Prof. Dr. J. Giesl M. Hark

Notes:

- To solve the programming exercises you should use the Glasgow Haskell Compiler **GHC**, available for free at `https://www.haskell.org/ghc/`. You can use the command "`ghci`" to start an interactive interpreter shell.
- Please register at `RWTHonline` for the Exercise Course until Friday, April 19.
- Please solve these exercises in **groups of four**!
- The solutions must be handed in **directly before (very latest: at the beginning of)** the exercise course on Wednesday, 24.04.2019, 14:30, in lecture hall **AH I**. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl's office (until 30 minutes before the exercise course starts).
- In addition, please upload the solutions for programming exercises in a single **ZIP**-archive via `RWTHmoodle`. Please name your archive `Sheet_i_Mat1_Mat2_Mat3_Mat4.zip`, where `i` is the number of the sheet and `Mat_1`...`Mat_4` are the immatriculation numbers of the group members. It is sufficient if **one** of the group members uploads your solution. Files, which are not accepted by **GHC**, will not be marked.
- Please write the **names** and **immatriculation numbers** of all students on your solution. Also please staple the individual sheets!
- Exercises or exercise parts marked with a star are voluntary challenge exercises with advanced difficulty. They do not contribute to the points you need for taking part in the final exam. Nevertheless, the points that you achieve in these exercises are added to your score.

Exercise 1 (Function types): ((0.5 + 0.5 + 1) + 1 = 3 points)

a) Give examples of Haskell function declarations with the following types and briefly explain their semantics. Your solutions **must not** ignore any of their arguments completely.

i) `Bool -> Bool -> Int`

ii) `[Int] -> [Bool] -> Int`

iii) `[Bool] -> (Bool -> Int) -> [Int]`

b) Suppose that `f` has the type `Bool -> [Int] -> Int`.

What is the type of `\x y -> f ((f True x)>0) [y]`?

Exercise 2 (Lists): (2 + 3 = 5 points)

a) For each of the following equations, if possible, give pairwise different example values for `x`, `y`, and `z` such that the equation holds. Otherwise explain why such an assignment is not possible.

i) `[[x],[y]]` == `[y]:z`

ii) `([x] ++ z):y` == `(x:z):y`

iii) `[[]] ++ ([x]:y)` == `([x]:z)`

iv) `(x:y):z` == `(y ++ [x]):z`

The operator `++` concatenates two lists. For example:
`[1, 2, 3] ++ [2, 3] = [1, 2, 3, 2, 3]`.

b) Consider the following patterns

p1) `([x]++y):ys`

p2) `(x:y)++ys`

and the following terms:

t1) `[[]]`

t2) `[[1,2],[3]]`

For each pair of a pattern and a term, indicate whether the pattern matches the term. If so, provide the appropriate matching substitution. Otherwise, explain why the pattern does not match the term.

Does there exist a term that is matched by p1 but not by p2? Justify your answer.

Exercise 3 (Programming): (2 + 2 + 3 + 3 = 10 points)

Note that you may use constructors like `[]`, `:`, `True`, `False` in all of the following subexercises. You may also write auxiliary functions if needed or reuse functions from earlier subexercises (even if you did not manage to implement them).

a) Write a Haskell–function `myrem`, where `myrem x y` is the remainder of the integer division when dividing `x` by `y`. So for example, `myrem 14 3 == 2`. If `y == 0` then `myrem x 0 == x`. If `y < 0` then `myrem x y == myrem x (-y)`.

```
myrem :: Int -> Int -> Int
```

You may not use any predefined functions except comparisons, `+`, and `-`.

b) Write a Haskell-function `count` that given a list `xs` and an element `x` returns the number of occurences of `x` in `xs`. E.g., `count 2 [0,2,2,0,2,5,0,2] == 4` wheras `count (-7) [0,2,2,0,2,5,0,2] == 0`.

```
count :: Int -> [Int] -> Int
```

You may not use any predefined functions except comparisons and `+`.

c) Write a Haskell-function `simplify` that given a list `xs` returns a list of pairs as follows. The resulting list contains the pair `(x,n)` if and only if `x` occurs in `xs` `n` times and `n > 0`. E.g., `simplify [0,2,2,0,2,5,0,2] == [(0,3),(2,4),(5,1)]`.

```
simplify :: [Int] -> [(Int,Int)]
```

You may not use any predefined functions except comparisons.

d) Write a Haskell-function `multUnion` that given two lists of pairs `xs` and `ys` concatenates these lists where each “multiple occurence” is simplified as follows: If `xs` contains a pair `(x,n)` and `ys` contains `(x,m)`, then the result contains `(x,n+m)`. You may assume that in both `xs` and `ys` an integer occurs at most once as first entry of a pair. Moreover, assume that the lists are sorted in ascending order w.r.t. the first entry of the pair. Make sure that the resulting list is sorted in ascending order w.r.t. the first entry of the pair as well. E.g., `multUnion[(0,3),(2,4),(5,1)] [(-1,1),(0,4)] == [(-1,1),(0,7),(2,4),(5,1)]`.

```
multUnion :: [(Int,Int)] -> [(Int,Int)] -> [(Int,Int)]
```

You may not use any predefined functions except comparisons and `+`.

Exercise 4 (Infix Operators): (2+1* points)

Define a Haskell function `^^^` in infix notation with the type declaration

```
(^^^) ::  [Int] -> [Int] -> Int
```

such that the following holds for lists of equal length:

- The function call `xs ^^^ ys` evaluates to `xs` to the power of `ys` interpreted as vectors, where the negative entries of `ys` are ignored. In other words, `[x1, x2, ..., xn] ^^^ [y1, y2, ..., yn] == x1 ^ y1 * x2 ^ y2 * ...  * xn ^ yn`. For example `[1, 4, 5] ^^^ [7, 2, 3]` evaluates to `1 ^ 7 * 4 ^ 2 * 5 ^ 3 == 2000` and `[1, 4, 5] ^^^ [5, -1, 0]` evaluates to `1 ^ 5 * 5 ^ 0 == 1`.
- `xs ^^^ ys * z`, where `xs` and `ys` have type `[Int]` and `z` has type `Int`, is a valid expression.

The function `^^^` may behave arbitrarily if the two arguments have different lengths. You may not use any predefined functions except `*`, `^`, and comparisons. You may, of course, use constructors like `[]` and `:`.
You can get one bonus point if you solve the exercise even without using the predefined function `^`.

Hints:

- The binding priority of `*` is 7.
- The empty product defaults to `1`, i.e. `[] ^^^ [] == 1`.
- Note that `0 ^ 0 == 1`.