Prácticas de Lógica

Prolog

Faraón Llorens Largo
Mª Jesús Castel de Haro

DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN E
INTELIGENCIA ARTIFICIAL
Universidad de Alicante

Contenido

© 1996-2001 Faraón Llorens

© 1996-2001 Faraón Llorens

Estos apuntes pretenden ser un documento de apoyo para los estudiantes que realizan las prácticas en el lenguaje de programación Prolog, en alguna de las asignaturas de lógica impartidas por el departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante: *Lógica de Primer Orden*, *Lógica Computacional* y *Ampliación de Lógica*.

El Prolog descrito, tanto la sintaxis como los operadores básicos, siguen el estándar de Edimburgo [Clocksin y Mellish, 1987] así como las especificaciones ISO [Covington, 1993] y [Deransart y otros, 1996].

Se trata de una breve exposición de la forma de trabajo, de las características principales y de los predicados predefinidos estándar del lenguaje Prolog. No se trata de una completa descripción de la sintaxis y la semántica de Prolog ni de la forma de programar en dicho lenguaje. Para ello existen excelentes libros, algunos de ellos referenciados en la bibliografía. Son muy recomendables los libros [Clocksin y Mellish, 1987], [Bratko, 1990] y [Sterling y Shapiro, 1994], y para un nivel más avanzado [O'Keefe, 1990]

No se pretende explorar toda la potencia del lenguaje de programación lógica Prolog. Eso sobrepasaría las pretensiones de nuestras asignaturas. Simplemente queremos mostrar a los estudiantes cómo pueden escribir programas (bases de conocimientos) con el lenguaje de la lógica, por medio de hechos y reglas. Posteriormente podemos ejecutar estos programas realizando preguntas que el sistema nos responderá a partir de la información que conoce. Es decir, no vamos a estudiar Prolog desde el punto de vista de un lenguaje de programación, sino como una aplicación directa de la lógica de primer orden.

Por todo ello, únicamente vamos a describir un pequeño subconjunto de predicados predefinidos, en especial aquellos con marcado carácter lógico, que no interfieren en el control, no entraremos en detalle en los predicados de entrada/salida. Todos los ejemplos han sido probados utilizando SWI-Prolog [Wielemaker, 2001], que es el interprete/compilador utilizado en las clases prácticas.

Para cualquier duda o sugerencia podéis dirigiros a la dirección de correo electrónico:

logica@dccia.ua.es

Más información se puede encontrar en el sitio web:

http://www.dccia.ua.es/logica/prolog

© 1996-2001 Faraón Llorens

1. PROGRAMACIÓN LÓGICA

El lenguaje de programación PROLOG ("PROgrammation en LOGique") fue creado por Alain Colmerauer y sus colaboradores alrededor de 1970 en la Universidad de Marseille-Aix[1], si bien uno de los principales protagonistas de su desarrollo y promoción fue Robert Kowalski[2] de la Universidad de Edimburgh. Las investigaciones de Kowalski proporcionaron el marco teórico, mientras que los trabajos de Colmerauer dieron origen al actual lenguaje de programación, construyendo el primer interprete Prolog. David Warren[3], de la Universidad de Edimburgh, desarrolló el primer compilador de Prolog (WAM – "Warren Abstract Machine"). Se pretendía usar la lógica formal como base para un lenguaje de programación, es decir, era un primer intento de diseñar un lenguaje de programación que posibilitara al programador especificar sus problemas en lógica. Lo que lo diferencia de los demás es el énfasis sobre la especificación del problema. Es un lenguaje para el procesamiento de información simbólica. PROLOG es una realización aproximada del modelo de computación de Programación Lógica sobre una máquina secuencial. Desde luego, no es la única realización posible, pero sí es la mejor elección práctica, ya que equilibra por un lado la preservación de las propiedades del modelo abstracto de Programación Lógica y por el otro lado consigue que la implementación sea eficiente.

