# Deducción automática

(Vol. 1: Construcción lógica de sistemas lógicos)

José A. Alonso Jiménez y Joaquín Borrego Díaz
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

30 de diciembre de 2002

# Índice General

**Prólogo** **6**

**1 Prólogo** **7**

**2 Introducción a la programación lógica con Prolog** **9**

2.1 El sistema deductivo de Prolog . . . . . . . . . . . . . . . . . . 9
2.1.1 Deducción Prolog en lógica proposicional . . . . . . . . 9
2.1.2 Deducción Prolog en lógica relacional . . . . . . . . . . 14
2.1.3 Deducción Prolog en lógica funcional . . . . . . . . . . . 15
2.2 Listas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2.1 Representación de listas . . . . . . . . . . . . . . . . . . 20
2.2.2 Concatenación de listas . . . . . . . . . . . . . . . . . . 21
2.2.3 La relación de pertenencia . . . . . . . . . . . . . . . . . 23
2.3 Disyunciones . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.4 Operadores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.5 Aritmética . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.6 Control mediante corte . . . . . . . . . . . . . . . . . . . . . . 27
2.7 Negación . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.8 El condicional . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.9 Predicados sobre tipos de término . . . . . . . . . . . . . . . . 35
2.10 Comparación y ordenación de términos . . . . . . . . . . . . . 36
2.11 Procesamiento de términos . . . . . . . . . . . . . . . . . . . . 38
2.12 Procedimientos aplicativos . . . . . . . . . . . . . . . . . . . . 40
2.13 Todas las soluciones . . . . . . . . . . . . . . . . . . . . . . . . 41

**3 Elementos de lógica proposicional** **43**
3.1 Sintaxis de la lógica proposicional . . . . . . . . . . . . . . . . . 43
3.2 Valores de verdad . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.3 Funciones de verdad . . . . . . . . . . . . . . . . . . . . . . . . 45
3.4 Valor de una fórmula en una interpretación . . . . . . . . . . . 46
3.5 Interpretaciones principales de una fórmula . . . . . . . . . . . 47
3.6 Modelo de una fórmula . . . . . . . . . . . . . . . . . . . . . . . 49
3.7 Cálculo de los modelos de una fórmula . . . . . . . . . . . . . . 50
3.8 Satisfacibilidad . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.9 Satisfacibilidad con MACE . . . . . . . . . . . . . . . . . . . . 52
3.10 Contramodelo de una fórmula . . . . . . . . . . . . . . . . . . . 53
3.11 Validez. Tautologías . . . . . . . . . . . . . . . . . . . . . . . . 53
3.12 Satisfacibilidad y validez . . . . . . . . . . . . . . . . . . . . . . 54
3.13 Interpretaciones principales de un conjunto de fórmulas . . . . 54
3.14 Modelo de un conjunto de fórmulas . . . . . . . . . . . . . . . . 56
3.15 Cálculo de modelos de conjuntos de fórmulas . . . . . . . . . . 56
3.16 Consistencia de un conjunto de fórmulas . . . . . . . . . . . . . 57
3.17 Consistencia con MACE . . . . . . . . . . . . . . . . . . . . . . 58
3.18 Consecuencia lógica . . . . . . . . . . . . . . . . . . . . . . . . 58
3.19 Consecuencia lógica e inconsistencia . . . . . . . . . . . . . . . 59
3.20 Consecuencia lógica con MACE . . . . . . . . . . . . . . . . . . 60
3.21 Aplicaciones del razonamiento proposicional . . . . . . . . . . . 61

**4 Sistemas deductivos proposicionales** **75**
4.1 Tableros semánticos . . . . . . . . . . . . . . . . . . . . . . . . 75
4.1.1 Ejemplos de demostraciones por tableros semánticos . . 75
4.1.2 Notación uniforme . . . . . . . . . . . . . . . . . . . . . 76
4.1.3 Procedimiento de completación de tableros . . . . . . . 78
4.1.4 Tableros cerrados . . . . . . . . . . . . . . . . . . . . . . 81
4.1.5 Teorema por tableros . . . . . . . . . . . . . . . . . . . 81
4.1.6 Deducción por tableros . . . . . . . . . . . . . . . . . . 82
4.2 Cláusulas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.2.1 Equivalencia lógica . . . . . . . . . . . . . . . . . . . . . 84
4.2.2 Forma normal negativa . . . . . . . . . . . . . . . . . . 84
4.2.3 Forma normal conjuntiva . . . . . . . . . . . . . . . . . 86
4.2.4 Transformación a cláusulas . . . . . . . . . . . . . . . . 88
4.2.5 Transformación a cláusulas con OTTER . . . . . . . . . 90

4.3 Resolución . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
4.3.1 Motivación de resolución . . . . . . . . . . . . . . . . . . . 91
4.3.2 Regla de resolución proposicional . . . . . . . . . . . . . 92
4.3.3 Demostraciones por resolución . . . . . . . . . . . . . . 93
4.3.4 Procedimiento elemental de búsqueda de refutación . . . 94
4.3.5 Resolución con OTTER . . . . . . . . . . . . . . . . . . . 95
4.3.6 Resolución de OTTER en Prolog . . . . . . . . . . . . . 99

**Bibliografía** **103**

# Capítulo 1

# Prólogo

Este libro constituye el primer volumen de una serie sobre deducción automática. Su objetivo es la presentación de Prolog como un sistema de deducción automática y la construcción en Prolog de sistemas de deducción proposicional.

En el primer capítulo presentamos la programación lógica en Prolog como una aplicación de la deducción automática a la vez que se introducen los conocimientos de Prolog necesarios en los siguientes capítulos del libro.

En el segundo capítulo se formaliza en Prolog los conceptos básicos de la lógica proposicional y se muestra cómo pueden resolverse algunos problemas con OTTER y MACE.

En el tercer capítulo se construyen en Prolog distintos cálculos lógicos proposicionales.

En los siguientes volúmenes continuaremos la construcción de sistemas lógicos (para la lógica de primer orden con igualdad), estudiaremos el razonamiento asistido por ordenador (con OTTER, MACE y ACL2) y aplicaciones del razonamiento automático.

El material de este volumen y los siguientes forma parte de cursos impartidos por los autores en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla. Se puede acceder a dichos cursos en la Red a través de `http://www.cs.us.es/~jalonso`.

# Capítulo 2

# Introducción a la programación lógica con Prolog

En este capítulo presentamos el lenguaje de programación lógica Prolog con un doble objetivo: como una aplicación de la deducción automática y como soporte para la construcción de los sistemas lógicos que realizaremos en los siguientes capítulos.

Los textos fundamentales de Prolog son [2], [5], [10] y [11].

## 2.1 El sistema deductivo de Prolog

En esta sección vamos a presentar el procedimiento básico de deducción de Prolog: la resolución SLD. La presentación la haremos ampliando sucesivamente la potencia expresiva del lenguaje considerado.

### 2.1.1 Deducción Prolog en lógica proposicional

En esta sección vamos a estudiar el sistema deductivo de Prolog en el caso de la lógica proposicional. Vamos a desarrollar el estudio mediante el siguiente ejemplo.

**Ejemplo 2.1.1 [Problema de clasificación de animales]**
*Disponemos de una base de conocimiento compuesta de reglas sobre clasificación de animales y hechos sobre características de un animal.*

- *Regla 1: Si un animal es ungulado y tiene rayas negras, entonces es una cebra.*
- *Regla 2: Si un animal rumia y es mamífero, entonces es ungulado.*
- *Regla 3: Si un animal es mamífero y tiene pezuñas, entonces es ungulado.*
- *Hecho 1: El animal tiene es mamífero.*
- *Hecho 2: El animal tiene pezuñas.*
- *Hecho 3: El animal tiene rayas negras.*

*Demostrar a partir de la base de conocimientos que el animal es una cebra.*

**Demostración:** Una forma de demostrarlo es razonando hacia atrás. El problema inicial consiste en demostrar que el animal es una cebra. Por la regla 1, el problema se reduce a demostrar que el animal es ungulado y tiene rayas negras. Por la regla 3, el problema se reduce a demostrar que el animal es mamífero, tiene pezuñas y tiene rayas negras. Por el hecho 1, el problema se reduce a demostrar que el animal tiene pezuñas y tiene rayas negras. Por el hecho 2, el problema se reduce a demostrar que el animal tiene rayas negras. Que es cierto por el hecho 3. □

Para resolver el problema anterior con Prolog tenemos que considerar las siguientes cuestiones: (1) cómo se representan las reglas, (2) cómo se representan los hechos, (3) cómo se representan las bases de conocimientos en Prolog, (4) cómo se inicia una sesión Prolog, (5) cómo se carga la base de conocimiento, (6) cómo se plantea el objetivo a demostrar y (7) cómo se interpreta la respuesta. Una vez resuelto, se plantean las siguientes cuestiones: (8) cómo ha realizado Prolog la búsqueda de la demostración, (9) cuál la demostración obtenida y (10) como se corresponde dicha demostración con la anteriormente presentada.

Para representar una regla, se empieza por elegir los símbolos para los átomos que aparecen en la regla. Para la regla 1, podemos elegir los símbolos `es_ungulado`, `tiene_rayas_negras` y `es_cebra`. La regla 1 puede representarse como

**Si** `es_ungulado` **y** `tiene_rayas_negras` **entonces** `es_cebra`

Usando las conectivas lógicas la expresión anterior se escribe mediante la fórmula

`es_ungulado` ∧ `tiene_rayas_negras` → `es_cebra`

donde ∧ representa la conjunción y →, el condicional. La fórmula anterior se representa en Prolog, mediante la cláusula

`es_cebra :- es_ungulado, tiene_rayas_negras.`

Se puede observar que la transformación ha consistido en invertir el sentido de la escritura y sustituir las conectivas por `:-` (condicional inversa) y `,` (conjunción). El átomo a la izquierda de `:-` se llama la cabeza y los átomos a la derecha se llama el cuerpo de la regla.

Para representar los hechos basta elegir los símbolos de los átomos. Por ejemplo, el hecho 2 se representa en Prolog por

`tiene_rayas_negras.`

es decir, el símbolo del átomo terminado en un punto. Los hechos pueden verse como cláusulas con el cuerpo vacío.

Para representar la base de conocimiento en Prolog, se escribe en un fichero (por ejemplo, `animales.pl`) cada una de las reglas y los hechos [1].

```
es_cebra    :- es_ungulado, tiene_rayas_negras.  % Regla 1
es_ungulado :- rumia, es_mamífero.               % Regla 2
es_ungulado :- es_mamífero, tiene_pezuñas.       % Regla 3
es_mamífero.                                     % Hecho 1
tiene_pezuñas.                                   % Hecho 2
tiene_rayas_negras.                              % Hecho 3
```

Al lado de cada regla y de cada hecho se ha escrito un comentario (desde `%` hasta el final de la línea).

Para iniciar una sesión de Prolog (con SWI Prolog) se usa la orden `pl`. La base de conocimiento se carga en la sesión Prolog escribiendo el nombre entre corchetes y terminado en un punto. La pregunta se plantea escribiendo el átomo y un punto.

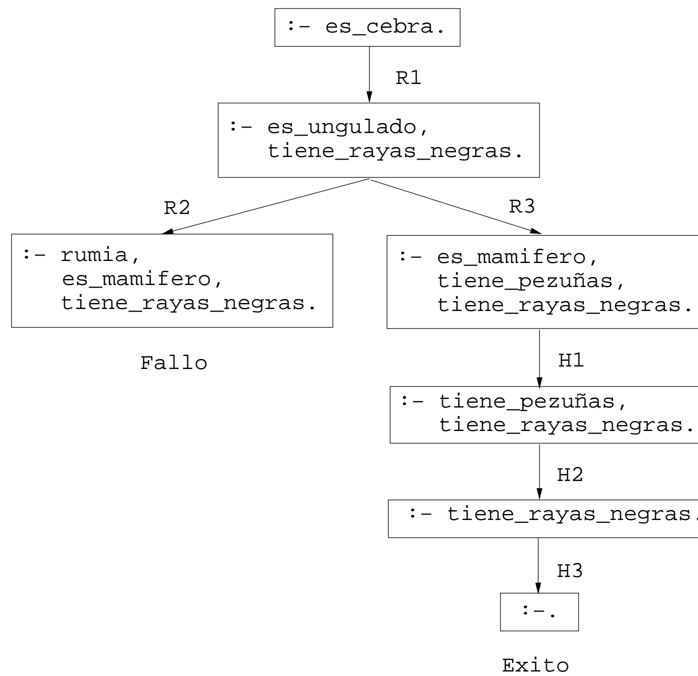
```
> pl
Welcome to SWI-Prolog (Version 5.0.3)
Copyright (c) 1990-2002 University of Amsterdam.

?- [animales].
```

[1] En la versión 5.0 de SWI Prolog, para que no dé error en los predicados no definidos, hay que añadirle la línea `:- set_prolog_flag(unknown,warning).`

```
Yes

?- es_cebra.
Yes
```

La respuesta `Yes` significa que ha demostrado que el animal es una cebra.

Podemos ver cómo Prolog ha obtenido la demostración mediante el árbol de deducción:

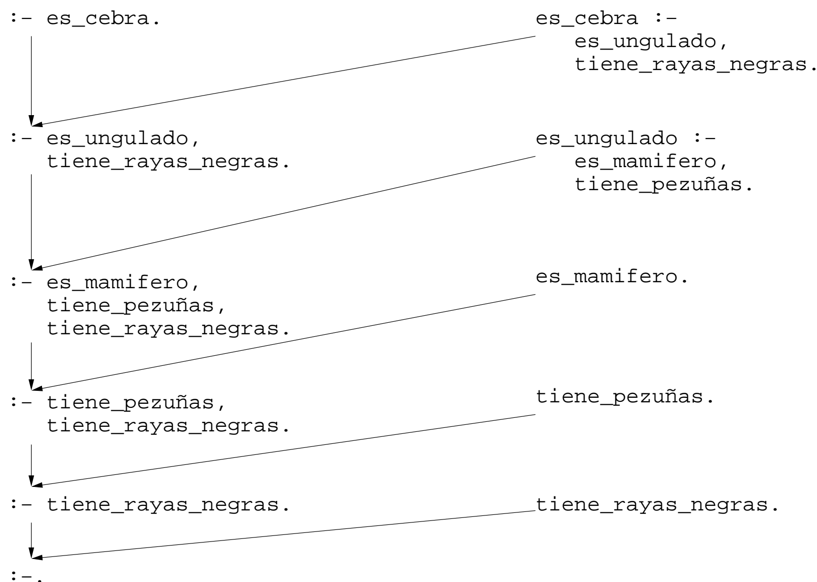



La búsqueda de la prueba es una búsqueda en profundidad en un espacio de estados, donde cada estado es una pila de problemas por resolver. En nuestro ejemplo, el estado inicial consta de un único problema (`es_cebra`). Buscamos en la base de hechos una cláusula cuya cabeza coincida con el primer problema de la pila, encontrando sólo la regla 1. Sustituimos el problema por el cuerpo de la regla. dando lugar a la pila `es_ungulado, tiene_rayas_negras`. Para el primer problema tenemos dos reglas cuyas cabezas coinciden (las reglas 2 y 3). Consideramos en primer lugar la regla 2, produciendo la pila de problemas `rumia, es_mamífero, tiene_rayas_negras`. El primer problema no coincide

con la cabeza de ninguna cláusula. Se produce un fallo y se reconsidera la elección anterior. Consideramos ahora la cláusula 3, produciendo la pila de problemas `es_mamífero, tiene_pezuñas, tiene_rayas_negras`. Cada uno de los problemas restantes coincide con uno de los hechos, con lo que obtenemos una solución del problema inicial.

Podemos observar que el árbol tiene dos ramas: una rama de fallo (su hoja es no vacía y su primer átomo no coincide con la cabeza de ninguna regla) y una rama de éxito (su hoja es vacía).

A partir de la rama de éxito podemos extraer la siguiente demostración (por resolución SLD):




Leída en sentido contrario, y con notación lógica, se obtiene la siguiente demostración

| | | |
|---|---|---|
| 1 | `tiene_rayas_negras` | Hecho 3 |
| 2 | `tiene_pezuñas` | Hecho 2 |
| 3 | `es_mamífero` | Hecho 1 |
| 4 | `es_ungulado` | Regla 3 y líneas 3 y 2 |
| 5 | `es_cebra` | Regla 1 y líneas 4 y 1 |

### 2.1.2 Deducción Prolog en lógica relacional

En esta sección vamos a ampliar la presentación del sistema deductivo de Prolog a la lógica relacional (con variables, cuantificadores, constantes y símbolos de relación). A lo largo de la sección se mostrará el uso de la unificación.

La presentación se basará en la siguiente base de conocimientos:

- Hechos: 6 y 12 son divisibles por 2 y por 3.
- Hecho: 4 es divisible por 2.
- Regla: Los números divisibles por 2 y por 3 son divisibles por 6.

Para representar la base de conocimiento usaremos las constantes `2`, `3`, `6` y `12` y el predicado binario `divide`, entendiendo que se verifica si el primer argumento divide al segundo. Los hechos se representan mediante 4 cláusulas unitarias. La regla, se puede expresar como *para todo X: si X es divisible por 2 y X es divisible por 3, entonces X es divisible por 6.* La representación lógica de la regla es

$$(\forall X)[\text{divide}(2, X) \wedge \text{divide}(3, X) \rightarrow \text{divide}(6, X)]$$

y su representación Prolog es

```
divide(6,X) :- divide(2,X), divide(3,X).
```

en la que observamos que aparece la variable `X` (en Prolog se consideran variables las palabras que empiezan por mayúscula) y que está universalmente cuantificada de manera implícita. La representación en Prolog de la base de conocimientos es

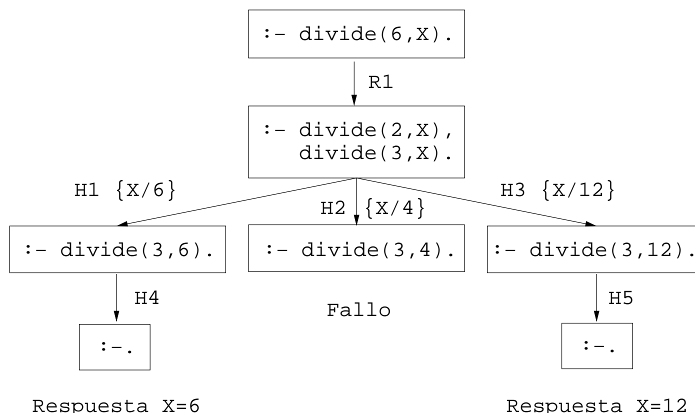
```
divide(2,6).                                  % Hecho 1
divide(2,4).                                  % Hecho 2
divide(2,12).                                 % Hecho 3
divide(3,6).                                  % Hecho 4
divide(3,12).                                 % Hecho 5
divide(6,X) :- divide(2,X), divide(3,X).      % Regla 1
```

Usando la base de conocimiento podemos determinar los números divisibles por `6` como se muestra a continuación

```
?- divide(6,X).
X = 6 ;
X = 12 ;
No
```

Después de obtener la primera respuesta se determina otra pulsando punto y coma. Cuando se intenta buscar otra responde `No` que significa que no hay más respuestas.

El árbol de deducción correspondiente a la sesión anterior es




Podemos observar en el árbol dos ramas de éxito y una de fallo. Además, el paso entre objetivos se ha ampliado: no se exige que el primer objetivo sea igual que la cabeza de una cláusula, sino que sea unificable (es decir, que exista una sustitución que los haga iguales); por ejemplo, en el segundo paso el objetivo `divide(2,X)` se unifica con el hecho `divide(2,6)` mediante la sustitución de `X` por `6` (representada por `{X/6}`. Componiendo las sustituciones usadas en una rama de éxito se obtiene la respuesta.

### 2.1.3 Deducción Prolog en lógica funcional

En esta sección volvemos a ampliar la presentación del sistema deductivo de Prolog al caso de la lógica funcional (con símbolos de función). Además, presentaremos el primer ejemplo de definición recursiva y detallaremos el cálculo de unificadores.

