Prof. Dr. J. Giesl
M. Brockschmidt, F. Emmes

Exam in Functional Programming SS 2012 (V3B)

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	**44**	
Exercise 3	**40**	
Exercise 4	**10**	
Exercise 5	**6**	
Sum	**120**	

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 + 4 + 4 + 4 + 4 = 20 points)

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

a) Is $\sqsubseteq$ always a complete partial order for flat domains like $\mathbb{Z}_\perp, \mathbb{B}_\perp, \ldots$?

b) Can the function $f : \mathbb{Z}_\perp \to \mathbb{Z}$ with $f(x) = \begin{cases} 1 & \text{if } x \in \mathbb{Z} \text{ and } x \leq 0 \\ 0 & \text{otherwise} \end{cases}$ be implemented in Haskell?

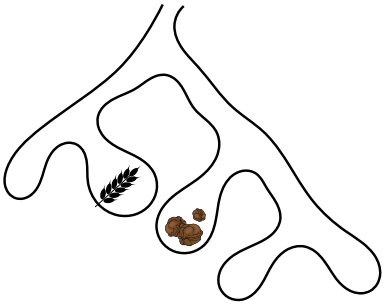
c) Is $g : (\mathbb{Z} \to \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$ with $g(h) = \begin{cases} 0 & \text{if } h(x) \neq \perp \quad \text{for all } x \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$ continuous?

d) If a lambda term t can be reduced to s with $\to_{\beta\delta}$ using an outermost strategy, can t also be reduced to s with $\to_{\beta\delta}$ using an innermost strategy? Here, you may choose an arbitrary delta-rule set δ.

e) The $\to_{\beta\delta}$ reduction in lambda calculus is confluent. Is Simple Haskell also confluent?

Exercise 2 (Programming in Haskell): (10 + 10 + 8 + 10 + 6 = 44 points)

We define a polymorphic data structure `HamsterCave` to represent hamster caves which can contain different types of food.



```
data HamsterCave food
    = EmptyTunnel
    | FoodTunnel food
    | Branch (HamsterCave food) (HamsterCave food)
    deriving Show
```

The data structure `HamsterFood` is used to represent food for hamsters. For example, `exampleCave` is a valid expression of type `HamsterCave HamsterFood`.

```
data HamsterFood = Grain | Nuts deriving Show

exampleCave :: HamsterCave HamsterFood
exampleCave = Branch
    (Branch EmptyTunnel (FoodTunnel Grain))
    (Branch (FoodTunnel Nuts) (Branch EmptyTunnel EmptyTunnel))
```

a) Implement a function `digAndFillCave :: Int -> HamsterCave HamsterFood`, such that for any integer number `n` > 1, `digAndFillCave n` creates a hamster cave without empty tunnels of depth `n`, such that the number of `FoodTunnel`s containing `Grain` equals the number of `FoodTunnel`s containing `Nuts`. Here, the depth of a cave is the maximal number of "nodes" on any path from the entry of the cave to a dead end. Thus, `exampleCave` has depth 4.

b) Implement a fold function `foldHamsterCave`, including its type declaration, for the data structure `HamsterCave`. As usual, the fold function replaces the data constructors in a `HamsterCave` expression by functions specified by the user. The first argument of `foldHamsterCave` should be the function for the case of the empty tunnel, the second argument the function for the case of the food tunnel, and the third argument the function for the case of a branch. As an example, the following function definition uses `foldHamsterCave` to determine the number of dead ends (either with or without food) in a cave, such that the call `numberOfDeadEnds exampleCave` returns `5`.

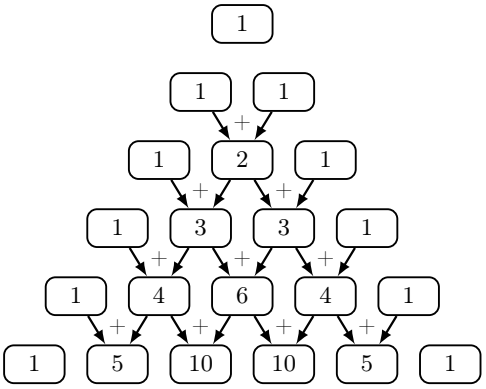
```
numberOfDeadEnds :: HamsterCave food -> Int
numberOfDeadEnds cave = foldHamsterCave 1 (\_ -> 1) (+) cave
```

c) Implement the function `collectFood :: HamsterCave food -> (HamsterCave food, [food])`, which returns a tuple for a given hamster cave. The first argument of the tuple is the same hamster cave as the one given to the function, but without any food (i.e., every `FoodTunnel` is replaced by an `EmptyTunnel`). The second argument is a list of all the food that was removed from the cave. For the definition of `collectFood`, use only one defining equation where the right-hand side is a call to the function `foldHamsterCave`.