El lenguaje PROLOG juega un importante papel dentro de la Inteligencia Artificial, y se propuso como el lenguaje nativo de las máquinas de la quinta generación ("Fifth Generation Kernel Language", FGKL) que quería que fueran Sistemas de Procesamiento de Conocimiento. La expansión y el uso de este lenguaje propició la aparición de la normalización del lenguaje Prolog con la norma ISO (propuesta de junio de 1993).

PROLOG es un lenguaje de programación para ordenadores que se basa en el lenguaje de la Lógica de Primer Orden y que se utiliza para resolver problemas en los que entran en juego *objetos* y *relaciones* entre ellos. Por ejemplo, cuando decimos "Jorge tiene una moto", estamos expresando una relación entre un objeto (Jorge) y otro objeto en particular (una moto). Más aún, estas relaciones tienen un orden específico (Jorge posee la moto y no al contrario). Por otra parte, cuando realizamos una pregunta (¿Tiene Jorge una moto?) lo que estamos haciendo es indagando acerca de una relación. Además, también solemos usar reglas para describir relaciones: "dos personas son hermanas si ambas son hembras y tienen los mismos padres". Como veremos más adelante, esto es lo que hacemos en Prolog.

[1] A. Colmerauer. *Les Systèmes-Q, ou un formalisme pouranalyser et synthétiser des phrases sur ordinateur*. Internal Report 43, Computer Science Dpt., Université de Montreal, septiembre, 1970.
A. Colmerauer, H. Kanoui, P. Roussel y R. Pasero. *Un Systeme de Communication Homme-Machine en Français*, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[2] R. Kowalski. *Predicate Logic as a Programming Language*. En Proc. IFIP, Amsterdam, 1974.

[3] D. Warren. *The runtime environment for a prolog compiler using a copy algorithm*. Technical Report 83/052, SUNY and Stone Brook, New York, 1983.

© 1996-2001 Faraón Llorens

Una de las ventajas de la programación lógica es que se especifica *qué* se tiene que hacer (*programación declarativa*), y no *cómo* se debe hacer (*programación imperativa*). A pesar de esto, Prolog incluye algunos predicados predefinidos meta-lógicos, ajenos al ámbito de la Lógica de Primer Orden, (var, nonvar, ==, ...), otros extra-lógicos, que tienen un efecto lateral, (write, get, ...) y un tercer grupo que nos sirven para expresar información de control de como realizar alguna tarea (el corte, ...). Por tanto, Prolog ofrece un sistema de programación práctico que tiene algunas de las ventajas de claridad y declaratividad que ofrecería un lenguaje de programación lógica y, al mismo tiempo, nos permite un cierto control y operatividad.

© 1996-2001 Faraón Llorens

2. PROLOG Y EL LENGUAJE DE LA LÓGICA DE PRIMER ORDEN

La Lógica de Primer Orden analiza las frases sencillas del lenguaje (fórmulas atómicas o elementales) separándolas en *Términos* y *Predicados*. Los términos hacen referencia a los objetos que intervienen y los predicados a las propiedades o relaciones entre estos objetos. Además, dichas fórmulas atómicas se pueden combinar mediante *Conectivas* permitiéndonos construir fórmulas más complejas, llamadas fórmulas moleculares.

2.1. PREDICADOS

Se utilizan para expresar propiedades de los objetos, *predicados monádicos*, y relaciones entre ellos, *predicados poliádicos*. En Prolog los llamaremos **hechos**. Debemos tener en cuenta que:

- Los nombres de todos los objetos y relaciones deben comenzar con una letra minúscula.
- Primero se escribe la relación o propiedad: *predicado*
- Y los objetos se escriben separándolos mediante comas y encerrados entre paréntesis: *argumentos*.
- Al final del hecho debe ir un punto ("**.**").

```
simbolo_de_predicado(arg1,arg2,...,argn).
```

Tanto para los símbolos de predicado como para los argumentos, utilizaremos en Prolog constantes atómicas.

