**Exercise 1 (4 + 5 + 4 + 6 + 5 = 24 points)**

The following data structure represents polymorphic binary trees that contain values only in special Value nodes that have a single successor:

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

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

```
(Node (Value 'a' (Value 'b' Leaf)) (Node (Node Leaf Leaf) (Value 'c' Leaf)))
```

Implement the following functions in Haskell.

(a) The function `foldTree` of type

```
(a -> b -> b) -> (b -> b -> b) -> b -> Tree a -> b
```

works as follows: `foldTree f g h x` replaces all occurrences of the constructor `Value` in the tree $x$ by $f$, it replaces all occurrences of the constructor `Node` in $x$ by $g$, and it replaces all occurrences of the constructor `Leaf` in $x$ by $h$. So for the tree $t$ above,

```
foldTree (:) (++) [] t
```

should compute

```
((++) ((:) 'a' ((:) 'b' [])) ((++) ((+) [] []) ((:) 'c' []))
```

which in the end results in "abc" (i.e., in the list ['a', 'b', 'c']). Here, `Value` is replaced by (:), `Node` is replaced by (++) , and `Leaf` is replaced by [] .

```
foldTree f g h (Value n x) = f n (foldTree f g h x)
foldTree f g h (Node x y) = g (foldTree f g h x) (foldTree f g h y)
foldTree _ _ h Leaf = h
```
(b) Use the `foldTree` function from (a) to implement the `average` function which has the type `Tree Int -> Int` and returns the average of the values that are stored in the tree. This should be accomplished as follows:

- Use `foldTree` with suitable functions as arguments in order to compute the *sum* of the values stored in the trees.
- Use `foldTree` with suitable functions as arguments in order to compute the *number of Value-objects in the tree*.
- Perform integer division with the pre-defined function `div :: Int -> Int -> Int` on these values to obtain the result.

Here your function is required to work correctly only on those trees that contain the constructor `Value` at least once.

\[
\text{average } t = \text{div} \left( \text{foldTree } (+) (+) 0 \ t \right) \left( \text{foldTree } (\lambda x \ y \to y+1) (+) 0 \ t \right)
\]
(c) Consider the following data type declaration for natural numbers:

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

A graphical representation of the first four levels of the domain for \text{Nats} could look like this:

\[
\text{Succ (Succ Zero)} \quad \text{Succ (Succ (Succ ⊥))}
\]

\[
\text{Succ Zero} \quad \text{Succ (Succ ⊥)}
\]

\[
\text{Zero} \quad \text{Succ ⊥}
\]

Sketch a graphical representation of the first three levels of the domain for the data type \text{Tree Bool}.
(d) We call a list $ys$ of integers an $n$-times even product of a list $xs$ if $ys$ has length $n$ and if all elements of $ys$ are even numbers that occur in $xs$. The goal of this exercise is to write a function `evenProducts :: [Int] -> Int -> [[Int]]` that takes a list of integers $xs$ and a natural number $n$ and returns a list that contains all $n$-times even products of $xs$. For example, `evenProducts [4,5,6] 2 = [[4,4], [4,6], [6,4], [6,6]]`.

The following declarations are already given:

```haskell
evenProducts :: [Int] -> Int -> [[Int]]
evenProducts xs 0 = []
evenProducts xs 1 = map (\z -> [z]) (filter even xs)
```

Please give the declaration of `evenProducts` for the missing case of numbers that are at least 2. Perform your implementation only with the help of a list comprehension, i.e., you should use exactly one declaration of the following form:

```haskell
evenProducts xs (n+2) = [ ... | ... ]
```

```haskell
evenProducts xs (n+2) = [ y:ys | y <- xs, even y, ys <- evenProducts xs (n+1) ]
```
(e) We define the \textit{level} \(n\) of a tree as the list of those values that are at distance \(n\) from the root of the tree. Here, the root node has distance 0 from the root, and a non-root node has distance \(n + 1\) from the root if its parent node has distance \(n\) from the root.

Write a Haskell function \textbf{level} :: Tree \(a\) \(
\rightarrow\) \textbf{Int} \(
\rightarrow\) \[\textbf{a}\] which, given a tree \(t\) and a natural number \(n\), computes the list of all values in \(t\) that occur there at level \(n\) (with repetition, i.e., a value should appear in the result list as many times as it appears on level \(n\)).

