]` that converts an integer into a binary representation, i.e. a list of bits (0 or 1). Example: `toBits 29 = [1,1,1,0,1]`. Note the bits in the result should be in most-significant to least-significant order.

Hint: Define an auxiliary function to obtain bits using the remainders of successive divisions by 2. To obtain the "correct" order back, just reverse the resulting list.

3.8 Write a definition of the function `fromBits :: [Int] -> Int` that performs the inverse transformation of the previous one, that is, converts bits in a binary representation to the corresponding integer.

Hint: this exercise is easier if you think about what numeric operation correspond to a *left-shift* in binary.

Higher order functions

3.9 Re-write the following definition of a function to compute all positive divisors of an integer using `filter` instead of a list comprehension.

```
divisors :: Integer > [Integer]
divisors n = [d | d<-[1..n], n‘mod‘d == 0]
```

3.10 Let us write the `isPrimeFast :: Integer -> Bool` function using higher-order functions instead of recursion. Recall that n is prime if and only if n is greater than 1 and no number in the range between 2 and $\lfloor\sqrt{n}\rfloor$ is a divisor of n.

Use the higher-order function `all` to express the "no number in the range..." part of the above condition. To compute the integer part of the square root you can use `floor (sqrt (fromIntegral n))`.

3.11 Write alternative definitions for the following Prelude functions. You should given them diferent names to avoid clashes, e.g. `myappend` instead of `++`.

(a) `(++) :: [a] -> [a] -> [a]`, using `foldr`;

(b) `concat :: [[a]] -> [a]`, using `foldr`;

(c) `reverse :: [a] -> [a]`, using `foldr`;

(d) `reverse :: [a] -> [a]`, using `foldl`;

(e) `elem :: Eq a => a -> [a] -> Bool`, using `any`.

Sugestion: use the diagrams in Lecture slides for understanding the fold operations as structural transformations on lists. Then all you have to do is solve for some unknown function f and initial value z.

3.12 Define the `fromBits :: [Int] -> Int` of Exercise 3.8 that converts a list of bits into the corresponding integer using `foldl` instead of recursion.

Example: `fromBits [1,1,0,1] = 13`.

3.13 Write a recursive definition of the function `group :: Eq a => [a] -> [[a]]` that breaks a list into groups of consecutive equal values.

Example: `group "AAABBACCC" = ["AAA", "BB", "A", "CCC"]`.

Hint: you may use the Prelude functions `takeWhile` and `dropWhile` to split the first consecutive run of identical elements from the argument list.

3.14 Permutations of a list.

(a) First define an auxiliary function `intercalate :: a -> [a] -> [[a]]` that computes all possible lists with a value in some position of the given list.
Example: `intercalate 0 [1,2] = [[0,1,2],[1,0,2],[1,2,0]]`

(b) Write a recursive definition of a function `permutations :: [a] -> [[a]]` that computes all permutations using the above function. Example:

```
  permutations [1,2,3]
= [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```