Exam in Functional Programming SS 2014 (V3B)

First Name: ____________________________________________

Last Name: ____________________________________________

Matriculation Number: __________________________________

Course of Studies (please mark exactly one):

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

<table>
<thead>
<tr>
<th>Exercise 1</th>
<th>Available Points</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise 2</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Exercise 3</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Exercise 4</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Exercise 5</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>Exercise 6</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td>90</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 points)

a) Is \( f \rightarrow (f \text{ True}) (f \ 1) \) 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 confluent, 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): \(5 + 7 + 7 = 19\) 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 Wagons 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 \(t\), \(\text{fold } f \text{ res } t\) removes the constructor Locomotive, replaces Wagon by \(f\), and replaces Empty by \(\text{res}\).

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 a` as input and returns an object of type `Train a` 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.
Exercise 3 (List Comprehensions): \((3 + 3 + 5 = 11 \text{ points})\)

a) Write a Haskell function \(\text{divisors} :: \text{Int} \rightarrow \text{[Int]}\) to compute the list of all proper divisors of a given number \(x\). Here, you can assume \(x \geq 2\). The result of \(\text{divisors} \ x\) includes 1, but not the number \(x\) itself. So for example, \(\text{divisors} \ 6 = [1,2,3]\). Use only one defining equation where the right-hand side is a list comprehension.

\[ \text{divisors} \ x = \{ \text{Int} \} \]

**Hint:** The function \(\text{mod} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\) can be used to compute the modulo of two integers.

b) Write a Haskell expression in form of a list comprehension to compute all perfect numbers. A number \(x\) with \(x \geq 2\) is perfect if and only if the sum of its proper divisors is equal to itself. For example, 6 is perfect, since its proper divisors are 1, 2, and 3 and the sum of its proper divisors is 6. In your solution, you should use the function \(\text{divisors}\) even if you were not able to solve the previous exercise part.

**Hint:** The function \(\text{sum} :: \text{[Int]} \rightarrow \text{Int}\) computes the sum of a list of integers.

c) Write a Haskell expression in form of a list comprehension to compute all semiperfect numbers. A number \(x\) with \(x \geq 2\) is semiperfect if and only if the sum of all or some of its proper divisors is equal to itself. For example, 12 is semiperfect: Its proper divisors are 1, 2, 3, 4, and 6 and the sum of 2, 4, and 6 is 12. In your solution, you should use the function \(\text{divisors}\) even if you were not able to solve exercise part (a). Moreover, you may use the function \(\text{sum}\) and the following functions:

- The function \(\text{exists} :: (\text{a} \rightarrow \text{Bool}) \rightarrow \text{[a]} \rightarrow \text{Bool}\) tests whether there is an element in the given list that satisfies the given predicate.
- The function \(\text{subsequences} \ [\text{a}] \rightarrow \text{[[a]]}\) computes all subsequences of the given list. For example, we have:

\[
\text{subsequences} \ [1,2,3] = [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
\]
Exercise 4 (Semantics): (10 + 10 + 6 + 3 = 29 points)

a) i) Let \( L \) = \{ [], [[]], [[[]]], \ldots \}, i.e., \( L \) contains all lists where \( m \) opening brackets are followed by \( m \) closing brackets for an \( m \in \mathbb{N} \setminus \{0\} \). Let \( \leq_{nl} \subseteq L \times L \) 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., \([[]] \leq_{nl} [[]] \) because the nesting level of \([[]] \) is one and the nesting level of \( [[]] \) is two.

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

ii) Let \( L_0 \) be the set of all Haskell lists containing only zeros (e.g., \([\ ] \in L_0 \) 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 \( f \):

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

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

iii) Give the definition of the least fixpoint of \( \phi_{\Phi} \) 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:

\[
\begin{align*}
\text{data Nats} & = Z \mid S \text{Nats} \\
\text{data Train} a & = L \text{(Train } a\text{)} \quad \text{-- a Locomotive} \\
& \mid W a \text{(Train } a\text{)} \quad \text{-- a Wagon} \\
& \mid E \quad \text{-- an Empty Train} \\
\text{data Cargo} & = \text{NC} \quad \text{-- No Cargo} \\
& \mid \text{P Int} \quad \text{-- n Persons}
\end{align*}
\]

Give a graphical representation of the first three levels of the domain for the type \( \text{Train Cargo} \). The third level contains the element \( W (P \perp) \perp \), 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 Train consists of a single Locomotive at its beginning followed by a sequence of Wagons and Empty at its end.") does not hold for this exercise.
d) Consider the definition for \texttt{Nats} from the previous exercise part, i.e., \texttt{data Nats = Z | S Nats}.

Moreover, consider the following Haskell function \texttt{f'}:

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

Write a function \texttt{fNat} : \texttt{Nats} \rightarrow \texttt{Nats} \rightarrow \texttt{Nats} in \texttt{Simple Haskell} which, for natural numbers, computes the same result as the function \texttt{f'}. That means, if \( n, m \geq 0 \) and \( f' n m = x \), then we have \( fNat (\texttt{S} n \texttt{Z}) (\texttt{S} m \texttt{Z}) = \texttt{S} x \texttt{Z} \). You can assume a predefined function \texttt{mult} :\texttt{Nats} \rightarrow \texttt{Nats} \rightarrow \texttt{Nats} to multiply two natural numbers. However, there is no predefined function to subtract natural numbers of type \texttt{Nats}. Your solution should use the functions defined in the transformation from the lecture such as \texttt{isa}, \texttt{argof}, and \texttt{bot}. You do not have to use the transformation rules from the lecture, though.
Exercise 5 (Lambda Calculus): 

a) Reconsider the function $f'$ from the previous exercise:

\[
f' :: \text{Int} \rightarrow \text{Int} \\
f' x 0 = 1 \\
f' x y = x \times f' x (y - 1)
\]

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 $(==), (\times)$ or $(-)$.
b) Let

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

and

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

Please reduce \( \text{fix} \ t00 \) 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

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

whenever possible.
Exercise 6 (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 $W$. 