Functional programming

Excercises

Basics

1. Give another possible calculation for the function application *double (double 2)* from the sheets.
2. Show that `sum [x] = x` for any number *x*.
3. Define a function *product* that produces the product of a list of numbers, and show, using a calculation like in ex. 1, that `product [2,3,4] = 24`.
4. In the demonstration you saw the `qsort` implementation of quicksort. How should this definition be modified to produce the result in reverse sorted order?
5. What would the effect of changing <= into < be in the definition of quicksort? Think of what is allowed in lists.
6. `last` returns the last element of a non-empty list. Define this function in terms of prelude list functions
7. `init` removes the last element of a non-empty list. Define this function in terms of prelude list functions

Types and classes

1. What are the types of the following values:
   ```
   [‘a’, ‘b’, ’c’]
   (‘a’, ‘b’, ’c’)
   [(False, ‘O’), (True, ‘1’)]
   ([False, True],[‘0’,’1’])
   [tail, init, reverse]
   ```
2. Write definitions for functions with the following types. It is not really important what they do, as long as they have the specified types!
   ```
   bools :: [Bool]
   nums :: [[Int]]
   add :: Int -> Int -> Int -> Int
   copy :: a -> (a,a)
   ```
3. What are the types of the following functions (think about type variables and class constraints, and only use GHCi to validate your answers 😉):
   ```
   second xs = head (tail xs)
   swap (x, y) = (y, x)
   pair x y = (x, y)
   double x = x * 2
   palindrome xs = reverse xs == xs
   twice f x = f (f x)
   ```
4. Think about why it is not feasible for function types to be instances of the *Eq* type class. When would it be feasible? *Hint*: two functions of the same type are equal if they always return equal results for equal arguments.

Defining functions

1. Using prelude functions, define a function `halve :: [a]-> ([a], [a])` that splits an even-length list in two halves.

2. Define a function `third:: [a] -> a` that returns the third element in a list containing at least 3 elements using:
    a. *head* and *tail*
    b. !!
    c. Pattern matching
3. Consider a function `safetail :: [a] -> [a]` that behaves as the prelude function tail, except that it maps the empty list to itself, so it is safe to pass an empty list to safetail. tail produces an error in this case. Define safetail using:
    a. A conditional expression
    b. Guarded equations
    c. Pattern matching

    Hint: you can use the library function `null`, which returns `True` if the list passed as an argument to it is empty.
4. Define the disjunction operator `or’` using pattern matching. There are multiple solutions, try to find at least 2.
5. Redefine the following version of the conjunction operator *and’* using conditional expressions instead of pattern matching:
    ```
    and’ True True = True
    and’ _ _       = False
    ```
6. Do the same for the following versions:
    ```
    and’ True b  = b
    and’ False _ = False
    ```
    What do you notice?
7. Show how the curried function definition `mult x y z = x * y * z` can be understood in terms of lambda expressions
8. The *Luhn* algorithm is used to check credit card numbers for simple errors such as mistyping a digit. It works as follows:
    - Consider each digit a separate number
    - Moving left, double every other number from the second last
    - Subtract 9 from each number that is now greater than 9
    - Add all the resulting numbers together
    - If the total is divisible by 10, the number is valid

    Define a function `luhnDouble :: Int -> Int` that doubles a digit and subtracts 9 if the result is greater than 9.
    Using luhnDouble and the integer remainder function *mod*, define a function `luhn :: Int -> Int -> Int -> Int -> Bool` that decides if a four-digit card number is valid. For example:

    ```
    > luhn 1 7 8 4
    True

    > luhn 4 7 8 3
    False
    ```

List comprehensions

1. Using a list comprehension, give an expression that calculates the sum 1^2 + 2^2 + … + 100^2
2. Write the method `length` for a list using list comprehension

3. Suppose that a coordinate grid of size m x n is given by the list of all pairs (x,y) of integers such that 0 <= x <= m and 0 <= y <= n. Using a list comprehension, define a function `grid :: Int -> Int -> [(Int, Int)]` that returns a coordinate grid of the given size.
4. Using a list comprehension and the function grid from the previous exercise, write a function `square :: Int -> [(Int, Int)]` that returns a coordinate square of size n x n, excluding the diagonal from (0,0) to (n,n). For example:
   ```
   > square 2
   [(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)]
   ```
5. Write the method `replicate:: Int -> a -> [a]` that produces a list of identical elements using list comprehension. For example: `replicate 3 True` gives `[True, True, True]`
6. A triple (x, y, z) is Pythagorean if x^2 + y^2 = z^2. Using a list comprehension, define a functions `pyths :: Int -> [(Int, Int, Int)]` that returns the list of all pythagorean triples whoe are at most a given limit. For example `pyths 10` gives all Pythagorean triples for which x, y or z is at most 10.
7. A positive integer is *perfect* if it equals the sum of its factors, excluding the number itself. Using a list comprehension and the function factors, define a functions `perfects :: Int -> [Int]` that returns the list of all perfect numbers up to a given limit. For example *perfects 500* gives *[6, 28, 496]*
8. Show how the single comprehension `[(x,y) | x <- [1,2], y <- [4,5]]` with two generators can be expressed using two comprehensions with single generators. Hint: use the library method concat and nest one comprehension within the other.
9. The scalar product of two lists of integers xs and ys of length n is given by the sum of the products of corresponding integers: $\sum_{i=0}^{n-1}(xs_i * ys_i)$. Write a function `scalarproduct :: [Int] -> [Int] -> Int` that returns the scalar product.
10. (If we get this far) modify the Caesar cipher demonstration program to handle upper-case letters.

Recursion

1. Write a recursive function `sumdown :: Int -> Int` that returns the sum of the non-negative integers from a given value down to zero. For example sumdown 3 should give 3 + 2 + 1 + 0 = 6
2. Write a recursive implementation of the exponentiation operator ^
3. Define a recursive function `euclid :: Int -> Int -> Int` that implements Euclid's algorithm for finding the greatest common divisor for two non-negative integers. If the numbers are eual, this number is the result, otherwise, the smaller number of the two is subtracted from the larger, and the process is repeated recursively with the smaller number and the result of the subtraction. For example:
   ```
   > euclid 6 27
   3
   ```
4. Write recursive implementations for the functions *length, drop,* and *init*.
5. Write recursive implementations for the following Prelude operations, without looking at their implementation! If you don't know what they do, try them out first. Often the name + type signature is a good hint.
   a. `and :: [Bool] -> Bool`
   b. `concat :: [[a]] -> [a]`
   c. `replicate :: Int -> a -> [a]`

    d. `(!!) :: [a] -> Int -> a`
    e. `elem :: Eq a => a -> [a] -> Bool`

6. Define a recursive function `merge :: Ord a => [a] -> [a] -> [a]` that merges two sorted lists into a single sorted list. For example: merge [2, 5, 6] [1, 3, 4] gives [1,2,3,4,5,6]. Don't use any library functions that sort to solve this!
7. Using *merge* define a function `msort :: Ord a => [a] -> [a]` that implements *merge sort*, in which the empty list and singleton list are already sorted, and any other list is sorted by merging together the two halves of the list separately. Hint: first define a function `halve :: [a] -> ([a], [a])` that splits a list into two halves whose lengths differ by at most one.
8. Write recursive definitions for the library functions `sum`, `take` and `last`