RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET
INFORMATIK 2
RWTH Aachen · D-52056 Aachen · GERMANY
http://verify.rwth-aachen.de/

LuFG Informatik II

Functional Programming
Exam, March 19, 2010

Prof. Dr. Jürgen Giesl
Carsten Fuhs

First name: ______________________________

Last name: ______________________________

Matr. number: ______________________________

Course of study (please mark exactly one):

○ Master of SSE

- On every sheet please give your **first name**, **last name**, and **matriculation number**.
- You must solve the exam **without** consulting any **extra documents** (e.g., course notes).
- Make sure your answers are readable. Do not use **red pens or pencils**.
- Please answer the exercises on the **exercise sheets**. If needed, also use the back sides of the exercise sheets.
- Answers on extra sheets can only be accepted if they are clearly marked with your name, your matriculation number, and the **exercise number**.
- **Cross out** text that should not be considered in the evaluation.
- Students that try to cheat **do not pass** the exam.
- At the end of the exam, please return **all sheets together with the exercise sheets**.

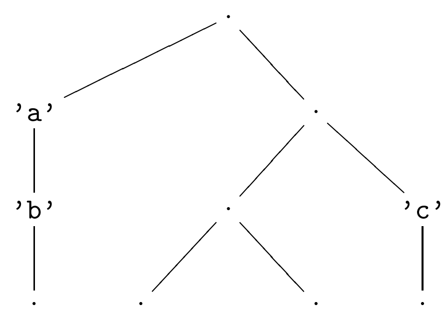
	Total number of points	Number of points obtained
Exercise 1	24	
Exercise 2	9	
Exercise 3	6	
Exercise 4	9	
Exercise 5	10	
Total	58	
Grade	-	

First name	Last name	Matriculation number

Exercise 1 (4 + 5 + 4 + 5 + 6 = 24 points)

The following data structure represents polymorphic binary trees that contain values only in special `Value` nodes that have a single successor:
`data Tree a = Leaf | Node (Tree a) (Tree a) | Value a (Tree a)`

Consider the tree `t` of characters on the right-hand side. The representation of `t` as an object of type `Tree Char` in Haskell would be:



`(Node (Value 'a' (Value 'b' Leaf)) (Node (Node Leaf Leaf) (Value 'c' Leaf)))`

Implement the following functions in Haskell.

(a) The function `foldTree` of type

`(a -> b -> b) -> (b -> b -> b) -> b -> Tree a -> b`

works as follows: `foldTree f g h x` replaces all occurrences of the constructor `Value` in the tree `x` by `f`, it replaces all occurrences of the constructor `Node` in `x` by `g`, and it replaces all occurrences of the constructor `Leaf` in `x` by `h`. So for the tree `t` above,

`foldTree (:) (++) [] t`

should compute

`((++) ((:) 'a' ((:) 'b' [])) ((++) ((++) [] []) ((:) 'c' [])))`,

which in the end results in `"abc"` (i.e., in the list `['a','b','c']`). Here, `Value` is replaced by `(:)`, `Node` is replaced by `(++)`, and `Leaf` is replaced by `[]`.

First name	Last name	Matriculation number

(b) Use the `foldTree` function from (a) to implement the `average` function which has the type `Tree Int -> Int` and returns the average of the values that are stored in the tree. This should be accomplished as follows:

- Use `foldTree` with suitable functions as arguments in order to compute the *sum* of the values stored in the trees.
- Use `foldTree` with suitable functions as arguments in order to compute the *number of* `Value`*-objects in the tree.*
- Perform integer division with the pre-defined function `div :: Int -> Int -> Int` on these values to obtain the result.

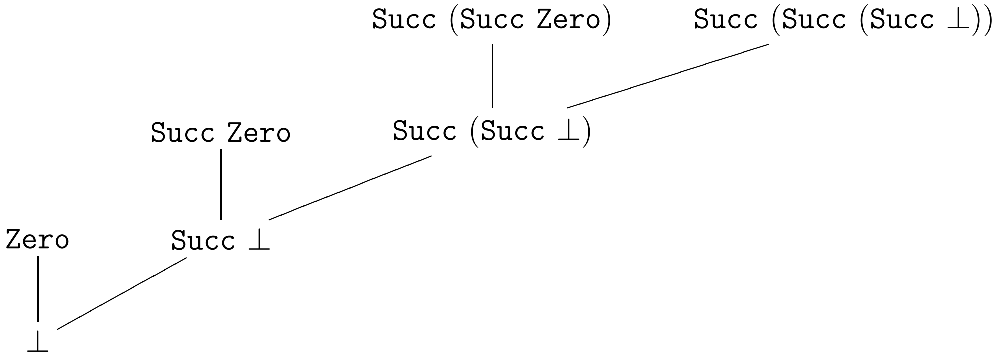
Here your function is required to work correctly only on those trees that contain the constructor `Value` at least once.

First name	Last name	Matriculation number

(c) Consider the following data type declaration for natural numbers:

```
data Nats = Zero | Succ Nats
```

A graphical representation of the first four levels of the domain for `Nats` could look like this:



Sketch a graphical representation of the first three levels of the domain for the data type `Tree Bool`.

First name	Last name	Matriculation number

(d) Write a Haskell function `printStars` that first reads a string from the user, then prints this string on the console, converts the string to a number `n` (using the pre-defined function `read`) and in the end also prints `n` times the character `'*'` on the console. Also give the type declaration for your function.

