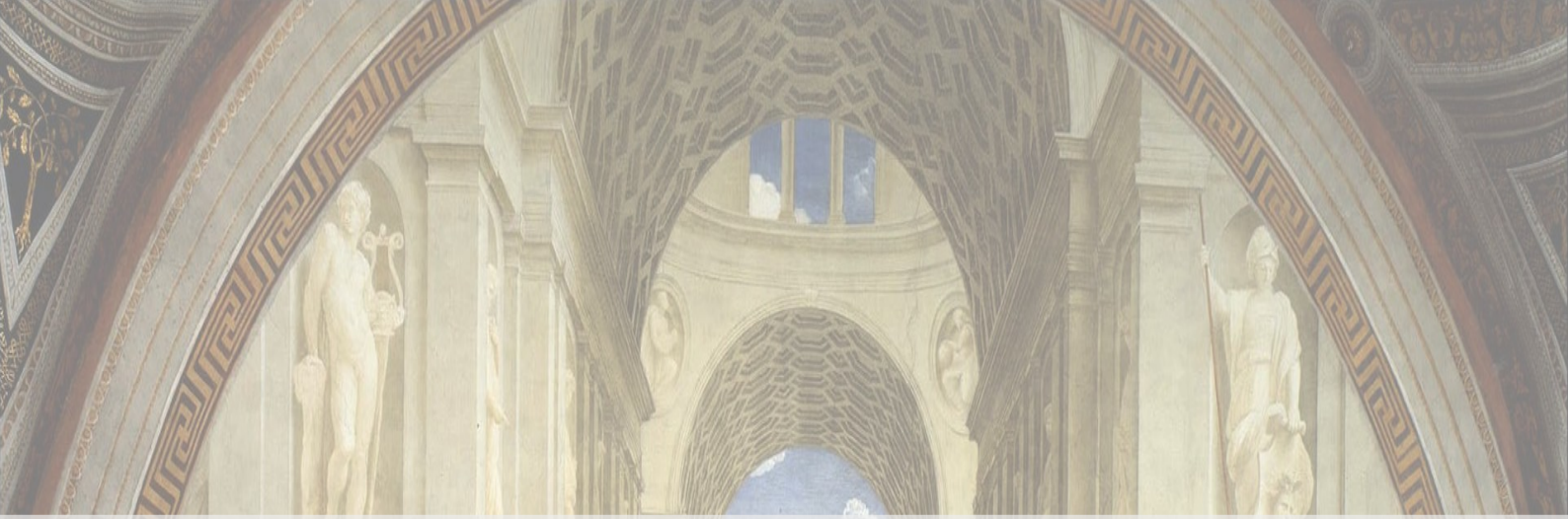

Desarrollo de la Lógica Proposicional y de Primer Orden bajo el paradigma funcional y la orientación Web.

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

Curso 2019/20

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

Capítulo 1

Introducción. Objetivos y organización del proyecto

1.1. Introducción

El proyecto surge desde 2 inspiraciones: por un lado mi interés para con el desarrollo de la Teoría de la Lógica Matemática y por otro mi gusto y vocación en el ámbito docente, por lo que la idea de poder desarrollar una herramienta, enfocada al ámbito académico, para el trabajo con diversas lógicas, me despertó una gran motivación.

El presente proyecto, pretende llevar a cabo el desarrollo de una herramienta que permita dar un punto de vista más práctico de los contenidos de la Lógica Proposicional y la Lógica de Primer Orden, al mismo tiempo que sirva al alumno como complemento para la comprensión de los conceptos y la realización y razonamiento de los ejercicios relacionados con dichos conceptos.

El desarrollo del proyecto se llevará a cabo bajo el paradigma de la programación funcional (con el lenguaje Elm) y la posibilidad que este ofrece para su sencilla implementación web para permitir la elaboración de una herramienta interactiva, sencilla y accesible.

1.2. Antecedentes del proyecto

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 y tras comprobar que son excasas las herramientas prácticas que abordan estos contenidos desde un punto de vista académica se planteó la realización de un proyecto análogo con el uso de otras herramientas y otros lenguajes, aunque finalmente, el proyecto no pudo llevarse a cabo.

No obstante la idea del desarrollo de esta herramienta seguía en mente y tras cursar la asignatura de *Programación Declarativa* con el profesor D. Miguel Ángel Martínez del Amor y el lenguaje Haskell, planteé a Fernando la posibilidad de retomar el proyecto pero bajo el lenguaje *Haskell* y el paradigma funcional. Sin embargo, una herramienta similar había sido desarrollada por el profesor D. José Antonio Alonso Jiménez en su obra *Lógica en Haskell*, en la que realiza un amplio desarrollo de la Lógica Proposicional y completada por D. Eduardo Paluzo en su TFG, en el que aborda con una metodología análoga al anterior los conceptos de la Lógica de Primer Orden.

Fue entonces cuando Fernando me propuso realizar una 'traducción' (incorporando algunos aspectos complementarios) de dichas obras, dentro del mismo paradigma pero bajo otro lenguaje, orientado además al ámbito Web, el lenguaje *Elm*.

1.3. Objetivos del proyecto

Como hemos comentado anteriormente, el proyecto persigue un objetivo dual:

- Por un lado, desarrollar una librería completa (compuesta de una serie de módulos) en el lenguaje *Elm*, que nos permita llevar a cabo la definición de conjuntos de fórmulas proposicionales y de primer orden, y la aplicación de los algoritmos básicos para tratar de abordar la satisfactibilidad de los mismos, mediante la aplicación de algoritmos fundamentales.
- Por otro lado, llevar a cabo una implementación Web, de manera que el sistema sea usable por los alumnos de manera sencilla y visual, para que sirva como complemento de los conceptos y técnicas abordados en la asignatura de *Lógica Informática.*

Además podemos destacar una serie de objetivos u competencias complementarias que aporta la realización de este proyecto como es la introducción al manejo de herramientas de publicación y manejo de versiones (*git*) o el manejo de *Latex*.

1.4. Estructura del proyecto

El proyecto se estructará en 2 partes fundamentales:

1.4.1. Módulos funcionales

En primer lugar, se llevará a cabo desarrollo de una librería completa que nos permita trabajar, tanto con fórmulas del ámbito de la Lógica Proposicional como de la Lógica de Primer Orden mediante la implementación de distintos módulos funcionales, que se describen detalladamente a lo largo de este documento y que recogen las estructuras, algoritmos y funciones necesarias para abordar los conceptos y técnicas vistas en la asignatura de *Lógica Informática*.

1.4.2. Interfaz gráfica (GUI)

En segundo lugar, se llevará a cabo el diseño e implementación de una interfaz web, que permita el uso del sistema desde 2 ámbitos distintos, por una parte desde el punto de vista de la definición de fórmulas y la aplicación directa de las funciones y técnicas sobre dichas fórmulas, y en segundo lugar la posibilidad de trabajar con las funciones de una manera más cercana a la *Programación Declarativa*.

Para ello se combinará, además del uso de los módulos funcionales, comentados anteriormente, los lenguajes *Elm*, *Html*, *CSS* y *js* para llevar a cabo la implementación de dicha interfaz.

1.4.3. Propósito y Estructura de este documento

Este documento pretende cumplir un doble objetivo:

- Por una parte se pretene que este documento pueda servir como material didáctico, a lo largo de los capítulos integraremos el desarrollo de los conceptos teóricos con las implementaciones llevadas a cabo, de forma que dichos conceptos se vean reflejados de forma casi directa en los códigos presentados.
- Por otra parte, es objeto de este documento servir como documentación del proyecto y manual de uso de la herramienta. Para facilitar esto, al final de cada uno de los capítulos se recogen los códigos completos de los módulos que intervienen en dicho capítulo, así como varias tablas resumen, en el que se presentan los tipos y funciones disponibles en cada uno de los módulos, junto a una somera descripción de las mismas.

Así, el documento se estructura en distintos capítulos, que servirán de unidades didácticas, y un conjunto de anexos en el que se presentan un conjunto de módulos complementarios, desarrollados para el funcionamiento del sistema.

Capítulo 2

Sintaxis y Semántica

2.1. Descripción general del capítulo

En este capítulo se recogen, de forma detallada, los módulos implementados que abordan el ámbito de la sintaxis y semántica de la Lógica Proposicional y Primer Orden. Complementariamente, en el *Anexo A. Parsers* se encuentra el desarrollo de varios Parsers, que nos permiten acercar la escritura natural de las fórmulas a la definición en el sistema.

