Exercise 1 (Programming in Haskell): (6 + 6 + 7 + 16 = 35 points)

We define a polymorphic data structure `Warehouse` to represent a warehouse that can contain goods.

```
data Warehouse a
    = Corridor (Warehouse a) (Warehouse a)
    | Shelf [a] deriving Show
```

The data structure `Goods` is used to represent different types of goods.

```
data Goods
    = NoGoods | Box | Barrel deriving (Show, Eq)
```

For example, `aWarehouse` is a valid expression of type `Warehouse Goods`.

```
aWarehouse = Corridor (Shelf [Box, Box]) (Corridor (Shelf [Barrel, NoGoods]) (Shelf [Box]))
```

The following function can be used to *fold* a `Warehouse`:

```
fold :: (a -> a -> a) -> a -> Warehouse a -> a
fold f res (Shelf xs) = foldr f res xs
fold f res (Corridor left right) = f (fold f res left) (fold f res right)
```

In the following exercises, you are allowed to use the functions given above, functions implemented in preceding parts of the exercise, and predefined functions from the Haskell-Prelude. Moreover, you are always allowed to implement additional auxiliary functions.

a) Implement a function `buildWarehouse :: [a] -> Int -> Warehouse a` that gets a list and a number indicating how many items may be stored in one shelf. The function should return a warehouse containing all items in the input list, distributed on the minimal number of shelves possible while respecting the limit given by the second input.

If the input list is the empty list, `Shelf []` should be returned. If the integer argument is less than 1 the function may behave arbitrarily.

For example `buildWarehouse [Box, Box, Barrel, NoGoods, Box] 2` should result in a warehouse with 3 shelves which together contain all elements from the list. A possible result would be `aWarehouse`, as defined above.

b) Implement a function `mapWarehouse` together with its type declaration (`mapWarehouse :: ...`). The function `mapWarehouse` gets a unary function and a `Warehouse`, and applies the function to every element stored in the lists of the `Shelf` objects of the `Warehouse`. For example, the expression `mapWarehouse (\x -> if x == Box then NoGoods else x)` returns a function that replaces all `Box`es in a `Warehouse` by `NoGoods`.

c) Implement a function `inventory :: Warehouse Goods -> (Int, Int)` that returns a tuple containing the number of `Box`es and `Barrel`s in the input `Warehouse`. For example `inventory aWarehouse` should yield `(3, 1)`.

Hints:

You may use `fold` given above and `mapWarehouse` from the previous task (even if you have not solved the previous task).

d) In this part of the exercise, you should implement a program that controls a forklift. The goal of the forklift is to store `Goods` in a `Warehouse`. Goods can only be placed in free spaces indicated by `NoGoods` or stacked on top of a shelf.

Implement a function `storeGoods :: Warehouse Goods -> Goods -> IO ()`. Its input is a `Warehouse` that contains `Goods`. It processes the `Warehouse` as follows:

For a `Corridor`, it prints `"Do you want to turn left or right? (l|r)"`. If the user answers `"l"`, it proceeds with the first argument of the `Corridor`, if the answer is `"r"` it continues with the second argument of `Corridor`, otherwise an error message `"Incorrect input"` is printed and the question is repeated.

Once a `Shelf` is encountered the user is asked `"How high do you want to raise the fork?"` and an integer `k` is read. The program then distinguishes the following cases:

- The number `k` is less than 0: An error message `"Your forklift fell over"` is printed.
- The number `k` is a position of the list in the `Shelf`: (Here an object `Shelf [x_0, ..., x_n]` has positions `0`, ..., `n` and the object at position `i` is `x_i`.) If the list contains `NoGoods` at position `k`, the message `"Inserted the <Goods>"` is printed where `<Goods>` is replaced by `Box` or `Barrel` depending on the input, otherwise the message `"No room in the shelf"` is printed.
- The number `k` is at least the length of the list: The message `"Stacked on top of the shelf"` is printed.

Afterwards the program terminates.