Los números naturales se pueden representar mediante una constante `0` y un símbolo de función unitaria `s` que representan el cero y el sucesor respectivamente. De esta forma, `0`, `s(0)`, `s(s(0))`, .... representan a los números naturales $0, 1, 2, \ldots$. Vamos a definir la relación `suma(X,Y,Z)` que se verifique si `Z` es la suma de los números naturales `X` e `Y` con la anterior notación. La definición, por recursión en el primer argumento, se basa en las identidades

```
0 + Y = Y
s(X) + Y = s(X+Y)
```

que se traduce en las fórmulas

$$(\forall Y)[\text{suma}(0, Y, Y)]$$
$$(\forall X, Y, Z)[\text{suma}(X, Y, Z) \rightarrow \text{suma}(s(X), Y, s(Z))]$$

y éstas en el programa Prolog

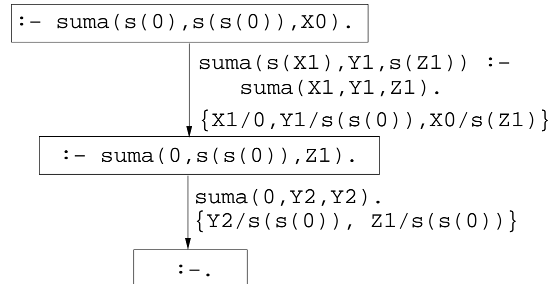
```
suma(0,Y,Y).                             % R1
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).        % R2
```

Vamos a usar el programa para responder a distintas siguientes cuestiones y explicar cómo se obtienen las respuestas.

La primera cuestión consiste en calcular la suma de `s(0)` y `s(s(0))`. La forma de plantear la cuestión en Prolog y la respuesta obtenida es

```
?- suma(s(0),s(s(0)),X).
X = s(s(s(0)))
Yes
```

El árbol de deducción es




```
Resp.: X = X0 = s(Z1) = s(s(s(0)))
```

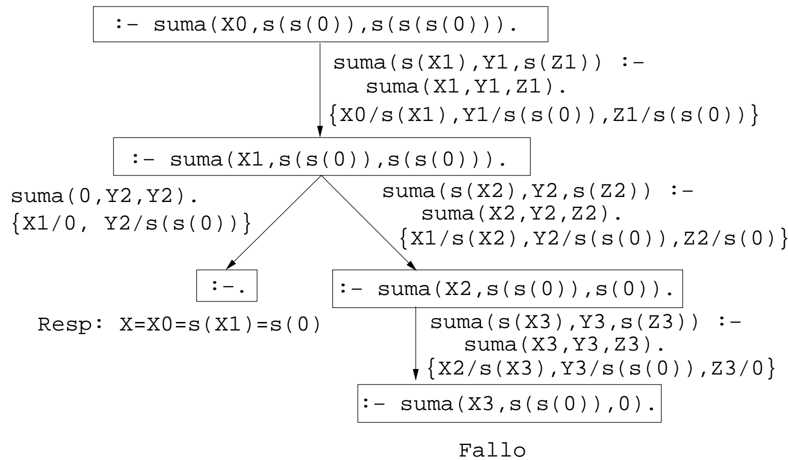
Del árbol vamos a comentar la separación de variables, las unificaciones y el cálculo de la respuesta. Para evitar conflicto con las variables, se cambia de nombre añadiendo el índice a las del objetivo inicial y para las cláusulas del programa se añade como índice el nivel del árbol. El nodo inicial sólo tiene un sucesor con la regla 2, porque es la cabeza de la única regla con la que unifica; efectivamente el átomo `suma(s(0),s(s(0)),X0)` no es unificable

con `suma(0,Y1,Y1)` porque los primeros argumentos son átomos sin variables distintos y sí es unificable con `suma(s(X1),Y1,s(Z1)` mediante la sustitución `{X1/0, Y1/s(s(0)), X0/s(Z1)}` que aplicada a ambos átomos da el átomo `suma(s(0),s(s(0)),s(Z1))`. Lo mismo sucede con el segundo nodo. Finalmente, la respuesta se calcula componiendo las sustituciones realizadas en la rama de éxito a las variables iniciales: `X` se sustituye inicialmente por `X0`, en el primer paso se sustituye `X0` por `s(Z1)` y en el segundo se sustituye `Z1` por `s(s(0))` con lo que el valor por el que sustituye `X` es `s(s(s(0)))`.

La segunda cuestión es cómo calcular la resta de `s(s(s(0)))` y `s(s(0))`. Para ello no es necesario hacer un nuevo programa, basta con observar que $X = a - b \leftrightarrow X + a = b$ y plantear la pregunta

```
?- suma(X,s(s(0)),s(s(s(0)))).
X = s(0) ;
No
```

En el árbol de deducción es



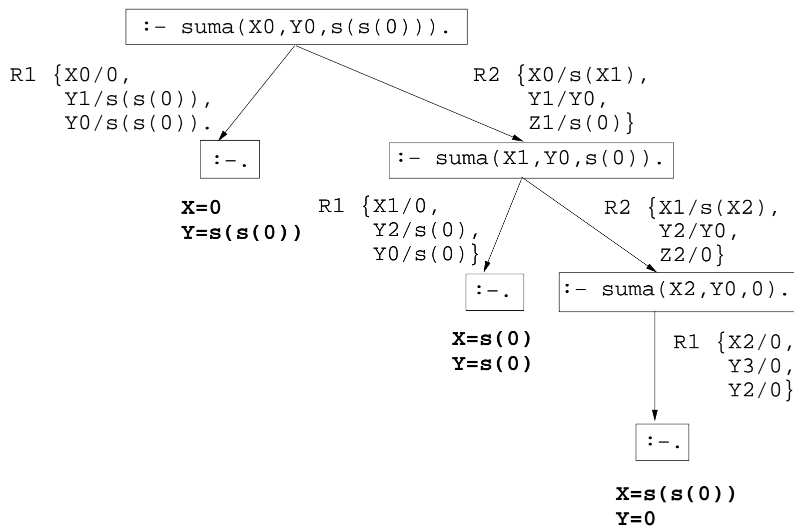

en el que se observa que al intentar obtener una segunda respuesta se produce una rama de fallo, ya que el último objetivo de la segunda rama no es unificable con la cabeza de la primera cláusula (porque el segundo y tercer argumentos del objetivo son términos sin variables distintos) ni con la de la segunda (porque los terceros argumentos son términos sin variables distintos).

La tercera cuestión es descomponer el número 2 en suma de dos números naturales; es decir resolver la ecuación `X + Y = 2`. Tampoco para este problema

se necesita un nuevo programa, basta realizar la siguiente consulta

```
?- suma(X,Y,s(s(0))).
X = 0
Y = s(s(0)) ;
X = s(0)
Y = s(0) ;
X = s(s(0))
Y = 0 ;
No
```

con la que se obtienen las tres soluciones `2=0+2=1+1=2+0`. El árbol de deducción es




Vamos a comentar la unificación de la primera resolución. Los átomos a unificar son $t_1 =$ `suma(X0,Y0,s(s(0)))` y $t_2 =$`suma(0,Y1,Y1)`. Para unificar los primeros argumentos necesitamos la sustitución $\sigma_1 =$ `{X0/0}`. Aplicando $\sigma_1$ a los átomos obtenemos $\sigma_1(t_1) =$`suma(0,Y0,s(s(0)))` y $\sigma_1(t_2) =$`suma(0,Y1,Y1)`. Para unificar los segundos argumentos podemos usar la sustitución $\sigma_2 =$`{Y0/Y1}`. Aplicando $\sigma_2$ a los átomos obtenemos $\sigma_2(\sigma_1(t_1)) =$`suma(0,Y1,s(s(0)))` y $\sigma_2(\sigma_1(t_2)) =$`suma(0,Y1,Y1)`. Para unificar los terceros argumentos necesitamos la sustitución $\sigma_3 =$`{Y1/s(s(0))}`. Aplicando $\sigma_3$ a los átomos obtenemos $\sigma_3(\sigma_2(\sigma_1(t_1))) = \sigma_3(\sigma_2(\sigma_1(t_2))) =$`suma(0,s(s(0)),s(s(0)))`. En definitiva,

un unificador de $t_1$ y $t_2$ se obtiene componiendo las anteriores sustituciones $\sigma = \sigma_3\sigma_2\sigma_1$ ={X0/0, Y0/s(s(0)), Y1/s(s(0))}

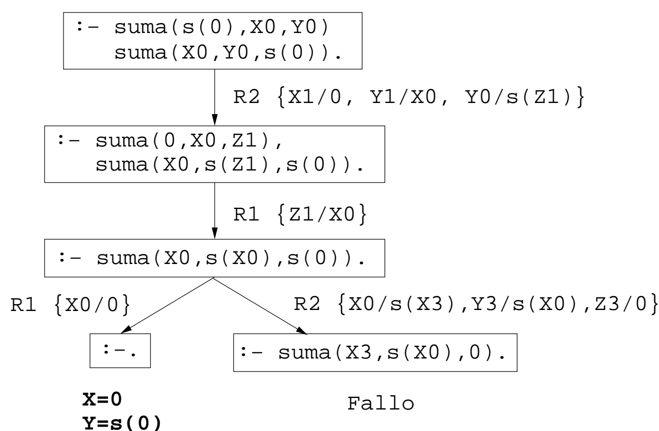
La cuarta cuestión es resolver el sistema de ecuaciones

$$1 + X = Y$$
$$X + Y = 1$$

En este caso basta plantear una pregunta compuesta

```
?- suma(s(0),X,Y), suma(X,Y,s(0)).
X = 0
Y = s(0) ;
No
```

El árbol de deducción es




En este ejemplo se observa que el unificador del primer objetivo y la cabeza de la cláusula se le aplica a los restantes objetivos: en el primer paso se ha sustituido la variable `Y0` del segundo objetivo por `s(Z1)`.

## 2.2 Listas

En esta sección vamos a estudiar cómo se representan las listas en Prolog y a definir relaciones sobre listas que usaremos en el capítulo siguiente.

### 2.2.1 Representación de listas

De manera análoga a la construcción de los naturales a partir de `0` y `s`, las listas puede definirse mediante una constante `[]` (que representa la lista vacía) y un símbolo de función binario `.` (de manera que que si `L` es una lista el término `.(a,L)` representa la lista obtenida añadiendo el elemento `a` a la lista `L`). Nótese que no todas las expresiones `.(a,b)` son listas, sino sólo las que se obtienen mediante las siguientes reglas:

- La lista vacía `[]` es una lista.
- Si `L` es una lista, entonces `.(a,L)` es una lista.

Por ejemplo, la lista cuyo único elemento es `a` se representa por `.(a,[])` y la lista cuyos elementos son `a` y `b` se representa por `.(a,.(b,[]))`. Para simplificar la notación, Prolog admite escribir las listas utilizando corchetes y separando sus elementos por comas; por ejemplo, las listas anteriores pueden escribirse como `[a]` y `[a,b]`, respectivamente. Para comprobar la correspondencia, podemos utilizar la unificación de Prolog (`=`):

```
?- .(X,Y) = [a].
X = a
Y = []

?- .(X,Y) = [a,b].
X = a
Y = [b]

?- .(X,.(Y,Z)) = [a,b].
X = a
Y = b
Z = []
```

En el segundo ejemplo se observa que si `.(X,Y)` es una lista, entonces `X` es el primer elemento e `Y` es el resto de la lista. En la escritura abreviada de listas en Prolog, dicho término puede escribirse como `[X|Y]`. Otros ejemplos, usando dicha notación son

```
?- [X|Y] = [a,b].
X = a
```

```
Y = [b]

?- [X|Y] = [a,b,c,d].
X = a
Y = [b, c, d]

?- [X,Y|Z] = [a,b,c,d].
X = a
Y = b
Z = [c, d]
```

### 2.2.2 Concatenación de listas

Vamos a definir la relación `conc(A,B,C)` que se verifique si `C` es la lista obtenida escribiendo los elementos de la lista `B` a continuación de los elementos de la lista `A`. Por ejemplo, `conc([a,b],[c,d],C)` se verifica si `C` es `[a,b,c,d]`. La definición, por recursión en el primer argumento, puede hacerse mediante las siguientes reglas:

- Si `A` es la lista vacía, entonces la concatenación de `A` y `B` es `B`.
- Si `A` es una lista cuyo primer elemento es `X` y cuyo resto es `D`, entonces la concatenación de `A` y `B` es una lista cuyo primer elemento es `X` y cuyo resto es la concatenación de `D` y `B`.

Una representación de las reglas da el siguiente programa

```
conc(A,B,C) :- A=[], C=B.
conc(A,B,C) :- A=[X|D], conc(D,B,E), C=[X|E].
```

que puede simplificarse, introduciendo patrones en los argumentos,

```
conc([],B,B).
conc([X|D],B,[X|E]) :- conc(D,B,E).
```

Hay que resaltar la analogía entre la definición de `conc` y la de `suma`, Además, como hicimos en el caso de la `suma`, podemos usar `conc` para resolver distintas cuestiones como

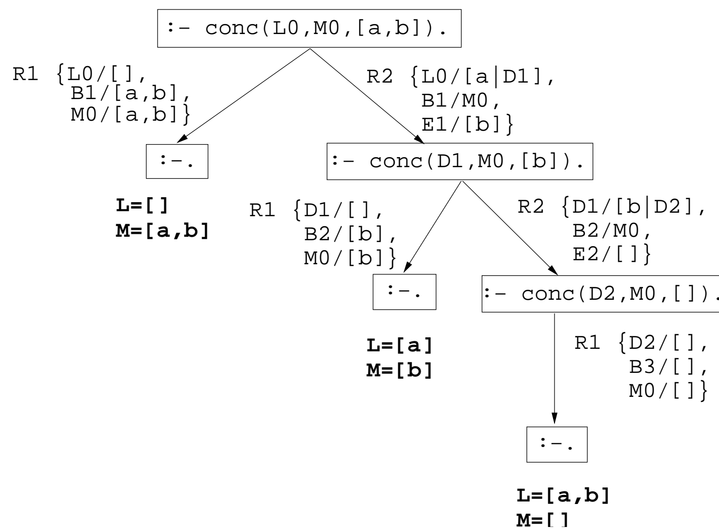
1. ¿Cuál es el resultado de concatenar las listas `[a,b]` y `[c,d,e]`?.

2. ¿Qué lista hay que añadirle al lista `[a,b]` para obtener `[a,b,c,d]`?.

3. ¿Qué dos listas hay que concatenar para obtener `[a,b]`?.

```
?- conc([a,b],[c,d,e],L).
L = [a, b, c, d, e]
?- conc([a,b],L,[a,b,c,d]).
L = [c, d]
?- conc(L,M,[a,b]).
L = []
M = [a, b] ;
L = [a]
M = [b] ;
L = [a, b]
M = [] ;
No
```

El árbol de deducción correspondiente a la última cuestión es




en el que vuelve a resaltar la analogía con el correspondiente a la tercera cuestión de la `suma`.

La relación `conc` está predefinida en Prolog como `append`.

### 2.2.3 La relación de pertenencia

Vamos a definir la relación `pertenece(X,L)` que se verifique si `X` es un elemento de la lista `L`. Para definirla basta observar que para que un elemento pertenezca a una lista tiene que ser igual al primer elemento de la lista o tiene que pertenecer al resto de la lista

```
pertenece(X,[X|L]).
pertenece(X,[Y|L]) :- pertenece(X,L).
```

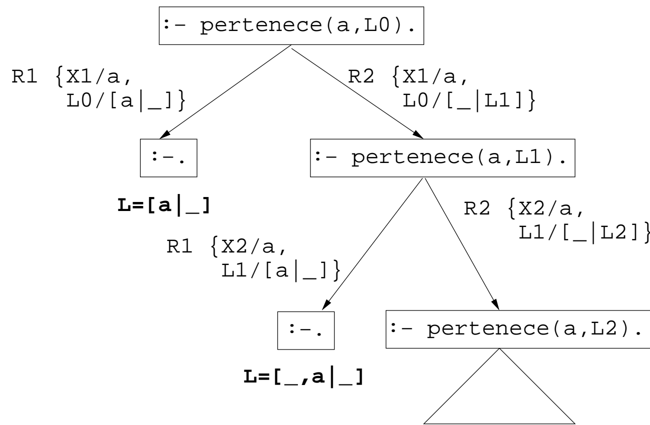
Puesto que la primera cláusula no depende de la variable `L` y la segunda no depende de la variable `Y` podemos sustituirla por la variable anónima `_`. La definición queda como

```
pertenece(X,[X|_]).
pertenece(X,[_|L]) :- pertenece(X,L).
```

Con la relación pertenece podemos determinar si un elemento pertenece a una lista, calcular los elementos de una lista y determinar la forma de las listas que contengan un elemento, como se muestra en los siguientes ejemplos

```
?- pertenece(b,[a,b,c]).
Yes

?- pertenece(d,[a,b,c]).
No

?- pertenece(X,[a,b,a]).
X = a ;
X = b ;
X = a ;
No

?- pertenece(a,L).
L = [a|_G233] ;
L = [_G232, a|_G236] ;
L = [_G232, _G235, a|_G239]
Yes
```

En el último ejemplo hay infinitas respuestas: una lista con `a` como primer elemento, segundo, tercero, etc. En las respuestas aparecen variables anónimas internas (`_G232`, `_G233`, ...). Su árbol de deducción es




La relación `pertenece` está predefinida en Prolog como `member`.

## 2.3 Disyunciones

Se pueden escribir disyunciones en Prolog usando el operador `;`. De esta forma, la relación `pertenece` puede definirse por

```
pertenece(X,[Y|L]) :- X=Y ; pertenece(X,L).
```

Desde el punto de vista deductivo, la anterior definición se transforma en

```
pertenece(X,[Y|L]) :- X=Y.
pertenece(X,[Y|L]) :- pertenece(X,L).
```

## 2.4 Operadores

Prolog permite la declaración de operadores indicando su nombre, tipo y precedencia. Además, dispone de un conjunto de operadores previamente declarados. Uno de hechos es el operador `+` que está declarado de tipo `yfx` (que significa que es infijo y asocia por la izquierda) y precedencia 500. Podemos comprobar el carácter infijo y la asociatividad mediante los siguientes ejemplos,

```
?- +(X,Y) = a+b.
X = a
Y = b
?- +(X,Y) = a+b+c.
X = a+b
Y = c
?- +(X,Y) = a+(b+c).
X = a
Y = b+c
?- a+b+c = (a+b)+c.
Yes
?- a+b+c = a+(b+c).
No
```

La siguiente tabla contiene otros operadores aritméticos predeclarados

| Precedencia | Tipo | Operadores | |
|---|---|---|---|
| 500 | yfx | +,- | Infijo asocia por la izquierda |
| 500 | fx | - | Prefijo no asocia |
| 400 | yfx | *, / | Infijo asocia por la izquierda |
| 200 | xfy | ^ | Infijo asocia por la derecha |

Podemos observar la diferencia de asociatividad entre `+` y `^` en los ejemplos siguientes

```
?- X^Y = a^b^c.
X = a
Y = b^c
?- a^b^c = (a^b)^c.
No
?- a^b^c = a^(b^c).
Yes
```

También podemos observar cómo se agrupan antes los operadores de menor precedencia

```
?- X+Y = a+b*c.
X = a
Y = b*c
```

```
?- X*Y = a+b*c.
No
?- X*Y = (a+b)*c.
X = a+b
Y = c
?- a+b*c = a+(b*c).
Yes
?- a+b*c = (a+b)*c.
No
```

Se pueden definir operadores como se muestra a continuación

```
:-op(800,xfx,estudian).
:-op(400,xfx,y).

juan y ana estudian lógica.
```

Hemos declarado a `estudian` e `y` como operadores infijo no asociativos y hemos utilizado en la escritura de la cláusula. Podemos también usarlo en las consultas

```
?- Quienes estudian lógica.
Quienes = juan y ana

?- juan y Otro estudian Algo.
Otro = ana
Algo = lógica
```

Usaremos la definición de operadores en la declaración de las conectivas (pág. 44).

## 2.5 Aritmética

Hemos visto en la sección cómo construir expresiones aritméticas. También pueden evaluarse mediante `is` como se muestra a continuación

```
?- X is 2+3^3.
X = 29
?- X is 2+3, Y is 2*X.
X = 5
Y = 10
```

Cuando Prolog encuentra una expresión de la forma `V is E` (donde `V` es un a variable y `E` es una expresión aritmética), evalúa `E` y le asigna su valor a `V`. Además de las operaciones, se disponen de los operadores de comparación `<`, `=<`, `>` y `>=` como operadores infijo.

Usando la evaluación aritmética podemos definir nuevas relaciones. Como ejemplo, veamos una definición de la relación `factorial(X,Y)` que se verifica si `Y` es el factorial de `X`.

```
factorial(1,1).
factorial(X,Y) :-
   X > 1,
   A is X - 1,
   factorial(A,B),
   Y is X * B.
```

Con la anterior definición se pueden calcular factoriales

```
?- factorial(3,Y).
Y = 6 ;
No
```

El árbol de deducción correspondiente se muestra en la figura 2.1

## 2.6 Control mediante corte

Prolog dispone del corte (`!`) como método para podar la búsqueda y aumentar la eficiencia de los programas. Un caso natural en donde aplicar la poda es en los problemas con solución única [2]. Por ejemplo, consideremos la relación `nota(X,Y)` que se verifica si `Y` es la calificación correspondiente a la nota `X`; es decir, `Y` es `suspenso` si `X` es menor que `5`, `Y` es `aprobado` si `X` es mayor o igual que `5` pero menor que `7`, `Y` es `notable` si `X` es mayor que `7` pero menor que `9` e `Y` es sobresaliente si `X` es mayor que `9`. Una definición de `nota` es

```
nota(X,suspenso)       :- X < 5.
nota(X,aprobado)       :- X >= 5, X < 7.
nota(X,notable)        :- X >= 7, X < 9.
nota(X,sobresaliente) :- X >= 9.
```

[2] El origen de `!` es el símbolo matemático $(\exists! x)A(x)$ que indica que existe un único $x$ tal que $A(x)$.

