Introducción a la programación lógica con Prolog

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 17 de Junio de 2006 (versión de 25 de septiembre de 2006)

Índice

1. Introducción

En este trabajo presentamos el lenguaje de programación lógica Prolog con un doble objetivo: como una aplicación de la deducción automática estudiada en la asignatura de "Lógica informática" y como soporte de la asignatura de "Programación declarativa".

Los textos fundamentales de Prolog son

1. I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison–Wesley, 3 edition, 2001.
2. W. F. Clocksin y C. S. Mellish. *Programming in Prolog*. Springer–Verlag, 4 edition, 1994.
3. L. Sterling y E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.
4. T. Van Le. *Techniques of Prolog Programming (with implementation of logical negation and quantified goals)*. John Wiley, 1993.

Algunas introducciones a Prolog publicadas en la Red son

1. P. Blackburn, J. Bos y K. Striegnitz. *Learn Prolog Now!*, 2001.
2. T. Cornell. *Introduction to Prolog*, 1998.
3. U. Endriss. *An Introduction to Prolog Programming*. Universiteit van Amsterdam, 2005.
4. J. E. Labra. *Programación práctica en Prolog*. Cuaderno Didáctico Num. 13. Ed. Servitec, 1998.
5. F. Llorens y M. J. Castel. *Prácticas de Lógica: Prolog*. Universidad de Alicante, 2001.
6. H. Pain y R. Dale. *An Introduction to Prolog*, 2002.
7. G. Ritchie. *Prolog Step–by–Step*. University of Edinburgh, 2002.
8. D. Robertson. *Prolog*. University od Edinburgh, 2003.
9. T. Smith. *Artificial Intelligence Programming in Prolog*. University of Edinburgh, 2004.

2. 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. 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 [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 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 vamos a considerar las siguientes cuestiones:

1. cómo se representan en Prolog las reglas,
2. cómo se representan en Prolog los hechos,
3. cómo se representan en Prolog las bases de conocimientos,
4. cómo se inicia una sesión Prolog,
5. cómo se carga en Prolog la base de conocimiento,
6. cómo se representa en Prolog el objetivo a demostrar,
7. cómo se interpreta la respuesta de Prolog,
8. cómo ha realizado Prolog la búsqueda de la demostración,
9. cuál es la demostración obtenida y
10. cómo 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` $\wedge$ `tiene_rayas_negras` $\rightarrow$ `es_cebra`

donde $\wedge$ representa la conjunción y $\rightarrow$, 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.

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.18)
Copyright (c) 1990-2006 University of Amsterdam.

?- [animales].
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 en la Figura 1.

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.

[1] En SWI Prolog, para que no dé error en los predicados no definidos, hay que añadirle al fichero con la base de conocimiento la línea

```
:- set_prolog_flag(unknown,fail).
```

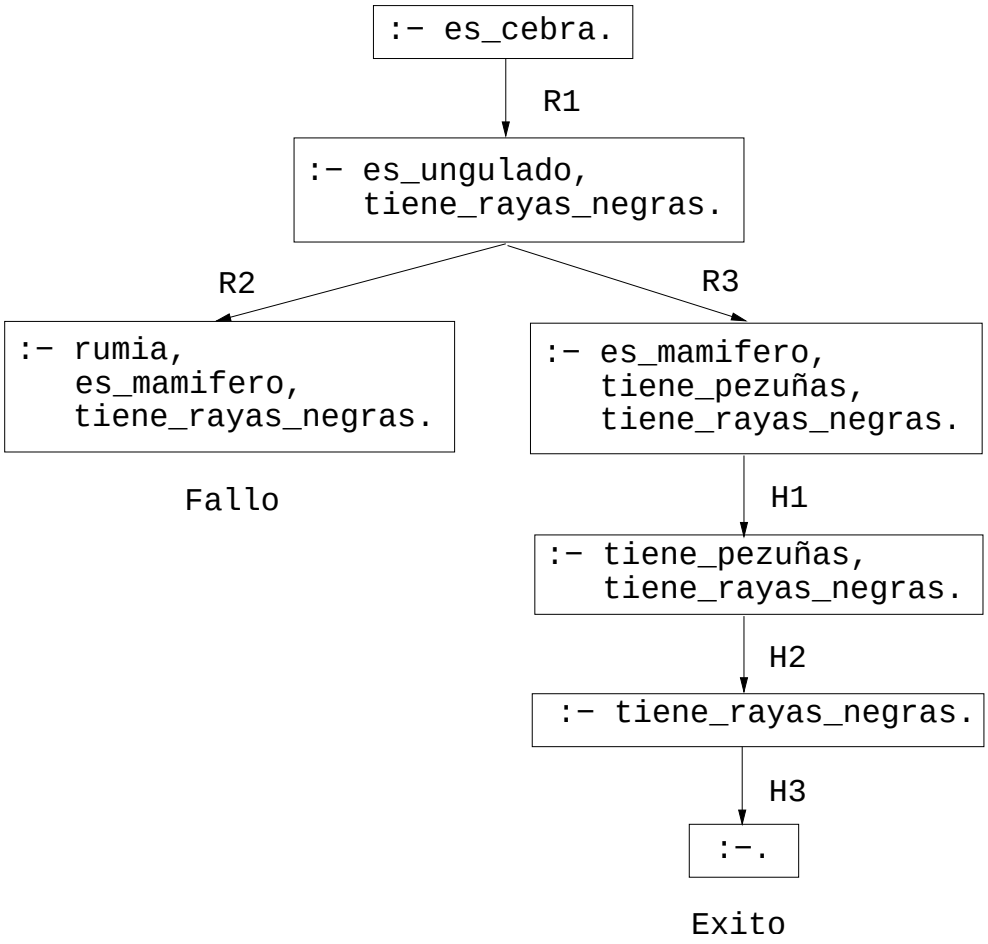

Figura 1: Árbol de deducción del problema de los animales

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 demostración (por resolución SLD) que se muestra en la Figura 2 (página 8).

