Exam in Functional Programming SS 2014 (V3M)

First Name: ____________________________________________

Last Name: _____________________________________________

Matriculation Number: ___________________________________

Course of Studies (please mark exactly one):

- Informatik Bachelor
- Informatik Master
- Mathematik Master
- Software Systems Engineering Master
- Other: __________________________

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Available Points</th>
<th>Achieved Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise 1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Exercise 2</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Exercise 3</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Exercise 4</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Exercise 5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

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): \(3 + 3 + 3 = 9 \text{ points}\)

a) Is \(\textbf{f} \rightarrow (\textbf{f} \; \text{True}) \; (\textbf{f} \; 1)\) well typed in Haskell? Give the expression's type or briefly explain why it is not well typed.

b) Prove or disprove: If a relation \(\triangleright\subseteq A \times A\) is conuenent, then every element of \(A\) has a normal form with respect to \(\triangleright\).

c) Are there monotonic functions which are not continuous? If so, give an example. Otherwise, give a brief explanation.
Exercise 2 (Programming in Haskell):  

(5 + 7 + 7 + 9 = 28 points)

We define a polymorphic data structure `Train` to represent trains that can contain different types of cargo.

```haskell
data Train a  
  = Locomotive (Train a)  
  | Wagon a (Train a)  
  | Empty deriving Show  
```

The data structure `Cargo` is used to represent different types of cargo.

```haskell
type Quantity = Int  
type Weight = Int -- in kg  
data Cargo  
  = NoCargo  
  | Persons Quantity  
  | Goods Weight deriving Show  
```

For example, `aTrain` is a valid expression of type `Train Cargo`.

```haskell
aTrain = Locomotive (Wagon (Goods 100) (Wagon (Persons 10) (Wagon (Goods 200) Empty)))
```

Like `aTrain`, you can assume that every `Train` consists of a single `Locomotive` at its beginning followed by a sequence of `Wagon`s and `Empty` at its end.

The following function can be used to `fold` a `Train`.

```haskell
fold :: (a -> b -> b) -> b -> Train a -> b  
fold _ res Empty = res  
fold f res (Locomotive t) = fold f res t  
fold f res (Wagon c t) = f c (fold f res t)
```

So for a `Train t`, `fold f res t` removes the constructor `Locomotive`, replaces `Wagon` by `f`, and replaces `Empty` by `res`.

In the following exercises, you are allowed to use predefined functions from the Haskell-Prelude.
a) Implement a function `filterTrain` together with its type declaration (`filterTrain :: ...`). The function `filterTrain` gets a predicate and an object of type `Train` as input and returns an object of type `Train` that only contains those wagons from the given `Train` whose cargo satisfies the predicate. For example, assume that the function `areGoods` is implemented as follows:

```haskell
areGoods :: Cargo -> Bool
areGoods (Goods _) = True
areGoods _ = False
```

Then the expression `filterTrain areGoods aTrain` should be evaluated to `Locomotive (Wagon (Goods 100) (Wagon (Goods 200) Empty))`.

b) Implement a function `buildTrain :: [Cargo] -> Train Cargo`. In the resulting `Train`, a single `Wagon` must not contain more than 1000 kg of `Goods`. If the input list contains `Goods` that weigh more than 1000 kg, then these `Goods` must not be contained in the resulting train. Apart from this restriction, all the `Cargo` given via the input list has to be contained. Moreover, the resulting `Train` has to consist of a single `Locomotive` at its beginning, followed by a sequence of `Wagons` and `Empty` at its end. In your solution, you should use the function `filterTrain` even if you could not solve the previous exercise part.

For example, `buildTrain [Persons 10, Goods 2000, Goods 1000]` should be evaluated to the expression `Locomotive (Wagon (Persons 10) (Wagon (Goods 1000) Empty))`. 
c) Implement a function `weight` together with its type declaration which computes the weight of all `Goods` in a train of type `Train Cargo`. For the definition of `weight`, use only one defining equation where the right-hand side is a call to the function `fold`.

For example, `weight aTrain` should be evaluated to 300.

d) In this part of the exercise, you should create a program that controls a robber. The goal of the robber is to steal `Goods`, but when he tries to transport more than 1000 kg, he gets too slow and is caught by the police.

