UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Licenciatura en Ciencias de la Computación

“Método de tableaux”

Lógica Computacional 2019-2

Alumnas:
•López Pérez Frida Fernanda. •No. de cuenta: 315110520
•Sánchez Rangel Joanna Lizeth. •No. de cuenta: 315060982

Marzo 24, 2019

1 .- Introducción

La lógica matemática es una forma de razonamiento, mediante la cual por medio de reglas y técnicas nos puede ayudar a determinar la validez de un argumento, mediante enunciados validos que nos ayudaran a justificar una proposición. Para esto debemos utilizar una secuencia de afirmaciones que han sido verificadas con anterioridad,de los cuales se pueden deducir o derivar nuevas afirmaciones.

La lógica en ciencias de la computación nos ayuda a ver el comportamiento de ciertos programas, puesto que el origen de los modelos matemáticos de las computadoras en gran parte residen de la lógica.

Un ejemplo de la lógica es en la *inteligencia artificial,* ya que la lógica es el fundamento de todos los métodos de representación de conocimiento y el razonamiento.

Así decimos que la lógica es lenguaje que describe sin ambigüedades. De esta forma es que la lógica puede caracterizarse de distintas formas. Tradicionalmente se ha hecho mediante alguna de las aproximaciones siguientes.

1) De manera intuitiva, ligando las ideas entre si.
2) Una interpretación semántica.
3) Un sistema formal de pruebas.

De entre todas las metodologías de demostración para un razonamiento lógico, hoy hablaremos de los *métodos de tableaux* el cual es el método que presenta los sistemas lógicos de una forma intuitiva, clara y concisa, por lo que actualmente son usados como el primer método deductivo, por otra parte en los últimos años han desarrollado diversos sistemas lógicos, originados por una multitud de aplicaciones, debido a que se adaptan perfectamente a distintas exigencias.

2 .- Lógica Proposicional

Antes de introducir el concepto de lógica proposicional, cuestionemos:
¿Que es una proposición?
Una proposición o expresión lógica es simplemente un enunciado u oración en el lenguaje natural, a las cuales les podemos asignar un valor lógico. Con un valor lógico nos referimos a 0 ó 1, verdadero o falso, según sea el caso.
Ejemplo:

Son proposiciones:

•El día esta soleado.
•Mañana es sábado.
•La puerta es azul.
•No hice la tarea.

NO son proposiciones:

•¿Que es lo que esta en la mesa?
•¡La silla se va a romper!

Es necesario para utilizar la lógica proposicional definir los conectivos lógicos, que son operadores para poder expresar una proposición de manera mas sencilla y resumida.

2.1 Conectivos u operadores lógicos

Los conectivos lógicos se definen de la siguiente manera:

Negación:

La negación (denotada por ¬) se aplica en una formula φ, siendo varp una formula, donde su correspondencia en español es: no, no es cierto que, es falso que, etc. su tabla de verdad es:

φ	¬ φ
0	1
1	0

Conjunción

La conjunción (denotada por $\wedge$) se aplica para dos formulas φ y Ψ, donde su correspondencia al español es ”y”. Su tabla de verdad es:

φ	Ψ	φ $\wedge$ Ψ
1	1	1
0	1	0
1	0	0
0	0	0

Disyunción

La disyunción (denotada por $\vee$) se aplica para dos formulas φ y Ψ, donde su correspondencia al español es ”o”. Su tabla de verdad es:

φ	Ψ	φ $\vee$ Ψ
1	1	1
0	1	1
1	0	1

0	0	0

Implicación

La implicación (denotada por →) se aplica para dos formulas φ y Ψ, donde su correspondencia al español es ”Si φ entonces Ψ , Ψ si φ, etc”. Su tabla de verdad es:

φ	Ψ	φ → Ψ
1	1	1
0	1	0
1	0	1
0	0	1

Equivalencia

La equivalencia (denotada por ↔) se aplica para dos formulas φ y Ψ , donde su correspondencia al español es ” φ si y solo si Ψ , Ψ es condición necesaria y suficiente para φ etc”. Su tabla de verdad es:

φ	Ψ	φ ↔ Ψ
1	1	1
0	1	0
1	0	0
0	0	1

2.2 Lenguaje de la lógica proposicional (PROP)

∗Símbolos o variables proposicionales (un numero infinito) :p 1 , ..., p n , ...
∗Constantes lógicas :>, ⊥.
∗Conectivos u operadores lógicos : ¬, ∧, ∨, → , ↔ .
•Todos son operadores binarios a excepción de ¬; es un operador un-ario,

ejemplo: (¬ p)

El conjunto de expresiones o formulas atómicas denotado por ATOM, esta compuesto de:

Las variables proposicionales : p 1 ...p n ...

Las constantes :>, ⊥.

Las expresiones correctas para el lenguaje de lógica proposicional, se definen de la siguiente manera:

1. Si p ∈ ATOM entonces p ∈ PROP . Es decir, toda fórmula atómica es una fórmula.
2. Si φ ∈ PROP entonces ((¬φ) ∈ PROP).
3. Si φ, Ψ ∈ PROP entonces (φ ∧ Ψ), (φ ∨ Ψ), (φ → Ψ), (φ ↔ Ψ) ∈ PROP .
4. Son todas.

Ejemplo:

•(p ↔ q) ∨ (r → s)
•(p ↔ (r ∧ s)) → (p ∨ t)
•(((p ∨ r) → s) ∧ q)

2.2.1 Equivalencias lógicas

Existen ocasiones en las que podemos tener un proposición muy extensa, que al formalizarla en el lenguaje PROP podemos hacer uso de muchos conectivos, los cuales no son nada favorables a la hora de realizar nuestro tableau, así que aplicando equivalencias podemos llegar una solución bastante pequeña en algunos casos.

Asociatividad:	(P ∧ Q) ∧ R ≡ P ∧ (Q ∧ R)
Asociatividad:	(P ∨ Q) ∨ R ≡ P ∨ (Q ∨ R)
Identidad :	P ∨ ⊥ ≡ P
Identidad :	P ∧T ≡ P

Idempotencia:	$P \vee P \equiv P$
Idempotencia:	$P \wedge P \equiv P$
Elemento nulo:	$P \vee T \equiv T$
Elemento nulo:	$P \wedge \bot \equiv \bot$
Conmutatividad:	$P \vee Q \equiv Q \vee P$
Conmutatividad:	$P \wedge Q \equiv Q \wedge P$
Tercero excluido:	$P \vee \neg P \equiv T$
Contradicción:	$P \wedge \neg P \equiv \bot$
Doble negación:	$\neg\neg P \equiv P$
Distributividad:	$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$
Distributividad:	$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
De Morgan:	$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
De Morgan:	$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
Eliminación de implicación:	$P \rightarrow Q \equiv \neg P \vee Q$
Eliminación de doble implicación:	$P \leftrightarrow Q \equiv (\neg P \vee Q) \wedge (P \vee \neg Q)$
Absorción:	$P \vee (P \wedge Q) \equiv P$
Absorción:	$P \wedge (P \vee Q) \equiv P$
Neutro:	$P \vee \bot \equiv P$
Neutro:	$P \wedge T \equiv P$

2.2.2 Interpretación de fórmulas

Cuando tenemos una PROP, nuestro objetivo es ver si es satisfacible en algún caso, definiré los conceptos a partir de la siguiente función.