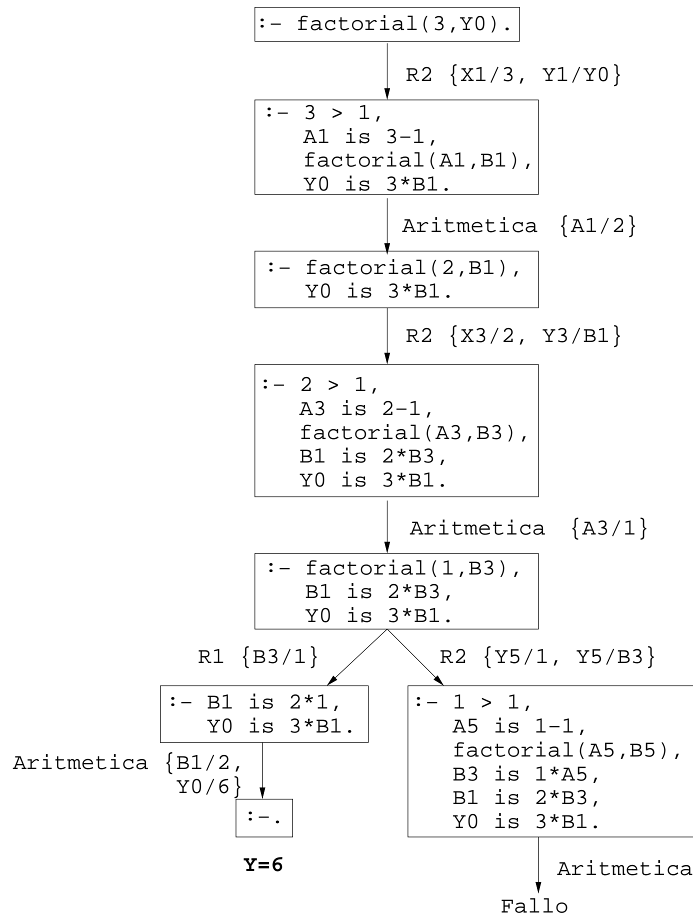
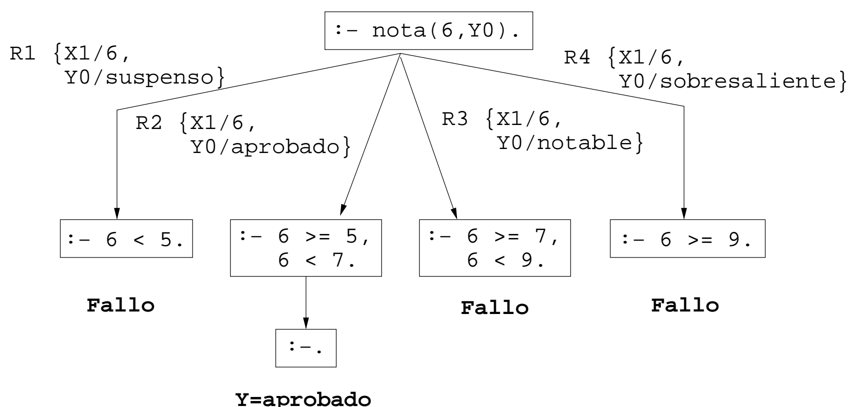



Figura 2.1: Deducción del factorial

Si calculamos la calificación correspondiente a un `6`,

```
?- nota(6,Y).
Y = aprobado;
No
```

que genera el siguiente árbol de deducción




Vemos que se realizan cálculos que no son necesarios:

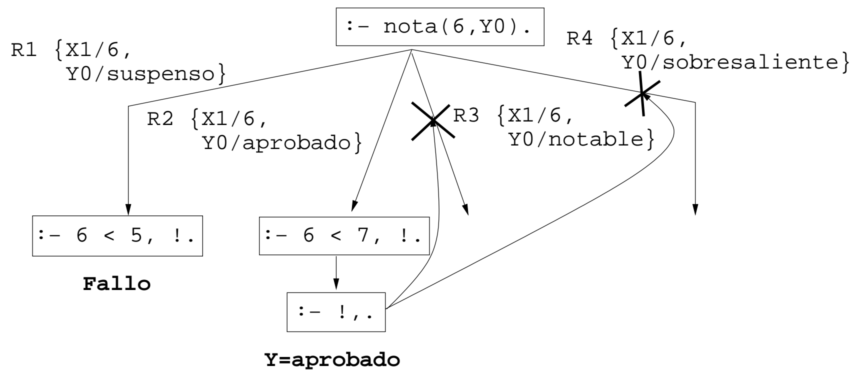
- cuando llega a la segunda rama, el objetivo `6 < 5` ha fallado con lo que no es necesario comprobar en la segunda rama que `6 >= 5`,

- cuando encuentra la solución en la segunda rama y se pregunta por otras soluciones, debe de responder `No` sin necesidad de búsqueda porque la solución es única.

Estos cálculos se pueden evitar modificando el programa introduciendo cortes

```
nota(X,suspenso)      :- X < 5, !.
nota(X,aprobado)      :- X < 7, !.
nota(X,notable)       :- X < 9, !.
nota(X,sobresaliente).
```

Con la nueva definición y la misma pregunta el árbol de deducción es




Vemos que el efecto del corte es la eliminación de las alternativas abiertas por debajo del padre de la cláusula que ha introducido ese corte.

Junto al aumento de la eficiencia, el corte supone una pérdida del sentido declarativo de los programas pudiendo producir respuesta no deseadas como la siguiente

```
?- nota(6,sobresaliente).
Yes
```

Las respuestas correctas se obtienen cuando el primer argumento es un número y el segundo una variable, lo que se puede indicar en la documentación mediante `nota(+X,-Y)`.

Otro uso del corte se da en los casos en los que se desea sólo la primera solución. Por ejemplo, hemos visto que la relación `member(X,L)` permite determinar si `X` pertenece a `L`, pero una vez encontrado si se pide otra solución vuelve a buscarla

```
?- member(X,[a,b,a,c]), X=a.
X = a ;
X = a ;
No
```

Si sólo deseamos la primera solución y que no busque otras, podemos usar la relación `memberchk`

```
?- memberchk(X,[a,b,a,c]), X=a.
X = a ;
No
```

La definición de `memberchk` es

```
memberchk(X,[X|_]) :- !.
memberchk(X,[_|L]) :- memberchk(X,L).
```

## 2.7 Negación

Mediante el corte se puede definir la negación como fallo: para demostrar la negación de una propiedad `P`, se intenta demostrar `P` si se consigue entonces no se tiene la negación y si no se consigue entonces se tiene la negación

```
no(P) :- P, !, fail.          % No 1
no(P).                        % No 1
```

donde `fail` es un átomo que siempre es falso.

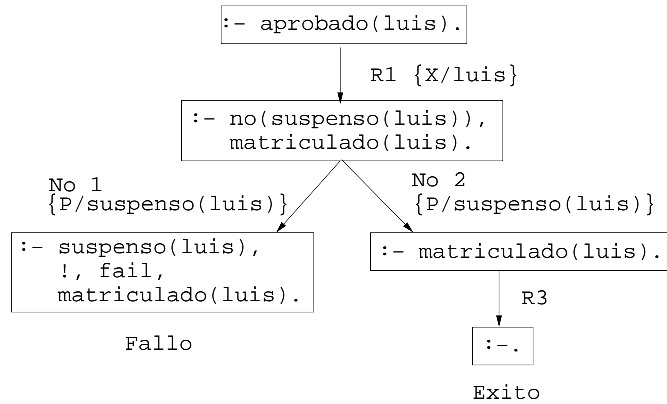
Vamos a explicar el comportamiento de la negación en el siguiente programa

```
aprobado(X) :- no(suspenso(X)), matriculado(X).    % R1
matriculado(juan).                                 % R2
matriculado(luis).                                 % R3
suspenso(juan).                                    % R4
```

y con las consultas

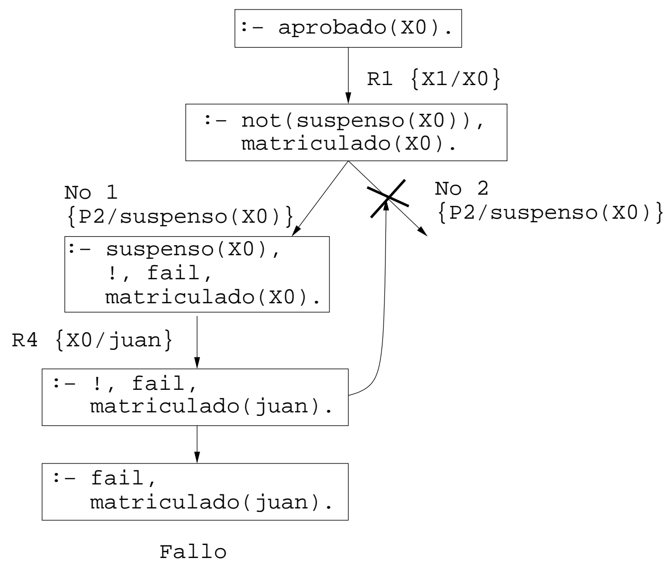
```
?- aprobado(luis).
Yes

?- aprobado(X).
No
```

La respuesta a la primera pregunta es la esperada: `luis` está aprobado porque no figura entre los suspensos y está matriculado. El correspondiente árbol de deducción es




para demostrar que `luis` no está suspenso intenta probar que `luis` está suspenso, al fallar (rama izquierda) da por demostrado que luis está suspenso (rama derecha).

La respuesta a la segunda pregunta parece contradecir a la primera. Su correspondiente árbol de deducción es



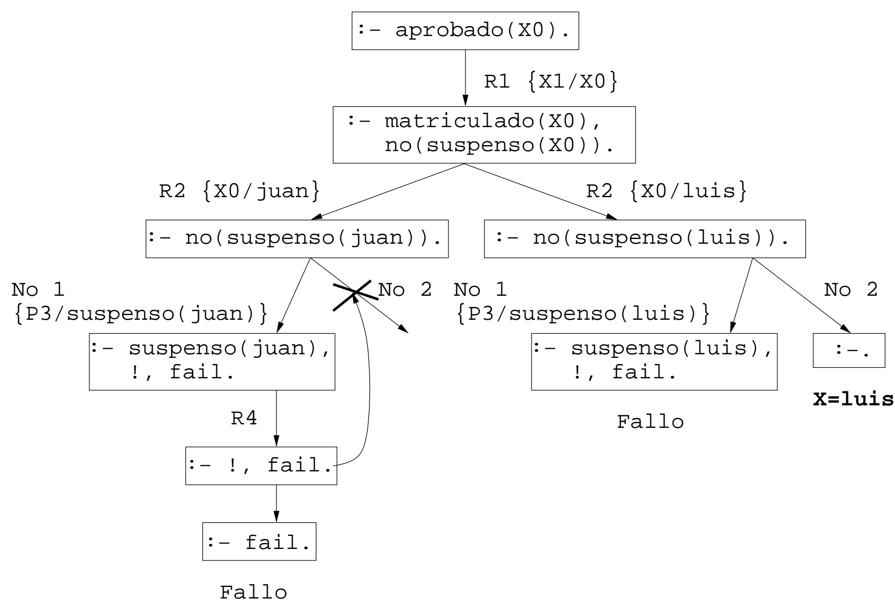

La diferencia con la primera es la presencia de variables libres. Si cambiamos el orden de las condiciones en la regla 1

```
aprobado(X) :- matriculado(X), no(suspenso(X)).   % R1
```

y volvemos a preguntar

```
?- aprobado(X).
X = luis
Yes
```

obtenemos la respuesta esperada: `luis` está aprobado. Su árbol de deducción es




La relación de segundo orden `no` (porque su argumento es una relación) está predefinida en Prolog mediante `not` o `\+`.

Como una aplicación, vamos a estudiar las definiciones con negación y con corte de la relación predefinida `delete(L1,X,L2)` que se verifica si `L2` es la lista obtenida eliminando los elementos de `L1` unificables simultáneamente con `X`; por ejemplo,

```
?- delete([a,b,a,c],a,L).
L = [b, c] ;
No
```

```
?- delete([a,Y,a,c],a,L).
Y = a
L = [c] ;
No
?- delete([a,Y,a,c],X,L).
Y = a
X = a
L = [c] ;
No
```

La definición de `delete` usando negación es

```
delete_1([],_,[]).
delete_1([X|L1],Y,L2) :-
   X=Y,
   delete_1(L1,Y,L2).
delete_1([X|L1],Y,[X|L2]) :-
   not(X=Y),
   delete_1(L1,Y,L2).
```

y, eliminando la negación mediante corte, su definición es

```
delete_2([],_,[]).
delete_2([X|L1],Y,L2) :-
   X=Y, !,
   delete_2(L1,Y,L2).
delete_2([X|L1],Y,[X|L2]) :-
   % not(X=Y),
   delete_2(L1,Y,L2).
```

La segunda cláusula puede simplificarse introduciendo la unificación en sus argumentos

```
delete_2([X|L1],X,L2) :- !, delete_2(L1,Y,L2).
```

## 2.8 El condicional

En Prolog se dispone del condicional `->`. Usando el condicional, se puede definir la relación `nota` (pág. 27)

```
nota(X,Y) :-
   X < 5 -> Y = suspenso ;          % R1
   X < 7 -> Y = aprobado ;          % R2
   X < 9 -> Y = notable ;           % R3
   true  -> Y = sobresaliente.      % R4
```

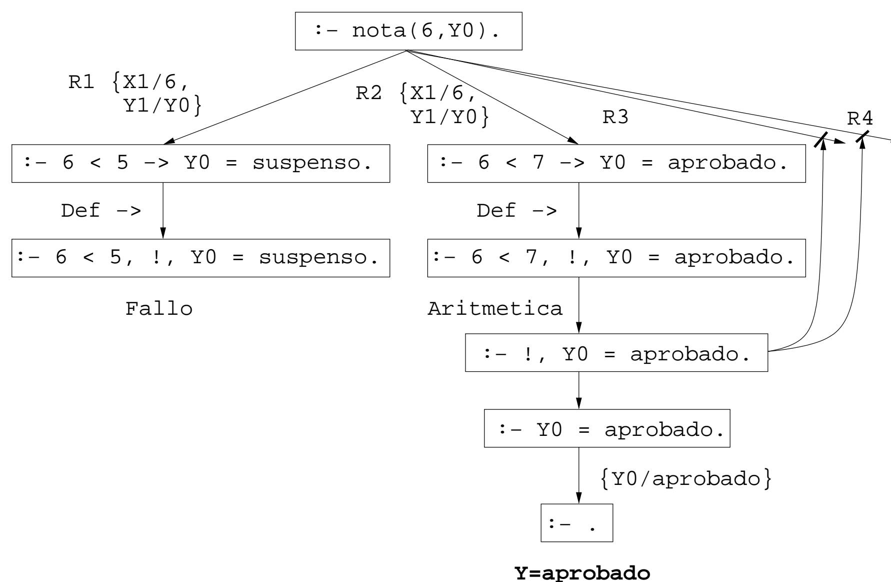
donde el condicional está declarado como operador infijo (de menor precedencia que la disyunción) y definido por

```
P -> Q :- P, !, Q.                  % Def. ->
```

y la constante verdad (`true`) está definida por

```
true.
```

El árbol de deducción corespondiente a la pregunta `nota(6,Y)` es




El árbol es análogo al de la definición con corte (pág 29).

## 2.9 Predicados sobre tipos de término

Existen distintos predicados para comprobar los tipos de términos: variables (`var`), átomos (`atom`), cadenas (`string`), números (`number`) y términos com-

puestos (compound). El predicado atomic sirve para reconocer los términos atómicos (es decir, a las variables, átomos, cadenas y números). Por ejemplo,

```
?- var(X1).                 =>  Yes
?- var(_).                  =>  Yes
?- var(_X1).                =>  Yes
?- X1=a, var(X1).           =>  No
?- atom(atomo).             =>  Yes
?- atom('atomo').           =>  Yes
?- atom([]).                =>  Yes
?- atom(3).                 =>  No
?- atom(1+2).               =>  No
?- number(123).             =>  Yes
?- number(-25.14).          =>  Yes
?- number(1+3).             =>  No
?- X is 1+3, number(X).     =>  X=4  Yes
?- compound(1+2).           =>  Yes
?- compound(f(X,a)).        =>  Yes
?- compound([1,2]).         =>  Yes
?- compound([]).            =>  No
?- atomic(atomo).           =>  Yes
?- atomic(123).             =>  Yes
?- atomic(X).               =>  No
?- atomic(f(1,2)).          =>  No
```

## 2.10 Comparación y ordenación de términos

Para comprobar si dos términos son idénticos se dispone del operador ==. La relación de identidad es más fuerte que la de unificación = como se comprueba en los siguientes ejemplos

```
?- f(X) = f(Y).
X = _G164
Y = _G164
Yes
?- f(X) == f(Y).
No
```

```
?- f(X) == f(X).
X = _G170
Yes
```

Los términos están ordenados según el siguiente orden (de menor a mayor):

- las variables (de más viejas a más recientes),
- los números (según sus valores),
- los átomos (en orden alfabético),
- las cadenas (en orden alfabético) y
- los términos compuestos (primero los de mayor aridad y los de la misma aridad ordenados según el símbolo de función (alfabéticamente) y sus argumentos (de izquierda a derecha)).

La relación básica de comparación del orden de términos es `@<`: la relación `T1 @< T2` se verifica si el término `T1` es anterior a `T2` en el orden de los términos. Por ejemplo,

```
?- Z = f(X,Y), X @< Y.  =>  Yes
?- Z = f(X,Y), Y @< X.  =>  No
?- X @< 3.              =>  Yes
?- 3 @< X.              =>  No
?- 21 @< 123.           =>  Yes
?- 12 @< a.             =>  Yes
?- a @< 12.             =>  No
?- ab @< ac.            =>  Yes
?- a @< ac.             =>  Yes
?- a21 @< a123.         =>  No
?- g @< f(b).           =>  Yes
?- f(b) @< f(a,b).      =>  Yes
?- f(a,b) @< f(a,a).    =>  No
?- f(a,b) @< f(a,a(1)). =>  Yes
?- [a,1] @< [a,3].      =>  Yes
?- [a] @< [a,3].        =>  Yes
```

Usando la relación `@<` se puede definir la relación `ordenada(L1,L2)` que se verifica si `L2` es la lista obtenida ordenando de manera creciente los distintos elementos de `L1`

```
ordenada([],[]).
ordenada([X|R],Ordenada) :-
   divide(X,R,Menores,Mayores),
   ordenada(Menores,Menores_ord),
   ordenada(Mayores,Mayores_ord),
   append(Menores_ord,[X|Mayores_ord],Ordenada).

divide(_,[],[],[]).
divide(X,[Y|R],Menores,[Y|Mayores]) :-
   X @< Y, !,
   divide(X,R,Menores,Mayores).
divide(X,[Y|R],[Y|Menores],Mayores) :-
   Y @< X, !, divide(X,R,Menores,Mayores).
divide(X,[_Y|R],Menores,Mayores) :-
   % X == _Y,
   divide(X,R,Menores,Mayores).
```

Como ejemplo

```
?- ordenada([c4,2,a5,2,c3,a5,2,a5],L).
L = [2, a5, c3, c4] ;
No

?- ordenada([f(a,b),Y,a2,a,,X,f(c),1,f(X,Y),f(Y,X),[a,b],[b],[]],L),
   X=x, Y=y.
Y = y
X = x
L = [y,x,1,[],a,a2,f(c),[a,b],[b],f(y,x),f(x,y),f(a,b)]
Yes
```

El predicado predefinido correspondiente a `ordena` es `sort`.