Implement a function `robTrain :: Train Cargo -> IO ()`. Its input is a `Train` that just contains `Goods`. It ignores the `Locomotive` and processes the remainder of the `Train` as follows:

For a wagon with `n` kg `Goods`, it prints "Do you want to pick up the goods? (y|n)". If the user answers "y", it prints "You have stolen `n` kg goods.". Otherwise, there is no output.

Afterwards, if the accumulated `Goods` of the robber exceed the limit of 1000 kg, it prints "You were caught by the police." and terminates. Otherwise, if the whole `Train` has been processed (i.e., `Empty` is reached), it prints "You successfully robbed the train." and terminates. Otherwise, the program continues with the next part of the `Train`.

A successful run might look as follows:

```haskell
*Main> robTrain (Locomotive (Wagon (Goods 1000) (Wagon (Goods 200) Empty)))
Do you want to pick up the goods? (y|n) n
Do you want to pick up the goods? (y|n) y
You have stolen 200 kg goods.
You successfully robbed the train.
```

In the following run, the user is caught by the police:

```haskell
*Main> robTrain (Locomotive (Wagon (Goods 1000) (Wagon (Goods 200) Empty)))
Do you want to pick up the goods? (y|n) y
You have stolen 1000 kg goods.
Do you want to pick up the goods? (y|n) y
You have stolen 200 kg goods.
You were caught by the police.
```
**Hint:** You should use the function `getLine :: IO String` to read the input from the user. To print a `String`, you should use the function `putStr :: String -> IO ()` or the function `putStrLn :: String -> IO ()`, if the output should end with a line break. You should use the function `show :: Int -> String` to convert an `Int` to a `String`. To save space, you may assume that the following declarations exist in your program:

```hs
pickUp, caught, success :: String
pickUp = "Do you want to pick up the goods? (y|n)"
caught = "You were caught by the police."
success = "You successfully robbed the train."
```
Exercise 3 (Semantics): \(10 + 10 + 6 = 26 \text{ points}\)

a) i) Let \(L_j = \{[], [[[]], \ldots], \ldots\}\), i.e., \(L_j\) contains all lists where \(m\) opening brackets are followed by \(m\) closing brackets for an \(m \in \mathbb{N} \setminus \{0\}\). Let \(\leq_{nl} \subseteq L_j \times L_j\) be the relation that compares the nesting-level of two lists. More formally, if \(nl(x)\) is the nesting level of the list \(x\) and \(\leq \subseteq \mathbb{N} \times \mathbb{N}\) is the usual less-or-equal relation, then

\[
l \leq_{nl} l' \iff nl(l) \leq nl(l')
\]

So we have, e.g., \(\emptyset \leq_{nl} [\emptyset]\) because the nesting level of \(\emptyset\) is one and the nesting level of \([\emptyset]\) is two.

1) Give an example for an infinite chain in \((L_j, \leq_{nl})\).
2) Prove or disprove: the partial order \(\leq_{nl}\) is complete on \(L_j\).

ii) Let \(L_0\) be the set of all Haskell lists containing only zeros (so, e.g., \(\emptyset \in L_0\) and \([0,0,0] \in L_0\) and let \(\leq_{len} \subseteq L_0 \times L_0\) be the relation that compares the length of two lists where all infinite lists are considered to have the same length. More formally, if \(len(x)\) is the length of the list \(x\) and \(\leq \subseteq \mathbb{N} \cup \{\infty\} \times \mathbb{N} \cup \{\infty\}\) is the usual less-or-equal relation, then

\[
l \leq_{len} l' \iff len(l) \leq len(l')
\]

1) Give an example for an infinite chain in \((L_0, \leq_{len})\).
2) Prove or disprove: the partial order \(\leq_{len}\) is complete on \(L_0\).
b)  i) Consider the following Haskell function $f$:

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

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

ii) Let $\phi$ be the semantics of the function $\phi$. Give the definition of $\phi^n(\bot)$ in closed form for any $n \in \mathbb{N}$, i.e., give a non-recursive definition of the function that results from applying $\phi$ $n$-times to $\bot$.