2.1.1. Estructura del capítulo

El capítulo se encuentra estructurado en distintas secciones, a través de las cuales se abordan los conceptos fundamentales del ámbito sintáctico-semántico de la LP y la LPO, presentados conjuntamente con los módulos que implementan dichos conceptos:

Módulos

- **Módulo SintaxSemanticsLP**. Recoge las implementaciones de los tipos fundamentales relacionados con los aspectos sintáctico-semánticos de la Lógica Proposicional.
- **Módulo SintaxSemanticsLPO**. Recoge las implementaciones de los tipos fundamentales relacionados con los aspectos sintáctico-semánticos de la Lógica de Primer Orden.

Módulos Complementarios (*A1. Parsers*)

- **Módulo LP_Parser**. Recoge la implementación de un Parser, que permite la escritura de fórmulas LP según la estructura natural, permitiendo el uso de operadores infijos (negación, conjunción, disyunción, implicación, equivalencia).
- **Módulo LPO_Parser**. Recoge la implementación de un Parser, que permite la escritura de fórmulas LPO según la estructura natural, permitiendo, además de los operadores LP (negación, conjunción, disyunción, implicación, equivalencia), el uso de operadores LPO (existencial y universal).
- **Módulo LP_toString**. Recoge algunas funciones destinadas a la presentación de las fórmulas en formato de cadena de texto y en formato *Latex*.

2.2. Conceptos básicos de la Lógica Proposicional

2.2.1. Caracetización básica de la Lógica Proposicional

La Lógica surge como método de modelado del siguiente problema:

> Dado un conjunto de asertos (afirmaciones), $\mathcal{BC}$ (*Base de conocimiento*), y una afirmación, $\mathcal{A}$, decidir si $\mathcal{A}$ ha de ser necesariamente cierta supuestas ciertas las fórmulas de $\mathcal{BC}$.

De manera que para abordar este problema desde el punto de vista lógico-proposicional, resultan necesarios los siguientes elementos:

- Un lenguaje que permita expresar de forma precisa las afirmaciones. (Sintáxis)
- Una definición clara de qué se entiende por *afirmación cierta* (Semántica)
- Mecanismos efectivos (y a poder ser eficientes) que garanticen la corrección (y preferentemente la completitud) en las deducciones. (Algoritmos de resolución)

A lo largo de los distintos capítulos abordaremos estos puntos para las dos representaciones más comunes, la Lógica Proposicional (*LP* o *PL*) y la Lógica de Primer Orden (*LPO* o *FOL*). P or el momento vamos a comenzar este primer capítulo abordando los dos primeros puntos para la Lógica Proposicional y la Lógica de Primer Orden, mientras que el desarrollo de las técnicas deductivas se abordará a lo largo del resto de capítulos.

Características fundamentales de la LP.

- Sus expresiones (denominadas f*órmulas proposicionales* o *proposiciones*) modelan afirmaciones que pueden considerarse *ciertas* o *falsas*.
- Las fórmulas proposicionales (en adelante fórmulas (si no existe ambigüedad)), se construyen mediante un conjunto de expresiones básicas (*fórmulas atómicas* o *átomos*) y conjunto de operadores (*conectivas lógicas*). Dichas conectivas permiten modelar los siguientes tipos de afirmaciones:
  - *Conjunción*: '... tal ... Y ... cual ...'
  - *Disyunción*: '... tal ... O ... cual ...'
  - *Implicación*: 'SI tal ... ENTONCES ... cual ...'
  - *Equivalencia*: '... tal ... SI Y SÓLO SI ... cual ...'
  - *Negación*: 'NO es cierto tal ...'

  Profundizaremos en este aspecto en la próxima sección, cuando veamos la *Sintaxis de la LP*.
- El lenguaje sólo permite modelar este tipo de afirmaciones, por lo que muchas veces puede ser difícil (o imposible) representar el problema en este tipo de Lógica, y es necesario recurrir a otras más ricas (*LPO, Lógicas Modales, Lógica Fuzzy, etc*).
- Aunque esta Lógica puede resultar de una aparente sencillez, el problema *SAT* corresponde a la categoría de problemas NP-completos, esto es, no existe ningún algoritmo capaz de resolver el problema planteado en un tiempo polinomial de ejecución.

