Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 11, 10 points		Deadline: Tuesday, January 16, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_11.hs` provided on the proseminar page.
- Your `*.hs` file must be compilable with `ghc`.
- Upload your solution to Exercise 1 in OLAT (`*.txt` or PDF or as part of `*.hs`)
- Upload your solution to Exercise 2 as `*.hs` file in OLAT.

Exercise 1 *Evaluation Strategies* — 5 p.

Consider the following functions.

```
-- program 1
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

filter f [] = []
filter f (x : xs)
  | f x = x : filter f xs
  | otherwise = filter f xs

smaller p xs = filter (\x -> x < p) xs
bigger p xs = filter (\x -> x >= p) xs

qsort [] = []
qsort (x:[]) = [x]
qsort (x:xs) = qsort (smaller x xs) ++ x : qsort (bigger x xs)

-- program 2
double x = x + x

take 0 _ = []
take _ [] = []
take n (x : xs) = x : take (n - 1) xs

map f [] = []
map f (x : xs) = f x : map f xs
```

1. Evaluate the expression `qsort ([2] ++ [1])` step-by-step for two evaluation strategies, cf. slide 11/8.
   - (a) call-by-value (1 point) and (b) call-by-name (1 point)
2. Evaluate the expression `take 1 (map double [3 + 5, 7 + 8])` step-by-step for three evaluation strategies:
   - (a) call-by-value (1 point), (b) call-by-name (1 point), and (c) call-by-need (1 point)

Solution 1

1. `qsort ([2] ++ [1])`

   - call-by-value

     ```
     qsort ([2] ++ [1])
     = qsort (2 : [] ++ [1])
     = qsort [2,1]
     = qsort (smaller 2 [1]) ++ 2 : qsort (bigger 2 [1])
     = qsort (filter (\x -> x < 2) [1]) ++ 2 : qsort (bigger 2 [1])
     = qsort (1 : filter (\x -> x < 2) []) ++ 2 : qsort (bigger 2 [1])
     = qsort [1] ++ 2 : qsort (bigger 2 [1])
     = [1] ++ 2 : qsort (bigger 2 [1])
     = [1] ++ 2 : qsort (filter (\x -> x >= 2) [1])
     = [1] ++ 2 : qsort (filter (\x -> x >= 2) [])
     = [1] ++ 2 : qsort []
     = [1] ++ [2]
     = 1 : [] ++ [2]
     = [1,2]
     ```

   - call-by-name

     ```
     qsort ([2] ++ [1])
     = qsort (2 : [] ++ [1])
     = qsort [2,1]
     = qsort (smaller 2 [1]) ++ 2 : qsort (bigger 2 [1])
     = qsort (filter (\x -> x < 2) [1]) ++ 2 : qsort (bigger 2 [1])
     = qsort (1 : filter (\x -> x < 2) []) ++ 2 : qsort (bigger 2 [1])
     = qsort [1] ++ 2 : qsort (bigger 2 [1])
     = [1] ++ 2 : qsort (bigger 2 [1])
     = 1 : [] ++ 2 : qsort (bigger 2 [1])
     = 1 : 2 : qsort (bigger 2 [1])
     = 1 : 2 : qsort (filter (\x -> x >= 2) [1])
     = 1 : 2 : qsort (filter (\x -> x >= 2) [])
     = 1 : 2 : qsort []
     = [1,2]
     ```

2. `take 1 (map double [3 + 5, 7 + 8])`

   - call-by-value

     ```
     take 1 (map double [3 + 5, 7 + 8])
     = take 1 (map double [8, 7 + 8])
     = take 1 (map double [8, 15])
     = take 1 (8 + 8 : map double [15])
     = take 1 (16 : map double [15])
     = take 1 (16 : 15 + 15 : map double [])
     = take 1 (16 : 30 : map double [])
     = take 1 [16,30]
     = 16 : take 0 [30]
     = 16 : []
     = [16]
     ```

   - call-by-name

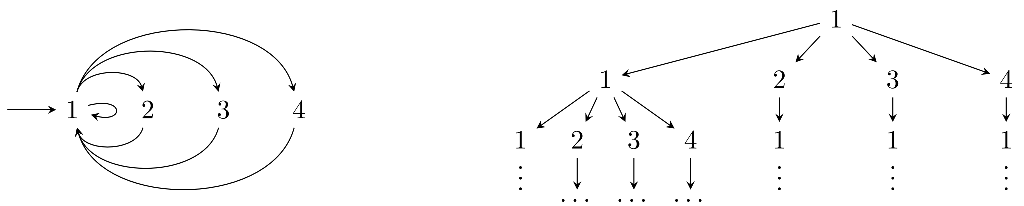
     ```
     take 1 (map double [3 + 5, 7 + 8])
     = take 1 ((double (3 + 5)) : map double [7 + 8])
     = double (3 + 5) : take 0 (map double [7 + 8])
     = (3 + 5) + (3 + 5) : take 0 (map double [7 + 8])
     = 8 + (3 + 5) : take 0 (map double [7 + 8])
     = 8 + 8 : take 0 (map double [7 + 8])
     = 16 : take 0 (map double [7 + 8])
     = 16 : []
     = [16]
     ```

