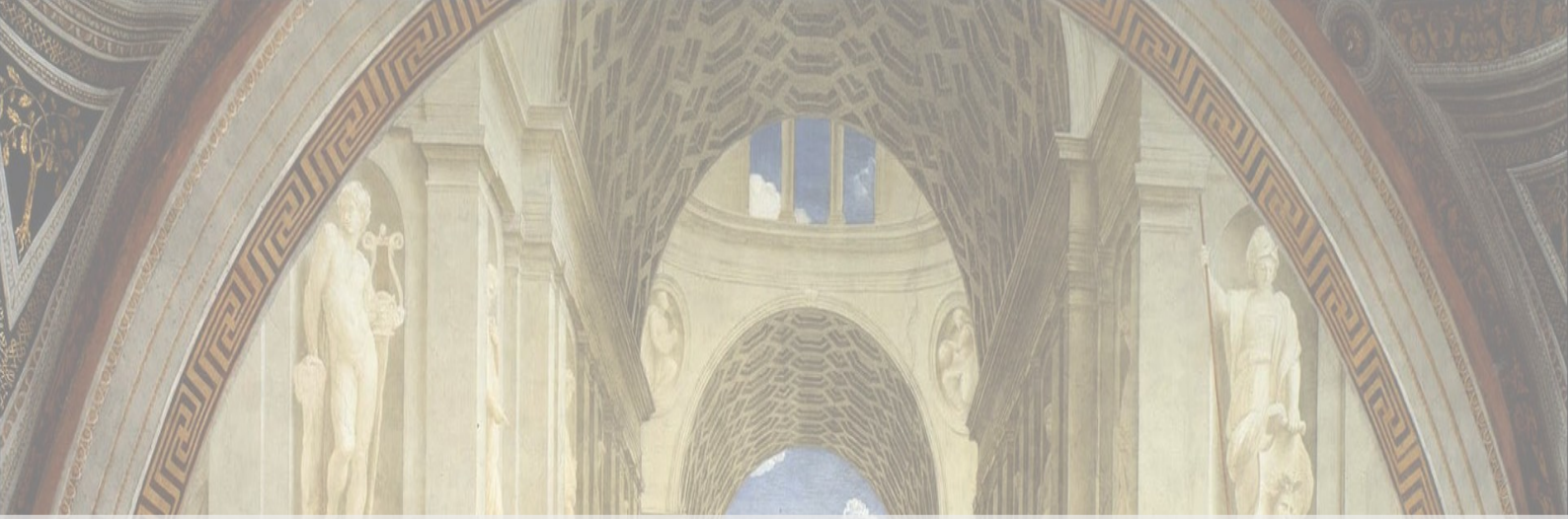

Desarrollo de la Lógica Proposicional y de Primer Orden bajo el paradigma funcional y la orientación Web.

Proyecto de Alumnía Interna 2019/20

Autor:
Ramos González, Víctor

Tutor:
Sancho Caparrini, Fernando
Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE SEVILLA

Desarrollo de la Lógica Proposicional y de Primer Orden bajo el paradigma funcional y la orientación Web.

Víctor Ramos González

Tutor: Fernando Sancho Caparrini
Ciencias de la Computación e Inteligencia Artificial
E.T.S. Ingeniería Informática
Universidad de Sevilla

Septiembre 2020

Resumen

El proyecto aborda los conceptos y algoritmos básicos de la Lógica Proposicional y la Lógica de primer, desde un punto de vista implementativo a través de un lenguaje encuadrado en el paradigma funcional (Elm).

El proyecto, basado en la asignatura de Lógica Informática, busca una doble finalidad, por un lado servir como una somera introducción a la programación declarativa al mismo que tiempo que proporcionar al alumnado herramientas intuitivas y de sencillo uso en la realización de los ejercicios que apoyen los contenidos teóricos que se desarrollan en dicha asignatura.

