DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Sistemas Inteligentes de Gestión

Tutorial de PROLOG

© *Juan Carlos Cubero & Fernando Berzal*

Índice

Programación de sistemas expertos en PROLOG

El PROLOG, cuyo nombre proviene del francés "PROgrammation en LOGique", es un lenguaje de programación declarativa muy utilizado en Inteligencia Artificial, principalmente en Europa.

El lenguaje fue a principios de los años 70 en la Universidad de Aix-Marseille (Marsella, Francia) por los profesores Alain Colmerauer y Philippe Roussel, como resultado de un proyecto de procesamiento de lenguajes naturales. Alain Colmerauer y Robert Pasero trabajaban en la parte del procesamiento del lenguaje natural y Jean Trudel y Philippe Roussel en la parte de deducción e inferencia del sistema. Interesado por el método de resolución SL, Trudel persuadió a Robert Kowalski para que se uniera al proyecto, dando lugar a una versión preliminar del lenguaje PROLOG a finales de 1971, cuya versión definitiva apareció en 1972.

La primera versión de PROLOG fue programada en ALGOL W e, inicialmente, se trataba de un lenguaje totalmente interpretado. En 1983, David H.D. Warren desarrolló un compilador capaz de traducir PROLOG en un conjunto de instrucciones de una máquina abstracta, denominada Warren Abstract Machine. Se popularizó en la década de los 80 por la aparición de herramientas para microordenadores (p.ej. Turbo PROLOG, de Borland) y su adopción para el desarrollo del proyecto de la quinta generación de computadoras, para el que se desarrolló la implementación paralelizada del lenguaje llamada KL1. Las primeras versiones del lenguaje diferían, en sus diferentes implementaciones, en muchos aspectos de sus sintaxis, empleándose mayormente como forma normalizada el dialecto propuesto por la Universidad de Edimburgo , hasta que en 1995 se estableció un estándar ISO (ISO/IEC 13211-1), llamado ISO-Prolog.

http://es.wikipedia.org/wiki/Prolog

Bibliografía

- W.F. Clocksin & C.S. Mellish: **Programming in Prolog**. Springer-Verlag, 5th edition, 2003. ISBN 3540006788
- Ivan Bratko: **Prolog Programming for Artificial Intelligence.** Addison-Wesley, 4th edition, 2011. ISBN 0321417461
- T. Van Le: **Techniques of Prolog Programming**. Wiley, 1992. ISBN 047157175X
- Dennis Merrit: **Building Expert Systems in Prolog.** Springer, 1989. ISBN 0387970169
- Yoav Shoham: **Artificial Intelligence Techniques in Prolog**. Morgan Kaufmann, 1994. ISBN 1558601678
- Neil C. Rowe: **Artificial Intelligence Through Prolog**. Prentice-Hall, 1988. ISBN 0130486795

Cursos de PROLOG en la Web:

http://cs.union.edu/~striegnk/courses/esslli04prolog/
http://www.cs.bham.ac.uk/~pjh/prolog_course/se207.html
http://homepages.inf.ed.ac.uk/pbrna/prologbook/
PROLOG y la orientación a objetos:
http://www.cetus-links.org/oo_prolog.html

Compiladores e intérpretes de PROLOG

SWI Prolog [el utilizado en este curso]
http://www.swi-prolog.org/
Visual Prolog
http://www.visual-prolog.com/
AMZI-Prolog
http://www.amzi.com/
BIN-Prolog:
http://www.binnetcorp.com/BinProlog/
Gnu-Prolog:
http://www.gprolog.org/
P#
http://www.dcs.ed.ac.uk/home/jjc/psharp/psharp-1.1.3/dlpsharp.html
C#Prolog
http://sourceforge.net/projects/cs-prolog/
Kiss Prolog
http://sourceforge.net/projects/kissprolog/
Open Prolog (Apple Macintosh)
http://www.scss.tcd.ie/misc/open-prolog/
Kernel Prolog
http://www.binnetcorp.com/kprolog/Main.html
Ciao Prolog System
http://www.ciaohome.org/
LPA Prolog
http://www.lpa.co.uk/pro_log.htm
Strawberry Prolog
http://www.dobrev.com/index.html
jProlog
http://people.cs.kuleuven.be/~bart.demoen/PrologInJava/
Jinni
http://www.binnetcorp.com/Jinni/
YAProlog
http://www.dcc.fc.up.pt/~vsc/Yap/
tuProlog
http://www.alice.unibo.it/xwiki/bin/view/Tuprolog/
XSB
http://xsb.sourceforge.net/

El lenguaje PROLOG

Como shell para la programación de Sistemas Expertos Basados en Reglas, PROLOG usa Lógica de Predicados de Primer Orden (restringida a cláusulas de Horn) para representar datos y conocimiento, utiliza encadenamiento hacia atrás y una estrategia de control retroactiva sin información heurística (backtracking).

Elementos del lenguaje

- Hechos (átomos).
- Reglas (cláusulas de Horn).
- Preguntas u objetivos (conjunciones ó disyunciones de átomos).

Símbolos del Lenguaje

Caracteres

- Alfanuméricos: `A..Z a..z 0..9`
- Especiales: `+ - * / <> = :- &`

Tokens del lenguaje

- Los nombres de **predicados** y las **constantes** del dominio empiezan con minúscula y se construyen como cadenas de letras, dígitos y _

  p.ej. `pedro x25 x_1 funcion predicado`

  CUIDADO: Algunos ya están predefinidos: `atomic, var, repeat…`

  Para PROLOG, el predicado `padre/2` es distinto del predicado `padre/1`; es decir, `padre(juan)` y `padre(juan,maria)` son legales y hacen referencia a distintos predicados, aunque es mejor evitar este tipo de situaciones.

- **Literales** numéricos, como cualquier lenguaje de programación, y cadenas de caracteres entre comillas simples (p.ej. `'Pedro'`, `'Nueva York'`)
- **Variables**: Cadenas de letras, dígitos y _ empezando con mayúscula

  p.ej. `X Lista Persona`

- **Comentarios** como en C: `/* Comentario */`

NOTA: En muchos libros, a los tokens del lenguaje los llaman átomos, pero evitaremos esta terminología para no confundirlo con el término átomo de Lógica de Predicados.

Hechos

- Átomos en Lógica de Predicados.
- No se permiten disyunciones.
- Los nombres de los predicados empiezan con minúscula.
- El hecho debe terminar con un punto.

Lógica de Predicados	PROLOG
esHombre(Juan)	`eshombre(juan).`
gusta(Pedro,Calabaza)	`gusta(pedro,calabaza).`
esHombre(Pedro) ∧ esHombre(Juan)	`esHombre(pedro).` `esHombre(juan).`
esHombre(Pedro) ∨ esPerro(boby)	`/* No puede representarse */`
∃x quiere(Juan,x) ∧ ∃z quiere(Juan,z)	`quiere(juan,alguien1).` `quiere(juan,alguien2).`

Preguntas u objetivos

En tiempo de ejecución, aparece el prompt `?-` y el intérprete de PROLOG espera que el usuario introduzca un objetivo en forma de predicado, con o sin variables.

Al igual que en CLIPS, tendremos ficheros con la descripción de la base de conocimiento (hechos y reglas)

`baseconoc.pl (o .pro)`

```
esHombre(juan).
esHombre(pedro).
```

Para cargar un fichero, escribiremos:

```
?- [baseconoc].
```

/* Esto añade a la MT de Prolog los hechos del fichero `baseconoc.pl` */

```
?- esHombre(juan).
yes
?- esHombre(pedro).
yes
?- esHombre(juanCarlos).
no
?- esMamifero(leon).
no
```

En realidad, para las dos últimas preguntas, debería responder, respectivamente: "No tengo información suficiente" y "No está declarado el predicado `esMamifero`" (de hecho, algunos compiladores lo hacen).

Justificación: Hipótesis de Mundo Cerrado.

En una sesión de trabajo, PROLOG asume que todo lo que no se declare explícitamente (hechos o conclusiones que puedan demostrarse usando las reglas) es falso.

Formalmente, PROLOG va buscando en la memoria de trabajo, hechos que se unifiquen con el objetivo.

En general, una pregunta [*query*] puede ser más compleja que una simple consulta a la base de datos. Por ejemplo, puede ser una conjunción de átomos:

```
?- esHombre(juan), esHombre(pedro).
yes
```

En PROLOG, la coma representa el conectivo ∧. Puede usarse dentro de una pregunta y (como se verá después) dentro de una regla, pero no en la declaración de un hecho.

PROLOG intentará demostrar los objetivos en el orden especificado. Si no puede demostrar alguno de ellos, devolverá un "no". Si puede demostrarlos todos, devolverá un "yes".