## 2.11 Procesamiento de términos

La relación `?T =.. ?L` se verifica si `L` es una lista cuya primer elemento es el functor del término `T` y los restantes elementos de `L` son los argumentos de `T`. Por ejemplo,

```
?- padre(juan,luis) =.. L.
L = [padre, juan, luis]
?- T =.. [padre, juan, luis].
T = padre(juan,luis)
```

Como aplicación de `=..` consideremos la relación `alarga(F1,N,F2)` que se verifica si `F1` y `F2` son figuras geométricas del mismo tipo y el tamaño de la `F2` es el de la `F2` multiplicado por `N`, donde las figuras geométricas se representan como términos en los que el functor indica el tipo de figura y los argumentos su tamaño; por ejemplo,

```
?- alarga(triangulo(3,4,5),2,F).
F = triangulo(6, 8, 10)

?- alarga(cuadrado(3),2,F).
F = cuadrado(6)
```

La definición de `alarga` es

```
alarga(Figura1,Factor,Figura2) :-
   Figura1 =.. [Tipo|Argumentos1],
   multiplica_lista(Argumentos1,Factor,Argumentos2),
   Figura2 =.. [Tipo|Argumentos2].

multiplica_lista([],_,[]).
multiplica_lista([X1|L1],F,[X2|L2]) :-
   X2 is X1*F,
   multiplica_lista(L1,F,L2).
```

donde `multiplica_lista(L1,F,L2)` se verifica si `L2` es la lista obtenida multiplicando cada elemento de `L1` por `F`.

Otras relaciones para el procesamiento de términos son `functor(T,F,A)`, que se verifica si `F` es el functor del término `T` y `A` es su aridad, y `arg(N,T,A)`, que se verifica si `A` es el argumento del término `T` que ocupa el lugar `N`; por ejemplo,

```
?- functor(g(b,c,d),F,A).                 => F = g     A = 3
?- functor(T,g,2).                        => T = g(_G237,_G238)
?- functor([b,c,d],F,A).                  => F = '.'   A = 2
```

```
?- arg(2,g(b,c,d),X).                    => X = c
?- arg(2,[b,c,d],X).                     => X = [c, d]
?- functor(T,g,3),arg(1,T,b),arg(2,T,c). => T = g(b, c, _G405)
```

## 2.12 Procedimientos aplicativos

La relación `apply(T,L)` se verifica si es demostrable `T` después de aumentar el número de sus argumentos con los elementos de `L`; por ejemplo, si `producto` es la relación definida por,

```
producto(X,Y,Z) :- Z is X*Y.
```

entonces

```
?- producto(2,3,X).                        =>  X = 6
?- apply(producto,[2,3,X]).                =>  X = 6
?- apply(producto(2),[3,X]).               =>  X = 6
?- apply(producto(2,3),[X]).               =>  X = 6
?- apply(append([1,2]),[X,[1,2,3,4,5]]).   =>  X = [3, 4, 5]
```

La relación `apply` se puede definir mediante `=..`

```
apply(Termino,Lista) :-
   Termino =.. [Pred|Arg1],
   append(Arg1,Lista,Arg2),
   Atomo =.. [Pred|Arg2],
   Atomo.
```

La relación `maplist(P,L1,L2)` se verifica si se cumple el predicado `P` sobre los sucesivos pares de elementos de la las listas `L1` y `L2`; por ejemplo, si `sucesor` es la relación definida por

```
sucesor(X,Y) :- number(X), Y is X+1.
sucesor(X,Y) :- number(Y), X is Y-1.
```

entonces

```
?- maplist(sucesor,[2,4],[3,5]).   =>  Yes
?- maplist(succ,[0,4],[3,5]).      =>  No
?- maplist(sucesor,[2,4],Y).       =>  Y = [3, 5]
?- maplist(sucesor,X,[3,5]).       =>  X = [2, 4]
```

La relación `maplist` puede definirse mediante `apply`

```
maplist(_,[],[]).
maplist(R,[X1|L1],[X2|L2]) :-
   apply(R,[X1,X2]),
   maplist(R,L1,L2).
```

Usando `maplist` se puede redefinir `multiplica_lista` (pág. 39)

```
multiplica_lista(L1,F,L2) :-
   maplist(producto(F),L1,L2).
```

## 2.13 Todas las soluciones

La relación `findall(T,O,L)` se verifica si `L` es la lista de las instancias del término `T` que verifican el objetivo `O`. La relación `setof(T,O,L)` se verifica si `L` es la lista ordenada sin repeticiones de las instancias del término `T` que verifican el objetivo `O`. Por ejemplo,

```
?- findall(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).
L = [4, 3, 4, 2, 3]
?- setof(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).
L = [2, 3, 4]
?- setof(X,member(X,[d,4,a,3,d,4,2,3]),L).
L = [2, 3, 4, a, d]
?- findall(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).
L = []
Yes
?- setof(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).
No
```

En los últimos ejemplos se observa la diferencia entre `findall` y `setof` cuando no hay ninguna instancia que verifique el objetivo. Otra diferencia ocurre cuando hay variables libres; por ejemplo,

```
?- findall(X,member([X,Y],[[5,0],[3,0],[4,1],[2,1]]),L).
L = [5, 3, 4, 2]
?- setof(X,member([X,Y],[[5,0],[3,0],[4,1],[2,1]]),L).
Y = 0
```

```
L = [3, 5] ;
X = _G398
Y = 1
L = [2, 4] ;
No
```

El conjunto {`X` : (∃`Y`) `member([X,Y],[[5,0],[3,0],[4,1],[2,1]])`} puede calcularse con `setof` usando el cuantificador existencial (^)

```
?- setof(X,Y^member([X,Y],[[5,0],[3,0],[4,1],[2,1]]),L).
L = [2, 3, 4, 5]
```

Mediante `setof` se pueden definir las operaciones conjuntista. Para facilitar las definiciones, defimos la relación `setof0(T,0,L)` que es como `setof` salvo en el caso en que ninguna instancia de `T` verifique `O`, en cuyo caso `L` es la lista vacía

```
setof0(X,O,L) :- setof(X,O,L), !.
setof0(_,_,[]).
```

Las operaciones de intersección, unión y diferencia se definen por

```
interseccion(S,T,U) :-
   setof0(X, (member(X,S), member(X,T)), U).
union(S,T,U) :-
   setof0(X, (member(X,S); member(X,T)), U).
diferencia(S,T,U) :-
   setof0(X, (member(X,S), not(member(X,T))), U).
```

# Capítulo 3

# Elementos de lógica proposicional

En este capítulo construiremos con Prolog sistemas elementales de la lógica proposicional basados directamente en las definiciones semánticas. De paso comentaremos los elementos de OTTER y MACE relacionados con los conceptos introducidos.

Como textos fundamentales para este capítulo recomendamos [1], [3], [4], [6], [8], [9], [12], [13], [14] y [16].

## 3.1 Sintaxis de la lógica proposicional

El **alfabeto** de la lógica proposicional está formado por los **símbolos proposicionales**, las **conectivas lógicas** ($\neg$ (negación), $\wedge$ (conjunción), $\vee$ (disyunción), $\rightarrow$ (condicional) y $\leftrightarrow$ (equivalencia)) y los **símbolos auxiliares** ( "(" y ")"). A partir del alfabeto se definen las **fórmulas proposicionales** mediante las siguientes reglas:

- los símbolos proposicionales son fórmulas proposicionales (llamadas **fórmulas atómicas** o **átomos** ) y

- si $F$ y $G$ son fórmulas proposicionales, entonces $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ y $(F \leftrightarrow G)$ también lo son.

Por ejemplo, si $p$ y $q$ son símbolos proposicionales, las siguientes expresiones son fórmulas proposicionales $\neg p$, $(p \wedge q)$, $(q \rightarrow \neg q)$ y $((p \rightarrow q) \vee (q \rightarrow p))$.

Para simplificar la notación se introduce las siguientes reglas:

- Se eliminan lo paréntesis externos; por ejemplo, se escribe $p \wedge q$ en lugar de $(p \wedge q)$.
- Se define una precedencia entre las conectivas (de menor a mayor: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) de forma que las de menor precedencia se agrupan antes; por ejemplo,

  $p \wedge \neg q \vee r \rightarrow s$ representa a $((p \wedge \neg q) \vee r) \rightarrow s$ y
  
  $p \vee \neg q \wedge r \rightarrow \neg s \vee t$ representa a $(p \vee (\neg q \wedge r)) \rightarrow (\neg s \vee t)$.
- Las conectivas binarias asocian por la derecha; por ejemplo, $p \wedge q \wedge r$ representa a $p \wedge (q \wedge r)$.

Usaremos las siguiente meta-variables: $p, p_0, p_1, \dots, q, q_0, q_1, \dots$ para los símbolos proposicionales; $F, F_0, F_1, \dots, G, G_0, G_1, \dots$ para las fórmulas proposicionales; SP para el conjunto de símbolos proposicionales y Prop para el conjunto de fórmulas proposicionales.

En Prolog, consideraremos que los símbolos proposicionales son los átomos y usaremos los siguientes símbolos para las conectivas `-` (negación), `&` (conjunción), `v` (disyunción), `=>` (condicional) y `<=>` (equivalencia). Declaramos su precedencia, posición y asociatividad mediante con `op` (pág. 26)

```
:- op(610, fy, -).      % negación
:- op(620, xfy,&).      % conjunción
:- op(630, xfy,v).      % disyunción
:- op(640, xfy,=>).     % condicional
:- op(650, xfy,<=>).    % equivalencia
```

Con estas declaraciones se pueden usar las reglas de eliminación de paréntesis y no se admiten fórmulas sintácticamente incorrectas

En OTTER los símbolos de las conectivas son `-` (negación), `&` (conjunción), `|` (disyunción), `->` (condicional) y `<->` (equivalencia).

Los distintos símbolos para las conectivas se resumen en el cuadro

| Lógica | $\neg$ | $\wedge$ | $\vee$ | $\rightarrow$ | $\leftrightarrow$ |
|---|---|---|---|---|---|
| Prolog | `-` | `&` | `v` | `=>` | `<=>` |
| OTTER | `-` | `&` | `\|` | `->` | `<->` |

## 3.2 Valores de verdad

Los elementos del conjunto $\mathbb{B} = \{0, 1\}$ se llaman **valores de verdad** . Se dice que 0 es el valor **falso** y el 1 es el valor **verdadero** .

La representación en Prolog de los valores de verdad son los átomos `0` y `1`.

```
valor_de_verdad(0).
valor_de_verdad(1).
```

## 3.3 Funciones de verdad

Para cada conectiva existe una función de verdad:

- La **función de verdad de la negación** es $\mathrm{FV}_\neg : \mathbb{B} \to \mathbb{B}$ definida por:

$$\mathrm{FV}_\neg(i) = \begin{cases} 1, & \text{si } i = 0; \\ 0, & \text{si } i = 1. \end{cases}$$

- La **función de verdad de la conjunción** es $FV_\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ definida por:

$$\mathrm{FV}_\wedge(i, j) = \begin{cases} 1, & \text{si } i = j = 1; \\ 0, & \text{en otro caso.} \end{cases}$$

- La **función de verdad de la disyunción** es $\mathrm{FV}_\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ definida por:

$$\mathrm{FV}_\vee(i, j) = \begin{cases} 0, & \text{si } i = j = 0; \\ 1, & \text{en otro caso.} \end{cases}$$

- La **función de verdad del condicional** es $\mathrm{FV}_\to : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ definida por:

$$\mathrm{FV}_\to(i, j) = \begin{cases} 0, & \text{si } i = 1, j = 0; \\ 1, & \text{en otro caso.} \end{cases}$$

- La **función de verdad de la equivalencia** es $\mathrm{FV}_\leftrightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ definida por:

$$\mathrm{FV}_\leftrightarrow(i, j) = \begin{cases} 1, & \text{si } i = j; \\ 0, & \text{en otro caso.} \end{cases}$$

En Prolog definimos las funciones de verdad mediante las relaciones [1]

[1] Nótese que en Prolog se puede usar el mismo símbolo de relación con distintas aridades.

- `funcion_de_verdad(+Op,+V1,-V)` [2] que se verifica si $\mathrm{FV}_{\texttt{Op}}(\texttt{V1}) = \texttt{V}$ y
- `funcion_de_verdad(+Op,+V1,+V2,-V)` que se verifica si $\mathrm{FV}_{\texttt{Op}}(\texttt{V1},\texttt{V2}) = \texttt{V}$.

```
funcion_de_verdad(-,  1, 0).
funcion_de_verdad(-,  0, 1).

funcion_de_verdad(v,   0, 0, 0) :- !.
funcion_de_verdad(v,   _, _, 1).
funcion_de_verdad(&,   1, 1, 1) :- !.
funcion_de_verdad(&,   _, _, 0).
funcion_de_verdad(=>,  1, 0, 0) :- !.
funcion_de_verdad(=>,  _, _, 1).
funcion_de_verdad(<=>, X, X, 1) :- !.
funcion_de_verdad(<=>, _, _, 0).
```

## 3.4 Valor de una fórmula en una interpretación

Una **interpretación** es una aplicación $I : \mathrm{SP} \to \mathbb{B}$. En las aplicaciones que vamos a considerar el conjunto de símbolos proposicionales es finito. Las interpretaciones las representaremos mediante listas de pares formados por un símbolo proposicional y un valor de verdad. Por ejemplo, `[(p,1),(r,0),(u,1)]` representa la interpretación que hace verdadero a $p$ y $u$ y falso a $r$.

Para cada interpretación $I$ existe una única aplicación $\hat{I} : \textsc{Prop} \to B$ tal que:

1. Para todo $p_i \in \mathrm{SP}$, $\hat{I}(p_i) = I(p_i)$.
2. Para toda $F \in \textsc{Prop}$, $\hat{I}(\neg F) = H_\neg(\hat{I}(F))$.
3. Para toda $F, G \in \textsc{Prop}$ y toda conectiva binaria $\circ$, $\hat{I}((F \circ G)) = H_\circ(\hat{I}(F), \hat{I}(G))$.

Se dice que $\hat{I}(F)$ es el **valor de verdad** de $F$ respecto de $I$ y, simplificando la notación, se representa por $I(F)$. En Prolog se define mediante la relación

[2] Usaremos el convenio de marcar los argumentos de la especificaciones de las relaciones con `+` los de entrada (que son términos sin variables libres), con `-` los de salida (que son variables) y con `?` los de entrada o salida

`valor(+F,+I,-V)` que se verifica si el valor de la fórmula `F` en la interpretación `I` es `V`: por ejemplo,

```
?- valor((p v q) & (-q v r), [(p,1),(q,0),(r,1)], V).
V = 1
?- valor((p v q) & (-q v r), [(p,0),(q,0),(r,1)], V).
V = 0
```

```
valor(F, I, V) :-
   memberchk((F,V), I).
valor(-A, I, V) :-
   valor(A, I, VA),
   funcion_de_verdad(-, VA, V).
valor(F, I, V) :-
   F =.. [Op, A, B],
   valor(A, I, VA),
   valor(B, I, VB),
   funcion_de_verdad(Op, VA, VB, V).
```

## 3.5 Interpretaciones principales de una fórmula

Una **interpretación principal** de una fórmula $F$ es una aplicación $I : S \to \mathbb{B}$ donde $S$ es el conjunto de los símbolos proposicionales que aparecen en $F$. Si $I$ es una interpretación principal de $F$ y $I_1$, $I_2$ son dos interpretaciones que extienden a $I$, entonces $I_1(F) = I_2(F)$; en otras palabra, el valor de $F$ queda determinado por sus interpretaciones principales.

La relación `interpretaciones_formula(+F,-L)` se verifica si `L` es el conjunto de las interpretaciones principales de la fórmula `F`: por ejemplo,

```
?- interpretaciones_formula((p v q) & (-q v r),L).
L = [[ (p, 0),  (q, 0),  (r, 0)],
      [ (p, 0),  (q, 0),  (r, 1)],
      [ (p, 0),  (q, 1),  (r, 0)],
      [ (p, 0),  (q, 1),  (r, 1)],
      [ (p, 1),  (q, 0),  (r, 0)],
      [ (p, 1),  (q, 0),  (r, 1)],
      [ (p, 1),  (q, 1),  (r, 0)],
      [ (p, 1),  (q, 1),  (r, 1)]]
```

```
interpretaciones_formula(F,U) :-
   findall(I,interpretacion_formula(I,F),U).
```

donde `interpretacion_formula(?I,+F)` se verifica si `I` es una interpretación de la fórmula `F`; por ejemplo,

```
?- interpretacion_formula(I,(p v q) & (-q v r)).
I = [ (p, 0),  (q, 0),  (r, 0)] ;
I = [ (p, 0),  (q, 0),  (r, 1)] ;
I = [ (p, 0),  (q, 1),  (r, 0)] ;
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 0)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

```
interpretacion_formula(I,F) :-
   simbolos_formula(F,U),
   interpretacion_simbolos(U,I).
```

donde

- `simbolos_formula(+F,?U)` se verifica si `U` es el conjunto ordenado de los símbolos proposicionales de la fórmula `F`; por ejemplo,

```
?- simbolos_formula((p v q) & (-q v r), U).
U = [p, q, r]
```

- `interpretacion_simbolos(+L,-I)` se verifica si `I` es una interpretación para la lista de átomos `L`; por ejemplo,

```
?- interpretacion_simbolos([p,q,r],I).
I = [ (p, 0),  (q, 0),  (r, 0)] ;
I = [ (p, 0),  (q, 0),  (r, 1)] ;
I = [ (p, 0),  (q, 1),  (r, 0)] ;
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
```

```
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 0)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

*1*

```
simbolos_formula(F,U) :-
   simbolos_formula_aux(F,U1),
   sort(U1,U).

simbolos_formula_aux(F,[F]) :-
   atom(F).
simbolos_formula_aux(-F,U) :-
   simbolos_formula_aux(F,U).
simbolos_formula_aux(F,U) :-
   F =.. [_Op,A,B],
   simbolos_formula_aux(A,UA),
   simbolos_formula_aux(B,UB),
   union(UA,UB,U).

interpretacion_simbolos([], []).
interpretacion_simbolos([A|L],[(A,V)|IL]) :-
   valor_de_verdad(V),
   interpretacion_simbolos(L, IL).
```

## 3.6 Modelo de una fórmula

Se dice que una interpretación $I$ es un **modelo de la fórmula** $F$ (o que la $F$ **es válida en** $I$), y se representa por $I \models F$, si el valor de $F$ en $I$ es verdadero.

La relación `es_modelo_formula(+I,+F)` se verifica si la interpretación `I` es un modelo de la fórmula `Fp`; por ejemplo,

```
?- es_modelo_formula([(p,1),(q,0),(r,1)], (p v q) & (-q v r)).
Yes
?- es_modelo_formula([(p,0),(q,0),(r,1)], (p v q) & (-q v r)).
No
```

```
es_modelo_formula(I,F) :-
   valor(F,I,V),
   V = 1.
```

Nótese que en la definición anterior no puede usarse `valor(F,I,1)` porque el tercer argumento de `valor` tiene que ser una variable (por la definición abreviada de `funcion_de_verdad`) y daría errores como aceptar que `[(p,1),(q,0)]` es modelo de `p => q`.

## 3.7 Cálculo de los modelos de una fórmula

Los **modelos principales de una fórmula** $F$ son las interpretaciones principales de $F$ que son modelos de $F$.

La relación `modelo_formula(?I,+F)` se verifica si `I` es un modelo principal de la fórmula `F` y `modelos_formula(+F,-L)` se verifica si `L` es el conjunto de los modelos principales de la fórmula `F`; por ejemplo,

```
?- modelo_formula(I,(p v q) & (-q v r)).
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No

?- modelos_formula((p v q) & (-q v r),L).
L = [[ (p, 0),  (q, 1),  (r, 1)],
     [ (p, 1),  (q, 0),  (r, 0)],
     [ (p, 1),  (q, 0),  (r, 1)],
     [ (p, 1),  (q, 1),  (r, 1)]]
```

```
modelo_formula(I,F) :-
   interpretacion_formula(I,F),
   es_modelo_formula(I,F).

modelos_formula(F,L) :-
   findall(I,modelo_formula(I,F),L).
```

Nótese la diferencia entre la relación `es_modelo_formula` que dada una fórmula y una interpretación *comprueba* si la interpretación es modelo de la fórmula y `modelo_formula` que dada una fórmula *calcula* modelos *principales* de la misma.

```
?- es_modelo_formula([(p,0),(q,1),(r,1)], p v q).
Yes
?- modelo_formula([(p,0),(q,1),(r,1)], p v q).
No
?- modelo_formula([(p,0),(q,1)], p v q).
Yes
?- modelo_formula(I, p v q).
I = [ (p, 0), (q, 1)]
Yes
?- es_modelo_formula(I, p v q).
I = [ (p v q, 1)|_G320]
Yes
```

Una interpretación $I$ es modelo de una fórmula $F$ syss [3] es una extensión de un modelo principal de $F$.

## 3.8 Satisfacibilidad

Una fórmula es **satisfacible** si tiene modelo e **insatisfacible** si no lo tiene.

