

# TEMA 1

## Sintaxis y Semántica en LP y LPO

### 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.

### 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:

- **Módulo SyntaxSemanticsLP.** Recoge las implementaciones de los tipos fundamentales relacionados con los aspectos sintácticos y semánticos de la Lógica Proposicional.
- **Módulo SyntaxSemanticsLPO.** Recoge las implementaciones de los tipos fundamentales relacionados con los aspectos sintácticos y semánticos de la Lógica de Primer Orden.
- **Módulo IO\_LP.** Recoge las implementaciones de los métodos relacionados con la lectura y representación de las fórmulas LP.
- **Módulo IO\_LPO.** Recoge las implementaciones de los métodos relacionados con la lectura y representación de las fórmulas LPO.

## 1.1. Conceptos básicos de la Lógica Proposicional

### Caracterización 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 resultan necesarios los siguientes elementos:

- Un lenguaje que permita expresar de manera formal y precisa las afirmaciones, hechos e hipótesis.  
(Sintaxis)
- Una definición clara de qué se considera *afirmación cierta* (Semántica)

- Mecanismos efectivos (y a poder ser eficientes) que garanticen la corrección (y preferentemente la completitud) en las deducciones. (Algoritmos de decisión)

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*).

Por el momento vamos a comenzar este primer capítulo abordando los dos primeros puntos para la Lógica Proposicional y la Lógica de Primer Orden, dando una somera introducción al tercero para cada caso dejando el desarrollo de los algoritmos de decisión, que se irán abordando a lo largo del resto de capítulos.

## Características fundamentales de la LP.

- Sus expresiones (denominadas *fórmulas 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). Especificaremos este apartado cuando tratemos las limitaciones de la LP e introduzcamos la LPO.
- 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. Trataremos de nuevo este aspecto en la introducción a los algoritmos de decisión.

## 1.2. Fundamentos de la lógica proposicional

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

### Sintáxis de la lógica Proposicional

#### 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:  $VP = \{p_0, p_1, \dots, p, q, r\}$ .

Aunque, formalmente, no existe ninguna restricción, vamos a adoptar el siguiente criterio en relación a la sintaxis de los símbolos proposicionales:

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

- **Conectivas Lógicas.** Modelan las relaciones entre las distintas afirmaciones básicas (átomos). Podemos distinguir:
  - De aridad 1 o monoaria : *Negación* ( $\neg$ ).
  - De aridad 2 o binarias: *Conjunción* ( $\wedge$ ), *Disyunción* ( $\vee$ ), *Condicional* ( $\rightarrow$ ), *Bicondicional* ( $\leftrightarrow$ ).
- **Símbolos Auxiliares:** '(' y ')'. Permiten expresar relaciones de prioridad entre conectivas lógicas y evitar la ambigüedad en la interpretación de las fórmulas.

## Fórmula Proposicional

Se denomina expresión del lenguaje proposicional a cada una sucesión finita (y no vacía) de sus símbolos y conectivas. Un ejemplo de expresión puede ser la siguiente  $a \rightarrow b)ca\vee$ , otro puede corresponder a  $(a \rightarrow b) \vee c$ . Parece claro que el primero posee "una coherencia" o "equilibrio" que no parece tener el primero. De esta forma, las expresiones "bien formadas" las denominaremos **fórmulas proposicionales**.

Formalmente, El conjunto de las fórmulas proposicionales,  $PROP$ , es el menor conjunto de expresiones que verifica:

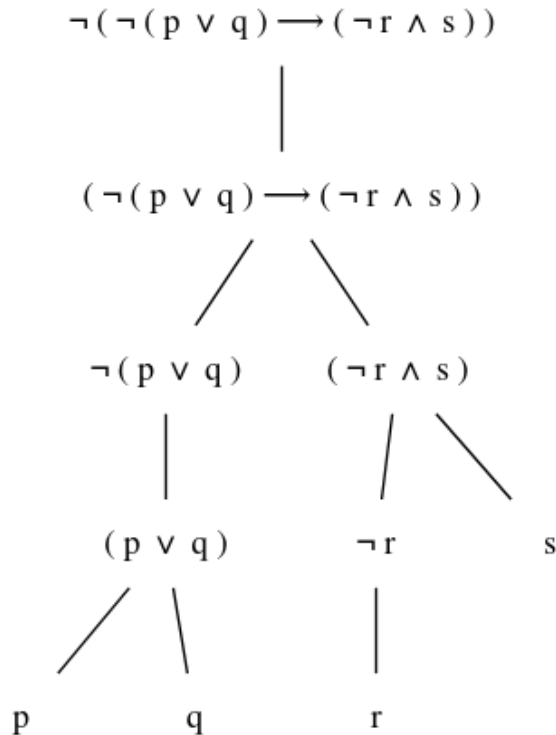
- $VP \subseteq PROP$
- Es cerrado bajo las conectivas lógicas, esto es:
  - Si una fórmula  $F \in PROP$ , entonces  $\neg F \in PROP$
  - Si las fórmulas  $F, G \in PROP$ , entonces  $(F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G) \in PROP$

De manera que se tiene una definición recursiva del concepto de *Fórmula Proposicional*, 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).

## Árboles de formación

Cada una de las fórmulas proposicionales lleva asociado un grafo de tipo árbol (esencialmente único), que muestra el desarrollo de formación de la fórmula, siguiendo la definición recursiva de la misma.

Por ejemplo a la fórmula:  $\neg(\neg(p \vee q) \rightarrow (\neg r \wedge s))$  le corresponde el árbol de formación:



### Prioridad de conectivas y Reducción de paréntesis.

Para facilitar la escritura y lectura de las fórmulas vamos a adoptar algunos criterios:

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.

### Principio de inducción sobre fórmulas

Gracias a la definición de *PROP* (y su estructura recursiva), para probar que toda fórmula proposicional satisface una cierta propiedad ( $\Psi$ ), podemos hacerlo aplicando el método de inducción sobre fórmulas.

De esta forma, probamos:

1. (Caso base). Probar que todos los elementos de *VP* tienen la propiedad  $\Psi$ .
2. (Paso de inducción).
  - Si  $F \in PROP$  tiene la propiedad  $\Psi$ , entonces  $\neg F$  tiene la propiedad  $\Psi$ .
  - Si  $F, G \in PROP$ , tienen la propiedad  $\Psi$  entonces  $(F \wedge G)$ ,  $(F \vee G)$ ,  $(F \rightarrow G)$ ,  $(F \leftrightarrow G) \in PROP$  tienen la propiedad  $\Psi$

### Escritura de las fórmulas en Logicus

Vamos a exponer cuál es la sintaxis de las fórmulas utilizando la librería Logicus, que nos permitirá definir fórmulas y aplicar diversos algoritmos sobre las mismas.

En la librería podemos definir las fórmulas de dos formas distintas, o bien acudiendo directamente a los constructores de las fórmulas (largo y engorroso), o bien utilizando el Parser (de forma análoga a la escritura natural de las fórmulas).

—→ Definición de fórmulas a partir de constructores

Si recordamos la definición recursiva que se ha dado para las fórmulas, expusimos que las mismas pueden corresponder o bien a átomos (*caso base*), o bien a la aplicación de las conectivas, teniendo en cuenta la aridad de éstas, sobre fórmulas proposicionales (*caso recursivo*).