As an example, consider again the tree \(t\) from the beginning of the exercise. Here we have \textbf{level} \(t\) 2 = [’b’,’c’] and \textbf{level} \(t\) 7 = [].

\begin{verbatim}
level :: Tree a -> Int -> [a]
level t 0 = case t of Value x u -> [x]
                 _ -> []
level t (n+1) = case t of Leaf -> []
                   Value x u -> level u n
                   Node u v -> (level u n) ++ (level v n)
\end{verbatim}
Exercise 2 (4 + 5 = 9 points)

Consider the following Haskell declarations for the `fib` function, which for a natural number \( x \) computes the value `fibonacci(x)`:

\[
\begin{align*}
\text{fib} & : \text{Int} \rightarrow \text{Int} \\
\text{fib} 0 & = 0 \\
\text{fib} 1 & = 1 \\
\text{fib} (x+2) & = \text{fib} (x+1) + \text{fib} x
\end{align*}
\]

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

\[
\begin{align*}
f_fib & : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \\
f_fib \text{ fib} 0 & = 0 \\
f_fib \text{ fib} 1 & = 1 \\
f_fib \text{ fib} (x+2) & = \text{fib} (x+1) + \text{fib} x
\end{align*}
\]

(b) We add the Haskell declaration `bot = bot`. For each \( n \in \mathbb{N} \) please determine which function is computed by `f_fib^n bot`. Here “\( f_fib^n \ bot \)” represents the \( n \)-fold application of `f_fib` to `bot`, i.e., it is short for `f_fib (f_fib ... (f_fib bot)...) \( n \) times`.

Let \( f_n : \mathbb{Z} \rightarrow \mathbb{Z} \) be the function that is computed by `f_fib^n bot`.

Give \( f_n \) in closed form, i.e., using a non-recursive definition. In this definition, you may use the function `fibonacci : \mathbb{N} \rightarrow \mathbb{N}` where `fibonacci(x)` computes the \( x \)-th Fibonacci number. Here it suffices to give the result of your calculations. You do not need to present any intermediate steps.

\[
(f_fib^n(\bot))(x) = \begin{cases} 
\text{fibonacci}(x), & \text{if } n > 0 \text{ and } 0 \leq x \leq n \\
\bot, & \text{otherwise}
\end{cases}
\]
Exercise 3 \((3 + 3 = 6 \text{ points})\)

Let \(D_1, D_2, D_3\) be domains with corresponding complete partial orders \(\sqsubseteq_{D_1}, \sqsubseteq_{D_2}, \sqsubseteq_{D_3}\). As we know from the lecture, then also \(\sqsubseteq_{(D_2 \times D_3)}\) is a complete partial order on \((D_2 \times D_3)\).

Now let \(f : D_1 \rightarrow D_2\) and \(g : D_1 \rightarrow D_3\) be functions.
We then define the function \(h : D_1 \rightarrow (D_2 \times D_3)\) via \(h(x) = (f(x), g(x))\).

(a) Prove or disprove: If \(f\) and \(g\) are strict functions, then also \(h\) is a strict function.

The statement does not hold. Consider the following counterexample: \(D_1 = D_2 = D_3 = B\) and \(f = g = \bot_{B \rightarrow B}\). Obviously \(f\) and \(g\) are strict functions, i.e., \(f(\bot_B) = g(\bot_B) = \bot_B\). However, we have \(h(\bot_B) = (\bot_B, \bot_B) \neq \bot_{(B \times B)}\).

(b) Prove or disprove: If \(f\) and \(g\) are monotonic functions, then also \(h\) is a monotonic function.