Índice general

1. Introducción. Objetivos y organización del proyecto **3**

1.1. Introducción 4

1.2. Objetivos del proyecto 4

1.3. Estructura del proyecto 4

1.3.1. Módulos funcionales 4

1.3.2. Interfaz gráfica 4

2. LP I. Sintaxis y Semántica **5**

2.1. Introducción. Descripción general del capítulo 6

2.2. Módulo SintaxSemanticsLP 7

2.2.1. Aspectos Sintácticos 7

2.2.2. Aspectos Semánticos 8

2.2.3. Código del módulo y resumen de funciones 11

2.3. LP Parser I. Módulo LP_Parser 15

2.3.1. Código del módulo y resumen de funciones 15

2.4. LP Parser II. Módulo LPBig_Parser 18

2.4.1. Módulo A_Expressions (Expr. Aritméticas) 19

2.4.2. Módulo B_Expressions (Expr. Booleanas) 19

2.4.3. Módulo LPBig_Parser 19

2.5. Módulo LP_toString 19

3. LP II. Tableros Semánticos **20**

4. LP III. Formas Normales y DPLL **21**

5. LP IV. Sistemas Deductivos **22**

6. LP V. Algoritmo de Resolución **23**

7. LP VI. Modelado y Resolución de PSR a través de la LP **24**

8. LPO I. Sintaxis y Semántica **25**

9. LPO II. Tableros Semánticos **26**

10.LPO III. Forma Prenex, Skolem y Teorema de Herbrand **27**

11.LPO V. Algoritmos de Unificación y Resolución **28**

12.LPO VI. Modelado y Resolución de PSR a través de la Lógica de Primer Orden **29**

Capítulo 1

Introducción. Objetivos y organización del proyecto

1.1. Introducción **4**
1.2. Objetivos del proyecto **4**
1.3. Estructura del proyecto **4**
1.3.1. Módulos funcionales 4
1.3.2. Interfaz gráfica 4

1.1. Introducción

Tras haber cursado la asignatura de Lógica Informática con el profesor D. Fernando Sancho Caparrini, director de este proyecto, se despertó mi gusto por la Teoría de la Lógica Matemática y Computacional. Aunque desde el punto de vista teórico la asignatura presentaba los contenidos de forma completa y con total formalismo y corrección, los profesores encargados de la misma pretendían darle a la misma un enfoque más práctico de manera que estos nuevos contenidos sirvieran como complemento a los conceptos formales presentados teóricamente.

1.2. Objetivos del proyecto

El proyecto persigue un doble objetivo,

1.3. Estructura del proyecto

El proyecto se estructará en 2 partes fundamentales:

1.3.1. Módulos funcionales

Se implementan distintos módulos, que se describen detalladamente a lo largo del proceso, en el que se desarrollarán las estructuras y métodos necesarios para abordar los algoritmos vistos en la asignatura.

1.3.2. Interfaz gráfica

Capítulo 2

LP I. Sintaxis y Semántica

2.1. Introducción. Descripción general del capítulo 6
2.2. Módulo SintaxSemanticsLP 7
2.2.1. Aspectos Sintácticos 7
2.2.2. Aspectos Semánticos 8
2.2.3. Código del módulo y resumen de funciones 11
2.3. LP Parser I. Módulo LP_Parser 15
2.3.1. Código del módulo y resumen de funciones 15
2.4. LP Parser II. Módulo LPBig_Parser 18
2.4.1. Módulo A_Expressions (Expr. Aritméticas) 19
2.4.2. Módulo B_Expressions (Expr. Booleanas) 19
2.4.3. Módulo LPBig_Parser 19
2.5. Módulo LP_toString 19

2.1. Introducción. Descripción general del capítulo

TO DO

2.2. Módulo SintaxSemanticsLP

En este primer capítulo vamos a estudiar, desde un punto de vista práctico, a través de las implementaciones, los distintos elementos que conforman la Lógica Proposicional, esto es la Síntaxis y la Semántica que se han presentado en los fundamentos teóricos.

2.2.1. Aspectos Sintácticos

Átomo y Fórmula Proposicional

Recordemos que las expresiones están formadas por símbolos proposicionales, organizados en átomos (expresiones básicas) y relaciones entre ellas (conectivas), de acuerdo a esto, podemos definir un símbolo proposicional como una cadena de texto:

```
type alias PSymb = String
```

Listing 2.1: Definición de Símbolo Proposicional como alias de String.

Y las conectivas lógicas como categorías del tipo recursivo *Prop*, de forma que:

```
type Prop = Atom PSymb
          | Neg Prop
          | Conj Prop Prop
          | Disj Prop Prop
          | Impl Prop Prop
          | Equi Prop Prop
```

Listing 2.2: Definición del tipo Prop (Proposición).

Esto nos permite definir todas las fórmulas proposicionales que se pueden formar, algunos ejemplos:

(a) $(p \wedge q) \vee (p \wedge r)$ (b) $(p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q$ (c) $(p \leftrightarrow q) \wedge (p \rightarrow \neg q) \wedge p$

```
a = Disj (Conj (Atom "p") (Atom "q")) (Conj (Atom "p") (Atom "r"))

b = Impl
        (Disj
            (Conj (Atom "p") (Atom "r"))
            (Conj (Neg (Atom "p")) (Atom "q"))
        )
        (Neg (Atom "q"))

c = Conj
        (Conj
            (Equi (Atom "p") (Atom "q"))
            (Impl (Atom "p") (Neg (Atom "q")))
        (Atom "p")
```

Listing 2.3: Ejemplos de definición de fórmulas proposicionales.