- call-by-need

```
take 1 (map double [3 + 5, 7 + 8])
= take 1 ((double (3 + 5)) : map double [7 + 8])
= double (3 + 5) : take 0 (map double [7 + 8])
= (3 + 5) + (3 + 5) : take 0 (map double [7 + 8])
= 8 + 8 : take 0 (map double [7 + 8]) -- sharing of 3 + 5
= 16 : take 0 (map double [7 + 8])
= 16 : []
= [16]
```

Exercise 2 *Lazyness and Infinite Data Structures* **5 p.**

A rooted graph consists of a set of edges between nodes – of the form (source, target) – and additionally has a distinguished node called root. For instance, Figure 1a contains a rooted graph with distinguished node 1 and edges $\{(1,1),(1,2),(1,3),(1,4),(2,1),(3,1),(4,1)\}$.
One way of representing (possibly infinite) rooted graphs is to use (possibly infinite) trees, the so-called *unwinding* of a graph. For example the rooted graph of Figure 1a can be represented by the unwinding shown in Figure 1b.



(a) (b)

Figure 1: A graph and its unwinding

In this exercise graphs and (infinite) trees are represented by the following Haskell type definitions:

```haskell
type Graph a = [(a, a)]
type RootedGraph a = (a, Graph a)
data Tree a = Node a [Tree a] deriving (Eq, Show)
```

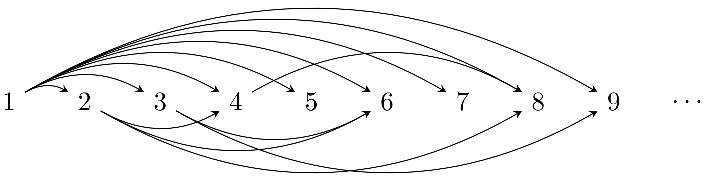
1. Implement a function `unwind :: Eq a => RootedGraph a -> Tree a` that converts a rooted graph into its tree representation. (1 point)

2. Implement a function `prune :: Int -> Tree a -> Tree a` such that `prune n t` results in a pruned tree where only the first `n` layers of the input tree are present. For example invoking `prune 2` on the infinite tree in Figure 1b drops all parts that are depicted by … and ⋮, and `prune 0` would return a tree that just contains the root node 1.

   Consider the tree that results from unwinding the rooted graph `(z, [(x,z), (z,x), (x,y), (y,x)])`, a figure of eight: $\longrightarrow$ z $\rightleftarrows$ x $\rightleftarrows$ y . What is the result of `prune 4` on this tree? (1 point)

3. Implement a function `narrow :: Int -> Tree a -> Tree a` that restricts the number of successors for each node of a tree to a given maximum (by dropping any surplus successors). For example, when calling the function `narrow 1` on the tree
$$\begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 2 & & 3 \end{tikzcd}$$
, the result would be the tree
$$\begin{tikzcd} 1 \arrow[d] \\ 2 \end{tikzcd}$$
. (1 point)

4. Define an infinite tree `mults :: Tree Integer` that represents the graph where every natural number, starting from 1 points to all its multiples: (1 point)



5. Describe the results of evaluating each of the following three expressions: `narrow 4 $ prune 2 mults`, `narrow 1 mults`, and `prune 1 mults`. (1 point)

Solution 2

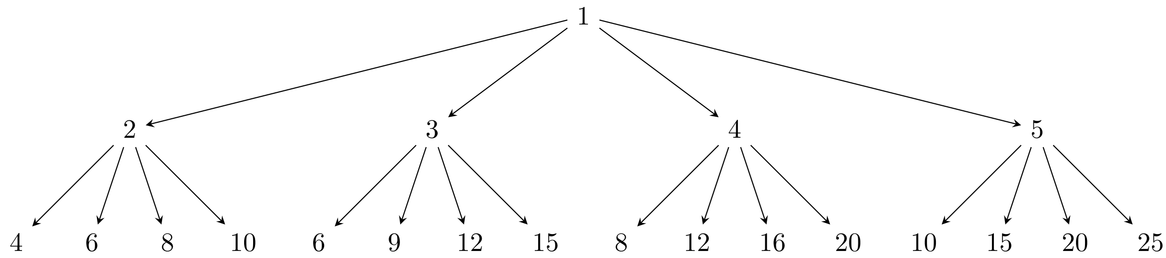
1.
```
unwind :: Eq a => RootedGraph a -> Tree a
unwind (n, g) = Node n (map (\ s -> unwind (s, g)) successors)
  where successors = map snd (filter ( (== n) . fst) g)
```

2.
```
prune :: Int -> Tree a -> Tree a
prune 0 (Node x ts) = Node x []
prune n (Node x ts) = Node x (map (prune (n - 1)) ts)
```

3.
```
narrow :: Int -> Tree a -> Tree a
narrow n (Node x ts) = Node x $ map (narrow n) $ take n ts
```

4.
```
mults :: Tree Integer
mults = go 1
  where go i = Node i $ map go $ map (i*) [2..]
```

5.
   - The expression `narrow 4 $ prune 2 mults`



   - The expression `narrow 1 mults` yields an infinite tree that structurally resembles the infinite list of powers of 2: $1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 8 \longrightarrow 16 \rightarrow 32 \rightarrow \cdots$
   - The expression `prune 1 mults` yields the following infinite tree: