Exercise 1 (2+2+2 points)

The following data structure represents binary trees only containing values in the inner nodes:
```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Consider the tree \( t \) of integers on the right-hand side. The representation of \( t \) as an object of type \( \text{Tree Int} \) in Haskell would be:
```
Node (Node Leaf 2 Leaf) 1 Leaf
```

Implement the following functions in Haskell.

(a) Please implement the function \( \text{swapTree} \) which returns the tree where all left children have been swapped with the corresponding right children. When computing \( \text{swapTree} \ t \) one would obtain the tree on the right-hand side. Apart from the function declaration, also give the most general type declaration for \( \text{swapTree} \). You may not use any higher-order functions.

\[
\text{swapTree} :: \text{Tree a} \rightarrow \text{Tree a}
\]

\[
\text{swapTree \ Leaf} = \text{Leaf}
\]

\[
\text{swapTree \ (Node \ l \ x \ r)} = \text{Node \ (swapTree \ r) \ x \ (swapTree \ l)}
\]
(b) The function `foldTree` of type `(a -> b -> a -> a) -> a -> Tree b -> a` works as follows: 

\[ \text{foldTree } n \ l \ t \] 
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`. Suppose there is the function `add`:

\[
\text{add :: Int -> Int -> Int -> Int} \\
\text{add } x \ y \ z = x + y + z
\]

For the tree `t` from (a), “foldTree add 0 t” should compute:

\[
\text{foldTree add 0 (Node (Node Leaf 2 Leaf) 1 Leaf)} \\
= \text{add (add 0 2 0 ) 1 0} \\
= 3
\]

Here, `Node` is replaced by `add` and `Leaf` is replaced by 0.

Now use the `foldTree` function to implement the `swapTree` function again.

\[
\text{swapTree} = \text{foldTree } (\backslash x \ y \ z \rightarrow \text{Node } z \ y \ x) \ \text{Leaf}
\]
(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 for the data type `Tree Bool`.

```
Node (Node ⊥ ⊥) ⊥ ⊥   Node ⊥ True ⊥

Node ⊥ False ⊥

Node Leaf ⊥ ⊥

Leaf
```

```
Node ⊥ ⊥ (Node ⊥ ⊥ ⊥)   Node ⊥ ⊥ Leaf

Node ⊥ ⊥ ⊥
```
Exercise 2 (2+3 points)

Consider the following Haskell declarations for the half function:

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

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

```haskell
f.half :: (Int -> Int) -> (Int -> Int)
f.half half (x+2) = 1 + (half x)
f.half half _ = 0
```

(b) We add the Haskell declaration \texttt{bot = bot}. For each $n \in \mathbb{N}$ determine which function from $\mathbb{Z}_\perp$ to $\mathbb{Z}_\perp$ is computed by $f.half^n$ bot. Here \texttt{“f.half” bot”} represents the $n$-fold application of \texttt{f.half} to \texttt{bot}, i.e., it denotes $f.half (f.half \ldots (f.half bot)\ldots)$. Give the function computed by \texttt{“f.half” bot”} in closed form, i.e., using a non-recursive definition.

$$ (f.half^n(\bot))(x) = \begin{cases} \lfloor \frac{x}{2^n} \rfloor, & \text{if } 1 < x < 2n \\ 0, & \text{if } x \leq 1 \land n > 0 \\ \bot, & \text{if } n = 0 \lor x = \bot \lor x \geq 2n \end{cases} $$
Exercise 3 (3+3 points)

Let \(\sqsubseteq\) 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.

We prove \(f^n(\bot) \sqsubseteq f^{n+1}(\bot)\) for all \(n \in \{0, 1, 2, \ldots\}\) by induction on \(n\).

- \(n = 0\): By definition we have \(\bot \sqsubseteq f(\bot)\).
- \(n \rightarrow n + 1\): The function \(f\) is continuous and therefore also monotonic.
  Thus, \(f^n(\bot) \sqsubseteq f^{n+1}(\bot)\) implies \(f^{n+1}(\bot) \sqsubseteq f^{n+2}(\bot)\).

(b) \(\cup \{ f^n(\bot) \mid n \in \{0, 1, 2, \ldots\} \}\) is a fixpoint of \(f\).

\[
\begin{align*}
f(\cup\{f^n(\bot) \mid n \in \{0, 1, 2, \ldots\}\}) & \overset{\text{\(f\) continuous}}{=} \cup f(\{f^n(\bot) \mid n \in \{0, 1, 2, \ldots\}\}) \\
& = \cup\{f^{n+1}(\bot) \mid n \in \{0, 1, 2, \ldots\}\} \\
& = \cup\{f^n(\bot) \mid n \in \{1, 2, \ldots\}\} \\
& = \cup(\{f^n(\bot) \mid n \in \{1, 2, \ldots\}\} \cup \{\bot\}) \\
& = \cup(\{f^n(\bot) \mid n \in \{1, 2, \ldots\}\} \cup \{f^0(\bot)\}) \\
& = \cup\{f^n(\bot) \mid n \in \{0, 1, 2, \ldots\}\}
\end{align*}
\]
Exercise 4 (3 points)

We define the following algebraic data type for lists:

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

Write declarations in **simple** Haskell for the function `maxList :: List Int -> Int`. Here, for empty lists the function should return `bot`. For non-empty lists, `maxList` should return the maximum of that list. For example, `maxList (Cons 1 (Cons (-2) Nil))` should return `1`.

Your solution should use the functions defined in the transformation from the lecture such as `sel_n,i`, `isa_constr`, `argof_constr`, and `bot`. You do not have to use the transformation rules from the lecture, though. Additionally, you may use the built-in function `max :: Int -> Int -> Int` for computing the maximum of two integers.