Como se puede apreciar, escribir las fórmulas de esta forma puede resultar una tarea ardua y propensa a errores, por eso, como mostraremos más adelante, se ha desarrollado un parser que nos permite escribir de forma más cómoda, sintética y visual las fórmulas. Aunque aquí no influyen las reglas de asociación vistas en los fundamentos teóricos, ya que dicho orden viene dado por el propio orden de aplicación, sí hemos de tenerlas en cuenta a la hora de realizar el parser.

Conjuntos de fórmulas

Definidas las fórmulas proposicionales, definiremos los conjuntos de fórmulas como listas de fórmulas proposicionales de manera que una misma fórmula puede aperecer varias veces en el conjunto. De esta forma:

```
type alias PropSet = List Prop
```

Listing 2.4: Definición de Conjunto de Fórmulas como Lista de Fórmulas Proposicionales

2.2.2. Aspectos Semánticos

Interpretaciones y Valor de verdad de fórmulas proposicionales.

Una vez provista la sintaxis, pasamos a desarrollar la semántica de la Lógica Proposicional. Aunque en secciones futuras mostraremos la implementación del parser, por ahora vasta conocer la estructura de las fórmulas para hacer abordable la implementación que se ha llevado a cabo de la semática del lenguaje de la lógica de proposiciones.

Como ya se ha comentado, y explicado en los fundamentos teóricos, hemos de abordar la interpretación de las fórmulas, y para ello hemos de crear las estructuras necesarias que nos permitan representar interpretaciones, esto es, definir los símbolos proposicionales con valor de verdad 1 y aquellos con valor 0. Para ello, hemos elegido una representación "dispersa"de manera que una interpretación corresponde a una lista de símbolos proposicionales que son los que son considerados verdaderos, los términos que no aparecen en la lista serán considerados como falsos.

```
type alias Interpretation = List PSymb
```

Listing 2.5: Definición de Interpretación como Lista de Símbolos Proposicionales

De esta forma podemos definir, de forma sencilla la evaluación de las fórmulas, de forma que el valor de verdad de un símbolo se reduce a la pertenencia del mismo a la lista de interpretación, de esta forma, podemos, dada la estructura recursiva definida para las fórmulas proposicionales, establecer una función, tambieén recursiva, que nos permite evaluar las mismas:

- (*Caso base*) Una fórmula atómica será verdadera si y sólo si el símbolo proposicional pertenece a la lista de interpretación.
- (Casos recursivos) Según la clase de fórmula:
  - La negación de una fórmula será verdadera respecto de una interpretación si y sólo si la evaluación de la fórmula es falsa.
  - La conjunción de dos fórmulas proposicionales será verdadera respecto de una interpretación si y sólo si la evaluación de ambas fórmulas respecto de dicha evaluación es verdadera.
  - La disyunción de dos fórmulas proposicionales será verdadera será verdadera respecto de una interpretación si y sólo si alguna de las evaluaciones de las dos fórmulas es evaluada verdadera respecto de dicha interpretación.
  - La implicación será verdadera respecto de una interpretación si y sólo si o la evaluación del antecedente es evaluado falso respecto de dicha interpretación o el consecuente es evaluado verdadero respecto de la misma.
  - La equivalencia será verdadera respecto de una interpretación si y sólo si la evaluación del antecedente coincide con la evaluación del consecuente.

```
valuation : Prop -> Interpretation -> Bool
valuation pr i =
    case pr of
        Atom p -> List.member p i
        Neg p -> not (valuation p i)
        Conj p q -> valuation p i &&  valuation q i
        Disj p q ->   valuation p i ||  valuation q i
        Impl p q ->   not (valuation p i) ||  valuation q i
        Equi p q ->   valuation (Impl p q) i &&  valuation (Impl q p) i
```

Listing 2.6: Función de evaluación de las fórmulas proposicionales

Modelos (tablas de verdad, satisfactibilidad y validez lógica)

Desde el punto de vista teórico las tablas de verdad corresponden a estructuras que reflejan el valor de verdad de una fórmula proposicional respecto de cada una de las posibles interpretaciones posibles para la fórmula. Entonces, para poder construir la tabla de verdad, primero hemos de calcular todas las interpretaciones posibles, que, dada la definición que hemos proporcionado para las interpretaciones, correspondería a todos los conjuntos posibles (*powerset*) que podríamos construir con los símbolos proposicionales que aparecen en la fórmula. Así:

```
symbInProp : Prop -> Set PSymb

symbInProp f=
    case f of
        Atom p -> Set.singleton p
        Neg p -> symbInProp p
        Conj p q -> Set.union (symbInProp p ) (symbInProp q)
        Disj p q -> Set.union (symbInProp p ) (symbInProp q)
        Impl p q -> Set.union (symbInProp p ) (symbInProp q)
        Equi p q -> Set.union (symbInProp p ) (symbInProp q)
```

