Prof. Dr. J. Giesl M. Hark

Exam in Functional Programming SuSe 19

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	**38**	
Exercise 2	**43**	
Exercise 3	**24**	
Exercise 4	**15**	
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 in either English or German.
- 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 be evaluated with **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**.

I hereby confirm that I feel healthy and able to participate in the exam.

_____
Signature

Exercise 1 (Programming in Haskell): (6 + 9 + 9 + 14 = 38 points)

We use the following data structure `Graph a` to represent *undirected* graphs in Haskell by a list of edges, where the nodes are labeled by values of type `a`.

```
data Graph a = Edges [(a,a)]
```

So the edges are represented as pairs of type `(a,a)`. For example, the graph from Fig. 1 can be represented by:

```
gExmp :: Graph Int
gExmp = Edges [(1,2),(1,4),(1,5),(2,3),(2,4),(3,4),(4,5),(4,6)]
```

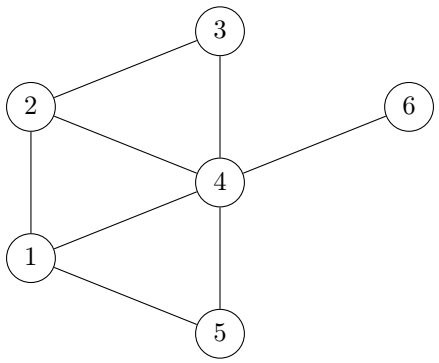


Figure 1: Graph of type `Graph Int`

In each of the following exercises you can use functions from the Haskell `Prelude` and from previous exercises even if you did not implement them. Moreover, you can always implement auxiliary functions.

a) Write a Haskell–function `neighbors` and also give its type declaration. Its first argument is a graph `g::Graph a` for a type `a` from the type class `Eq`, and its second argument is an element `v::a`. The call `neighbors g v` then results in the list of nodes directly connected by an edge to `v` in `g`. For example, `neighbors gExmp 2 == [1,3,4]`. The order of the resulting list is *irrelevant* and it does not matter if the list contains duplicates.

b) Consider the following function which computes a list of all values stored in a graph.

```
nodes :: Eq a => Graph a -> [a]
nodes (Edges es) = removeDuplicates (concat (map (\(x,y)->[x,y]) es))
  where
  removeDuplicates = foldr (\x ys -> x:(filter (x /=) ys)) []
```

Hence, `nodes gExmp` results in `[1,2,4,5,3,6]`.

Write a Haskell–function `existsPath` and also give its type declaration. Its first argument is a graph `g::Graph a` for a type `a` from the type class `Eq`, and its second and third argument are nodes `v::a` and `w::a`. The call `existsPath g v w` results in `True` if and only if there is a path connecting `v` and `w` in `g`. There is a path from `v` to `w` of at most length `n` if and only if `v == w` or there is a neighbor `u` of `v` and a path of at most length `n-1` from `u` to `w`. For example, `existsPath gExmp 6 3 == True`.

Note that a path from `v` to `w` exists if and only if there exists a path from `v` to `w` of at most `length (nodes g)`.

Hints:

- You may use the function `nodes`.
- The function `elem::Eq a => a -> [a] -> Bool` checks membership in a list, i.e., `elem x xs == True` if and only if `x` occurs in `xs`.

c) Write a Haskell–function `isConnected` and also give its type declaration. Its only argument is `g::Graph a` for a type `a` from the type class `Eq`. It returns `True` if and only if for every two elements `v` and `w` of `nodes g`, we have `existsPath g v w == True`. For example, `isConnected gExmp == True`.

Hints:

- You may again use the function `nodes` from part b) of the exercise.
- List comprehensions are particularly suitable to compute all pairs `(v,w)` of (distinct) elements of `nodes g`.
- The function `and::[Bool] -> Bool` conjuncts all elements in the input, e.g., `and [True, True, True] == True` but `and [True, False, True] == False`.

d) Implement a function `railNetwork::Graph String -> IO ()`. This function simulates a network of railway connections as a `Graph String` between cities. The user can either add a connection to the network or check if there exists a connection between two cities.

Furthermore, the following data declaration is given.

```
data NetworkInput = Track (String, String) | Connection (String, String) | Exit
```

When `railNetwork g` is called, it asks the user for an input. If the input is

- `"Track:Start;End"` then an edge from `Start` to `End` is added to `g` and the function `railNetwork` calls itself on the transformed graph again.
- `"Connection:Start;End"` then the user is informed whether there is a path from `Start` to `End` in `g`. The function `railNetwork` then calls itself again with the same argument.
- something else, then the program terminates.