Definimos el tipo de valores booleanos; que denotaremos como BOOL, se define como BOOL = 0,1

Un estado o asignación de variables proposicionales es una función:
$I : VarP \rightarrow BOOL$

Dado un estado $I : VarP \rightarrow BOOL$, definimos la interpretación de formulas con respecto a I como la función:

$I^* : PROP \rightarrow BOOL$; tal que:
- $I^*(P) = I(P)$; para $p \in VarP$; es decir $I^* \mid VarP = I$
- $I^*(T) = 1$
- $I^*(\perp) = 0$
- $I^*(\neg(\varphi)) = 1$ syss $I^*(\varphi) = 0$.
- $I^*(\varphi \wedge \Psi) = 1$ syss $I^*(\varphi) = I^*(\Psi) = 1$.
- $I^*(\varphi \vee \Psi) = 1$ syss $I^*(\varphi) = I^*(\Psi) = 0$.
- $I^*(\varphi \rightarrow \Psi) = 1$ syss $I^*(\varphi) = 1$ e $I^*(\Psi) = 0$.
- $I^*(\varphi \leftrightarrow \Psi) = 1$ syss $I^*(\varphi) = I^*(\Psi)$.

Definición :
Sea φ una formula; entonces decimos que:
- Si $I(\varphi) = 1$ para toda interpretación I, entonces φ es una tautologia o una formula valida; y lo denotamos por $\models \varphi$
- Si $I(\varphi) = 1$ para alguna interpretación I, entonces decimos que φ es satisfacible, que φ es verdadera en I, o también que I es modelo de φ; lo denotamos por $I \models \varphi$.

- Si $I(\varphi) = 0$ para alguna interpretación I, entonces decimos que φ es falsa, o que φ es insatisfacible en I, o también que I no es modelo de φ; lo denotamos por $I\varphi$.

- Si $I(\varphi) = 0$ para toda interpretación I, entonces φ es una contradicción o una formula no satisfacible.

3 .- Método de tableaux

Los tableaux nos sirven para establecer la satisfacibilidad de una fórmula (o conjunto de fórmulas) y consisten básicamente en el despliegue

sistemático de las condiciones de verdad de la fórmula (o fórmulas) en estudio.
Tienen el aspecto de árboles cuyas ramas representan las distintas posibilidades. Las ramas se cierran cuando en ellas aparecen contradicciones, entendiéndose que se trata de una posibilidad frustrada. Un árbol completamente desarrollado y con todas las ramas cerradas muestra que la fórmula (o fórmulas) es insatisfacible. Por el contrario, una rama abierta permite definir una interpretación que satisface a la fórmula (o fórmulas) del árbol.

También nos sirven los tableaux para verificar la corrección de un razonamiento puesto que una forma de demostrar que "Γ |= C" es demostrar que "Γ ∪ {¬C}" es insatisfacible, que no tiene modelo alguno. Los tableaux semánticos se pueden ver como un procedimiento sistemático de búsqueda de contra ejemplo; es decir de una interpretación que sea modelo de Γ pero que no lo sea de C. Puesto que las ramas cerradas significan posibilidades frustradas, un árbol con todas sus ramas cerradas es la demostración de que no hay contra-ejemplo y, por ende, de que "Γ |= C" (utilizando Γ |= C si y sólo si Γ ∪ {¬C} es insatisfacible). Por otra parte, puesto que un conjunto finito de fórmulas es satisfacible si y sólo si lo es la conjunción de todas ellas, lo único que tenemos que aprender es cómo resolver tableaux de una sola fórmula.

3.1 Las reglas de los tableux

Una vez definidos los tableaux podemos empezar por definir las reglas para trabajar con ellos.

Las reglas nos permiten descomponer sistemáticamente a las fórmulas obteniendo como resultado otras fórmulas mas simples. La descomposición finaliza cuando o bien se obtienen contradicciones explícitas (tales como B y ¬ B o ¬> T o ⊥) o no se pueden aplicar mas reglas pues todas las fórmulas han sido transformadas. Si las reglas llevan en todos los casos a una contradicción, A es contradictoria y concluimos que ¬A es válida. Si no, A será satisfacible y podemos extraer del propio árbol un modelo de A.

• α-reglas (α = 'y'):
1. De A ∧ B, se deduce A y B.
2. De ¬(A ∨ B) se deduce ¬A y ¬B.