Listing 2.7: Función para extraer los símbolos proposicionales que intervienen en una fórmula

```
allInterpretations : Prop -> List Interpretation
allInterpretations x =  Aux.powerset <| List.sort <| Set.toList
                                                    <| symbInProp x
```

Listing 2.8: Función para extraer las posibles interpretaciones para una fórmula propsicional

De esta forma podemos expresar la tabla de verdad como una lista de tuplas en las que el primer elemento corresponde a la interpretación y el segundo corresponde a la evaluación de la fórmula respecto de dicha valoración:

```
truthTable : Prop -> List (Interpretation, Bool)
truthTable x = List.map (\xs ->  (xs,valuation x xs)) <| allInterpretations x
```

Listing 2.9: Función para la construcción de la tabla de verdad de una fórmula

Una vez estudiado lo anterior, los modelos corresponden a las interpretaciones que son evaluadas verdaderas, esto es, de las posibles interpretaciones aquellas hacen la fórmula verdadera. Aquellas interpretaciones que hacen la fórmula falasa se denominan contramodelos. Así:

```
models : Prop -> List Interpretation
models x = List.filter (\y -> valuation x y) (allInterpretations x)

countermodels : Prop -> List Interpretation
countermodels x = List.filter (\y -> not (valuation x y))
                                              (allInterpretations x)
```

Listing 2.10: Función para el cálculo de los modelos de una fórmula proposicional

Definidos los modelos, podemos así mismo definir (funcionalmente) los conceptos de satisfactibilidad y validez, de forma que:

- Una fórmula es satisfactible si posee al menos un modelo.
- Una fórmula es lógicamente válida o tautología si toda interpretación es modelo de la fórmula.
- Una fórmula es insatisfactible o contradicción si no posee ningún modelo.

De esta forma:

```
satisfactibility : Prop -> Bool
satisfactibility x = List.any (\xs-> valuation x xs) (allInterpretations x)

validity : Prop -> Bool
validity x = List.all (\xs-> valuation x xs) (allInterpretations x)

insatisfactibility : Prop -> Bool
insatisfactibility x = not (satisfactibility x)
```

Listing 2.11: Funciones de Satisfactibilidad, Validez e Insatisfactibilidad

Conjuntos de Fórmulas. Modelos, Consistencia, Validez y Consecuencia Lógica

Vista la satisfactibilidad, modelos, etc. aplicadas a una fórmula, pasamos a desarrollar los métodos necesarios para el estudio de la satisfactibilidad (consistencia) , inconsistencia en conjuntos de fórmulas. Ahora los modelos del conjunto de fórmulas corresponden a las interpretaciones tales que hacen verdaderas todas las fórmulas del conjunto. Para obtener los modelos hemos, al igual que en el caso de las fórmulas, obtener el conjunto de símbolos proposicionales y a partir de estos el conjunto de todas las posibles interpretaciones. De forma que:

```
setSymbols : List Prop -> Set PSymb
setSymbols xs =
    List.foldr (\x acc -> Set.union acc (symbInProp x)) Set.empty xs

allSetInterpretations : List Prop -> List Interpretation
allSetInterpretations xs = Aux.powerset <| Set.toList <| setSymbols xs

isSetModel : List Prop -> Interpretation -> Bool
isSetModel xs i =  List.all (\x -> valuation x i) xs

allSetModels : List Prop -> List Interpretation
allSetModels xs = List.filter (isSetModel xs) (allSetInterpretations xs)

allSetCounterModels : List Prop -> List Interpretation
allSetCounterModels xs =
    List.filter (\x -> not(isSetModel xs x)) <| allSetInterpretations xs
```

Listing 2.12: Modelos y contramodelos en conjuntos de fórmulas proposicionales

De forma que ahora, es sencillo, comprobar la consistencia de un conjunto a partir de la definición: *'Un conjunto es consistente si posee, al menos, un modelo. En caso contrario es inconsistente'*

```
isConsistent : List Prop -> Bool
isConsistent xs =
    List.any (\x -> isSetModel xs x) <| allSetInterpretations xs

isInconsistent: List Prop -> Bool
isInconsistent xs = not(isConsistent xs)
```