In this exercise, you can assume that there is a function `parseNetworkInput::String -> NetworkInput` that translates a `String` into a `NetworkInput`, i.e., `parseNetworkInput "Track:Start;End" == Track ("Start", "End")`, `parseNetworkInput "Connection:Start;End" == Connection ("Start", "End")`, and `parseNetworkInput s == Exit` for all other strings `s`.

An example run should look as follows. Here, inputs of the user are written in *italics*.

```
*Main> railNetwork (Edges [])
Enter track or check connection
```

Track:Berlin;Wolfsburg

```
Track added
Enter track or check connection
```

Track:Wolfsburg;Cologne

```
Track added
Enter track or check connection
```

Connection:Cologne;Berlin

```
There is a connection from Cologne to Berlin
Enter track or check connection
```

Connection:Cologne;Munich

```
This connection does not exist
Enter track or check connection
```

Stop

```
*Main>
```

Hints:

- The function `putStrLn::String -> IO()` prints a string and performs a line break.
- The function `getLine::IO String` reads a string from the keyboard.

Exercise 2 (Semantics): ((9 + 11) + (5 + 7 + 4) + 7 = 43 points)

a) i) Let D be a domain and $\sqsubseteq$ a complete partial order on D. Let $f : D \to D$ be a continuous function w.r.t. $\sqsubseteq$ and $\text{Fix}(f) \subseteq D$ its set of fixpoints. Let $\sqsubseteq|_{\text{Fix}(f)}$ be the restriction of $\sqsubseteq$ to $\text{Fix}(f)$, i.e., for every $x, y \in \text{Fix}(f)$ let $x \sqsubseteq|_{\text{Fix}(f)} y$ hold iff $x \sqsubseteq y$. Prove that $\sqsubseteq|_{\text{Fix}(f)}$ is complete on $\text{Fix}(f)$.

ii) Let D be a domain, $\sqsubseteq$ a complete partial order on D, and $f : D \to D$ a continuous function w.r.t. $\sqsubseteq$. By i), f has a least fixpoint $\text{lfp}\ f$. Prove the following claim.

$$\text{If } d \in D \text{ with } f(d) \sqsubseteq d \text{ then } \text{lfp}\ f \sqsubseteq d.$$

Hints:

- Use the characterization of the least fixpoint from the Fixpoint Theorem and perform an induction proof.

b) i) Consider the following Haskell function `f`:

```
f :: (Int, Int, Int) -> Int
f (x, y, 0) = 1
f (x, y, n) = x * f(x, y, n-1) + y * f(x, y, n-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 $\phi_{\texttt{ff}}^m(\bot)$ in closed form for any $m \in \mathbb{N}$, i.e., give a non-recursive definition of the function that results from applying $\phi_{\texttt{ff}}$ m-times to $\bot$. Here, you should assume that `Int` can represent all integers, so no overflow can occur.

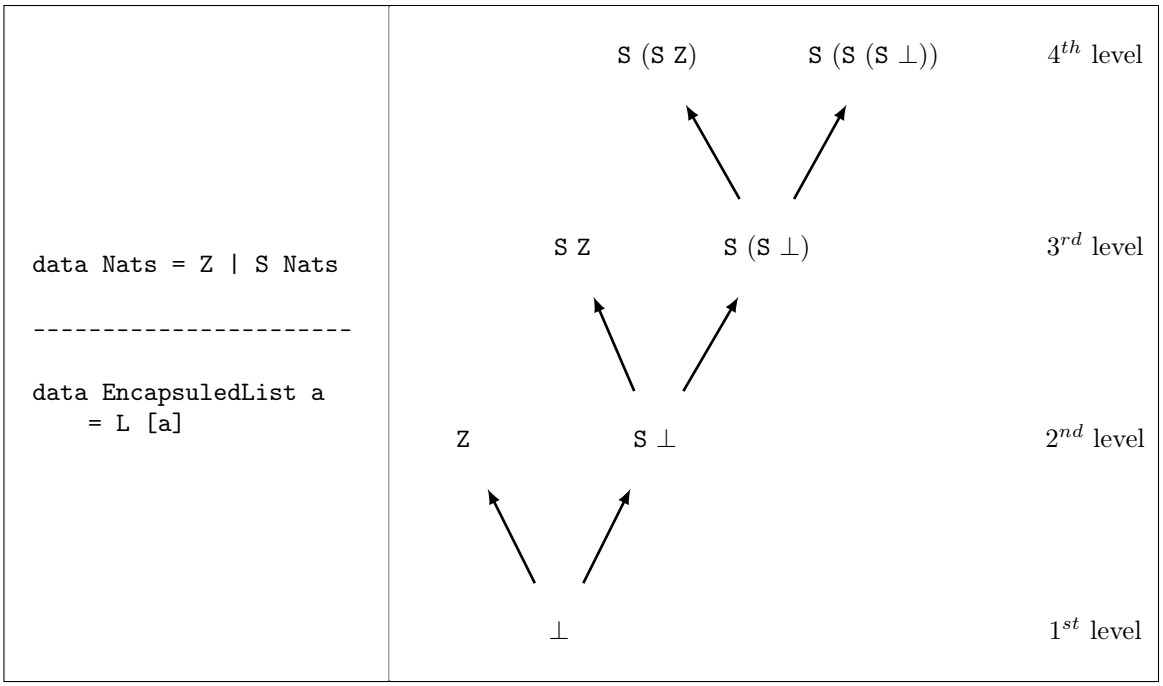
iii) Give the definition of the least fixpoint of $\phi_{\texttt{ff}}$ in closed form.

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