3. De ¬(A → B), se deduce A y ¬B.
4. De ¬¬A, se deduce A.
• β-reglas (β = 'ramificación'):
1. De A ∨ B, se deduce A y, en una rama nueva, separada, B.
2. De ¬(A ∧ B), se deduce ¬A y, en una rama nueva, separada, ¬B.
3. De A → B, se deduce ¬A y, en una rama nueva, separada, B.
nota: Una vez que la rama se cierre porque llegamos a contradicción, no se puede seguir trabajando sobre ella.

3.1.2 Ejemplo

Empezamos con A = (¬p ∧ ¬¬q) ∧ ¬q y llegamos a una contradicción.

1. (¬p ∧ ¬¬q) ∧ ¬q

por la α-regla de ∧ en 1

2. ¬p ∧ ¬¬q

3. ¬q

por la α-regla de ∧ en 2 **4. ¬p**

5. ¬¬q

por la α-regla de ¬¬ en 5 **6. q**

Este es un tableau cerrado para A, ya que todas sus ramas están cerradas.
Nota: Podríamos haber parado en la línea 5, ya que las líneas 3 y 5 son contradictorias.

3.2 Deducción del Algoritmo

Después de llegar aquí es claro que podemos deducir un algoritmo para resolver los tableaux, que es muy intuitivo o fácil de ver pero de importancia hacer notar.

Vamos a localizar los principales conectivos de la fórmula y de cada subformula y creamos el árbol
de la siguiente fórmula :
•La fórmula original aparece como la raíz del árbol.
•si el conectivo es una disyunción, de la raíz abren dos ramas una para el lado derecho de la fórmula y la otra para el otro lado, y si es una conjunción
se coloca uno debajo del otro.
•Se aplica de manera recursiva el algoritmo para cada subárbol.

3.3 Clasificación de las fórmulas a partir de los tableaux

Como ya sabemos los tableaux lógicos sirven en principio para determinar si una fórmula es o no satisfacible pero que mas…

Contradicción: Se hace el tableau de C y se comprueba que todas sus ramas están cerradas.

Satisfacible: Se hace el tableau de C y se comprueba que al menos una rama está abierta y completamente desarrollada.

Insatisfacible: Se hace el tableau de C y se comprueba que todas las ramas estén cerradas, por lo tanto sea un tableau cerrado.

Tautología: Se hace el tableau de ¬C y se comprueba que todas sus ramas están cerradas.

4.- Justificación de la estructura usada en la implementación

La estructura utilizada es listas, por lo cuál aplicando las reglas alfa y beta sólo tendremos conjunciones y disyunciones por ejemplo, el tableaux de una fórmula se verá así:

tableau (Disy(FA(Var "p"))(Neg(FA(Var "p"))))
[[[p]],[[no p]]]

Las funciones principales son :

Función vars: Recibe una proposición p y regresa una lista de variables atómicas. Obtiene la lista de variables que figuran en una fórmula proposicional.

```
vars :: PROP -> [ATOM]
vars p = eliminaRepetidos (varsAux p)
```

Función busca: Recibe una fórmula atómica y una lista de fórmulas atómicas. La función busca el una variable en la lista y si la encuentra regresa True, en caso contrario regresa False.

```
busca :: ATOM -> [ATOM] -> Bool
busca _ [] = False
busca (Var v) ((Var x):xs) = v == x || busca (Var v) xs
```

Función interp : Recibe un proposición, una lista de estados y regresa un booleano. Dado un estado, la función interp realiza la interpretación de fórmulas proposicionales.

La interpretación de una fórmula atómica, donde ésta es un valor Booleano será:

- Si el booleano es True, la interpretación será True sin importar que le pases como estado
- Si el booleano es False, la interpretación será False sin importar que le pases como estado.

La interpretación de una una fórmula atómica, donde ésta es es una Variable v, con el estado xs. Se busca la variable v en el estado.