Listing 2.13: Consistencia e Inconsistencia en Conjuntos Proposicionales

Por último nos queda definir el concepto de consecuencia lógica. Acudiendo a la definición: *'Una fórmula es consecuencia lógica de un conjunto de fórmulas si y sólo si todo modelo del conjunto es también modelo de la fórmula'*, pero también *'Una fórmula es consecuencia lógica de un conjunto de fórmulas si la unión del conjunto y el conjunto formado por la negación del la fórmula es inconsistente.'*. De esta forma podemos plantear dos desarrollos alternativos:

```
isConsecuence : List Prop -> Prop -> Bool
isConsecuence xs x = List.all (\y -> valuation x y) <| allSetModels xs

isConsecuence : List Prop -> Prop -> Bool
isConsecuence xs x = isInconsistent (xs ++ [Neg x])
```

Listing 2.14: Consecuencia Lógica

2.2.3. Código del módulo y resumen de funciones

Código del módulo

```
module Modules.SintaxSemanticsLP exposing (
    PSymb, Prop, Interpretation,PropSet,
    valuation, truthTable, models, countermodels, satisfactibility,
    validity, insatisfactibility, isSetModel, allSetModels,
    allSetCounterModels, isConsistent, isInconsistent, isConsecuence)

import List
import Set
import Modules.AuxiliarFunctions as Aux


-----------
-- TYPES --
-----------
type alias PSymb = String

type Prop = Atom PSymb
          | Neg Prop
          | Conj Prop Prop
          | Disj Prop Prop
          | Impl Prop Prop
          | Equi Prop Prop

type alias Interpretation = List PSymb
type alias PropSet = List Prop
```

```
-------------
-- METHODS --
-------------

valuation : Prop -> Interpretation -> Bool
valuation pr i =
    case pr of
        Atom p -> List.member p i
        Neg p -> not (valuation p i)
        Conj p q -> valuation p i &&  valuation q i
        Disj p q ->   valuation p i ||  valuation q i
        Impl p q ->   not (valuation p i) ||  valuation q i
        Equi p q ->   valuation (Impl p q) i &&  valuation (Impl q p) i

symbInProp : Prop -> Set.Set PSymb

symbInProp f=
    case f of
        Atom p -> Set.singleton p
        Neg p -> symbInProp p
        Conj p q -> Set.union (symbInProp p ) (symbInProp q)
        Disj p q -> Set.union (symbInProp p ) (symbInProp q)
        Impl p q -> Set.union (symbInProp p ) (symbInProp q)
        Equi p q -> Set.union (symbInProp p ) (symbInProp q)

allInterpretations : Prop -> List Interpretation
allInterpretations x =
    Aux.powerset <| List.sort <| Set.toList <| symbInProp x

truthTable : Prop -> List (Interpretation, Bool)
truthTable x = List.map (\xs ->  (xs,valuation x xs)) <| allInterpretations x

models : Prop -> List Interpretation
models x = List.filter (\y -> valuation x y) (allInterpretations x)

countermodels : Prop -> List Interpretation
countermodels x =
    List.filter (\y -> not(valuation x y)) (allInterpretations x)

satisfactibility : Prop -> Bool
satisfactibility x = List.any (\xs-> valuation x xs) (allInterpretations x)
validity : Prop -> Bool
validity x = models x== allInterpretations x
insatisfactibility : Prop -> Bool
insatisfactibility x = List.isEmpty (models x)

setSymbols : List Prop -> Set.Set PSymb
setSymbols xs =
    List.foldr (\x acc -> Set.union acc (symbInProp x)) Set.empty xs

allSetInterpretations : List Prop -> List Interpretation
allSetInterpretations xs = Aux.powerset <| Set.toList <| setSymbols xs

isSetModel : List Prop -> Interpretation -> Bool
isSetModel xs i =  List.all (\x -> valuation x i) xs

allSetModels : List Prop -> List Interpretation
allSetModels xs = List.filter (isSetModel xs) (allSetInterpretations xs)
```

```
allSetCounterModels : List Prop -> List Interpretation
allSetCounterModels xs =
    List.filter (\x -> not(isSetModel xs x)) <| allSetInterpretations xs

isConsistent : List Prop -> Bool
isConsistent xs =
    List.any (\x -> isSetModel xs x) <| allSetInterpretations xs

isInconsistent: List Prop -> Bool
isInconsistent xs = not(isConsistent xs)

isConsecuence : List Prop -> Prop -> Bool
--isConsecuence xs x = List.all (\y -> valuation x y) <| allSetModels xs
isConsecuence xs x = isInconsistent (xs ++ [Neg x])
```

Listing 2.15: Módulo SintaxSemanticsLP

Funciones disponibles

Tipo	Descripción
PSymb	Alias de *String*. Representa los símbolos proposicionales.
Prop	Representa a las proposiciones o fórmulas proposicionales. Posee varios constructores según el tipo de fórmula proposicional: (*Atom, Neg, Conj, Disj, Impl, Equi*).
Interpretation	Alias de *List PSymb*. Representa una interpretación, de forma que se consideran verdaderos los símbolos proposicionales que aparecen en la lista, y falsos aquellos que no aparecen.
PropSet	Alias de *List Prop*. Representa conjuntos de fórmulas.

