Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

---

# Functional Programming

SS 2017

Benjamin Dietrich

## Assignment #1

Submission Deadline: Thu, 4.5.2017

**Exercise 1: Types** **(5 Points)**

Please answer the following questions on Haskell's type system.

1. Consider the following types

   (a) `a -> b -> c -> d`

   (b) `a -> (b -> c) -> d`

   (c) `a -> b -> (c -> d)`

   Which pairs of types are equivalent and which are not? Explain.

2. Can you give multiple definitions of a function of type `(a, b) -> a` that **behave differently**, that is, return different values for the same argument? Assume that your function actually has to return a value (i.e. no crashes, no infinite loops and recursions).

3. Consider a function of type `[a] -> a` that returns a value (no crash, no infinite loop or recursion). Please answer the following questions based on the type.

   (a) Can this function find the largest element in the list?

   (b) Can it compute the sum of the elements?

   (c) Can the function be a *constant* function?

   (d) Can the function perform I/O operations?

   Explain your answers.

4. The following function is supposed to extract the first character from a given string.

   ```
   getFirstLetter :: [Char] -> [Char]
   getFirstLetter = head
   ```

   Fix any type errors.

**Exercise 2: Finger Exercises** **(10 Points)**

1. Define an infix operator that implements logical implication. You can use a conditional expression (`if ... then ... else`) or the boolean operators `(&&)`, `(||)`, `not`. The new operator's precedence should be less than the three boolean operators' above. Please give some example expressions to show this behavior.

   ```
   (==>) :: Bool -> Bool -> Bool
   ```

2. Define a function `distance` to calculate the Euclidean distance between to Points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$.

   ```
   distance :: (Int,Int) -> (Int,Int) -> Double
   ```

   **Note:** To avoid repetitive code or extract meaningful code pieces it can be very useful to **introduce local definitions** which can be used in the function body. In Haskell such definitions are declared with the keyword `where`. For example if we have:

   ```
   perimeter :: (Int, Int) -> Int
   perimeter rectangle = 2 * (fst rectangle) + 2 * (snd rectangle)
   ```

   we can, instead, write

   ```
   perimeter :: (Int, Int) -> Int
   perimeter rectangle = double width + double height
       where
         width = fst rectangle
         height = snd rectangle
         double n = 2 * n
   ```

   Try to use `where`-definitions in your solution for `distance`.

   **If you want an introduction and demonstration on `where`-definitions or have further questions, come to visit the tutorial on Tuesday!**

3. Write a function `gcdEuclid :: Int -> Int -> Int` that computes the greatest common divisor of two integers $i, j > 0$ using Euclid's algorithm. Use **guards** within your solution!

4. If $gcd(i, j) = 1$ for integers $i$ and $j$, then $i$ and $j$ are called *coprime*. Write a Haskell function `coprime :: Int -> Int -> Bool` to determine if two integers are coprime.

5. Write a Haskell function `phi :: Int -> Int` to compute Euler's totient function $\phi(m)$ for some positive integer $m$. $\phi(m)$ is defined as the number of positive integers equal to or smaller than $m$ that are *coprime* to $m$. Return an error message (using `error :: [Char] -> a`) if $m$ is not positive.