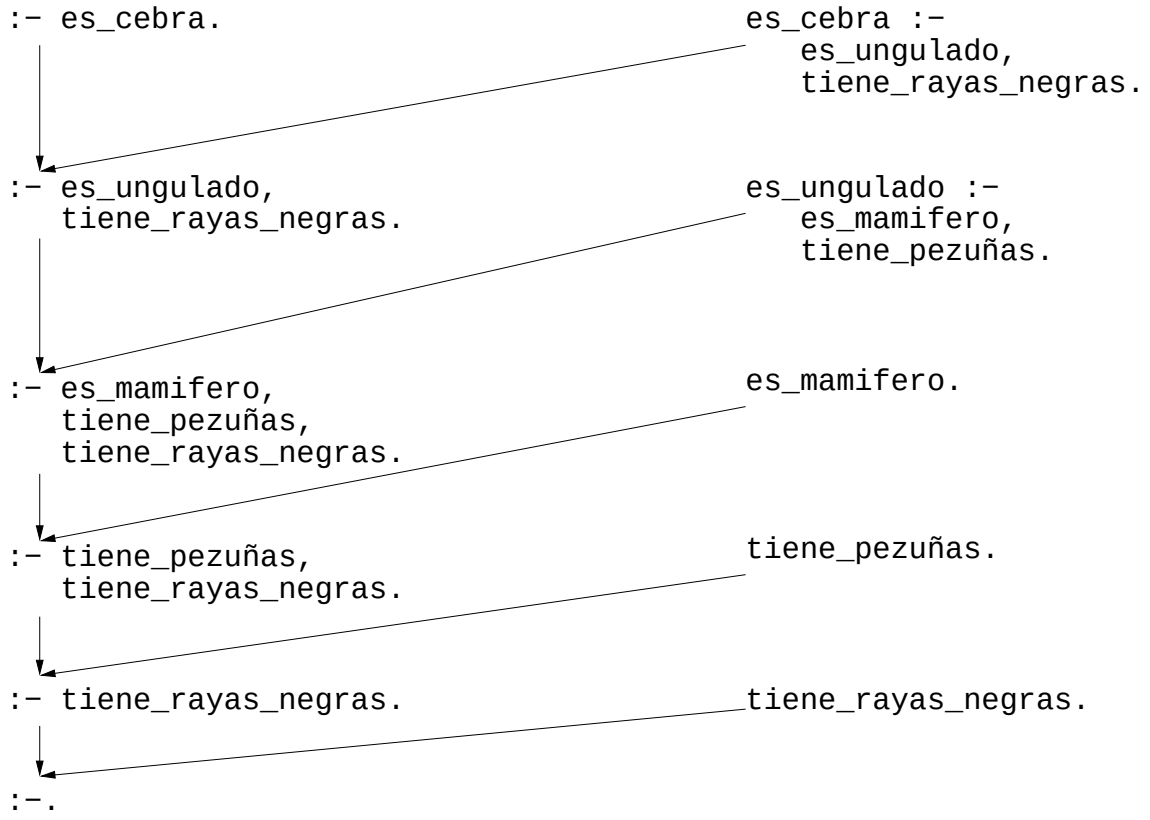

Figura 2: Demostración SLD del problema de los animales

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

1	es_ungulado $\wedge$ tiene_rayas_negras $\rightarrow$ es_cebra	Regla 1
2	rumia $\wedge$ es_mamífero $\rightarrow$ es_ungulado	Regla 2
3	es_mamífero $\wedge$ tiene_pezuñas $\rightarrow$ es_ungulado	Regla 3
4	es_mamífero	Hecho 1
5	tiene_pezuñas	Hecho 2
6	tiene_rayas_negras	Hecho 3
7	es_mamífero $\wedge$ tiene_pezuñas	$I_\wedge$ 4 y 6
8	es_ungulado	$E_\rightarrow$ 3 y 7
9	es_ungulado $\wedge$ tiene_rayas_negras	$I_\wedge$ 8 y 6
5	es_cebra	$E_\rightarrow$ 1 y 9

2.2. 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.
  - 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(X,Y)` que se verifica si `X` divide a `Y`. 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 se muestra en la Figura 3 (página 10). 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.

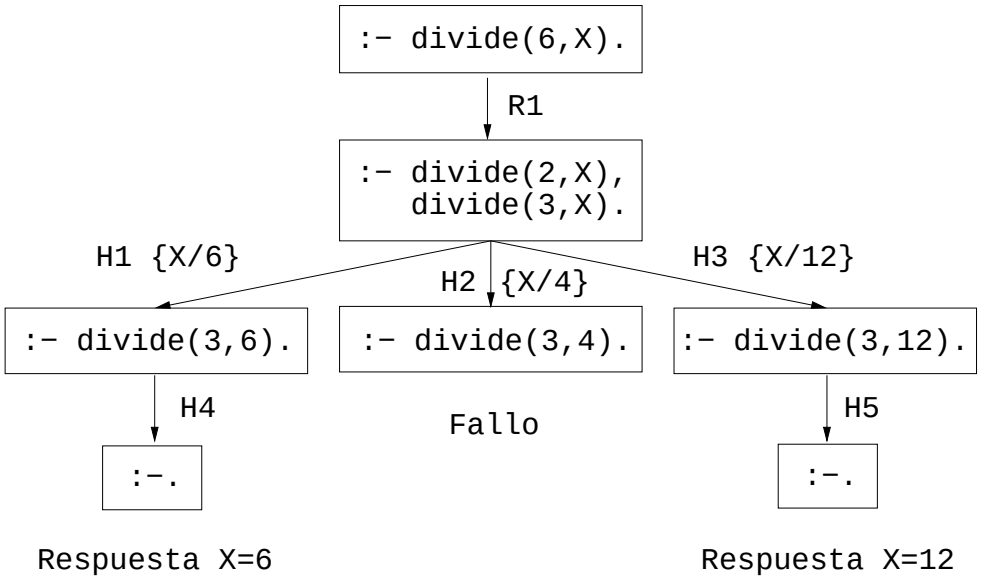

Figura 3: Árbol SLD del problema de divisibilidad

2.3. 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 verifica 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 se muestra en la Figura 4 (11). Del árbol vamos a comentar la separa-

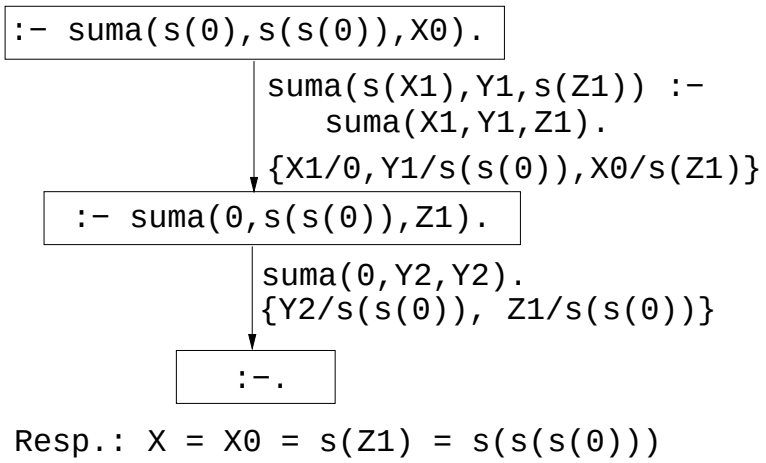

Figura 4: Árbol SLD del problema de la suma

ción de variables, las unificaciones y el cálculo de la respuesta. Para evitar conflicto con las variables, se cambia de nombre añadiendo el índice 0 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(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
```

El árbol de deducción correspondiente se muestra en la Figura 5 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)) ;
```

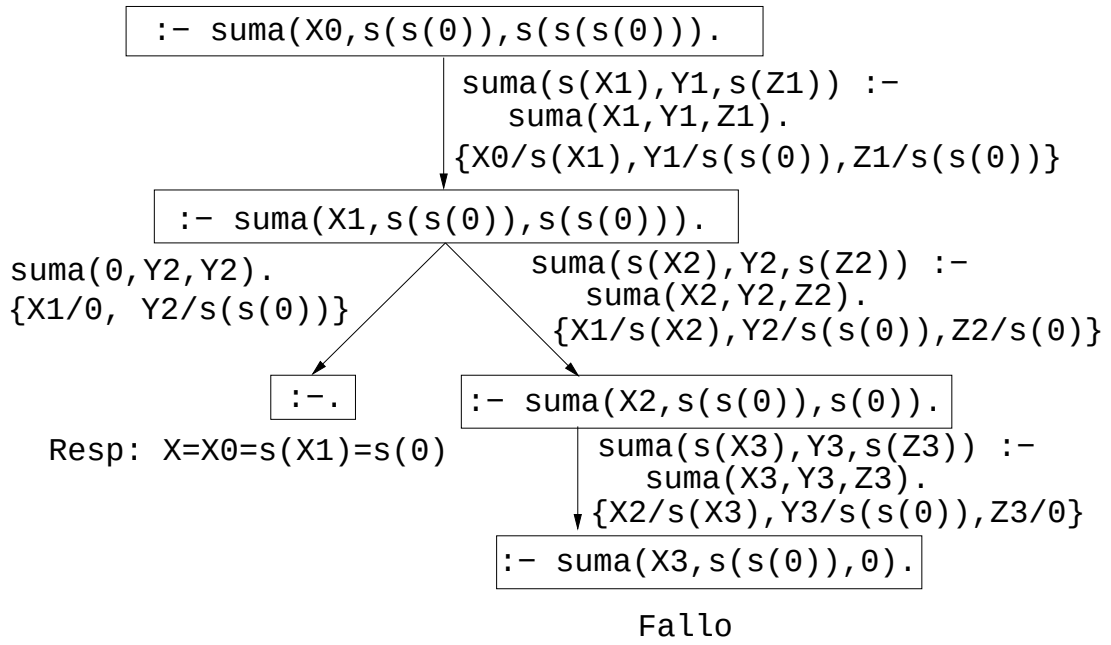

Figura 5: Árbol SLD del problema de la resta

```
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 correspondiente se muestra en la Figura 6. 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 σ_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 σ_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 σ_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)).
```

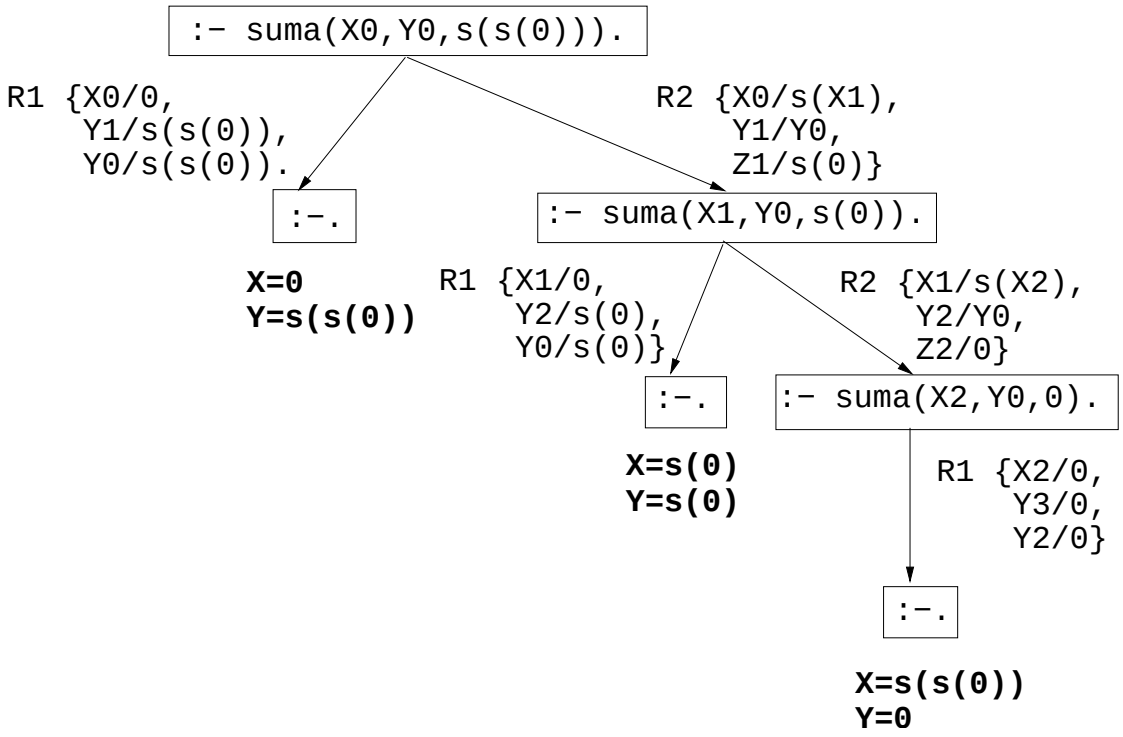

Figura 6: Árbol SLD del problema de la descomposición en suma dos elementos

```
X = 0
Y = s(0) ;
No
```

El árbol de deducción correspondiente se muestra en la Figura 7. 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)`.

3. Listas

En esta sección vamos a estudiar cómo se representan las listas en Prolog y a definir algunas relaciones sobre listas que se usan frecuentemente.

3.1. 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 si `L` es una lista el término `.(b,L)` representa la lista obtenida añadiendo el elemento `b` 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.

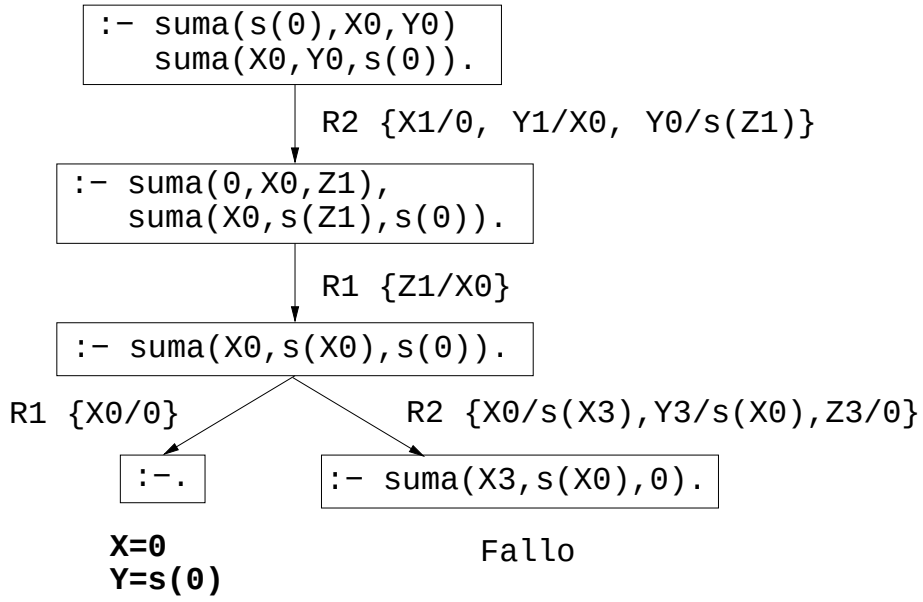

Figura 7: Árbol SLD correspondiente al sistema de ecuaciones

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]
```

3.2. Concatenación de listas

Vamos a definir una 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]`?.

   ```
   ?- conc([a,b],[c,d,e],L).
   L = [a, b, c, d, e]
   ```

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

   ```
   ?- conc([a,b],L,[a,b,c,d]).
   L = [c, d]
   ```

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

   ```
   ?- 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 se muestra en la Figura 8 (página 16) en el que vuelve a resaltar la analogía con el correspondiente a la tercera cuestión de la `suma`.

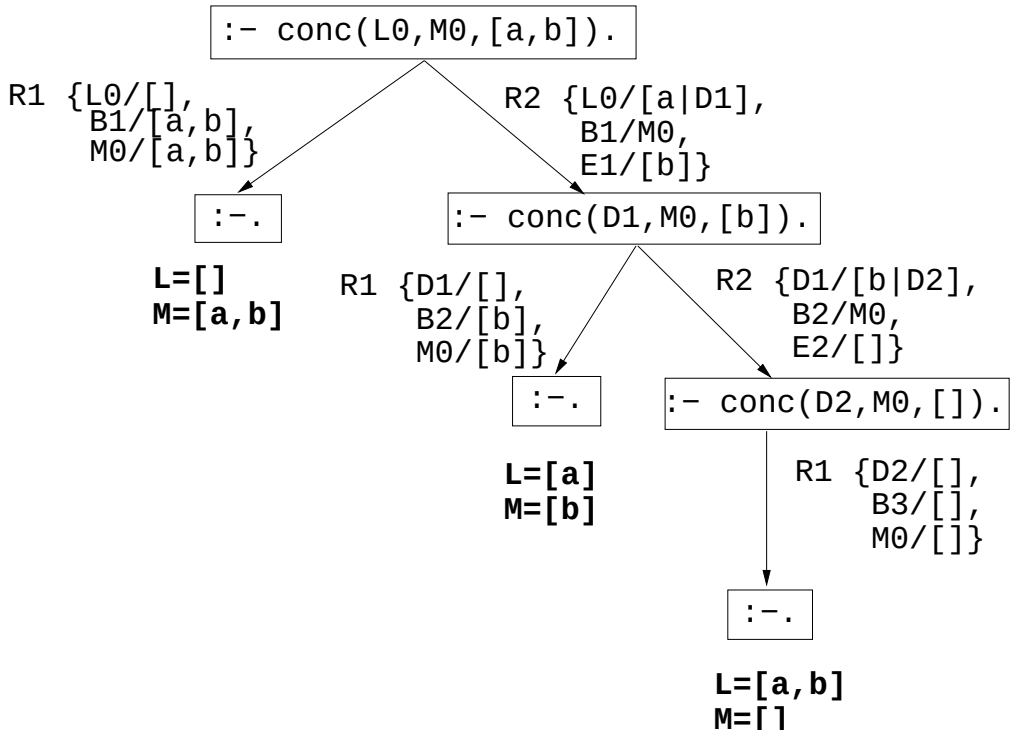

Figura 8: Árbol SLD de la concatenación

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

3.3. La relación de pertenencia

Vamos a definir una 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`, ...). El árbol de deducción se muestra en la Figura 9