Cada objetivo de una lista de objetivos tiene asociado un puntero que se encarga de recorrer la memoria de trabajo buscando átomos que se unifiquen con dicho objetivo.

```
?- esHombre(juan) , quiere(juan,elena).
      ⇒2                ⇒2

      ⇒1   esHombre(pedro).              /* no */
           esHombre(juan).
           quiere(juan,maria).
           quiere(juan,elena).
           quiere(pedro,raquel).
           quiere(pedro,belen).

      ⇒1   esHombre(pedro).              /* yes */
           esHombre(juan).
      ⇒2   quiere(juan,maria).           /* no  */
           quiere(juan,elena).
           quiere(pedro,raquel).
           quiere(pedro,belen).

           esHombre(pedro).
      ⇒1   esHombre(juan).               /* yes */
           quiere(juan,maria).
      ⇒2   quiere(juan,elena).           /* yes */
           quiere(pedro,raquel).
           quiere(pedro,belen).
```

Tipos de preguntas:

- **Sin variables**: La respuesta es, simplemente, `yes` o `no`.

```
?- esHombre(juan), esHombre(pedro).
yes
```

- **Con variables**, cuando estamos interesados en obtener todos los objetos que cumplen un objetivo.

```
?-esHombre(Persona).
Persona = juan ;
Persona = pedro ;
no
```

El punto y coma (`;`), introducido por el usuario, le indica al intérprete de PROLOG que siga buscando respuestas. El punto (`.`) se emplea cuando ya no nos interesa obtener más respuestas.

PROLOG va buscando en la MT hechos que se unifiquen con cada objetivo OBJ_i, empezando con el primero (`i=1`). Si no encuentra ninguno, devuelve directamente "`no`". En caso contrario, busca un unificador de máxima generalidad *u*, aplica dicha sustitución al resto de objetivos, y pasa a demostrar el siguiente (`i+1`).

PROLOG ofrecerá en pantalla las sustituciones de aquellas variables especificadas en los objetivos, que hicieron posible las unificaciones.

- **Con la variable anónima** (`_`), cuando estamos interesados en saber si existe algún objeto que cumpla un objetivo.

En el objetivo, la variable anónima hay que verla como una variable libre, sin ninguna cuantificación, que forzará una única sustitución: la primera que se encuentre.

La respuesta de PROLOG es, simplemente, `yes` o `no`

```
?- esHombre(_).
yes
```

EJEMPLOS

```
?- gusta(X,maria).      /* ¿A quién le gusta María */
?- gusta(_,maria).      /* ¿Hay alguien a quien le gusta María */
?- gusta(maria,_).      /* ¿Hay alguien que le guste a María */
?- gusta(juan,X) , gusta(maria,Y). /* ¿? */
?- gusta(juan,X) , gusta(maria,X). /* Gustos comunes de Juan&María */
```

Especificación de hechos

- **Átomos sin cuantificación**:

```
gusta(juan,maria).
```

- **Átomos con cuantificación existencial** (uso de variables de Skolem):

```
gusta(alguien_maria,maria).   /* ∃x (gusta(x,maria)) */
gusta(alguien_paula,paula).   /* ∃x (gusta(x,paula)) */
```

- **Átomos con cuantificación universal**,
  dependiendo del compilador de PROLOG:

```
gusta(X,maria).
gusta(_,maria).               /* ∀x (gusta(x,maria)) */
```

  Algunos compiladores no admiten que una variable aparezca una sola vez, de ahí el uso de la variable anónima _

  Cuidado con la interpretación de la variable anónima:

  HECHO: `gusta(juan,_).` `/* ∀x (gusta(juan,X)) */`

  OBJETIVO: `?- gusta(juan,_).` `/* gusta(juan,X) */`

NOTA: La disyunción de átomos no puede representarse en PROLOG:
Ni ∃x (P(x) ∨ Q(x)) ni ∀x (P(x) ∨ Q(x)) pueden representarse

EJEMPLOS

"Todos nos gustamos"
`gusta(X,X).`
Como la variable X aparece dos veces, este hecho es legal en cualquier compilador.

"Todos se gustan entre sí"
∀x ∀y gusta(x,y)
Para compiladores que admitan variables libres: `gusta(X,Y).`
Para los que no: `gusta(_,_).`

Aunque la variable anónima _ aparezca dos veces,
PROLOG sustituye internamente cada aparición por una variable distinta.

```
gusta(_,_).  ≡  gusta(_1234,_1235).
```

EJERCICIOS RESUELTOS

Especifique los siguientes hechos en PROLOG:

- *"Todos quieren a Juan y a María".*
  $\forall x$ (quiere(x,juan) $\wedge$ quiere(x,maria))

```
quiere(_ , juan).
quiere(_ , maria).
```

- *"Alguien quiere a Juan y a María".*
  $\exists x$ (quiere(x,juan) $\wedge$ quiere(x,maria))

```
quiere(alguien , juan).
quiere(alguien , maria).
```

Supongamos únicamente el siguiente hecho: `gusta(juan,_).`
¿Cómo se especificarían en PROLOG las siguientes consultas?

- ¿Le gusta algo a Juan?

```
?- gusta(juan,_).
yes
```

- ¿Le gustan a Juan las almendras?

```
?- gusta(juan,almendras).
yes
```

- ¿Qué es lo que le gusta a Juan?

```
?- gusta(juan,X).
X=_3456;                /* Lo interpretamos como "todo" */
no
```

Suponiendo los siguientes hechos:

```
esPadre(juan,pedro).
esPadre('Dios',_).
```

```
?- esPadre(X,pedro).
X=juan;
X='Dios';
no
?- esPadre(_,pedro).
yes
?- esPadre(X,_).
X=juan;
X='Dios';
no
?- esPadre(_,_).
yes
?- esPadre(_,X).
X=pedro;
X=_2567;
no
?- esPadre(X,Y).
X=juan   ,  Y=pedro;
X='Dios' ,  Y=_2567;
no
?- esPadre(personaInexistente,X).
no
```

Estrategia de control de PROLOG

```
esHombre(juan).
esHombre(pedro).
quiere(juan,maria).
quiere(juan,elena).
quiere(pedro,raquel).
quiere(pedro,belen).

?- esHombre(X) , quiere(X,Y).
```

Ejecución paso a paso de la resolución de la consulta:

```
        esHombre(X)
        esHombre(juan).
                              quiere(juan,Y)
                              quiere(juan,maria).

    X=juan , Y=maria ;

                              quiere(juan,Y)
                              quiere(juan,elena).

    X=juan , Y=elena ;
                              quiere(juan,Y)
                              /* FALLO */
        esHombre(X)
        esHombre(pedro)
                              quiere(pedro,Y)
                              quiere(pedro,raquel).

    X=pedro , Y=raquel ;

                              quiere(pedro,Y)
                              quiere(pedro,belen).

    X=pedro , Y=belen ;
                              quiere(pedro,Y)
                              /* FALLO */
        esHombre(X)
        /* FALLO */
    no
```

- Cada vez que se encuentra una solución, se intenta resatisfacer el último predicado (p.ej. `quiere(juan,Y)`). Para ello, PROLOG primero deshace la última sustitución (`Y=maria`) y busca otra sustitución válida.

- Cuando un objetivo no pueda demostrarse de otra forma, diremos que da FALLO. Entonces, se produce un retroceso y se intenta resatisfacer el predicado anterior (backtracking)

En general, si tenemos una lista de objetivos OBJ1, OBJ2, ... , OBJn y falla OBJ(i), entonces se intenta resatisfacer OBJ(i-1) a partir de la antigua marca de OBJ(i-1) y, a continuación, se procede a satisfacer OBJ(i) desde el principio.

```
esHombre(juan).
esHombre(pedro).
quiere(juan,maria).
quiere(juan,elena).
quiere(pedro,raquel).
quiere(pedro,belen).

?- esHombre(X), quiere(X,Y).


            esHombre(X)
            esHombre(juan).
            /* X=juan */
                                quiere(juan,Y)
                                quiere(juan,maria).
                                /* Y=maria */
      X=juan , Y=maria ;
      /* Deshace Y=maria y resatisface quiere(juan,Y) */

                                quiere(juan,Y)
                                quiere(juan,elena).
                                /* Y=elena */
      X=juan , Y=elena ;
      /* Deshace Y=elena y resatisface quiere(juan,Y) */

                                quiere(juan,Y)
                                /* FALLO => Vuelta atrás */
            esHombre(X)
            esHombre(pedro)
            /* X=pedro */
                                quiere(pedro,Y)
                                quiere(pedro,raquel).
                                /* Y=raquel */
      X=pedro , Y=raquel ;
      /* Deshace Y=raquel y resatisface quiere(pedro,Y) */

                                quiere(pedro,Y)
                                quiere(pedro,belen).
                                /* Y=belen */
      X=pedro , Y=belen ;
      /* Deshace Y=belen y resatisface quiere(pedro,Y) */

                                quiere(pedro,Y)
                                /* FALLO => Vuelta atrás */
            esHombre(X)
            /* FALLO => FIN */
      no
```

IMPORTANTE:
Sólo se resatisface un objetivo cuando tiene variables no ligadas.
Si todo son constantes, entonces no se resatisface.

```
esHombre(juan).
esHombre(pedro).
quiere(juan,maria).
quiere(juan,elena).
quiere(pedro,raquel).
quiere(pedro,belen).

?- esHombre(X), quiere(X,elena).
```

```
        esHombre(X)
        esHombre(juan).
        /* X=juan */
                            quiere(juan,elena)
                            quiere(juan,maria).
                            /* Éxito */
    X=juan ;
    /* Deshace X=juan y resatisface esHombre(X) */

        esHombre(X)
        esHombre(pedro)
        /* X=pedro */
                            quiere(pedro,elena)
                            /* FALLO => Vuelta atrás */

        /* Deshace X=pedro y resatisface esHombre(X) */

        esHombre(X)

        /* FALLO => FIN */
    no
```

Proceso de búsqueda con "vuelta atrás" [backtracking]:

- Cada vez que se encuentre una solución, se intenta resatisfacer el último predicado: corresponde a un retroceso en el proceso de búsqueda. Para ello, PROLOG debe deshacer la última sustitución.

