

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-4700 Lenguajes de programación
Profesor: Ignacio Trejos Zelaya

Asignación #4 [Proyecto]: procesamiento simbólico con lenguajes funcionales

Integrantes

Juan Sebastián Gamboa Botero - 2020030303
David Suárez Acosta - 2020038304

Fecha de entrega:

27 de Noviembre, 2021

Índice

Introducción	3
Funciones vars, gen_bools, as_vals y evalProp	4
Función vars	4
Estrategia	4
Código	4
Funciones Auxiliares	4
Pruebas y resultados obtenidos	5
Referencias consultadas	5
Función gen_bools	6
Estrategia	6
Código	6
Pruebas y resultados obtenidos	6
Referencias consultadas	7
Función as_vals	8
Estrategia	8
Código	8
Pruebas y resultados obtenidos	8
Referencias consultadas	9
Función evalProp	10
Estrategia	10
Código	10
Funciones Auxiliares	11
Pruebas y resultados obtenidos	11
Referencias consultadas	11
Función taut	12
Estrategia	12
Código	12
Pruebas y resultados obtenidos	13
Función fnd	14
Estrategia	14
Código	14
Funciones Auxiliares	16
Pruebas y resultados obtenidos	16
Referencias consultadas	17
Función bonita	18
Estrategia	18
Código	18
Pruebas y resultados obtenidos	19

Análisis y discusión de resultados	20
vars	20
gen_bools	20
as_vals	20
evalProp	21
taut	22
fnd	23
bonita	23
Discusión	23
Conclusiones de resultados	24
Problemas y limitaciones encontradas	24
Reflexión	24
Tareas realizadas por miembros	25
Referencias	25
Apéndices	26
Instrucciones para la ejecución del programa	26
Código fuente	26
Pruebas para validar	38
Detalles	38

Introducción

El objetivo detrás de esta 4ta asignación del curso consiste en que los estudiantes hayan construido procesadores de expresiones simbólicas en el lenguaje funcional de Haskell, esto incluye un comprobador de tautologías, un convertidor de las proposiciones hacia su forma normal disyuntiva y un 'impresor' de estas proposiciones lógicas con variables. Todos estos requerimientos de la asignación fueron completados satisfactoriamente por los estudiantes que asimilaron el lenguaje Haskell de forma correcta.

Este documento se divide en 4 secciones principales para describir y explicar todas las funciones requeridas, las cuales son: 'Funciones vars, gen_bools, as_vals y evalProp', 'Función taut', 'Función fnd', 'Función bonita'. Seguidamente de estas secciones se hace una discusión y análisis de los resultados obtenidos, una conclusión, se mencionan los problemas encontrados y más datos pertinentes a la realización del trabajo.

Funciones vars, gen_bools, as_vals y evalProp

Función vars

Estrategia

La estrategia utilizada consiste en recorrer recursivamente la Proposición n con la condición de parada en las variables siendo que retorne una lista con el string del nombre de la variable. Esto se logra hacer por medio del “case n of” para poder determinar qué tipo de Proposición es el n actual. En los demás casos se llama recursivamente a sí misma y por medio de appends (++) se hace una lista con todos los strings conseguidos (varsAux).

Adicionalmente, es necesario eliminar los nombres de las variables repetidas de la lista, por lo que se usa una función que elimina las repetidas de la lista ingresada (uniq). Para dar el resultado correcto se hace una función que combina estas dos funciones, esta es vars.

Código

```
-- / / / vars: determina la lista de las distintas variables
proposicionales / / /

uniq :: Ord a => [a] -> [a] -- ---> Delete duplicates, make unique <---
uniq = toList . fromList

varsAux :: Proposicion -> [String]
varsAux n = case n of
    (Const _) -> []
    (Variable v) -> [v]
    (Negacion p) -> varsAux p
    (Conjuncion (p1, p2)) -> varsAux p1 ++ varsAux p2
    (Disyuncion (p1, p2)) -> varsAux p1 ++ varsAux p2
    (Implicacion (p1, p2)) -> varsAux p1 ++ varsAux p2
    (Equivalencia (p1, p2)) -> varsAux p1 ++ varsAux p2

vars :: Proposicion -> [String]
vars n = uniq (varsAux n)
```

Funciones Auxiliares

En este caso la función auxiliar es uniq, como se había explicado anteriormente, su propósito consiste en hacer la lista “única” sin elementos repetidos, esto se consigue en una línea de

código al usar `toList` y `fromList`. Este código está inspirado en un comentario hecho a una pregunta en StackOverflow (Stevenson, 2013).

Pruebas y resultados obtenidos

Dentro del archivo `main.hs`, se crearon las variables `vars1`, `vars2` y `vars3` utilizando la función `vars` con distintas proposiciones cada una. Para probar que funcione correctamente, en la terminal antes de mostrar el valor de cada variable, se muestra la proposición en sí.

```
245  -- vars
    vars1 :: [String] | vars1 :: [String]
246  vars1 = vars prop1
    vars2 :: [String] | vars2 :: [String]
247  vars2 = vars prop2
    vars3 :: [String] | vars3 :: [String]
248  vars3 = vars prop3
249

PROBLEMS 44 OUTPUT TERMINAL DEBUG CONSOLE

*Main> -- Pruebas vars --
*Main> prop1
Disyuncion (Conjuncion (Variable "p",Variable "q"),Variable "r")
*Main> vars1
["p","q","r"]
*Main> prop2
Equivalencia (Implicacion (Variable "a",Variable "b"),Disyuncion (Negacion (Variable "a"),Variable "b"))
*Main> vars2
["a","b"]
*Main> prop3
Disyuncion (Implicacion (Variable "A0",Conjuncion (Variable "B",Negacion (Variable "C1"))),Disyuncion (Conjuncion (Variable "D",Implicacion (Variable "T",Negacion (Negacion (Variable "R"))))),Negacion (Variable "B")))
*Main> vars3
["A0","B","C1","D","R","T"]
```

Como se puede observar, se consiguen correctamente todos los nombres de las variables en las distintas proposiciones sin ningún repetido dentro de una lista.

Referencias consultadas

(Stevenson, 2013)

Función `gen_bools`

Estrategia

Para generar una tabla de verdad en Haskell se logra con facilidad por medio de `mapM`. Esta instrucción consiste en que asigna a cada elemento de una estructura una acción (*mapM*, *n.d.*), por lo que en nuestro caso la estructura es `'const [True, False]'` y la “acción” nuestra lista de strings `v` que tienen los nombres de las variables. Al ingresar esto, va a generar todas las posibles combinaciones diferentes de `'const [True, False]'` por el número de elementos que posee `v`, generando así una matriz que es una tabla de verdad con 2^n combinaciones. La idea fue obtenida gracias a otra pregunta en StackOverflow (*Generating a truth table*, 2016).