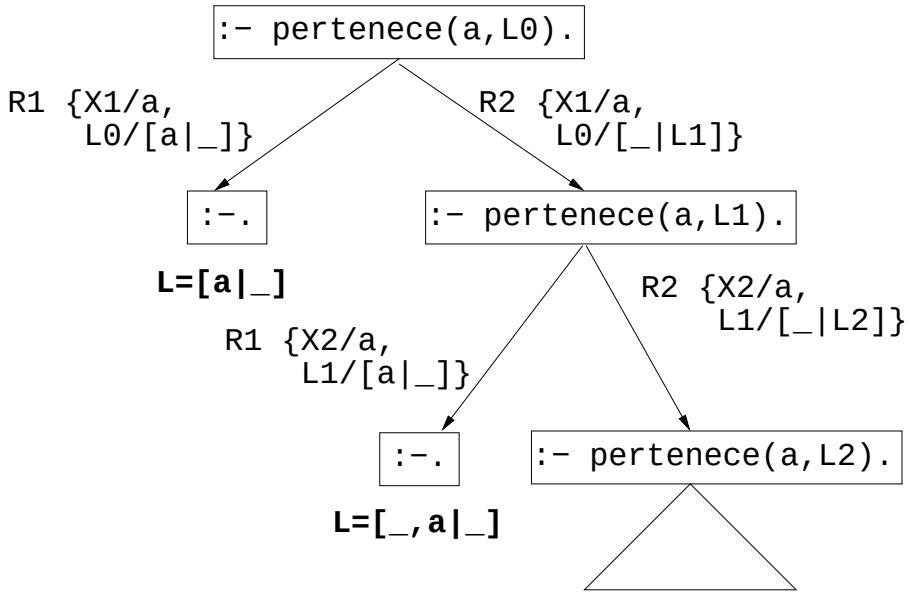

Figura 9: Árbol SLD del problema de pertenencia

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

4. 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).
```

5. Operadores

Prolog permite la declaración de operadores indicando su nombre, tipo y precedencia. Además, dispone de un conjunto de operadores previamente declarados. Uno de ellos es el operador + que está declarado de tipo `yfx` (que significa que es infijo (porque la `f` está en el centro) y asocia por la izquierda (porque la `y` está a la izquierda)) y con 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 infijos no asociativos y los 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
```

6. Aritmética

Hemos visto en la sección 5 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 10

