```
--Übungsblatt08 ; Thomas Alessandro Buse ; 192959 ; Gruppe: 17
--                                Paul Rüssmann ; 196683

--Aufgabe 8.1
data Tree a = V a | F a [Tree a] deriving (Show)
data Bintree a = Empty | Fork a (Bintree a) (Bintree a)

foldTree :: (a -> val)     -- bildet einen Knotenwert (a) auf den Rueckgabewert ab
  -> (a -> valL -> val)    -- bildet den aktuellen Knotenwert (a) zusammen mit dem Aggregat von den Kindern auf einen Rueckgabewert ab
  -> valL                  -- Rueckgabewert fuer eine leere Kindliste
  -> (val -> valL -> valL) -- faltet einen Wert mit dem Aggregat zusammen zum neuen Aggregat
  -> Tree a                -- der zu faltende Baum
  -> val                   -- Rueckgabewert

foldTree f _ _ _ (V a) = f a -- einfache Bildfunktion eines Knoten wird aufgerufen
foldTree f g nil h (F a ts) = g a $ -- zusammenfassung des Aggregat mit dem akutellen Knoten
  foldTrees f g nil h ts -- Erzeugung des Aggregats

foldTrees :: (a -> val)    -- bildet einen Knotenwert (a) auf den Rueckgabewert ab
  -> (a -> valL -> val)    -- bildet den aktuellen Knotenwert (a) zusammen mit dem Aggregat von den Kindern auf einen Rueckgabewert ab
  -> valL                  -- Rueckgabewert fuer eine leere Kindliste
  -> (val -> valL -> valL) -- faltet einen Wert mit dem Aggregat zusammen zum neuen Aggregat
  -> [Tree a]              -- die zu faltenden Baume
  -> valL                  -- Aggregat

foldTrees _ _ nil _ [] = nil   -- leere liste => gibt Wert fuer leere Liste zurueck
foldTrees f g nil h (t:ts) = h -- kombiniert Ruckgabewert einer ein Tree-faltung und Aggregat einer Liste
  (foldTree f g nil h t)       -- Faltung eines Baums
  (foldTrees f g nil h ts)     -- Faltung der Restliste zum Aggregat
```

```
foldBtree :: val -> (a -> val -> val -> val) -> Bintree a -> val
foldBtree val _ Empty = val
foldBtree val f (Fork a left right) = f a (foldBtree val f left) (foldBtree val f right)


--Aufgabe 8.1 a
or_ :: Tree Bool -> Bool
or_ = foldTree (id) (||) False (||)

--Aufgabe 8.1 b
preorderB :: Bintree a -> [a]
preorderB a = foldBtree [] (treeList) a

treeList :: a -> [a] -> [a] -> [a]
treeList a b c = (++) ((++) b [a]) c

--Aufgabe 8.2 a
data PosNat = One | Succ' PosNat

foldPosNat :: (PosNat -> val) -> (val -> val) -> PosNat -> val
foldPosNat f _ One = f One
foldPosNat f g (Succ' a) = g (foldPosNat f g a)

--Aufgabe 8.2 b

toInt :: PosNat -> Int
toInt a = foldPosNat idP (+1) a

idP :: PosNat -> Int
idP One = 1
```

```
--Aufgabe 8.4
data Mod10 = Z0 | Z1 | Z2 | Z3 | Z4 | Z5 | Z6 | Z7 | Z8 | Z9 deriving (Show, Eq, Ord, Enum)

succ10 Z9 = Z0
succ10 x  = succ x

pred10 Z0 = Z9
pred10 x  = pred x

f :: Mod10 -> Mod10
f x | x < Z4        = succ10 x
    | x > Z6        = pred10 x
    | otherwise     = x


fixpt :: (a -> a -> Bool) -- Vergleichsfunktion, muessen wir noch einen Schritt machen?
  -> (a -> a) -- Schrittfunktion, naechster Wert
  -> a -- Startwert/aktueller Wert
  -> a -- Rueckgabe
fixpt le phi a = if b `le` a then a else fixpt le phi b
  where b = phi a


--Aufgabe 8.4 a

lfp = fixpt (==) f Z0
gfp = fixpt (==) f Z9

--Aufgabe 8.4 b
evens :: [Mod10]
evens = (++) [Z0] [(succ10 (succ10 a)) | a <- evens]
```