- Cuando un objetivo no pueda demostrarse de otra forma, diremos que da FALLO. Entonces se produce otro retroceso en el proceso de búsqueda y se intenta resatisfacer el predicado anterior.

Operadores predefinidos

```
?- consult(<nombre de fichero>).
```

Añade todas las construcciones (hechos y reglas) del fichero a las que ya existen en memoria (correspondientes al fichero actual). Se añaden al final, tal cual.

```
?- reconsult(<nombre de fichero>).
```

Añade todas las construcciones (hechos y reglas) del fichero a las que ya existen en memoria. Se añaden al final, pero se borran todas aquellas reglas antiguas cuya cabecera coincide con alguna de las nuevas. Ídem con los hechos.

NOTA: Éste es realmente la acción realizada con *File* > *Consult* en SWI-Prolog.

```
?- [<fichero_1> , ... , <fichero_n>].
```

Equivale a realizar reconsult sobre los ficheros indicados.

Otros predicados predefinidos

`true`	Es un predicado que siempre devuelve un éxito
`fail`	Es un predicado que siempre devuelve un fallo
`var(X)`	Devuelve éxito si X no está instanciada.
`nonvar(X)`	Al revés.
`atom(X)`	Devuelve éxito si X es un "átomo" en el sentido de Prolog (constantes y cadenas de caracteres son “átomos”; variables, funciones y números no).
`integer(X)`	Devuelve éxito si X es un entero.
`atomic(X)`	Devuelve éxito si X es un "átomo" ó un entero.
`write(X)`	Escribe en el periférico por defecto el contenido de la variable X.
`read(X)`	Lee un valor del periférico por defecto y lo almacena en la variable X.
`nl`	Escribe un retorno de carro.
`==`	Compara si dos “átomos” son iguales.
`\==`	Compara si dos “átomos” son distintos.

IMPORTANTE: Todos estos predicados dan fallo en backtracking.

Reglas

Representación de reglas

"A Juan le gusta todo lo que le gusta a María"

$\forall x$ (gusta(maria,X) $\rightarrow$ gusta(juan,X))

```
gusta(juan,X)  :-  gusta(maria,X).
```

`gusta(juan,X)` es el consecuente (la cabecera de la regla).

`:-` se lee "si" [if]

`gusta(maria,X)` es el antecedente (el cuerpo de la regla).

El ámbito de la variable X llega hasta el punto `.`

En PROLOG existen dos tipos de objetivos:

- Los especificados directamente por el usuario, en el prompt `?-`.
- Los correspondientes a los antecedentes de las reglas (invocados directamente por PROLOG).

Cuando se intenta satisfacer un objetivo, PROLOG busca en la memoria de trabajo, secuencialmente, desde el principio hasta el final, o bien un hecho que se unifique con el objetivo, o bien una regla que tenga una cabecera que se unifique con el objetivo. En caso de encontrar una regla, PROLOG intentará demostrar los objetivos descritos en los antecedentes uno a uno, de izquierda a derecha.

persona.pro

```
gusta(maria,pedro).
gusta(juan,X) :- gusta(maria,X).
```

```
?- [persona].
?- gusta(juan,pedro).

        gusta(maria,pedro).                 /* FALLO */
        gusta(juan,X) :- gusta(maria,X).    /* ÉXITO */
    Unificación realizada por PROLOG: { X/pedro }
        gusta(juan,X) :- gusta(maria,X=pedro).
    Nuevo objetivo:  gusta(maria,pedro)
        gusta(maria,pedro).                 /* ÉXITO */
yes
```

En el fichero pueden ponerse los hechos junto con las reglas.

Usualmente, siempre pondremos primero los hechos y luego las reglas, ya que cuando PROLOG intenta satisfacer un objetivo, siempre lo hace recorriendo secuencialmente la base de conocimiento (de ahí que sea lógico colocar primero los hechos).

```
esHombre(juan).
esHombre(pedro).
observa(maria,pedro).
esFeliz(X) :- observa(maria,X).
```

De todas formas, es usual que el compilador obligue a que todos los predicados con el mismo nombre estén juntos (por eficiencia), por lo que el anterior consejo (poner primero los hechos y luego las reglas) se aplica sobre cada predicado.

```
esFeliz(pedro).
esHombre(juan).
esHombre(pedro).
observa(maria,pedro).
esFeliz(X) :- observa(maria,X).  /* Ilegal */

esHombre(juan).
esHombre(pedro).
observa(maria,pedro).
esFeliz(pedro).
esFeliz(X) :- observa(maria,X).  /* Correcto */
```

Formato de las Reglas

La cabeza de la regla sólo puede tener un único predicado.

Por lo tanto, la regla $\forall x\ (p(x) \rightarrow (q(x) \vee r(x)))$ no puede expresarse en PROLOG.

$\forall x\ (persona(x) \rightarrow (esRica(x) \vee esHonrada(x))$

No puede expresarse en PROLOG ☹

Por lo tanto, la regla $\forall x\ (p(x) \rightarrow (q(x) \wedge r(x)))$ sí puede expresarse en PROLOG, al poder dividirse en dos $\forall x\ (p(x) \rightarrow q(x))$ y $\forall x\ (p(x) \rightarrow r(x))$

$\forall x\ (persona(x) \rightarrow (tienePadre(x) \wedge tieneMadre(x)))$

puede dividirse en

$\forall x\ (persona(x) \rightarrow tienePadre(x))$

$\forall x\ (persona(x) \rightarrow tieneMadre(x))$

y escribirse en PROLOG como sigue:

```
tienePadre(X) :- persona(X).
tieneMadre(X) :- persona(X).
```

Si tenemos antecedentes conectados con $\vee$, también puede descomponerse la regla:

"Cualquier persona es feliz si es observada por Belén o por Federico"

$\forall x$ (obs(federico,x) $\vee$ obs(belen,x) $\rightarrow$ feliz(x))

es lógicamente equivalente a:

$\forall x$ (obs(federico,x) $\rightarrow$ feliz(x))
$\forall x$ (obs(belen,x) $\rightarrow$ feliz(x))

En PROLOG:

```
feliz(X) :- obs(federico,X).
feliz(X) :- obs(belen,X).
```

¡OJO! Las variables `X` de las dos reglas anteriores son diferentes.

En general, PROLOG sólo admite **cláusulas de Horn** (aquellas cláusulas con un único literal no negado como mucho), que corresponden a reglas $A \rightarrow B$ en su equivalente lógica $\neg A \vee B$ (el literal negado corresponde a la cabeza de la regla)

Resumiendo, PROLOG admite:

Hechos $\rightarrow$ Literales no negados.
Reglas $\rightarrow$ Reglas con un único consecuente no negado.

NOTA: Más adelante veremos cómo podemos incluir literales negados en los antecedentes de una regla (mediante un tipo especial de negación).

Backtracking en las reglas

Un único antecedente

```
obs(belen,carlos).
obs(federico,maria).
obs(federico,ines).
feliz(pedro).
feliz(X) :- obs(federico,X).
feliz(X) :- obs(belen,X).
```

Si se intenta satisfacer un objetivo de tipo `feliz` y falla la primera regla, PROLOG probará automáticamente con la segunda:

```
?- feliz(Quien).

      Quien=pedro ;

            feliz(Quien).
            feliz(X) :- obs(federico,Quien).
                  obs(federico, Quien).
                  /* Predicado obs, éxito con Quien=maria */
            /* Éxito 1ª regla con Quien=maria */

      Quien=maria ;

                  /* Backtracking */
                  obs(federico,Quien).
                  /* Predicado obs, éxito con Quien=ines
            /* Éxito 1ª regla con Quien=ines */

      Quien=ines ;

                  /* Backtracking */
                  obs(federico,Quien).
                  /* FALLO */
            /* Falla 1ª regla */
            /* Se prueba con la segunda regla. */
            feliz(X) :- obs(belen,Quien).
                  obs(belen,Quien)
                  /* Exito con Quien=carlos */
            /* Éxito 2ª regla con Quien=carlos */

      Quien=carlos ;

                  /* Backtracking */
                  Obs(belen, Quien)
                  /* FALLO */
            /* Falla 2ª regla */

      /* No hay más hechos 'feliz' ni reglas con cabecera 'feliz' */
      No
```

La disyunción pueda ponerse en la misma regla uusando el operador `;`

```
feliz(X) :- obs(federico,X) ; obs(belen,X).
```

NOTA: Será útil en la construcción de algunas reglas, para mejorar la eficiencia.

Algunas salidas pueden repetirse:

```
obs(belen,carlos).
obs(federico,maria).
obs(federico,carlos).
obs(_,maria).
feliz(pedro).
feliz(X) :- obs(federico,X). feliz(X) :- obs(belen,X).
```

```
?- feliz(X).

      X=pedro;
      X=maria;
      X=carlos;
      X=maria;
      X=carlos;
      X=maria;
```

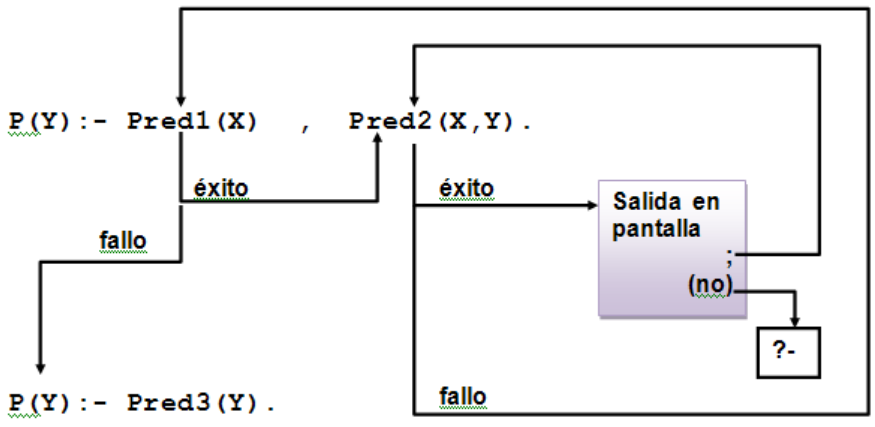
NOTA: Si quisiéramos el conjunto de todas las personas, sin repetidos, usaríamos **`setof`**, **`bagof`**.

Varios antecedentes

Si una regla tiene varios antecedentes `C :- A1 , A2 , A3`
tendrán que demostrarse todos ellos para poder demostrar `C`.
En caso contrario, la regla devolvería fallo y PROLOG buscaría otra alternativa.

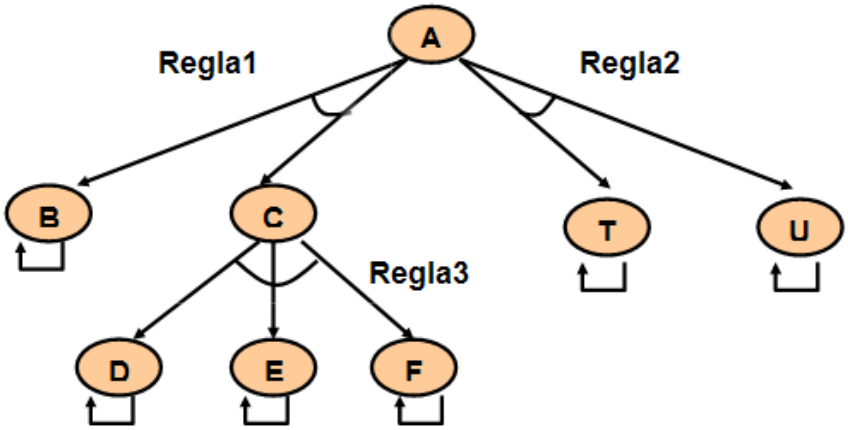
El backtracking funciona igual que con objetivos desde la línea de comandos:

- Si `A2` o `A3` devuelven fallo, se intenta resatisfacer el objetivo anterior.
- La regla devuelve fallo sólo cuando `A1` devuelve fallo.



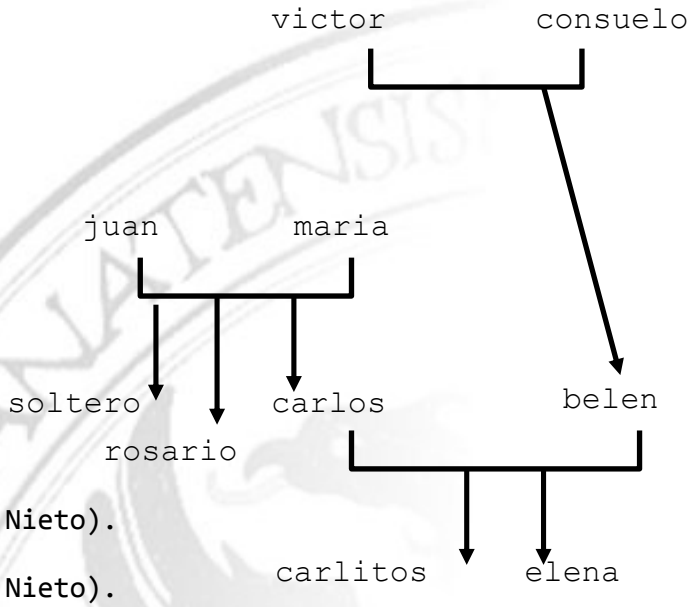
En general, el orden en el que PROLOG va intentando satisfacer los objetivos puede ilustrarse así:

```
A :-  B , C.          /* Regla 1 */
A :-  T , U.          /* Regla 2 */
C :-  D , E , F.      /* Regla 3 */
```

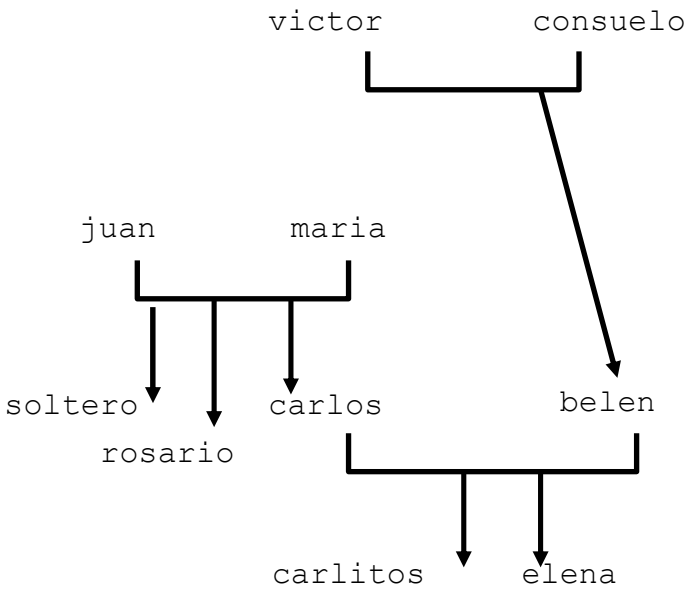


EJEMPLO

```
padre(juan,carlos).
padre(juan,rosario).
padre(juan,soltero).
padre(victor,belen).
padre(carlos,elena).
padre(carlos,carlitos).

madre(maria,soltero).
madre(maria,carlos).
madre(maria,rosario).
madre(consuelo,belen).
madre(belen,elena).
madre(belen,carlitos).

abuelo(Ab,Nieto) :-
  padre(Ab,Hijo) , padre(Hijo,Nieto).
abuelo(Ab,Nieto) :-
  padre(Ab,Hija) , madre(Hija,Nieto).
abuela(Ab,Nieto) :-
  madre(Ab,Hijo) , padre(Hijo,Nieto).
abuela(Ab,Nieto) :-
  madre(Ab,Hija) , madre(Hija,Nieto).
```



NOTA. Podría usarse un formato mejor para representar a los padres mediante un predicado `padres/3`, p.ej. `padres(carlos,belen,elena)`. Así se reúnen dos hechos en uno solo e implícitamente representamos el hecho de que carlos está casado con belen [ejercicio 5 de la relación de prácticas].



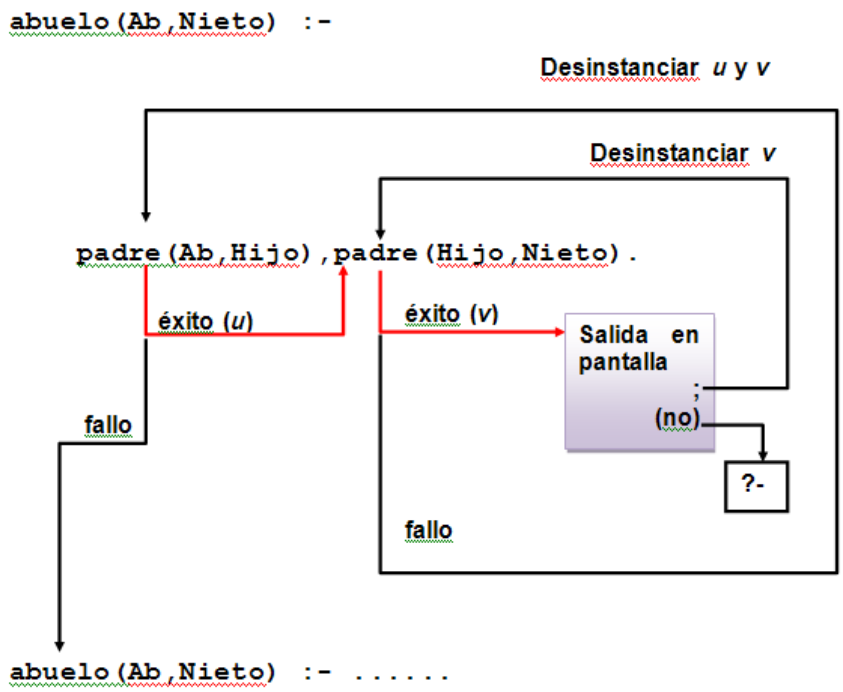

```
abuelo(Ab,Nieto) :-
 padre(Ab,Hijo) ,
 padre(Hijo,Nieto).
abuelo(Ab,Nieto) :-
 padre(Ab,Hija) ,
 madre(Hija,Nieto).

?- abuelo(juan,X).

      abuelo(Ab,Nieto) :- padre(Ab,Hijo) , padre(Hijo,Nieto).
            Ab/juan
            Nieto/X
                  padre(juan,Hijo)
                  Hijo/carlos
                        padre(juan,carlos) , padre(carlos,X).
                              padre(carlos,elena)
                              EXITO con X/elena
X=elena;

                        padre(juan,carlos) , padre(carlos,X).
                              padre(carlos,carlitos)
                              ÉXITO con X/carlitos
X=carlitos;
                  padre(juan,Hijo)
                  Hijo/rosario
                        padre(juan,rosario),padre(rosario,X).
                              padre(rosario,X)
                              FALLO

                  padre(juan,Hijo)
                  Hijo/soltero
                        padre(juan,soltero),padre(soltero,X).
                              padre(soltero,X)
                              FALLO

                  padre(juan,Hijo)
                  FALLO
```

