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 solve these exercises in **groups of four**!
- The solutions must be handed in *on paper* **directly before (very latest: at the beginning of)** the exercise course on Tuesday, 10.05.2016, 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!
- 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.
- You may use pre-defined functions, but no pre-defined modules except `Prelude`. At `https://hackage.haskell.org/package/base-4.8.2.0/docs/Prelude.html` you can find the documentation of `Prelude`.

Exercise 1 (Lazy Evaluation): (2 + 2 = 4 points)

a) Implement the infinite list `fibs :: [Int]` containing the Fibonacci sequence $1, 1, 2, 3, 5, \ldots$. The implementation should compute the first n Fibonacci numbers in linear time $\mathcal{O}(n)$.

Furthermore implement a function `fib :: Int -> Int` such that for any `n` ≥ 0, `fib n` returns the n^{th} Fibonacci number, also in linear time. Here the 0th Fibonacci number is 1 and the 4th Fibonacci number is 5.

b) Implement the infinite list `rands :: [Int]` of pseudo-random numbers following the linear congruential method. For that, we need parameters m (the *modulus*), a (the *multiplier*), and c (the *increment*). Then, for any given number r_n, the next pseudo-random number can be computed using the formula $r_{n+1} = ((a \cdot r_n) + c)\ mod\ m$.[1] Use $m = 2^{16}$, $a = 25713$ and $c = 13849$ and 42 as first random number.

Furthermore, define a function `randSum :: Int -> Int` such that `randSum k` yields the sum of the first `k` elements of `rands`.

Exercise 2 (List Comprehensions): (2 + 2 = 4 points)

In this exercise you *may not* write auxiliary functions and you *may not* use `where` or `let`.

a) Consider the following definition for an infinite list containing all primes from the lecture:

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = [y | y <- xs, y `mod` x /= 0]

dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)

primes :: [Int]
primes = dropall [2 ..]
```

[1] Here, *mod* describes the modulo operation, i.e, the remainder of division of one number by another.

Write a function `primeFactors :: Int -> [Int]` that returns a list of all prime factors of a given natural number. The list must contain each prime factor exactly once, even if it appears multiple times in the factorization of the number. For example, `primeFactors 12` should return `[2,3]`.

You may only use *one* defining equation and the right-hand side *must* be a single list comprehension. [2]

Hints:

Only test your implementation on relatively small numbers! Factorization is hard and this implementation is not the most efficient.

b) Consider the following incomplete implementation of the *merge sort* algorithm:

```
merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x < y     = x:(merge xs (y:ys))
                    | otherwise = y:(merge (x:xs) ys)
merge xs ys = xs ++ ys

mergeSort :: Ord a => [a] -> [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = merge
                 (mergeSort (splitList True xs))
                 (mergeSort (splitList False xs))
```

Write a function `splitList :: Bool -> [a] -> [a]` to complete the implementation. Intuitively, `splitList` returns one half of the list. The Boolean argument determines which of the two halves is returned. The function `splitList` must satisfy the following conditions:

- `splitList True myList ++ splitList False myList` should contain all elements from `myList`. (Not necessarily in the same order.)
- `length (splitList b myList) <= (length myList) div 2 + 1` is `True` for any list `myList` and any Boolean value `b`.
- You may only use *one* defining equation. The right-hand side *must* be a single list comprehension.
- You may use any built-in function[3] *except* `take`, `drop`, `span`, `break`, `splitAt`, or variants[4] of these functions.

Exercise 3 (IO): (2 + 2 + 4 + 1 + 2* = 11 points)

In this exercise an interactive program simulating a simple desktop calculator should be implemented. Consider the following example run:

```
*Main> main
Welcome to the simple Haskell calculator!
0
> 5
5
> +
> 25
30
> -
> /
> 15
```

[2] You may use one additional defining equation, such that there is one equation for numbers greater than 1 and one for numbers ≤ 1. For the case ≤ 1, the right-hand side should be `[]`. However, this is optional and we will only test your implementation on integers ≥ 2.

[3] From the module `Prelude`

[4] like `takeWhile`, etc.

```
2
> a
Invalid Input: a
2
> +
> 3
5
> quit
Bye!
*Main>
```

The calculator has two registers: `ans`, storing the last result, and `op`, storing the currently used operator. After displaying the welcome message, the operator is set to identity and the last result to `0`. That last result is displayed to the user. Then an interactive loop starts where the user is prompted for input and the simple calculator reacts to it. If the user

- enters a number `n`: `op ans n` gets evaluated. The result is displayed, the new operator is identity and the new `ans` is `op ans n`
- enters an operator `op'`: `op` is updated to `op'`, `ans` stays the same
- enters a `quit` command: The interactive loop terminates.
- enters something else: An error message is displayed, then `ans` is displayed again and `ans` and `op` stay the same.

After the interactive loop terminates, a goodbye message is displayed and the whole program terminates.
A framework for the implementation is given in the file `calculator.hs`. Only edit the parts between the comments `replace with implementation:` and `end replace`. Do not change any other code.

a) Implement the function `main` that first displays a welcome message, then evaluates `simpleCalculator 0 (\_ y -> y)` and then displays a goodbye message.

b) Implement the function `getInput` that displays a prompt `>`, then reads a line from standard input, and finally returns a `CalculatorInput` computed from the just-read string using the given function `parseCalculatorInput`.

c) Implement the function `simpleCalculator`. It should first ask the user for input using `getInput` and then take an appropriate action, depending on the input as described above.

d) Explain which concept prohibits that we change the `main` function such that it returns an `Int` representing the number of computations the user has performed.

e)* Could we nevertheless change the code such that different goodbye messages are displayed in the function `main`, depending on the number of computations the user has performed? Justify your answer!

Hints:

To output a line of text use `putStrLn :: String -> IO ()`, to read a line use `getLine :: IO String`.