Exercise 1 (Quiz): (3 + 3 + 3 = 9 points)

a) Give a type declaration for `f` such that `(f True) (f 1)` is well typed in Haskell or explain why such a type declaration cannot exist.

b) Prove or disprove: If $\succ \subseteq A \times A$ is confluent, then each $a \in A$ has at most one normal form w.r.t. $\succ$.

c) What is the connection between monotonicity, continuity, and computability?

Solution:

a) `f :: a -> b -> Bool`

b) Let q_1 and q_2 be $\succ$-normal forms of t. Then$t \succ^* q_1$ and $t \succ^* q_2$. By confluence of $\succ$, there must be a q such that $q_1 \succ^* q$ and $q_2 \succ^* q$. Since q_1 and q_2 are normal forms, we get $q_1 = q = q_2$.

c) Every computable function is continuous and every continuous function is monotonic.

Exercise 2 (Programming in Haskell): (6 + 7 + 7 = 20 points)

We define a polymorphic data structure `Tree e` for binary trees whose nodes store values of type `e`.

```
data Tree e = Node e (Tree e) (Tree e) | Empty
```

The data structure `Forest e` is used to represent lists of trees.

```
type Forest e = [Tree e]
```

Furthermore, we define the following data structure:

```
data Animal = Squirrel | None
```

For example, `aForest` is a valid expression of type `Forest Animal`.

```
aForest = [Node Squirrel Empty (Node Squirrel Empty Empty), Node None Empty Empty]
```

In this exercise, you may use full Haskell and predefined functions from the Haskell Prelude.

a) Implement a function `hunt` together with its type declaration that removes all `Squirrel`s from a `Forest Animal`, i.e., each occurrence of a `Squirrel` should be replaced by `None`.

For example, `hunt aForest` should be evaluated to `[Node None Empty (Node None Empty Empty), Node None Empty Empty]`.

b) Implement a function `fold :: (e -> res -> res -> res) -> res -> Tree e -> res` to fold a `Tree`. The first argument of `fold` is the function that is used to combine the value of the current `Node` with the subresults obtained for the two direct subtrees of the current `Node`. The second argument of `fold` is the start value, i.e., the initial subresult. The third argument is the `Tree` that has to be folded. So for a `Tree t`, `fold f x t` replaces the constructor `Node` by `f` and the constructor `Empty` by `x`.

As an example, consider the following function:

```
count :: Animal -> Int -> Int -> Int
count Squirrel x y = x + y + 1
count None x y = x + y
```

Then `fold count 0 (Node Squirrel Empty (Node Squirrel Empty Empty))` should evaluate to 2, i.e., this application of `fold` counts all `Squirrels` in a `Tree`.

c) Implement a function `isInhabited` together with its type declaration which gets a `Forest Animal` as input and returns `True` if and only if there is a `Tree` in the `Forest` that contains a `Squirrel`. For the definition of `isInhabited`, use only one defining equation where the right-hand side contains a call to the function `fold`. Of course, you may (and have to) use the function `fold` even if you were not able to solve exercise part (b). Moreover, you may use the function `count` from exercise part (b).

Note that the function `fold` operates on a `Tree`, whereas the function `isInhabited` operates on a `Forest`!

Solution:

a)
```
hunt :: Forest Animal -> Forest Animal
hunt forest = map f forest where
  f Empty = Empty
  f (Node _ l r) = Node None (f l) (f r)
```

b)
```
fold :: (e -> res -> res -> res) -> res -> Tree e -> res
fold f x Empty = x
fold f x (Node v l r) = f v (fold f x l) (fold f x r)
```

c)
```
isInhabited :: Forest Animal -> Bool
isInhabited xs = sum (fold count 0 xs) > 0
```

Exercise 3 (List Comprehensions): (4 + 7 = 11 points)

In this exercise, you can assume that there exists a function `divisors :: Int -> [Int]` where, for any natural number `x` ≥ 2, `divisors x` computes the list of all its proper divisors (including 1, but excluding `x`). So for example, `divisors 6 = [1,2,3]`.

a) Write a Haskell expression in form of a list comprehension to compute all *amicable pairs of numbers*. A pair of natural numbers (x, y) with $x > y \geq 2$ is amicable if and only if the sum of the proper divisors of x is equal to y and the sum of the proper divisors of y is equal to x. For example, $(284, 220)$ is amicable:

- The proper divisors of 284 are $1, 2, 4, 71$, and 142 and their sum is 220.
- The proper divisors of 220 are $1, 2, 4, 5, 10, 11, 20, 22, 44, 55$, and 110 and their sum is 284.

In other words, give a list comprehension for a list that only contains amicable pairs of numbers and, for every amicable pair of numbers p, there is an $n \in \mathbb{N}$ such that the n^{th} element of the list is p.

