Exercise 1 (Programming in Haskell): (7 + 7 + 5 + 18 = 37 points)

We define a polymorphic data structure `Maze` to represent a maze (i.e., a labyrinth) that can contain treasures and traps. The data structure `Discovery` is used to represent traps and treasures.

```
data Maze a
  = Intersection (Maze a) (Maze a)
  | Corridor a (Maze a)
  | DeadEnd a
  | Exit
```

```
data Discovery
  = Trap String
  | Treasure
```

For example, `aMaze` is a valid expression of type `Maze Discovery`.

```
aMaze = Intersection (Corridor (Trap "a trap door") (DeadEnd Treasure))
                     (Intersection (DeadEnd Treasure) Exit)
```

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 `buildMaze :: Int -> Maze Int` that gets a random seed as input.

The call `(buildMaze seed)` for an integer `seed` should create a maze according to the following rules:

- with probability $\frac{1}{5}$ the function returns `Exit`
- with probability $\frac{1}{5}$ the function returns a `DeadEnd` with a random number as argument
- with probability $\frac{1}{5}$ the function returns a `Corridor` with a random number and a random submaze
- with probability $\frac{2}{5}$ the function returns an `Intersection` with two random submazes, which differ with high probability (i.e., they are constructed from different seeds)

Hints:

- You are given a function `rand :: Int -> Int` which creates a new uniformly distributed (pseudo) random number from an input number, the seed. Already a small difference in the seed creates very different results (i.e., the sequence produced by consecutive calls `(rand seed)` and `(rand (seed+1))` is reasonably random). Also using a previous random number as new seed produces reasonably random sequences.

  For each decision during the generation, for each number inserted into the maze, and for each submaze a new random number should be used. So for any integer `seed` never use `(rand seed)` twice in a computation.
- To get a result with a certain probability, you should generate a (pseudo) random number and perform an appropriate case analysis depending on the value of this random number.

b) Implement a function `mapMaze` which gets a function and a maze as input. It should return a maze, where the function is applied to each element stored in the maze. For example, if it gets a function `even :: Int -> Bool` and a maze of type `Maze Int`, it should return a maze where we have `True` at each position where there was an even number in the input maze.

Also give a reasonable type declaration for the function!

c) Implement a function `populateMaze :: Maze Int -> Maze Discovery` which takes a maze as returned by `buildMaze` from part a) and returns a maze filled with different `Discoveries` which depend on the numbers that were in the maze before. The function should replace even numbers with `(Trap "a trap door")` and odd numbers by a `Treasure`.

Hints:

You may use `mapMaze` from the previous task (even if you have not solved the previous task).

d) In this part of the exercise, you should implement a small game where you control an adventurer who explores a `Maze Discovery`.

1) Implement a function `react :: Discovery -> Int -> IO Int` that handles the reaction to a `Discovery`. It gets a `Discovery` and the number of already collected treasures as input. For traps it should output a message indicating that a trap was reached including the `String` of the trap and that all treasures were lost. So the result in this case is 0, wrapped in the IO monad. For treasures, it should print that a treasure was found and the new number of treasures collected. The result in this case is the input number incremented by one, wrapped in the IO monad.

   Hints:

   - 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.

2) Implement a function `exploreMaze :: Maze Discovery -> [Maze Discovery] -> Int -> IO ()`. This function is the main loop of the game. The first input parameter is the maze in front of the adventurer, the second parameter is the current path of `Intersections` back to the starting point, and the third parameter is the number of already collected treasures.
   At an `Intersection` the user gets asked for input, the possibilities are `l` for left, `r` for right, or `b` for back. Depending on the input, the exploration should continue at the first (left) or second (right) argument of the `Intersection`, or for input `b` at the first `Intersection` in the list of `exploreMaze`'s second input parameter.
   At a `Corridor` or `DeadEnd`, first `react` should be called and after that, the exploration continues either at the following maze or, in case of a `DeadEnd`, at the first `Intersection` in the list of `exploreMaze`'s second input parameter.
   At an `Exit`, the user gets a message which indicates that the adventurer escaped the maze and prints the number of treasures collected.
   During the main loop you should always update the current path back to the starting point if you move out of an `Intersection` and the function `react` from 1) can help you to update the current number of treasures.
   We assume that one can also escape the maze via the entrance. So, whenever the adventurer should go back, either because of a `DeadEnd` or the user input `b`, but the list in the second input parameter is empty, the function should behave as if an `Exit` was encountered.

