```haskell
-- setting the "warn-incomplete-patterns" flag asks GHC to warn you
-- about possible missing cases in pattern-matching definitions
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

-- see https://wiki.haskell.org/Safe_Haskell
{-# LANGUAGE Safe #-}

module Assessed2 (applyfuns , updateNodes , graft , elimImplications ,
                  isInCNF , toCNF , binToRose , roseToBin) where

import Types

---------------------------------------------------------------------------------
---------------- DO **NOT** MAKE ANY CHANGES ABOVE THIS LINE --------------------
---------------------------------------------------------------------------------


{- Exercise 1 -}
applyfuns :: (a -> c) -> (b -> d) -> Tree a b -> Tree c d
applyfuns _ g (Leaf x) = Leaf (g x)
applyfuns f g (Fork left root right) = Fork (applyfuns f g left) (f root)
(applyfuns f g right)

{- Exercise 2 -}
updateNodes :: Route -> (a -> a) -> BinTree a -> BinTree a
updateNodes _ _ Empty = Empty
updateNodes [] f (Node left root right) = (Node left (f root) right)
updateNodes (GoLeft:rs) f (Node left root right) = Node (updateNodes rs f left) (f
root) right
updateNodes (GoRight:rs) f (Node left root right) = Node left (f root) (updateNodes
rs f right)

isValidRoute :: Route -> BinTree a -> Bool
isValidRoute [] _ = True
isValidRoute (_:_) Empty = False
isValidRoute (GoLeft:drs) (Node left _ _)= isValidRoute drs left
isValidRoute (GoRight:drs) (Node _ _ right) = isValidRoute drs right

{- Exercise 3 -}

graft :: Rose -> [ Rose ] -> (Rose , [ Rose ])

graft (Br []) (child:children) = (child, children)

graft parent [] = (parent, [])
graft parent [Br []] = (parent, [])

graft (Br [rose]) [child] = (Br [fst (graft rose [child])], snd (graft rose
[child]))
graft (Br [rose]) children = (Br [fst (graft rose children)], snd (graft rose
children))

graft (Br (rose:roses)) [child] = ((Br ( [fst (graft rose [child])] ++ (roses) )),
[])

graft (Br (rose:roses)) (child:children) = (Br ((toArrayOfRoses(fst (graft rose
[child]))) ++ (toArrayOfRoses( fst (graft  (Br roses)  children)))), snd (graft (Br
roses) children))
```

```
addAllRoses :: [Rose] -> [Rose] -> [Rose]
addAllRoses roses [] = roses
addAllRoses [] _ = []
addAllRoses (rose:roses) (child:children) =  [fst (graft rose [child])] ++
addAllRoses roses children

calcChild :: [Rose] -> [Rose] -> [Rose]

calcChild _ [] = []
calcChild [] children = children
calcChild (rose:roses) (child:children) =  snd (graft rose [child]) ++ calcChild
roses children

{- Exercise 4 -}

elimImplications :: Expr -> Expr
elimImplications (Var x) = (Var x)
elimImplications (Not x) = (Not (elimImplications x))
elimImplications (Conj x y) = (Conj (elimImplications x) (elimImplications y))
elimImplications (Disj x y) = (Disj (elimImplications x) (elimImplications y))
elimImplications (Implies x y) = (Disj (Not (elimImplications x)) (elimImplications
y))


isInCNF :: Expr -> Bool
isInCNF (Conj x y) = (isConj x) && (isConj y)
isInCNF (Disj x y) = (isDisj x) && (isDisj y)
isInCNF x = (isLiteral x)

isLiteral :: Expr -> Bool
isLiteral (Var x) = True
isLiteral (Not x) = (isLiteral x)
isLiteral _ = False

isConj :: Expr -> Bool
isConj (Conj x y) = (isConj x) && (isConj y)
isConj (Disj x y) = (isDisj x) && (isDisj y)
isConj x = isLiteral x

isDisj :: Expr -> Bool
isDisj (Disj x y) = (isDisj x) && (isDisj y)
isDisj x = isLiteral x

toCNF :: Expr -> Expr

toCNF = distribute.removeMultipleNots.deMorgan.elimImplications


removeMultipleNots :: Expr -> Expr
