EX 4 - solution sheet

Introduction

Please note that there can be other solutions than those listed in this document.

This is a literate Haskell file which is available as PDF, as well as literate Haskell source code (.lhs extension). If you are looking at the PDF version, you should download the .lhs file, which can be loaded into ghci or compiled with ghc. This document uses ***Markdown*** syntax, compiled to PDF using *Pandoc*.

Everything that is not enclosed in a code block like below is treated as text.

```
-- This is code
main :: IO ()
main = undefined
```

1 - List comprehension equivalence

$[x|x \leftarrow xs, y \leftarrow ys] \equiv [x|y \leftarrow ys, x \leftarrow xs]$ In the first list comprehension, the length of ys decides how many times each element of xs is replicated, before moving on to the next i.e. $xs = [1,2,3], ys = [1,2] \rightarrow [1,1,2,2,3,3]$.

In the second list comprehension, the length of ys decides how many times xs is cycled i.e. $xs = [1,2,3], ys = [1,2] \rightarrow [1,2,3,1,2,3]$

We have a few cases where the equation is true:

1. $|ys| = 1$ Will be the same as just using xs
2. $xs = [x,x,x,\ldots]$ If xs consist of only one value, the order they are added to the resulting list does not matter.
3. $xs = \emptyset$ Will always be empty
4. $ys = \emptyset$ Will always be empty

2 - Binary trees

Let us define the tree:

```haskell
data Tree a = Nil | Node a (Tree a) (Tree a) --deriving Show
-- A show instance that lets us view the tree in a bit more readable format
instance (Show a) => Show (Tree a) where
  show Nil = "Nil"
  show (Node x l r) = "(value=" ++ show x ++ ")\n"
        ++ "|--" ++ (drop 3 . unlines . map ("|   " ++) . lines $ show l)
        ++ "'--" ++ (drop 3 . unlines . map ("    " ++) . lines $ show r)
```

a - repeat

```haskell
repeatT :: a -> Tree a
repeatT x = Node x (repeatT x) (repeatT x)
```

b - take

```haskell
takeT :: Int -> Tree a -> Tree a
takeT n _ | n <= 0   = Nil
takeT _ Nil          = Nil
takeT n (Node x l r) = Node x (takeT (n-1) l) (takeT (n-1) r)
```

c - replicate

```haskell
replicateT :: Int -> a -> Tree a
replicateT n = takeT n . repeatT
```

3 - Directional functions

```haskell
data Direction = L | R deriving Show
```

a - elementAt

```haskell
elementAt ::[Direction] -> Tree a -> Maybe a
elementAt [] (Node x _ _)     = Just x
elementAt _  Nil              = Nothing
elementAt (L:xs) (Node _ l _) = elementAt xs l
elementAt (R:xs) (Node _ _ r) = elementAt xs r
```

b - modifyAt

```haskell
modifyAt :: (a -> a) -> [Direction] -> Tree a -> Tree a
modifyAt _ _ Nil           = Nil
modifyAt f []     (Node x l r) = Node (f x) l r
modifyAt f (L:xs) (Node x l r) = Node x (modifyAt f xs l) r
modifyAt f (R:xs) (Node x l r) = Node x l (modifyAt f xs r)
```

4 - toTree

We can use index computation as follows:

```
toTree :: [a] -> Tree a
toTree l = toTree' 0
  where
    toTree' n | n >= length l = Nil
              | otherwise = Node (l !! n) (toTree' (2*n+1)) (toTree' (2*n+2))
```

or `level` the list.

```
toTreeL :: [a] -> Tree a
toTreeL [] = Nil
toTreeL l = toTreeL' (level 0 l)
  where
    level :: Int -> [a] -> [[a]]
    level _ [] = []
    level n xs  = take (n+1) xs : level (2*n+1) (drop (n+1) xs)

    left  :: Int -> [[a]] -> [[a]]
    left _ []     = []
    left n (x:xs) = take (2^n `div` 2) x : left (n+1) xs

    right :: Int -> [[a]] -> [[a]]
    right _ []     = []
    right n (x:xs) = drop (2^n `div` 2) x : right (n+1) xs

    toTreeL' :: [[a]] -> Tree a
    toTreeL' ([x]:xs) = Node
      x
      (toTreeL' (left 1 xs))
      (toTreeL' (right 1 xs))
    toTreeL' _ = Nil
```

5 - Interpreted language

```haskell
data Expression =
  Var String -- Variable
  | Val Int  -- Integer literal
  | Op Expression Bop Expression -- Operation

data Bop
  = Plus
  | Minus
  | Times
  | Divide
  | Gt | Ge | Lt | Le | Eql
  deriving (Show, Eq)

type State = String -> Maybe Int

evalE :: State -> Expression -> Either String Int
evalE st (Var s) =
  case st s of
    Nothing -> Left ("Variable not defined: " ++ show s)
    Just state -> Right state
evalE _ (Val i)            = Right i
evalE st (Op exp1 op exp2) =
  let
    res1 = evalE st exp1
    res2 = evalE st exp2
    in case (res1, res2) of
      (Right v1, Right v2) -> Right (eval op v1 v2)
      (Left e1, _)         -> Left  e1
      (_, Left e2)         -> Left  e2
  where
    eval :: Bop -> Int -> Int -> Int
    eval Plus   a b = a + b
    eval Minus  a b = a - b
    eval Times  a b = a * b
    eval Divide a b = a `div` b
    eval Gt     a b = toInt (a >  b)
    eval Ge     a b = toInt (a >= b)
    eval Lt     a b = toInt (a <  b)
    eval Le     a b = toInt (a <= b)
    eval Eql    a b = toInt (a == b)
    toInt :: Bool -> Int
    toInt v = if v then 1 else 0
```

The state function could be something like the following:

```
myState :: State
myState x = lookup x [
    ("Klaus Meer", error "Counted to infinity, twice."),
    ("BoomBox", 5*10^6)
  ]

anotherState :: State
anotherState "Bjarne Toft" = Just 42
anotherState "test"        = Just 1
anotherState _             = Nothing
```