A run might look as follows:

```
*Main> exploreMaze aMaze [] 0
Go left, right, or back? (l|r|b) l
You reached a trap door and lost your treasures!
You found another treasure! You now have 1 treasures.
Go left, right, or back? (l|r|b) l
You reached a trap door and lost your treasures!
You found another treasure! You now have 1 treasures.
Go left, right, or back? (l|r|b) r
Go left, right, or back? (l|r|b) l
You found another treasure! You now have 2 treasures.
Go left, right, or back? (l|r|b) b
Go left, right, or back? (l|r|b) r
Go left, right, or back? (l|r|b) r
You escaped the maze with 2 treasures.
```

Hints:

- You should use the function `getChar :: IO Char` to read a character input from the user.
- You do not have to handle wrong user input correctly, i.e., you may assume that the user will only supply valid input.
- You do not have to pay attention to output formating (spaces, line breaks).
- You do not have do modify the maze, e.g, if treasures are found they are not removed but may be collected multiple times.

Solution:

a)
```
buildMaze :: Int -> Maze Int
buildMaze seed | m5 == 0 = Exit
               | m5 == 1 = DeadEnd (rand (seed+1))
               | m5 == 2  = Corridor (rand (seed+1)) (buildMaze (seed+2))
               | otherwise = Intersection (buildMaze (seed+1)) (buildMaze (seed+2))
               where m5 = (rand seed) `mod` 5
```

b)
```
mapMaze :: (a -> b) -> Maze a -> Maze b
mapMaze f (Intersection left right) = Intersection (mapMaze f left) (mapMaze f right)
mapMaze f (Corridor a maze) = Corridor (f a) (mapMaze f maze)
mapMaze f (DeadEnd a) = DeadEnd (f a)
mapMaze _ _ = Exit
```

c)
```
populateMaze :: Maze Int -> Maze Discovery
populateMaze maze = mapMaze populate maze
                    where populate x | even x = Trap "a trap door"
                                     | otherwise = Treasure
```

d) 1)
```
react :: Discovery -> Int -> IO Int
react (Trap s) t = do
  putStrLn ("You reached "++s++" and lost your treasures!")
  return 0
react Treasure t = do
  putStrLn ("You found another treasure! You now have "++(show t)++" treasures.")
  return (t+1)
```

2)
```
exploreMaze :: Maze Discovery -> [Maze Discovery] -> Int -> IO ()
exploreMaze (Intersection left right) ms t = do
  putStr "Go left, right, or back? (l|r|b) "
  input <- getChar
  putStrLn ""
  case input of
    'l' -> exploreMaze left ((Intersection left right):ms) t
    'r' -> exploreMaze right ((Intersection left right):ms) t
    'b' -> exploreMaze (head' ms) (tail ms) t
           where head' [] = Exit
                 head' (x:xs) = x
exploreMaze (Corridor a m) ms t = do
  new_t <- react a t
  exploreMaze m ms new_t
exploreMaze (DeadEnd a) ms t = do
  new_t <- react a t
  exploreMaze (head' ms) (tail ms) new_t
    where head' [] = Exit
          head' (x:xs) = x
exploreMaze Exit _ t = do
  putStrLn ("You escaped the maze with "++(show t)++" treasures.")
```

Exercise 2 (Semantics): (17 + 12 + 9 = 38 points)

a) i) Prove or disprove continuity for each of the functions $f, g : (\mathbb{Z}_\perp \to \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$

$$f(h) = \begin{cases} 0 & \text{if } h(0) = 0 \\ \perp & \text{otherwise} \end{cases}$$

$$g(h) = \begin{cases} 0 & \text{if } h(x) = 0 \text{ holds for all } x \in \mathbb{Z}_\perp \\ \perp & \text{otherwise} \end{cases}$$

If you want to make use of the fact that computable functions are continuous, give an implementation of the function and an argument for the correctness of the implementation.

ii) Let L denote the set of all Haskell lists of type `[Int]`. For example $[1, 3, 5, 5, 2]$, $[1, 2, 3, 2, 1]$, and the infinite list $[3, 6, 9, 12, \dots]$ are contained in L.

Let $\ell \in L$. We define $s : L \to \mathbb{N} \cup \{\infty\}$ where $s(\ell)$ is the length of the longest prefix of ℓ that is sorted in ascending order. For example $s([1, 3, 5, 5, 2]) = 3$, $s([4, 3, 1]) = 1$, and $s([3, 6, 9, 12, \dots]) = \infty$

Let $\leq_{sort} \;\subset\; L \times L$ be the partial order defined as $\ell_1 \leq_{sort} \ell_2$ if and only if $s(\ell_1) < s(\ell_2)$ or $\ell_1 = \ell_2$. Here, we have $n < \infty$ for every $n \in \mathbb{N}$, but $\infty \not< \infty$.

Prove or disprove each of the following statements:

1) There is an infinite chain in $(L, \leq_{sort})$.
2) The order $\leq_{sort}$ is complete on L.
3) The order $\leq_{sort}$ is confluent.

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

```
f :: (Int, Int) -> Int
f (x, 0) = x
f (x, y) = 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 $\phi_{\texttt{ff}}^n(\perp)$ 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 $\perp$. 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.

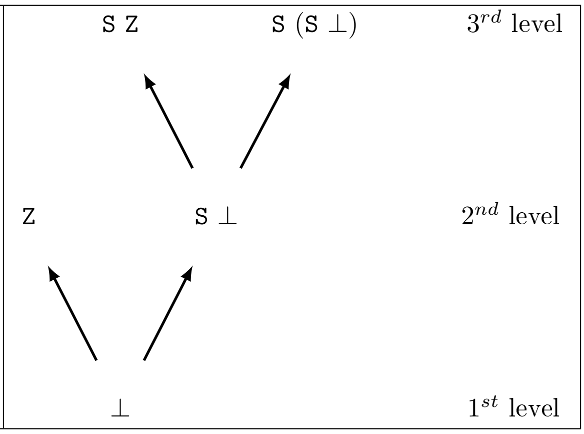
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for `Nats` on the right:

```
data Nats = Z | S Nats

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

data Maze a
    = C a (Maze a)
    | D a
    | E

data Discovery
    = Tp String | Ts
```



Give a graphical representation of the first three levels of the domain for the type `Maze Discovery`.

Solution:

a) i) The function f is continuous. Let $C = \{h_i \mid i \in \mathbb{N}\} \subset \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ be a chain. Either there is an i such that $h_i(0) = 0$, then also for all $j > i, h_j(0) = 0$ and thus $(\sqcup C)(0) = 0$. Hence, $f(\sqcup C) = 0$ and $\sqcup f(C) = \sqcup\{\perp, 0\} = 0$. Otherwise for all i we have $h_i(0) \neq 0$ and thus $(\sqcup C)(0) \neq 0$. Hence, $f(\sqcup C) = \perp$ and $\sqcup f(C) = \sqcup\{\perp\} = \perp$.

The function g is not continuous. Let $C' = \{h'_i \mid i \in \mathbb{N}\}$ where $h'_i(x) = 0$ iff $x < i$ or $x = \perp$, else $h'_i(x) = \perp$. For all $h_i \in C', g(h_i) = \perp$ but $\sqcup C'$ is the constant function 0, so $g(\sqcup C') = 0 \neq \sqcup g(C') = \sqcup\{\perp\} = \perp$.

ii) 1) $C = \{[1, 2, 1, 1, \dots], [1, 2, 3, 1, 1, \dots], [1, 2, 3, 4, 1, 1, \dots], \dots\}$ is an infinite chain.

2) The relation $\leq_{sort}$ is *not* a cpo. A relation $\leq_{sort}$ is a cpo iff L has a least element w.r.t. $\leq_{sort}$ and every $\leq_{sort}$-chain has a least upper bound in L. Obviously, the least element is the empty list [ ]. Consider C, defined as above. Obviously every sorted, infinite list is an upper bound. But as sorted, infinite lists are incomparable w.r.t. $\leq_{sort}$, there is no *least* upper bound.

3) The relation $\leq_{sort}$ is not confluent. We have $[\,] \leq_{sort} [1, 2, 3, \dots]$ and $[\,] \leq_{sort} [2, 4, 6, \dots]$, but all infinite, sorted lists are incomparable. Thus, there is no q such that $[1, 2, 3, \dots] \leq_{sort} q$ and $[2, 4, 6, \dots] \leq_{sort} q$.

b) i)

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

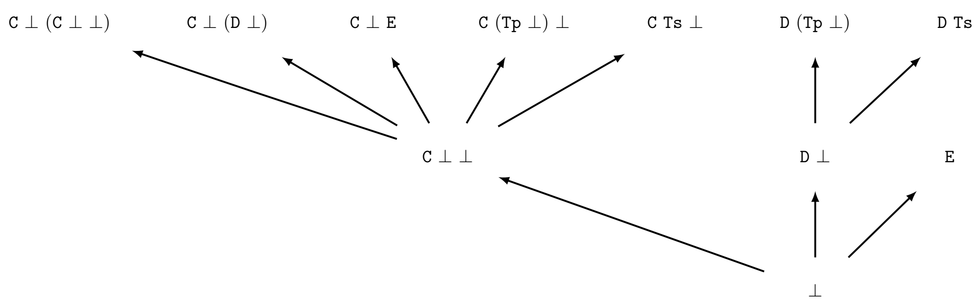
ii)

$$(\phi^n_{\texttt{ff}}(\perp))(x, y) = \begin{cases} y! \cdot x & \text{if } 0 \leq y < n \wedge x \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

iii)

$$(\mathsf{lfp}\ \phi_{\texttt{ff}})(x, y) = \begin{cases} y! \cdot x & \text{if } 0 < y \wedge x \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

c)



Exercise 3 (Lambda Calculus): (9 + 10 + 6 = 25 points)

a) Consider the following Haskell function:

```
len :: List a -> Int
len Nil = 0
len (Cons x xs) = 1 + len xs
```

Here `Cons` and `Nil` are the list constructors as defined in the lecture.

Please give an equivalent function in simple Haskell. Here, you can of course use predefined functions like `isa_Nil`, `argof_Cons`, and `sel_n_k`. Additionally, implement the function in the lambda calculus, i.e., give a lambda term q such that, for all lists `list` and `y` $\in \mathbb{Z}$, `y` is the length of `list` if and only if `len list` can be reduced to `y` 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 $(+)$.

You do not have to use the transformation algorithms presented in the lecture. It is sufficient to just give an equivalent simple program and an equivalent lambda term.

b) Let

$$t = \lambda\, g\ x\ y.\, \texttt{if}\ (x * y == 0)\ y\ (g\ x\ (y + x))$$

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 0 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

$$\lambda\, g\ x\ y.\, \texttt{if}\ (x * y == 0)\ y\ (g\ x\ (y + x))$$

whenever possible.

c) Consider $\lambda x.x\, a\, b$ as a representation of pairs of values (a, b) in pure lambda calculus.

Give a definition for a pure lambda term $\overline{\texttt{apply}}$ which applies a given term f to both elements of a pair, i.e., $\overline{\texttt{apply}}\ f\ (\lambda x.x\, a\, b) \rightarrow_\beta^* \lambda x.x\,(f\, a), (f\, b))$ should hold. You may use the shorthand notations $\overline{\texttt{True}} = \lambda x\, y.x$ and $\overline{\texttt{False}} = \lambda x\, y.y$ in your solution.

Explain your solution shortly!

Solution:

a)

```
len = \xs -> if (isa_Nil xs) then 0 else
                (if (isa_Cons xs) then (1 + sel_2_2 (argof_Cons xs) else bot)
```

$\texttt{fix}\ (\lambda\, f\ \texttt{xs}.\, \texttt{if}\ (\texttt{isa}_{\texttt{Nil}}\ \texttt{xs})\ 0\ (\texttt{if}\ (\texttt{isa}_{\texttt{Cons}}\ xs)\ (1 + f\ (\texttt{sel}_{2,2}\,(\texttt{argof}_{\texttt{Cons}}\ \texttt{xs})))\ (\texttt{bot}))$

Alternatively:

```
len = \xs -> if (isa_Nil xs) then 0 else
                (1 + sel_2_2 (argof_Cons xs))
```

$\texttt{fix}\ (\lambda\, f\ \texttt{xs}.\,\texttt{if}\ (\texttt{isa}_{\texttt{Nil}}\ \texttt{xs})\ 0\ (1 + f\ (\texttt{sel}_{2,2}\,(\texttt{argof}_{\texttt{Cons}}\ \texttt{xs}))))$

b)

$$
\begin{aligned}
& \texttt{fix}\ t\ 3\ 0 \\
\rightarrow_\delta\ & (\lambda\, f.\,(f\ (\texttt{fix}\ f)))\ t\ 3\ 0 \\
\rightarrow_\beta\ & t\ (\texttt{fix}\ t)\ 3\ 0 \\
\rightarrow_\beta\ & (\lambda\, x\ y.\,\texttt{if}\ (x * y == 0)\ y\ ((\texttt{fix}\ t)\ x\ (y + x)))\ 3\ 0 \\
\rightarrow_\beta\ & (\lambda\, y.\,\texttt{if}\ (3 * y == 0)\ y\ ((\texttt{fix}\ t)\ 3\ (y + 3)))\ 0 \\
\rightarrow_\beta\ & \texttt{if}\ (3 * 0 == 0)\ 0\ ((\texttt{fix}\ t)\ 3\ (0 + 3)) \\
\rightarrow_\delta\ & \texttt{if}\ (0 == 0)\ 0\ ((\texttt{fix}\ t)\ 3\ (0 + 3)) \\
\rightarrow_\delta\ & \texttt{if True}\ 0\ ((\texttt{fix}\ t)\ 3\ (0 + 3)) \\
\rightarrow_\delta\ & (\lambda\, x\ y.\,x)\ 0\ ((\texttt{fix}\ t)\ 3\ (0 + 3)) \\
\rightarrow_\beta\ & (\lambda\, y.\,0)\ ((\texttt{fix}\ t)\ 3\ (0 + 3)) \\
\rightarrow_\beta\ & 0
\end{aligned}
$$

c)

$$\overline{\texttt{apply}} = \lambda\, f\ p.\,\lambda\, x.\,x\ (f\ (p\ \overline{\texttt{True}}))\ (f\ (p\ \overline{\texttt{False}}))$$

Using $\overline{\texttt{True}}$ and $\overline{\texttt{False}}$ we can select the first and second element of the pair respectively. Inside the template for a new pair the function f is applied to each element of the pair individually.

Exercise 4 (Type Inference): (20 points)

Using the initial type assumption $A_0 := \{f :: \forall a.(a \to \mathit{List}\,a)\}$, infer the type of the expression $f\ (\lambda x.f\,x)$ using the algorithm $\mathcal{W}$.
Indicate the computed most general type or explain the problem the algorithm encounters if the expression is not well typed.

Solution:

$$
\begin{aligned}
& \mathcal{W}(A_0, f\ (\lambda x.f\,x)) \\
& \quad \mathcal{W}(A_0, f) = (id, b_1 \to \mathit{List}\,b_1) \\
& \quad \mathcal{W}(A_0, \lambda x.f\,x) \\
& \quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, f\,x) \\
& \quad\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, f) = (id, b_3 \to \mathit{List}\,b_3) \\
& \quad\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, x) = (id, b_2) \\
& \quad\quad \mathit{mgu}(b_3 \to \mathit{List}\,b_3, b_2 \to b_4) = [b_2/b_3, b_4/\mathit{List}\,b_3] \\
& \quad\quad = ([b_2/b_3, b_4/\mathit{List}\,b_3], \mathit{List}\,b_3) \\
& \quad = ([b_2/b_3, b_4/\mathit{List}\,b_3], b_3 \to \mathit{List}\,b_3)
\end{aligned}
$$

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