Código

```
-- / / / gen_bools: produce todas las posibles combinaciones de valores
booleanos para n variables proposicionales (uses vars as parameter) / / /
gen_bools :: Traversable t => t b -> [t Bool]
gen_bools v = mapM (const [True, False]) v
```

Pruebas y resultados obtenidos

```

250  -- gen_bools
      gb1 :: [[Bool]] | gb1 :: [[Bool]]
251  gb1 = gen_bools vars1
      gb2 :: [[Bool]] | gb2 :: [[Bool]]
252  gb2 = gen_bools vars2
      gb3 :: [[Bool]] | gb3 :: [[Bool]]
253  gb3 = gen_bools vars3

```

PROBLEMS 44 OUTPUT TERMINAL DEBUG CONSOLE

```

*Main> gb1
[[True,True,True],[True,True,False],[True,False,True],[True,False,False],[False,True,True],[False,True,Fa
lse],[False,False,True],[False,False,False]]
*Main> gb2
[[True,True],[True,False],[False,True],[False,False]]
*Main> gb3
[[True,True,True,True,True],[True,True,True,True,True,False],[True,True,True,True,False,True],[True,True,True,Tr
ue,False,False],[True,True,True,False,True,True],[True,True,True,False,True,False],[True,True,True,False,False,Tr
ue],[True,True,True,False,False,False],[True,True,False,True,True,True],[True,True,False,True,True,False],[True,Tr
ue,False,True,True,False,False],[True,True,False,False,True,True],[True,True,False,False,True,True],[True,True,
False,False,False,False],[True,True,False,False,True],[True,True,False,False,False,False],[True,True,False,False,
False,True],[True,True,False,False,False,False],[False,True,True,True,True,True],[False,True,True,True,True,False
],[False,True,True,True,False,True],[False,True,True,True,False,False],[False,True,True,False,True,True],[False,
True,True,True,True,False],[False,True,True,True,False,False],[False,True,True,True,True,False],[False,True,True,
True,True],[False,True,False,True,True,False],[False,True,False,True,False,True],[False,True,False,True,False,Tru
e],[False,True,False,False,True,True],[False,True,False,False,True,False],[False,True,False,False,False,True],[False,
True,False,False,False],[False,False,True,True,True,True],[False,False,True,True,False],[False,False,True,True,
False,True],[False,False,True,True,False,False],[False,False,True,False,True,True],[False,False,True,False,True,
False],[False,False,True,True,False],[False,False,False,True,False,True],[False,False,False,True,False,False],[False,
False,False,False,True],[False,False,False,False,True,False],[False,False,False,False,False,True],[False,False,
False,False,False]]

```

Se realizaron las pruebas de gen_bools con las mismas proposiciones usadas en las pruebas anteriores y más específicamente, con las variables conseguidas al probar vars.

Como se puede observar:

- gb1 género $2^3 = 8$ combinaciones.
- gb2 género $2^2 = 4$ combinaciones.
- gb3 género $2^6 = 64$ combinaciones.

Cada una de estas variables correctamente representa una tabla de verdad. Ver Figura 1.1 y Figura 1.2 y comparar resultados con gb1 y gb2 respectivamente (por efectos de espacio es poco práctico colocar la comprobación de gb3 por sus 64 combinaciones).

Referencias consultadas

(*Generating a truth table*, 2016), (*mapM*, n.d.)

Función `as_vals`

Estrategia

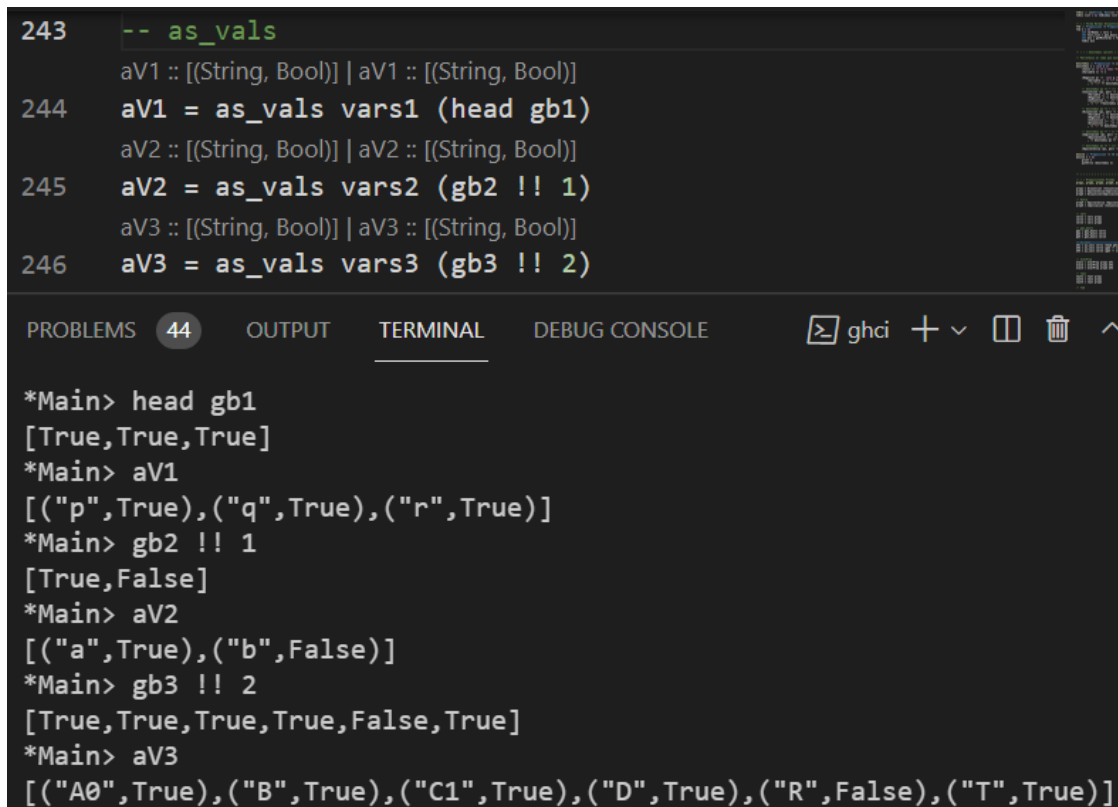
El hecho de combinar dos listas y devolver una con cada de los elementos combinados en partes es lo que hace 'zip' en Haskell como tal (*Zip, n.d.*), por lo que en este caso simplemente es necesario definir los valores que va a aceptar y retornar la función `as_vals`, y luego se iguala a `zip`.

Ahora, el uso en contexto de la función con las funciones anteriores consiste en que a la lista de los nombres de las variables (conseguida en `vars`) van a combinarse con una de las listas dentro de la lista obtenida con `gen_bools`, y esto es la asignación de valores.

Código

```
-- / / / as_vals: dada una lista de variables proposicionales sin
repeticiones, la combina con una lista de valores booleanos (uses vars and
one list of gen_bools) -> zip / / /
as_vals :: [String] -> [Bool] -> [(String, Bool)]
as_vals = zip
```

Pruebas y resultados obtenidos



```
243 -- as_vals
    aV1 :: [(String, Bool)] | aV1 :: [(String, Bool)]
244 aV1 = as_vals vars1 (head gb1)
    aV2 :: [(String, Bool)] | aV2 :: [(String, Bool)]
245 aV2 = as_vals vars2 (gb2 !! 1)
    aV3 :: [(String, Bool)] | aV3 :: [(String, Bool)]
246 aV3 = as_vals vars3 (gb3 !! 2)
```

PROBLEMS 44 OUTPUT TERMINAL DEBUG CONSOLE ghci + - [] [X] ^

```
*Main> head gb1
[True,True,True]
*Main> aV1
[("p",True),("q",True),("r",True)]
*Main> gb2 !! 1
[True,False]
*Main> aV2
[("a",True),("b",False)]
*Main> gb3 !! 2
[True,True,True,True,False,True]
*Main> aV3
[("A0",True),("B",True),("C1",True),("D",True),("R",False),("T",True)]
```

Como se puede observar, correctamente se consigue la asignación de valores para cada uno de los casos.

Referencias consultadas

(Zip, n.d.)

Función evalProp

Estrategia

La estrategia empleada para poder evaluar una proposición dada una asignación de valores consiste en recorrer recursivamente la Proposición y dependiendo del tipo de esta se realiza su respectiva acción (Negación: not, Conjunción: &&, Disyunción: ||, Implicación: ==>, Equivalencia: <=>). En el caso de que sea una variable, se tiene que conseguir el valor que está asociado a esta para que sucedan las operaciones matemáticas, por lo que la función de searchVal tiene este propósito.

Código

```
-- / / / evalProp: evalúa una proposición dada una asignación de valores
-- (uses as_vals) / / /

-- / / Search function

-- Gets the value assigned to the variable
searchValAux :: Eq t => t -> [(t, p)] -> Int -> p
searchValAux varName list index = do
    let val = list !! index
    if fst val == varName
        then snd val
        else searchValAux varName list (index-1)

-- Calls the aux
searchVal :: Eq t => t -> [(t, p)] -> p
searchVal var list = searchValAux var list (length list -1)

-- / / Operators needed

-- Implication
(==>) :: Bool -> Bool -> Bool
(==>) a b = not a || b

-- Equivalence
(<=>) :: Bool -> Bool -> Bool
(<=>) a b = a ==> b && b ==> a
```

```

--evalProp :: Proposicion -> [(String, Bool)] -> Bool
evalProp n val_list = case n of
    (Const valor) -> valor
    (Variable var) -> searchVal var val_list
    (Negacion p) -> not (evalProp p val_list)
    (Conjuncion (p1, p2)) -> evalProp p1 val_list && evalProp p2 val_list
    (Disyuncion (p1, p2)) -> evalProp p1 val_list || evalProp p2 val_list
    (Implicacion (p1, p2)) -> evalProp p1 val_list ==> evalProp p2
val_list
    (Equivalencia (p1, p2)) -> evalProp p1 val_list <=> evalProp p2
val_list

```

Funciones Auxiliares

Como fue mencionado anteriormente, searchVal y searchValAux son utilizadas para conseguir el valor asignado al string del nombre de una variable. Esto es logrado por medio de recorrer la asignación de pares recursivamente y comparar si el string actual es el mismo al string buscado, si lo es se retorna el segundo valor de la tupla en donde está el string.

Adicional a las funciones anteriores, se programaron los operadores de implicación y equivalencia para ser utilizados dentro de la función evalProp. Esto fue logrado gracias a StackOverflow (*How do I create an operator in Haskell?*, 2012).

```

261  -- evalProp
      eval1 :: Bool | eval1 :: Bool
262  eval1 = evalProp prop1 aV1
      eval2 :: Bool | eval2 :: Bool
263  eval2 = evalProp prop2 aV2
      eval3 :: Bool | eval3 :: Bool
264  eval3 = evalProp prop3 aV3
265
PROBLEMS 44 OUTPUT TERMINAL
*Main> eval1
True
*Main> eval2
True
*Main> eval3
False

```

Pruebas y resultados obtenidos

Como se puede observar, evalProp recibe una proposición y una asignación de valores que han sido utilizados anteriormente, al realizarse para cada caso, se puede observar cómo evalúa las proposiciones correctamente en todos los casos (comprobación en 'Análisis y discusión de resultados').

Referencias consultadas

(*How do I create an operator in Haskell?*, 2012)

Función taut

Estrategia

Una proposición es una tautología si al evaluarla con todas las combinaciones de la tabla de verdad, da verdadero en todos los casos, por lo que esto es exactamente el objetivo de tautAux y taut. La función tautAux consiste en que evalúa la proposición dada con cada lista de la matriz de gen_bools, y en el caso en que alguna evaluación (evalProp) de falso, imprime en consola que no es una tautología y unos valores que llegaron a falsificarla. En el caso en que no suceda esto y recorra todas las posibles combinaciones de gen_bools, imprime que es una tautología. Adicionalmente, la función taut imprime antes de llamar a tautAux la proposición como fue indicado.

Código

```
-- / / / taut (if evalProp == True para todos los valores posibles) / / /

tautAux :: Proposicion -> [[Bool]] -> [String] -> Int -> IO ()
tautAux n full_list varNames index = do
    let list = full_list !! index
    let val_list = as_vals varNames list
    if index == -1 -- Every single one has been True -> "Si es una
tautologia"
        then putStrLn "Si es una tautologia"
        else if not(evalProp n val_list) -- False, must give answer on
where it failed -> Ej: "No es una tautologia debido a que se falsifica con
A: False, B: True"
        then putStrLn ("No es una tautologia debido a que se falsifica
con " ++ show val_list)
        else tautAux n full_list varNames (index - 1) -- Recursiva

taut :: Proposicion -> IO ()
taut n = do
    print n
    let varNames = vars n
    let full_list = gen_bools varNames
    tautAux n full_list varNames (length full_list -1)
```

Pruebas y resultados obtenidos

```
266  -- taut
    taut1 :: IO () | taut1 :: IO ()
267  taut1 = taut prop1
    taut2 :: IO () | taut2 :: IO ()
268  taut2 = taut prop2
    taut3 :: IO () | taut3 :: IO ()
269  taut3 = taut prop3
270

PROBLEMS 44 OUTPUT TERMINAL DEBUG CONSOLE

*Main> taut1
Disyuncion (Conjuncion (Variable "p",Variable "q"),Variable "r")
No es una tautologia debido a que se falsifica con [("p",False),("q",False),("r",False)]
*Main> taut2
Equivalencia (Implicacion (Variable "a",Variable "b"),Disyuncion (Negacion (Variable "a"),Variable "b"))
Si es una tautologia
*Main> taut3
Disyuncion (Implicacion (Variable "A0",Conjuncion (Variable "B",Negacion (Variable "C1"))),Disyuncion (Conjuncion
(Variable "D",Implicacion (Variable "T",Negacion (Negacion (Variable "R")))),Negacion (Variable "B")))
No es una tautologia debido a que se falsifica con [("A0",True),("B",True),("C1",True),("D",False),("R",False),("
T",False)]
```

Como se puede observar, para cada uno de los casos en la terminal imprime la proposición y luego menciona si esta es o no una tautología (y en el caso que no sea, los valores que la falsifican). Comprobación en ‘Análisis y discusión de resultados’.

Función fnd

Estrategia

Un minitérmino es una expresión lógica que contiene todas las variables relacionadas entre sí únicamente con los operadores lógicos AND (conjunción) y NOT (negación) (*MINITERMINOS Y maxiterminos*, *n.d.*). La forma normal disyuntiva (fnd) es una forma de representar una expresión lógica como una “suma de productos” o más específicamente, una suma de miniterminos (Lordi, 2018), así que con esto en mente, la forma normal disyuntiva se obtiene al sumar (disyunción) los minitérminos que dan un valor de verdadero.

La estrategia aplicada para lograr este concepto consiste en conseguir todos los minitérminos de una proposición en una lista al evaluarla con sus distintas combinaciones en la tabla de verdad y que de un valor de verdadero (getMinterms). Con esta lista de miniterminos, el siguiente paso es transformar cada minitérmino a proposiciones con conjunciones y negaciones, mientras que cada uno de estos minitérminos se relacionan entre sí por medio de disyunciones. Esto nos da una proposición en el formato que deseamos.

Código

```
-- / / / fnd: Forma Normal Disyuntiva (Consigo los miniterminos de cuando
es true y se suman -> Escribirlo de forma Proposicion) / / /
-- OR of ANDs, a sum of products (* : and / + : or)

-- / / Minterms -> Consigue matriz de minterms (cada variable dentro de
cada lista es una multiplicacion/conjuncion/and y las listas en si son una
suma/disyuncion/or)
getMinterms :: Proposicion -> [[Bool]] -> [String] -> Int -> [(String,
Bool)]
getMinterms n full_list varNames index = do
    let list = full_list !! index
    let val_list = as_vals varNames list
    if index == -1 -- End
    then []
    else if evalProp n val_list -- True, get val_list
        then val_list : getMinterms n full_list varNames (index - 1)
        else getMinterms n full_list varNames (index - 1)

-- / Transforma los valores de una lista de 1 minterm en sus respectivas
operaciones (Conjuncion / Negacion)
toConjAux :: [(String, Bool)] -> Int -> Proposicion
```

```

toConjAux list index = do
    let tup0 = head list
    let v0 = Variable (fst tup0)
    let tup1 = list !! 1
    let v1 = Variable (fst tup1)
    let tupX = list !! index
    let vX = Variable (fst tupX)

    if index >= 2
    then if snd tupX
        then Conjuncion(toConjAux list (index-1), vX)
        else Conjuncion(toConjAux list (index-1), Negacion vX)
    else if snd tup0 && snd tup1
        then Conjuncion(v0, v1)
        else if not(snd tup0) && not (snd tup1)
            then Conjuncion(Negacion v0, Negacion v1)
            else if not(snd tup0)
                then Conjuncion(Negacion v0, v1)
                else Conjuncion(v0, Negacion v1)

toConj :: [(String, Bool)] -> Proposicion
toConj list = do toConjAux list (length list -1)

-- / Transforma los valores de una lista de todos los minterms en sus
respectivas operaciones (Disyuncion / Conjuncion / Negacion)
toDisAux :: [[(String, Bool)]] -> Int -> Proposicion
toDisAux list index = do
    let p0 = toConj (head list)
    let p1 = toConj (list !! 1)
    let pX = toConj (list !! index)

    if index >= 2
    then Disyuncion(toDisAux list (index-1), pX)
    else Disyuncion(p0, p1)

toDis :: [[(String, Bool)]] -> Proposicion
toDis list = do toDisAux list (length list -1)

```



```
-- / / Forma Normal Disyuntiva
fnd :: Proposicion -> Proposicion
fnd n = do
    let varNames = vars n
    let full_list = gen_bools varNames
    let min = getMinterms n full_list varNames (length full_list -1)
    toDis min
```

Funciones Auxiliares

La función `fnd` contiene una gran cantidad de funciones auxiliares, la función `getMinterms` consigue una lista de miniterminos (lista de asignaciones de valores) como se mencionó anteriormente, al utilizar `evalProp` con todos los valores de `gen_bools` unido con `vars` por medio de `as_vals`, si da verdadero, la lista de `as_vals` es un minitermino como tal, y si encuentra más, estos miniterminos se agregan todos a una lista.

Esta lista se transforma por medio de las funciones `toConj`, `toDis` y sus respectivas auxiliares. Las funciones `toConj` y `toConjAux` reciben una lista de asignación de valores y lo convierte a una proposición hecha de conjunciones y negaciones en los casos que sean necesarios. La función `toDis` y `toDisAux` recursivamente cambian por completo la lista de listas de asignaciones de valores en la proposición final al agregar disyunciones entre los datos obtenidos por `toConj`.

Con todo esto hecho, la función `fnd` une todas estas funciones auxiliares para conseguir el resultado deseado.

Pruebas y resultados obtenidos

```
271 -- fnd
    fnd1 :: Proposition | fnd1 :: Proposition
272 fnd1 = fnd prop1
    fnd2 :: Proposition | fnd2 :: Proposition
273 fnd2 = fnd prop2
    fnd3 :: Proposition | fnd3 :: Proposition
274 fnd3 = fnd prop3
275
```

PROBLEMS 43 OUTPUT TERMINAL DEBUG CONSOLE

```
*Main> fnd1
Disyuncion (Disyuncion (Disyuncion (Disyuncion (Conjuncion (Conjuncion (Negacion (Variable "p"),Negacion (Variable "q")),Variable "r"),Conjuncion (Conjuncion (Negacion (Variable "p"),Variable "q"),Variable "r")),Conjuncion (Conjuncion (Variable "p",Negacion (Variable "q")),Variable "r")),Conjuncion (Conjuncion (Variable "p",Variable "q"),Negacion (Variable "r"))),Conjuncion (Conjuncion (Variable "p",Variable "q"),Variable "r"))
*Main> fnd2
Disyuncion (Disyuncion (Disyuncion (Conjuncion (Negacion (Variable "a"),Negacion (Variable "b")),Conjuncion (Negacion (Variable "a"),Variable "b")),Conjuncion (Variable "a",Negacion (Variable "b"))),Conjuncion (Variable "a",Variable "b"))
*Main> -- fnd3 da un resultado muy grande para la captura
```

Como se puede observar, consigue la proposición en forma normal disyuntiva para las proposiciones anteriores de manera correcta (como fue mencionado en la captura de pantalla, el resultado de `fnd3` es muy grande para poder representarlo visualmente, por lo que no se incluye en esta. Comprobación en 'Análisis y discusión de resultados'.

Referencias consultadas

(*MINITERMINOS Y maxiterminos, n.d.*), (Lordi, 2018).

Función bonita

Estrategia

La estrategia utilizada para la impresión de una proposición consiste en recursivamente recorrerla en “Inorder” para ir imprimiendo los símbolos en sus respectivas posiciones. Para el caso de donde poner los paréntesis, estos se agregan al lado en donde se asocia cada operador en el caso que el operador que esté a ese lado tenga un menor orden de presencia al actual como se interpreta en las indicaciones. Esto es lo que sucede en bonitaAux, mientras que bonita además de llamar a la otra función, antes imprime la proposición en sí.