Bien, pues la implementación dada en la librería para las fórmulas corresponde fielmente a dicha definición. De forma que:

- Los símbolos proposicionales se definen como cadenas de caracteres, (recordando el criterio, estos caracteres deben ir en minúscula).

```
type alias PSymb = String
```

- Las fórmulas proposicionales se definen según la estructura recursiva presentada.

```
type FormulaLP = Atom PSymb           -- caso base
      | Neg FormulaLP                 -- }
      | Conj FormulaLP FormulaLP      -- }
      | Disj FormulaLP FormulaLP      -- } casos recursivos
      | Impl FormulaLP FormulaLP      -- }
      | Equi FormulaLP FormulaLP      -- }
      | Insat                         -- fórmula insatisfactible
```

Esto nos permite definir todas las fórmulas proposicionales. Para poder hacer uso de la librería logicus tenemos que importar de los módulos correspondientes las funciones y tipos necesarios para nuestro trabajo.

Expongamos algunos ejemplos:

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

```
import Modules.IO_LP exposing (toLatexFLP)
import Modules.SintaxSemanticsLP exposing (FormulaLP(..))

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

```
a : ((p ∧ q) ∨ (p ∧ r))
```

```

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

```

$$b : ((p \wedge r) \vee (\neg p \wedge q)) \rightarrow \neg q$$

```

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

```

$$c : ((p \leftrightarrow q) \wedge (p \rightarrow \neg q)) \wedge p$$

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. Para poder utilizarlo se han establecido una serie de requisitos sintácticos, análogos a los presentados anteriormente:

- Los símbolos proposicionales deben comenzar por un caracter en minúscula seguido, opcionalmente, de caracteres en minúscula, dígitos o el símbolo '\_'.
- Para las conectivas se usarán los siguientes símbolos, mantiéndose la prioridad de las conectivas definida (en el orden de prioridad descendente expuesto en la tabla).

<u>Conectiva Lógica</u>	<u>Símbolo Logicus</u>
Negación ( $\neg$ )	'NOT'
Conjunción ( $\wedge$ )	'AND'
Disyunción ( $\vee$ )	'OR'
Implicación ( $\rightarrow$ )	'IMPLIES'
Equivalencia ( $\leftrightarrow$ )	'EQUIV'

- Los paréntesis se utilizan de igual forma en que se han definido en el lenguaje formal de la lógica proposicional, con los símbolos '(' y ')'. No son necesarios los paréntesis externos de las fórmulas.
- En caso de uso repetido de una misma conectiva, se realizará asociación por la derecha.

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 tener claras las reglas sintácticas a seguir, aunque, son análogas a las planteadas por el lenguaje formal de la lógica proposicional.

Para definir los ejemplos expuestos anteriormente:

```
import Modules.SyntaxSemanticsLP exposing (FormulaLP(..))
import Modules.IO_LP exposing (toLatexFLP, fromStringToFLP, extractReadFLP)
```

```
a : FormulaLP
a = fromStringToFLP "(p AND q) OR (p AND t)" |> extractReadFLP
```

$a : ((p \wedge q) \vee (p \wedge t))$

```
b : FormulaLP
b = fromStringToFLP "(p AND r) OR (NOT p AND q) IMPLIES NOT q" |> extractReadFLP
```

$b : (((p \wedge r) \vee (\neg p \wedge q)) \rightarrow \neg q)$

```
c : FormulaLP
c = fromStringToFLP "(p EQUIV q) AND (p IMPLIES NOT q) AND p" |> extractReadFLP
```

$c : ((p \leftrightarrow q) \wedge ((p \rightarrow \neg q) \wedge p))$

Nótese que la función *fromStringToFLP* proporciona una tupla de dos elementos el primero de ellos corresponde a la fórmula leída y el segundo a una cadena de texto. Bien, si cometemos algún error sintáctico en la escritura de la fórmula el Parser no será capaz de interpretarla correctamente por lo que devolverá una tupla (*Nothing,error*), en el que el segundo elemento corresponderá al mensaje de error y el primer elemento a un objeto vacío.

Una vez leída la fórmula correctamente podemos extraerla utilizando la función *extractReadFLP*. Si utilizamos dicha función sobre una fórmula leída incorrectamente, se extraerá como fórmula la fórmula insatisfactible.

→ Árboles de formación en Logicus

La librería también permite la representación de los árboles de formación. La función *formtree* muestra la representación del árbol de formación en formato texto DOT.

Por ejemplo para la fórmula  $\neg(\neg(p \vee q) \rightarrow (\neg r \wedge s))$ :

```
import Modules.IO_LP exposing (toLatexFLP, fromStringToFLP, extractReadFLP, formTree)
import Modules.SyntaxSemanticsLP exposing (FormulaLP(..))
import Modules.AuxForLitvis exposing (showGraphViz)
```

```
a : FormulaLP
a = fromStringToFLP "NOT ( NOT (p OR q) IMPLIES (NOT r AND s))" |> extractReadFLP
```

```
fta : String
fta = formTree a
```

De forma que se obtiene:

## Conjuntos de fórmulas

Definido  $PROP$ , los conjuntos de fórmulas no son más que subconjuntos de  $PROP$ , esto es, corresponden a agrupaciones de fórmulas proposicionales.

## Conjuntos de fórmulas en Logicus

Los conjuntos proposicionales se definen como listas de fórmulas proposicionales. De esta forma:

```
type alias LPSet = List FormulaLP
```

De forma que la definición de estos se realiza como listas de objetos *FormulaLP*. Si por ejemplo queremos definir el conjunto:

$$M = \{(p \wedge q) \vee (p \wedge r), (p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q, (p \leftrightarrow q) \wedge (p \rightarrow \neg q) \wedge p\}$$

Podríamos hacerlo:

```
import Modules.IO_LP exposing (..)
import Modules.SyntaxSemanticsLP exposing (..)

a : FormulaLP
a = fromStringToFLP "(p AND q) OR (p AND r)" |> extractReadFLP

b : FormulaLP
b = fromStringToFLP "(p AND r) OR (NOT p AND q) IMPLIES NOT q" |> extractReadFLP

c : FormulaLP
c = fromStringToFLP "(p EQUIV q) AND (p IMPLIES NOT q) AND p" |> extractReadFLP

m : LPSet
m = [a,b,c]
```

De forma que se obtiene:

$$M : \{((p \wedge q) \vee (p \wedge r)), (((p \wedge r) \vee (\neg p \wedge q)) \rightarrow \neg q), ((p \leftrightarrow q) \wedge ((p \rightarrow \neg q) \wedge p))\}$$

## Semántica de la lógica Proposicional

### Interpretaciones, Modelos, Satisfactibilidad y Validez Lógica

Una vez provista la sintaxis, pasamos a desarrollar la semántica de la Lógica Proposicional. Como ya comentamos, hemos de abordar la interpretación de certeza o veracidad de las fórmulas. Para esto es necesario conocer los conceptos de *valor de verdad* y *función de verdad*.

- **Valor de verdad.** Los elementos del conjunto  $\{0, 1\}$  se denominan valores de verdad o valores booleanos. Representan si un hecho es cierto o no, de forma que el valor 1 se asocia a *verdadero* y el valor 0 a *falso*.
- **Funciones de verdad.** Corresponden a funciones que devuelven un valor de verdad según el valor de verdad de los argumentos. Así, el significado (valor de verdad asociado) de cada una de las conectivas



lógicas viene dado por una función de verdad, de forma que:

$$\begin{aligned}
 H_{\neg}(i) &= \begin{cases} 1 & \text{si } i = 0 \\ 0 & \text{si } i = 1 \end{cases} & H_{\wedge}(i, j) &= \begin{cases} 1 & \text{si } i = j = 1 \\ 0 & \text{e.o.c} \end{cases} \\
 H_{\vee}(i, j) &= \begin{cases} 0 & \text{si } i = j = 0 \\ 1 & \text{e.o.c} \end{cases} & H_{\rightarrow}(i, j) &= \begin{cases} 0 & \text{si } i = 1, j = 0 \\ 1 & \text{e.o.c} \end{cases} \\
 H_{\leftrightarrow}(i, j) &= \begin{cases} 1 & \text{si } i = j \\ 0 & \text{e.o.c} \end{cases}
 \end{aligned}$$

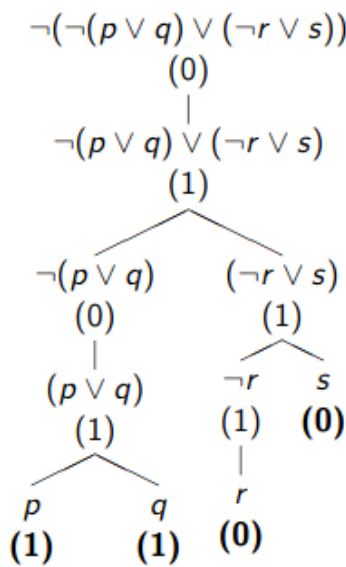
Visto esto, pasamos a estudiar el valor de verdad de las fórmulas proposicionales. Para ello debemos definir el valor de verdad de las variables proposicionales, (denominadas *valoraciones* o *interpretaciones*) y a partir de éstas y las funciones de verdad de las conectivas, podemos extender cada valoración,  $v$ , de forma única, al conjunto de todas las fórmulas de manera que se verifica:

$$\begin{aligned}
 v((F \wedge G)) &= H_{\wedge}(v(F), v(G)) & v((F \vee G)) &= H_{\vee}(v(F), v(G)) \\
 v((F \rightarrow G)) &= H_{\rightarrow}(v(F), v(G)) & v((F \leftrightarrow G)) &= H_{\leftrightarrow}(v(F), v(G))
 \end{aligned}$$

Dada una fórmula  $F \in PROP$ , se dice que  $v(F)$  es el valor de verdad de  $F$  respecto de la valoración  $v$ .

De esta forma, es sencillo realizar el cálculo del valor de verdad de una fórmula respecto de una valoración, recurriendo al árbol de formación de la fórmula, evaluando las subfórmulas, desde las hojas (variables proposicionales) hasta el nodo raíz (la fórmula completa).

Por ejemplo el cálculo de la valoración de  $F \equiv \neg(\neg(p \vee q) \vee (\neg r \vee s))$  respecto de  $v \equiv \{p = 1, q = 1, r = 0, s = 0\}$ :



Además del cálculo a través del Árbol de Formación, existe otro método (equivalente) para el cálculo del valor de verdad de una fórmula respecto de una valoración, cálculo a través de Tabla. Dado que, dada una valoración,  $v$ ,

el valor de verdad de una fórmula  $F$  respecto de  $v$  está determinado por los valores de verdad de las subfórmulas de  $F$ , podemos construir una tabla que recorra los valores de sus subfórmulas. Para el ejemplo anterior:

p	q	r	s	$\neg r$	$p \vee q$	$\neg(p \vee q)$	$\neg r \vee s$	$\neg(p \vee q) \vee \neg(r \vee s)$	$\neg(\neg(p \vee q) \vee (\neg r \vee s))$
1	1	0	0	1	1	0	1	1	0

Tablas de Verdad

Una tabla de verdad corresponde a una estructura similar a la anterior (nosotros sólo reflejaremos el valor de las variables proposicionales y el valor de verdad de la fórmula completa), en la que en cada fila se presenta la valoración y el valor de verdad de la fórmula respecto a la misma, para toda interpretación posible (que corresponda a las variables proposicionales presentes en la fórmula).

Para la fórmula anterior, su tabla de verdad correspondería a:

p	q	r	s	$\neg(\neg(p \vee q) \vee (\neg r \vee s))$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0

p	q	r	s	$\neg(\neg(p \vee q) \vee (\neg r \vee s))$
1	1	1	0	1
1	1	1	1	0

## Modelos, Satisfactibilidad y Validez Lógica

Una vez estudiado lo anterior, vamos dar unas cuantas definiciones:

- **Modelo.** Se dice que una fórmula  $F$  es válida en una valoración  $v$  o equivalentemente que  $v$  es **modelo** de  $F$  si  $v(F) = 1$  y se denota por  $v \models F$ . En caso contrario, se dice que  $v$  es **contramodelo** de  $F$  y se denota  $v \not\models F$ .
- **Satisfactibilidad.** Una fórmula  $F$  se dice **satisfactible** (o consistente) si existe una valoración  $v$  que es modelo de  $F$ . En caso contrario se dice que  $F$  es **insatisfactible** (o inconsistente), y se representa por  $\perp$ .
- **Validez lógica o Tautología.** Una fórmula  $F$  se dice **tautología** (o (lógicamente) válida) si toda valoración es modelo de  $F$  y se denota  $\models F$ .

—→ Relación entre Validez y Satisfactibilidad

LEMA: Para cada  $F \in PROP$  se verifica:

- Si  $F$  es tautología, entonces  $F$  es satisfactible.
- $F$  es tautología si y sólo si  $F$  es insatisfactible.

## Semántica de Fórmulas LP en Logicus

La librería dispone de funciones que modelan todos los conceptos semánticos que hemos visto hasta ahora.

—→ Valores, Funciones de Verdad e Interpretaciones

Como hemos estudiado los **valores de verdad** corresponden a 1 (*verdadero*) y 0 (*falso*). Elm ya provee esos valores booleanos en el tipo `Bool`, por lo que no es necesario realizar ninguna definición alternativa para este concepto.

Las definiciones de las funciones de verdad asociadas a las conectivas corresponden directamente a la aplicación de dichas funciones en la evaluación de las fórmulas. Antes de ver la evaluación resulta necesario ver la definición que se ha dado para las interpretaciones. Para ello, se ha elegido una representación "dispersa" de manera que una interpretación corresponde a una lista de símbolos proposicionales (variables 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
```

Con esta definición resulta sencillo llevar a cabo la evaluación de las fórmulas. Tal y como se ha planteado formalmente, podemos distinguir el proceso de evaluación en 2 casos:

- **Evaluación de variables.** Una variable será verdadera si en la interpretación le otorga dicho valor, esto es, según la definición dada, será verdadera si pertenece a la lista de la interpretación, y será falsa en caso contrario.
- **Evaluación de conectivas.** Según la definición formal dada:

$$H_{\neg}(i) = \begin{cases} 1 & \text{si } i = 0 \\ 0 & \text{si } i = 1 \end{cases} \quad H_{\wedge}(i, j) = \begin{cases} 1 & \text{si } i = j = 1 \\ 0 & \text{e.o.c} \end{cases}$$
$$H_{\vee}(i, j) = \begin{cases} 0 & \text{si } i = j = 0 \\ 1 & \text{e.o.c} \end{cases} \quad H_{\rightarrow}(i, j) = \begin{cases} 0 & \text{si } i = 1, j = 0 \\ 1 & \text{e.o.c} \end{cases}$$
$$H_{\leftrightarrow}(i, j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{e.o.c} \end{cases}$$

De forma que:

```
valuation : FormulaLP -> 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
    Insat -> Basics.False
```

NOTA: nótese que la fórmula insatisfactible, lógicamente, es falsa respecto de cualquier valoración.

De forma que la función *valuation* nos permite calcular la valoración de una fórmula respecto de una valoración. Por ejemplo, calculemos la valoración de  $F \equiv \neg(\neg(p \vee q) \vee (\neg r \vee s))$  respecto de  $v \equiv \{p = 1, q = 1, r = 0, s = 0\}$ :

```
import Modules.IO_LP exposing (..)
import Modules.SyntaxSemanticsLP exposing (..)

f : FormulaLP
f = fromStringToFLP "NOT (NOT (p OR q) OR (NOT r OR s))" |> extractReadFLP

i : Interpretation
i = ["p", "q"]
```

False

## —→ Tabla de Verdad, Modelos, Satisfactibilidad y Validez

Como se ha expuesto anteriormente una tabla de verdad recoge las valoraciones de una fórmula respecto a todas las posibles interpretaciones que integran a sus símbolos proposicionales. Bien dada la definición de las interpretaciones resulta bien sencillo el cálculo de todas las posibles interpretaciones, de hecho el conjunto de interpretaciones corresponde a todos los posibles subconjuntos del conjunto formado por los símbolos proposicionales que participan en la fórmula. De esta forma:

```
symbInProp : FormulaLP -> 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)
    Insat -> Set.empty

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

Y la tabla de verdad corresponde al conjunto de pares (*Interpretación, Valoración*), que corresponde a las entradas cada una de las entradas (filas) de la tabla.

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

Para el ejemplo anterior:

```
import Modules.IO_LP exposing (..)
import Modules.SintaxSemanticsLP exposing (..)
import Modules.AuxForLitvis exposing (showTable)
import Modules.AuxiliarFunctions exposing (uncurry)
import Bool.Extra exposing (toString)

f : FormulaLP
f = fromStringToFLP "NOT (NOT (p OR q) OR (NOT r OR s))" |> extractReadFLP

-- Ejecutamos: truthTable f |> truthTableToMDFormat (symbInProp f) |> uncurry (showTable)
```

<i>p</i>	<i>q</i>	<i>r</i>	<i>s</i>	<i>Valoración</i>
0	0	0	0	<i>False</i>
0	0	0	1	<i>False</i>
0	0	1	0	<i>False</i>
0	0	1	1	<i>False</i>

$p$	$q$	$r$	$s$	Valoración
0	1	0	0	<i>False</i>
0	1	0	1	<i>False</i>
0	1	1	0	<i>True</i>
0	1	1	1	<i>False</i>
1	0	0	0	<i>False</i>
1	0	0	1	<i>False</i>
1	0	1	0	<i>True</i>
1	0	1	1	<i>False</i>
1	1	0	0	<i>False</i>
1	1	0	1	<i>False</i>
1	1	1	0	<i>True</i>
1	1	1	1	<i>False</i>

Una vez calculadas todas las posibles interpretaciones resulta sencillo calcular los modelos y contramodelos de una fórmula proposicional, sin más que seleccionar aquellos que sean evaluados como verdaderos o falsos, respectivamente. Tal que:

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

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

De forma que para la fórmula anterior, el conjunto de modelos y contramodelos correspondería a:

```
modelos_f : List Interpretation
modelos_f = models f
```

$\{q, r\}, \{p, r\}, \{p, q, r\}$

```
contramodelos_f : List Interpretation
contramodelos_f = countermodels f
```

$\{\}, \{s\}, \{r\}, \{r, s\}, \{q\}, \{q, s\}, \{q, r, s\}, \{p\}, \{p, s\}, \{p, r, s\}, \{p, q\}, \{p, q, s\}, \{p, q, r, s\}$

Del mismo modo resulta casi trivial decidir la satisfactibilidad y validez de las fórmulas proposicionales, sin más que comprobar si todas las interpretaciones hacen a la fórmula verdadera (*tautología*), alguna de ellas la hace verdadera (*fórmula satisfactible*), o no la hace ninguna de ellas (*fórmula insatisfactible*). Así:

```

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

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

insatisfactibility : FormulaLP -> Bool
insatisfactibility x = List.all (\xs-> not(valuation x xs)) (allInterpretations x)

```

De forma que en el caso de la fórmula anterior, sabemos que es satisfactible ya que el conjunto de modelos no era vacío, y que no es tautología ya que el de contramodelos tampoco lo era. Utilizando las funciones provistas por la librería:

```

isSatisfactible_f : Bool
isSatisfactible_f = satisfactibility f

```

True

```

isTaut_f : Bool
isTaut_f = validity f

```

False

```

isInsatisfactible_f : Bool
isInsatisfactible_f = insatisfactibility f

```

False

## Conjuntos de Fórmulas. Modelos y Consistencia.

De forma análoga a la presentada para las fórmulas proposicionales podemos definir los conceptos anteriores, aplicados a conjuntos de fórmulas de forma que:

- **Modelo.** Se dice que una valoración  $v$  es **modelo** de un conjunto de fórmulas  $U$  si para toda fórmula  $F \in U$  se tiene que  $v(F) = 1$  y se denota por  $v \models U$ . En caso contrario, se dice **contramodelo**.
- **Consistencia** Un conjunto de fórmulas  $U$  se dice **consistente** si existe una valoración  $v$  que es modelo de  $U$ . En caso contrario se dice que  $U$  es **inconsistente**.

### Consecuencia Lógica.

Una fórmula  $F$  es consecuencia lógica (o se sigue) de un conjunto de fórmulas  $U$ , y se denota por  $U \models F$ , si toda valoración que es modelo de  $U$  es también modelo de  $F$ .

Es precisamente este concepto el que permite formular el problema básico en el marco de la lógica proposicional, que planteamos como objetivo de la LP.

### Relación entre consecuencia lógica, consistencia y validez

*PROPOSICIÓN:* Sea  $U = \{F_1, F_2, \dots, F_n\} \subseteq PROP$  y  $F \in PROP$  son equivalentes:

- $(U = \{F_1, F_2, \dots, F_n\}) \models F$
- $\left( \bigwedge_{F_i} U = (F_1 \wedge F_2 \wedge \dots \wedge F_n) \right) \rightarrow F \in TAUT$
- $(U \cup \{\neg F\} = \{F_1, F_2, \dots, F_n, \neg F\}) \models \perp$

## Semántica de Conjuntos LP en Logicus

Al igual que en el caso de las fórmulas proposicionales podemos realizar una traslación directa de los conceptos formalmente expuestos a las implementaciones, de forma que el cálculo de las posibles interpretaciones, análogamente al caso de las fórmulas, se reduce al cálculo de los posibles subconjuntos de los símbolos proposicionales presentes en las fórmulas, correspondientes a la unión de los conjuntos de símbolos asociados a cada una de las fórmulas. De esta forma:

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

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

De forma que el cálculo de los modelos, contramodelos, consistencia e inconsistencia se reduce a aplicar las definiciones sobre el conjunto de fórmulas:

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

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

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

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

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

## —→ Consecuencia lógica

Dada la definición de consecuencia lógica y teniendo en cuenta la relación entre consecuencia lógica, consistencia y validez (último caso), funcionalmente hemos planteado dos desarrollos alternativos para el concepto de Consecuencia Lógica:

(1) Acudiendo a la propia definición. (2) Acudiendo al tercer punto de la proposición anterior.



