Exercise 1 \((4 + 5 + 4 + 5 + 6 = 24 \text{ points})\)

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

\[
\text{data Tree } a = \text{Leaf} \mid \text{Node} (\text{Tree } a) (\text{Tree } a) \mid \text{Value } a (\text{Tree } a)
\]

Consider the tree \(t\) of characters on the right-hand side. The representation of \(t\) as an object of type \texttt{Tree Char} in Haskell would be:

\[
(\text{Node} (\text{Value} 'a' (\text{Value} 'b' \text{Leaf})) (\text{Node} (\text{Node} \text{Leaf} \text{Leaf}) (\text{Value} 'c' \text{Leaf})))
\]

Implement the following functions in Haskell.

(a) The function \texttt{foldTree} of type

\[
(a \to b \to b) \to (b \to b \to b) \to b \to \text{Tree } a \to b
\]

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

\[
\text{foldTree } (:) \text{ ((++) } [] \text{ ) } t
\]

should compute

\[
((\text{++) } ((:) 'a' ((:) 'b' []))) ((\text{++) } ((+) [] [])) ((:) 'c' []))
\]

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

\[
\text{foldTree } f\ g\ h\ (\text{Value } n\ x) = f\ n\ (\text{foldTree } f\ g\ h\ x)
\]

\[
\text{foldTree } f\ g\ h\ (\text{Node } x\ y) = g\ (\text{foldTree } f\ g\ h\ x)\ (\text{foldTree } f\ g\ h\ y)
\]

\[
\text{foldTree } \_\_\ h\ \text{Leaf} = h
\]
(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:

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

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) Write a Haskell function `printStars` that first reads a string from the user, then prints this string on the console, converts the string to a number n (using the pre-defined function `read`) and in the end also prints n times the character '⋆' on the console. Also give the type declaration for your function.

You may use the do-notation, but you are not obliged to use it. You do not have to check whether the input string is really a number. Some of the following pre-defined functions can be helpful:

- `getline :: IO String` reads a string from the user
- `read :: String -> Int` converts a string to a number
- `putStr :: String -> IO ()` writes a string to the console

An example run should look as given below. Here the string “7” was read from the user.

```
Main> printStars
7
7******
```

-- without do-notation
```
printStars :: IO ()
printStars = getline >>= \s -> putStr s >> putStr (take (read s) (repeat '⋆'))
```

-- alternative: with do-notation
```
printStars2 :: IO ()
printStars2 = do s <- getline
  putStr s
  putStr (take (read s) (repeat '⋆'))
```
(e) 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:

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

```
evenProducts xs (n+2) = [ y:ys | y <- xs, even y, ys <- evenProducts xs (n+1)]
```
Exercise 2 (4 + 5 = 9 points)

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

```haskell
defib :: Int -> Int
defib 0 = 0
defib 1 = 1
defib (x+2) = fib (x+1) + fib x
```

(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.

```haskell
f_fib :: (Int -> Int) -> (Int -> Int)
f_fib fib 0 = 0
f_fib fib 1 = 1
f_fib fib (x+2) = fib (x+1) + fib x
```

(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 $f_fib \ (f_fib \ (\ldots \ (f_fib \ bot)\ldots))$.

Let $f_n : \mathbb{Z}_\bot \to \mathbb{Z}_\bot$ 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} \to \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 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 $\sqsubseteq_{(D_2 \times D_3)\perp}$ is a complete partial order on $(D_2 \times D_3)\perp$.

Now let $f : D_1 \to D_2$ and $g : D_1 \to D_3$ be functions.
We then define the function $h : D_1 \to (D_2 \times D_3)\perp$ 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: $D_1 = D_2 = D_3 = \mathbb{B}_\perp$ and $f = g = \bot_{\mathbb{B}_\perp} : \mathbb{B}_\perp \to \mathbb{B}_\perp$. Obviously $f$ and $g$ are strict functions, i.e., $f(\bot_{\mathbb{B}_\perp}) = g(\bot_{\mathbb{B}_\perp}) = \bot_{\mathbb{B}_\perp}$. However, we have $h(\bot_{\mathbb{B}_\perp}) = (\bot_{\mathbb{B}_\perp}, \bot_{\mathbb{B}_\perp}) \neq \bot_{(\mathbb{B}_\perp \times \mathbb{B}_\perp)\perp}$.

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

Let $x \sqsubseteq_{D_1} y$. Then we have:

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

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

Consider the following data structure for polymorphic lists:

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

(a) Please translate the following Haskell expression into an equivalent lambda term (e.g., using Lam). Recall that pre-defined functions like odd or (+) are translated into constants of the lambda calculus.

It suffices to give the result of the transformation.

```haskell
let f = \x -> if (odd x) then (\y -> x) else f ((+) x 3)
in f
```

```
fix (λf. x. if (odd x) (λy. x) (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}) \\
\mathcal{W}(A_0, \lambda x. x) \\
\mathcal{W}(A_0 + \{x : b_2\}, x) \\
= (id, b_2) \\
\mathcal{W}(A_0, y) \\
= (id, b_4)
\end{align*}

$$
\begin{align*}
\text{mgu}( (b_1 \rightarrow (\text{List } b_1 \rightarrow \text{List } b_1)), ((b_2 \rightarrow b_2) \rightarrow b_3) ) &= [ b_1/(b_2 \rightarrow b_2), b_3/(\text{List } b_2 \rightarrow \text{List } b_2) \rightarrow \text{List } (b_2 \rightarrow b_2) ] \\
\mathcal{W}(A_0, \text{List}(b_2 \rightarrow b_2)) &= ( b_1/(b_2 \rightarrow b_2), b_3/(\text{List } b_2 \rightarrow \text{List } b_2) \rightarrow \text{List } (b_2 \rightarrow b_2) ) \\
\end{align*}
$$

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