Tabla 2.1: Módulo SintaxSemanticsLP I. Tipos

Método	Descripción
valuation	*valuation: Prop ->Interpretation ->Prop* Calcula el valor de verdad de una proposición según la interpretación dada.
truthTable	*truthTable: Prop ->List (Interpretation, Bool)* Calcula la tabla de verdad asociada a una fórmula proposicional, devolviéndola como una lista de pares (Interpretación, Valoración).
models	*models: Prop ->List Interpretation* Calcula los modelos de una fórmula proposicional.
countermodels	*countermodels: Prop ->List Interpretation* Calcula los contramodelos de una fórmula proposicional.
satisfactibility	*satisfactibility: Prop ->Bool* Decide si una fórmula proposicional es satisfactible o no.
validity	*validity: Prop ->Bool* Decide si una fórmula proposicional es tautología o no.
insatisfactibility	*insatisfactibility: Prop ->Bool* Decide si una fórmula proposicional es insatisfactible o no.
isSetModel	*isSetModel: List Prop ->Interpretation ->Bool* Decide si una interpretación es modelo de un conjunto de fórmulas proposicionales o no.
allSetModels	*allSetModels: List Prop ->List Interpretation* Calcula los modelos asociados a un conjunto de fórmulas proposicionales.
allSetCounterModels	*allSetCounterModels: List Prop ->List Interpretation* Calcula los contramodelos asociados a un conjunto de fórmulas proposicionales.
isConsistent	*isConsistent: List Prop ->Bool* Decide si un conjunto de fórmulas proposicionales es consistente o no.
isInconsistent	*isInconsistent: List Prop ->Bool* Decide si un conjunto de fórmulas proposicionales es inconsistente o no.
isConsecuence	*isConsecuence: List Prop ->Prop ->Bool* Decide si una fórmula es consecuencia lógica de un conjunto de fórmulas proposicionales.

Tabla 2.2: Módulo SintaxSemanticsLP II. Funciones

2.3. LP Parser I. Módulo LP_Parser

En la sección anterior se ha mostrado el desarrollo necesario para la definición de las fórmulas proposicionales, pero como ya comentamos realizarlo a través de los constructores puede ser una tarea pesada por ello vamos a definir un lenguaje (a través de un Parser), que nos permita escribir las fórmulas de forma más natural, utilizando operadores infijos (*LP_Parser*) o utilizar operadores *Big*, que nos permitan expresar la conjunción/disyunción de muchas fórmulas análogas sin necesidad de tener que escribirlas una a una (*LPBig_Parser*)

En esta sección vamos a definir un Parser que nos permita trasladar la definición de las fórmulas desde un lenguaje mucho más cercano al lenguaje de escritura de la lógica a los constructores que se han definido en el módulo anterior, y con los que trabajaran los distintos algoritmos que se van a ir desarrollando. Para ello vamos a hacer uso de algunas herramientas que nos provee Elm para la construcción del Parser.

Vamos a definir las reglas asociadas a nuestro lenguaje:

- Las variables proposicionales (símbolos proposicionales) han de comenzar por una letra minúscula y han de contener exclusivamente caracteres alfanuméricos.
- Cada una de las fórmulas debe ir escrita entre paréntesis. Aunque en nuestro lenguaje esto es fundamental, la función parseadora de fórmulas ya incorpora estos paréntesis a la cadena leída, por lo que no es necesario que se escriban explícitamente los paréntesis.
- Se admiten los siguientes operadores:

Operador	Descripción
¬	Representa la Negación.
&	Representa la Conjunción.
|	Representa la Disjunción.
->	Representa la Implicación.
<->	Representa la Equivalencia.

Tabla 2.3: LP_Parser. Operadores Proposicionales

- Se admiten espacios (pero no son obligatorios) entre los símbolos proposicionales y los operadores.
- La prioridad de los operadores viene dada según el orden de prioridad definido por la lógica proposicional, según el orden en el que aparecen en la tabla anterior.
- Para la definición de conjuntos de fórmulas proposicionales, seguir las reglas anteriores para cada una de las fórmulas, separando éstas por el símbolo *;* .