Hint: The function `sum :: [Int] -> Int` computes the sum of a list of integers.

b) Write a Haskell expression in form of a list comprehension to compute all *practical numbers*. A natural number $x \geq 2$ is practical if and only if each smaller number $y \in \{1, \ldots, x-1\}$ is equal to the sum of some of x's proper divisors. For example, 6 is practical: Its proper divisors are $1, 2$, and 3 and we have $4 = 3 + 1$ and $5 = 3 + 2$.

In your solution, you may use the function `sum` and the following functions:

- The function `any :: (a -> Bool) -> [a] -> Bool` tests whether there is an element in the given list that satisfies the given predicate.
- The function `all :: (a -> Bool) -> [a] -> Bool` tests whether all elements in the given list satisfy 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) `[(x,y) | x<-[3..], y<-[2..x-1], sum (divisors x) == y, sum (divisors y) == x]`

b) `[x | x<-[2..], all (\y -> any (\s -> sum s == y) (subsequences (divisors x))) [2..x-1]]`

Exercise 4 (Semantics): (12 + 7 + 5 + 4 = 28 points)

a) i) Let $\mathbb{N}^\infty$ be the set of all infinite sequences of natural numbers (e.g., $[0, 0, 2, 2, 4, 4, \dots] \in \mathbb{N}^\infty$) and let $\leq_p \subseteq \mathbb{N}^\infty \times \mathbb{N}^\infty$ be the relation that compares infinite sequences of natural numbers by their *prefix sums*. The n^{th} prefix sum $p_n(s)$ for some $n \in \mathbb{N}$ of a sequence $s \in \mathbb{N}^\infty$ is the sum of the first n elements of s. We have $s \leq_p s'$ if and only if $s = s'$ or there is an $n \in \mathbb{N}$ such that $p_n(s) < p_n(s')$ and $p_m(s) = p_m(s')$ for all $m \in \{0, \dots, n-1\}$.

1) Prove that $\leq_p$ is transitive.

2) Give an example for an infinite chain in $(\mathbb{N}^\infty, \leq_p)$.

3) Prove or disprove: The partial order $\leq_p$ is complete on $\mathbb{N}^\infty$.

ii) Prove or disprove: The partial order $\leq$ is complete on $\mathbb{N}$. Here, $\leq$ is the usual "less than or equal" relation.

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

```haskell
f :: Int -> Int
f 0 = 1
f x = x * x * f (x - 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 of `ff`. You may use full Haskell for `ff`.

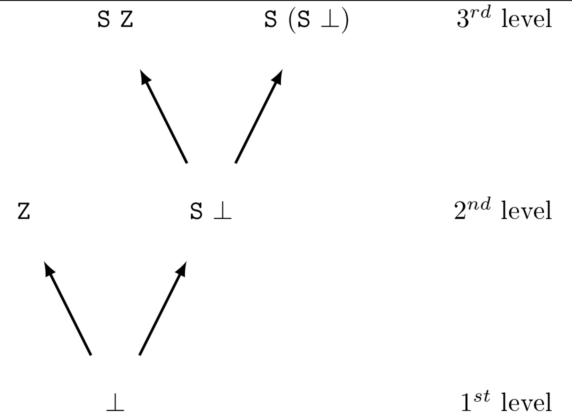
ii) Let $\phi_{\texttt{ff}}$ be the semantics of the function `ff`. Give the least fixpoint of $\phi_{\texttt{ff}}$ in closed form, i.e., give a non-recursive definition of the least fixpoint of $\phi_{\texttt{ff}}$.

Hint: For natural numbers x, the factorial function can be defined as follows:

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot (x-1)! & \text{if } x > 0 \end{cases}$$

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:

```haskell
data Nats = Z | S Nats
--------------------------------------
type Forest e = [Tree e]
data Tree e =
    Node e (Tree e) (Tree e)
    | Empty
```



Give a graphical representation of the first three levels of the domain for the type `Forest Int`. The third level contains the element `Empty:`$\bot$, for example.

d) Reconsider the definition for `Nats` from the previous exercise part, i.e., `data Nats = Z | S Nats`. Moreover, reconsider the function `f`:

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

Write a function `fNat :: Nats -> Nats` in **Simple** Haskell which, for natural numbers, computes the same result as the function `f`. That means, if `n >= 0` and `f n = x`, then we have `fNat (S^n Z) = S^x Z`. You can assume that there exists 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 $\texttt{isa}_{\underline{\texttt{constr}}}$ and $\texttt{argof}_{\underline{\texttt{constr}}}$. You do not have to use the transformation rules from the lecture, though.

Solution:

a) i) 1) Let $x \leq_p y \leq_p z$. If $x = y$ or $y = z$, we are done. Otherwise, there is an $n \in \mathbb{N}$ such that $p_n(x) < p_n(y)$ and $p_m(x) = p_m(y)$ for all $m \in \{1, \dots, n-1\}$ and an $n' \in \mathbb{N}$ such that $p_{n'}(y) < p_{n'}(z)$ and $p_m(y) = p_m(z)$ for all $m \in \{1, \dots, n'-1\}$. Let $n'' = min(n, n')$. Then $p_{n''}(x) < p_{n''}(z)$ and $p_m(x) \leq p_m(z)$ for all $m \in \{1, \dots, n''-1\}$ and thus $x \leq_p z$.

2) $\{[0,0,0,\dots],[1,0,0,\dots],[2,0,0,\dots],\dots\}$

3) Consider the chain above. Its least upper bound is $[\infty,0,0,\dots] \notin \mathbb{N}^\infty$. Thus, $\leq_p$ is not a cpo on $\mathbb{N}^\infty$.

ii) Consider the chain $\mathbb{N}$. Its least upper bound is $\infty \notin \mathbb{N}$. Thus, $\leq$ is not complete on $\mathbb{N}$.

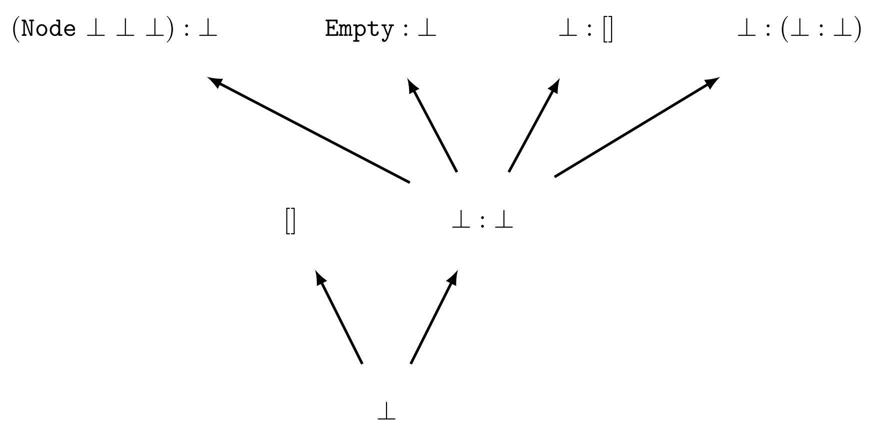
b) i)

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

ii)

$$(\mathsf{lfp}\ \phi_{\texttt{ff}})(x) = \begin{cases} (x!)^2 & \text{if } 0 \leq x \\ \bot & \text{otherwise} \end{cases}$$

c)



d)
```
fNat = \x ->
          if (isa_Z x) then 1
          else mult x (mult x (fNat (argof_S x)))
```

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

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

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

Please implement this function in the Lambda Calculus, i.e., give a term `t` such that, for all `x`, `y` $\in \mathbb{Z}$, `f x == y` if and only if `t x` can be reduced to `y` via WHNO-reduction with the $\to_{\beta\delta}$-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 $\texttt{t} = \lambda g\ x.\texttt{if}\ (x == 0)\ x\ (g\ x)$ and

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

Please reduce $\texttt{fix}\,\texttt{t}\,0$ 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 $\lambda g\ x.\texttt{if}\ (x == 0)\ x\ (g\ x)$ whenever possible.

Solution:

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

b)

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

Exercise 6 (Type Inference): (10 points)

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

Solution:

$\mathcal{W}(A_0, \lambda f.g\,(f\,x))$
$\quad\mathcal{W}(A_0 + \{f :: b_1\}, g\,(f\,x))$
$\quad\quad\mathcal{W}(A_0 + \{f :: b_1\}, g) = (id, b_2)$
$\quad\quad\mathcal{W}(A_0 + \{f :: b_1\}, f\,x)$
$\quad\quad\quad\mathcal{W}(A_0 + \{f :: b_1\}, f) = (id, b_1)$
$\quad\quad\quad\mathcal{W}(A_0 + \{f :: b_1\}, x) = (id, b_3)$
$\quad\quad mgu(b_1, b_3 \to b_4) = [b_1/b_3 \to b_4]$
$\quad\quad = ([b_1/b_3 \to b_4], b_4)$
$\quad mgu(b_2, b_4 \to b_5) = [b_2/b_4 \to b_5]$
$\quad = ([b_1/b_3 \to b_4, b_2/b_4 \to b_5], b_5)$
$= ([b_1/b_3 \to b_4, b_2/b_4 \to b_5], (b_3 \to b_4) \to b_5)$