```

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

```

NOTA: Se ha dejado la segunda como implementación final.

Veámos un ejemplo, dado el conjunto de fórmulas  $U = \{p \rightarrow q, q \rightarrow p \wedge r\}$  y la fórmula  $F = p \rightarrow (p \rightarrow q) \rightarrow r$ , veámos si se cumple  $U \models F$ .

```

import Modules.IO_LP exposing (..)
import Modules.SintaxSemanticsLP exposing (..)
import Bool.Extra exposing (toString)

u1 : FormulaLP
u1 = fromStringToFLP "p IMPLIES q" |> extractReadFLP

u2 : FormulaLP
u2 = fromStringToFLP "q IMPLIES p AND r" |> extractReadFLP

uSet : LPSet
uSet = [u1, u2]

f : FormulaLP
f = fromStringToFLP "p IMPLIES (p IMPLIES q) IMPLIES r" |> extractReadFLP

-- Ejecutamos toString <| isConsequence uSet f

```

True

De hecho, acudiendo a la definición podemos ver que, en efecto, todo modelo del conjunto  $U$  es también modelo de la fórmula  $F$ .

```

modelos_U : List Interpretation
modelos_U = allSetModels uSet

modelos_F : List Interpretation
modelos_F = models f

{-Ejecutamos:
  interpretations2MDFormat modelos_U
  interpretations2MDFormat modelos_F
-}

```

De forma que los modelos de  $U$  corresponden a:

$\{\}, \{r\}, \{p, q, r\}$

Y los de  $F$ :

$\{\}, \{r\}, \{q\}, \{q, r\}, \{p\}, \{p, r\}, \{p, q, r\}$

De forma que, en efecto, todo modelo de  $U$  es también modelo de  $F$ , luego  $F$  se sigue de  $U$ .

# Algoritmos de decisión en LP

Como hemos señalado vamos a presentar de forma somera los algoritmos de decisión en LP, e iremos desarrollando algunos de los algoritmos más importantes a lo largo de los distintos capítulos.

Dado un conjunto de fórmulas  $U$ , un **algoritmo de decisión** para  $U$  es aquél que dada una fórmula  $A \in PROP$ , devuelve **SI** cuando  $A \in U$  y **NO** cuando  $A \notin U$ .

Esto da pie a la definición de algunos problemas con un especial interés:

- $SAT = \{A \in PROP : A \text{ es satisfactible}\}.$
- $TAUT = \{A \in PROP : A \text{ es tautología}\}.$
- Fijado  $U \subseteq PROP$ , la Teoría de  $U$  corresponde a:  $\mathcal{T}(U) = \{A \in PROP : U \models A\}.$

Precisamente, un algoritmo de decisión para  $\mathcal{T}(U)$  proporciona una respuesta al Problema Básico que planteamos al comienzo del capítulo. Por tanto, podemos reducir dicho problema a uno nuevo:

Obtener un algoritmo que, dado un conjunto finito de fórmulas proposicionales,  $U$ , y una fórmula,  $F$ , decida si  $U \models F$ .

Y este a su vez se reduce a comprobar la satisfactibilidad de una cierta fórmula (o bien la validez de otra), hemos aquí el problema conocido como *Problema SAT*.

Notemos que ya hemos visto un algoritmo, el de las Tablas de Verdad, que resuelve, de forma simple, el problema, pero la complejidad de dicho algoritmo es exponencial en el número de símbolos proposicionales, lo que lo hace inabordable para fórmulas de cierta complejidad, incluso computacionalmente.

Hemos de señalar que existen otros algoritmos de decisión del problema *SAT*, algunos de los cuales abordaremos a lo largo de los siguientes capítulos, pero aún no se ha encontrado ninguno capaz de resolver el problema eficientemente (complejidad polinomial), y, de hecho, se duda (fuertemente) de la existencia del mismo. De hecho, determinar la satisfactibilidad de una fórmula proposicional se trata de un problema NP-completo.

## 1.3 De Proposiciones a Predicados. Conceptos Básicos de Lógica de Primer Orden

### Limitaciones de la Lógica Proposicional

Aunque la lógica proposicional posee un semántica sencilla y existen algoritmos de decisión (poco eficientes) para sus problemas básicos, como *SAT* o la consecuencia lógica, la expresividad de LP es bastante limitada, esto hace que muchos problemas no sean modelables en LP, bien porque requieren un gran número de fórmulas o fórmulas de gran tamaño, o

bien porque no puedan ni siquiera expresarse en este lenguaje. El siguiente ejemplo presenta un razonamiento que es válido, sin embargo no es expresable en LP:

1. Todo hombre es mortal.
2. Sócrates es hombre.

3. Por tanto, Sócrates es mortal.

¿Cómo expresar el concepto de ser hombre? ¿Cómo expresar quién es Sócrates?, pero aún más ¿Cómo expresar que todos es mortal?. Es aquí precisamente donde comienza el ámbito de la Lógica de Primer Orden.

## Caracterización de la Lógica de Primer Orden

La Lógica de Primer Orden (LPO) extiende la Lógica Proposicional, ganando capacidad expresiva, que permite abordar cuestiones como:

- Realizar cuantificación sobre los objetos de un dominio, esto es, expresar en qué medida se tiene una propiedad sobre un conjunto de objetos.
- Representar propiedades de los objetos particulares del dominio por medio de predicados y funciones.
- Trabajar con subconjuntos de objetos que pueden venir caracterizados por propiedades que se describen por medio de predicados y funciones.

## 1.4. Fundamentos de la Lógica de Primer Orden

De forma análoga a como se ha visto para la lógica proposicional vamos a estudiar los elementos que definen la Lógica de Primer Orden (Sintaxis y Semántica) dejando para capítulos futuros el desarrollo de los Algoritmos de Decisión.

### Sintaxis de la Lógica de Primer Orden

Formalmente,

La **Lógica de Primer Orden** o **Lógica de Predicados** es un sistema formal diseñado para estudiar los métodos inferenciales en los lenguajes de primer orden.

Un **lenguaje de primer orden** corresponde a un lenguaje formal que consta de:

- Símbolos lógicos (comunes a todos los lenguajes): En los que se engloban:
  - Un conjunto de *Variables*:  $V = \{x, x_0, x_1, \dots, y, y_0, \dots\}$
  - *Conectivas lógicas* :  $\neg$  (negación),  $\wedge$  (conjunción),  $\vee$  (disyunción),  $\rightarrow$  (implicación),  $\leftrightarrow$  (equivalencia).