Obsérvese la ineficiencia por la estructura escogida para los datos…

```
abuelo(Ab,Nieto) :- padre(Ab,Hija) , madre(Hija,Nieto).
      Ab/juan
      Nieto/X
            padre(juan,Hija)
            Hija/carlos
                  padre(juan,carlos) , madre(carlos,X).
                        madre(carlos,X)
                        FALLO
            padre(juan,Hija)
            Hija/rosario
                  padre(juan,rosario),madre(rosario,X).
                        madre(rosario,X)
                        FALLO
            padre(juan,Hija)
            Hija/soltero
                  padre(juan,soltero),madre(soltero,X).
                        madre(soltero,X)
                        FALLO
            padre(juan,Hija)
            FALLO
FALLO
```

no

IMPORTANTE: Este ejemplo demuestra que, cuando en un antecedente se liga alguna variable a algún valor, dicha sustitución se aplica al resto de los antecedentes de la regla. Esto mismo ocurre al principio, en el emparejamiento (unificación) del objetivo con la cabeza de la regla.



NOTA: La salida por pantalla corresponderá a aquellas variables del objetivo que coinciden con las presentes en las sustituciones u y v.

```
abuelo(Ab,Nieto) :-
      padre(Ab,Hijo) , padre(Hijo,Nieto).
abuelo(Ab,Nieto) :-
      padre(Ab,Hija) , madre(Hija,Nieto).


?- abuelo(juan,elena).

      padre(juan,carlos), padre(carlos,elena).
            ÉXITO y SALIDA
yes
```

En los objetivos constantes no hay backtracking.

NOTA: `padre(juan,Hijo)` haría backtracking si no se hubiese demostrado el objetivo.

```
?- abuelo(Ab,elena).
      /* Empieza aplicando la primera regla */
      /*    (la de los abuelos paternos)    */
      padre(Ab,Hijo)
            padre(Ab,carlos) , padre(carlos,elena).
Ab=juan;
      /* Ahora intenta resatisfacer el objetivo padre(Ab,Hijo) */
      padre(Ab,Hijo)
            padre(juan, rosario), padre(rosario, elena).
                  FALLO
            padre(juan,soltero),padre(soltero,elena).
                  FALLO
            padre(victor,belen),padre(belen,elena).
                  FALLO
            padre(carlos,elena),padre(elena,elena).
                  FALLO
            padre(carlos,carlitos),padre(carlitos,elena).
                  FALLO
      FALLO
      /* Ahora intentaría demostrar el objetivo abuelo(Ab,elena) */
      /* con la segunda regla (los abuelos maternos)             */
      padre(Ab,Hija)
            padre(juan, carlos), madre(carlos, elena).
                  FALLO
            padre(juan, rosario), madre(rosario, elena).
                  FALLO
            padre(juan,soltero),madre(soltero,elena).
                  FALLO
            padre(victor,belen),madre(belen,elena).
                  FALLO
            padre(carlos,elena),madre(elena,elena).
                  FALLO
            padre(carlos,carlitos),madre(carlitos,elena).
                  FALLO
      FALLO
no
```

Eficiencia en el uso de reglas

Puede apreciarse la ineficiencia del último ejemplo de la sección anterior:

Deberíamos poder especificar que la búsqueda sea distinta cuando el primer argumento (abuelo) sea variable con respecto a la búsqueda cuando dicho argumento es constante.

```
?- abuelo(Ab,elena).
```

Primero, se intenta satisfacer el objetivo `padre(Ab,Hijo)`. Pero esto no resulta intuitivo, porque se debería buscar el padre de `elena` en primer lugar…

SOLUCIÓN: Una regla que se aplique cuando el segundo argumento sea constante.

Planteemos una primera alternativa:

```
abuelo(Ab,Nieto) :- var(Ab),
      padre(Hijo,Nieto) , padre(Ab,Hijo)
abuelo(Ab,Nieto) :- var(Nieto),
      padre(Ab,Hijo) , padre(Hijo,Nieto)
```

Estas reglas funcionan correctamente en los ejemplos anteriores.

```
?- abuelo(X,elena).
?- abuelo(juan,X).
```

Pero no funcionan correctamente si preguntamos por:

```
?- abuelo(X,Y).
```

Los resultados salen duplicados porque, al ser variables los dos argumentos, se aplican las dos reglas con backtracking…

SOLUCIÓN CORRECTA:

```
abuelo(Ab,Nieto) :- var(Nieto),
      padre(Ab,Hijo) , padre(Hijo,Nieto)
abuelo(Ab,Nieto) :- nonvar(Nieto),
      padre(Hijo,Nieto) , padre(Ab,Hijo)
```

Mediante `var` y `nonvar` especificamos casos excluyentes:
Mejoraremos la eficiencia de las reglas cuando preveamos que los objetivos contengan variables no instanciadas, a costa de complicar el código, ya que debemos duplicar las reglas que teníamos definidas (en el ejemplo de los abuelos, debemos duplicar las 4 reglas, 2 para los abuelos y 2 para las abuelas, de forma que tendremos 8 reglas).

SOLUCIÓN ALTERNATIVA: el uso de cortes (!).

Uso de la variable anónima

En Lógica de Predicados: $\forall x\ \forall y\ \forall z\ \forall w\ (p(x,y) \wedge q(z,w) \rightarrow r(x))$

Algunos compiladores permiten usar nombres distintos de variables de forma aislada:

```
r(X) :- p(X,Y) , q(Z,W).
```

pero esto puede provocar errores sintácticos cuando queremos poner la misma variable y nos equivocamos al escribirla. Por tanto, mejor usar la variable anónima _

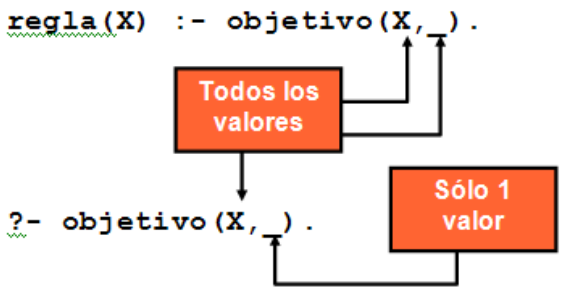
Dentro de una regla, la variable anónima es una más, cuantificada universalmente. De hecho, distintas apariciones de _ en una misma regla, corresponde a variables distintas.

```
r(X) :- p(X,_) , q(_,_).
```

equivale realmente a

```
r(X) :- p(X,Y) , q(Z,W).
```

IMPORTANTE: _ es una variable más, por lo que en backtracking, se intentará resatisfacer el predicado, deshaciendo la última unificación:



EJEMPLO $\forall x\ \forall y\ \forall z$ (padres(x,y,z) → casados(x,y))

```
padres(juan,belen,carlos).
padres(juan,belen,elena).
casados(Padre,Madre) :- padres(Padre,Madre,_).

?- casados(juan,Quien).
Quien = belen;
Quien = belen;
no
```

Cuidado cuando utilicemos la variable anónima en la cabecera de la regla:

"Todos los hombres tienen un padre"
$\forall x \exists y$ (esHombre(x) $\rightarrow$ padre(y,x))

```
padre(_,X) :- esHombre(X).
```

INCORRECTA
Esta regla representaría: $\forall x \forall y$ (esHombre(x) $\rightarrow$ padre(y,x))

La forma correcta se obtiene transformando primero en F.N.C.:
$(\forall x)$ esHombre(x) $\rightarrow$ PadreDe(elPadreDe(x),x)

```
padreDe(elPadreDe(X),X) :- esHombre(X).
```

```
?- padreDe(Quien,juan).
```

PROLOG unifica
padreDe(Quien,juan) con padreDe(elPadreDe(X),X)
y por lo tanto realiza las sustituciones:
X/Juan y Quien/elPadreDe(X)
es decir,
X/Juan y Quien/elPadreDe(juan)

```
Quien = elPadreDe(juan).
```

EJERCICIO RESUELTO
Supongamos las siguientes relaciones de una base de datos relacional:

```
proveedor(IdProv, NombreProv, CiudadProv)
pieza(IdPieza, NombrePieza, PrecioPieza)
prov_Pieza(IdProv, IdPieza, Cantidad)
```

Queremos construir un predicado que responda a la siguiente consulta sobre la BDR: Indique las ciudades donde puede ser suministrada una pieza (indicada por su nombre).

```
ciudades_para_pieza(NombrePieza,CiudadProv) :-
      pieza(IdPieza,NombrePieza,_),
      prov_Pieza(IdProv,IdPieza,_),
      proveedor(IdProv,_,CiudadProv).

?- ciudades_para_pieza(tornillo, Ciudad).
?- ciudades_para_pieza(NombrePieza, granada).
```

NOTA: Si pasan una variable en `NombrePieza`, la regla funcionará correctamente pero tendremos la misma ineficiencia que con el ejemplo de los abuelos.