A successful run might look as follows:

```
*Main> storeGoods (Corridor (Shelf [Barrel, NoGoods, Box]) (Shelf [Box])) Barrel
Do you want to turn left or right? (l|r) l
How high do you want to raise the fork? 1
Inserted the Barrel
```

In the following run, the goods are stacked on top of a shelf:

```
*Main> storeGoods (Corridor (Shelf [Barrel, NoGoods, Box]) (Shelf [Box])) Barrel
Do you want to turn left or right? (l|r) y
Incorrect input
Do you want to turn left or right? (l|r) r
How high do you want to raise the fork? 42
Stacked on top of the shelf
```

Hints:

You should use the function `getChar :: IO Char` to read a character input from the user. Moreover, you may assume there is a function `getInt :: IO Int` that reads an integer from the command line. To print a `String`, you should use the function `putStr :: String -> IO ()` or the function `putStrLn :: String -> IO ()`, if the output should end with a line break. You should use the function `show :: Goods -> String` to convert a `Goods` object to a `String`. To save space, you may also assume that the following additional declarations exist in your program:

```
leftRight, incorrect, howHigh, fellOver, noRoom, stacked :: String
leftRight = "Do you want to turn left or right? (l|r) "
incorrect = "Incorrect input"
howHigh = "How high do you want to raise the fork? "
fellOver = "Your forklift fell over"
noRoom = "No room in the shelf"
stacked = "Stacked on top of the shelf"
```

Solution:

a)
```
buildWarehouse :: [a] -> Int -> Warehouse a
buildWarehouse [] _ = Shelf []
buildWarehouse xs max | length xs <= max = Shelf xs
                      | otherwise = Corridor
                        (Shelf (take max xs))
                        (buildWarehouse (drop max xs) max)
```

b)
```
mapWarehouse :: (a -> b) -> Warehouse a -> Warehouse b
mapWarehouse f (Shelf xs) = Shelf (map f xs)
mapWarehouse f (Corridor left right) =
  Corridor (mapWarehouse f left) (mapWarehouse f right)
```

c)
```
inventory :: Warehouse Goods -> (Int, Int)
inventory w =
  fold (\(x, y) (x', y') -> (x+x', y+y')) (0, 0) (mapWarehouse invHelper w)
    where invHelper Box = (1, 0)
          invHelper Barrel = (0, 1)
          invHelper _ = (0, 0)
```

d)
```
storeGoods :: Warehouse Goods -> Goods -> IO ()
storeGoods (Corridor left right) g = do
  putStr leftRight
  input <- getChar
  putStrLn ""
  case input of
    'l' -> storeGoods left g
    'r' -> storeGoods right g
    _ -> do
      putStrLn incorrect
      storeGoods (Corridor left right) g
storeGoods (Shelf xs) g = do
  putStr howHigh
  input <- getInt
  case () of
    _ | input < 0 -> putStrLn fellOver
      | input < (length xs) -> if (xs !! input) == NoGoods
                               then putStrLn ("Inserted the "++(show g))
                               else putStrLn noRoom
      | otherwise -> putStrLn stacked
```

Exercise 2 (Semantics): (17 + 16 + 9 = 42 points)

a) i) Let D_1, D_2, and D_3 be domains with complete orders $\sqsubseteq_1$, $\sqsubseteq_2$, and $\sqsubseteq_3$, respectively. Let $g : D_1 \to D_2$ and $f : D_2 \to D_3$ be strict and monotonic functions. Here, we say that g is *strict* on a (possibly non-flat) domain D_1 if and only if $g(\bot_{D_1}) = \bot_{D_2}$. As usual, $f \circ g$ is the composition of f and g (i.e., $(f \circ g)(x) = f(g(x))$). Prove or disprove whether $f \circ g$ is:

1) strict

2) monotonic

ii) Let $\Sigma = \{a, \ldots, z\}$, let Σ^* be the set of all finite words over the alphabet Σ, and let $\leq_{lex}$ denote the usual lexicographic order on Σ^* (which corresponds to the order of words in a lexicon). So, for example $abc \leq_{lex} abcd$ and $xyz \leq_{lex} xzz$.

Prove or disprove each of the following statements:

1) There is an infinite chain in $(\Sigma^*, \leq_{lex})$.

2) The order $\leq_{lex}$ is complete on Σ^*.

3) The order $\leq_{lex}$ is confluent.

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

```
f :: (Int, Int) -> Int
f (x, 0) = 0
f (x, y) = (x * y) + f (x, y - 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 for `ff`. You may use full Haskell for `ff`.

ii) Let $\phi_{\texttt{ff}}$ be the semantics of the function `ff`. Give the definition of $\phi^n_{\texttt{ff}}(\bot)$ in closed form for any $n \in \mathbb{N}$, i.e., give a non-recursive definition of the function that results from applying $\phi_{\texttt{ff}}$ n-times to $\bot$. Here, you should assume that `Int` can represent all integers, so no overflow can occur.

iii) Give the definition of the least fixpoint of $\phi_{\texttt{ff}}$ in closed form.

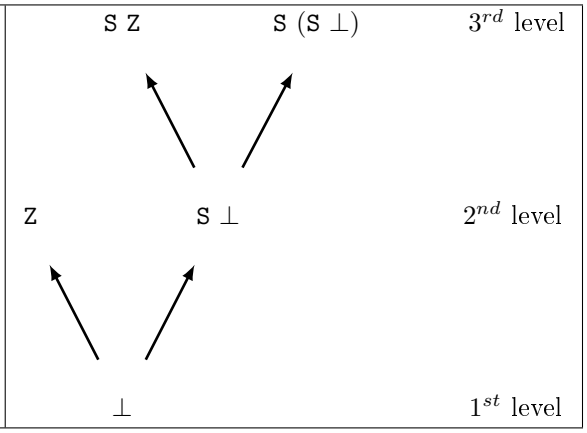
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for `Nats` on the right:

```
data Nats = Z | S Nats

-------------------------------------

data Warehouse a
    = Corridor
        (Warehouse a)
        (Warehouse a)
    | Shelf [a]