2.3.1. Código del módulo y resumen de funciones

Código del módulo

```
module Modules.LP_Parser exposing(parserProp, parserPropSet)

import Char
import Set
import String
import Maybe

import Parser exposing (Parser, run, variable, oneOf, succeed, spaces, (|.),
                        (|=), symbol, lazy, andThen)

import Modules.SintaxSemanticsLP exposing (PSymb, Prop, atomProp, negProp,
                                           conjProp, disjProp, implProp,
                                           equiProp)
```

```
parserProp : String -> (Maybe (Prop), String)
parserProp x =
  if x == "" then
        (Maybe.Nothing, "Argument is empty")
  else
    case run lpParser ("(" ++ x ++ ")") of

      Ok y-> (Maybe.Just y, "")

      Err y -> (Maybe.Nothing, Debug.toString y)

parserPropSet : String -> List (Maybe(Prop), String)
parserPropSet x =  List.map parserProp <| String.split ";" x

typeVar : Parser PSymb
typeVar =
  variable
    { start = Char.isLower
    , inner = \c -> Char.isAlphaNum c
    , reserved = Set.fromList []
    }

lpParser : Parser Prop
lpParser =
  oneOf
  [
    succeed atomProp
      |. spaces
      |= typeVar
      |. spaces

  , succeed identity
    |. symbol "("
    |. spaces
    |= lazy(\_ -> expression)
    |. spaces
    |. symbol ")"
    |. spaces

  , succeed negProp
    |.spaces
    |.symbol "¬"
    |.spaces
    |= lazy(\_ -> lpParser)
  ]

expression : Parser Prop
expression =
  lpParser |> andThen (expressionAux [])

type Operator = AndOp | OrOp | ImplOp | EquivOp

operator : Parser Operator
operator =
    oneOf
    [ Parser.map (\_ -> AndOp) (symbol "&")
    , Parser.map (\_ -> OrOp) (symbol "|")
    , Parser.map (\_ -> ImplOp) (symbol "->")
    , Parser.map (\_ -> EquivOp) (symbol "<->")
    ]
```

```
expressionAux : List (Prop, Operator) -> Prop -> Parser Prop
expressionAux revOps expr =
  oneOf
    [ succeed Tuple.pair
        |. spaces
        |= operator
        |. spaces
        |= lpParser
        |> andThen
             (\(op, newExpr) -> expressionAux ((expr,op) :: revOps) newExpr)
    , lazy (\_ -> succeed (finalize revOps expr))
    ]

finalize : List (Prop, Operator) -> Prop -> Prop
finalize revOps finalExpr =
  case revOps of
    [] ->
      finalExpr


    (expr, AndOp) :: otherRevOps ->
      finalize otherRevOps (conjProp expr finalExpr)


    (expr, OrOp) :: (expr2, AndOp) :: otherRevOps ->
      disjProp (finalize ( (expr2, AndOp) :: otherRevOps) expr) finalExpr

    (expr, OrOp) :: otherRevOps ->
      finalize otherRevOps (disjProp expr finalExpr)


    (expr, ImplOp) :: (expr2, AndOp) :: otherRevOps ->
      implProp (finalize ( (expr2, AndOp) :: otherRevOps) expr) finalExpr

    (expr, ImplOp) :: (expr2, OrOp) :: otherRevOps ->
       implProp (finalize ( (expr2, OrOp) :: otherRevOps) expr) finalExpr

    (expr, ImplOp) :: otherRevOps ->
      finalize otherRevOps (implProp expr finalExpr)


    (expr, EquivOp) :: (expr2, AndOp) :: otherRevOps ->
      equiProp (finalize ( (expr2, AndOp) :: otherRevOps) expr) finalExpr

    (expr, EquivOp) :: (expr2, OrOp) :: otherRevOps ->
       equiProp (finalize ( (expr2, OrOp) :: otherRevOps) expr) finalExpr

    (expr, EquivOp) :: (expr2, ImplOp) :: otherRevOps ->
       equiProp (finalize ( (expr2, ImplOp) :: otherRevOps) expr) finalExpr

    (expr, EquivOp) :: otherRevOps ->
      finalize otherRevOps (equiProp expr finalExpr)
```

Listing 2.16: Módulo LP_Parser

En el anexo *A1. El Lenguaje Elm. Parsers* se explica cada una de las funciones del módulo Parser utilizadas en el código anterior.

Funciones disponibles

Método	Descripción
parserProp	*parserProp : String ->(Maybe (Prop), String)* Recibe una String y devuelve una tupla que contiene una Maybe Prop (Just Prop [si no hay errores] o Nothing [en caso de error], y una String (vacía () [si no hay error] o mensaje del error [en caso de error]).
parserPropSet	*parserPropSet : String ->List (Maybe(Prop), String)* Recibe una String y devuelve una lista de tuplas (análogas a las proporcionadas por *parserPropSet*) con cada una de las lecturas de las fórmulas.

Tabla 2.4: Módulo LP_Parser. Funciones disponibles

2.4. LP Parser II. Módulo LPBig_Parser

En la sección anterior desarrollamos un Parser para simplificar la escritura de las fórmulas. En esta vamos a realizar otro Parser que nos va a permitir utilizar los operadores *bigAnd* ($\bigwedge$) y *bigOr* ($\bigvee$), enfocado, principalmente, al modelado de restricciones para CSPs (*Constraint Satisfaction Problems*).