2.3. Módulo SintaxSemanticsLP

En este primera sección vamos a abordar desde un punto de vista teórico-práctico, los elementos base que conforman la Lógica Proposicional, esto es la Síntaxis y la Semántica, mostrando unificadamente los desarrollos formales como las implementaciones llevadas a cabo para modelar cada uno de ellos.

2.3.1. Aspectos Sintácticos

El alfabeto proposicional

El concepto '*alfabeto proposicional*' referencia al conjunto de símbolos que forman parte de este lenguaje. Podemos distinguir las siguientes categorías:

- **Variables proposicionales o átomos**. Ya hemos señalado previamente que todo problema está representado por relaciones entre un conjunto finito de afirmaciones básicas, dichas afirmaciones se representan por símbolos proposicionales: $VP = \{p_0, p_1, \ldots, p, q, r\}$.

  En el lenguaje LIUS, corresponden a cadenas de caracteres:

```
type alias PSymb = String
```

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

  Aunque esta definición admite cualquier cadena como símbolo proposicional, vamos a adoptar algunas reglas de notación, que presentaremos a lo largo del capítulo y que vamos a implementar en el parser que nos permitirá pasar de cadenas que representan fórmulas en un lenguaje cercano al lenguaje formal de la lógica a fórmulas proposicionales reconocidas por el sistema. Adoptamos el siguiente criterio en relación a los símbolos proposicionales:

  '*Los símbolos proposicionales deben comenzar por una letra minúscula, seguida (opcionalmente) de otros caractéres en minúscula o dígitos numéricos, exclusivamente.*'

  [Ver la implementación del módulo *LP_Parser* para ver la implementación de dicho criterio. (*Ver A1. LP_Parser*)]

- **Conectivas Lógicas**. Modelan las relaciones entre las distintas afirmaciones básicas (si es que las hay). Podemos distinguir:
  - De aridad 1 o monoaria : *Negación* ($\neg$)
  - De aridad 2 o binarias: *Conjunción* ($\wedge$), *Disyunción* ($\vee$), *Condicional* ($\rightarrow$), *Bicondicional* ($\leftrightarrow$).

  En el lenguaje LIUS, corresponden a constructores, los veremos seguidamente cuando abordemos el concepto de fórmula.
- **Símbolos Auxiliares**: '*(*' y '*)*'. Permiten expresar relaciones de prioridad entre conectivas lógicas y evitar la ambigiüedad en la interpretación de las fórmulas.

  En el lenguaje LIUS (en la definición fundamental) no son necesarios, ya que la prioridad viene dada por el orden de uso de los constructores, pero sí serán necesarios en la definición del Parser, en el que se escribirán del mismo modo y cumplirán la función que se ha definido para los mismos.

Fórmula Proposicional

Una fórmula proposicional corresponde a una fórmula atómica (un símbolo proposicional) o a conjunto de símbolos proposicionales (con cardinalidad mayor o igual que 2), relacionados entre sí por alguna de las conectivas lógicas. Formalmente, El conjunto de las fórmulas proposicionales, **PROP**, es el menor conjunto de expresiones que verifica:

- $VP \subseteq PROP$
- Es cerrado bajo las conectivas lógicas, esto es:
  - Si una fórmula $F \in PROP$, entonces $\neg F \in PROP$
  - Si las fórmulas $F, G \in PROP$, entonces $(F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G) \in PROP$

De manera que se tiene una definición recursiva del concepto de *Fórmula*, tal que el caso base corresponde a una fórmula básica (*átomo*) y el caso recursivo corresponde a la aplicación de una conectiva sobre una o dos fórmulas (según la aridad de la conectiva).