iii) Give the definition of the least fixpoint of $\phi$ in closed form.
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for \( \text{Nats} \) on the right:

\[
\begin{array}{c}
\text{data Nats} = Z \mid S \text{Nats} \\
\text{-------------------------------------}
\end{array}
\]

\[
\begin{array}{c}
\text{data Train a} \\
= L (\text{Train a}) \quad \text{-- a Locomotive} \\
| W a (\text{Train a}) \quad \text{-- a Wagon} \\
| E \quad \text{-- an Empty Train}
\end{array}
\]

\[
\begin{array}{c}
\text{data Cargo} \\
= N C \quad \text{-- No Cargo} \\
| P \text{ Int} \quad \text{-- n Persons}
\end{array}
\]

\[
\begin{array}{c}
S \quad Z \\
S (S \bot) \quad 3^{\text{rd}} \text{ level}
\end{array}
\]

\[
\begin{array}{c}
Z \quad \bot \\
S \bot \quad 2^{\text{nd}} \text{ level}
\end{array}
\]

\[
\begin{array}{c}
\bot \quad 1^{\text{st}} \text{ level}
\end{array}
\]

Give a graphical representation of the first three levels of the domain for the type \( \text{Train Cargo} \). The third level contains the element \( W (P \bot) \bot \), for example. Note that the domain for the type \( \text{Train Cargo} \) also contains Trains with multiple locomotives, Trains without \( E \) at their ends, and so on. In other words, the assumption from Exercise 2 ("Assume that every \text{Train} consists of a single \text{Locomotive} at its beginning followed by a sequence of \text{Wagons} and \text{Empty} at its end.") does not hold for this exercise.
Exercise 4 (Lambda Calculus): (4 + 8 + 5 = 17 points)

a) Reconsider the function \( f' \) from the previous exercise:

\[
\begin{align*}
f' & : \text{Int} \rightarrow \text{Int} \\
f' \; x \; 0 & = 1 \\
f' \; x \; y & = x \; \ast \; f' \; x \; (y \; - \; 1)
\end{align*}
\]

Please implement this function in the Lambda Calculus, i.e., give a term \( f \) such that, for all \( x, y, z \in \mathbb{Z} \), \( f' \; x \; y = z \) if and only if \( f \; x \; y \) can be reduced to \( z \) via WHNO-reduction with the \( \rightarrow_{\beta\delta} \)-relation and the set of rules \( \delta \) as introduced in the lecture to implement Haskell. You can use infix notation for predefined functions like \((=)=\), \((\ast)\) or \((-)\).
b) Let

\[ t = \lambda add \ x \ y. \ \text{if} \ (y == 0) \ x \ (\text{add} \ (x + 1) \ (y - 1)) \]

and

\[ \delta = \{ \text{if True} \rightarrow \lambda x \ y. \ x, \]
\[ \quad \text{if False} \rightarrow \lambda x \ y. \ y, \]
\[ \quad \text{fix} \rightarrow \lambda f. \ f(\text{fix} \ f) \} \]
\[ \cup \{ \ x - y \rightarrow z \ | \ x, y, z \in \mathbb{Z} \land z = x - y \} \]
\[ \cup \{ \ x + y \rightarrow z \ | \ x, y, z \in \mathbb{Z} \land z = x + y \} \]
\[ \cup \{ \ x == \text{True} \ | \ x \in \mathbb{Z} \} \]
\[ \cup \{ \ x == \text{False} \ | \ x, y \in \mathbb{Z}, x \neq y \} \]

Please reduce \texttt{fix t 0 0} by WHNO-reduction with the \( \rightarrow_{\beta \delta} \)-relation. List all intermediate steps until reaching weak head normal form, but please write “\( t \)” instead of

\[ \lambda add \ x \ y. \ \text{if} \ (y == 0) \ x \ (\text{add} \ (x + 1) \ (y - 1)) \]

whenever possible.
c) Consider the Boolean operator \texttt{nor} where \texttt{nor}(x, y) holds if and only if \texttt{or}(x, y) \texttt{ does not} hold. Using the representation of Boolean values in the pure \texttt{$\lambda$}-calculus presented in the lecture, i.e., \texttt{True} is represented as \texttt{$\lambda$x y.x} and \texttt{False} as \texttt{$\lambda$x y.y}, give a pure $\lambda$-term for \texttt{nor} in $\rightarrow$\texttt{B}$\rightarrow$ normal form.
Exercise 5 (Type Inference): (10 points)

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