Código

```
-- / / / bonitaAux (print) / / /

-- Paréntesis al lado que asocian si la precedencia de ese lado es menor a
la del caso actual

bonitaAux :: Proposicion -> [Char]
bonitaAux n = case n of
  (Const c) -> if c then "True" else "False"
  (Variable v) -> v

  (Negacion p) -> case p of
    (Variable _) -> "~" ++ bonitaAux p
    _ -> "~(" ++ bonitaAux p ++ ")"

  -- bonitaAux p1 ++ " /\ " ++ bonitaAux p2
  (Conjuncion (p1, p2)) -> case p1 of
    (Variable _) -> bonitaAux p1 ++ " /\ " ++ bonitaAux p2
    (Negacion _) -> bonitaAux p1 ++ " /\ " ++ bonitaAux p2
    (Conjuncion (_, _)) -> bonitaAux p1 ++ " /\ " ++ bonitaAux p2
    _ -> "(" ++ bonitaAux p1 ++ ") /\ " ++ bonitaAux p2

  -- bonitaAux p1 ++ " \/ " ++ bonitaAux p2
  (Disyuncion (p1, p2)) -> case p1 of
    (Variable _) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    (Negacion _) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    (Conjuncion (_, _)) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    (Disyuncion (_, _)) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    _ -> "(" ++ bonitaAux p1 ++ ") \/ " ++ bonitaAux p2
```

```

-- bonitaAux p1 ++ " => " ++ bonitaAux p2
(Implicacion (p1, p2)) -> case p1 of -- >> Derecha << --
    (Equivalencia (_, _)) -> bonitaAux p1 ++ " => (" ++ bonitaAux p2
++ " )"
    _ -> bonitaAux p1 ++ " => " ++ bonitaAux p2

-- bonitaAux p1 ++ " <=> " ++ bonitaAux p2
(Equivalencia (p1, p2)) -> bonitaAux p1 ++ " <=> " ++ bonitaAux p2

bonita :: Proposicion -> IO ()
bonita n = do
    print n
    putStrLn (bonitaAux n)

```

Pruebas y resultados obtenidos

```

276 -- bonita
    b1 :: IO () | b1 :: IO ()
277 b1 = bonita prop1
    b2 :: IO () | b2 :: IO ()
278 b2 = bonita prop2
    b3 :: IO () | b3 :: IO ()
279 b3 = bonita prop3
    b4 :: IO () | b4 :: IO ()
280 b4 = bonita (Conjuncion(Disyuncion(Variable "a", Variable "b"), Variable "c"))
281
PROBLEMS 44 OUTPUT TERMINAL DEBUG CONSOLE

*Main> b1
Disyuncion (Conjuncion (Variable "p",Variable "q"),Variable "r")
p /\ q \/ r
*Main> b2
Equivalencia (Implicacion (Variable "a",Variable "b"),Disyuncion (Negacion (Variable "a"),Variable "b"))
a => b <=> ~a \/ b
*Main> b3
Disyuncion (Implicacion (Variable "A0",Conjuncion (Variable "B",Negacion (Variable "C1"))),Disyuncion (Conjuncion
(Variable "D",Implicacion (Variable "T",Negacion (Negacion (Variable "R")))),Negacion (Variable "B")))
(A0 => B /\ ~C1) \/ D /\ T => ~(~R) \/ ~B
*Main> b4
Conjuncion (Disyuncion (Variable "a",Variable "b"),Variable "c")
(a \/ b) /\ c

```

En el caso de los resultados obtenidos, se puede observar correctamente la impresión de b1, b2 y b4. Para el caso de b3, unos paréntesis no se encuentran en donde deberían de estar debido a la forma en que se interpretó como deberían de ponerse estos (solo al lado asociado y si tiene menor precedencia).

Análisis y discusión de resultados

vars

Los datos obtenidos están correctos.

gen_bools

Los datos obtenidos están correctos, adicionalmente se muestra su comprobación por medio de las siguientes figuras:

Figura 1.1: Patrones de una Tabla de verdad de 3 variables

P	Q	R
True	True	True
True	True	False
True	False	True
True	False	False
False	True	True
False	True	False
False	False	True
False	False	False

Figura 1.2: Patrones de una Tabla de verdad de 2 variables

a	b
True	True
True	False
False	True
False	False

as_vals

Los datos obtenidos están correctos.

evalProp

Comprobación de cada uno de los casos:

eval1: True

- Proposición: $p \wedge q \vee r$
- Asignación de valores: $[("p", \text{True}), ("q", \text{True}), ("r", \text{True})]$
- Solución:

$$\text{True} \wedge \text{True} \vee$$
$$\text{True}$$

$$\text{True}$$

eval2: True

- Proposición: $a \Rightarrow b \Leftrightarrow \neg a \vee b$
- Asignación de valores: $[("a", \text{True}), ("b", \text{False})]$
- Solución:

$$\text{True} \Rightarrow \text{False} \Leftrightarrow \neg(\text{True}) \vee \text{False}$$

$$\text{False} \Leftrightarrow \text{False} \vee \text{False}$$

$$\text{True}$$

eval3: False

- Proposición: $(A0 \Rightarrow (B \wedge \neg C1)) \vee (D \wedge (T \Rightarrow \neg(\neg R))) \vee \neg B$
- Asignación de valores: $[("A0", \text{True}), ("B", \text{True}), ("C1", \text{True}), ("D", \text{True}), ("R", \text{False}), ("T", \text{True})]$
- Solución:

$$(\text{True} \Rightarrow (\text{True} \wedge \neg(\text{True}))) \vee (\text{True} \wedge (\text{True} \Rightarrow \neg(\neg(\text{False})))) \vee \neg(\text{True})$$

$$(\text{True} \Rightarrow (\text{True} \wedge \text{False})) \vee (\text{True} \wedge (\text{True} \Rightarrow \text{False}) \vee \text{False})$$

$$\text{False} \vee (\text{True} \wedge \text{False} \vee \text{False})$$

$$\text{False}$$

taut

Comprobación de cada uno de los casos:

taut1:

- Proposición: $p \wedge q \vee r$
- Falsificación: $[("p", \text{False}), ("q", \text{False}), ("r", \text{False})]$
- Demostración:

$$\text{False} \wedge \text{False} \vee \text{False}$$

False

taut2:

- Proposición: $a \Rightarrow b \Leftrightarrow \neg a \vee b$
- Demostración Tautología:

Figura 1.3: Tabla de verdad de $a \Rightarrow b \Leftrightarrow \neg a \vee b$

a	b	$a \Rightarrow b$	$\neg a$	$\neg a \vee b$	$a \Rightarrow b \Leftrightarrow \neg a \vee b$
True	True	True	False	True	True
True	False	False	False	False	True
False	True	True	True	True	True
False	False	True	True	True	True

taut3:

Comprobada que no es una tautología con eval3. Adicionalmente, los valores impresos $[("A0", \text{True}), ("B", \text{True}), ("C1", \text{True}), ("D", \text{False}), ("R", \text{False}), ("T", \text{False})]$ también la falsifican.