Let \(x \sqsubseteq_D y\). Then we have:

\[
\begin{align*}
h(x) & = (f(x), g(x)) & \text{\(f\) and \(g\) are monotonic, def. of \(\sqsubseteq_{(D_2 \times D_3)}\)} \\
\sqsubseteq_{(D_2 \times D_3)} & (f(y), g(y)) \\
& = h(y)
\end{align*}
\]

Hence, also \(h\) is monotonic. \(\square\)
Exercise 4 \((4 + 5 = 9\) points\)

Consider the following data structure for polymorphic lists:

\[
data\ \text{List}\ a = \text{Nil} \mid \text{Cons}\ a\ (\text{List}\ a)
\]

(a) Please translate the following Haskell expression into an equivalent lambda term (e.g., using \textit{Lam}). Recall that pre-defined functions like \texttt{odd} or \texttt{(+) are translated into constants of the lambda calculus. It suffices to give the result of the transformation.}

\[
\text{let } f = \lambda x \to \text{if } (\text{odd} x) \text{ then } (\lambda y \to x) \text{ else } f ((+) x 3) \\
in f
\]

\[
\text{fix } (\lambda f. \text{if } (\text{odd} x) (\lambda y. f ((+) x 3)))
\]
(b) Let $\delta$ be the set of rules for evaluating the lambda terms resulting from Haskell, i.e., $\delta$ contains at least the following rules:

\[
\begin{align*}
\text{fix} & \rightarrow \lambda f. f (\text{fix } f) \\
\text{times } 3 \ 2 & \rightarrow 6
\end{align*}
\]

Now let the lambda term $t$ be defined as follows:

\[
t = (\lambda x. (\text{fix } \lambda g. x)) (\lambda z. (\text{times } 3 \ 2))
\]

Please reduce the lambda term $t$ by WHNO-reduction with the $\rightarrow_{\beta\delta}$-relation. You have to give all intermediate steps until you reach weak head normal form (and no further steps).

\[
\begin{align*}
(\lambda x. (\text{fix } \lambda g. x)) (\lambda z. (\text{times } 3 \ 2)) & \rightarrow_{\beta} \text{fix } (\lambda g. \lambda z. (\text{times } 3 \ 2)) \\
& \rightarrow_{\delta} (\lambda f. f (\text{fix } f)) (\lambda g. \lambda z. (\text{times } 3 \ 2)) \\
& \rightarrow_{\beta} (\lambda g. \lambda z. (\text{times } 3 \ 2)) (\text{fix } (\lambda g. \lambda z. (\text{times } 3 \ 2))) \\
& \rightarrow_{\beta} \lambda z. (\text{times } 3 \ 2)
\end{align*}
\]
Exercise 5 (10 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.

$$((\text{Cons } \lambda x. x) \; y)$$

The initial type assumption $A_0$ contains at least the following:

$$
\begin{align*}
A_0(\text{Cons}) &= \forall a. (a \rightarrow (\text{List } a \rightarrow \text{List } a)) \\
A_0(x) &= \forall a. a \\
A_0(y) &= \forall a. a
\end{align*}
$$

\[
\begin{align*}
\mathcal{W}(A_0, ((\text{Cons } \lambda x. x) \; y)) \\
\mathcal{W}(A_0, (\text{Cons } \lambda x. x)) \\
\mathcal{W}(A_0, \text{Cons}) &= (id, (b_1 \rightarrow (\text{List } b_1 \rightarrow \text{List } b_1))) \\
\mathcal{W}(A_0, \lambda x. x) \\
&= (id, b_2) \\
&= (id, (b_2 \rightarrow b_2)) \\
mgu((b_1 \rightarrow (\text{List } b_1 \rightarrow \text{List } b_1)), ((b_2 \rightarrow b_2) \rightarrow b_3)) \\
&= ([ b_1/(b_2 \rightarrow b_2), b_3/(\text{List } b_2 \rightarrow \text{List } b_2) \rightarrow \text{List } (b_2 \rightarrow b_2)] \\
\mathcal{W}(A_0, y) \\
&= (id, b_4) \\
mgu((\text{List } (b_2 \rightarrow b_2) \rightarrow \text{List } (b_2 \rightarrow b_2)), (b_4 \rightarrow b_5)) = [ b_4/\text{List } (b_2 \rightarrow b_2), b_5/\text{List } (b_2 \rightarrow b_2)] \\
&= ([ b_1/(b_2 \rightarrow b_2), b_3/(\text{List } b_2 \rightarrow \text{List } b_2) \rightarrow \text{List } (b_2 \rightarrow b_2)), b_4/\text{List } (b_2 \rightarrow b_2), b_5/\text{List } (b_2 \rightarrow b_2)], \\
&\text{List } (b_2 \rightarrow b_2) \\
\end{align*}
\]

Resulting type: $\text{List } (b_2 \rightarrow b_2)$