En el lenguaje LIUS, hemos definido el concepto de Fórmula Proposicional, como un nuevo tipo, dando una definición recursiva análoga a la presentada anteriormente, mediante los constructores *Atom* (Átomo), *Conj* (Conjunción), *Disj* (Disyunción), *Impl* (Implicación o Condicional), *Equi* (Equivalencia o Bicondicional) y un tipo más que *Insat* que corresponde a la fórmula insatisfactible (la veremos cuando abordemos la *Semántica de LP*).

```
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 (Fórmula Proposicional).

Prioridad de las conectivas y Reducción de Paréntesis

1. Omitimos los paréntesis externos
2. Tomaremos como orden de precedencia de las conectivas (de mayor a menor): $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Para la conectiva $\leftrightarrow$ se recomienda mantener los paréntesis en todos los casos.
3. Cuando una conectiva se usa repetidamente, se asocia por la derecha.

Si definimos las fórmulas a partir de los constructores, estas reglas no son necesarias, ya que no puede existir ambigüedad alguna en la definición fórmula, pero sí son relevantes en el caso de la declaración a partir del Parser.

[Ver la implementación del módulo *LP_Parser* para ver la implementación de dicho criterio. (*Ver A1. LP_Parser*)]

Esto nos permite definir todas las fórmulas proposicionales, presentamos la construcción de las siguientes 3 fórmulas a modo de ejemplo:

(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, se ha desarrollado un parser que nos permite escribir de forma más cómoda, sintética y visual las fórmulas.

A partir de este momento todas las fórmulas de los ejemplos se definirán utilizando el Parser por lo que llegados a este punto se recomienda realizar un estudio del mismo, para tener claras las reglas sintácticas a seguir, aunque, son análogas a las planteadas por el lenguaje formal de la lógica proposicional. (*Ver Anexo A1. LP_Parser*).

Árboles de formación

Los árboles de formación corresponden a grafos de tipo árbol que muestran el desarrollo de formación de las fórmulas (siguiendo la definición recursiva de las mismas) por lo que el árbol de formación asociado a una fórmula es esencialmente único.

En el lenguaje LIUS podemos generar estos árboles de formación utilizando la función *formTree* definida en el módulo que estamos tratando. En la siguiente página se muestra el código de implementación de la función. (La complejidad del código reside más en aspectos técnicos relacionados con el lenguaje más que en el algoritmo, por lo que no es necesario entender el código, es suficiente con entender el método y conocer la existencia de la función).

Para poder visualizar el árbol de formación:

$\Longrightarrow$ *COMPLETAR*

```
formTree : Prop -> Graph String ()
formTree x =
    case x of
        Atom psymb -> fromNodesAndEdges [Node 0 psymb] []
        Neg p ->
            let (nodes, edges) = formTreeAux p 1 in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::nodes) (Edge 0 1 ()::edges)
        Conj p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Disj p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Impl p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Equi p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Insat -> fromNodesAndEdges [Node 0 (toStringProp x)] []

formTreeAux : Prop -> NodeId -> (List (Node String), List (Edge ()))
formTreeAux x nodeid=
    case x of
        Atom psymb -> ([Node nodeid psymb], [])
        Neg p ->
            let
                nextid = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
            in
                let
                    (nodes, edges) = formTreeAux p nextid
                in
                    (Node nodeid (toStringProp x)::nodes,
                                        Edge nodeid nextid ()::edges)
```

```
        Conj p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                            edges1 ++ edges2)
        Disj p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                            edges1 ++ edges2)
        Impl p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                            edges1 ++ edges2)
        Equi p q ->
           let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
```

```
                ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                    [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                        edges1 ++ edges2)
    Insat -> ([Node nodeid (toStringProp x)], [])
```

Listing 2.4: Definición de la función formTree

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 2.5: Definición de Conjunto de Fórmulas como Lista de Fórmulas Proposicionales

2.3.2. 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 2.6: 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 2.7: 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 2.8: 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 2.9: 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 2.10: 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 2.11: 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 2.12: 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 2.13: 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 2.14: 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 2.15: Consecuencia Lógica

2.4. Módulo SintaxSemanticLPO

2.5. Código y funciones de los módulos

2.5.1. SintaxSemanticsLP

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 2.16: 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

2.6. Ejercicios Propuestos

2.7. Resolución de los Ejercicios Propuestos

Capítulo 3

LP II. Tableros Semánticos

Anexos

Anexo A

Parsers

A.1. Módulo LP_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 A.1: 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 *;* .

A.1.1. 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 A.1: 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 A.2: Módulo LP_Parser. Funciones disponibles

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));’