removeMultipleNots (Not (Not x)) = (removeMultipleNots x)
removeMultipleNots (Var x) = (Var x)
removeMultipleNots (Not x) = (Not (removeMultipleNots x))
removeMultipleNots (Conj x y) = (Conj (removeMultipleNots x) (removeMultipleNots
y))
removeMultipleNots (Disj x y) = (Disj (removeMultipleNots x) (removeMultipleNots
y))
removeMultipleNots (Implies x y) = (Implies (removeMultipleNots x)
(removeMultipleNots y))
```

```
deMorgan :: Expr -> Expr
deMorgan (Var x) = (Var x)
deMorgan (Not (Disj x y)) = (Conj (Not (deMorgan x)) (Not (deMorgan y)))
deMorgan (Not (Conj x y)) = (Disj (Not (deMorgan x)) (Not (deMorgan y)))
deMorgan (Not x) = (Not (deMorgan x))
deMorgan (Conj x y) = (Conj (deMorgan x) (deMorgan y))
deMorgan (Disj x y) = (Disj (deMorgan x) (deMorgan y))
deMorgan (Implies x y) = (Implies (deMorgan x) (deMorgan y))

distribute :: Expr -> Expr
distribute (Disj x (Conj y z)) = (Conj (Disj x y) (Disj x z))
distribute (Disj (Conj x y) z) = (Conj (Disj x z) (Disj y z))
distribute (Var x) = (Var x)
distribute (Not x) = (Not (distribute x))
distribute (Conj x y) = (Conj (distribute x) (distribute y))
distribute (Disj x y) = (Disj (distribute x) (distribute y))
distribute (Implies x y) = (Implies (distribute x) (distribute y))




{- Exercise 5 -}



toArrayOfRoses :: Rose -> [Rose]
toArrayOfRoses (Br roses) = roses


binToRose :: Bin -> Rose

--Root
binToRose (Branch (Root) (Root)) = Br []

--addChild
binToRose (Branch (child) (Root)) =

   if        (checkChild child == 1) || (checkChild child == 2) then ((Br
[binToRose child]))
    else  (Br (toArrayOfRoses(binToRose child)))


--addSibling
binToRose (Branch (Root) (rightChild)) = if (((binToRose rightChild) == (Br [])) ||
((rightChild == (Branch (Branch Root (Branch Root Root)) Root) ))) then (Br ([Br
[]] ++ [binToRose rightChild]))
else (Br ([Br []] ++ toArrayOfRoses (binToRose rightChild)))

--addBoth
binToRose (Branch (leftChild) (rightChild)) = Br ([Br[binToRose leftChild]] ++
[binToRose rightChild])

binToRose Root = Br[]
```

```
checkChild :: Bin -> Int
checkChild (Branch (Root) (Root)) = 1
checkChild (Branch (_) (Root)) = 2
checkChild (Branch (Root) (_)) = 3
checkChild (Branch (leftChild) (rightChild)) = 4
checkChild _ = 5
```

```
roseToBin :: Rose -> Bin

--Root
roseToBin (Br[]) = (Branch Root Root)

--Child
roseToBin (Br[child]) = (Branch (roseToBin child) Root)

--Sibling
roseToBin (Br(child:children)) = (Branch (Branch (toBin (roseToBin(child)))
(addSiblings children)) (Root))


addSiblings :: [Rose] -> Bin
addSiblings [child] =  roseToBin child
addSiblings (child:children) = (Branch (Branch (toBin (roseToBin(child)))
(addSiblings children)) (Root))
addSiblings [] = Root

toBin :: Bin -> Bin
toBin (Branch leftChild rightChild) = leftChild
toBin Root = Root


--encoding : left children are the children of the parent
--           right children are the siblings of the parent
--           Branch Root Root is the Br[]

{-

-- eg   Bin =        x                =       x           (Br [])
                  /    \
                x       x



        Bin =      x             =   x                  [Br[Br[]]]
                /   \                |
                x    x               x
              /  \
             x    x


        Bin =      x             =    x                 [Br[Br[Br[]],Br[]]]
```

```
     /   \                /  \
     x    x              x    x
   /  \                  |
  x    x                 x
 / \  / \
x   x x x


     -}
```