Práctica 4

David Rivera Morales -- 320176876

Israel Rivera -- 320490747

La función asocia_der

Función que recibe una LProp y no importa el orden en que se ejecuten las operaciones no altere el resultado, siempre y cuando se mantenga intacta la secuencia de los operandos, de izquierda a la derecha sobre los elementos de la expresión

```
asocia_der :: LProp -> LProp
asocia_der (Conj (Conj x y) z) = Conj x (Conj y z)
asocia_der (Conj x (Conj y z)) = Conj x (Conj y z)
asocia_der (Disy (Disy x y) z) = Disy x (Disy y z)
asocia_der (Disy x (Disy y z)) = Disy x (Disy y z)
asocia_der (Impl (Impl x y) z) = Impl x (Impl y z)
asocia_der (Impl x (Impl y z)) = Impl x (Impl y z)
asocia_der (Syss (Syss x y) z) = Syss x (Syss y z)
asocia_der (Syss x (Syss y z)) = Syss x (Syss y z)
asocia_der x = x
```

La función asocia_izq

Función que recibe una LProp y no importa el orden en que se ejecuten las operaciones no altere el resultado, siempre y cuando se mantenga intacta la secuencia de los operandos, de derecha a la izquierda sobre los elementos de la expresión

```
asocia_izq :: LProp -> LProp
asocia_izq (Conj (Conj x y) z) = Conj (Conj x y) z
asocia_izq (Conj x (Conj y z)) = Conj (Conj x y) z
asocia_izq (Disy (Disy x y) z) = Disy (Disy x y) z
asocia_izq (Disy x (Disy y z)) = Disy (Disy x y) z
asocia_izq (Impl (Impl x y) z) = Impl (Impl x y) z
asocia_izq (Impl x (Impl y z)) = Impl (Impl x y) z
asocia_izq (Syss (Syss x y) z) = Syss (Syss x y) z
asocia_izq (Syss x (Syss y z)) = Syss (Syss x y) z
asocia_izq x = x
```

La función conm

Función que recibe una LProp, en el que el orden de los factores no altera el resultado, de forma exhaustiva sobre los elementos de la expresión cuyo operador lógico sea conjunción o disyunción.

```
conm :: LProp -> LProp
conm (Conj x y) = Conj y x
conm (Disy x y) = Disy y x
conm (Impl x y) = Impl y x
conm (Syss x y) = Syss y x
conm x = x
```

La función dist

Función que recibe una LProp que tenga conectores de conjunción y disyunción estos se reformulen estructuralmentemanteniendo el mismo resultado, de forma exhaustiva sobre toda la expresión.

```
dist :: LProp -> LProp
dist (Conj x (Disy y z)) = Disy (Conj x y) (Conj x z)
dist (Conj (Disy y z) x) = Disy (Conj y x) (Conj z x)
dist (Disy x (Conj y z)) = Conj (Disy x y) (Disy x z)
dist (Disy (Conj y z) x) = Conj (Disy y x) (Disy z x )
dist (Impl x y) = Impl x y
dist (Syss x y) = Syss x y
dist x = x
```

La función deMorgan

Función que le aplica a una LProp, teniendo en cuenta que el opuesto de una conjunción es equivalente a la disyunción que se forma con los opuestos o negaciones de las proposiciones que conforman la conjunción y la negación de la disyunción se puede expresar como una conjunción conformada por los opuestos o negaciones de las proposiciones involucradas en la disyunción.

```
deMorgan :: LProp -> LProp
deMorgan (Neg (Conj x y)) = Disy (Neg x) (Neg y)
deMorgan (Neg (Disy x y)) = Conj (Neg x) (Neg y)
deMorgan x = x
```

La función equiv_op

Función que recibe una LProp y debido a que sus valores de verdad siempre eson iguales se pueden sustituir una por otra,sin afectar esos valores de verdad

```
equiv_op :: LProp -> LProp
equiv_op (Conj x y) = Conj (equiv_op x) (equiv_op y)
equiv_op (Disy x y) = Disy (equiv_op x) (equiv_op y)
equiv_op (Impl x y) = Disy (Neg (equiv_op x)) (equiv_op y)
equiv_op (Syss x y) = Conj (Disy (Neg (equiv_op x)) (equiv_op y)) (Disy
```

```
(Neg (equiv_op y)) (equiv_op x))
equiv_op x = x
```

La función dobleNeg

Función que quita las dobles negaciones de una LProp, de forma en la cual vuelva a quedar en su forma original en cualquier caso

```
dobleNeg :: LProp -> LProp
dobleNeg (Neg (Neg x)) = dobleNeg x
dobleNeg (Conj x y) = Conj (dobleNeg x) (dobleNeg y)
dobleNeg (Disy x y) = Disy (dobleNeg x) (dobleNeg y)
dobleNeg (Impl x y) = Impl (dobleNeg x) (dobleNeg y)
dobleNeg (Syss x y) = Syss (dobleNeg x) (dobleNeg y)
dobleNeg (Neg x) = Neg (dobleNeg x)
dobleNeg x = x
```

La función num_conectivos

Función que redibe una LProp y cada que exista un conectivo lógico agrega 1 hasta terminar toda la expresión, haciendo que el contador crezca por cada conectivo encontrado, respondiendo con el número de conectivos existentes

```
num_conectivos :: LProp -> Int
num_conectivos PTrue = 0
num_conectivos PFalse = 0
num_conectivos (Var x) = 0
num_conectivos (Neg x) = 1 + num_conectivos x
num_conectivos (Conj x y) = 1 + num_conectivos x + num_conectivos y
num_conectivos (Disy x y) = 1 + num_conectivos x + num_conectivos y
num_conectivos (Impl x y) = 1 + num_conectivos x + num_conectivos y
num_conectivos (Syss x y) = 1 + num_conectivos x + num_conectivos y
```

La función interpretacion

Esta función va a tomar una LProp ψ y una asignación para regresar la interpretacion de ψ a partir de los valores de la asignación

```
interpretacion:: LProp -> Asignacion -> Int
interpretacion PTrue asig = 1
interpretacion PFalse asig = 0
interpretacion (Var a) asig = asignaValor a asig
interpretacion (Neg    expr)  vs = (interpretacion expr vs)-1
interpretacion (Conj   exp1 exp2) vs = if (interpretacion exp1 vs) == 1 &&
(interpretacion exp2 vs) == 1 then 1 else 0
```

```
interpretacion (Disy   exp1 exp2) vs = if (interpretacion exp1 vs) == 0 ||
(interpretacion exp2 vs) == 0 then 1 else 0 --check this
interpretacion (Impl   exp1 exp2) vs = if (interpretacion exp2 vs) == 1 ||
(interpretacion exp1 vs)-1 == 0 then 0 else 1
interpretacion (Syss exp1 exp2) vs = if (interpretacion exp1 vs) ==
(interpretacion exp2 vs) then 1 else 0

asignaValor:: Eq a => a -> [(a,b)] -> b
asignaValor x ((a,b):xs) = if a == x then b else asignaValor x xs
```