Ejemplos (`ej01.pl`):

```
/* Predicados monádicos: PROPIEDADES */

/* mujer(Per) <- Per es una mujer    */
mujer(clara).
mujer(chelo).

/* hombre(Per) <- Per es un hombre   */
hombre(jorge).
hombre(felix).
hombre(borja).

/* moreno(Per) <- Per tiene el pelo de color oscuro */
moreno(jorge).


/* Predicados poliádicos: RELACIONES */
```

© 1996-2001 Faraón Llorens

```
/* tiene(Per,Obj) <- Per posee el objeto Obj */
tiene(jorge,moto).

/* le_gusta_a(X,Y) <- a X le gusta Y */
le_gusta_a(clara,jorge).
le_gusta_a(jorge,clara).
le_gusta_a(jorge,informatica).
le_gusta_a(clara,informatica).

/* es_padre_de(Padre,Hijo-a) <- Padre es el padre de Hijo-a */
es_padre_de(felix,borja).
es_padre_de(felix,clara).

/* es_madre_de(Madre,Hijo-a) <- Madre es la madre de Hijo-a */
es_madre_de(chelo,borja).
es_madre_de(chelo, clara).

/* regala(Per1,Obj,Per2) <- Per1 regala Obj a Per2 */
regala(jorge, flores, clara).
```

2.2. TÉRMINOS

Los términos pueden ser *constantes* o *variables*, y suponemos definido un dominio no vacío en el cual toman valores (Universo del Discurso). En la práctica se toma como dominio el Universo de Herbrand. Para saber cuántos individuos del universo cumplen una determinada propiedad o relación, *cuantificamos* los términos.

Las **constantes** se utilizan para dar nombre a objetos concretos del dominio, dicho de otra manera, representan individuos conocidos de nuestro Universo. Además, como ya hemos dicho, las constantes atómicas de Prolog también se utilizan para representar propiedades y relaciones entre los objetos del dominio. Hay dos clases de constantes:

- *Átomos*: existen tres clases de constantes atómicas:
  - Cadenas de letras, dígitos y subrayado (_) empezando por letra minúscula.
  - Cualquier cadena de caracteres encerrada entre comillas simples (').
  - Combinaciones especiales de signos: "?-", ":-", ...

- *Números*: se utilizan para representar números de forma que se puedan realizar operaciones aritméticas. Dependen del ordenador y la implementación[4].
  - *Enteros*: en la implementación de Prolog-2 puede utilizarse cualquier entero que el intervalo $[-2^{23},2^{23}-1]=[-8.388.608,8.388.607]$.

[4] Los ordenadores, vía hardware, resuelven eficientemente el manejo de los números y de las operaciones aritméticas, por tanto, en la práctica, la programación lógica lo deja en sus manos, trabajando en una aritmética estándar independiente del lenguaje.

© 1996-2001 Faraón Llorens

- *Reales*: decimales en coma flotante, consistentes en al menos un dígito, opcionalmente un punto decimal y más dígitos, opcionalmente *E*, un «+» o «-» y más dígitos.

Ejemplos de constantes:

átomos válidos	átomos no válidos	números válidos	nº no válidos
f	2mesas	-123	123-
vacio	Vacio	1.23	.2
juan_perez	juan-perez	1.2E3	1.
'Juan Perez'	_juan	1.2E+3	1.2e3
a352	352a	1.2E-3	1.2+3

Las **variables** se utilizan para representar objetos cualesquiera del Universo u objetos desconocidos en ese momento, es decir, son las incógnitas del problema. Se diferencian de los átomos en que empiezan siempre con una letra mayúscula o con el signo de subrayado (_). Así, deberemos ir con cuidado ya que cualquier identificador que empiece por mayúscula, será tomado por Prolog como una variable. Para trabajar con objetos desconocidos cuya identidad no nos interesa, podemos utilizar la *variable anónima* (_). Las variables anónimas no están compartidas entre sí.