Unificación (=)

Supongamos dos predicados:

predicado (término1 , función(término2))
predicado (x , y)

Ambos predicados pueden unificarse en Lógica de predicados con la sustitución

{ x / término1, y / función(término2) }

En PROLOG se usa el término unificar tanto para hablar de sustitución de una variable por un término como de unificación de términos en el sentido clásico de Lógica de Predicados.

```
writeln(X) :- write(X),nl.

?- writeln('Hola').
```

PROLOG también provee un operador específico para implementar la unificación: =

Este operador no se usará para unificar átomos (en el sentido de Lógica de Predicados) sino términos; es decir, constantes, variables y funciones de ellas.

a) Las constantes y términos en general son unificables únicamente con ellos mismos:

```
?- cte1 = cte1                  /* Éxito */
?- cte1 = cte2                  /* Fallo */
?- func(cte1) = func(cte1)      /* Éxito */
```

b) `X=Y` (o bien `Y=X`), cuando `X` es una variable no instanciada e `Y` una variable instanciada con una constante o un término en general (sin variables). Entonces, `X=Y` instancia `X` con el valor de `Y`.

```
?- X=cte                        /* Éxito */
X=cte
?- X=cte,write(X),nl.           /* Éxito */
cte
X=cte
?- Y=cte, X=Y.                  /* Éxito */
Y=cte
X=cte
?- Y=f(cte), X=Y.               /* Éxito */
Y=f(cte)
X=f(cte)
```

c) Dos términos son unificables si cada uno de sus argumentos lo es.
NOTA: En el caso de las funciones, obviamente sus nombres han de coincidir.

```
?- funcion(cte)=funcion(X)    /* Éxito */
X=cte
?- f(cte,otraf(cte2))=f(X,Y)  /* Éxito */
X=cte
Y=otraf(cte2)
```

d) `X=Y` (o bien `Y=X`) cuando `X` e `Y` son variables no instanciadas. Entonces, ambas variables compartirán el mismo valor en su ámbito.

```
?- f(X)=f(Y) , X=2 , write(Y) , nl.      /* Éxito */
2
X=2
Y=2
```

El operador `\=` es el operador de no unificable. Por lo tanto, devuelve éxito cuando `=` devuelve fallo y viceversa.

EJEMPLO

```
tieneGarras(aguila_calzada).
vuela(aguila_calzada).
tienePicoFuerte(aguila_calzada).

esUnTipoDe(Animal, Especie) :-
      tieneGarras(Animal),
      vuela(Animal),
      tienePicoFuerte(Animal),
      Especie = rapaz.


?- esUnTipoDe(aguila_calzada, E).
E = rapaz;
```

Aritmética

Predicados relacionales

```
X==Y   X\==Y   X<Y    X>Y    X=<Y    X>=Y    X=:=Y
```

== representa igualdad estricta, por lo que sólo es éxito en los casos:

```
?- cte == cte
?- Var == Var  /* el mismo nombre de variable */
```

\== será éxito cuando == sea fallo (y viceversa)

Si alguna variable no está instanciada, con los operadores relacionales
< , >, =<, >=, =:= se producirá un error.

=:= compara el resultado de dos expresiones aritméticas (ver siguiente apartado)

```
?- 4-1 =:= 1+2          ?- X+2 =:= 1+2
yes                     Error
```

Operadores y asignación

Algunos de los operadores definidos en PROLOG son

```
X+Y    X-Y    X*Y    X/Y    X mod Y
```

Si bien cada compilador concreto ofrecerá sus propios operadores.

Estos operadores son los únicos ejemplos de funciones en PROLOG tal y como se entienden en un lenguaje imperativo (esto es, devuelven un valor al utilizarlas).

La asignación del resultado de una función a una variable se realiza con el operador `is`

```
(<variable>|<constante>) is  <función aritmética>
```

- Si en la parte izquierda aparece una variable, `is` realiza una asignación.
- Si aparece una constante, se realiza una comparación.

```
?- 1 is 1          ?- 1 is 1+0        ?- 1 is 1+1
yes                yes                no

?- X is 1+2        ?- 1+2 is 1+2      ?- X+2 is 1+2
X=3;               no                 no
no
```

Resumiendo:

Predicado	Relación	Sustitución de variables	Evaluación aritmética
==	identidad	NO	NO
=	unificable	SÍ (izq+der)	NO
=:=	mismo valor	NO	SÍ (izq+der)
is	asignación / comparación	SÍ (izq)	SÍ (der)

```
cuadrado(X,Y) :- Y is X*X.


?-    cuadrado(3,Y).
Y=9;
no    /* is da fallo en backtracking */

?-    cuadrado(3,10).
no

?-    cuadrado(X,9).
ERROR: is/2: Arguments are not sufficiently instantiated
```

En el último caso, sería deseable responder:

```
?-    cuadrado(X,9).
X=3
```

Para conseguirlo, distinguiremos cuando el primer argumento es una variable y de cuando no lo es:

```
cuadrado(X,Y) :- var(X),nonvar(Y), X is sqrt(Y).
cuadrado(X,Y) :- nonvar(X), Y is X*X.
```

Recursividad

Supongamos que queremos averiguar quiénes son los ascendientes de una persona. El problemas es obviamente recursivo y se implementa en PROLOG de forma natural:

```
ascend(Asc,Desc) :- padre(Asc,Desc).          /* Caso base */

ascend(Asc,Desc) :- padre(Asc,Hijo),
                    ascend(Hijo,Desc).        /* Llamada recursiva */
```

Supongamos el siguiente árbol genealógico:

```
padre(pedro,juan).
padre(juan,inma).
padre(juan,carlos).
padre(carlos,elena).
```

```
?- ascend(pedro,X).

      /* Regla 1 */
      padre(pedro,X).
            X/juan
            padre(pedro,juan)
            ÉXITO
X=juan;
            FALLO

      /* Regla 2 */
      padre(pedro,Hijo),ascend(Hijo,X).
            padre(pedro,Hijo)
                  padre(pedro,juan)
                  EXITO
            Hijo=juan
            ascend(juan,X)
                  /* Regla 1 */
                  padre(juan,X)
                        X/inma
                        padre(juan,inma)
                        ÉXITO
X=inma;
                        X/carlos
                        padre(juan,carlos)
                        ÉXITO
X=carlos;
                        FALLO
```

```
                        /* Regla 2 */
                        padre(juan,Hijo),ascend(Hijo,X).
                              padre(juan,inma)
                                    EXITO
                              Hijo=inma
                              ascend(inma,X)
                                    /* Regla 1 */
                                    padre(inma,X)
                                    FALLO
                                    /* Regla 2 */
                                    padre(inma,Hijo),ascend(Hijo,X).
                                          padre(inma,Hijo)
                                          FALLO
                              FALLO

                              padre(juan,carlos)
                                    ÉXITO
                              Hijo=carlos
                              ascend(carlos,X)
                                    /* Regla 1 */
                                    padre(carlos,X)
                                          X/elena;
                                          padre(carlos,elena)
                                                ÉXITO
X=elena;
                                          FALLO
                                    /* Regla 2 */
                                    padre(carlos,Hijo),ascend(Hijo,X).
                                          padre(carlos,elena)
                                          Hijo=elena.
                                          ascend(elena,X)
                                          /* Regla 1 */
                                          ...
                                          FALLO
                                          /* Regla 2 */
                                          FALLO
                        FALLO                          <-- ascend(carlos,X)
            FALLO                                      <-- ascend(juan,X)
no                                                     <-- ascend(pedro,X)
```

¿Valdría lo siguiente?

```
ascend(Asc,Desc) :- padre(Asc,Desc).
ascend(Asc,Desc) :- ascend(Hijo,Desc), padre(Asc,Hijo),
```

En objetivos con variables, el objetivo se intenta resatisfacer, por lo que se realizaría una llamada recursiva y nunca terminaría la ejecución del programa PROLOG.

