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.

Solution:

a) No, because the most general type schema for this Haskell expression is non-flat, but such type schemata are not allowed in Haskell.

b) Counterexample: $A = \{a\}$ with $a \succ a$. Obviously, the relation is confluent and a does not have a normal form w.r.t. $\succ$.

c) Yes, e.g., the function $g : (\mathbb{Z}_\perp \to \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$ defined by

$$g(f) = \begin{cases} 0, & \text{if } f(x) \neq \perp_{\mathbb{Z}_\perp} \text{ for all } x \in \mathbb{Z} \\ \perp_{\mathbb{Z}_\perp}, & \text{otherwise} \end{cases}$$

is monotonic, but not continuous.

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.

```
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.

Solution:

a)
```
filterTrain :: (a -> Bool) -> Train a -> Train a
filterTrain _ Empty = Empty
filterTrain p (Locomotive t) = Locomotive (filterTrain p t)
filterTrain p (Wagon c t) = if (p c) then (Wagon c t') else t'
  where t' = filterTrain p t
```

b)
```
buildTrain :: [Cargo] -> Train Cargo
buildTrain cargo = filterTrain (\c -> case c of
                                  Goods x -> x <= 1000
                                  _ -> True)
                               (Locomotive (foldr Wagon Empty cargo))
```

c)
```
weight :: Train Cargo -> Int
weight = fold (\c res -> case c of
                 Goods x -> res + x
                 _ -> res)
              0
```

Exercise 3 (List Comprehensions): (3 + 3 + 5 = 11 points)

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

Hint: The function `mod :: Int -> Int -> 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 `divisors` even if you were not able to solve the previous exercise part.

Hint: The function `sum :: [Int] -> 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 `divisors` even if you were not able to solve exercise part (a). Moreover, you may use the function `sum` and the following functions:

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

  `subsequences [1,2,3] = [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`

Solution:

a)
```
divisors :: Int -> [Int]
divisors x = [y|y<-[1..x-1], x `mod` y == 0]
```

b)
```
perfect :: [Int]
perfect = [x|x<-[2..], sum (divisors x) == x]
```

c)
```
semiperfect :: [Int]
semiperfect = [x|x<-[2..], exists (\s -> sum s == x) (subsequences (divisors x))]
```

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 $\leq \subset \mathbb{N} \times \mathbb{N}$ 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.

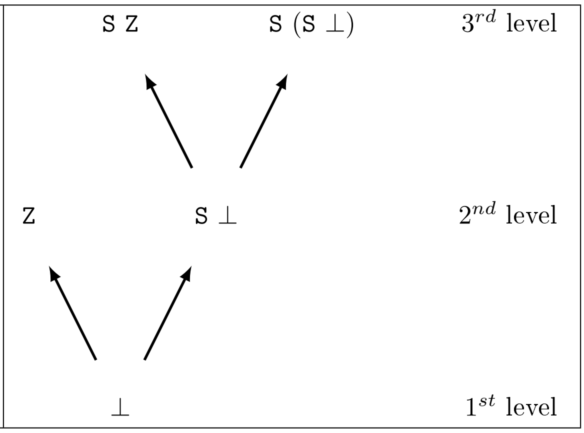
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for `Nats` on the right:

```
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.

d) Consider the definition for `Nats` from the previous exercise part, i.e., `data Nats = Z | S Nats`.

Moreover, consider the following Haskell function `f'`:

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

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

Solution:

a) i) 1) $\{$`[]`, `[[]]`, `[[[]]]`, $\ldots\}$

2) Consider the chain above. Since it contains infinitely many elements with increasing nesting level, its upper bounds have to have infinite nesting level. Since lists with infinite nesting level are not contained in $L_{[]}$, $\leq_n$ is not a cpo.

ii) 1) $\{$`[]`, `[0]`, `[0,0]`, $\ldots\}$

2) The relation $\leq_{len}$ is a cpo iff L_0 has a least element w.r.t. $\leq_{len}$ and every $\leq_{len}$-chain has a least upper bound in L_0. Obviously, the least element is the empty list `[]`. Let C be a chain. If C is finite, then the longest list in C is the least upper bound. Otherwise, the infinite list l_∞ containing only zeros (as defined by `zeros=0:zeros`) is the least upper bound. Thus, $\leq_{len}$ is a cpo.

b) i)