For example, a call `collectFood exampleCave` should return the following tuple:

```
(Branch (Branch EmptyTunnel EmptyTunnel)
        (Branch EmptyTunnel (Branch EmptyTunnel EmptyTunnel))
,[Grain,Nuts])
```

d) Implement a cyclic data structure `pascalsTriangle :: [[Int]]` (consisting of lists of lists of `Int`s) that represents Pascal's triangle. The first row of the triangle is represented by the first list of integers (`[1]`), the second row by the second list (`[1,1]`), and so forth. Each row in Pascal's triangle is constructed from its preceding row, by adding each pair of consecutive numbers. For this, it is assumed that all numbers lying outside of the preceding row are zeros.
Hint: You should use use the function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`, which applies the function given as its first argument to combine the elements of two lists. For example `zipWith (++) ["a","b"] ["c", "d", "e"]` results in the list `["ac","bd"]`. Note that the length of the resulting list is the smallest length of both input lists.



e) Write a Haskell expression in form of a *list comprehension* to compute all prime numbers. To determine if a number `i` is prime, test whether no number from `2` to `i - 1` divides `i`. You may use the functions `all :: (a -> Bool) -> [a] -> Bool` where `all p xs` is `True` iff `p x` is `True` for all elements `x` of the list `xs`, the function `not :: Bool -> Bool`, and the function `divides` as defined below.

```
divides :: Int -> Int -> Bool
i `divides` j = j `mod` i == 0
```

Exercise 3 (Semantics): (21 + 10 + 5 + 4 = 40 points)

a) i) Let $\sqsubseteq$ be a cpo on D and $f : D \to D$ be continuous. Prove the fixpoint theorem, i.e., that $\sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ exists and that this is the least fixpoint of f. You may use all other results from the lecture in your proof.

ii) Let $D = 2^{\mathbb{N}}$, i.e., D is the set of all sets of natural numbers and let $\subseteq$ denote the usual subset relation.

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

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

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

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 `mult`:

```
mult :: (Int, Int) -> Int
mult (0, y) = 0
mult (x, y) = y + mult (x - 1, y)
```

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

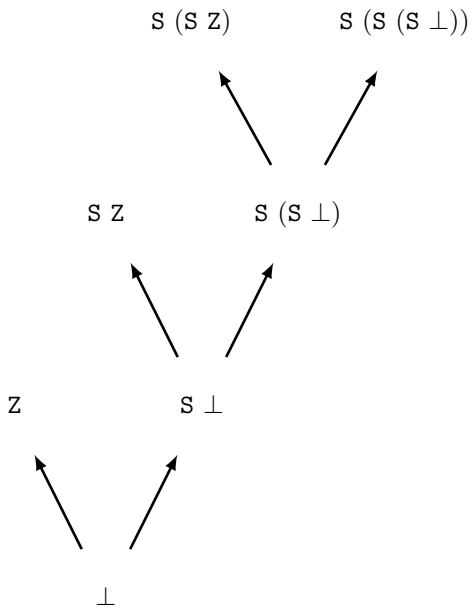
ii) Let $\phi_{\texttt{f\_mult}}$ be the semantics of the function `f_mult`. Give the semantics of $\phi^n_{\texttt{f\_mult}}(\bot)$ for $n \in \mathbb{N}$, i.e., the semantics of the n-fold application of $\phi_{\texttt{f\_mult}}$ to $\bot$.

iii) Give all fixpoints of $\phi_{\texttt{f\_mult}}$ and mark the least fixpoint.

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 X = A X Y | B Y
data Y = E Y | H
```

Give a graphical representation of the first three levels of the domain for the type `X`. The third level contains the element `A (A ⊥ ⊥) ⊥`, for example.

d) Consider the usual definition for `Nats` above, i.e., `data Nats = Z | S Nats`.

Write a function `plus :: Nats -> Nats -> Nats` in **Simple** Haskell that computes the sum of two natural numbers, i.e., `plus S(S(Z)) S(Z)` should yield `S(S(S(Z)))`. Your solution should use the functions defined in the transformation from the lecture such as $\texttt{sel}_{n,i}$, $\texttt{isa}_{\underline{\texttt{constr}}}$, $\texttt{argof}_{\underline{\texttt{constr}}}$, and `bot`. You do not have to use the transformation rules from the lecture, though.

Exercise 4 (Lambda Calculus): (4 + 6 = 10 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 `LessThan`, `+` to `Plus` 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 quot = \x y -> if x < y then 0 else 1 + quot (x-y) y in quot v w
```

b) Let $t = \lambda fact.(\lambda x.(\texttt{If}\ (\texttt{LessThanOrE}\ x\ 1)\ 1\ (\texttt{Times}\ x\ (fact\ (\texttt{Minus}\ x\ 1)))))$ and

$$\begin{aligned}\delta = \ & \{\ \texttt{If True} \to \lambda x\ y.x, \\ & \quad \texttt{If False} \to \lambda x\ y.y, \\ & \quad \texttt{fix} \to \lambda f.f(\texttt{fix}\ f)\} \\ \cup\ & \{\ \texttt{Minus}\ x\ y \to z \mid x, y \in \mathbb{Z} \wedge z = x - y\} \\ \cup\ & \{\ \texttt{Times}\ x\ y \to z \mid x, y \in \mathbb{Z} \wedge z = x \cdot y\} \\ \cup\ & \{\ \texttt{LessThanOrE}\ x\ y \to b \mid x, y \in \mathbb{Z} \wedge ((x \leq y \wedge b = \texttt{True}) \vee (x > y \wedge b = \texttt{False}))\}\end{aligned}$$

Please reduce `fix` t 1 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.

Exercise 5 (Type Inference): (6 points)

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