Ejemplos de variables:

X	
Sumando	
Primer_factor	
_indice	
_	(variable anónima)

Una variable está *instanciada* cuando existe un objeto determinado representado por ella. Y está *no instanciada* cuando todavía no se sabe lo que representa la variable. Prolog no soporta asignación destructiva de variables, es decir, cuando una variable es instanciada su contenido no puede cambiar. Como veremos más adelante, la manipulación de datos en la programación lógica se realiza por medio de la unificación.

Explícitamente Prolog no utiliza los símbolos de **cuantificación** para las variables, pero implícitamente sí que lo están. En general, todas las variables que aparecen están cuantificadas universalmente, ya que proceden de la notación en forma clausal, y, por tanto, todas las variables están cuantificadas universalmente aunque ya no escribamos explícitamente el cuantificador (paso 6ª de la transformación a forma clausal : eliminación de cuantificadores universales). Pero veamos que significado tienen dependiendo de su ubicación. Si las fórmulas atómicas de un programa lógico contienen variables, el significado de estas es :

- Las variables que aparecen en los hechos están cuantificadas universalmente ya que en una cláusula todas las variables que aparecen están cuantificadas universalmente de modo implícito.

gusta(jorge,X). equivale a la fórmula $\forall x$ gusta(jorge,x)

y significa que a jorge le gusta cualquier cosa

© 1996-2001 Faraón Llorens

- Las variables que aparecen en la cabeza de las reglas (átomos afirmados) están cuantificadas universalmente. Las variables que aparecen en el cuerpo de la regla (átomos negados), pero no en la cabeza, están cuantificadas existencialmente.

  abuelo(X,Y) :- padre(X,Z), padre(Z,Y). equivale a la fórmula

  $\forall x\ \forall y\ \forall z\ [abuelo(x,y) \vee \neg padre(x,z) \vee \neg padre(z,y)]$
  $\forall x\ \forall y\ [abuelo(x,y) \vee \forall z\ \neg[padre(x,z) \wedge padre(z,y)]\ ]$
  $\forall x\ \forall y\ [abuelo(x,y) \vee \neg\exists z\ [padre(x,z) \wedge padre(z,y)]\ ]$
  $\forall x\ \forall y\ [\ \exists z\ [padre(x,z) \wedge padre(z,y)] \rightarrow abuelo(x,y)\ ]$

  que significa que para toda pareja de personas, una será el abuelo de otra si existe alguna persona de la cuál el primero es padre y a su vez es padre del segundo

- Las variables que aparecen en las preguntas están cuantificadas existencialmente.

  ?- gusta(jorge,X) equivale a la fórmula $\forall x\ \neg gusta(jorge,x) \equiv \neg\exists x\ gusta(jorge,x)$

  y que pregunta si existe algo que le guste a jorge, ya que utilizamos refutación y por tanto negamos lo que queremos demostrar.

2.3. CONECTIVAS LÓGICAS

Puede que nos interese trabajar con sentencias más complejas, fórmulas moleculares, que constarán de fórmulas atómicas combinadas mediante conectivas. Las conectivas que se utilizan en la Lógica de Primer Orden son: *conjunción*, *disyunción*, *negación* e *implicación*.

La **conjunción**, "y", la representaremos poniendo una coma entre los objetivos "**,**" y consiste en *objetivos* separados que Prolog debe satisfacer, uno después de otro:

```
X , Y
```

Cuando se le da a Prolog una secuencia de objetivos separados por comas, intentará satisfacerlos por orden, buscando objetivos coincidentes en la Base de Datos. Para que se satisfaga la secuencia se tendrán que satisfacer todos los objetivos.

La **disyunción**, "o", tendrá éxito si se cumple alguno de los objetivos que la componen. Se utiliza un punto y coma ";" colocado entre los objetivos:

```
X ; Y
```

La disyunción lógica también la podemos representar mediante un conjunto de sentencias alternativas, es decir, poniendo cada miembro de la disyunción en una cláusula aparte, como se puede ver en el ejemplo `es_hijo_de`.