La relación `es_satisfacible(+F)` se verifica si la fórmula `F` es satisfacible; por ejemplo,

```
?- es_satisfacible((p v q) & (-q v r)).
Yes
?- es_satisfacible((p & q) & (p => r) & (q => -r)).
No
```

```
es_satisfacible(F) :-
   interpretacion_formula(I,F),
   es_modelo_formula(I,F).
```

Obsérvese que la variable `I` sólo ocurre en el cuerpo de la regla y, por tanto, está existencialmente cuantificada.

[3] "syss" es una abreviatura de "si y sólo si"

## 3.9 Satisfacibilidad con MACE

MACE se puede usar para decidir la satisfacibilidad de una fórmula. Por ejemplo, para decidir la satisfacibilidad de la fórmula $(p \lor q) \land (\neg q \lor r)$ creamos un fichero (p.e `ej_insatisfacibilidad.in`) cuyo contenido es

```
formula_list(usable).
(p | q) & (-q | r).
end_of_list.
```

donde hemos escrito la fórmula con las conectivas en la notación de OTTER terminada en un punto dentro de la lista de fórmulas usables. Para aplicarle MACE al fichero, se ejecuta

```
> mace <ej_insatisfacibilidad.in
```

obteniendo como resultado

```
Exit by max_models parameter. The set is satisfiable
```

lo que indica la satisfacibilidad de la fórmula. Si además, queremos que calcule un modelo, la llamada es

```
> mace -p <ej_insatisfacibilidad.in >ej_insatisfacibilidad.out
```

y en el fichero de salida (`ej_insatisfacibilidad.out`) se encuentra un modelo

```
======================= Model #1 at 0.00 seconds:
 p: T
 q: F
 r: F
end_of_model
```

que se corresponde con $I_5$ de la página **??**.

En el caso de la fórmula insatisfacible $(p \land q) \land (p \to r) \land (q \to \neg r)$, el contenido del fichero de entrada es

```
formula_list(sos).
(p & q) & (p -> r) & (q -> -r).
end_of_list.
```

y el mensaje obtenido al aplicarle MACE es

```
The search is complete. No models were found.
```

lo que indica su insatisfacibilidad.

## 3.10 Contramodelo de una fórmula

Se dice que una interpretación $I$ es un **contramodelo de la fórmula** $F$ (o que $F$ **no es válida en** $I$), y se representa por $I \not\models F$, si $I$ no es un modelo de $F$. Un **contramodelo principal** de $F$ es una interpretación principal de $F$ que no es modelo de $F$.

La relación `contramodelo_formula(?I,+F)` se verifica si `I` es un contramodelo principal de la fórmula `F`: por ejemplo,

```
?- contramodelo_formula(I, p <=> q).
I = [ (p, 0),  (q, 1)] ;
I = [ (p, 1),  (q, 0)] ;
No
?- contramodelo_formula(I, p => p).
No
```

```
contramodelo_formula(I,F) :-
   interpretacion_formula(I,F),
   not(es_modelo_formula(I,F)).
```

## 3.11 Validez. Tautologías

Se dice que una fórmula $F$ es **válida** (o que $F$ es una **tautología**), y se representa por $\models F$, si todas las interpretaciones son modelos de $F$. Para decidir si $F$ es una tautología basta comprobar que no tiene contramodelos principales.

La relación `es_tautologia(+F)` se verifica si la fórmula `F` es una tautología; por ejemplo,

```
?- es_tautologia((p => q) v (q => p)).
Yes
?- es_tautologia(p => q).
No
```

```
es_tautologia(F) :-
   \+ contramodelo_formula(_I,F).
```

De nuevo notamos es carácter existencial de la variable `_I` que la ponemos como anónima ya que no se usa en ningún otro átomo.

## 3.12 Satisfacibilidad y validez

El **problema de la satisfacibilidad** consiste en dada una fórmula $F$ determinar si es satisfacible. El **problema de la validez** consiste en dada $F$ determinar si es válida. Ambos problemas están relacionados: $F$ es válida syss $\neg F$ es insatisfacible.

Partiendo de dicha relación, se puede dar una definición alternativa de la relación `es_tautologia`

```
es_tautologia_alt(F) :-
   \+ es_satisfacible(-F).
```

Basándose en la misma relación, puede usarse MACE para problemas de validez. Para decidir la validez de una fórmula $F$ se aplica MACE a un fichero cuya lista de fórmulas usables es $\neg F$. Por ejemplo, si $F$ es $(p \rightarrow q) \vee (q \rightarrow p))$, se aplica MACE al fichero cuyo contenido es

```
formula_list(usable).
-((p -> q) | (q -> p)).
end_of_list.
```

y se obtiene

```
The search is complete.  No models were found.
```

lo que significa que la fórmula $\neg F$ es insatisfacible y, por tanto, la fórmula $F$ es válida.

## 3.13 Interpretaciones principales de un conjunto de fórmulas

Una **interpretación principal** de un conjunto de fórmulas $S$ es una aplicación $I$ del conjunto de los símbolos proposicionales que aparecen en $S$ en $\mathbb{B}$.

La relación `interpretaciones_conjunto(+S,-L)` se verifica si `L` es el conjunto de las interpretaciones principales del conjunto `S`: por ejemplo,

```
?- interpretaciones_conjunto([p => q, q=> r],U).
U = [[ (p, 0),  (q, 0),  (r, 0)],
     [ (p, 0),  (q, 0),  (r, 1)],
```

```
[ (p, 0),  (q, 1),  (r, 0)],
[ (p, 0),  (q, 1),  (r, 1)],
[ (p, 1),  (q, 0),  (r, 0)],
[ (p, 1),  (q, 0),  (r, 1)],
[ (p, 1),  (q, 1),  (r, 0)],
[ (p, 1),  (q, 1),  (r, 1)]]
```

```
interpretaciones_conjunto(S,U) :-
   findall(I,interpretacion_conjunto(I,S),U).
```

donde `interpretacion_conjunto(?I,+S)` se verifica si `I` es una interpretación del conjunto de fórmulas `S`; por ejemplo,

```
?- interpretacion_conjunto(I,[p => q, q=> r]).
I = [ (p, 0),  (q, 0),  (r, 0)] ;
I = [ (p, 0),  (q, 0),  (r, 1)] ;
I = [ (p, 0),  (q, 1),  (r, 0)] ;
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 0)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

```
interpretacion_conjunto(I,S) :-
   simbolos_conjunto(S,U),
   interpretacion_simbolos(U,I).
```

donde `interpretacion_simbolos` es la relación definida en [1] (pág. 49) y la relación `simbolos_conjunto(+S,?U)` se verifica si `U` es el conjunto ordenado de los símbolos proposicionales del conjunto de fórmulas `S`; por ejemplo,

```
?- simbolos_conjunto([p => q, q=> r],U).
U = [p, q, r]
```

```
simbolos_conjunto(S,U) :-
   simbolos_conjunto_aux(S,U1),
   sort(U1,U).

simbolos_conjunto_aux([],[]).
simbolos_conjunto_aux([F|S],U) :-
   simbolos_formula(F,U1),
   simbolos_conjunto_aux(S,U2),
   union(U1,U2,U).
```

## 3.14 Modelo de un conjunto de fórmulas

Se dice que una interpretación $I$ es un **modelo del conjunto de fórmulas** $S$, y se representa por $I \models S$, si $I$ es modelo de todas las fórmulas de $S$.

La relación `es_modelo_conjunto(+I,+S)` se verifica si la interpretación `I` es un modelo del conjunto de fórmulas `S`; por ejemplo,

```
?- es_modelo_conjunto([(p,1),(q,0),(r,1)],
                      [(p v q) & (-q v r),q => r]).
Yes
?- es_modelo_conjunto([(p,0),(q,1),(r,0)],
                      [(p v q) & (-q v r),q => r]).
No
```

```
es_modelo_conjunto(_I,[]).
es_modelo_conjunto(I,[F|S]) :-
   es_modelo_formula(I,F),
   es_modelo_conjunto(I,S).
```

## 3.15 Cálculo de modelos de conjuntos de fórmulas

Los **modelos principales de un conjunto de fórmulas** $S$ son las interpretaciones principales de $S$ que son modelos de $S$.

La relación `modelo_conjunto(?I,+S)` se verifica si `I` es un modelo principal del conjunto de fórmulas `S` y `modelos_formula(+S,-L)` se verifica si `L` es el

conjunto de los modelos principales del conjunto de fórmulas `S`; por ejemplo,

```
?- modelo_conjunto(I,[(p v q) & (-q v r),p => r]).
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No

?- modelos_conjunto([(p v q) & (-q v r),p => r],L).
L = [[ (p, 0),  (q, 1),  (r, 1)],
     [ (p, 1),  (q, 0),  (r, 1)],
     [ (p, 1),  (q, 1),  (r, 1)]]
```

```
modelo_conjunto(I,S) :-
   interpretacion_conjunto(I,S),
   es_modelo_conjunto(I,S).

modelos_conjunto(S,L) :-
   findall(I,modelo_conjunto(I,S),L).
```

Una interpretación $I$ es modelo de un conjunto de fórmulas $S$ syss es una extensión de un modelo principal de $S$.

## 3.16 Consistencia de un conjunto de fórmulas

Un conjunto de fórmulas es **consistente** si tiene modelo e **inconsistente** en caso contrario.

La relación `consistente(+S)` se verifica si el conjunto de fórmulas `S` es consistente e `inconsistente(+S)`, si es inconsistente; por ejemplo,

```
?- consistente([(p v q) & (-q v r),p => r]).
Yes
?- consistente([(p v q) & (-q v r),p => r, -r]).
No
?- inconsistente([(p v q) & (-q v r),p => r, -r]).
Yes
?- inconsistente([(p v q) & (-q v r),p => r]).
No
```

```
consistente(S) :-
   modelo_conjunto(_I,S), !.

inconsistente(S) :-
   \+ modelo_conjunto(_I,S).
```

## 3.17 Consistencia con MACE

Con MACE se puede decidir la consistencia de conjuntos de fórmulas $S$ y, en su caso, encontrar un modelo. Por ejemplo, si $S$ es el conjunto $\{(p \vee q) \wedge (-q \vee r), p \to r\}$ se escribe un fichero (p.e. `ej_consistencia.in`) cuyo contenido es

```
formula_list(sos).
(p | q) & (-q | r).
p -> r.
end_of_list.
```

se aplica MACE al fichero

```
> mace -p <ej_consistencia.in >ej_consistencia.out
```

y se obtiene en el fichero de salida

```
=== Model #1 at 0.00 seconds:
 p: T
 q: F
 r: T
end_of_model
Exit by max_models parameter. The set is satisfiable
```

## 3.18 Consecuencia lógica

Se dice que una fórmula $F$ es **consecuencia lógica** de un conjunto de fórmulas $S$, y se representa por $S \models F$, si todos los modelos de $S$ son modelos de $F$. Para comprobar que $S \models F$ basta comprobar que todas las interpretaciones principales de $S \cup \{F\}$ que son modelos de $S$ también son modelos de $F$.

La relación `es_consecuencia(+S,+F)` se verifica si la fórmula `F` es consecuencia del conjunto de fórmulas `S`; por ejemplo,

```
?- es_consecuencia([p => q, q => r], p => r).
Yes
?- es_consecuencia([p], p & q).
No
```

```
es_consecuencia(S,F) :-
   \+ contramodelo_consecuencia(S,F,_I).
```

donde `contramodelo_consecuencia(+S,+F,?I)` se verifica si `I` es una interpretación principal de $\mathtt{S} \cup \{\mathtt{F}\}$ que es modelo del conjunto de fórmulas `S` pero no es modelo de la fórmula `F`; por ejemplo,

```
?- contramodelo_consecuencia([p], p & q, I).
I = [ (p, 1),  (q, 0)] ;
No
?- contramodelo_consecuencia([p => q, q=> r], p => r, I).
No
```

```
contramodelo_consecuencia(S,F,I) :-
   interpretacion_conjunto(I,[F|S]),
   es_modelo_conjunto(I,S),
   \+ es_modelo_formula(I,F).
```

## 3.19 Consecuencia lógica e inconsistencia

Los problemas de consecuencia lógica e inconsistencia son equivalentes en el siguiente sentido: $S \models F$ syss $S \cup \{\neg F\}$ es inconsistente. Además, en el caso de que $S$ sea un conjunto finito $\{F_1, \dots, F_n\}$ las siguientes condiciones son equivalentes

- $\{F_1, \dots, F_n\} \models F$
- $\models F_1 \wedge \cdots \wedge F_n \rightarrow F$
- $\neg(F_1 \wedge \cdots \wedge F_n \rightarrow F)$ es insatisfacible
- $\{F_1, \dots, F_n, \neg F\}$ es inconsistente

Basándonos en la anterior relación podemos dar la siguiente definición alternativa de `es_consecuencia`

```
es_consecuencia_alt(S,F) :-
   inconsistente([-F|S]).
```

## 3.20 Consecuencia lógica con MACE

Con MACE se pueden resolver los problemas de consecuencia lógica. Por ejemplo, para decidir si $p \wedge r$ es consecuencia de $\{p \leftrightarrow q, r \leftrightarrow (p \wedge q)\}$ se aplica MACE al fichero cuyo contenido es

```
formula_list(usable).
p <-> q.
r <-> (p & q).
-(p & r).
end_of_list.
```

y se obtiene

```
==== Model #1 at 0.00 seconds:
 p: F
 q: F
 r: F
end_of_model
Exit by max_models parameter. The set is satisfiable
```

que significa que la fórmula no es consecuencia del conjunto, ya que si se interpretan todos los símbolos como falso se obtiene un modelo del conjunto que no es modelo de la fórmula.

Como segundo ejemplo, para decidir si $p \leftrightarrow r$ es consecuencia del conjunto $\{p \leftrightarrow q, r \leftrightarrow (p \wedge q)\}$ se aplica MACE al fichero cuyo contenido es

```
formula_list(sos).
p <-> q.
r <-> (p & q).
-(p <-> r).
end_of_list.
```

y se obtiene

```
The search is complete.  No models were found.
```

que significa que sí es consecuencia.

## 3.21 Aplicaciones del razonamiento proposicional

En esta sección vamos aplicar los procedimientos anteriores a la resolución de problemas.

**Ejemplo 3.21.1 [El problema de los veraces y los mentirosos]**
En una isla hay dos tribus, la de los veraces (que siempre dicen la verdad) y la de los mentirosos (que siempre mienten). Un viajero se encuentra con tres isleños A, B y C y cada uno le dice una frase

- A dice "B y C son veraces syss C es veraz"
- B dice "Si A y B son veraces, entonces B y C son veraces y A es mentiroso"
- C dice "B es mentiroso syss A o B es veraz"

Determinar a qué tribu pertenecen A, B y C. ([3] pág. 72)

**Solución:** Representaremos por `a`, `b` y `c` que A, B y C son veraces. Por tanto, `-a`, `-b` y `-c` representan que A, B y C son mentirosos. Para determinar las tribus calculamos los modelos del conjunto de fórmulas correspondientes a las tres frases.

```
?- modelos_conjunto([a <=> (b & c <=> c),
                     b <=> (a & c => b & c & -a),
                     c <=> (-b <=> a v b)],
                    L).
L = [[ (a, 1), (b, 1), (c, 0)]]
```

Por tanto, A y B son veraces y C es mentiroso. □

**Ejemplo 3.21.2 [El problema de los animales]**
A partir de la base de conocimiento de los animales (pág 10), demostrar que el animal es una cebra.

**Solución:** Basta comprobar la relación `es_consecuencia`

```
?- es_consecuencia(
     [es_ungulado & tiene_rayas_negras => es_cebra,
```

```
           rumia & es_mamifero => es_ungulado,
           es_mamifero & tiene_pezugnas => es_ungulado,
           es_mamifero,
           tiene_pezugnas,
           tiene_rayas_negras],
          es_cebra).
     Yes
```

□

**Ejemplo 3.21.3 [Problema de los trabajos]**
Juan, Sergio y Carlos trabajan de programador, ingeniero y administrador (aunque no necesariamente en este orden). Juan le debe 1000 euros al programador. La esposa del administrador le ha prohibido a su marido pedir dinero prestado (y éste le obedece). Sergio está soltero. Determinar el trabajo de cada uno. ([15] pág. 192)

**Solución:** Usaremos los siguientes símbolos proposicionales `cp` (Carlos es programador), `ci` (Carlos es ingeniero), `ca` (Carlos es administrador), `jp` (Juan es programador), `ji` (Juan es ingeniero), `ja` (Juan es administrador), `sp` (Sergio es programador), `si` (Sergio es ingeniero) y `sa` (Sergio es administrador).

Calculamos los modelos del conjunto de fórmulas correspondiente al problema (hemos comentado el significado de cada fórmula)

```
?- modelos_conjunto(
        [% Juan es programador, ingeniero o administrador:
         jp v ji v ja,
         % Sergio es programador, ingeniero o administrador:
         sp v si v sa,
         % Carlos es programador, ingeniero o administrador:
         cp v ci v ca,
         % No hay más de un programador:
         (jp & -sp & -cp) v (-jp & sp & -cp) v (-jp & -sp & cp),
         % No hay más de un ingeniero:
         (ji & -si & -ci) v (-ji & si & -ci) v (-ji & -si & ci),
         % No hay más de un administrador:
         (ja & -sa & -ca) v (-ja & sa & -ca) v (-ja & -sa & ca),
         % Juan le debe 1000 pesetas al programador
         % [Luego, Juan no es el programador]:
```

```
          -jp,
          % La esposa del administrador le ha prohibido a
          % su marido pedir dinero prestado (y éste le obedece)
          % [Luego, Juan no es el administrador]:
          -ja,
          % Sergio está soltero [Luego, no es el adminitrador]:
          -sa],
         L).
L = [[(ca,1),(ci,0),(cp,0),
      (ja,0),(ji,1),(jp,0),
      (sa,0),(si,0),(sp,1)]].
```

Tiene un único modelo. Los átomos verdaderos en el modelo son `ca`, `ji` y `sp`, que significan que Carlos es administrador, Juan es ingeniero y Sergio es programador. □

**Ejemplo 3.21.4 [El problema de los cuadrados]**
Existe nueve símbolos proposicionales que se pueden ordenar en un cuadrado:

```
a1  a2  a3
b1  b2  b3
c1  c2  c3
```

Se sabe que existe alguna letra tal que para todos los números las fórmulas son verdaderas (es decir, existe una fila de fórmulas verdaderas). El objetivo de este ejercicio demostrar que para cada número existe una letra cuya fórmula es verdadera (es decir, en cada columna existe una fórmula verdadera).

**Solución:** Basta comprobar que la siguiente fórmula es una tautología

```
?- es_tautologia((a1 & a2 & a3) v
                 (b1 & b2 & b3) v
                 (c1 & c2 & c3)
                 =>
                 (a1 v b1 v c1) v
                 (a2 v b2 v c2) v
                 (a3 v b3 v c3)).
Yes
```

□

**Ejemplo 3.21.5 [Problema del coloreado del pentágono (con dos colores)]**
Demostrar que es imposible colorear los vértices de un pentágono de rojo o azul de forma que los vértices adyacentes tengan colores distintos.

**Solución:** Numeramos los vértices consecutivos del pentágono con los números $1, 2, 3, 4$ y 5. Usamos los símbolos `ri` ($1 \leq$ `i` $\leq 5$) para representar que el vértice `i` es rojo y los símbolos `aj` ($1 \leq$ `j` $\leq 5$) para representar que el vértice `j` es azul. Para resolver el problema basta comprobar que es inconsistente el conjunto de fórmulas que expresa que se ha coloreado los vértices de la forma deseada

```
?- inconsistente(
     [% El vértice i (1 <= i <= 5) es azul o rojo:
      a1 v r1, a2 v r2, a3 v r3, a4 v r4, a5 v r5,

      % Un vértice no puede tener dos colores:
      a1 => -r1, r1 => -a1, a2 => -r2, r2 => -a2,
      a3 => -r3, r3 => -a3, a4 => -r4, r4 => -a4,
      a5 => -r5, r5 => -a5,

      % Dos vértices adyacentes no pueden ser azules:
      -(a1 & a2), -(a2 & a3), -(a3 & a4),
      -(a4 & a5), -(a5 & a1),

      % Dos vértices adyacentes no pueden ser rojos:
      -(r1 & r2), -(r2 & r3), -(r3 & r4),
      -(r4 & r5), -(r5 & r1)]).
Yes
```

□

**Ejemplo 3.21.6 [Problema del coloreado del pentágono (con tres colores)]**
Demostrar que es posible colorear los vértices de un pentágono de rojo, azul o negro de forma que los vértices adyacentes tengan colores distintos.

**Solución:** Con la representación del ejemplo anterior, basta calcular un modelo del conjunto de fórmulas que expresa que se ha coloreado los vértices de la forma deseada

```
?- modelo_conjunto(I,
    [% El vértice i (1 <= i <= 5) es azul, rojo o negro:
     a1 v r1 v n1, a2 v r2 v n2, a3 v r3 v n3,
     a4 v r4 v n4, a5 v r5 v n5,

     % Un vértice no puede tener dos colores:
     a1 => -r1 & -n1, r1 => -a1 & -n1, n1 => -a1 & -r1,
     a2 => -r2 & -n2, r2 => -a2 & -n2, n2 => -a2 & -r2,
     a3 => -r3 & -n3, r3 => -a3 & -n3, n3 => -a3 & -r3,
     a4 => -r4 & -n4, r4 => -a4 & -n4, n4 => -a4 & -r4,
     a5 => -r5 & -n5, r5 => -a5 & -n5, n5 => -a5 & -r5,

     % Dos vértices adyacentes no pueden ser azules:
     -(a1 & a2), -(a2 & a3), -(a3 & a4),
     -(a4 & a5), -(a5 & a1),

     % Dos vértices adyacentes no pueden ser rojos:
     -(r1 & r2), -(r2 & r3), -(r3 & r4),
     -(r4 & r5), -(r5 & r1),

     % Dos vértices adyacentes no pueden ser negros:
     -(n1 & n2), -(n2 & n3), -(n3 & n4),
     -(n4 & n5), -(n5 & n1)]).