```
data Nats = Z | S Nats

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

data EncapsuledList a
    = L [a]
```



Give a graphical representation of the first four levels of the domain for the type `EncapsuledList Bool`.

Exercise 3 (Lambda Calculus): ((5 + 5) + 8 + 6 = 24 points)

a) Consider the following function computing the difference of two natural numbers (if the first argument is not smaller than the second). Here, natural numbers are represented by the data structure `Nats` defined by `data Nats = Z | S Nats`.

```
minus :: Nats -> Nats -> Nats
minus x Z = x
minus (S x) (S y) = minus x y
```

i) Please give an equivalent function in *simple* Haskell.

ii) Implement the function `minus` in the *lambda calculus*, i.e., give a lambda term q such that, for all natural numbers m, n the term q ($\mathtt{S}^m$ `Z`) ($\mathtt{S}^n$ `Z`) can be reduced to

$$\begin{cases} \mathtt{S}^{m-n}\ \mathtt{Z}, & \text{if } m \geq n \\ \mathtt{bot}, & \text{otherwise} \end{cases}$$

via WHNO-reduction with the $\rightarrow_{\beta\delta}$-relation and the set of rules δ as introduced in the lecture to implement Haskell.

Hints:

- You can use infix notation for predefined functions like `(&&)` in both simple Haskell and the lambda calculus.
- You do *not* have to use the transformation algorithms presented in the lecture. It is sufficient to just give an equivalent simple program and an equivalent lambda term.

b) Let

$$t = \lambda\ g\ n.\ \texttt{if}\ (n <= 0)\ 1\ (g\ n)$$

and

$$\begin{aligned}\delta = \ & \{\ \texttt{if True} \to \lambda x\ y.\, x,\\ & \quad \texttt{if False} \to \lambda x\ y.\, y,\\ & \quad \texttt{fix} \to \lambda f.\, f(\texttt{fix}\ f)\}\\ \cup\ & \{\ x <= y \to \texttt{True} \mid x \leq y \in \mathbb{Z}\}\\ \cup\ & \{\ x <= y \to \texttt{False} \mid x > y \in \mathbb{Z}\}\end{aligned}$$

Please reduce `fix` t 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

$$\lambda\ g\ n.\ \texttt{if}\ (n <= 0)\ 1\ (g\ n)$$

whenever possible.

c) Consider the representation of natural numbers in the pure lambda calculus presented in the lecture (i.e., $n \in \mathbb{N}$ is represented by the term $\overline{n} = \lambda\, f\; x.\; f^n\; x$) and the representation of Booleans (i.e., $\overline{\texttt{True}} = \lambda\, x\; y.x$ and $\overline{\texttt{False}} = \lambda\; x\; y.y$). Give a pure lambda term for the function `isZero`, such that $\texttt{isZero}(\lambda\, f\; x.\; f^n\; x)$ can be reduced to $\overline{\texttt{False}}$ if $n > 0$ and to $\overline{\texttt{True}}$ if $n = 0$.

Explain your solution shortly. You may give a reduction sequence as explanation.

Hints:

- A function $\lambda\; x.t$ where x does *not* occur as a free variable in the term t is a constant function. Then applying $\lambda\; x.t$ to a term multiple times still evaluates to t.

Exercise 4 (Type Inference): (15 points)

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

Hints:

- When writing $\mathcal{W}(A, t) = (\theta, t)$, you do not have to give the full substitution θ, but it is enough to give the parts of θ that concern the free variables in the type schemas of A.