7. 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.
```

Si calculamos la calificación correspondiente a un `6`,

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

se genera el árbol de deducción que se muestra en la Figura 11. 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

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

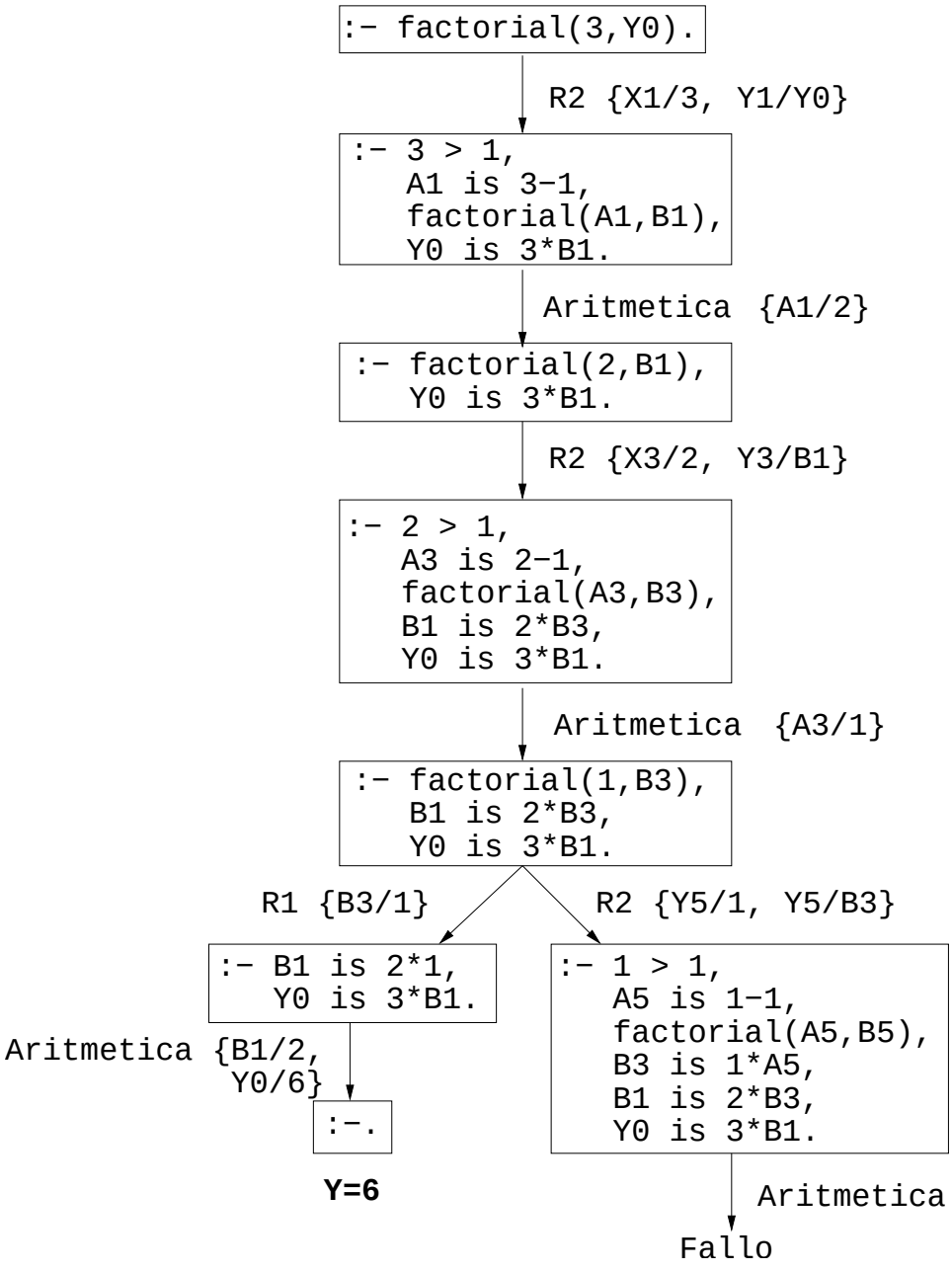

Figura 10: Deducción del factorial

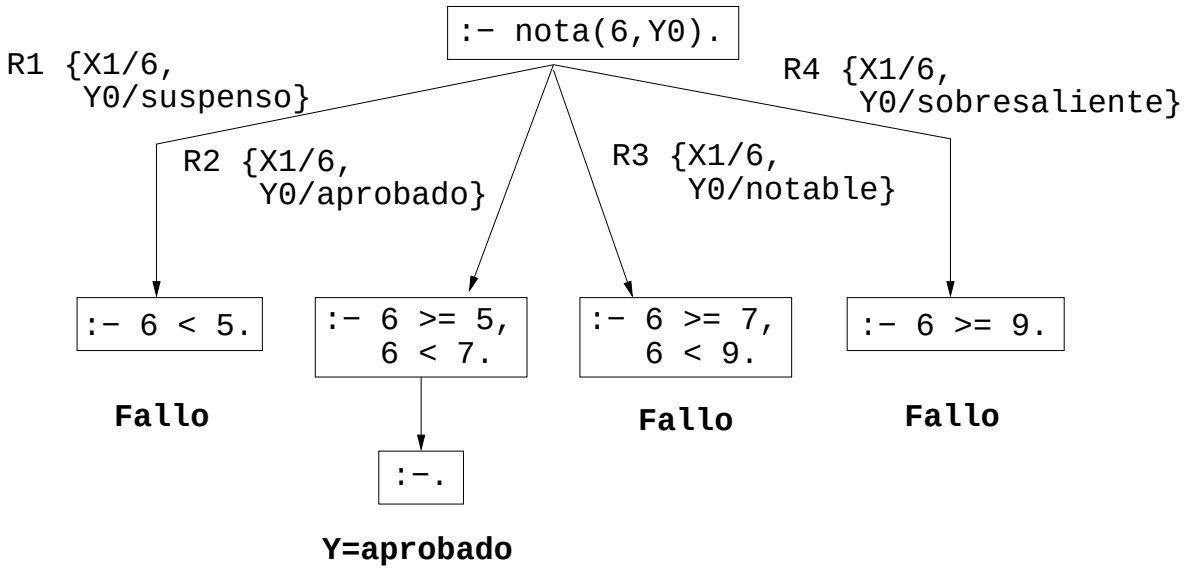

Figura 11: Árbol SLD del problema de las notas

```
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 el que se muestra en la Figura 12. Vemos que el efecto del corte es la eliminación de las alternativas abiertas por debajo

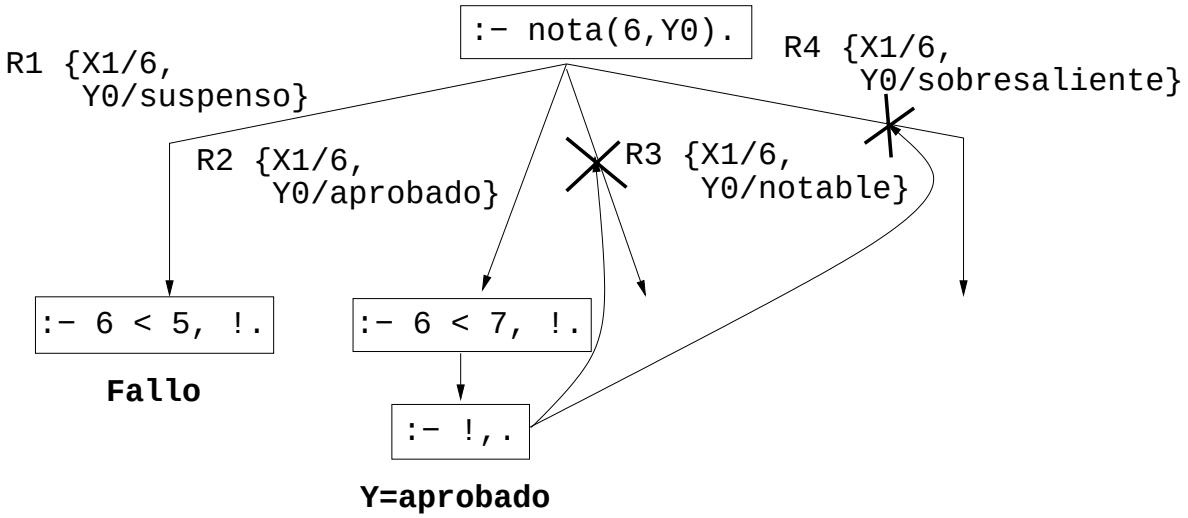

Figura 12: Árbol SLD del problema de las notas con corte

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 finalizado 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).
```

8. 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 2
```

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 se muestra en la Figura 13 (página 24). Para demostrar que `luis` no está suspenso intenta probar que `luis` está suspenso,