En este caso la complejidad del Parser es más alta, ya que hemos de recojer el conjunto de valores, que participarán en las formulas definida y estarán sujetos a una serie de filtros o condiciones (expresiones aritmetico-booleanas), por tanto hemos de realizar la adaptación de todos estos aspectos. Vamos a tratar de resumir los requisitos más relevantes y mostrar las reglas de sintáctico-semánticas asociadas a este lenguaje, y después veremos el desarrollo que se ha desarrollado de cada uno de los puntos.

Reglas Sintáctico-Semánticas

- Lo primero que vamos a presentar es la estructura general de las fórmulas con operadores *Big*, a lo largo de la sección iremos desarrollando cada una de las partes de estas expresiones. Mostremos la estructura:

  Operador {Declar. índices} {Restricc. índices} Fórmula

- ***Operador***. Existen 2 posibles operadores, el operador de conjunción múltiple (*BigAnd*), representado mediante el símbolo '&_'; y el operador de disyunción múltiple (*BigOr*), representado por el símbolo '|_'
- ***Declaración de índices***. Corresponde a la declaración de las variables que actuarán de índices de los símbolos proposicionales de las fórmulas. Además hemos de establecer los valores (numéricos enteros) que se permiten para los índices. La sintaxis de estas declaraciones es sencilla, el identificador debe comenzar por el símbolo '_' y ha de seguir por un secuencia de caracteres alfabéticos (en mayúsculas) y/o dígitos numéricos y seguido, los valores que puede tomar dicha variable. Para ello, se permiten 2 alternativas, declaración por valores o declaración por rango. La declaración por valores consiste en definir los valores concretos que tomará la variable, para ello se da la secuencia de valores entre llaves y separados por comas (ej. _I$\{1, 2, 3, 4\}$). En la declaración por rango se da el rango de valores en el que se moverá la variable. Para ello se da, entre corchetes, el primer número del rango seguido de *':'* y seguido del último número del rango (ej _I$[1 : 4]$).
- ***Restricción de índices***. Podemos establecer una restricción (y sólo una) sobre los valores de las variables, que puede afectar a todas (o sólo a algunas) de las variables declaradas. Corresponde a una expresión booleana que modela las posibles restricciones sobre los valores que pueden tomar las variables. Aunque solo se permite una única restricción, esta puede estar formada por varias condiciones, así tenemos distintas estructuras y operadores que podemos utilizar en estas expresiones:
  - *Condiciones*. Establecen posibles restricciones del problema modelado.
    - Condiciones básicas. Representan expresamente el valor de verdad de la condición. *'T'* corresponde a *True* y *'F'* corresponde a *False*.
    - Condiciones comparativas. Escrita entre corchetes, compara dos expresiones aritméticas. Está compuesta por tres elementos, las expresiones que son comparadas (*LHS* y *RHS*) y el operador comparador, que puede ser uno de *mayor o igual* (>=), *menor o igual* (>=), *mayor que* (>), *igual* (=), *distinto* (!=).

Las expresiones aritméticas corresponden a una expresión que puede contener valores numéricos enteros, variables de índices, de las previamente declaradas, y/o operadores aritméticos: *suma* (‘+’), *diferencia* (‘-’), *producto* (‘*’), *cociente entero* (‘/’) y módulo (‘%’).

Ejemplo: ‘&_ _I[0:7], _J[0:7]T(p_I_J ->&_ _K[-7:7],_U[0:7],_V[0:7] [_K!=0]AND[_U=_I+_K]AND[_V=_J-_K] (¬ p_U_V));’

2.4.1. Módulo A_Expressions (Expr. Aritméticas)

2.4.2. Módulo B_Expressions (Expr. Booleanas)

2.4.3. Módulo LPBig_Parser

2.5. Módulo LP_toString

Capítulo 3

LP II. Tableros Semánticos

Capítulo 4

LP III. Formas Normales y DPLL

Capítulo 5

LP IV. Sistemas Deductivos

Capítulo 6

LP V. Algoritmo de Resolución

Capítulo 7

LP VI. Modelado y Resolución de PSR a través de la LP

Capítulo 8

LPO I. Sintaxis y Semántica

Capítulo 9

LPO II. Tableros Semánticos

Capítulo 10

LPO III. Forma Prenex, Skolem y Teorema de Herbrand

Capítulo 11

LPO V. Algoritmos de Unificación y Resolución

Capítulo 12

LPO VI. Modelado y Resolución de PSR a través de la Lógica de Primer Orden