Prof. Dr. J. Giesl D. Korzeniewski

Notes:

- To solve the programming exercises you should use the Glasgow Haskell Compiler **GHC**, available for free at `http://www.haskell.org/ghc/`. You can use the command "`ghci`" to start an interactive interpreter shell.
- Please register at `https://aprove.informatik.rwth-aachen.de/fp16/` (https, not http!) until Wednesday, April 20.
- 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 Tuesday, 26.04.2016, 12:15pm, in lecture hall **AH 4**. 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 send the solutions to programming exercises to lehre_lufgi2@cs.rwth-aachen.de.
- Please write the **names** and **immatriculation numbers** of all students on your solution. Also please staple the individual sheets!

Exercise 1 (Function types): (1.5 + 1.5 = 3 points)

a) Give examples of Haskell function declarations with the following types and briefly explain their semantics:

i) `Int -> Int -> [Int]`

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

iii) `[a] -> Int -> [a]`

b) Suppose that `f` has the type `Int -> Int -> Bool` and that `g` has the type `(Int -> Bool) -> [Int] -> Int`. Let `h` be defined by `h x y = f (g (f y) x)`. What is the type of `h`?

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

a) Indicate for which of the following equations there exist pairwise different values of `x`, `y`, and `z`, such that the equation holds.

Give example values of `x`, `y`, and `z` or explain why such an assignment is not possible.

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

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

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

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

p2) `x:y:ys`

and the following terms:

t1) `[[1,2,3]]`

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

t3) `[[1,2], []]`

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.

Exercise 3 (Programming): (2 + 2 + 3 + 2 = 9 points)

Note that you may use constructors like `[]`, `:`, `True`, `False` in all of the following subexercises. You may also write auxiliary functions if needed.

a) Write a Haskell function that computes the digit sum of a natural number

```
digitSum :: Int -> Int
```

For example, `digitSum 5` and `digitSum 140` should both yield `5` (since $1 + 4 + 0 = 5$) and `digitSum 327` should result in `12` (since $3+2+7 = 12$). The function may behave arbitrarily on negative integers.

You may not use any predefined functions except comparisons, `+`, `div` (integer division), and `mod` (modulo).

b) Write a Haskell function that checks if a given list of integers is sorted in ascending order.

```
isSorted :: [Int] -> Bool
```

For example, `isSorted [-2, 0, 5, 5, 23] == True` and `isSorted [5, 7, 2, 1] == False`.

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

c) Write a Haskell function that computes the intersection of two unsorted lists, i.e., a list containing all elements that are contained in both lists.

```
intersect :: [Int] -> [Int] -> [Int]
```

For example, `intersect [0,-1] [9,0,3,-1]` yields `[0, -1]`, and `intersect [4,3,7] [5,3,4]` yields `[4, 3]`. The order of the elements may be arbitrary and the function may behave arbitrarily on lists containing duplicates.

You may not use any predefined functions except comparisons.

d) Write a Haskell function that computes the intersection of two sorted and duplicate-free lists, that is more efficient than the function from c) by using the fact that the lists are sorted in ascending order.

```
sortedIntersect :: [Int] -> [Int] -> [Int]
```

You may not use any predefined functions except comparisons.

Exercise 4 (Infix Operators): (2 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 the inner product of `xs` and `ys` interpreted as vectors, i.e `[x1, x2, ..., xn] *+ [y1, y2, ..., yn] == x1*y1 + x2*y2 + ... + xn*yn`. For example `[1, 4, 5] *+ [3, 2, 7]` evaluates to `46` and `[4, -2, 6, 1] *+ [2, 3, 4, 7]` to `33`.
- `xs *+ ys *+ zs` is the same as `(xs *+ ys) *+ zs`, and

- `x + ys *+ zs` is the same as `x + (ys *+ zs)`.

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 `:`.
Note that the binding priority of `+` is 6.