La interpretación de la negación de una proposición será la interpretación de p bajo el estado. Obteniendo la interpretación de p, se buscará la interpretación contraria con la función not del preludio de Haskell, ya que inicialmente se busca la negación.

La interpretación de la conjunción de dos proposiciones p y q, sérá verdad si el valor de p es verdad y el valor de q es verdad, es decir p y q deberán ser ambas forzosamente verdaderas.Para verificar esto se utiliza el operador && de Haskell.Después se hace recursión sobre las proposiciones p y q.

La interpretación de la disyunción de dos proposiciones p y q, sérá verdad si el valor de p es verdad o el valor de q es verdad. Es decir, sólo nos importa que alguna de las dos sea verdadera. Para verificar esto se utiliza el operador || de Haskell.Después se hace recursión sobre las proposiciones p y q.

La interpretación de la implicación se hará bajo la equivalencia lógica:

p -> q = ¬ p ∨ q

Por lo tanto, se hace recursión sobre la interpretación de no p (usando la función not de Haskell) y la recursión de la interpretación de q.

La interpretación de la equivalencia es verdad si la interpretación de p es igual a la interpretación de q.

```
interp :: PROP -> [Estado] -> Bool
interp (FA (Cte c)) _ = c
```

```
interp (FA (Var v)) xs = busca (Var v) xs
interp (Neg p) xs = not (interp p xs)
interp (Conj p q) xs = (interp p xs) && (interp q xs)
interp (Disy p q) xs = (interp p xs) || (interp q xs)
interp (Impl p q) xs = not (interp p xs) || (interp q xs)
interp (Syss p q) xs = (interp p xs) == (interp q xs)
```

Función subconjuntos: Recibe una lista y obtiene las sublistas de la lista pasada como parámetro.

```
subconjuntos :: [a] -> [[a]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = [x:ys | ys <- xss] ++ xss
        where xss = subconjuntos xs
```

Función estados: Recibe una Proposición p y regresa una lista de listas de estados. Obtiene los posibles estados de las variables de una fórmula proposicional.

```
estados :: PROP -> [[Estado]]
estados p = subconjuntos (vars p)
```

Función esTautología: Recibe una proposición y regresa True si la proposición es una Tautología, en caso contrario regresa False. Por definición, la fórmula f es tautología si se evaluan verdadero en todos los estados posibles.

```
esTautologia :: PROP -> Bool
esTautologia f = null (filter (\y -> (y == False)) (map (\x -> interp f x) (estados f)))
```

Función es Satisfacible: Recibe una proposición y regresa True si la proposición f es satisfacible, en caso contrario regresa False. Por definición, la proposición f es satisfacible si existe un modelo que haga verdadera a f.

```
esSatisfacible :: PROP -> Bool
esSatisfacible f = not (esContradiccion f)
```

Función esContradicción: Recibe una proposición y regresa True si la proposición f es contradicción, en caso contrario regresa True. Por definición, la proposición f es contradicción si se evalúan a falso en todos los estados.

```
esContradiccion :: PROP -> Bool
esContradiccion f = null (filter (\y -> (y == True)) (map (\x -> interp f x) (estados f)))
```

Función eliminaRepetidos: Recibe una lista y regresa la lista sin elementos repetidos. Es decir, deja la primera aparición del elemento que encuentra.

```
eliminaRepetidos :: Eq a => [a] -> [a]
eliminaRepetidos [] = []
eliminaRepetidos (x:xs) = x:(eliminaRepetidos (filter (\y->y/=x) xs))
```

Función iN : Recibe una proposición y regresa una proposición donde introduce la negación.

```
iN :: PROP -> PROP
iN (FA a) = FA a
iN (Neg p) = iNA p
iN (Conj p q) = Conj (iN p) (iN q)
iN (Disy p q) = Disy (iN p) (iN q)
iN p = p
```

