```haskell
module Lomoba where
import Grafo
import List
import qualified Data.List (union)
import Tipos
-- ---------------------------------Sección 6--------- Lomoba --------------------------
-- Ejercicio 10
foldExp ::     (Prop -> a)    -- Función para aplicar a (Var p)
            -> (a -> a)       -- Función para aplicar a (Not e)
            -> (a -> a -> a)  -- Función para aplicar a (Or e1 e2)
            -> (a -> a -> a)  -- Función para aplicar a (And e1 e2)
            -> (a-> a)        -- Función para aplicar a (D e)
            -> (a -> a)       -- Función para aplicar a (B e)
            -> Exp -> a       -- Función que construye el fold
foldExp fVar fNot fOr fAnd fd fb (Var p) = fVar p
foldExp fVar fNot fOr fAnd fd fb (Not e) = fNot (foldExp fVar fNot fOr fAnd fd fb e)
foldExp fVar fNot fOr fAnd fd fb (Or a b) = fOr
                                            (foldExp fVar fNot fOr fAnd fd fb a)
                                            (foldExp fVar fNot fOr fAnd fd fb b)
foldExp fVar fNot fOr fAnd fd fb (And a b) = fAnd (foldExp fVar fNot fOr fAnd fd fb a)
                                              (foldExp fVar fNot fOr fAnd fd fb b)
foldExp fVar fNot fOr fAnd fd fb (D e) = fd (foldExp fVar fNot fOr fAnd fd fb e)
foldExp fVar fNot fOr fAnd fd fb (B e) = fb (foldExp fVar fNot fOr fAnd fd fb e)

-- Ejercicio 11
-- Calcula la visibilidad de la f ormula. Cada vez que aparece <> o [] debo
-- incrementar en 1 el valor de la misma.
--    * Var es el caso base, por lo que utilizare const 0.
--    * Not no sumara ningun valor, por lo que utilizare id.
--    * Or y And que seran bifurcaciones en el arbol de recursion y por lo tanto
--      tomaremos el maximo resultante de ambas ramas de la recursion.
--    * D y B seran los casos en donde tendre que sumar uno, por lo que aplicare
--      la funcion + con la valuacion parcial en el primero de sus parametros.
visibilidad :: Exp -> Integer
visibilidad = foldExp fVar fNot fOr fAnd fd fb
    where   fVar = const 0
            fNot = id
            fOr = max
            fAnd = max
            fd = (+1)
            fb= (+1)

-- Ejercicio 12
-- Extraer las variables proposicionales que aparecen en la formula, sin repetir.
--    * Var es el caso base, por lo que utilizare una lambda que dado p me devuelve [p].
--    * Or y And que seran bifurcaciones en el arbol de recursion y por lo tanto
--      utilizaremos la union de conjuntos.
--    * Not, D y B no adicionaran ningun simbolo, por lo que utilizare id.
extraer :: Exp -> [Prop]
extraer = foldExp fVar fNot fOr fAnd fd fb
    where   fVar = (\p -> [p])
            fNot = id
            fOr = Data.List.union
            fAnd = Data.List.union
            fd = id
            fb = id
```

```haskell

-- Ejercicio 13
eval :: Modelo -> Mundo -> Exp -> Bool
eval mod w exp = (eval' mod exp) w

-- Dado un modelo y una expresión, devuelve una función
-- que para un Mundo dado (en el modelo), devuelve la evaluación.
-- En los pasos de D y B, se aplica la función recursiva sobre todos
-- los mundos vecinos y se busca que en alguno o en todos la expresión
-- sea true.
eval' :: Modelo -> Exp -> (Mundo -> Bool)
eval' (K g mundosTrue) =
    foldExp
        (\p w -> w `elem` (mundosTrue p)) -- ::Prop -> (Mundo -> Bool)
        (\rec w -> not (rec w)) -- ::(Mundo -> Bool) -> Mundo -> Bool
        (\rec1 rec2 w -> (rec1 w) || (rec2 w))
        (\rec1 rec2 w -> (rec1 w) && (rec2 w))
        (\rec w -> or (map rec (vecinos g w))) -- rec::(Mundo -> Bool)
        (\rec w -> and (map rec (vecinos g w)))



-- Ejercicio 14
-- Dadas todas las variables proposicionales del grafo, se devuelven los mundos
-- que al evaluarlos dan verdadero
valeEn :: Exp -> Modelo -> [Mundo]
valeEn exp mod@(K g mundosTrue) = filter (eval' mod exp) (nodos g)

-- Ejercicio 15
-- Usando foldr, voy construyendo un nuevo modelo de Kripke, partiendo
-- del modelo pasado por argumento y sacándole los mundos donde no vale
-- la expresión. Por cada mundo que saco, le saco el nodo al grafo y
-- lo saco del valor de retorno de la función (para cada símbolo
-- proposicional)
quitar :: Exp -> Modelo -> Modelo
quitar e mod@(K g mundosTrue) =
            foldr (\w (K gRec fRec) ->
                            K (sacarNodo w gRec)
                              (\prop -> filter (/=w) (fRec prop))
                  )
                  mod                   -- Si e vale en todos los mundos, devuelvo el modelo original
                  (valeEn (Not e) mod) -- mundos donde NO vale e

-- Ejercicio 16
-- Compara el modelo original con el modelo resultante de quitarle los mundos tales
-- que no valga e.
cierto :: Modelo -> Exp -> Bool
cierto mod@(K g mundosTrue) e = (sort (nodos g) == sort (valeEn e mod))


```