Exercise 1 (2+2+2 points)

The following data structure represents binary trees only containing values at the leaves:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

Consider the tree $t$ of integers on the right-hand side. The representation of $t$ as an object of type `Tree Int` in Haskell would be:

```
Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)
```

![Binary Tree Diagram]

Implement the following functions in Haskell.

(a) The function `foldTree` of type $(a \rightarrow a \rightarrow a) \rightarrow (b \rightarrow a) \rightarrow Tree b \rightarrow a$ works as follows: `foldTree n l t` replaces all occurrences of the constructor `Node` in the tree $t$ by $n$ and it replaces all occurrences of the constructor `Leaf` in $t$ by $l$. So for the tree $t$ above, `foldTree (+) id t` should compute $(+) ((+) (id 1) (id 2)) (id 3)$ which finally results in 6. Here, `Node` is replaced by $(+)$ and `Leaf` is replaced by `id`.

(b) Use the `foldTree` function from (a) to implement the `maxTree` function which returns the largest (w.r.t. $>$) element of the tree. Apart from the function declaration, also give the most general type declaration for `maxTree`.

```
maxTree :: Ord a => Tree a -> a
maxTree = foldTree max id
```
(c) Consider the following data type declaration for natural numbers:

```haskell
data Nats = Zero | Succ Nats
```

A graphical representation of the first four levels of the domain for `Nats` could look like this:

```
Succ (Succ Zero)   Succ (Succ (Succ ⊥))
```
```
Succ Zero          Succ (Succ ⊥)
```
```
Zero               Succ ⊥
```
```
⊥
```

Sketch a graphical representation of the first three levels of the domain $D_{\text{Tree Bool}}$ for the data type `Tree Bool`. 
Exercise 2 (2+3 points)

Consider the following Haskell declarations for the `double` function:

```haskell
double :: Int -> Int
double (x+1) = 2 + (double x)
double _ = 0
```

(a) Give the Haskell declarations for the higher-order function `f_double` corresponding to `double`, i.e., the higher-order function `f_double` such that the least fixpoint of `f_double` is `double`. In addition to the function declaration(s), also give the type declaration of `f_double`. Since you may use full Haskell for `f_double`, you do not need to translate `double` into simple Haskell.

```haskell
f_double :: (Int -> Int) -> (Int -> Int)
f_double double (x+1) = 2 + (double x)
f_double double _ = 0
```

(b) We add the Haskell declaration `bot = bot`. For each \( n \in \mathbb{N} \) determine which function is computed by `f_double^n bot`. Here “`f_double^n bot`” represents the \( n \)-fold application of `f_double` to `bot`, i.e., it is short for `f_double (f_double \ldots (f_double bot)\ldots)`. Give the function in closed form, i.e., using a non-recursive definition.
Exercise 3 (3+3 points)

Let \(\subseteq\) be a complete order and let \(f\) be a function which is continuous (and, therefore, also monotonic).

Prove or disprove the following statements:

(a) \[\{ f^n(\bot) \mid n \in \{0, 1, 2, \ldots\} \}\] is a chain.

(b) \[\cup \{ f^n(\bot) \mid n \in \{0, 1, 2, \ldots\} \}\] is a fixpoint of \(f\).
Exercise 4 (3 points)

We define the following algebraic data type for lists:

data List a = Nil | Cons a (List a)

Write a program in simple Haskell which computes the function \( \text{sum :: List Int -> Int.} \) Here, \( \text{sum} \) adds all integers in a list of integers. For example, \( \text{sum (Cons 1 (Cons (-2) Nil))} \) should return \(-1\).

Your solution should use the functions defined in the transformation from the lecture such as \( \text{sel}_{n,i} \), \( \text{isa}_{\text{constr}} \), and \( \text{argof}_{\text{constr}} \). You do not have to use the transformation rules from the lecture, though.
Exercise 5 (2+3 points)

Consider the following data structure for natural numbers:

\[
data \text{Nats} = \text{Succ} \text{Nats} \mid \text{Zero}
\]

Let \( \delta \) be the set of rules from Definition 3.3.5, i.e., \( \delta \) contains at least the following rules:

\[
\begin{align*}
\text{fix} & \rightarrow \lambda f. f (\text{fix } f) \\
\text{if False} & \rightarrow \lambda x \ y. y \\
\text{isa}_\text{zero} (\text{Succ} (\text{Succ Zero})) & \rightarrow \text{False}
\end{align*}
\]

(a) Please translate the following Haskell-expression into a lambda term using \( \text{Lam} \). It suffices to give the result of the transformation.

\[
\text{let } g = \lambda x \rightarrow \text{if } (\text{isa}_\text{Zero } x) \text{ then } \text{Zero} \text{ else } \text{Succ } (g \ (\text{argof}_\text{Succ } x))
\]

\[
\text{in } g \ (\text{Succ } (\text{Succ Zero}))
\]

(b) Reduce the lambda term from (a) by WHNO-reduction with the \( \rightarrow_{\beta\delta} \)-relation. You do not have to give the intermediate steps but only the weak head normal form (which is not the same as the normal form).
Exercise 6 (4 points)

Use the type inference algorithm $W$ to determine the most general type of the following $\lambda$-term under the initial type assumption $A_0$. Show the results of all sub-computations and unifications, too. If the term is not well typed, show how and why the $W$-algorithm detects this.

$$\text{fix} (\lambda x. \text{Succ } x)$$

In this exercise, please use the initial type assumption $A_0$ as presented in the lecture. This type assumption contains at least the following:

$$A_0(\text{Succ}) = \text{Nats} \rightarrow \text{Nats}$$
$$A_0(\text{fix}) = \forall a. (a \rightarrow a) \rightarrow a$$