© 1996-2001 Faraón Llorens

La **negación** lógica no puede ser representada explícitamente en Prolog, sino que se representa implícitamente por la falta de aserción : “no”, tendrá éxito si el objetivo *X* fracasa. No es una verdadera negación, en el sentido de la Lógica, sino una negación “por fallo”. La representamos con el predicado predefinido *not* o con \+:

```
not(X)
\+ X
```

La **implicación** o **condicional**, sirve para significar que un hecho depende de un grupo de otros hechos. En castellano solemos utilizar las palabras “si ... entonces ...”. En Prolog se usa el símbolo “:-” para representar lo que llamamos una **regla**:

```
cabeza_de_la_regla :- cuerpo_de_la_regla.
```

La cabeza describe el hecho que se intenta definir; el cuerpo describe los objetivos que deben satisfacerse para que la cabeza sea cierta. Así, la regla:

C :- O_1, O_2, ..., O_n.

puede ser leída declarativamente como:

“La demostración de la cláusula *C* se sigue de la demostración de los objetivos O_1, O_2, ..., O_n.”

o procedimentalmente como:

“Para ejecutar el procedimiento *C*, se deben llamar para su ejecución los objetivos O_1,O_2,...,O_n.”

Otra forma de verla es como una implicación lógica “al revés” o “hacia atrás”:

cuerpo_de_la_regla → cabeza_de_la_regla

Un mismo nombre de variable representa el mismo objeto siempre que aparece en la regla. Así, cuando *X* se instancia a algún objeto, todas las *X* de dicha regla también se instancian a ese objeto (*ámbito de la variable*).

Por último, daremos una serie de definiciones. Llamaremos **cláusulas** de un predicado tanto a los hechos como a las reglas. Una colección de cláusulas forma una **Base de Conocimientos**.

Ejemplos (`ej01.pl`):

```
/* Conjunción de predicados   */
le_gusta_a(clara,jorge), le_gusta_a(clara,chocolate).

/* Disyunción de predicados   */
le_gusta_a(clara,jorge); le_gusta_a(jorge,clara).

/* Negación de predicados     */
not(le_gusta_a(clara,jorge)).
```

© 1996-2001 Faraón Llorens

```
/* o también como */
\+ le_gusta_a(clara,jorge).

/* Condicional: REGLAS        */

/* novios(Per1,Per2) <- Per1 y Per2 son novios */
novios(X,Y) :-    le_gusta_a(X,Y),
                  le_gusta_a(Y,X).

/* hermana_de(Per1,Per2) <- Per1 es la hermana de Per2 */
hermana_de(X,Y)  :-  mujer(X),
                     es_padre_de(P,X), es_madre_de(M,X),
                     es_padre_de(P,Y), es_madre_de(M,Y).

/* Ejemplo de disyunción con ; y con diferentes cláusulas */
/* 1. con ; sería :                                       */
es_hijo_de(X,Y)  :-  (es_padre_de(Y,X) ; es_madre_de(Y,X)).

/* 2. con cláusulas diferentes quedaría:                  */
es_hijo_de(X,Y)  :-  es_padre_de(Y,X).
es_hijo_de(X,Y)  :-  es_madre_de(Y,X).
```


© 1996-2001 Faraón Llorens

3. ESTRUCTURA DE UN PROGRAMA

El hecho de programar en Prolog consiste en dar al ordenador un Universo finito en forma de hechos y reglas, proporcionando los medios para realizar inferencias de un hecho a otro. A continuación, si se hacen las preguntas adecuadas, Prolog buscará las respuestas en dicho Universo y las presentará en la pantalla. La programación en Prolog consiste en:

- declarar algunos HECHOS sobre los objetos y sus relaciones,
- definir algunas REGLAS sobre los objetos y sus relaciones, y
- hacer PREGUNTAS sobre los objetos y sus relaciones.

Programa Prolog: Conjunto de afirmaciones (*hechos* y *reglas*) representando los conocimientos que poseemos en un determinado dominio o campo de nuestra competencia.

Ejecución del programa: Demostración de un Teorema en este Universo, es decir, demostración de que una conclusión se deduce de las premisas (afirmaciones previas).