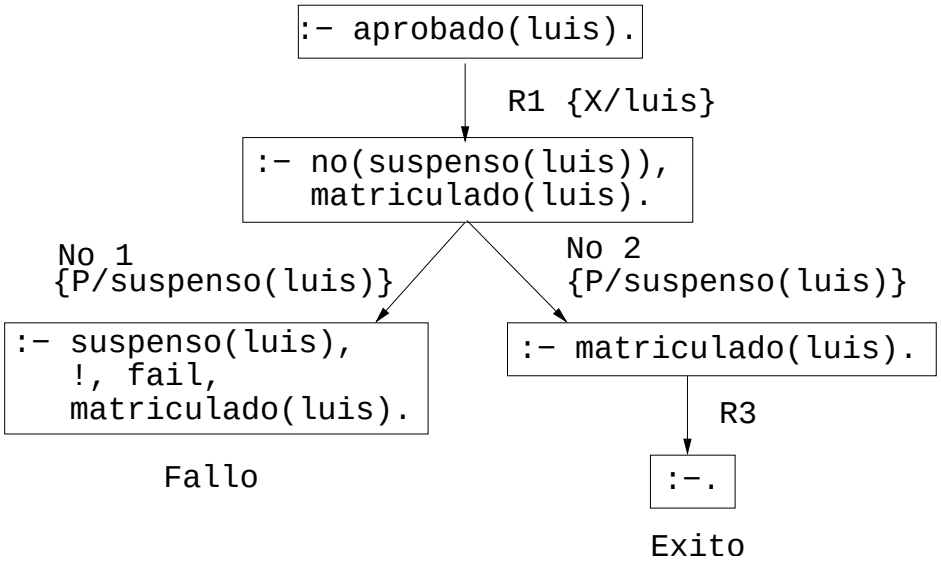

Figura 13: Árbol SLD con negación (I))

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 se muestra en la Figura 14 (página 25). 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 se muestra en la Figura 15.

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
```

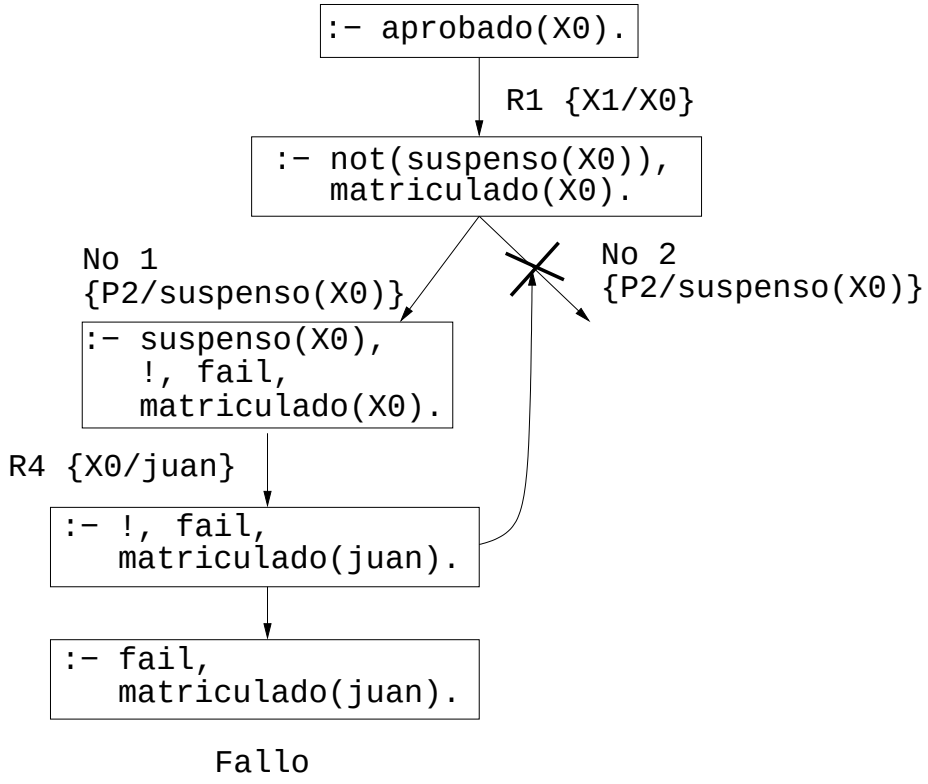

Figura 14: Árbol SLD con negación (II))

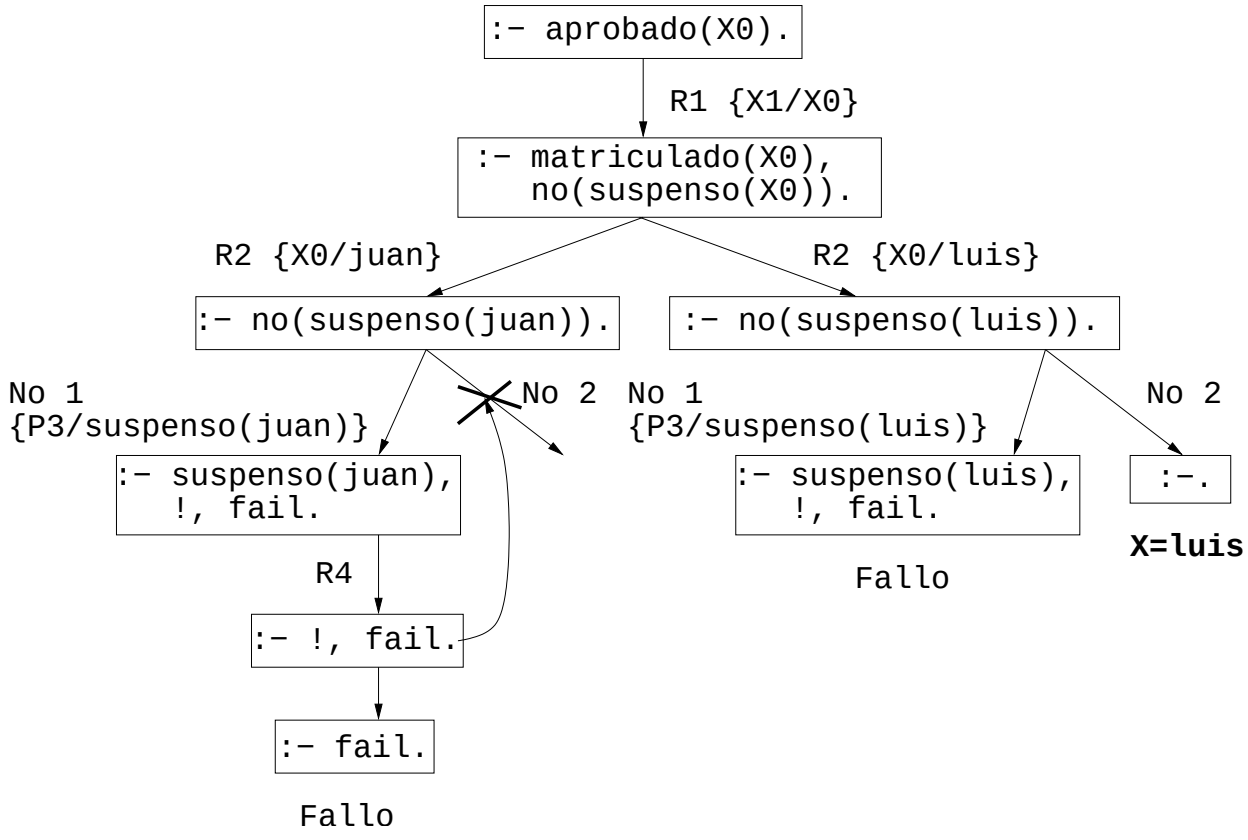

Figura 15: Árbol SLD con negación (III))

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

9. El condicional

En Prolog se dispone del condicional `->`. Usando el condicional, se puede definir la relación `nota` (página 20)