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

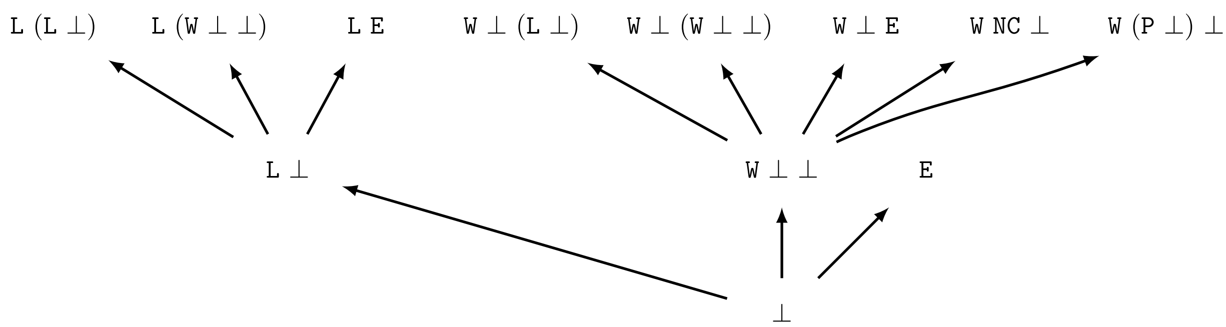
ii)

$$(\phi_{\texttt{ff}}^n(\bot))(x,y) = \begin{cases} 1 & \text{if } y = 0 \land 0 < n \\ x^y & \text{if } 0 < y < n \land x \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

iii)

$$(\mathsf{lfp}\ \phi_{\texttt{ff}})(x,y) = \begin{cases} 1 & \text{if } y = 0 \\ x^y & \text{if } 0 < y \land x \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

c)



d)

```
fNat = \x -> \y ->
          if (isaZ y) then 1
          else mult x (fNat x (argofS y))
```

Exercise 5 (Lambda Calculus): (4 + 8 = 12 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

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

and

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

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

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

whenever possible.

Solution:

a) $\mathtt{fix}\ (\lambda\, f\ x\ y.\, \mathtt{if}\ (y == 0)\ 1\ (x * (f\ x\ (y-1))))$

b)

$$
\begin{array}{ll}
 & \texttt{fix}\ t\ 0\ 0 \\
\to_\delta & (\lambda f.\,(f\ (\texttt{fix}\ f)))\ t\ 0\ 0 \\
\to_\beta & t\ (\texttt{fix}\ t)\ 0\ 0 \\
\to_\beta & (\lambda x\ y.\,\texttt{if}\ (y == 0)\ x\ ((\texttt{fix}\ t)\ (x+1)\ (y-1)))\ 0\ 0 \\
\to_\beta & (\lambda y.\,\texttt{if}\ (y == 0)\ 0\ ((\texttt{fix}\ t)\ (0+1)\ (y-1)))\ 0 \\
\to_\beta & \texttt{if}\ (0 == 0)\ 0\ ((\texttt{fix}\ t)\ (0+1)\ (0-1)) \\
\to_\delta & \texttt{if True}\ 0\ ((\texttt{fix}\ t)\ (0+1)\ (0-1)) \\
\to_\delta & (\lambda x\ y.\,x)\ 0\ ((\texttt{fix}\ t)\ (0+1)\ (0-1)) \\
\to_\beta & (\lambda y.\,0)\ ((\texttt{fix}\ t)\ (0+1)\ (0-1)) \\
\to_\beta & 0
\end{array}
$$

Exercise 6 (Type Inference): (10 points)

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

Solution:

$$
\begin{array}{l}
\mathcal{W}(A_0, \lambda f.f\,(f\,x)) \\
\quad \mathcal{W}(A_0 + \{f :: b_1\}, f\,(f\,x)) \\
\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, f) = (id, b_1) \\
\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, f\,x) \\
\quad\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, f) = (id, b_1) \\
\quad\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, x) = (id, b_2) \\
\quad\quad mgu(b_1, b_2 \to b_3) = [b_1/b_2 \to b_3] \\
\quad\quad = ([b_1/b_2 \to b_3], b_3) \\
\quad mgu(b_2 \to b_3, b_3 \to b_4) = [b_2/b_3, b_4/b_3] \\
\quad = ([b_1/b_3 \to b_3, b_2/b_3, b_4/b_3], b_3) \\
= ([b_1/b_3 \to b_3, b_2/b_3, b_4/b_3], (b_3 \to b_3) \to b_3)
\end{array}
$$