Estructuras Discretas 2023-1

Práctica 4: Lógica proposicional.

Lourdes del Carmen González Huesca
María Fernanda Mendoza Castillo
Juan Alfonso Garduño Solís
Alan Moreno de la Rosa

19 de septiembre de 2022
Fecha de entrega: Sábado 29 de octubre de 2022

Introducción.

Para esta práctica vamos a repasar, reforzar y abstraer temas de la lógica proposicional utilizando nuestro lenguaje de programación favorito: **Haskell**.
La lógica proposicional es un lenguaje formal del cual se puede decidir la validez o invalidez de algunos razonamientos deductivos. Este lenguaje está formado por proposiciones, conectivos lógicos, símbolos de puntuación y constantes lógicas.

Equivalencia

Dos fórmulas ψ y ϕ son lógicamente equivalentes si es que tienen siempre los mismos valores de verdad, por lo tanto se puede sustituir una por la otra sin afectar los valores de verdad. Cuando dos fórmulas ψ y ϕ son lógicamente equivalentes se denota $\psi \equiv \phi$ si y solo si $\psi \leftrightarrow \phi$ es una tautología. A continuación se enlistan algunas de las leyes de equivalencia más importantes dentro de la lógica proposicional.

Ley o Equivalencia	Nombre
$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ $(A \vee B) \vee C \equiv A \vee (B \vee C)$	Asociatividad
$A \wedge B \equiv B \wedge A$ $A \vee B \equiv B \vee A$	Conmutatividad
$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$	Distributividad
$A \wedge True \equiv A$ $A \vee False \equiv A$	Identidad
$A \wedge False \equiv False$ $A \vee True \equiv True$	Elemento nulo o dominación
$A \wedge A \equiv A$ $A \vee A \equiv A$	Idempotencia
$\neg(A \wedge B) \equiv \neg A \vee \neg B$ $\neg(A \vee B) \equiv \neg A \wedge \neg B$	De Morgan
$P \wedge \neg P \equiv False$	Contradicción
$A \vee \neg A \equiv True$	Tercero excluido
$\neg\neg A \equiv A$	Doble negación
$A \to B \equiv \neg A \vee B$ $A \leftrightarrow B \equiv (\neg A \vee B) \wedge (\neg B \vee A)$ $A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$	Equivalencia de operadores
$A \wedge (A \vee B) \equiv A$ $A \vee (A \wedge B) \equiv A$	Ley de absorción

Implementación.

Construimos el tipo de dato `LProp` para modelar la sintáxis de la lógica proposicional de la siguiente forma:

```
data LProp = PTrue | PFalse | Var Nombre | Neg LProp |
             Conj Lprop LProp | Disy LProp LProp | Impl LProp LProp |
             Syss LProp LProp

type Nombre = String               -- Nombre es un sinonimo para String.
type Asignacion = [(Nombre, Int)] -- Una asignación es una lista de tuplas.
```

Ejercicios.

Da la implementación cada una de las siguientes funciones utilizando `LProp`, cada función contribuye a tu calificación en ***un punto*** así que puedes alcanzar un 11.
En los ejemplos se utiliza **!** para representar la negación pero se espera que para el primer ejercicio la negación se implemente utilizando el símbolo ¬.

`show` Crea la instancia de la clase show para LProp utilizando los símbolos adecuados.

`vars` Función que recibe una LProp y regresa la lista con todas las variables que aparecen en la expresión.

```
vars (Syss (Impl (Var "a") (Var "b")) (Impl (Var "c") (Conj (Var "b") (Var "c")))) ...
     = ["a","b","c"]
vars (Impl PTrue PFalse) = []
```

`asocia_der` Función que recibe una LProp y aplica la ley de la asociatividad hacia la derecha sobre los elementos de la expresión.

```
asocia_der (Conj (Conj (Var "a") (Var "b")) (Var "c")) = (a ^ (b ^ c))
asocia_der (Conj (Var "a") (Conj (Var "b") (Var "c"))) = (a ^ (b ^ c))
asocia_der (Disy (Disy (Var "p") (Var "q")) (Var "r")) = (p v (q v r))
asocia_der (Disy (Var "p") (Disy (Var "q") (Var "r"))) = (p v (q v r))
asocia_der (Syss PFalse PFalse) =  (False <-> False)
```

`asocia_izq` Lo mismo que `asocia_der` pero para el otro lado.

```
asocia_izq (Conj (Conj (Var "a") (Var "b")) (Var "c")) = ((a ^ b) ^ c)
asocia_izq (Conj (Var "a") (Conj (Var "b") (Var "c"))) = ((a ^ b) ^ c)
asocia_izq (Disy (Disy (Var "p") (Var "q")) (Var "r")) = ((p v q) v r)
asocia_izq (Disy (Var "p") (Disy (Var "q") (Var "r"))) = ((p v q) v r)
asocia_izq (Syss PTrue PTrue) = (True <-> True)
```