Programa Prolog
Base de Conocimientos + Motor de Inferencia

Un sistema Prolog está basado en un comprobador de teoremas por resolución para *cláusulas de Horn*. La regla de resolución no nos dice que cláusulas elegir ni que literales unificar dentro de cada cláusula. La estrategia de resolución particular que utiliza Prolog es una forma de *resolución de entrada lineal* (árbol de búsqueda estándar). Para la búsqueda de cláusulas alternativas para satisfacer el mismo objetivo, Prolog adopta una estrategia de *primero hacia abajo* (recorrido del árbol en profundidad). Por todo esto, el orden de las cláusulas (hechos y reglas) de un determinado procedimiento es importante en Prolog, ya que determina el orden en que las soluciones serán encontradas, e incluso puede conducir a fallos en el programa. Más importante es, si cabe, el orden de las metas a alcanzar dentro del cuerpo de una regla.

3.1. PREGUNTAS

Las preguntas son las herramientas que tenemos para recuperar la información desde Prolog. Al hacer una pregunta a un programa lógico queremos determinar si esa pregunta es *consecuencia lógica* del programa. Prolog considera que todo lo que hay en la Base de Datos es verdad, y lo que no, es falso. De manera que si Prolog responde "yes" es que ha podido demostrarlo, y si responde "no" es que no lo ha podido

© 1996-2001 Faraón Llorens

demostrar (no debe interpretarse como “falso” si no que con lo que Prolog conoce no puede demostrar su veracidad).

Cuando se hace una pregunta a Prolog, éste efectuará una búsqueda por toda la Base de Datos intentando encontrar hechos que *coincidan* con la pregunta. Dos hechos “coinciden” (se pueden unificar) si sus predicados son el mismo (se escriben de igual forma) y si cada uno de los respectivos argumentos son iguales entre sí.

```
?- simbolo_de_predicado(arg1,arg2,...,argn).
```

Ejemplos (`ej01.pl`):

```
?- le_gusta_a(clara,jorge).
yes

?- le_gusta_a(jorge,cafe).
no

?- capital_de(madrid,españa).
no
```

Cuando a Prolog se le hace una pregunta con una variable, dicha variable estará inicialmente no instanciada. Prolog entonces recorre la Base de Datos en busca de un hecho que *empareje* con la pregunta: los símbolos de predicado y el número de argumentos sean iguales, y emparejen los argumentos. Entonces Prolog hará que la variable se instancie con el argumento que esté en su misma posición en el hecho. Prolog realiza la búsqueda por la Base de Datos en el orden en que se introdujo. Cuando encuentra un hecho que empareje, saca por pantalla los objetos que representan ahora las variables, y marca el lugar donde lo encontró. Prolog queda ahora a la espera de nuevas instrucciones, sacando el mensaje “More (y/n) ?”[5]:

- si pulsamos “n”+<RETURN> cesará la búsqueda,
- si pulsamos “y”+<RETURN> reanudará la búsqueda comenzando donde había dejado la marca; decimos que Prolog está intentando *resatisfacer* la pregunta.

Ejemplos (`ej01.pl`):

```
?- le_gusta_a(jorge,X).
X=clara
More (y/n)? y
X=informatica
More (y/n)? y
no
```

La conjunción y el uso de variables pueden combinarse para hacer preguntas muy interesantes:

```
?- le_gusta_a(clara,jorge),le_gusta_a(jorge,cafe).
no
?- le_gusta_a(clara,X),le_gusta_a(jorge,X).
```

[5] Depende de la implementación. Lo descrito aquí es como funciona el Prolog-2. El SWI-Prolog no saca mensaje y queda a la espera. Para activar la nueva búsqueda de soluciones basta con pulsar ";". Con <↵> finaliza.

