# Exam in Functional Programming SS 2012 (V3M)

**First Name:** ______________________

**Last Name:** ______________________

**Matriculation Number:** ______________________

**Course of Studies** (please mark **exactly** one):

- ○ Informatik Bachelor
- ○ Informatik Master
- ○ Other: ______________
- ○ Mathematik Master
- ○ Software Systems Engineering Master

| | Available Points | Achieved Points |
|---|---|---|
| **Exercise 1** | **20** | |
| **Exercise 2** | **38** | |
| **Exercise 3** | **37** | |
| **Exercise 4** | **20** | |
| **Exercise 5** | **6** | |
| **Sum** | **121** | |

Notes:

- **On all sheets** (including additional sheets) you must write **your first name, your last name and your matriculation number**.
- Give your answers in readable and understandable form.
- Use **permanent** pens. Do not use red or green pens and do not use pencils.
- Please write your answers on the exam sheets (also use the reverse sides).
- For each exercise part, give **at most one** solution. Cancel out everything else. Otherwise all solutions of the particular exercise part will give you **0 points**.
- If we observe any **attempt of deception**, the whole exam will be evaluated to **0 points**.
- At the end of the exam, hand in **all sheets together with the sheets containing the exam questions**.

## Exercise 1 (Quiz): (4 + 5 + 5 + 3 + 3 = 20 points)

Give a short proof sketch or a counterexample for each of the following statements:

**a)** Monotonic unary functions are always strict.

**b)** Strict unary functions on flat domains are always monotonic.

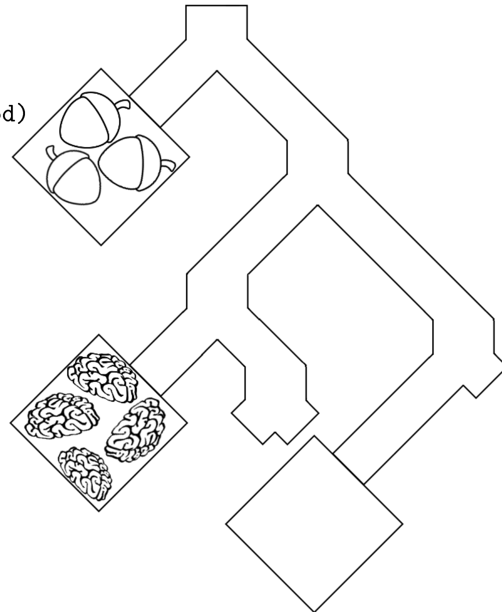
**c)** Let $\mathbb{B}$ be the Boolean values $\texttt{true}, \texttt{false}$.
Is $f : (\mathbb{B} \to \mathbb{B}_\perp) \to \mathbb{Z}$ with $f(g) = \begin{cases} 1 & \text{if } g(x) \neq \texttt{true} \text{ for all } x \in \mathbb{B} \\ 0 & \text{otherwise} \end{cases}$ monotonic?

**d)** Is $\to_\alpha$ terminating?

**e)** Is $\to_\alpha$ confluent?

## Exercise 2 (Programming in Haskell): (8 + 10 + 10 + 10 = 38 points)

We define a polymorphic data structure `ZombieHalls` to represent a zombie-infested school whose classrooms contain different types of food:



```
data ZombieHalls food =
    HallwayFork (ZombieHalls food) (ZombieHalls food)
  | HallwayClassroom (Int, food) (ZombieHalls food)
  | HallwayEnd
```

Here, we use three data constructors: One representing the case that the hallway forks and we can go in two directions, one for the case that we have a classroom on one side and can continue in the hallway and finally one case for the end of a hallway. The data structure `ZombieFood` is used to represent food for zombies. As example, consider the following definition of `exampleSchool` of type `ZombieLabyrinth ZombieFood`, corresponding to the illustration on the right:

```
data ZombieFood = Brains | Nuts deriving Show

exampleSchool :: ZombieHalls ZombieFood
exampleSchool =
    HallwayClassroom (3, Nuts)
     (HallwayFork
       (HallwayClassroom (4, Brains)
         (HallwayFork HallwayEnd HallwayEnd))
       (HallwayClassroom (0, Brains) HallwayEnd))
```

**a)** Implement a function `buildSchool :: Int -> ZombieHalls ZombieFood` such that for any integer number `n` $\geq 0$, it returns a structure of hallways containing $2^{n+1}$ classrooms in total. Of these, one half should each contain one brain and the other should each contain one nut.