I = [ (a1,0),(a2,0),(a3,0),(a4,0),(a5,1),
      (n1,0),(n2,1),(n3,0),(n4,1),(n5,0),
      (r1,1),(r2,0),(r3,1),(r4,0),(r5,0)].
```

El modelo obtenido significa colorear el vértice 1 de rojo, el 2 de negro, el 3 de rojo, el 4 de negro y el 5 de azul. □

**Ejemplo 3.21.7 [Problema del palomar]**
Cuatro palomas comparten tres huecos. Demostrar que dos palomas tienen que estar en la misma hueco.

**Solución:** Mediante las variables `pihj` (`i`$= 1, 2, 3, 4$ y `j`$=1, 2, 3$) representamos que la paloma `i` está en la hueco `j`. Basta demostrar que es inconsistente el

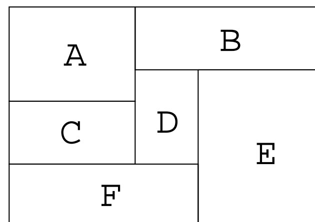
conjunto de fórmulas que indica que cada paloma está en un hueco y que en ningún hueco hay más de una paloma

```
?- inconsistente([% La paloma 1 está en alguna hueco:
                  p1h1 v p1h2 v p1h3,

                  % La paloma 2 está en alguna hueco:
                  p2h1 v p2h2 v p2h3,

                  % La paloma 3 está en alguna hueco:
                  p3h1 v p3h2 v p3h3,

                  % La paloma 4 está en alguna hueco:
                  p4h1 v p4h2 v p4h3,

                  % No hay dos palomas en la hueco 5:
                  -p1h1 v -p2h1,
                  -p1h1 v -p3h1,
                  -p1h1 v -p4h1,
                  -p2h1 v -p3h1,
                  -p2h1 v -p4h1,
                  -p3h1 v -p4h1,

                  % No hay dos palomas en la hueco 6:
                  -p1h2 v -p2h2,
                  -p1h2 v -p3h2,
                  -p1h2 v -p4h2,
                  -p2h2 v -p3h2,
                  -p2h2 v -p4h2,
                  -p3h2 v -p4h2,

                  % No hay dos palomas en la hueco 7:
                  -p1h3 v -p2h3,
                  -p1h3 v -p3h3,
                  -p1h3 v -p4h3,
                  -p2h3 v -p3h3,
                  -p2h3 v -p4h3,
```

```
                  -p3h3 v -p4h3]).
Yes
```

□

**Ejemplo 3.21.8 [Problema de los rectángulos]**
Un rectángulo se divide en seis rectángulos menores como se indica en la figura:




Demostrar que si cada una de los rectángulos menores tiene un lado cuya medida es un número entero, entonces la medida de alguno de los lados del rectángulo mayor es un número entero ([7] pág. 8).

**Solución:** Usamos 14 variables proposicionales (con el significado indicado): `base` (la base del rectángulo mayor es un número entero), `altura` (la altura del rectángulo mayor es un número entero), `base_a` (la base del rectángulo A es un número entero), `altura_a` (la altura del rectángulo A es un número entero), .... Basta comprobar que la fórmula `base v altura` es consecuencia del conjunto de fórmulas que indica las relaciones entre los rectángulos

```
?- es_consecuencia(
     [base_a v altura_a,
      base_b v altura_b,
      base_c v altura_c,
      base_d v altura_d,
      base_e v altura_e,
      base_f v altura_f,

      base_a <=> base_c,

      base_a & base_d => base_f,
      base_f & base_a => base_d,
      base_f & base_d => base_a,
```

```
 base_d & base_e => base_b,
 base_b & base_d => base_e,
 base_b & base_e => base_d,

 base_a & base_b => base,
 base   & base_a => base_b,
 base   & base_b => base_a,

 base_a & base_d & base_e => base,
 base   & base_a & base_d => base_e,
 base   & base_a & base_e => base_d,
 base   & base_d & base_e => base_a,

 base_f & base_e => base,
 base   & base_f => base_e,
 base   & base_e => base_f,

 altura_d & altura_f => altura_e,
 altura_e & altura_d => altura_f,
 altura_e & altura_f => altura_d,

 altura_a & altura_c & altura_f => altura,
 altura   & altura_a & altura_c => altura_f,
 altura   & altura_a & altura_f => altura_c,
 altura   & altura_c & altura_f => altura_a,

 altura_b & altura_d & altura_f => altura,
 altura   & altura_b & altura_d => altura_f,
 altura   & altura_b & altura_f => altura_d,
 altura   & altura_d & altura_f => altura_b,

 altura_b & altura_e => altura,
 altura   & altura_b => altura_e,
 altura   & altura_e => altura_b],
base v altura).
```

```
Yes
```

□

**Ejemplo 3.21.9 [El problema de las 4 reinas]**
Calcular las formas de colocar 4 reinas en un tablero de 4x4 de forma que no haya más de una reina en cada fila, columna o diagonal.

**Solución:** Usaremos los símbolo `cij` ($1 \leq \mathtt{i}, \mathtt{j} \leq 4$) para indicar que hay una reina en la fila `i` columna `j` del tablero. Las soluciones del problema son los modelos del conjunto de fórmulas que establecen las restricciones del problema

```
?- modelos_conjunto([
   % En la 1ª fila hay una reina:
   c11 v c12 v c13 v c14,
   % En la 2ª fila hay una reina:
   c21 v c22 v c23 v c24,
   % En la 3ª fila hay una reina:
   c31 v c32 v c33 v c34,
   % En la 4ª fila hay una reina:
   c41 v c42 v c43 v c44,
   % Si en una casilla hay reina, entonces no hay más reinas en
   % su fila, su columna y su diagonal:
   c11 => (-c12 & -c13 & -c14) &
          (-c21 & -c31 & -c41) &
          (-c22 & -c33 & -c44),
   c12 => (-c11 & -c13 & -c14) &
          (-c22 & -c32 & -c42) &
          (-c21 & -c23 & -c34),
   c13 => (-c11 & -c12 & -c14) &
          (-c23 & -c33 & -c43) &
          (-c31 & -c22 & -c24),
   c14 => (-c11 & -c12 & -c13) &
          (-c24 & -c34 & -c44) &
          (-c23 & -c32 & -c41),
   c21 => (-c22 & -c23 & -c24) &
          (-c11 & -c31 & -c41) &
          (-c32 & -c43 & -c12),
   c22 => (-c21 & -c23 & -c24) &
```

```
          (-c12 & -c32 & -c42) &
          (-c11 & -c33 & -c44  & -c13 & -c31),
   c23 => (-c21 & -c22 & -c24) &
          (-c13 & -c33 & -c43) &
          (-c12 & -c34 & -c14  & -c32 & -c41),
   c24 => (-c21 & -c22 & -c23) &
          (-c14 & -c34 & -c44) &
          (-c13 & -c33 & -c42),
   c31 => (-c32 & -c33 & -c34) &
          (-c11 & -c21 & -c41) &
          (-c42 & -c13 & -c22),
   c32 => (-c31 & -c33 & -c34) &
          (-c12 & -c22 & -c42) &
          (-c21 & -c43 & -c14  & -c23 & -c41),
   c33 => (-c31 & -c32 & -c34) &
          (-c13 & -c23 & -c43) &
          (-c11 & -c22 & -c44  & -c24 & -c42),
   c34 => (-c31 & -c32 & -c33) &
          (-c14 & -c24 & -c44) &
          (-c12 & -c23 & -c43),
   c41 => (-c42 & -c43 & -c44) &
          (-c11 & -c21 & -c31) &
          (-c14 & -c23 & -c32),
   c42 => (-c41 & -c43 & -c44) &
          (-c12 & -c22 & -c32) &
          (-c31 & -c24 & -c33),
   c43 => (-c41 & -c42 & -c44) &
          (-c13 & -c23 & -c33) &
          (-c21 & -c32 & -c34),
   c44 => (-c41 & -c42 & -c43) &
          (-c14 & -c24 & -c34) &
          (-c11 & -c22 & -c33)],
  L),
L = [[(c11, 0), (c12, 0), (c13, 1), (c14, 0),
      (c21, 1), (c22, 0), (c23, 0), (c24, 0),
      (c31, 0), (c32, 0), (c33, 0), (c34, 1),
```

```
 (c41, 0), (c42, 1), (c43, 0), (c44, 0)],
[(c11, 0), (c12, 1), (c13, 0), (c14, 0),
 (c21, 0), (c22, 0), (c23, 0), (c24, 1),
 (c31, 1), (c32, 0), (c33, 0), (c34, 0),
 (c41, 0), (c42, 0), (c43, 1), (c44, 0)]]
```

Gráficamente los modelos son

|   |   | R |   |
|---|---|---|---|
| R |   |   |   |
|   |   |   | R |
|   | R |   |   |

|   | R |   |   |
|---|---|---|---|
|   |   |   | R |
| R |   |   |   |
|   |   | R |   |

□

**Ejemplo 3.21.10 [Problema de Ramsey]**
Probar el caso más simple del teorema de Ramsey: entre seis personas siempre hay (al menos) tres tales que cada una conoce a las otras dos o cada una no conoce a ninguna de las otras dos.

**Solución:** Numeramos las personas mediante $1, 2, 3, 4, 5$ y $6$. Usamos los símbolos proposicionales `pij` ($1 \leq i < j \leq 6$) para indicar que las personas `i` y `j` se conocen. Basta demostrar que la fórmula que indica que hay tres personas que se conocen entre ellas o que hay tres personas tales que cada una desconoce a las otras dos es una tautología

```
?- es_tautologia(% Hay 3 personas que se conocen entre ellas:
                 (p12 & p13 & p23) v
                 (p12 & p14 & p24) v
                 (p12 & p15 & p25) v
                 (p12 & p16 & p26) v
                 (p13 & p14 & p34) v
                 (p13 & p15 & p35) v
                 (p13 & p16 & p36) v
                 (p14 & p15 & p45) v
```

```
                    (p14 & p16 & p46) v
                    (p15 & p16 & p56) v
                    (p23 & p24 & p34) v
                    (p23 & p25 & p35) v
                    (p23 & p26 & p36) v
                    (p24 & p25 & p45) v
                    (p24 & p26 & p46) v
                    (p25 & p26 & p56) v
                    (p34 & p35 & p45) v
                    (p34 & p36 & p46) v
                    (p35 & p36 & p56) v
                    (p45 & p46 & p56) v

                    % Hay 3 personas tales que cada una
                    % desconoce a las otras dos:
                    (-p12 & -p13 & -p23) v
                    (-p12 & -p14 & -p24) v
                    (-p12 & -p15 & -p25) v
                    (-p12 & -p16 & -p26) v
                    (-p13 & -p14 & -p34) v
                    (-p13 & -p15 & -p35) v
                    (-p13 & -p16 & -p36) v
                    (-p14 & -p15 & -p45) v
                    (-p14 & -p16 & -p46) v
                    (-p15 & -p16 & -p56) v
                    (-p23 & -p24 & -p34) v
                    (-p23 & -p25 & -p35) v
                    (-p23 & -p26 & -p36) v
                    (-p24 & -p25 & -p45) v
                    (-p24 & -p26 & -p46) v
                    (-p25 & -p26 & -p56) v
                    (-p34 & -p35 & -p45) v
                    (-p34 & -p36 & -p46) v
                    (-p35 & -p36 & -p56) v
                    (-p45 & -p46 & -p56)).
  Yes
```

□

La siguiente tabla recoge estadísticas sobre los problemas de esta sección. La primera columna identifica el problema, la segunda indica el número de símbolos proposicionales en el problema, la tercera el número de inferencias realizadas en su resolución y la cuarta el tiempo (en segundos) empleado en su resolución.

| Problema | Símbolos | Inferencias | Tiempo |
|---|---:|---:|---:|
| mentirosos | 3 | 646 | 0.00 |
| animales | 6 | 4,160 | 0.00 |
| trabajos | 9 | 71,044 | 0.07 |
| cuadrados | 9 | 56,074 | 0.06 |
| pentágono | 15 | 117,716 | 0.13 |
| palomar | 12 | 484,223 | 0.50 |
| rectángulos | 14 | 1,026,502 | 1.08 |
| 4 reinas | 16 | 15,901,695 | 19.90 |
| Ramsey | 15 | 29,525,686 | 44.27 |

# Capítulo 4

# Sistemas deductivos proposicionales

En el capítulo anterior estudiamos un procedimiento para decidir la validez de las fórmulas calculando el valor de la fórmula en todas sus interpretaciones principales. En este capítulo estudiaremos procedimientos para decidir la validez sin necesidad de construir las interpretaciones y que, además, nos proporcionen pruebas de la validez.

Como textos recomendamos los del capítulo anterior.

## 4.1 Tableros semánticos

### 4.1.1 Ejemplos de demostraciones por tableros semánticos

Veamos un ejemplo de cómo se puede probar la validez de una fórmula.

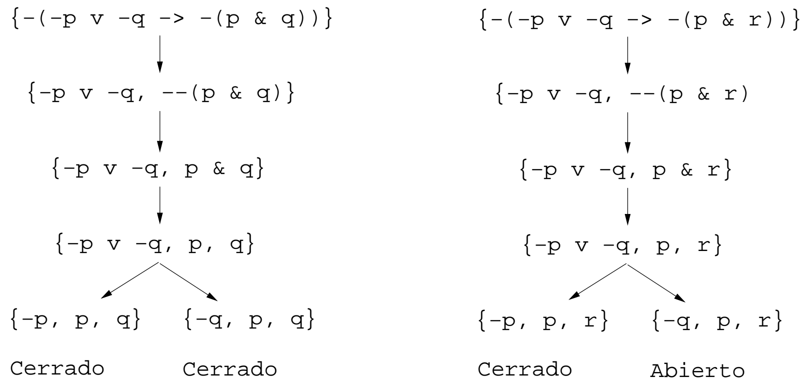
$$
\begin{array}{l}
\neg p \vee \neg q \to \neg(p \wedge q) \text{ es válida} \\
\qquad \text{syss } \{\neg(\neg p \vee \neg q \to \neg(p \wedge q))\} \text{ es inconsistente} \\
\qquad \text{syss } \{\neg p \vee \neg q, \neg\neg(p \wedge q)\} \text{ es inconsistente} \\
\qquad \text{syss } \{\neg p \vee \neg q, p \wedge q\} \text{ es inconsistente} \\
\qquad \text{syss } \{p, q, \neg p \vee \neg q\} \text{ es inconsistente} \\
\qquad \text{syss } \{p, q, \neg p\} \text{ es inconsistente y} \\
\qquad\qquad \{p, q, \neg q\} \text{ es inconsistente}
\end{array}
$$

Por tanto la fórmula es válida. En cambio,

$$\begin{array}{l} \neg p \vee \neg q \to \neg(p \wedge r) \text{ es válida} \\ \quad \text{syss } \{\neg(\neg p \vee \neg q \to \neg(p \wedge r))\} \text{ es inconsistente} \\ \quad \text{syss } \{\neg p \vee \neg q, \neg\neg(p \wedge r)\} \text{ es inconsistente} \\ \quad \text{syss } \{\neg p \vee \neg q, p \wedge r\} \text{ es inconsistente} \\ \quad \text{syss } \{p, q, \neg p \vee \neg r\} \text{ es inconsistente} \\ \quad \text{syss } \{p, q, \neg p\} \text{ es inconsistente y} \\ \qquad \{p, q, \neg r\} \text{ es inconsistente} \end{array}$$

no es válida ya que $\{p, q, \neg r\}$ es consistente.

Los razonamientos anteriores se pueden representar gráficamente mediante los siguientes árboles




El de la izquierda es una demostración de $\neg p \vee \neg q \to \neg(p \wedge q)$ (porque sus dos ramas están cerradas (es decir, contienen una fórmula y su negación)) y el de la derecha muestra que $\neg p \vee \neg q \to \neg(p \wedge q)$ no es válida (porque su rama derecha es abierta (es decir, es un conjunto de literales no contradictorios)).

### 4.1.2 Notación uniforme

Desde el punto de vista sintáctico las fórmulas se clasifican en atómicas, negaciones, conjunciones, disyunciones, implicaciones y equivalencias. Desde el punto de vista semántico nos interesa clasificar las fórmulas en literales, doble negaciones, fórmulas alfas y fórmulas betas.

Un **literal** es un átomo o la negación de un átomo. La relación `literal(+F)` se verifica si la fórmula `F` es un literal.

```
literal(F)  :- atom(F).
literal(-F) :- atom(F).
```

La fórmula $F$ es una **doble negación** si existe una fórmula $G$ tal que $F$ es de la forma $\neg\neg G$. Entonces $\models F \leftrightarrow G$.

Las fórmulas **alfa** , junto con sus componentes, son las indicadas en la siguiente tabla

| | | |
|---|---|---|
| $A_1 \wedge A_2$ | $A_1$ | $A_2$ |
| $\neg(A_1 \rightarrow A_2)$ | $A_1$ | $\neg A_2$ |
| $\neg(A_1 \vee A_2)$ | $\neg A_1$ | $\neg A_2$ |

Si $F$ es una fórmula alfa y sus componentes son $F_1$ y $F_2$, entonces $\models F \leftrightarrow F_1 \wedge F_2$.

La relación `alfa(+A,-A1,-A2)` se verifica si `A` es una fórmula alfa y sus componentes son `A1` y `A2`.

```
alfa(A1 & A2,     A1,  A2).
alfa(-(A1 => A2), A1,  -A2).
alfa(-(A1 v A2),  -A1, -A2).
```

Las fórmulas **beta** , junto con sus componentes, son las indicadas en la siguiente tabla

| | | |
|---|---|---|
| $B_1 \vee B_2$ | $B_1$ | $B_2$ |
| $B_1 \rightarrow B_2$ | $\neg B_1$ | $B_2$ |
| $\neg(B_1 \wedge B_2)$ | $\neg B_1$ | $\neg B_2$ |
| $B_1 \leftrightarrow B_2$ | $B_1 \wedge B_2$ | $\neg B_1 \wedge \neg B_2$ |
| $\neg(B_1 \leftrightarrow B_2)$ | $B_1 \wedge \neg B_2$ | $\neg B_1 \wedge B_2$ |

Si $F$ es una fórmula beta y sus componentes son $F_1$ y $F_2$, entonces $\models F \leftrightarrow F_1 \vee F_2$.

La relación `beta(+B,-B1,-B2)` se verifica si `B` es una fórmula beta y sus componentes son `B1` y `B2`.

