Prof. Dr. J. Giesl F. Frohn

Exam in Functional Programming SS 2014 (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	**9**	
Exercise 2	**28**	
Exercise 3	**26**	
Exercise 4	**17**	
Exercise 5	**10**	
Sum	**90**	

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 -> (f 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 $\succ \subseteq A \times A$ is confluent, then every element of A has a normal form with respect to $\succ$.

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.

```
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.

```
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`.

```
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`.

```
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 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:

```
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:

```
*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 succesfully robbed the train.
```

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

```
*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:

```
pickUp, caught, success :: String
pickUp = "Do you want to pick up the goods? (y|n) "
caught = "You were caught by the police."
success = "You succesfully robbed the train."
```

Exercise 3 (Semantics): (10 + 10 + 6 = 26 points)

a) i) Let $L_{[]} = \{$`[]`,`[[]]`,`[[[]]]`$,\dots\}$, 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 $\leq \subset \mathbb{N} \times \mathbb{N}$ 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 (so, 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 $\leq \subset \mathbb{N} \cup \{\infty\} \times \mathbb{N} \cup \{\infty\}$ 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 `ff` corresponding to `f`, i.e., the higher-order function `ff` such that the least fixpoint of `ff` is `f`. In addition to the function declaration, please also give the type declaration for `ff`. You may use full Haskell for `ff`.

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

iii) Give the definition of the least fixpoint of $\phi_{\texttt{ff}}$ in closed form.

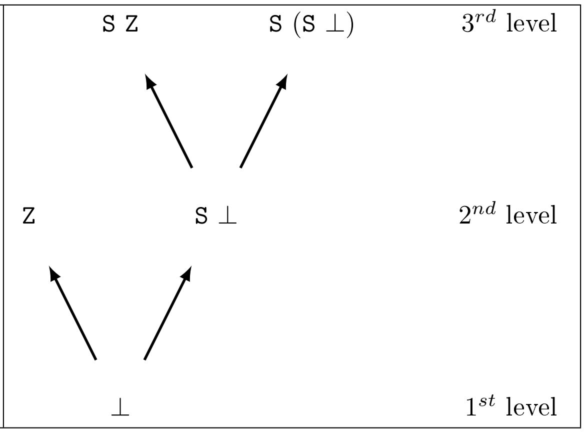
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for `Nats` on the right:

```
data Nats = Z | S Nats

-------------------------------------

data Train a
    = L (Train a)   -- a Locomotive
    | W a (Train a) -- a Wagon
    | E             -- an Empty Train

data Cargo
    = NC    -- No Cargo
    | P Int -- n Persons
```



Give a graphical representation of the first three levels of the domain for the type `Train Cargo`. The third level contains the element `W (P ⊥) ⊥`, for example. Note that the domain for the type `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.

Exercise 4 (Lambda Calculus): (4 + 8 + 5 = 17 points)

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

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

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

b) Let

$$t = \lambda\, add\ x\ y.\, \texttt{if}\ (y == 0)\ x\ (add\ (x+1)\ (y-1))$$

and

$$\begin{aligned}\delta = \ & \{\ \texttt{if True} \to \lambda\, x\ y.\, x, \\ & \quad \texttt{if False} \to \lambda\, x\ y.\, y, \\ & \quad \texttt{fix} \to \lambda\, f.\, f(\texttt{fix}\ f)\} \\ \cup\ & \{\ x - y \to z \mid x, y, z \in \mathbb{Z} \wedge z = x - y\} \\ \cup\ & \{\ x + y \to z \mid x, y, z \in \mathbb{Z} \wedge z = x + y\} \\ \cup\ & \{\ x == x \to \texttt{True} \mid x \in \mathbb{Z}\} \\ \cup\ & \{\ x == y \to \texttt{False} \mid x, y \in \mathbb{Z}, x \neq y\}\end{aligned}$$

Please reduce $\texttt{fix}\, t\, 0\, 0$ 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

$$\lambda\, add\ x\ y.\, \texttt{if}\ (y == 0)\ x\ (add\ (x+1)\ (y-1))$$

whenever possible.

c) Consider the Boolean operator `nor` where `nor`(x, y) holds if and only if `or`(x, y) does *not* hold. Using the representation of Boolean values in the pure λ-calculus presented in the lecture, i.e., `True` is represented as $\lambda\, x\ y.\, x$ and `False` as $\lambda\, x\ y.\, y$, give a pure λ-term for `nor` in $\rightarrow_\beta$-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 $\lambda f.f\,(f\,x)$ using the algorithm $\mathcal{W}$.