**b)** Implement a fold function `foldZombieHalls`, including its type declaration, for the data structure `ZombieHalls`. As usual, the fold function replaces the data constructors in a `ZombieHalls` expression by functions specified by the user. The first argument of `foldZombieHalls` should be the function for the case of a `HallwayFork`, the second argument should replace the `HallwayClassroom` constructor and the third argument should replace the `HallwayEnd` data constructor. As an example, consider the following function definition, which uses `foldZombieHalls` to determine the number of dead ends in a `ZombieHalls` structure, where a classroom does not count as dead end. Hence, the call `numberOfDeadEnds exampleSchool` returns 3.

```
numberOfDeadEnds :: ZombieHalls food -> Int
numberOfDeadEnds school = foldZombieHalls (+) (\_ r -> r) 1 school
```

**c)** Implement the function `bcCounter :: ZombieHalls ZombieFood -> (Int, Int)`, which counts the number of **b**rains and **c**lassrooms in a given school and returns the two numbers as a tuple of integers. The first part of the tuple should be the number of brains in the school and the second should be the number of classrooms. For the definition of `bcCounter`, use only one defining equation where the right-hand side **is just one call to the function** `foldZombieHalls`. However, you may use and define non-recursive auxiliary functions.

For example, a call `bcCounter exampleSchool` should return the tuple `(4, 3)`.

**d)** We now want to write a program that trains a zombie to choose its food on a computer. To do this, implement a function `train :: IO Int` that asks the user if they want to eat **b**rains or **n**uts. The user should answer with either `b` or `n`. If the answer is neither `b` nor `n`, an error should be reported and the question should be repeated. If the answer is valid, a short success message should be shown and then, the function should return the number of tries it took the user to type a valid answer.

Calling `train` hence might look as follows:

```
*Main> train
Please enter 'b' for braaaaaains and 'n' for nuts:  braiiiins!
I did not understand you.  Please enter 'b' for braaaaaains and 'n' for nuts:  b
Good boy!  Here are your brains!
2
```

Note that the last line of this run is provided by the Haskell Interpreter, showing the return value of `train`.

Hints:

- You should use the function `getLine :: IO String` to read the input from the user and the function `putStr :: String -> IO ()` to print a `String`.
- To save space, you may assume that the following declarations exist in your program:

```
question, answerB, answerN, answerO :: String
question = "Please enter 'b' for braaaaaains and 'n' for nuts: "
answerB = "Good boy! Here are your brains!\n"
answerN = "You are a weird zombie. Here are your nuts!\n"
answerO = "I did not understand you. "
```

## Exercise 3 (Semantics): (22 + 10 + 5 = 37 points)

**a)** i) Let $\sqsubseteq_{D_1}$ and $\sqsubseteq_{D_2}$ be complete partial orders on $D_1$ resp. $D_2$ and $f : D_1 \to D_2$ a function. Prove that $f$ is continuous if and only if $f$ is monotonic and for all chains $S$ in $D_1$, $f(\sqcup S) \sqsubseteq_{D_2} \sqcup f(S)$ holds.

ii) Let $D = \mathbb{N} \to \{1\}_\perp$, i.e., $D$ is the set of all functions mapping the natural numbers to $\perp$ or 1. Let $\sqsubseteq$ be defined as usual on functions.

1) Prove that every chain $S \sqsubseteq D$ has a least upper bound w.r.t. the relation $\sqsubseteq$.

2) Prove that $\sqsubseteq$ is a cpo on $D$.

3) Give an example for an infinite chain in $(D, \sqsubseteq)$.

4) Give a monotonic, non-continuous function $f : D \to D$. You do not need to prove that $f$ has these properties.

**b)** i) Consider the following Haskell function `exp`:

```
exp :: (Int, Int) -> Int
exp (x, 0) = 1
exp (x, y) = x * exp (x, y - 1)
```

Please give the Haskell declaration for the higher-order function `f_exp` corresponding to `exp`, i.e., the higher-order function `f_exp` such that the least fixpoint of `f_exp` is `exp`. In addition to the function declaration, please also give the type declaration of `f_exp`. You may use full Haskell for `f_exp`.