```
beta(B1 v B2,      B1,       B2).
beta(B1 => B2,     -B1,      B2).
beta(-(B1 & B2),   -B1,      -B2).
beta(B1 <=> B2,    B1 & B2,  -B1 & -B2).
beta(-(B1 <=> B2), B1 & -B2, -B1 & B2).
```

### 4.1.3 Procedimiento de completación de tableros

Un tablero correspondiente a un conjunto de fórmulas $S$ es un árbol construido mediante las siguientes reglas:

(I) El árbol cuyo único nodo tiene como etiqueta $S$ es un tablero de $S$.

(C) Sea $\mathcal{T}$ un tablero de $S$ y $S_1$ la etiqueta de una hoja de $\mathcal{T}$.

(C.1) Si $S_1$ cerrado (es decir, que contiene una fórmula y su negación), entonces el árbol obtenido añadiendo como hijo de $S_1$ el nodo etiquetado con `cerrado` es un tablero de $S$.

(C.2) Si $S_1$ es abierto (es decir, es un conjunto de literales que no contiene una fórmula y su negación), entonces el árbol obtenido añadiendo como hijo de $S_1$ el nodo etiquetado con `abierto` es un tablero de $S$.

(C.3) Si $S_1$ contiene una doble negación $\neg\neg F$, entonces el árbol obtenido añadiendo como hijo de $S_1$ el nodo etiquetado con $(S_1 - \{\neg\neg F\}) \cup \{F\}$ es un tablero de $S$.

(C.4) Si $S_1$ contiene una fórmula alfa $F$ de componentes $F_1$ y $F_2$, entonces el árbol obtenido añadiendo como hijo de $S_1$ el nodo etiquetado con $(S_1 - \{F\}) \cup \{F_1, F_2\}$ es un tablero de $S$.

(C.5) Si $S_1$ contiene una fórmula beta $F$ de componentes $F_1$ y $F_2$, entonces el árbol obtenido añadiendo como hijos de $S_1$ los nodos etiquetados con $(S_1 - \{F\}) \cup \{F_1\}$ y $(S_1 - \{F\}) \cup \{F_2\}$ es un tablero de $S$.

Un tablero semántico de $S$ es **completo** si no se le puede aplicar ninguna de las reglas de expansión; es decir, todas sus hojas son abiertas o cerradas. La relación `tablero_completo(+S,-Tab)` se verifica si `Tab` es un tablero completo del conjunto de fórmulas `S`; por ejemplo,

```
?- tablero_completo([- (-p v -q => - (p & r))],T).
T = t([- (-p v -q => - (p & r))],
      t([-p v -q, - -(p & r)],
        t([p & r, -p v -q],
          t([p, r, -p v -q],
            t([-p, p, r], cerrado, vacio),
            t([-q, p, r], abierto, vacio)),
          vacio),
```

```
            vacio),
          vacio)
    Yes
```

En el ejemplo se obseva que el término `t(S,Izq,Dcha)` representa el tablero de raíz `S`, rama izquierda `Izq` y rama derecha `Dcha`. Para considerar el tablero como un árbol binario hemos introducido el símbolo `vacio` para completar los nodos que tienen sólo un hijo. La definición de la relación `tablero_completo` es

```
tablero_completo(S,Tab) :-
   completacion(t(S,_Izq,_Dcha),Tab).
```

La relación `completacion(+Tab1,-Tab2)` se verifica si `Tab2` es una completación (i.e. un tablero completo) del tablero `Tab1`

```
completacion(t(Flas,Izq1,Dcha1),t(Flas,Izq3,Dcha3)) :-
   paso(t(Flas,Izq1,Dcha1),t(Flas,Izq2,Dcha2)), !,
   completacion(Izq2,Izq3),
   completacion(Dcha2,Dcha3).
completacion(Tab,Tab).
```

La relación `paso(+Tab1,-Tab2)` se verifica si `Tab2` es un tablero obtenido aplicando una regla de completación al tablero `Tab1`

```
paso(t(S1,_,_),t(S1, cerrado, vacio)) :-     % C1
   cerrada(S1), !.
paso(t(S1,_,_),t(S1, abierto, vacio)) :-     % C2
   lista_de_literales(S1), !.
paso(t(S1,_,_),t(S1, Izq, vacio)) :-         % C3
   regla_doble_negacion(S1, S2), !,
   Izq = t(S2, _, _).
paso(t(S1,_,_),t(S1, Izq, vacio)) :-         % C4
   regla_alfa(S1, S2), !,
   Izq = t(S2, _, _).
paso(t(S1,_,_),t(S1, Izq, Dcha)) :-          % C5
   regla_beta(S1, S2, S3),
   Izq  = t(S2, _, _),
   Dcha = t(S3, _, _).
```

La definición de `paso` se basa en las siguientes relaciones:

- `cerrada(+S)` que se verifica si `S` es una lista cerrada (i.e. que contiene una fórmula y su negación).

```
cerrada(S) :- member(-F,S), member(F,S).
```

- `lista_de_literales(+S)` que se verifica si $S$ es una lista de literales.

```
lista_de_literales([]).
lista_de_literales([F|S]) :-
   literal(F),
   lista_de_literales(S).
```

- `regla_doble_negacion(+S1,-S2)` que se verifica si `S1` es una lista de fórmulas que contiene una doble negación `--F` y `S2` es la lista de fórmulas obtenidas sustituyendo en `S1` la fórmula `--F` por `F`. Por ejemplo,

```
?- regla_doble_negacion([- -(q v r), p => r],L).
L = [q v r, p => r]
?- regla_doble_negacion([p v (q v r), p => r],L).
No
```

```
regla_doble_negacion(S1, [F|S2]) :-
   member(- -F, S1), !,
   delete(S1, - -F, S2).
```

- `regla_alfa(+S1,-S2)` que se verifica si `S1` es una lista de fórmulas que contiene una fórmula alfa `A` y `S2` es la lista de fórmulas obtenidas sustituyendo en `S1` la fórmula `A` por sus componentes. Por ejemplo,

```
?- regla_alfa([p & (q v r), p => r],L).
L = [p, q v r, p => r]
?- regla_alfa([p v (q v r), p => r],L).
No
```

```
regla_alfa(S1, [A1,A2|S2]) :-
   member(A, S1),
   alfa(A, A1, A2), !,
   delete(S1, A, S2).
```

- `regla_beta(+S1,-S2,-S3)` que se verifica si `S1` es una lista de fórmulas que contiene al menos una fórmula beta `B` y `S2` es la lista de fórmulas obtenidas sustituyendo en `S1` la fórmula `B` por una de sus componentes y `S3`, por la otra. Por ejemplo,

```
?- regla_beta([p & (q v r), p => r],L1,L2).
L1 = [-p, p & (q v r)]
L2 = [r, p & (q v r)]
?- regla_beta([p & (q v r), -(p => r)],L1,L2).
No
```

```
regla_beta(S1, [B1|RS1], [B2|RS1]) :-
   member(B, S1),
   beta(B, B1, B2),
   delete(S1, B, RS1).
```

### 4.1.4 Tableros cerrados

Un tablero es **cerrado** si todas sus hojas están etiquetadas con `cerrado` o `vacio`. La relación `es_cerrado(+Tab)` se verifica si `Tab` es un tablero cerrado.

```
es_cerrado(t(_,Izq,Dcha)) :-
   es_cerrado(Izq),
   es_cerrado(Dcha).
es_cerrado(cerrado).
es_cerrado(vacio).
```

### 4.1.5 Teorema por tableros

Una fórmula $F$ es un **teorema (mediante tableros semánticos)** (y se representa por $\vdash_{Tab} F$) si tiene una prueba mediante tableros; es decir, si $\{\neg F\}$ tiene un tablero completo cerrado.

El cálculo de tableros semánticos es adecuado y completo; es decir, una fórmula es válida ($\models F$) syss es teorema mediante tableros semánticos ($\vdash_{Tab} F$).

La relación `prueba(+F,-Tab)` se verifica si `Tab` es una prueba (mediante tableros semánticos) de la fórmula `F`. Por ejemplo,

```
?- prueba(-p v -q => -(p & q),T).
T = t([- (-p v-q => - (p & q))],
      t([-p v-q, - -(p & q)],
        t([p&q, -p v -q],
          t([p, q, -p v -q],
            t([-p, p, q], cerrado, vacio),
            t([-q, p, q], cerrado, vacio)),
          vacio),
        vacio),
      vacio)
?- prueba(-p v -q => -p,T).
No
```

```
prueba(F,Tab) :-
   tablero_completo([-F],Tab), !,
   es_cerrado(Tab).
```

La relación `es_teorema(+F)` se verifica si la fórmula `F` es teorema (mediante tableros semánticos). Por ejemplo,

```
?- es_teorema(-p v -q => -(p & q)).
Yes
?- es_teorema(-p v -q => -(p & r)).
No
```

```
es_teorema(F) :-
   prueba(F,_Tab).
```

### 4.1.6 Deducción por tableros

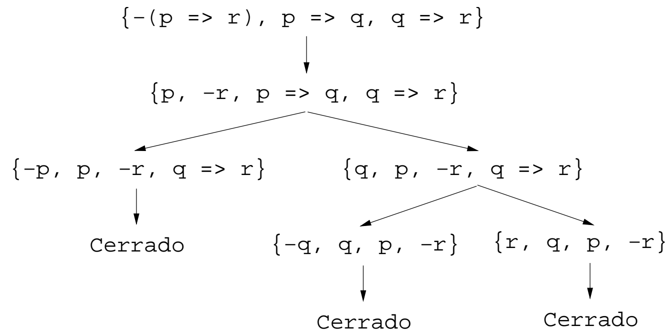
La fórmula $F$ es **deducible (mediante tableros semánticos)** a partir del conjunto de fórmulas $S$ (y se representa por $S \vdash_{Tab} F$) si existe una prueba mediante tableros de que $F$ a partir de $S$; es decir, existe un tablero completo cerrado de $S \cup \{\neg F\}$.

La fórmula $F$ es consecuencia de $S$ syss $F$ es deducible mediante tableros a partir de $S$; es decir, $S \models F$ syss $S \vdash_{Tab} F$.

La relación `prueba_deducible_tab(+S,+F,-Tab)` se verifica si `Tab` es una prueba por tableros semánticos de que la fórmula `F` es deducible del conjunto de fórmulas S; por ejemplo,

```
?- prueba_deducible_tab([p => q, q => r], p => r,T).
T = t([- (p => r), p => q, q => r],
      t([p, -r, p => q, q => r],
        t([-p, p, -r, q => r], cerrado, vacio),
        t([q, p, -r, q => r],
          t([-q, q, p, -r], cerrado, vacio),
          t([r, q, p, -r], cerrado, vacio))),
      vacio)
Yes
?- prueba_deducible_tab([p => q, q => r], p <=> r,T).
No
```

Gráficamente, la prueba anterior es




La definición de `prueba_deducible_tab` es

```
prueba_deducible_tab(S,F,Tab) :-
   tablero_completo([-F|S],Tab), !,
   es_cerrado(Tab).
```

La relación `es_deducible_tab(+S,+F)` se verifica si la fórmula `F` es deducible (mediante tableros) del conjunto de fórmulas `S`.

```
es_deducible_tab(S,F) :-
   prueba_deducible_tab(S,F,_Tab).
```

## 4.2 Cláusulas

El objetivo de esta sección es la transformación de las fórmulas en una forma equivalente adecuada para el cálculo de resolución.

### 4.2.1 Equivalencia lógica

Las fórmulas $F$ y $G$ son **equivalentes** (y se representa por $F \equiv G$) si $\models F \leftrightarrow G$. La relación `es_equivalente(+F,+G)` se verifica si las fórmulas `F` y `G` son equivalentes; por ejemplo,

```
?- es_equivalente(-(p & q), -p v -q).
Yes
?- es_equivalente(-(p & q), -p & -q).
No
```

```
es_equivalente(F,G) :-
   es_tautologia(F <=> G).
```

### 4.2.2 Forma normal negativa

Una fórmula **está en forma normal negativa** si no contiene las conectivas $\rightarrow$, $\leftrightarrow$ y la negación no se aplica a subfórmulas compuestas. Por ejemplo, $(\neg p \vee q) \wedge (\neg q \vee p)$ está en forma normal negativa y $(p \rightarrow q) \wedge (q \rightarrow p)$ y $\neg(p \wedge q)$ no están forma normal negativa.

Una fórmula $G$ **es una forma normal negativa** de la fórmula $F$ si $G$ está en forma normal negativa y es equivalente a $F$. Por ejemplo, una forma normal negativa de $p \leftrightarrow q$ es $(\neg p \vee q) \wedge (\neg q \vee p)$.

Aplicando a una fórmula $F$ los siguientes pasos se obtiene una forma normal negativa de $F$:

1. Eliminar las equivalencias usando la relación

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \quad (1)$$

2. Eliminar las implicaciones usando la equivalencia
$$A \to B \equiv \neg A \vee B \tag{2}$$

3. Interiorizar las negaciones usando las equivalencias
$$\neg(A \wedge B) \equiv \neg A \vee \neg B \tag{3}$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B \tag{4}$$
$$\neg\neg A \equiv A \tag{5}$$

La relación `fnn(+F,?G)` se verifica si `G` es una forma normal negativa de la fórmula `F`; por ejemplo,

```
?- fnn(p <=> q,F).
F = (-p v q) & (-q v p)
?- fnn(p v -q => r, F).
F = (-p & q) v r
?- fnn(p v -q => r, (-p & q) v r).
Yes
?- fnn(p v -q => r, -p & (q v r)).
No
?- fnn(p & (q => r) => s, F).
F = (-p v (q & -r)) v s
```

```
fnn(F, G) :-
   elimina_equivalencias(F,F1),
   elimina_implicaciones(F1,F2),
   interioriza_negaciones(F2,G).
```

donde la relación `elimina_equivalencias(+F,?G)` se verifica si `G` es la fórmula obtenida eliminando las equivalencias de la fórmula `F`, usando la reducción (1).

```
elimina_equivalencias(A <=> B, (A1 => B1) & (B1 => A1)) :- !,
   elimina_equivalencias(A, A1),
   elimina_equivalencias(B, B1).
elimina_equivalencias(A, B) :-
   A =.. [Op|L1], !,
   maplist(elimina_equivalencias,L1,L2),
   B =.. [Op|L2].
elimina_equivalencias(A, A).
```

la relación `elimina_implicaciones(+F,?G)` se verifica si `G` es la fórmula obtenida eliminando las implicaciones de la fórmula `F`, usando la reducción (2)

```
elimina_implicaciones(A => B, -A1 v B1) :- !,
   elimina_implicaciones(A, A1),
   elimina_implicaciones(B, B1).
elimina_implicaciones(A, B) :-
   A =.. [Op|L1], !,
   maplist(elimina_implicaciones,L1,L2),
   B =.. [Op|L2].
elimina_implicaciones(A, A).
```

la relación `interioriza_negaciones(+F,?G)` se verifica si `G` es la fórmula obtenida interiorizando las negaciones de la fórmula `F` (que no tiene `=>` ni `<=>`) de forma que las negaciones se apliquen sólo sobre símbolos proposicionales. Las reglas de interiorización son las reduciones (3), (4) y (5).

```
interioriza_negaciones(-(A & B), A1 v B1) :- !,
   interioriza_negaciones(-A, A1),
   interioriza_negaciones(-B, B1).
interioriza_negaciones(-(A v B), A1 & B1) :- !,
   interioriza_negaciones(-A, A1),
   interioriza_negaciones(-B, B1).
interioriza_negaciones((-(-A)),  A1) :- !,
   interioriza_negaciones(A, A1).
interioriza_negaciones(A, B) :-
   A =.. [Op|L1], !,
   maplist(interioriza_negaciones,L1,L2),
   B =.. [Op|L2].
interioriza_negaciones(A, A).
```

### 4.2.3 Forma normal conjuntiva

Un **literal positivo** es una fórmula atómica y un **literal negativo** es la negación de una fórmula atómica. Un **literal** es un literal positivo o negativo. Usaremos $L, L_1, L_2, \dots$ como variables para literales.

Una fórmula está en **forma normal conjuntiva** si es una conjunción de disyunciones de literales; es decir, es de la forma

$$(L_{1,1} \vee \cdots \vee L_{1,n_1}) \wedge \cdots \wedge (L_{m,1} \vee \cdots \vee L_{m,n_m}).$$

Por ejemplo, $(\neg p \vee q) \wedge (\neg q \vee p)$ está en forma normal conjuntiva y $(\neg p \vee q) \wedge (q \rightarrow p)$ no está en forma normal conjuntiva.

Una fórmula $G$ **es una forma normal conjuntiva** de la fórmula $F$ si $G$ está en forma normal conjuntiva y es equivalente a $F$. Por ejemplo, una forma normal conjuntiva de $\neg(p \wedge (q \rightarrow r))$ es $(\neg p \vee q) \wedge (\neg p \vee \neg r)$.

Aplicando a una fórmula $F$ los siguientes pasos se obtiene una forma normal conjuntiva de $F$:

1. Calcular una forma normal negativa de $F$.

2. Interiorizar las disyunciones usando la propiedad distributiva de la disyunción sobre la conjunción

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \quad (6)$$

$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C) \quad (7)$$

La relación `fnc(+F,?G)` se verifica si `G` es una forma normal conjuntiva de la fórmula `F`; por ejemplo,

```
?- fnc(p & (q => r), F).
F = p& (-q v r)
?- fnc(-(p & (q => r)), F).
F = (-p v q)& (-p v-r)
```

```
fnc(F,G) :-
   fnn(F,F1),
   interioriza_disyunciones(F1,G).
interioriza_disyunciones(A v (B & C), ABC) :- !,
   interioriza_disyunciones(A v B, AB),
   interioriza_disyunciones(A v C, AC),
   interioriza_disyunciones(AB & AC, ABC).
interioriza_disyunciones((A & B) v C, ABC) :- !,
   interioriza_disyunciones(A v C, AC),
   interioriza_disyunciones(B v C, BC),
   interioriza_disyunciones(AC & BC, ABC).
```

```
interioriza_disyunciones(A, B) :-
   A =.. [Op|L],
   maplist(interioriza_disyunciones,L,L1),
   ( L1 = L ->
     B =.. [Op|L1]
   ; % not(L1 = L) ->
     B1 =.. [Op|L1],
     interioriza_disyunciones(B1,B)).
interioriza_disyunciones(A, A).
```

### 4.2.4 Transformación a cláusulas

Una **cláusula** es un conjunto de literales. Usaremos los símbolos $C, C_1, C_2, \dots$ como variables sobre cláusulas. El **valor de una cláusula** $C$ en una interpretación $I$ es

$$I(C) = \begin{cases} 1, & \text{si existe un } L \in C \text{ tal que } I(L) = 1 \\ 0, & \text{en caso contrario.} \end{cases}$$

Una fórmula $F$ y una cláusula $C$ son **equivalentes** si $I(F) = I(C)$ para cualquier interpretación $I$. Si $C = \{L_1, L_2, \dots, L_n\}$, entonces $C$ es equivalente a $L_1 \lor L_2 \lor \dots \lor L_n$. La **cláusula vacía** es el conjunto vacío de cláusulas y se representa por $\square$. En cualquier interpretación $I$, $I(\square) = 0$.

Las **fórmulas clausales** son disyunciones de literales; es decir, las fórmulas obtenidas mediante las siguientes reglas

- Si $F$ es un literal, entonces $F$ es una fórmula clausal
- Si $F$ y $G$ son fórmulas clausales, entonces $F \lor G$ es una fórmula clausal.

Por ejemplo, $p$, $\neg p$ y $\neg p \lor (q \lor \neg r)$ son fórmulas clausales y $\neg p \lor (q \land \neg r)$ no es una fórmula clausal.

El siguiente procedimiento transforma fórmulas clausales en cláusulas equivalentes

$$\text{Cláusula}(F) = \begin{cases} \{F\}, & \text{si } F \text{ es un literal;} \\ \text{Cláusula}(F_1) \cup \text{Cláusula}(F_2), & \text{si } F = (F_1 \lor F_2) \end{cases}$$

La relación `clausula(+F,-C)` se verifica si `C` es una cláusula equvalente a la fórmula clausal `F`; por ejemplo,

```
?- clausula(p,C).
C = [p]
?- clausula(-p,C).
C = [-p]
?- clausula((-p v r) v (-p v q),C).
C = [q, r, -p]
```

```
clausula(L1 v L2, S) :- !,
   clausula(L1, S1),
   clausula(L2, S2),
   append(S1, S2, S3),
   sort(S3,S).
clausula(L, [L]).
```

El **valor de un conjunto de cláusulas** $S$ en una interpretación $I$ es

$$I(S) = \begin{cases} 1, & \text{si para toda } C \in S,\ I(C) = 1 \\ 0, & \text{en caso contrario.} \end{cases}$$

Una interpretación $I$ es **modelo de un conjunto de cláusulas** $S$ si $I(S) = 1$. Un conjunto de cláusulas es **inconsistente** si no tiene modelos.

Una fórmula $F$ y un conjunto de cláusulas $S$ son **equivalentes** si $I(F) = I(S)$ para cualquier interpretación $I$. Si $S = \{C_1, C_2, \dots, C_n\}$, entonces $S$ es equivalente a $C_1 \wedge C_2 \wedge \dots \wedge C_n$.

El siguiente procedimiento transforma fórmulas en forma normal conjutiva en conjuntos cláusulas equivalentes

$$\text{Cláu–FNC}(F) = \begin{cases} \text{Cláu–FNC}(F_1) \cup \text{Cláu–FNC}(F_2), & \text{si } F = F_1 \wedge F_2 \\ \{\text{Cláusula}(F)\}, & \text{en caso contrario} \end{cases}$$

La relación `clausulas_FNC(+F,?S)` se verifica si `S` es un conjunto de cláusulas equivalente a la fórmula en forma normal conjuntiva `F`; por ejemplo,