```
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 correspondiente a la pregunta `nota(6,Y)` se muestra en la Figura 16 (página 27). El árbol es análogo al de la definición con corte del la Figura 12 (página 22).

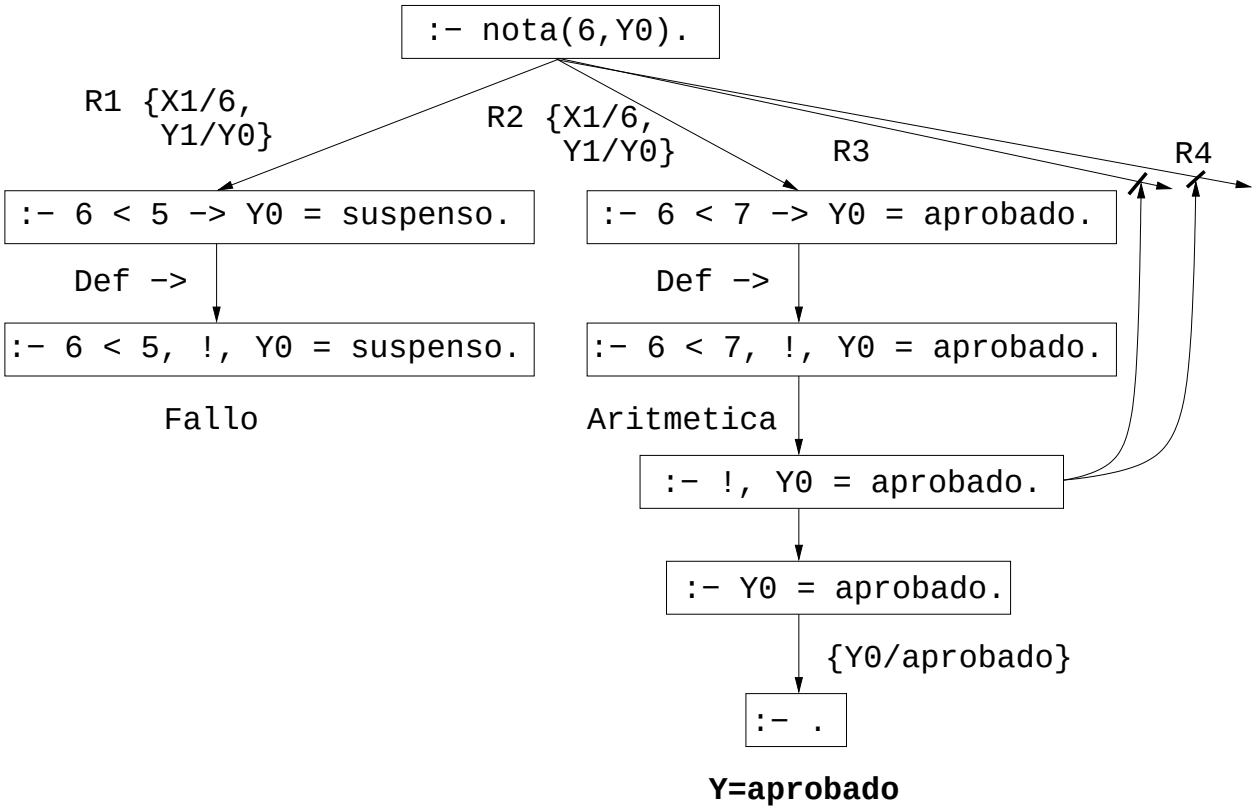

Figura 16: Árbol SLD con condicional

10. Predicados sobre tipos de término

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

11. 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 menor 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`. Por 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
```

```
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).
```

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

12. Procesamiento de términos

La relación `?T =.. ?L` se verifica si `L` es una lista cuyo 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 `F1` 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(triángulo(3,4,5),2,F).
F = triángulo(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].
```

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

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

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)
```

13. 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 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. 30)

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

14. 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` : $(\exists$`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 conjuntistas. Para facilitar las definiciones, definimos 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).
```

Referencias

[1] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison–Wesley, 3 edition, 2001.

[2] W. F. Clocksin y C. S. Mellish. *Programming in Prolog*. Springer–Verlag, 4 edition, 1994.

[3] R. A. O'Keefe. *The Cratf of Prolog*. The MIT Press, 1990.

[4] L. Sterling y E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

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