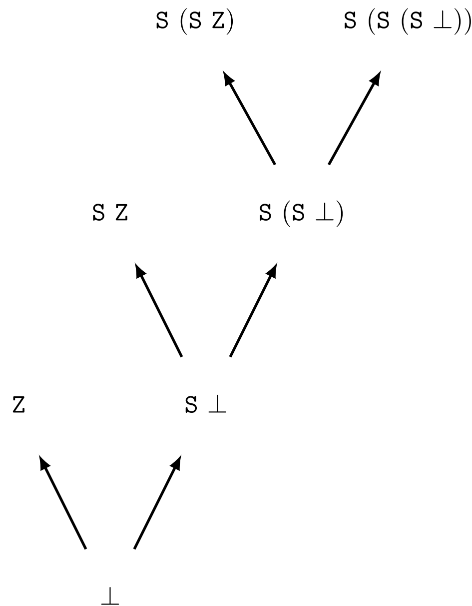
ii) Let $\phi_{\texttt{f_exp}}$ be the semantics of the function `f_exp`. Give the semantics of $\phi^n_{\texttt{f_exp}}(\bot)$ for $n \in \mathbb{N}$, i.e., the semantics of the $n$-fold application of $\phi_{\texttt{f_exp}}$ to $\bot$.

iii) Give the least fixpoint of $\phi_{\texttt{f_exp}}$.

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

```
data Nats = Z | S Nats
```

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




Now consider the following data type declarations:

```
data U = V
data T a = C | D (T a) | E a a
```

Give a graphical representation of the first three levels of the domain for the type `T U`. The third level contains the element `D C`, for example.

## Exercise 4 (Lambda Calculus): (4 + 6 + 10 = 20 points)

**a)** Please translate the following Haskell expression into an equivalent lambda term (e.g., using $\mathcal{L}am$). Translate the pre-defined function `>` to `GreaterThan`, `+` to `Plus`, `*` to `Times` and `-` to `Minus` (remember that the infix notation of `>`, `+`, `*`, `-` is not allowed in lambda calculus). It suffices to give the result of the transformation:

```
let sqrt = \x a -> if a * a > x then a - 1 else sqrt x (a + 1) in sqrt u 0
```

**b)** Let $t = \lambda \mathit{fromto}.\lambda x.\lambda y.\mathtt{If}\ (\mathtt{Eq}\ x\ y)\ \mathtt{Nil}\ (\mathtt{Cons}\ x\ (\mathit{fromto}\ (\mathtt{Plus}\ x\ 1)\ y))$ and

$$\begin{aligned} \delta = &\ \{\ \mathtt{If\ True} \to \lambda x.\lambda y.x, \\ &\quad \mathtt{If\ False} \to \lambda x.\lambda y.y, \\ &\quad \mathtt{Fix} \to \lambda f.f\ (\mathtt{Fix}\ f)\} \\ \cup &\ \{\ \mathtt{Plus}\ x\ y \to z \mid x, y \in \mathbb{Z} \wedge z = x + y\} \\ \cup &\ \{\ \mathtt{Eq}\ x\ y \to \mathtt{False} \mid x, y \in \mathbb{Z} \wedge x \neq y\} \\ \cup &\ \{\ \mathtt{Eq}\ x\ y \to \mathtt{True} \mid x, y \in \mathbb{Z} \wedge x = y\} \end{aligned}$$

Please reduce $\mathtt{Fix}\ t\ 1\ 2$ by WHNO-reduction with the $\to_{\beta\delta}$-relation. List **all** intermediate steps until reaching weak head normal form, but please write "$t$" instead of the term it represents whenever possible. However, you may combine several subsequent $\to_\beta$-steps.

**c)** We use the representation of Boolean values in the pure $\lambda$-calculus presented in the lecture. So, $\overline{\texttt{True}}$ is $\lambda x\ y.x$ and $\overline{\texttt{False}}$ is $\lambda x\ y.y$. Using this representation, give pure $\lambda$-terms for the following functions:

- $\overline{\texttt{If}}$, which gets three arguments, of which the first argument is Boolean and determines if the second or third value is returned (i.e., it behaves as you would expect). For example, $\overline{\texttt{If}}\ \overline{\texttt{True}}\ x\ y \to_\beta^+ x$.

- $\overline{\texttt{Xor}}$, which takes two Boolean values and returns true if and only if exactly one of them is $\overline{\texttt{True}}$. You may use $\overline{\texttt{True}}$ and $\overline{\texttt{False}}$ in your solution.

## Exercise 5 (Type Inference): (6 points)

Using the initial type assumption $A_0 := \{y :: \forall a.a \to a\}$ infer the type of the expression $\lambda x.(y\,x)\,x$ using the algorithm $\mathcal{W}$.