Función fnn: Recibe una proposición y regresa la forma normal negativa de la proposición. Por definición, la forma normal negativa de una proposición es proposición que no tenga implicaciones y equivalencias. Además que la negación sólo afecta a variables atómicas.

```
fnn :: PROP -> PROP
```

```
fnn p = iN (eI (eE p))
```

Función eE: Recibe una proposición y regresa una proposición sin equivalencias.

```
p <-> q = (p -> q) ∧ (q -> p)

eE :: PROP -> PROP
eE (FA a) = (FA a)
eE (Neg p) = Neg(eE p)
eE (Conj p q) = Conj (eE p) (eE q)
eE (Disy p q) = Disy (eE p) (eE q)
eE (Impl p q) = Impl (eE p) (eE q)
eE (Syss p q) = Conj (Impl (eE p) (eE q)) (Impl (eE q) (eE p))
```

Función eI: Recibe una proposición y regresa una proposición sin implicaciones.

```
p -> q = ¬ p ∨ q

eI :: PROP -> PROP
eI (FA a) = (FA a)
eI (Neg p) = Neg (eI p)
eI (Conj p q) = Conj (eI p) (eI q)
eI (Disy p q) = Disy (eI p) (eI q)
eI (Impl p q) = Disy (Neg (eI p)) (eI q)
eI (Syss p q) = Conj (eI (Impl p q)) (eI (Impl q p))
```

Función fnc: Recibe una proposición y regresa la forma Normal conjuntiva de la proposición. Por definición, una proposición está en forma normal conjuntiva si y sólo si está de la forma:

C1 ∧ C2 ∧ C3 ∧ ... ∧ Ci donde 1 <= i , y Ci es una cláusula.

```
fnc :: PROP -> PROP
fnc p = fnca (fnn p)
```

Función: dD: Recibe dos proposiciones y lo que hace es regresar uan proposición con las disyunciones distribuídas.

```
dD :: PROP -> PROP -> PROP
dD (Conj p q) r = Conj (dD q r) (dD p r)
dD r (Conj p q) = Conj (dD r p) (dD r q)
dD p q = Disy p q
```

Función alfaregla: Recibe una lista de list de Proposiciones y regresa una lista de proposiciones aplicando alfa regla. Se siguen las siguientes reglas :

- De A ∧ B se dedce A y B
- De ¬(A ∨ B) se deduce ¬A y ¬B
- De ¬(A -> B) se deduce A y ¬B

```
alfaregla :: [[PROP]] -> [[[PROP]]]
alfaregla [] = []
alfaregla [[]] = []
alfaregla [(x:xs)] = [alfaAux (x:xs)]
alfaregla ((x:xs):ys) = [alfaAux (x:xs)] ++ (alfaregla ys)
```

Función betaregla: Recibe una proposición y regresa una lista de proposiciones aplicando beta regla. Se siguen las siguientes reglas :

- De A ∨ B se dedce A y, en una lista distinta B
- De ¬(A ∧ B) se deduce ¬A, y en una lista distinta ¬B
- De (A -> B) se deduce ¬A, y en una lista distinta ¬B

```
betaregla :: [[PROP]] -> [[[PROP]]]
betaregla [] = []
betaregla [[]] = []
betaregla [(x:xs)] = betaAux (x:xs)
betaregla ((x:xs):ys) = betaAux (x:xs) ++ (betaregla ys)
```

Función tableau: Recibe una proposición y dibuja su Tableau

```
tableau :: PROP -> [[[PROP]]]
tableau (FA a) = [[[FA a]]]
tableau a = betareglaAux (betareglaAux (betareglaAux [[[fnc a]]]))
```

Para probar el proyecto, se hace lo siguiente

```
*Proyecto1> esTautologia  (Disy(FA(Var "p"))(Neg(FA(Var "p"))))
True

*Proyecto1> esSatisfacible (Conj(Neg(FA(Var "p")))(FA(Var "p")))
False

*Proyecto1> tableau (Disy(FA(Var "p"))(Neg(FA(Var "p"))))
  [[[p]],[[no p]]]
```