Cuando esperemos objetivos con variables, debemos poner un predicado no recursivo (que alguna vez pueda devolver FALLO en backtracking) **ANTES de la llamada recursiva.**

EJERCICIO
Cálculo de la suma de los primeros N enteros.
Además, cuando N sea 0 ó negativo, devolveremos 1.

```
suma(N,Resultado) :-
      N=<1 , Resultado is 1.

suma(N,Resultado) :-
      N>1 ,
      suma(N-1,ResultAux),
      Resultado is ResultAux+N.
```

> NOTA: En la primera regla, en vez de asignar 1 a Resultado, podemos poner directamente `suma(N,1) :- N=<1`.

¿Qué pasaría si quitásemos la comprobación `N>1` *en la segunda regla?*

```
?- suma(1,X).
X=1;
X=2;  !!!!!
.....
```

Era de esperar:

La segunda regla sólo se debe aplicar en los casos en los que N>1. Es importante que no se nos olvide **especificar todas las condiciones de aplicabilidad de las reglas**.

Listas

Muy utilizadas en PROLOG para almacenar series de términos (e incluso otras listas):

```
[a , f(b) , c]

[a , [b ,f(b)] , h , [i , j]]

[]    /* Lista vacía */
```

Una lista se compone de:

- **Cabeza** [*head*]: El primer elemento de la lista
  (lo que hay antes de la primera coma).

- **Cola** [*tail*]: La lista formada por todos los elementos menos el primero
  (esto es, toda la lista menos su cabeza).

```
[a,b,c]
Cabeza:     a
Cola:       [b,c]
```

Para extraer la cabeza y la cola de lista se usa la notación:

[<Cabeza> | <Cola>]

Lista	[X|_]	[_|X]
[a,b,c]	a	[b,c]
[[a,b],c]	[a,b]	[c]
[a,[b,c]]	a	[[b,c]]
[a,[b,c],d]	a	[[b,c],d]
[X+Y,Z]	X+Y	[Z]
[]	ninguna	Ninguna
[a]	a	[]

La notación `[Cabeza|Cola]` permite unificaciones más complejas como:

```
?- [a,b,c,d] = [Cab1,Cab2|Cola]
Cab1=a
Cab2=b
Cola=[c,d]
```

Incluso podemos usar esta notación para construir una lista nueva (en vez de extraer el contenido de una lista):

```
?- L1=[a,b],L2=[c,d],L3=[L1|L2].
L1 = [a, b]
L2 = [c, d]
L3 = [[a, b], c, d]

?- L1=[a,b],L2=[c,d],L3=[L1,L2].
L1 = [a, b]
L2 = [c, d]
L3 = [[a, b], [c, d]]

?- L1=a,L2=[c,d],L3=[L1|L2].
L1 = a
L2 = [c, d]
L3 = [a, c, d]
```

EJEMPLO
Mostrar por pantalla los elementos de una lista.

```
mostrarLista(Lista) :-
      Lista=[Cabeza|Resto], write(Cabeza), nl, mostrarLista(Resto).
```

La unificación podemos hacerla directamente en la misma cabeza de la regla:

```
mostrarLista([Cabeza|Resto]) :-
      write(Cabeza) , nl , mostrarLista(Resto).
```

Cuando sólo nos quede la lista vacía, la unificación `[]=[Cabeza|Resto]` dará fallo. Esto puede darnos problemas si esta regla es utilizada desde otra regla:

```
mostrarListaAsteriscos(Lista) :-
      mostrarLista(Lista), write('***'), nl.
```

¡Siempre da fallo!

Para resolverlo, basta poner:

```
mostrarLista([]).

mostrarLista([Cabeza|Resto]) :-
      write(Cabeza) , nl , mostrarLista(Resto)
```

EJERCICIO
Hallar la longitud de una lista (es decir, el número de elementos que contiene).

Interfaz:

```
?- contarElementos([],N).
```

Implementación:

```
contarElementos([],0).

contarElementos([Cab|Cola],Total) :-
      contarElementos(Cola,Aux),
      Total is Aux+1.
```

EJERCICIO
Contar el número de veces que un elemento se encuentra repetido en una lista.

Interfaz:

```
?- contarRep(Elem, Lista, NumVeces)
```

Implementación:

```
contarRep(_, Lista, N):-
      Lista = [], N=0.

contarRep(Elem, Lista, N) :-
      Lista = [Cab|Cola],
      Elem == Cab ,
      contarRep(Elem,Cola,Aux),
      N is Aux+1.

contarRep(Elem, Lista, N) :-
      Lista = [Cab|Cola],
      Elem \== Cab ,
      contarRep(Elem,Cola,Aux),
      N is Aux.
```

Aplicando la unificación directamente en las cabeceras, obtenemos:

```
contarRep(_,[],0).

contarRep(Cab,[Cab|Cola],N) :-
      contarRep(Cab,Cola,Aux) ,
      N is Aux+1.

contarRep(Elem,[Cab|Cola],N) :-
      Elem \== Cab ,
      contarRep(Elem,Cola,Aux),
      N is Aux.
```

EJERCICIO
Borrar un elemento de una lista.
p.ej. Borrando a de [b,a,g,a,h,b], quedaría [a,g,a,h,a]

Interfaz

```
?- borrar(Elem,Lista,Res)
```

Implementación

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

borrar(Cab, [Cab|Cola],Res) :-
      borrar(Elem,Cola,Aux),
      Res = Aux.

borrar(Elem, [Cab|Cola],Res) :-
      Elem \== Cab,
      borrar(Elem,Cola,Aux),
      Res = [Cab|Aux].
```

Modificación de la memoria de trabajo

Pueden añadirse elementos a la memoria de trabajo durante la sesión de trabajo.

Para ello, se usan los predicados

- `asserta` (para añadir al principio) y
- `assertz` (para añadir al final).

Para eliminar hechos (y también reglas) se usa `retract`. Pero sólo pueden borrarse hechos añadidos con `assert(a/z)`, y no los almacenados con `consult`.

```
borra_primero :- retract(esPadre(_)).
```

sólo borraría el primer `esPadre(_)`. Para borrarlos todos, hay que escribir:

```
borra_todos :-
      retract(esPadre(_)),
      borra_todos.
```

o bien:

```
borraTodos :-
      retract(esPadre(_)),
      fail.
borraTodos.
```

NOTA: Algunos compiladores, como SWI-Prolog, dan fallo cuando se añade un predicado que coincide con alguno de los predicados añadidos desde un fichero.

EJERCICIO: Sobre el ejemplo de los proveedores, construya un predicado que añada una tupla a la relación de proveedores manteniendo la restricción de llave primaria. Use el predicado `not/1` que devuelve fallo (éxito) si el argumento es éxito (fallo).

```
add_prov(IdProv,Nombre,Ciudad) :-
   nonvar(IdProv),nonvar(Nombre),nonvar(Ciudad),
   proveedores(IdProv,_,_),
   fail.

add_prov(IdProv,Nombre,Ciudad) :-
   nonvar(IdProv),nonvar(Nombre),nonvar(Ciudad),
   not(proveedores(IdProv,_,_)),
   assertz(proveedores(IdProv,Nombre,Ciudad)).
```

Obviamente, la primera regla puede suprimirse y el programa seguiría funcionando correctamente.

El corte (!)

Interrupción del backtracking

PROLOG proporciona el operador de corte `!` para interrumpir el backtracking, es decir, para cambiar la estrategia de control y no seguir buscando por ciertas ramas.

```
Obj :- Obj1, Obj2, Obj3, !, Obj4, Obj5, Obj6
```

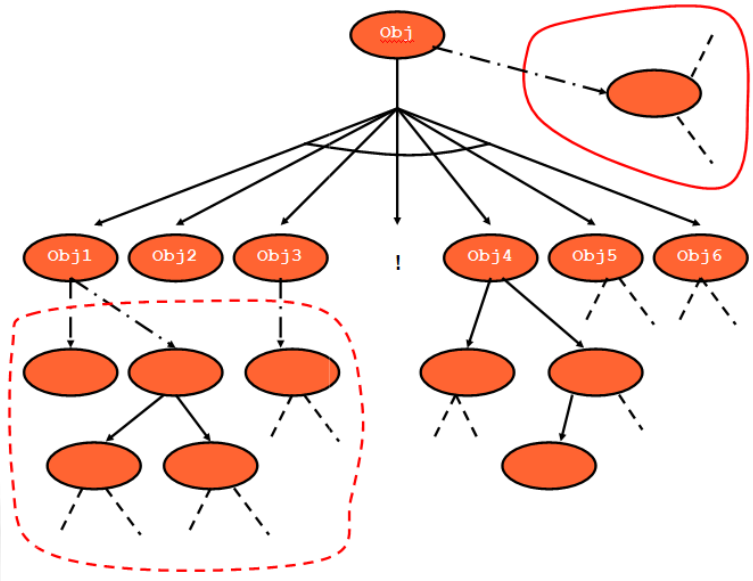
- Si falla `Obj3`, se vuelve atrás y se intenta `Obj2` (backtracking).
- Si tiene éxito Obj3, pasa el corte y se satisface el objetivo `Obj4`, pero ya no se volverá a intentar `Obj3`
- Si falla `Obj4`, falla la regla.

Por tanto, ahora la regla puede fallar en dos situaciones:
cuando `Obj1` falla y cuando `Obj4` falla

Una segunda consecuencia de haber pasado por el corte es que no se intenta aplicar ninguna otra regla con cabecera `Obj.`

```
Obj :- Obj1, Obj2, Obj3, !, Obj4, Obj5, Obj6
```

Supongamos que `Obj3` tiene éxito:
Las ramas que el corte impide explorar son las encerradas en círculo.



NOTA: Hasta que se consigue que `Obj3` sea éxito, sí se explora parte del grafo:
El backtracking está permitido antes de que `Obj3` sea éxito

EJEMPLO

Supongamos que, en una biblioteca, si una persona tiene prestado un libro, sólo se le permite consultar la base de datos y leer libros dentro del recinto. En caso contrario, también se le permite el préstamo a domicilio, así como préstamos de otras librerías.

Las consultas a la BD son servicios básicos:

```
servicio_basico(consulta_BD).
servicio_basico(consulta_interna).
```

Las consultas externas son servicios adicionales:

```
servicio_adicional(prestamo_interno).
servicio_adicional(prestamo_externo).
```

Los servicios completos son los básicos y los adicionales.