  - *Cuantificadores*:  $\exists$  (existencial),  $\forall$  (universal).
  - *Símbolos auxiliares*:  $'($  y  $)'$
- Símbolos no lógicos (propios de cada lenguaje): En los que se engloban:
  - Un conjunto de *Constantes*:  $L_C = \{a, b, \dots, a_0, a_1, \dots\}$
  - Un conjunto de *símbolos de función*:  $L_F = \{f_0, f_1, \dots\}$ , cada uno con su aridad correspondiente.
  - Un conjunto de **símbolos de predicado**:  $L_P = \{P_0, P_1, \dots, Q, Q_0, \dots\}$ , cada uno con su aridad correspondiente.

Dos notas:

- Los símbolos de predicado de aridad 0 actúan como símbolos proposicionales.

- El símbolo de igualdad ('=') no es un predicado común a todos los lenguajes de primer orden, pero si es corriente su aparición. La familia de lenguajes que incluyen este predicado es denominada *Lenguajes de Primer Orden con Igualdad*.

Este aparato permite la construcción de distintas expresiones, que componen las fórmulas de LPO. Vamos a exponer una diferenciación de dichas expresiones, distinguiendo *términos* y *fórmulas*.

## Términos en LPO

Los términos se identifican con posibles objetos del mundo. Englobando los siguientes elementos:

- Constantes para hablar de objetos específicos.
- Variables para hablar de objetos genéricos.
- Funciones aplicadas a otros términos más pequeños, según su aridad.

Para ejemplificar los conceptos expuestos a lo largo del tema vamos a modelar el mundo romano: (los superíndices indican la aridad)

$$LR = \{ \underbrace{César, Marco}_{\text{constantes}}, \underbrace{P^1, L^2, O^2, R^1, IA^2}_{\text{símbolos de predicado}}, \underbrace{f^1}_{\text{símbolo de función}} \}$$

tal que, donde, *César* y *Marco* son constantes, *P* y *R* son predicados unarios que denotan *ser pompeyano* y *ser romano*, respectivamente; *L*, *O*, *IA* son predicados binarios que denotan *ser leal a*, *odiar a*, *intentar asesinar a*, respectivamente; y *f* una función unaria que represente el concepto de *padre de*.

De forma que son términos de *LR*:

- *Constantes*: *Marco*, *César*.
- *Variables*: *x*, *y*, *x*<sub>1</sub>, ...
- *Funciones*: *f*(*César*), *f*(*x*), *f*(*f*(*x*)), ...

## Fórmulas en LPO

Las fórmulas se identifican con afirmaciones sobre los objetos del mundo, permitiendo hablar de la veracidad o falsedad de las afirmaciones. Están formadas a partir de predicados sobre términos, y construcciones lógicas de estos predicados (conjunciones, implicaciones, cuantificaciones, etc.). Veámoslo formalmente.

Las fórmulas pueden corresponder a:

- Átomos o Fórmulas atómicas. Corresponden a las expresiones  $p(t_1, t_2, \dots, t_n)$ , tal que *p* es un símbolo de predicado de aridad *n* y *t<sub>i</sub>* son términos.
- Fórmulas no atómicas. Corresponden a expresiones formadas a partir de fórmulas atómicas, mediante el empleo de conectivas y/o cuantificadores.

De esta forma, son fórmulas de *L*:

- Toda fórmula atómica.

- Si  $F$  y  $G$  son fórmulas de  $L$  entonces  $\neg F, F \wedge G, F \vee G, F \rightarrow G, F \leftrightarrow G$ , también son fórmulas de  $L$ .
- Si  $x$  es una variable y  $F$  es una fórmula de  $L$ , entonces  $\exists x F$  y  $\forall x F$  son también fórmulas de  $L$ .

Volviendo al mundo romano, algunas posibles fórmulas del lenguaje  $LR$ :

- Átomos:  $P(\text{César}), L(\text{César}, \text{Marco}), IA(\text{Marco}, f(x))$ .
- Fórmulas compuestas:  $\forall x \exists y L(x, y), \forall x (R(x) \rightarrow (L(x, \text{César}) \vee O(x, \text{César})))$

→ Reglas de simplificación de la notación

Para facilitar la lectura y escritura de las fórmulas vamos a tomar varios criterios de notación:

- Se omitirán los paréntesis externos.
- Las prioridades de las conectivas siguen el mismo orden que el expuesto en LP:  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$  (para la última se recomienda mantener los paréntesis).
- Los cuantificadores tienen prioridad sobre las conectivas.

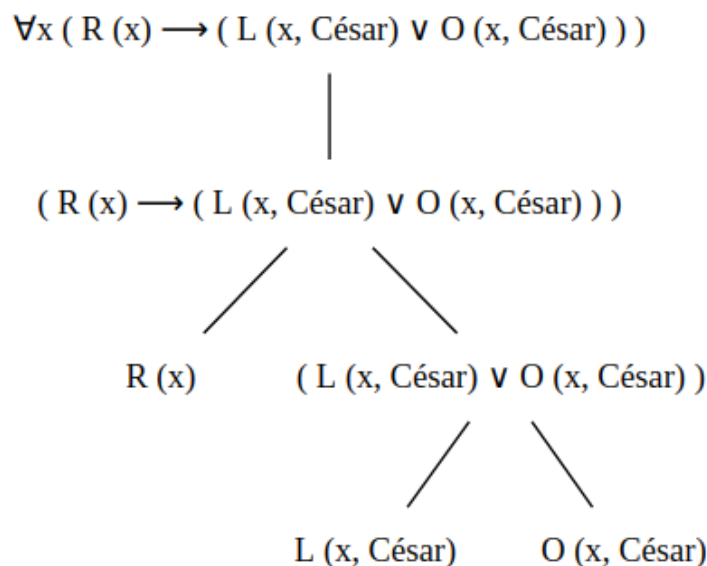
### Árboles de formación.

Al igual que en las fórmulas de LP, la definición de las fórmulas en LPO presenta una estructura recursiva:

- *Caso base.* Fórmulas atómicas.
- *Casos recursivos* Aplicación de las conectivas y cuantificadores sobre fórmulas de LPO.

Y de forma análoga a la presentada en LP, se puede plasmar dicha estructura recursiva en un grafo tipo árbol (esencialmente único), de forma que el nodo raíz corresponde a la fórmula completa y las hojas corresponden a las fórmulas atómicas que participan en la fórmula. Al igual que en LP, toda fórmula que aparezca en algún nodo (ya sea un nodo interno o una hoja) diremos que es **subfórmula** de la fórmula original.

Para la fórmula  $\forall x (R(x) \rightarrow (L(x, \text{César}) \vee O(x, \text{César})))$ , se tendría el AF asociado:



## Sintaxis de Fórmulas LPO en Logicus

Al igual que hicimos con la lógica LP, vamos a exponer cuál es la sintaxis de las fórmulas utilizando la librería Logicus, que nos permitirá definir fórmulas y aplicar diversos algoritmos sobre las mismas.

Bajo la librería podemos definir las fórmulas de dos formas distintas, o bien acudiendo directamente a los constructores de las fórmulas (largo y engorroso), o bien utilizando el Parser (de forma análoga a la escritura natural de las fórmulas).

—→ Definición de fórmulas a partir de constructores (Módulo *SyntaxSemanticsLPO*)

Antes de introducirnos en el estudio de la sintaxis de fórmulas vamos a presentar la sintaxis y definición de términos. Recordemos que la estructura de los términos corresponde a una definición recursiva con variables y constantes como casos *base* y funciones.

Aunque la definición en la librería sigue esta idea, se ha decidido modelar las constantes como funciones independientes de variables, de forma que en la librería se tiene que un término es o bien una variable o bien una función.