data Goods
    = NoGoods | Box | Barrel
```



Give a graphical representation of the first three levels of the domain for the type `Warehouse Goods`. You may abbreviate the constructors to the first two letters (i.e., you may write `Co` instead of `Corridor`).

Solution:

a) i) 1)

$$\begin{aligned}(f \circ g)(\bot_{D_1}) &= f(g(\bot_{D_1})) && \text{Def. of } \circ \\ &= f(\bot_{D_2}) && g \text{ is strict} \\ &= \bot_{D_3} && f \text{ is strict}\end{aligned}$$

Thus $f \circ g$ is strict.

2) Let $d_1, d_1' \in D_1$ such that $d_1 \sqsubseteq_{D_1} d_1'$. Because g is monotonic, we have that $g(d_1) \sqsubseteq_{D_2} g(d_1')$. Because f is monotonic, we also have $f(g(d_1)) \sqsubseteq_{D_3} f(g(d_1'))$. Therefore, $(f \circ g)(d_1) \sqsubseteq_{D_3} (f \circ g)(d_1')$. So, $f \circ g$ is monotonic.

ii) 1) $C = \{z, zz, zzz, \dots\}$ is an infinite chain.

2) The relation $\leq_{lex}$ is *not* a cpo. A relation $\leq_{lex}$ is a cpo iff Σ^* has a least element w.r.t. $\leq_{lex}$ and every $\leq_{lex}$-chain has a least upper bound in Σ^*. Obviously, the least element is the empty word ϵ. Consider $C = \{z, zz, zzz, \dots\}$. For every word $w \in \Sigma^*$ of length $n \in \mathbb{N}$ we have $w \leq_{lex} z^{n+1}$, $w \neq z^{n+1}$ and $z^{n+1} \in \Sigma^*$. Because Σ^* contains only finite words, this implies that the chain C has no least upper bound in Σ^*.

3) The relation $\leq_{lex}$ is confluent. If $u \leq_{lex} v$, $u \leq_{lex} w$ and n is the maximum of the lengths of v and w, then we have $v \leq_{lex} z^n$ and $w \leq_{lex} z^n$. This suffices since $\leq_{lex}$ is transitive and thus $\leq^*_{lex} = \leq_{lex}$.

b) i)

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

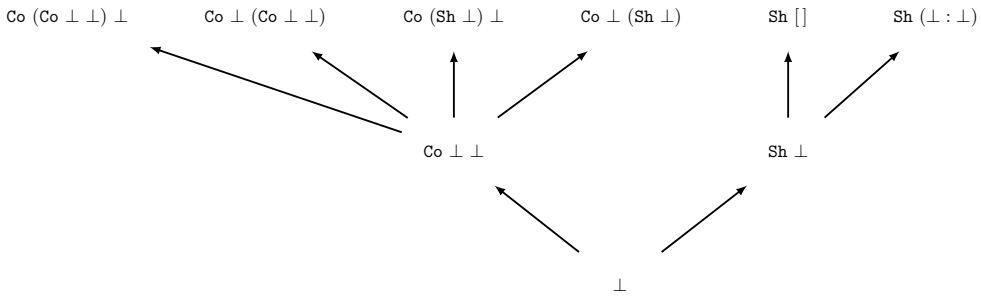
ii)

$$(\phi^n_{\texttt{ff}}(\bot))(x,y) = \begin{cases} 0 & \text{if } y = 0 \wedge 0 < n \\ \frac{x \cdot y \cdot (y+1)}{2} & \text{if } 0 < y < n \wedge x \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

iii)

$$(\mathsf{lfp}\ \phi_{\texttt{ff}})(x,y) = \begin{cases} 0 & \text{if } y = 0 \\ \frac{x \cdot y \cdot (y+1)}{2} & \text{if } 0 < y \wedge x \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

c)



Exercise 3 (Lambda Calculus): (5 + 14 + 6 = 25 points)

a) Consider the following variant of the function from Exercise 2b:

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

Please implement this function in the lambda calculus, i.e., give a lambda term q such that, for all $\texttt{x}, \texttt{y}, \texttt{z} \in \mathbb{Z}$, `f' x y == z` if and only if $q\,\texttt{x}\,\texttt{y}$ can be reduced to `z` via WHNO-reduction with the $\rightarrow_{\beta\delta}$-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

$$t = \lambda g\ x\ y.\, \texttt{if}\ (x == 1)\ \texttt{Nil}\ (\texttt{Cons}\ y\ (g\ (x-1)\ (y*x)))$$

and

$$\begin{aligned} \delta = \ & \{\ \texttt{if True} \rightarrow \lambda x\ y.\, x, \\ & \quad \texttt{if False} \rightarrow \lambda x\ y.\, y, \\ & \quad \texttt{fix} \rightarrow \lambda f.\, f(\texttt{fix}\ f)\} \\ \cup\ & \{\ x - y \rightarrow z \mid x,y,z \in \mathbb{Z} \wedge z = x - y\} \\ \cup\ & \{\ x + y \rightarrow z \mid x,y,z \in \mathbb{Z} \wedge z = x + y\} \\ \cup\ & \{\ x * y \rightarrow z \mid x,y,z \in \mathbb{Z} \wedge z = x \cdot y\} \\ \cup\ & \{\ x == x \rightarrow \texttt{True} \mid x \in \mathbb{Z}\} \\ \cup\ & \{\ x == y \rightarrow \texttt{False} \mid x,y \in \mathbb{Z}, x \neq y\} \end{aligned}$$