```
?- clausulas_FNC(p & (-q v r), S).
S = [[p], [r, -q]]
?- clausulas_FNC((-p v q) & (-p v -r), S).
S = [[q, -p], [-p, -r]]
```

```
clausulas_FNC(A1 & A2, S) :- !,
   clausulas_FNC(A1, S1),
   clausulas_FNC(A2, S2),
   union(S1, S2, S).
clausulas_FNC(A, [S]) :-
   clausula(A, S).
```

La relación `clausulas(+F,?S)` se verifica si `S` es un conjunto de cláusulas equivalente a la fórmula `F`; por ejemplo,

```
?- clausulas(p & (q => r),S).
S = [[p], [r, -q]]
```

```
clausulas(F,S) :-
   fnc(F,F1),
   clausulas_FNC(F1,S).
```

El conjunto de fórmulas $S_1$ y el conjunto de cláusulas $S_2$ son equivalentes si $I(S_1) = I(S_2)$ para cualquier interpretación $I$.

La relación `clausulas_conjunto(+CF,-CC)` se verifica si `CC` es un conjunto de cláusulas equivalente al conjunto de fórmulas `CF`; por ejemplo,

```
?- clausulas_conjunto([p => q, q => r],CC).
CC = [[q, -p], [r, -q]]
```

```
clausulas_conjunto([],[]).
clausulas_conjunto([F|CF],CC) :-
   clausulas(F,CC1),
   clausulas_conjunto(CF,CC2),
   union(CC1,CC2,CC).
```

### 4.2.5 Transformación a cláusulas con OTTER

Con OTTER se puede calcular los conjuntos de cláusulas equivalentes a conjuntos de fórmulas, Por ejemplo, para calcular un conjunto de cláusulas equivalente al conjunto de fórmulas $\{p \rightarrow q, q \rightarrow r\}$ se escribe un fichero (por ejemplo, `ej_clausulas.in`) cuyo contenido es

```
formula_list(sos).
p <-> q.
q -> r.
end_of_list.
```

se aplica OTTER al fichero de entrada (`ej_clausulas.in`)

```
> otter <ej_clausulas.in >ej_clausulas.out
```

y en el fichero de salida (`ej_clausulas.out`) se encuentra

```
-------> sos clausifies to:
list(sos).
1 [] -p|q.
2 [] p| -q.
3 [] -q|r.
end_of_list.
```

que corresponden a las tres cláusulas del conjunto $\{\{\neg p, q\}, \{p, \neg q\}, \{\neg q, r\}\}$ equivalente al de las fórmulas iniciales.

## 4.3 Resolución

### 4.3.1 Motivación de resolución

El problema de decidir si una fórmula $F$ es consecuencia lógica de un conjunto de fórmulas $S$ puede reducirse al de decidir la inconsistencia de un conjunto de cláusulas, ya que las siguientes condiciones son equivalentes:

- $S \models F$
- $S \cup \{\neg F\}$ es inconsistente
- Cláusulas($S \cup \{\neg F\}$) es inconsistente

Puesto que un conjunto de cláusulas $S$ es inconsistente syss la cláusula vacía es consecuencia de $S$, para decidir si $S$ es inconsistente basta generar consecuencias de $S$ hasta que se obtenga la cláusula vacía.

Antes de precisar la regla de inferencia a utilizar, observemos distintas reglas de inferencia y su representación mediante cláusulas:

- Modus Ponens

$$\frac{p \to q, \quad p}{q} \text{ [MP]} \qquad \frac{\{\neg p, q\}, \quad \{p\}}{\{q\}} \text{ [MP]}$$

- Modus Tollens

$$\frac{p \to q, \quad \neg q}{\neg p} \text{ [MT]} \qquad \frac{\{\neg p, q\}, \quad \{\neg q\}}{\{\neg p\}} \text{ [MT]}$$

- Encadenamiento

$$\frac{p \to q, \quad q \to r}{p \to r} \text{ [Encad]} \qquad \frac{\{\neg p, q\}, \quad \{\neg q, r\}}{\{\neg p, r\}} \text{ [Encad]}$$

Se observa que desde el punto de vista de cláusulas las tres reglas anteriores están incluida en la regla

$$\frac{\{p_1, \dots, r, \dots, p_m\}, \quad \{q_1, \dots, \neg r, \dots, q_n\}}{\{p_1, \dots, p_m, q_1, , \dots, q_n\}} \text{ [Resolución]}$$

que es la regla de resolución proposicional.

### 4.3.2 Regla de resolución proposicional

El **complementario de un literal** $L$ es

$$\overline{L} = \begin{cases} p, & \text{si } L = \neg p \\ \neg p, & \text{si } L = p \end{cases}$$

La relación `complementario(+L1,-L2)` se verifica si `L2` es el complementario del literal `L1`.

```
complementario(-A, A) :- !.
complementario(A, -A).
```

La cláusula $C$ es una **resolvente** de las cláusulas $C_1$ y $C_2$ si existe un literal $L$ tal que $L \in C_1$, $\overline{L} \in C_2$ y $C = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\}$.

La relación `resolvente(+C1,+C2,-C3)` se verifica si `C3` es una resolvente de las cláusulas `C1` y `C2`; por ejemplo,

```
?- resolvente([p,q],[-q,r],C).
C = [p, r] ;
No
?- resolvente([q, -p],[p, -q],C).
C = [p, -p] ;
C = [q, -q] ;
No
?- resolvente([q, -p], [p, q], C).
C = [q] ;
No
?- resolvente([q, -p], [q, r], C).
No
?- resolvente([p],[-p],C).
C = [] ;
No
```

```
resolvente(C1,C2,C) :-
   member(L1,C1),
   complementario(L1,L2),
   member(L2,C2),
   delete(C1, L1, C1P),
   delete(C2, L2, C2P),
   append(C1P, C2P, C3),
   sort(C3,C).
```

### 4.3.3 Demostraciones por resolución

Sea $S$ un conjunto de cláusulas. La sucesión $(C_1, \dots, C_n)$ es una **demostración por resolución** de la cláusula $C$ a partir de $S$ si $C = C_n$ y para todo $i \in \{1, ..., n\}$ se verifica una de las siguientes condiciones:

- $C_i \in S$;
- existen $j, k < i$ tales que $C_i$ es una resolvente de $C_j$ y $C_k$)

La cláusula $C$ es **demostrable por resolución** a partir de $S$ (y se representa por $S \vdash_{res} C$) si existe una demostración por resolución de $C$ a partir de $S$. Una **refutación por resolución** de $S$ es una demostración por resolución de

la cláusula vacía a partir de $S$. Se dice que $S$ es **refutable por resolución** si $S \vdash_{res} \square$

Sea $S$ un conjunto de fórmulas. Una **demostración por resolución** de $F$ a partir de $S$ es una refutación por resolución de Cláusulas($S \cup \{\neg F\}$). La fórmula $F$ es **demostrable por resolución** a partir de $S$ (y se representa por $S \vdash_{res} F$) si existe una demostración por resolución de $F$ a partir de $S$.

Una demostración por resolución de $p \wedge q$ a partir de $\{p \vee q, p \rightarrow q\}$ es

```
1 {p,q}    Hipótesis
2 {-p,q}   Hipótesis
3 {p,-q}   Hipótesis
4 {-p,-q}  Hipótesis
5 {q}      Resolvente de 1 y 2
7 {-q}     Resolvente de 3 y 4
8 {}       Resolvente de 5 y 7
```

El cálculo por resolución es adecuado y completo; es decir que dado un conjunto de fórmulas $S$ y una fórmula $F$, se tiene que $S \models F$ syss $S \vdash_{res} F$.

### 4.3.4 Procedimiento elemental de búsqueda de refutación

Un procedimiento elemental de búsqueda de una refutación de un conjunto de cláusulas $S$ consta de las siguientes reglas

- Si $S$ contiene a la cláusula vacía, entonces $S$ es inconsistente.
- Si $S$ contiene dos cláusulas $C_1$, $C_2$ que tienen una resolvente que no pertenece a $S$, entonces $S$ es inconsistente syss $S \cup \{C\}$ es inconsistente.
- En otro caso, $S$ es consistente.

La relación `refutacion(+S,-R)` se verifica si `R` es una refutación por resolución del conjunto de cláusulas `S`. Por ejemplo,

```
?- refutacion([[-p,-q],[p,q],[-p,q],[-q,p]],R).
R = [[p,-q],[q,-p],[p,q],[-p,-q],
     [q,-q],[p,-p],[-p],[q],[p],[]]
Yes
?- refutacion([[p,q],[-p,q],[-q,p]],R).
No
```

```
refutacion(S,R) :-
   maplist(sort,S,S1),
   refutacion_aux(S1,R).

refutacion_aux(S,R) :-
   member([],S), !,
   reverse(S,R).
refutacion_aux(S,R) :-
   member(C1,S),
   member(C2,S),
   resolvente(C1,C2,C),
   \+ member(C, S),
   % format('~N~w resolvente de ~w y ~w~n',[C,C1,C2]),
   refutacion_aux([C|S],R).
```

Si se elimina el comentario, se escriben las resolventes que se van obteniendo. Por ejemplo,

```
?- refutacion([[-p,-q],[p,q],[-p,q],[-q,p]],R).
[q, -q] resolvente de [-p, -q] y [p, q]
[p, -p] resolvente de [-p, -q] y [p, q]
[-p] resolvente de [-p, -q] y [q, -p]
[q] resolvente de [-p] y [p, q]
[p] resolvente de [q] y [p, -q]
[] resolvente de [p] y [-p]

R = [[p,-q],[q,-p],[p,q],[-p,-q],
     [q,-q],[p,-p],[-p],[q],[p],[]]

Yes
```

### 4.3.5 Resolución con OTTER

Par estudiar el procedimiento de búsqueda de refutaciones de OTTER vamos a analizar la refución correspondiente al ejemplo anterior. El contenido del fichero de entrada es

```
list(sos).
-p | -q.
p | q.
-p | q.
-q | p.
end_of_list.

set(binary_res).    % Resolución binaria
set(very_verbose).  % Presentación detallada
```

en el que se ha escrito las cláusulas en el conjunto soporte y se ha indicado la regla de inferencia y el nivel de detalle de la presentación

En el fichero de salida observamos que en primer lugar se han anotados las cláusulas

```
list(sos).
1 [] -p| -q.
2 [] p|q.
3 [] -p|q.
4 [] -q|p.
end_of_list.
```

a continuación se ha realizado la búsqueda de la refutación

```
=========== start of search ===========

given clause #1: (wt=2) 1 [] -p| -q.

given clause #2: (wt=2) 2 [] p|q.

  0 [binary,2.1,1.1] q| -q.

  0 [binary,2.2,1.2] p| -p.

given clause #3: (wt=2) 3 [] -p|q.

  0 [binary,3.1,2.1] q|q.
** KEPT (pick-wt=1): 5 [binary,3.1,2.1,factor_simp] q.
```

```
  0 [binary,3.2,1.2] -p| -p.
** KEPT (pick-wt=1): 6 [binary,3.2,1.2,factor_simp] -p.

given clause #4: (wt=1) 5 [binary,3.1,2.1,factor_simp] q.

  0 [binary,5.1,1.2] -p.
  Subsumed by 6.

given clause #5: (wt=1) 6 [binary,3.2,1.2,factor_simp] -p.

  0 [binary,6.1,2.1] q.
  Subsumed by 5.

given clause #6: (wt=2) 4 [] -q|p.

  0 [binary,4.1,5.1] p.
** KEPT (pick-wt=1): 7 [binary,4.1,5.1] p.

----> UNIT CONFLICT at   0.00 sec ----> 8 [binary,7.1,6.1] $F.
```

Finalmente muestra la prueba obtenida

```
---------------- PROOF ----------------
1 [] -p| -q.
2 [] p|q.
3 [] -p|q.
4 [] -q|p.
5 [binary,3.1,2.1,factor_simp] q.
6 [binary,3.2,1.2,factor_simp] -p.
7 [binary,4.1,5.1] p.
8 [binary,7.1,6.1] $F.
------------ end of proof -------------
```

En el procedimiento de búsqueda se consideran dos conjuntos de cláusulas: usable y soporte. En la elección de cláusula se considera el peso, que es el número de símbolos proposicionales en la cláusula. Además se eliminan cláusulas subsumidas (la cláusula $C$ subsume a la cláusula $D$ si $D \subseteq C$) y

tautológicas (una cláusula es una tautología si contiene un literal y su complementario).

La búsqueda se ha realizado siguiendo el siguiente procedimiento

```
    Mientras el soporte es no vacío y no se ha encontrado una
    refutación
       1. Seleccionar como cláusula actual la cláusula menos
          pesada del soporte;
       2. Mover la cláusula actual del soporte a usable;
       3. Calcular las resolventes de la cláusula actual con
          las cláusulas usables.
       4. Procesar cada una de las resolventes calculadas
          anteriormente.
       5. Añadir al soporte cada una de las cláusulas
          procesadas que supere el procesamiento.

 El procesamiento de cada una de resolventes consta de los
 siguientes pasos (los indicados con * son opcionales):
    *  1. Escribir la resolvente.
    *  2. Aplicar a la resolvente eliminación unitaria (i.e.
          elimina los literales de la resolvente tales que hay
          una cláusula unitaria complementaria en usable o
          en soporte).
       3. Descartar la resolvente y salir si la resolvente
          es una tautología.
       4. Descartar la resolvente y salir si la resolvente
          es subsumida por alguna cláusula de usable o del
          soporte (subsunción hacia adelante).
       5. Añadir la resolvente al soporte.
    *  6. Escribir la resolvente retenida.
       7. Si la resolvente tiene 0 literales, se ha
          encontrado una refutación.
       8. Si la resolvente tiene 1 literal, entonces buscar
          su complementaria (refutación) en usable y soporte.
    *  9. Escribir la demostración si se ha encontrado una
          refutación.
    -------------
```

```
* 10. Descartar cada cláusula de usable o del soporte
      subsumida por la resolvente (subsunción hacia atrás).

El paso 10 no se da hasta que los pasos 1-9 se han aplicado
a todas las resolventes.
```

### 4.3.6 Resolución de OTTER en Prolog

El procedimiento de resolución de OTTER puede implementarse en Prolog. La relación `escribe_refutacion(+U,+S)` se verifica si se existe una refutación con usables `U` y soporte `S`, en cuyo caso escribe una refutación. Por ejemplo,

```
?- escribe_refutacion([], [[p,q],[-p,q],[-q,p],[-p,-q]]).

Usable:

Soporte:
1 [] [p, q]
2 [] [-p, q]
3 [] [-q, p]
4 [] [-p, -q]

cláusula actual #1: 1 [] [p, q]

cláusula actual #2: 2 [] [-p, q]

  0 [2, 1] [q]
** RETENIDA: 5 [2, 1] [q]
5 subsume a 2
5 subsume a 1

cláusula actual #3: 5 [2, 1] [q]

cláusula actual #4: 3 [] [-q, p]

  0 [3, 5] [p]
** RETENIDA: 6 [3, 5] [p]
```

```
6 subsume a 3

cláusula actual #5: 6 [3, 5] [p]

cláusula actual #6: 4 [] [-p, -q]

  0 [4, 6] [-q]
** RETENIDA: 7 [4, 6] [-q]

 ----> CONFLICTO UNITARIO 8 [7,5] []

---------------- DEMOSTRACION ----------------
 1      [] [p, q]
 2      [] [-p, q]
 3      [] [-q, p]
 4      [] [-p, -q]
 5  [2, 1] [q]
 6  [3, 5] [p]
 7  [4, 6] [-q]
 8  [7, 5] []
----------- fin de la demostración -----------

Yes
```

Su definición es

```
escribe_refutacion(Usable,Soporte) :-
   refutacion(Usable,Soporte,Ref),
   escribe_prueba(Ref).
```

donde `refutacion(+Usable,+Soporte,-Ref)` se verifica si `Ref` es una refutación por resolución del conjunto de cláusulas de `Usable` y `Soporte` y `escribe_prueba(+Ref)` escribe la prueba `Ref`. La definición de `refutacion` es

```
refutacion(Usable,Soporte,Ref) :-
   inicia,                                       % 1
   format('~N~nUsable:~n',[]),                   % 2
   anotado(Usable,Usable1),
```

```
    format('~N~nSoporte:~n',[]),                % 3
    anotado(Soporte,Soporte1),
    ordenada_por_peso(Soporte1,Soporte2),       % 4
    refutacion_anotada(Usable1,Soporte2,[],Ref).  % 5
```

inicia el contador de cláusulas (1), anota las cláusulas de usable (2) y soporte (3), ordena las cláusulas del soporte por peso (4) y busca una refutación con las cláusulas anotadas (5). Una cláusula anotada es una expresión de la forma `N*H*C` donde `N` es el número de la cláusula, `H` es su historia (lista de padres) y `C` es la cláusula.

La definición de `refutacion_anotada` es

```
refutacion_anotada(Usable,Soporte,Subsumidas,Ref) :-
   Soporte = [_*_*[]|_],
   findall(C,(member(C,Usable);
              member(C,Soporte);
              member(C,Subsumidas)),
           S),
   prueba(S,Ref), !.
refutacion_anotada(Usable,[C|Soporte],Subsumidas,Ref) :-
   escribe_actual(C),
   resolventes(C,[C|Usable],S1),
   procesa(Usable,Soporte,S1,S2),
   ( memberchk(_*_*[],S2) ->
     append(S2,Soporte,Soporte1),
     refutacion_anotada([C|Usable],Soporte1,Subsumidas,Ref)
   ; % \+ memberchk(_*_*[],S2) ->
     elimina_subsumidas(S2,[C|Usable],Subsumidas,Usable1,
                        Subsumidas1),
     elimina_subsumidas(S2,Soporte,Subsumidas1,Soporte1,
                        Subsumidas2),
     append(Soporte1,S2,Soporte2),
     ordenada_por_peso(Soporte2,Soporte3),
     refutacion_anotada(Usable1,Soporte3,Subsumidas2,Ref)).
```

y la de `procesa(+Usable,+Soporte1,+Resolventes,-Retenidas)` es

```
procesa(_,_,[],[]).
```

```
procesa(Usable,Soporte,[C|S1],S2) :-
   escribe_resolvente(C),
   ( (subsumida(C,Usable) ;
      subsumida(C,Soporte);
      es_clausula_tautologica(C)) ->
     procesa(Usable,Soporte,S1,S2)
   ; % C es retenida ->
     numera(C,C1),
     ( conflicto_unitario(C1,Usable,Soporte,C2) ->
       S2 = [C2,C1]
     ; % C1 no es unitaria con complementaria
       % en Usable o Soporte ->
       elimacion_unitaria(Usable,Soporte,C1,C2),
       escribe_retenida(C2),
       ( C2 = N*H*[] ->
         format('~N~n -----> CLAUSULA VACIA: ~w ~w []',[N,H]),
         S2 = [C2]
       ; % C2 no es la cláusula vacía ->
         procesa(Usable,[C2|Soporte],S1,S3),
         S2 = [C2|S3]))).
```

Los restantes detalles del programa se deja como ejercicio y puede consultarse en la página del libro en la Red `http://www.cs.us.es/~jalonso/`.

# Bibliografía


[1] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 2 edition, 2001.

[2] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison–Wesley, 1986.

[3] S. Burris. *Logic for Mathematics and Computer Science*. Prentice–Hall, 1998.

[4] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[5] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer–Verlag, 4 edition, 1994.

[6] K. Doets. *From Logic to Logic Programming*. MIT Press, 1994.

[7] K. Doets and H.Ñivelle. Otter: A brief introduction. September 1997.

[8] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, 2nd ed. edition, 1995.

[9] J. Gallier. *Logic for Computer Science (Foundations of Automatic Theorem Proving)*. Harper & Row, 1986.

[10] E. S. L. Sterling. *L'art de Prolog*. Masson, 1990.

[11] T. V. Le. *Techniques of Prolog Programming (with implementation of logical negation and quantified goals)*. John Wiley, 1993.

[12] D. Maier and D. Warren. *Computing with Logic (Logic Programming with Prolog)*. Benjamin Cummings, 1988.

[13] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming (Vol. 1: Deductive Reasoning)*. Addison–Wesley, 1985.

[14] A.Ñerode and R. A. Shore. *Logic for Applications*. Springer–Verlag, second edition, 1997.

[15] S. Russell and P.Ñorvig. *Inteligencia artificial (un enfoque moderno)*. Prentice Hall Hispanoamericana, 1996.

[16] U. Schöning. *Logic for Computer Scientists*. Birkäuser, 1989.