You may use the `do`-notation, but you are not obliged to use it. You do not have to check whether the input string is really a number. Some of the following pre-defined functions can be helpful:

– `getLine :: IO String` reads a string from the user

– `read :: String -> Int` converts a string to a number

– `putStr :: String -> IO ()` writes a string to the console

An example run should look as given below. Here the string "7" was read from the user.

```
Main> printStars
7
7*******
```

First name	Last name	Matriculation number

(e) We call a list `ys` of integers an `n`-*times even product* of a list `xs` if `ys` has length `n` and if all elements of `ys` are even numbers that occur in `xs`. The goal of this exercise is to write a function `evenProducts :: [Int] -> Int -> [[Int]]` that takes a list of integers `xs` and a natural number `n` and returns a list that contains all `n`-times even products of `xs`. For example, `evenProducts [4,5,6] 2 = [[4,4], [4,6], [6,4], [6,6]]`.

The following declarations are already given:

```
evenProducts :: [Int] -> Int -> [[Int]]
evenProducts xs 0     = []
evenProducts xs 1     = map (\z -> [z]) (filter even xs)
```

Please give the declaration of `evenProducts` for the missing case of numbers that are at least 2. Perform your implementation only with the help of a **list comprehension**, i.e., you should use exactly one declaration of the following form:

```
evenProducts xs (n+2) = [ ... | ... ]
```

First name	Last name	Matriculation number

Exercise 2 (4 + 5 = 9 points)

Consider the following Haskell declarations for the `fib` function, which for a natural number x computes the value *fibonacci*(x):

```
fib :: Int -> Int
fib 0     = 0
fib 1     = 1
fib (x+2) = fib (x+1) + fib x
```

(a) Please give the Haskell declarations for the higher-order function `f_fib` corresponding to `fib`, i.e., the higher-order function `f_fib` such that the least fixpoint of `f_fib` is `fib`. In addition to the function declaration(s), please also give the type declaration of `f_fib`. Since you may use full Haskell for `f_fib`, you do not need to translate `fib` into simple Haskell.

(b) We add the Haskell declaration `bot = bot`. For each $n \in \mathbb{N}$ please determine which function is computed by $\texttt{f\_fib}^n\ \texttt{bot}$. Here "$\texttt{f\_fib}^n\ \texttt{bot}$" represents the n-fold application of `f_fib` to `bot`, i.e., it is short for $\underbrace{\texttt{f\_fib (f\_fib} \ldots \texttt{(f\_fib}}_{n \text{ times}}\ \texttt{bot)} \ldots)$.

Let $f_n : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ be the function that is computed by $\texttt{f\_fib}^n\ \texttt{bot}$.
Give f_n in **closed form**, i.e., using a non-recursive definition. In this definition, you may use the function *fibonacci* $: \mathbb{N} \to \mathbb{N}$ where *fibonacci*(x) computes the x-th Fibonacci number. Here it suffices to give the result of your calculations. You do not need to present any intermediate steps.

First name	Last name	Matriculation number

Exercise 3 (3 + 3 = 6 points)

Let D_1, D_2, D_3 be domains with corresponding complete partial orders $\sqsubseteq_{D_1}, \sqsubseteq_{D_2}, \sqsubseteq_{D_3}$. As we know from the lecture, then also $\sqsubseteq_{(D_2 \times D_3)_\perp}$ is a complete partial order on $(D_2 \times D_3)_\perp$.

Now let $f : D_1 \to D_2$ and $g : D_1 \to D_3$ be functions.
We then define the function $h : D_1 \to (D_2 \times D_3)_\perp$ via $h(x) = (f(x), g(x))$.

(a) Prove or disprove: If f and g are *strict* functions, then also h is a strict function.

(b) Prove or disprove: If f and g are *monotonic* functions, then also h is a monotonic function.

First name	Last name	Matriculation number

9

Exercise 4 (4 + 5 = 9 points)

Consider the following data structure for polymorphic lists:

```
data List a = Nil | Cons a (List a)
```

(a) Please translate the following Haskell expression into an equivalent lambda term (e.g., using $\mathcal{L}am$). Recall that pre-defined functions like `odd` or `(+)` are translated into constants of the lambda calculus.

It suffices to give the result of the transformation.

```
let f = \x -> if (odd x) then (\y -> x) else f ((+) x 3)
    in f
```

First name	Last name	Matriculation number

(b) Let δ be the set of rules for evaluating the lambda terms resulting from Haskell, i.e., δ contains at least the following rules:

$$\begin{aligned} \texttt{fix} &\rightarrow \lambda f.\ f\ (\texttt{fix}\ f) \\ \texttt{times}\ 3\ 2 &\rightarrow 6 \end{aligned}$$

Now let the lambda term t be defined as follows:

$$t = (\lambda x.\,(\texttt{fix}\ \ \lambda g.\,x))\ \ (\lambda z.\,(\texttt{times}\ 3\ 2))$$

Please reduce the lambda term t by WHNO-reduction with the $\rightarrow_{\beta\delta}$-relation. You have to give **all** intermediate steps until you reach **weak head normal form** (and no further steps).

First name	Last name	Matriculation number

Exercise 5 (10 points)

Use the type inference algorithm $\mathcal{W}$ to determine the most general type of the following lambda term under the initial type assumption A_0. Show the results of all sub-computations and unifications, too. If the term is not well typed, show how and why the $\mathcal{W}$-algorithm detects this.

$$((\texttt{Cons}\ \ \lambda x.\, x)\ y)$$

The initial type assumption A_0 contains at least the following:

$$\begin{array}{lcl} A_0(\texttt{Cons}) & = & \forall a.\ (a \to (\texttt{List}\ a \to \texttt{List}\ a)) \\ A_0(x) & = & \forall a.\ a \\ A_0(y) & = & \forall a.\ a \end{array}$$