Funciones auxiliares:

Función varsAux: Recibe una proposición p y regresa una lista de variables atómicas. En general, la función obtiene la lista de variables que figuran en una fórmula proposicional.

```
varsAux :: PROP -> [ ATOM ]
varsAux (FA (Cte b)) = []
varsAux (FA p) = [p]
varsAux (Neg x) = varsAux x
varsAux (Conj x y) = (varsAux x) ++ (varsAux y)
varsAux (Disy x y) = (varsAux x) ++ (varsAux y)
varsAux (Impl x y) = (varsAux x) ++ (varsAux y)
varsAux (Syss x y) = (varsAux x) ++ (varsAux y)
```

Función iNA: Recibe una proposición e introduce las negaciones a la proposición.

```
iNA :: PROP -> PROP
iNA (FA a) = Neg (FA a)
iNA (Neg p) = iN p
iNA (Conj p q) = Disy (iNA p) (iNA q)
iNA (Disy p q) = Conj (iNA p) (iNA q)
iNA p = p
```

Función fnca: Recibe una proposición y regresa su forma normal conjuntiva.

```
fnca :: PROP -> PROP
fnca (FA p) = (FA p)
fnca (Disy p q) = dD p q
fnca p = p
```

Función alfaAux: Recibe una lista de proposiciones y aplica las reglas para alfa regla.

```
alfaAux :: [PROP] -> [[PROP]]
alfaAux [] = []
alfaAux ((FA a):xs) = [[FA a]] ++ alfaAux xs
alfaAux ((Neg a):xs) = [[Neg a]] ++ alfaAux xs
alfaAux ((Conj a b):xs) = alfaAux [a] ++ alfaAux [b] ++ alfaAux
xs
alfaAux ((Disy a b):xs) = [[Disy a b]] ++ alfaAux xs
```

Función betaAux: Recibe una lista de proposiciones y aplica las reglas para beta regla.

```
betaAux :: [PROP] -> [[[PROP]]]
betaAux [] = []
betaAux (x:xs) = betaAux2 x ++ betaAux xs

betaAux2 :: PROP -> [[[PROP]]]
betaAux2 (FA a) = [[[FA a]]]
betaAux2 (Neg a) = [[[Neg a]]]
betaAux2 (Conj a b) = [alfaAux [Conj a b]]
betaAux2 (Disy a b) = betaAux2 a ++ betaAux2 b

betareglaAux :: [[[PROP]]] -> [[[PROP]]]
betareglaAux [] = []
betareglaAux [[]] = []
betareglaAux [[[]]] = []
betareglaAux (x:xs) = betaregla x ++ betareglaAux xs
```

5.- Demostración de teoremas

1. *Teorema (correctud):* ***Si hay un tableau cerrado para Γ entonces Γ no tiene un modelo.***
   supongamos un Tableau cerrado en Γ llamado T, entonces todas las ramas son cerradas por lo que sabemos que no podemos exhibir un modelo ≡ a que Γ es un conjunto insatisfacible por lo tanto sabemos que Γ no tiene un modelo.

2. *Teorema (completud):* ***Si Γ no tiene un modelo entonces existe un tableau T(Γ) cerrado.***
   Si Γ no tiene un modelo, Γ es insatisfacible entonces sea T un tableau que representa a Γ . como Γ es insatisfacible T es cerrado.

6.- Bibliografía

(1)W. Hodges, Logic Pelican (Penguin), 1997.
(2)Miranda Perea, F.E., AND Viso Gurovich, E. Matemáticas discretas. La prensas de las ciencias, Ciudad de México, México, 2016.
(3) Hunt, M., AND Ryan, M. Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, New York, NY, USA, 2004.
(4) Miranda Perea, Favio E., Notas para el curso: Lógica Computacional, UNAM, 2017.