Please reduce `fix` t 3 1 by WHNO-reduction with the $\rightarrow_{\beta\delta}$-relation. List **all** intermediate steps until reaching weak head normal form, but please write "t" instead of

$$\lambda g\ x\ y.\texttt{if}\ (x == 1)\ \texttt{Nil}\ (\texttt{Cons}\ y\ (g\ (x-1)\ (y*x)))$$

whenever possible.

c) Consider the representation of natural numbers in the pure lambda calculus presented in the lecture, i.e., $n \in \mathbb{N}$ is represented by the term $\lambda f\ x.\ f^n\ x$. Give a pure lambda term for multiplication, i.e., a term `mult` such that $\texttt{mult}(\lambda f\ x.\ f^n\ x)(\lambda f\ x.\ f^m\ x)$ can be reduced to $\lambda f\ x.\ f^{n\cdot m}\ x$.

Explain your solution shortly. You may give a reduction sequence as explanation.

Solution:

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

b)

$$\begin{aligned}
& \texttt{fix}\ t\ 3\ 1 \\
\rightarrow_\delta\ & (\lambda f.(f\ (\texttt{fix}\ f)))\ t\ 3\ 1 \\
\rightarrow_\beta\ & t\ (\texttt{fix}\ t)\ 3\ 1 \\
\rightarrow_\beta\ & (\lambda x\ y.\texttt{if}\ (x == 1)\ \texttt{Nil}\ (\texttt{Cons}\ y\ ((\texttt{fix}\ t)\ (x-1)\ (y*x))))\ 3\ 1 \\
\rightarrow_\beta\ & (\lambda y.\texttt{if}\ (3 == 1)\ \texttt{Nil}\ (\texttt{Cons}\ y\ ((\texttt{fix}\ t)\ (3-1)\ (y*3))))\ 1 \\
\rightarrow_\beta\ & \texttt{if}\ (3 == 1)\ \texttt{Nil}\ (\texttt{Cons}\ 1\ ((\texttt{fix}\ t)\ (3-1)\ (1*3))) \\
\rightarrow_\delta\ & \texttt{if False Nil}\ (\texttt{Cons}\ 1\ ((\texttt{fix}\ t)\ (3-1)\ (1*3))) \\
\rightarrow_\delta\ & (\lambda x\ y.y)\ \texttt{Nil}\ (\texttt{Cons}\ 1\ ((\texttt{fix}\ t)\ (3-1)\ (1*3))) \\
\rightarrow_\beta\ & (\lambda y.y)\ (\texttt{Cons}\ 1\ ((\texttt{fix}\ t)\ (3-1)\ (1*3))) \\
\rightarrow_\beta\ & \texttt{Cons}\ 1\ ((\texttt{fix}\ t)\ (3-1)\ (1*3))
\end{aligned}$$

c)

$$\texttt{mult} = \lambda n\ m\ f.n\ (m\ f) \text{ or}$$

$$\texttt{mult} = \lambda n\ m\ f.m\ (n\ f)$$

The representation of m applies the function f m times to some x: $(m\ f)$. This is applied n times to some x by supplying it as the function to the representation of n. This results in $n \cdot m$ applications of f to some x, which represents $n \cdot m$.

$$
\begin{aligned}
&\texttt{mult}(\lambda f\ x.\, f^n\ x)(\lambda f\ x.\, f^m\ x)\\
\rightarrow_\beta &(\lambda m\ f.\, (\lambda f\ x.\, f^n\ x)\ (m\ f))(\lambda f\ x.\, f^m\ x)\\
\rightarrow_\beta &\lambda f.\, (\lambda f\ x.\, f^n\ x)\ ((\lambda f\ x.\, f^m\ x)\ f)\\
\rightarrow_\beta &\lambda f.\, (\lambda f\ x.\, f^n\ x)\ (\lambda x.\, f^m\ x)\\
\rightarrow_\beta &\lambda f.\, (\lambda x.\, (\lambda x.\, f^m\ x)^n\ x)\\
= &\lambda f.\, \Big(\lambda x.\, \big((\lambda x.\, f^m\ x)\big(\ldots((\lambda x.\, f^m\ x)\ x)\big)\big)\Big)\\
\rightarrow_\beta^* &\lambda f.\, (\lambda x.\, f^{m\cdot n}\ x)\\
= &\lambda f\ x.\, f^{m\cdot n}\ x
\end{aligned}
$$

Exercise 4 (Type Inference): (18 points)

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

Solution:

$\mathcal{W}(A_0, \lambda x.h\,(x\,h))$
- $\mathcal{W}(A_0 + \{x :: b_1\}, h\,(x\,h))$
  - $\mathcal{W}(A_0 + \{x :: b_1\}, h) = (id, b_2)$
  - $\mathcal{W}(A_0 + \{x :: b_1\}, x\,h)$
    - $\mathcal{W}(A_0 + \{x :: b_1\}, x) = (id, b_1)$
    - $\mathcal{W}(A_0 + \{x :: b_1\}, h) = (id, b_3)$
  - $mgu(b_1, b_3 \rightarrow b_4) = [b_1/(b_3 \rightarrow b_4)]$
  - $= ([b_1/(b_3 \rightarrow b_4)], b_4)$
- $mgu(b_2, b_4 \rightarrow b_5) = [b_2/(b_4 \rightarrow b_5)]$
- $= ([b_1/(b_3 \rightarrow b_4), b_2/(b_4 \rightarrow b_5)], b_5)$

$= ([b_1/(b_3 \rightarrow b_4), b_2/(b_4 \rightarrow b_5)], (b_3 \rightarrow b_4) \rightarrow b_5)$