```haskell
maxList = \xs -> if (isaCons xs)
    then if (isaNil (sel_2_2 (argofCons xs)))
        then sel_2_1 (argofCons xs)
        else max (sel_2_1 (argofCons xs)) (maxList (sel_2_2 (argofCons xs)))
    else bot
```
Exercise 5 (2+4 points)

Consider the following data structures for natural numbers and polymorphic lists:

```haskell
data Nats = Zero | Succ Nats
data List a = Nil | Cons a (List a)
```

Let $\delta$ be the set of rules from Definition 3.3.5, i.e., $\delta$ contains among others the following rules:

$$
\begin{align*}
\text{fix} & \rightarrow \lambda f. f (\text{fix } f) \\
\text{if } \text{True} & \rightarrow \lambda x y. x \\
\text{isa}_{N_1} \text{ Nil} & \rightarrow \text{True}
\end{align*}
$$

(a) Please translate the following Haskell-expression into an equivalent lambda term (e.g., using $\mathcal{Lam}$). It suffices to give the result of the transformation.

```haskell
let length = \xs \rightarrow \text{if (isa}_{N_1} \xs \text{ then Zero}
else \text{Succ (length (sel}_{2,2} \text{ (argof}_{\text{Cons} \xs)}))}
in length
```

$$
\text{fix } (\lambda \text{length } \xs. \text{if (isa}_{N_1} \xs \text{ Zero (Succ (length (sel}_{2,2} \text{ (argof}_{\text{Cons} \xs)})))}))
$$
(b) Let “fix $t$” be the lambda term from (a). Please reduce “(fix $t$) Nil” by WHNO-reduction with the $\rightarrow_{\beta\delta}$-relation. You have to give all intermediate steps until one reaches weak head normal form.

We have $t = (\lambda length \text{ } xs. \text{ if } (\text{isa}_{\text{Nil}} \text{ } xs) \text{ Zero } (\text{Succ } (\text{length } (\text{sel}_{2,2} (\text{argof}_{\text{Cons}} \text{ } xs))))))$

\[
\begin{align*}
\text{fix } t \text{ Nil} \\
&\rightarrow_\delta (\lambda f. f (\text{fix } f)) \text{ Nil} \\
&\rightarrow_\beta t (\text{fix } t) \text{ Nil} \\
&\rightarrow_\beta (\lambda xs. \text{ if } (\text{isa}_{\text{Nil}} \text{ } xs) \text{ Zero } (\text{Succ } (\text{fix } t (\text{sel}_{2,2} (\text{argof}_{\text{Cons}} \text{ } xs)))))) \text{ Nil} \\
&\rightarrow_\beta \text{ if } (\text{isa}_{\text{Nil}} \text{ } \text{Nil}) \text{ Zero } (\text{Succ } (\text{fix } t (\text{sel}_{2,2} (\text{argof}_{\text{Cons}} \text{ } \text{Nil})))) \\
&\rightarrow_\delta \text{ if True Zero } (\text{Succ } (\text{fix } t (\text{sel}_{2,2} (\text{argof}_{\text{Cons}} \text{ } \text{Nil})))) \\
&\rightarrow_\delta (\lambda x y. x) \text{ Zero } (\text{Succ } (\text{fix } t (\text{sel}_{2,2} (\text{argof}_{\text{Cons}} \text{ } \text{Nil})))) \\
&\rightarrow_\beta (\lambda y. \text{ Zero}) (\text{Succ } (\text{fix } t (\text{sel}_{2,2} (\text{argof}_{\text{Cons}} \text{ } \text{Nil})))) \\
&\rightarrow_\beta \text{ Zero}
\end{align*}
\]
Exercise 6 (4 points)

Use the type inference algorithm \( \mathcal{W} \) 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 \( \mathcal{W} \)-algorithm detects this.

\[
\lambda f. f \ (\text{Succ Zero})
\]

The initial type assumption \( A_0 \) contains at least the following:

\[
\begin{align*}
A_0(f) &= \forall a. a \\
A_0(\text{Succ}) &= \text{Nats} \rightarrow \text{Nats} \\
A_0(\text{Zero}) &= \text{Nats}
\end{align*}
\]

\[
W(A_0, \lambda f. f \ (\text{Succ Zero}))
\]

\[
W(A_0 + \{f :: b_0\}, f \ (\text{Succ Zero}))
\]

\[
W(A_0 + \{f :: b_0\}, f)
= (\text{id, } b_0)
\]

\[
W(A_0 + \{f :: b_0\}, \text{Succ})
= (\text{id, } (\text{Nats} \rightarrow \text{Nats}))
\]

\[
W(A_0 + \{f :: b_0\}, \text{Zero})
= (\text{id, } \text{Nats})
\]

building mgu of \((\text{Nats} \rightarrow \text{Nats})\) and \((\text{Nats} \rightarrow b_1) = [b_1/\text{Nats}]\)

\[
= ([b_1/\text{Nats}], \text{Nats})
\]

building mgu of \(b_0\) and \((\text{Nats} \rightarrow b_2) = [b_0/(\text{Nats} \rightarrow b_2)]\)

\[
= ([b_1/\text{Nats}, b_0/(\text{Nats} \rightarrow b_2)], b_2)
\]

\[
= ([b_1/\text{Nats}, b_0/(\text{Nats} \rightarrow b_2)], ((\text{Nats} \rightarrow b_2) \rightarrow b_2))
\]

Resulting type: \(((\text{Nats} \rightarrow b_2) \rightarrow b_2)\)