```
type Term = Var String
          | Func String (List Term)
```

De forma que las constantes corresponderían a objetos con la estructura *Func String [ ]*.

De forma que podemos definir términos de forma sencilla mediante el uso de estos constructores. Volviendo al ejemplo presentado anteriormente, (*Constantes: Marco, César; Variables:  $x$ ,  $y$ ,  $x_1$ , ...; Funciones:  $f(\text{César})$ ,  $f(x)$ ,  $f(f(x))$* ):

```
import Modules.SyntaxSemanticsLPO exposing (..)

cesar = Func "cesar" []

marco = Func "marco" []

x = Var "x"

x_1 = Var "x_1"

f_cesar = Func "f" [cesar]

f_f_cesar = Func "f" [Func "f" [cesar]]
```

Ahora que ya podemos definir términos en LPO podemos pasar a ver la definición de fórmulas. Si recordamos, las fórmulas están definidas con una estructura recursiva, de forma que las fórmulas corresponden o bien a átomos (*caso base*), o bien a la aplicación de las conectivas y/o cuantificadores sobre otras fórmulas (*caso recursivo*).

Bien, pues la implementación dada en la librería para las fórmulas sigue fielmente dicha definición. De forma que: (definiendo el tipo *Variable* como alias de *Term*)

```

type alias Variable = Term

type FormulaLPO = Pred String (List Term)
                | Equal Term Term
                | Neg FormulaLPO
                | Conj FormulaLPO FormulaLPO
                | Disj FormulaLPO FormulaLPO
                | Impl FormulaLPO FormulaLPO
                | Equi FormulaLPO FormulaLPO
                | Exists Variable FormulaLPO
                | Forall Variable FormulaLPO
                | Insat

```

De forma que esto nos permite expresar sintácticamente todas las fórmulas en LPO. Para los ejemplos expuestos:

```

import Modules.SintaxSemanticsLPO exposing (..)
import Modules.IO_LPO exposing (..)

```

```

f1 : FormulaLPO
f1 = Pred "P" [Func "cesar" []]

```

$$P(\textit{cesar})$$

```

f2 : FormulaLPO
f2 = Pred "L" [Func "cesar" [], Func "marco" []]

```

$$L(\textit{cesar}, \textit{marco})$$

```

f3 : FormulaLPO
f3 = Pred "IA" [Func "cesar" [], Func "f" [Var "x"]]

```

$$IA(\textit{cesar}, f(x))$$

```

f4 : FormulaLPO
f4 = Forall (Var "x") (Exists (Var "y") (Pred "L" [Var "x", Var "y"]))

```

$$\forall x \exists y L(x, y)$$

```

f5 : FormulaLPO
f5 = Forall (Var "x") (Impl (Pred "R" [Var "x"]) (Disj (Pred "L" [Var "x", Func "cesar" []]) (Pred

```

$$\forall x (R(x) \longrightarrow (L(x, \textit{cesar}) \vee O(x, \textit{cesar})))$$

—→ Definición de fórmulas LPO con Parser (Módulo IO LPO).

Aunque ya podemos definir todas las fórmulas en LPO, su escritura resulta una tarea muy pesada, para aliviar esa carga se ha desarrollado un Parser que permite la lectura de fórmulas a partir de una cadena de texto en el que las fórmulas se escriben de forma análoga a como se dan en el lenguaje formal de la lógica. Se establecen algunas reglas sintácticas en su uso:

- Las variables se escriben con una letra en minúscula seguidas de caracteres alfanuméricos o el símbolo '\_'. Algunos ejemplos: 'x', 'x\_1', 'x1', ...
- Las funciones se definen siguiendo la notación prefija, según el patrón:

*[Símbolo de función] [parámetros]*

Tal que:

- El *símbolo de función* debe comenzar por el carácter '\_', seguido de una serie de caracteres, entre estos se admiten caracteres alfanuméricos y también símbolos, excepto '[', ']', '(', ')', '{', '}', ';'.
- En los *parámetros* se pueden dar 2 casos, o bien la función corresponde a una constante y por tanto no tiene parámetros, con lo cual bastaría escribir el símbolo de función (ejemplos: '\_a', '\_PEDRO', '\_1') o bien es una función (dependiente de, al menos, un término), en tal caso los argumentos se dan en una lista acotada por corchetes ('[', ']') y tras cada argumento ha de ir un ';' (ejemplos: '\_f[x;]', '\_g[x;y;]', '\_+[x;\_1;\_·[y;\_2;]]').
- Los predicados se definen utilizando la notación prefija según el patrón:

*[símbolo de predicado] [parámetros]*

Tal que:

- El *símbolo de predicado* debe comenzar por un carácter en mayúscula o un símbolo (exceptuando '\_', '!', '(', ')', '[', ']', '{', '}', ')') (también se admiten dígitos numéricos pero no se recomienda), seguido de una serie de caracteres, entre estos se admiten caracteres alfanuméricos y también símbolos (exceptuando '(', ')', '[', ']', '{', '}', ')'). Son palabras reservadas (y por tanto no se pueden usar como predicados) "NOT", "AND", "OR", "IMPLIES", "EQUIV", "EXISTS", "FORALL", "INSAT"
- En los *parámetros* se pueden dar 2 casos, o bien la función es un predicado proposicional (no tiene parámetros), con lo cual bastaría escribir el símbolo de predicado (ejemplos: 'P', 'Q\_1') o bien es un predicado n-ario (dependiente de, al menos, un parámetro), en tal caso los argumentos se dan en una lista acotada por corchetes ('[', ']') y tras cada argumento ha de ir un ';' (ejemplos: 'P[x;]', '>=[x;y;]', 'MAX[x;\_1;\_·[y;\_2;]]').

Hay una excepción, *predicado de igualdad* (binaria) se utilizará de forma infija separando los términos con el símbolo '=' (ejemplos: 'x=y', 'x\_4=\_3', '\_f[x;]=\_f[y;]') (El predicado de igualdad es de uso exclusivamente binario).

*[término] = [término]*



- Los cuantificadores se expresan de forma análoga al lenguaje formal de LPO según el patrón:  
 $[símbolo\ de\ cuantificador] [variable] [fórmula]$

Tal que:

- Los *símbolos de cuantificadores* corresponden a:

<u>Cuantificador</u>	<u>Símbolo Logicus</u>
Existencial ( $\exists$ )	'EXISTS'
Universal ( $\forall$ )	'FORALL'

- La *variable* debe ir entre llaves ('{', '}'), respetando las reglas establecidas para la definición de variables.
- La *fórmula* escrita respetando los criterios dados para la escritura de las fórmulas.
- Las conectivas se usarán de forma infija (salvo para la *negación*), utilizando los siguientes símbolos, manteniéndose la prioridad de las conectivas definida (en el orden de prioridad descendente expuesto en la tabla).

<u>Conectiva</u>	<u>Símbolo Logicus</u>
Negación ( $\neg$ )	'NOT'
Conjunción ( $\wedge$ )	'AND'
Disyunción ( $\vee$ )	'OR'
Implicación ( $\rightarrow$ )	'IMPLIES'
Equivalencia ( $\leftrightarrow$ )	'EQUIV'

- Los paréntesis se utilizan de igual forma en que se han definido en el lenguaje formal de la lógica proposicional, con los símbolos '(' y ')'. No son necesarios los paréntesis externos de las fórmulas.
- En caso de uso repetido de una misma conectiva y/o cuantificador, se realizará asociación por la derecha.

Al igual que ocurría en el Parser de LP una vez leída la fórmula con la función *fromStringToFLPO* es necesario extraerla utilizando la función *extractReadFLPO*.

Vamos a mostrar algunos ejemplos de definición de las fórmulas utilizando el Parser de LPO.

- Ejemplo 1:  $\forall x(R(x) \rightarrow (L(x, \textit{César}) \vee O(x, \textit{César})))$