`conm` Función que recibe una LPropr y aplica la ley de la conmutatividad de forma exhaustiva sobre los elementos de la expresión cuyo operador lógico sea conjunción o disyunción.

```
conm (Conj (Neg (Var "u")) PTrue) = (True ^ !u)
conm (Disy (Var "s") (Var "w")) = (w v s)
conm (Syss (Var "u") (Conj PTrue (Var "u"))) = (u <-> (u v True))
conm (Disy (Conj (Var "a") (Var "b")) (Var "c")) = (c v (b ^ a))
conm (Neg (Disy (Var "a") (Impl (Var "b") (Var "a")))) = ! ((b -> a) v a)
```

`dist` Función que recibe una LProp y aplica la ley de distributividad de forma exhaustiva sobre toda la expresión.

```
dist (Conj (Var "d") (Disy (Var "e") (Var "f"))) =  ((d ^ e) v (d ^ f))
dist (Disy (Var "s") (Conj (Var "t") (Var "u"))) =  ((s v t) ^ (s v u))
dist (Conj (Var "a") (Impl (Var "b") (Var "a"))) = ((b -> a) ^ a)
dist (Var "r") = r
dist (Disy (Var "i") PFalse) = (False v i)
dist PTrue =  True
```

`deMorgan` Función que le aplica a una LProp las leyes de De morgan.

```
deMorgan (Neg (Conj (Var "a") (Var "b"))) = (! a v !b)
deMorgan (Neg (Disy (Var "c") (Var "d"))) = (! c ^ !d)
deMorgan (Syss (Impl (Var "p") PFalse) (Var "s")) = ((p -> False) <-> s)
deMorgan (Neg (Neg (Var "o"))) = ! ! o
deMorgan (Neg (Impl (Var "h") (Var "j"))) = ! (h -> j)
```

`equiv_op` Función que recibe una LProp y aplica la equivalencia de operadores que se describe al inicio de este documento.

```
equiv_op (Impl PFalse (Var "q")) = (! False v q)
equiv_op (Syss PFalse PTrue) = ((! True v False) ^ (! False v True))
equiv_op (Syss (Var "k") (Neg PFalse)) = ((! k v ! False) ^ (! ! False v k))
equiv_op (Disy (Var "c") (Conj (Var "u") (Var "b"))) = (c v (u ^ b))
equiv_op (Conj (Disy (Var "s") (Var "w")) (Impl (Var "e") (Var "d"))) ...
          = ((s v w)^(! e v d))
```

`dobleNeg` Función que quita las dobles negaciones de una LProp.

```
dobleNeg (Neg (Neg (Conj (Neg (Neg (Var "x"))) (Var "y")))) = (x ^ y)
dobleNeg (Neg (Conj (Neg (Neg (Var "x"))) (Var "y"))) = !(x ^ y)
dobleNeg (Disy (Neg (Neg (Var "z"))) (Conj (Var "q") (Var "a"))) = (z v (q ^ a))
dobleNeg (Neg (Neg (Var "KLM "))) KLM
dobleNeg (Neg (Neg (Neg (Neg (Neg (Var "s")))))) = ! s
```

`num_conectivos` Función que redibe una LProp y contesta con el número de conectivos lógicos en la expresión.

```
num_conectivos (Var "ok") = 0
num_conectivos (Disy (Var "u") (Disy (Var "w") (Var "a"))) = 2
num_conectivos (Syss (Var "n") (Neg (Conj PTrue (Var "m")))) = 3
num_conectivos (Syss (Impl (Var "e") (Var "k")) ...
               (Impl (Var "j") (Conj (Var "a") (Var "l")))) = 4
num_conectivos (Neg (Neg (Neg (Neg (Neg (Var "s")))))) = 5
```

`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 (Impl (Impl (Var "p") (Var "q")) (Var "r"))
                [("p" ,0) ,("q" ,0) ,("r" ,0) ] = 0
interpretacion (Impl (Impl (Var "p") (Var "q")) (Var "r"))
               [("p" ,1) ,("q" ,0) ,("r" ,0) ] = 1
interpretacion (Disy (Impl (Var "p") (Var "q")) (Impl (Var "q") (Var "p")))
                [("p" ,1) ,("q" ,0) ] = 1
interpretacion (Syss (Var "x") (Var "y"))
                    [("x" ,0),("y" ,1) ]  = 0
interpretacion (Impl (Conj (Var "s") (Var "t")) (Var "r"))
                [("s" ,0) ,("t" ,1) ,("r" ,0) ]  = 1
```

Notas.

- Las instrucciones para entregar las prácticas están en la página del curso, si aún no sabes cuál es haz click aquí.
- Se responden dudas en el correo pero de preferencia enviame mensaje por Telegram. Puedes acceder a mi contacto haciendo click aqui.

No dudes en contactarme ante cualquier duda, aunque no sea de la materia.