```
servicio_completo(X) :- servicio_basico(X).
servicio_completo(X) :- servicio_adicional(X).
```

Ahora, si el cliente tiene un libro prestado, se le ofrecen sólo los servicios básicos:

```
servicio(Cliente,Servicio) :-
      prestado(Cliente,_),
      ! ,
      servicio_basico(Servicio).
```

A todos los clientes que **no** tengan prestado un libro, se les ofrecen todos los servicios:

```
servicio(_,Servicio) :-
      servicio_completo(Servicio).
```

Por ejemplo:

```
cliente('Juan').
cliente('Pedro').
cliente('Maria').
…
prestado('Juan',clave107).
prestado('Pedro',clave145).

?- servicio('Juan',Serv).
Serv = consulta_BD;
Serv = consulta_interna;
no.
?- servicio('Maria',Serv).
Serv = consulta_BD;
Serv = consulta_interna;
Serv = prestamo_interno;
Serv = prestamo_externo;
no.
```

Sin embargo, si ponemos:

```
?- servicio(Cliente,Serv).
Cliente = 'Juan'
Serv = consulta_BD;
Cliente = 'Juan'
Serv = consulta_interna;
no.
```

Sólo aparecen los servicios disponibles para Juan (no salen los básicos de Pedro ni los completos de María) al no haber backtracking después del corte:

```
servicio(Cliente,Servicio) :-
      prestado(Cliente,_),
      ! , /* Una vez llegados aquí, no hay vuelta atrás */
      servicio_basico(Servicio).
```

MORALEJA

El uso del corte en reglas cuya cabecera puede venir con variables no instanciadas, puede dar problemas.

Para solucionar el problema, utilizamos un predicado auxiliar que nos "separe" la aplicación del corte del resto del proceso de búsqueda:

```
servicioAUX(Cliente,Servicio) :-
      prestado(Cliente,_),
      ! ,
      servicio_basico(Servicio).

servicioAUX(_,Servicio) :-
      servicio_completo(Servicio).

servicio(Cliente,Servicio) :-
      cliente(Cliente),
      servicioAUX(Cliente,Servicio).
```

- Si `Cliente` viene no instanciado, el predicado `cliente(Cliente)` fijará un cliente y buscará sus servicios. Al hacer backtracking ,volverá a buscar otro cliente (y así con todos.)

- Si `Cliente` viene instanciado, el predicado `cliente(Cliente)` tendrá éxito si es un cliente fichado y fallará en otro caso.

EJERCICIO
Aplicar cortes, donde se pueda, para mejorar la eficiencia del ejercicio en el que se contaba el número de veces que un elemento está repetido en una lista

```
contarRep(Elem,[],0) :- !.

contarRep(Cab,[Cab|Cola],N) :-
      contarRep(Cab,Cola,Aux) , N is Aux+1 , !.

contarRep(Elem,[Cab|Cola],N) :-
      contarRep(Elem,Cola,Aux) , N is Aux.
```

¡OJO! En esta versión de `contarRep`, debemos ver la segunda regla para poder entender la tercera. Por eso, si no resulta necesario, es mejor evitar el uso de cortes.

Aplicaciones usuales del corte

Por el buen camino

"Si has llegado hasta aquí, entonces has elegido la regla correcta para conseguir el objetivo buscado. No intentes otras reglas."

EJEMPLO
El del préstamo de libros.

EJEMPLO
Sumar los valores menores o iguales que N.

```
suma(N,1)   :- N=<1.
suma(N,Res) :- N>1, suma(N-1,ResAux), Res is ResAux + N.
```

Con el corte aumentamos la eficiencia, aunque el código se hace más difícil de leer:

```
suma(N,1)   :- N=<1 , !.
suma(N,Res) :- suma(N-1,ResAux), Res is ResAux + N.
```

Por el mal camino

"Si has llegado hasta aquí, entonces deberías parar en el intento de satisfacer el objetivo buscado, pues no vas por el buen camino."

Se consigue en PROLOG gracias a la combinación corte-fallo.

EJEMPLO
Sobre una base de datos de contribuyentes, queremos definir los contribuyentes normales como aquellos que

- No son extranjeros.
- Si están casados, su cónyuge no ingresa más de 2000.
- En otro caso, sus ingresos propios están entre 700 y 2000.

Primer intento:

```
cont_normal(X) :- extranjero(X),fail.

cont_normal(X) :- ingresosNormales(X).
```

```
?- cont_normal('John Wiley').
yes !!!! <--- Falla porque PROLOG aplica la segunda regla
```

Solución: Combinación corte-fallo.

```
cont_normal(X) :- extranjero(X), ! , fail.

cont_normal(X) :-
      casado(X,Y) , ingresos(Y,Ing), Ing>2000 , ! , fail.

cont_normal(X) :-
      ingresos(X,Ing) , Ing>700, Ing<2000.
```

NOTA: En CLIPS, cuando no queríamos que se aplicase otra regla después de ejecutar una dada, suprimíamos un hecho de control que teníamos que incluir en la regla.

El operador de negación (not)

PROLOG implementa un tipo especial de negación con el operador `not`. La idea es que si un predicado `p` tiene éxito, `not(p)` debe fallar, y viceversa.

```
conyuge_es_Rico(X) :-
      casado(X,Y), ingresos(Y,Ing), Ing>2000

cont_normal(X) :-
      not(extranjero(X)),
      not(conyuge_es_rico(X)),
      ingresos(X,Ing) , Ing>700, Ing<2000.


?- cont_normal('John').
no                              /* extranjero('John') tiene éxito */
                                /* not (extranjero('John')) falla */
```

Pero el uso de **`not`** no está exento de problemas:

- Preguntas con constantes no incluidas en la base de conocimiento:

  ```
  ?- cont_normal('ExtranjeroNoFichado').
  yes !!!
  ```

  Por eso se dice que `not` representa una negación por fallo: Si PROLOG no puede demostrar (falla en demostrar) `extranjero('ExtranjeroNoFichado')`, entonces `not(extranjero('ExtranjeroNoFichado'))` se asume cierto. Es consecuencia de la **Hipótesis de Mundo Cerrado**.

  MORALEJA: Cuando en una regla interviene un `not`, no debemos preguntar por constantes que no estén incluidas en nuestra base de conocimiento.

- Preguntas con variables:

  ```
  ?- cont_normal(VAR).
  ```

  Cuando PROLOG evalúa `not(extranjero(VAR))`, algunos compiladores generan un error. Otros compiladores fallan si existe algún átomo del tipo `extranjero/1`, por lo que el objetivo `cont_normal(VAR)` termina con **`no`** !!!!!

  MORALEJA: Cuando en una regla interviene un `not`, debemos tener cuidado al incluir variables en las preguntas.

  La razón interna es que `not` se implementa con la combinación corte-fallo. Y, siempre que hay un corte, puede haber problemas con variables no instanciadas.

Solución a los problemas del uso de `not` (análoga a la del préstamo de libros):

```
conyugeRico(X) :-
      casado(X,Y), ingresos(Y,Ing), Ing>2000

cont_normal(X) :-
      contribuyente(X),
      not(extranjero(X)),
      not(conyugeRico(X)),
      ingresos(Y,Ing), Ing>700, Ing<2000.


?- cont_normal('ExtranjeroNoFichado').
no
      /* contribuyente('ExtranjeroNoFichado') devuelve FALLO */


?- cont_normal(X).
X=juan;
…
      /* contribuyente(X) va instanciando X en backtracking */
```

Si fuese legal, la definición de `not` sería como sigue:

```
      not (p(X)) :- p(X),!,fail.
      not (p(_)).
```

- Si `p(X)` tiene éxito para algún valor de `X`,
  la primera regla nos dice que `not(p(X))` debe fallar.

- Si `p(X)` falla para todos los posibles valores de `X`,
  la segunda regla nos dice que `not(p(X))` tiene éxito.