```
import Modules.SintaxSemanticsLPO exposing (..)
import Modules.IO_LPO exposing (..)

f1 : FormulaLPO
f1 = fromStringToFLPO "FORALL{x}(R[x;] IMPLIES (L[x;_cesar;] OR O[x; _cesar;]))" |> extractReadFLF

-- Ejecutamos (toLatexFLPO f1) para mostrar la fórmula
```

Que corresponde a:

$$\forall x (R(x) \longrightarrow (L(x, \textit{cesar}) \vee O(x, \textit{cesar})))$$

- Ejemplo 2 (tomando el lenguaje matemático), definamos la propiedad biyectiva. Para expresarla consideremos el siguiente lenguaje:  $L = \{O, I, f\}$ , tal que:
  - $O$  es un predicado de aridad 1 que expresa si el objeto pertenece al conjunto origen.
  - $I$  es un predicado de aridad 1 que expresa si el objeto pertenece al conjunto imagen.
  - $f$  es una función que dado un elemento del conjunto origen obtiene el correspondiente elemento del conjunto imagen y dado un elemento del conjunto imagen obtiene el correspondiente del conjunto origen.

La propiedad biyectiva se formula como:

*Una relación es biyectiva si para todo elemento del conjunto origen existe un único elemento del conjunto imagen con el que está relacionado y todo elemento del conjunto imagen está relacionado con algún elemento del origen*

Esto es:

$$\forall x, y \in O (x \neq y \rightarrow f(x) \neq f(y)) \wedge \forall x \in I \exists y \in O (f(x) = y)$$

```
biyectiva : FormulaLPO
biyectiva = fromStringToFLPO
            "FORALL{x} FORALL{y} (O[x;] AND O[y;] AND NOT (x=y) IMPLIES NOT (_f[x;]=_f[y;])"
            |> extractReadFLPO

-- Ejecutamos (toLatexFLPO biyectiva) para mostrar la fórmula
```

$$(\forall x \forall y ((O(x) \wedge (O(y) \wedge \neg(x = y))) \longrightarrow \neg(f(x) = f(y))) \wedge \forall x (I(x) \longrightarrow \exists y (O(y) \wedge (f(y) = x))))$$

→ Árboles de formación LPO en Logicus

La librería también permite la representación de los árboles de formación LPO. La función *formtree* (del módulo *IO\_LPO*) muestra la representación del árbol de formación en formato texto DOT, de forma análoga a como lo hacía en el caso de LP

Por ejemplo para la fórmula  $\forall x(R(x) \rightarrow (L(x, \textit{César}) \vee O(x, \textit{César})))$ :

```
import Modules.IO_LPO exposing (..)
import Modules.SyntaxSemanticsLPO exposing (..)
import Modules.AuxForLitvis exposing (showGraphViz)

f : FormulaLPO
f = fromStringToFLPO "FORALL{x}{R[x;] IMPLIES (L[x;_cesar;] OR O[x; _cesar;])}" |> extractReac

ft_f : String
ft_f = formTree f
```

De forma que se obtiene:

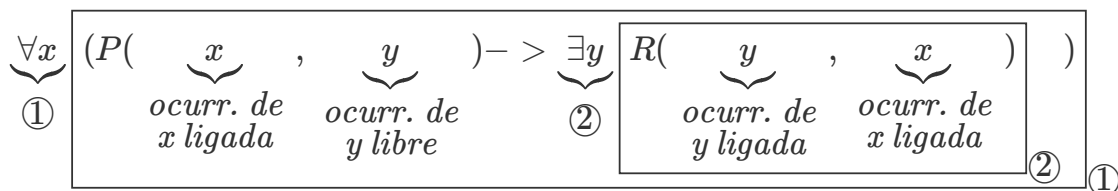
Árbol de formación de  $\forall x(R(x) \rightarrow (L(x, \textit{César}) \vee O(x, \textit{César})))$

### Alcance de los cuantificadores.

Hemos estudiado la sintáxis formal de las fórmulas, pero aún faltan algunos detalles por completar para establecer qué fórmulas están bien formadas y cuáles no. Un tema importante es el tratamiento de la cuantificación, esto es, a qué apariciones u ocurrencias (denominadas estancias) de una variable afecta un cuantificador.

Una ocurrencia de una variable está afectada por un cuantificador (se dice que es una **estancia u ocurrencia ligada**) si hay un cuantificador sobre dicha variable actuando sobre la (sub)fórmula que la contiene. En otro caso diremos que se trata de una **estancia u ocurrencia libre**

Para comprenderlo mejor vamos a explicar un ejemplo:



En la fórmula podemos apreciar que el cuantificador,  $\forall x$ , afecta a toda la fórmula siguiente, por tanto todas las ocurrencias de  $x$  que aparezcan en la fórmula serán ocurrencias ligadas. Sin embargo,  $P(x, y)$  no está afectado por ningún otro cuantificador, por tanto, la ocurrencia de  $y$  en  $P(x, y)$  es libre. Por otra parte,  $R(x, y)$  sí está afectado por el cuantificador  $\exists y$ , por tanto tanto la ocurrencia de  $x$  como la de  $y$  en  $R(y, x)$  son ligadas.

De forma similar se definen las variables *libres* y *ligadas* como:

Una variable  $x$  se dice **variable libre** en una fórmula  $F$  si existe alguna ocurrencia libre de  $x$  en  $F$ .

Una variable  $x$  se dice **variable ligada** en una fórmula  $F$  si existe alguna ocurrencia ligada de  $x$  en  $F$ .

De forma que en una fórmula una variable puede ser al mismo tiempo *variable libre* y *variable ligada*.

Por ejemplo, en la fórmula anterior  $x$  es una variable *exclusivamente cerrada* (todas sus ocurrencias son ligadas), sin embargo  $y$  es una variable *libre y ligada* (hay una ocurrencia libre y otra ligada).

En base a los conceptos anteriores se definen los conceptos:

Se dice que un **término** es **cerrado** si no contiene ninguna variable.

Por ejemplo:  $César$ ,  $f(Marco)$ ,  $f(f(César))$ ; son términos cerrados en el lenguaje  $LR$ .

Se dice que una **fórmula** es **cerrada** si no contiene variables libres, o equivalentemente si todas las estancias de todas las variables son ligadas.

Por ejemplo las fórmulas  $\forall x(L(x, César) \vee O(César, x))$ ,  $\forall xL(x, César) \rightarrow \neg \exists yO(César, y)$  son una *fórmulas cerradas*, mientras que las fórmulas  $\forall xL(x, César) \vee O(César, x)$ ,  $\forall xL(x, César) \rightarrow \neg \exists yIA(y, x)$  no lo son.

Se dice que una **fórmula** es **abierta** si no contiene cuantificadores, esto es todas las variables son, exclusivamente, *variables libres*.

Por ejemplo son fórmulas abiertas  $P(x)$ ,  $R(César)$ ,  $P(x) \leftrightarrow \neg IA(f(x), Marco)$ .

Los algoritmos de decisión que vamos a estudiar trabajan únicamente sobre fórmulas cerradas, por lo que hemos de *clausurar* todas aquellas fórmulas que no sean cerradas, cuantificando las variables libres de la fórmula.

<i>Continuará</i>
-------------------