fnd

Comprobación de fnd2 y fnd1.

fnd2:

- Respuesta:

Disyuncion (Disyuncion (Disyuncion (Conjuncion (Negacion (Variable "a"),Negacion (Variable "b")),Conjuncion (Negacion (Variable "a"),Variable "b")),Conjuncion (Variable "a",Negacion (Variable "b")),Conjuncion (Variable "a",Variable "b"))

- FND: $(\neg a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$

La respuesta obtenida se puede visualizar como FND. Esta es verdadera debido a que la proposición 2 es una tautología, como se puede observar en taut2. Debido a esto, todas las posibles combinaciones de la tabla de verdad son minitérminos (conjunciones) y están sumando entre sí (disyunciones).

fnd1:

- Respuesta:

Disyuncion (Disyuncion (Disyuncion (Disyuncion (Conjuncion (Conjuncion (Negacion (Variable "p"),Negacion (Variable "q")),Variable "r"),Conjuncion (Conjuncion (Negacion (Variable "p"),Variable "q"),Variable "r")),Conjuncion (Conjuncion (Variable "p",Negacion (Variable "q")),Variable "r")),Conjuncion (Conjuncion (Variable "p",Variable "q"),Negacion (Variable "r"))),Conjuncion (Conjuncion (Variable "p",Variable "q"),Variable "r"))

- FND: $(\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r) \vee (p \wedge q \wedge \neg r) \vee (p \wedge q \wedge r)$

La respuesta obtenida se puede visualizar como FND. Esta es verdadera debido que al probar cada uno de los minitérminos, estos son los únicos que al ser evaluados con la proposición dan un valor verdadero.

bonita

No todas las impresiones están correctas por la posición de ciertos paréntesis, explicación en 'Función bonita'.

Discusión

Los datos obtenidos muestran que el procesamiento simbólico con lenguajes funcionales es algo no solo posible, si no también bastante eficiente e interesante.

Conclusiones de resultados

Se implementaron correctamente la mayoría de las funciones indicadas. ‘vars’, ‘gen_bools’, ‘as_vals’, ‘evalProp’, ‘taut’ y ‘fnd’ dan respuestas correctas en todos los casos probados, mientras que en el caso de ‘bonita’, en algunos casos los paréntesis no son colocados en donde deberían de estar. Cabe recalcar que la impresión de los símbolos y las variables fueron logradas correctamente.

Problemas y limitaciones encontradas

Como ha sido mencionado en varias de las secciones anteriores, los paréntesis en la función ‘bonita’. La lógica detrás de la implementación de estos paréntesis es bastante clara y es congruente, pero no se aplican correctamente en todos los casos tratados.

Adicionalmente a esto, una limitación que se nos presentó al trabajar con Haskell fue la dificultad de dividir el código en distintos módulos para tener dos archivos (uno con las funciones y el otro con las pruebas). No se logró esto por lo que solo tenemos un archivo de código “main.hs”.

Reflexión

Llegamos a la conclusión que los lenguajes al ser funcionales ofrecen varias herramientas para trabajar en la creación y evaluación de expresiones recursivas. Esta recursión fue un elemento imprescindible para lograr los resultados obtenidos. No solo debido a que el paradigma de ambos lenguajes lo facilita bastante, sino también gracias a que varias de las soluciones consisten en recorrer una estructura de esta forma para ir dividiéndola en partes y agrupar los resultados deseados en algo nuevo. Unido a esto, consideramos que varias de las funciones realizadas fueron hechas de la forma más óptima posible al utilizar recursividad pura.

Inicialmente este trabajo comenzó como un desafío debido a la inexperiencia de uso de ambos lenguajes, pero “datatype” en SML y “data” en Haskell sirven como una clara representación de una proposición en donde se puede observar con facilidad todas las partes que la componen. Al tener esta representación bien definida y comprendida, la realización de los problemas fue bastante intuitiva.

Tareas realizadas por miembros

Podemos decir con certeza que ambos miembros trabajaron en todas las partes del código por medio de llamadas realizadas en Discord en donde inicialmente se intentó comprender los requerimientos indicados, y conforme las llamadas subsecuentes se aportaron ideas de cómo desarrollar todas las funciones. En estas llamadas se compartió la pantalla para analizar y escribir el código mientras se comentaba y se intentaba realizar de la forma más óptima posible. La realización de este proyecto se debe gracias al esfuerzo grupal mostrado por ambos estudiantes.

Referencias

Generating a truth table. Stack Overflow. (2016). Retrieved November 23, 2021, from <https://stackoverflow.com/questions/35120766/generating-a-truth-table>.

How do I create an operator in Haskell? Stack Overflow. (2012). Retrieved November 23, 2021, from <https://stackoverflow.com/questions/9356442/how-do-i-create-an-operator-in-haskell>.

Lordi, M. (2018). *Forma normal disyuntiva* [Video]. Youtube. Retrieved November 24, 2021, from <https://www.youtube.com/watch?v=we9FBfrOQ9Y>

mapM. Hoogle. (n.d.). Retrieved November 23, 2021, from <https://hoogle.haskell.org/?hoogle=mapM>.

MINITERMINOS Y maxiterminos. ELECTRONICA DIGITAL - ELECTRONICA DIGITAL Introduccion. (n.d.). Retrieved November 26, 2021, from <https://electronicadigital6bm.es.tl/MINITERMINOS-Y-MAXITERMINOS.htm>.

Stevenson, B. (2013). *Removing duplicates from a list in Haskell without elem*. Stack Overflow. Retrieved November 23, 2021, from <https://stackoverflow.com/questions/16108714/removing-duplicates-from-a-list-in-haskell-without-elem>.

Zip. Hoogle. (n.d.). Retrieved November 26, 2021, from <https://hoogle.haskell.org/?hoogle=zip>.

Apéndices

Instrucciones para la ejecución del programa

1. Tener instalado Haskell correctamente con todas sus herramientas necesarias.
2. En una terminal, meterse en la dirección en donde se encuentra main.hs
3. Escribir el comando “ghci” para poder empezar a correr código en Haskell.
4. Escribir el comando “:l main.hs” para cargar el archivo
5. Para cualquiera de las pruebas, escribir el nombre de la variable que se desea observar sus contenidos en la terminal. Por ejemplo vars1, gb1, aV1, taut1, entre otros.
6. En el caso de que se desee probar una funcion en especifico con su propio ejemplo, basarse en las pruebas escritas en el código que empiezan en la línea 219.

Código fuente

main.hs

```
import Data.Set
import Data.List ()
import Data.Typeable

data Proposicion = Const Bool
  | Variable String
  | Negacion Proposicion
  | Conjuncion (Proposicion, Proposicion)
  | Disyuncion (Proposicion, Proposicion)
  | Implicacion (Proposicion, Proposicion)
  | Equivalencia (Proposicion, Proposicion)
  deriving Show

-- / / / vars: determina la lista de las distintas variables
-- proposicionales / / /

uniq :: Ord a => [a] -> [a] -- ---> Delete duplicates, make unique <---
uniq = toList . fromList

varsAux :: Proposicion -> [String]
varsAux n = case n of
  (Const _) -> []
```

```

    (Variable v) -> [v]
    (Negacion p) -> varsAux p
    (Conjuncion (p1, p2)) -> varsAux p1 ++ varsAux p2
    (Disyuncion (p1, p2)) -> varsAux p1 ++ varsAux p2
    (Implicacion (p1, p2)) -> varsAux p1 ++ varsAux p2
    (Equivalencia (p1, p2)) -> varsAux p1 ++ varsAux p2

vars :: Proposicion -> [String]
vars n = uniq (varsAux n)

-- / / / gen_bools: produce todas las posibles combinaciones de valores
-- booleanos para n variables proposicionales (uses vars as parameter) / / /
gen_bools :: Traversable t => t b -> [t Bool]
gen_bools v = mapM (const [True, False]) v

-- / / / as_vals: dada una lista de variables proposicionales sin
-- repeticiones, la combina con una lista de valores booleanos (uses vars and
-- one list of gen_bools) -> zip / / /
as_vals :: [String] -> [Bool] -> [(String, Bool)]
as_vals = zip

-- / / / evalProp: evalúa una proposición dada una asignación de valores
-- (uses as_vals) / / /

-- / / Search function

-- Gets the value assigned to the variable
searchValAux :: Eq t => t -> [(t, p)] -> Int -> p
searchValAux varName list index = do
    let val = list !! index
    if fst val == varName
    then snd val
    else searchValAux varName list (index-1)

```

```

-- Calls the aux
searchVal :: Eq t => t -> [(t, p)] -> p
searchVal var list = searchValAux var list (length list -1)

-- / / Operators needed

-- Implication
(==>) :: Bool -> Bool -> Bool
(==>) a b = not a || b

-- Equivalence
(<=>) :: Bool -> Bool -> Bool
(<=>) a b = a ==> b && b ==> a

--evalProp :: Proposicion -> [(String, Bool)] -> Bool
evalProp n val_list = case n of
    (Const valor) -> valor
    (Variable var) -> searchVal var val_list
    (Negacion p) -> not (evalProp p val_list)
    (Conjuncion (p1, p2)) -> evalProp p1 val_list && evalProp p2 val_list
    (Disyuncion (p1, p2)) -> evalProp p1 val_list || evalProp p2 val_list
    (Implicacion (p1, p2)) -> evalProp p1 val_list ==> evalProp p2
val_list
    (Equivalencia (p1, p2)) -> evalProp p1 val_list <=> evalProp p2
val_list

-- / / / taut (if evalProp == True para todos los valores posibles) / / /

tautAux :: Proposicion -> [[Bool]] -> [String] -> Int -> IO ()
tautAux n full_list varNames index = do
    let list = full_list !! index
    let val_list = as_vals varNames list
    if index == -1 -- Every single one has been True -> "Si es una
tautologia"
        then putStrLn "Si es una tautologia"

```

```

        else if not(evalProp n val_list) -- False, must give answer on
where it failed -> Ej: "No es una tautologia debido a que se falsifica con
A: False, B: True"
        then putStrLn ("No es una tautologia debido a que se falsifica
con " ++ show val_list)
        else tautAux n full_list varNames (index - 1) -- Recursiva

taut :: Proposicion -> IO ()
taut n = do
    print n
    let varNames = vars n
    let full_list = gen_bools varNames
    tautAux n full_list varNames (length full_list -1)

-- / / / fnd: Forma Normal Disyuntiva (Consigo los miniterminos de cuando
es true y se suman -> Escribirlo de forma Proposicion) / / /
-- OR of ANDs, a sum of products (* : and / + : or)

-- / / Minterms -> Consigue matriz de minterms (cada variable dentro de
cada lista es una multiplicacion/conjuncion/and y las listas en si son una
suma/disyuncion/or)
getMinterms :: Proposicion -> [[Bool]] -> [String] -> Int -> [(String,
Bool)]
getMinterms n full_list varNames index = do
    let list = full_list !! index
    let val_list = as_vals varNames list
    if index == -1 -- End
    then []
    else if evalProp n val_list -- True, get val_list
        then val_list : getMinterms n full_list varNames (index - 1)
        else getMinterms n full_list varNames (index - 1)

-- / Transforma los valores de una lista de 1 minterm en sus respectivas
operaciones (Conjuncion / Negacion)
toConjAux :: [(String, Bool)] -> Int -> Proposicion
toConjAux list index = do

```

```

let tup0 = head list
let v0 = Variable (fst tup0)
let tup1 = list !! 1
let v1 = Variable (fst tup1)
let tupX = list !! index
let vX = Variable (fst tupX)

if index >= 2
  then if snd tupX
    then Conjunction(toConjAux list (index-1), vX)
    else Conjunction(toConjAux list (index-1), Negacion vX)
  else if snd tup0 && snd tup1
    then Conjunction(v0, v1)
    else if not(snd tup0) && not (snd tup1)
      then Conjunction(Negacion v0, Negacion v1)
    else if not(snd tup0)
      then Conjunction(Negacion v0, v1)
      else Conjunction(v0, Negacion v1)

toConj :: [(String, Bool)] -> Proposicion
toConj list = do toConjAux list (length list -1)

-- / Transforma los valores de una lista de todos los minterms en sus
-- respectivas operaciones (Disyuncion / Conjunction / Negacion)
toDisAux :: [[(String, Bool)]] -> Int -> Proposicion
toDisAux list index = do
  let p0 = toConj (head list)
  let p1 = toConj (list !! 1)
  let pX = toConj (list !! index)

  if index >= 2
    then Disyuncion(toDisAux list (index-1), pX)
    else Disyuncion(p0, p1)

toDis :: [[(String, Bool)]] -> Proposicion
toDis list = do toDisAux list (length list -1)

```

```

-- / / Forma Normal Disyuntiva
fnd :: Proposicion -> Proposicion
fnd n = do
    let varNames = vars n
    let full_list = gen_bools varNames
    let min = getMinterms n full_list varNames (length full_list -1)
    toDis min

-- / / / bonitaAux (print) / / /

-- Paréntesis al lado que asocian si la precedencia de ese lado es menor a
la del caso actual

bonitaAux :: Proposicion -> [Char]
bonitaAux n = case n of
    (Const c) -> if c then "True" else "False"
    (Variable v) -> v

    (Negacion p) -> case p of
        (Variable _) -> "~" ++ bonitaAux p
        _ -> "~(" ++ bonitaAux p ++ ")"

-- bonitaAux p1 ++ " /\ " ++ bonitaAux p2
(Conjuncion (p1, p2)) -> case p1 of
    (Variable _) -> bonitaAux p1 ++ " /\ " ++ bonitaAux p2
    (Negacion _) -> bonitaAux p1 ++ " /\ " ++ bonitaAux p2
    (Conjuncion (_, _)) -> bonitaAux p1 ++ " /\ " ++ bonitaAux p2
    _ -> "(" ++ bonitaAux p1 ++ " /\ " ++ bonitaAux p2

-- bonitaAux p1 ++ " \/ " ++ bonitaAux p2
(Disyuncion (p1, p2)) -> case p1 of
    (Variable _) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    (Negacion _) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    (Conjuncion (_, _)) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    (Disyuncion (_, _)) -> bonitaAux p1 ++ " \/ " ++ bonitaAux p2
    _ -> "(" ++ bonitaAux p1 ++ " \/ " ++ bonitaAux p2

-- bonitaAux p1 ++ " => " ++ bonitaAux p2

```



```

vars3 = vars prop3

-- gen_bools
gb1 = gen_bools vars1
gb2 = gen_bools vars2
gb3 = gen_bools vars3

-- as_vals
aV1 = as_vals vars1 (head gb1)
aV2 = as_vals vars2 (gb2 !! 1)
aV3 = as_vals vars3 (gb3 !! 2)

-- evalProp
eval1 = evalProp prop1 aV1
eval2 = evalProp prop2 aV2
eval3 = evalProp prop3 aV3

-- taut
taut1 = taut prop1
taut2 = taut prop2
taut3 = taut prop3

-- fnd
fnd1 = fnd prop1
fnd2 = fnd prop2
fnd3 = fnd prop3

-- bonita
b1 = bonita prop1
b2 = bonita prop2
b3 = bonita prop3
b4 = bonita (Conjuncion(Disyuncion(Variable "a", Variable "b"), Variable
"c"))

```

Código facilitado por el profesor usado de referencia:

sintaxis.sml

```
(* Lenguaje de proposiciones con constantes. No tiene variables *)

(* Aqui definimos la sintaxis abstracta de nuestro pequenno
   lenguaje de proposiciones con constantes *)

datatype Proposicion =
  constante      of bool
| variable       of string
| negacion       of Proposicion
| conjuncion     of Proposicion * Proposicion
| disyuncion     of Proposicion * Proposicion
| implicacion    of Proposicion * Proposicion
| equivalencia  of Proposicion * Proposicion
;

fun imprimir prop =
  case prop of
    constante false      => "false"
  | constante true       => "true"
  | variable nombre      => nombre
  | negacion prop1       => "negacion (" ^ imprimir prop1 ^
") "
  | conjuncion (prop1, prop2) => "conjuncion (" ^ imprimir prop1 ^
", " ^ imprimir prop2 ^ ") "
  | disyuncion (prop1, prop2) => "disyuncion (" ^ imprimir prop1 ^
", " ^ imprimir prop2 ^ ") "
  | implicacion (prop1, prop2) => "implicacion (" ^ imprimir prop1 ^
", " ^ imprimir prop2 ^ ") "
  | equivalencia (prop1, prop2) => "equivalencia (" ^ imprimir prop1 ^
", " ^ imprimir prop2 ^ ") "
;

nonfix ~:
val ~: = negacion

infix 7 :&&:
val (op :&&:) = conjuncion
```

```

infix 6 :||:
val (op :||:) = disyuncion

infixr 5 :=>:
val (op :=>:) = implicacion

infix 4 :<=>:
val (op :<=>:) = equivalencia

;

val pru1 = (variable "a") :&&: (variable "b") ;
val pru2 = (variable "x") :&&: (variable "y") ;
val pru3 = pru1 :||: pru2 ;
val pru4 = pru3 :=>: pru3 ;

```

vars.sml

```

(* filter filtra una lista de acuerdo con un predicado p *)

fun filter p []          = []
|   filter p (x::xs) = if p x then x :: filter p xs else filter p xs
;

(* nub obtiene una lista sin duplicados a partir de una lista arbitraria
*)
fun nub []              = []
|   nub (x::xs) = x :: (nub (filter (fn y => x <> y) xs))
;

(* Extraer variables en una proposición *)
(* Estrategia:
    - sacar variables una a una en listas unitarias,
    - concatenar listas cuando hay conectivos lógicos,
    - eliminar duplicados (si lo hay) en la lista final *)

fun vars prop =
let

```

```

fun las_vars prop =
  case prop of
    constante _
      => []
  | variable var
      => [var]
  | negacion prop1
      => las_vars prop1
  | conjuncion (prop1, prop2)
      => let val vars1 = las_vars prop1
            and vars2 = las_vars prop2
          in vars1 @ vars2
          end
  | disyuncion (prop1, prop2)
      => let val vars1 = las_vars prop1
            and vars2 = las_vars prop2
          in vars1 @ vars2
          end
  | implicacion (prop1, prop2)
      => let val vars1 = las_vars prop1
            and vars2 = las_vars prop2
          in vars1 @ vars2
          end
  | equivalencia (prop1, prop2)
      => let val vars1 = las_vars prop1
            and vars2 = las_vars prop2
          in vars1 @ vars2
          end
in
  nub (las_vars prop) (* elimina valores repetidos *)
end
;

```

evalProp.sml

```

(* Ambientes. *)
(* Los ambientes son representados como listas de pares de objetos *)

(* Por ahora no lo implementamos como un abstype.

Podria ser asi:

```

```

type ('a,'b) Ambiente = ('a * 'b) list

Pero lo hacemos asi: *)

type Identificador = string

type 'a Ambiente = (Identificador * 'a) list

(* Las siguientes declaraciones implementan la busqueda en el ambiente.
   Cuando un identificador no esta definido en el ambiente, se levanta
   una excepcion. *)

exception NoEstaEnElDominio of Identificador

fun busca ident []
  = raise NoEstaEnElDominio ident
| busca ident ((ident',valor)::ambiente)
  = if ident = ident'
    then valor
    else busca ident ambiente

(* Evaluador de proposiciones con variables.
   Hay un caso para cada variante de proposición.
   *)

fun evalProp ambiente prop =
  case prop of
    constante valor
      => valor
  | variable var
      => busca var ambiente
  | negacion prop1
      => not (evalProp ambiente prop1)
  | conjuncion (prop1, prop2)
      => let val valor1 = evalProp ambiente prop1
            and valor2 = evalProp ambiente prop2
          in valor1 andalso valor2

```

```

        end
    | disyuncion (prop1, prop2)
        => let val valor1 = evalProp ambiente prop1
            and valor2 = evalProp ambiente prop2
            in valor1 orelse valor2
        end
    | implicacion (prop1, prop2)
        => let val valor1 = evalProp ambiente prop1
            and valor2 = evalProp ambiente prop2
            in case (valor1, valor2) of
                (true, false) => false
                | _           => true
            end
        end
    | equivalencia (prop1, prop2)
        => let val valor1 = evalProp ambiente prop1
            and valor2 = evalProp ambiente prop2
            in valor1 = valor2
            end
        end
;

```

Pruebas para validar

Por cada función realizada, se hicieron una gran cantidad de pruebas para observar si los resultados dados eran los correctos. En el caso de que esté dando un valor erróneo, analizar en donde se puede estar dando el posible error para corregirlo.

Detalles

Si se desea replicar todas las pruebas realizadas en este trabajo, literalmente se tienen que ingresar las variables en la terminal y se obtendrán los mismos resultados.