© 1996-2001 Faraón Llorens

```
X=informatica
More (y/n)? n
yes
```

Hemos buscado algo que le guste tanto a Clara como a Jorge. Para ello, primero averiguamos si hay algo que le guste a Clara, marcándolo en la Base de Conocimientos e instanciando la variable *X*. Luego, averiguamos si a Jorge le gusta ese objeto *X* ya instanciado. Si el segundo objetivo no se satisface, Prolog intentará resatisfacer el primer objetivo. Es importante recordar que cada objetivo guarda su propio marcador de posición.

```
?- le_gusta_a(clara,informatica); le_gusta_a(jorge,cafe).
yes
?- le_gusta_a(clara,cafe); le_gusta_a(jorge,cafe).
no
?- le_gusta_a(clara,X); le_gusta_a(jorge,X).
X=jorge
More (y/n)? y
X=informatica
More (y/n)? y
X=clara
More (y/n)? y
X=informatica
More (y/n)? y
no
?- not(le_gusta_a(clara,jorge)).
no
?- not(le_gusta_a(jorge,cafe)).
yes
?- hermana_de(clara,borja).
yes
?- hermana_de(borja,X).
no
?- hermana_de(clara,X).
X=borja
More (y/n)? n
yes
```

Ejercicio:

Piensa que ocurriría si a la pregunta de que si queremos más soluciones le contestamos que sí. ¿ Cómo lo solucionarías ?

© 1996-2001 Faraón Llorens

4. SINTAXIS

La sintaxis de un lenguaje describe la forma en la que se nos está permitido juntar palabras entre sí. Los programas en Prolog se construyen a partir de **términos**. Un término es una *constante*, una *variable* o una *estructura*. Todo término se escribe como una secuencia de *caracteres*. Para escribir un comentario lo encerraremos entre los signos /* y */ o desde el símbolo % hasta el final de línea. Así, Prolog pasa por alto los comentarios, pero los debemos añadir a nuestros programas para aclararlos y que el propio programa quede documentado (Ver apartado 10.2 Estilo De Programación En Prolog).

```
/* ... comentario ... */
```

```
% comentario de una sola línea
```

Ejemplo :

```
/* Esto es un comentario
   de más de una línea    */

% mujer(Per) <- Per es una mujer
mujer(clara).     % Esto es también comentario
mujer(chelo).
```

4.1. CARACTERES

Los nombres de constantes y variables se construyen a partir de cadenas de caracteres. Prolog reconoce dos tipos de caracteres:

- *Imprimibles*: hacen que aparezca un determinado signo en la pantalla del ordenador. Se dividen en cuatro categorías:
  letras mayúsculas: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.
  letras minúsculas: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.
  dígitos numéricos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
  signos: ! " # $ % & ' ( ) = - ^ | / \ { } [ ] _ @ + * ; : < > , . ?

- *No imprimibles*: no aparecen en forma de signo en la pantalla, pero realizan una determinada acción: nueva línea, retorno de carro, ...

Cada carácter tiene un entero entre 0 y 127 asociado a él, este es su código ASCII ("American Standard Code for Information Interchange").

© 1996-2001 Faraón Llorens

4.2. ESTRUCTURAS

Una **estructura** es un único objeto que se compone de una colección de otros objetos, llamados componentes, lo que facilita su tratamiento. Una estructura se escribe en Prolog especificando su *nombre*, y sus *componentes* (*argumentos*). Las componentes están encerradas entre paréntesis y separadas por comas; el nombre se escribe justo antes de abrir el paréntesis:

$$nombre\ (\ comp_1, comp_2, ..., comp_n\ )$$

Por ejemplo, podemos tener una estructura con el nombre libro, que tiene tres componentes: titulo, autor y año de edición. A su vez el autor puede ser una estructura con dos componentes: nombre y apellido.

```
libro(logica_informatica, autor(jose, cuena), 1985)
```

Como se puede ver, en Prolog, la sintaxis para las estructuras es la misma que para los hechos. Como podrás comprobar cuando trabajes más a fondo en Prolog, hay muchas ventajas en representar incluso los mismos programas Prolog como estructuras. Las estructuras se verán con más detalle en el apartado 8.3 Componentes De Estructuras.

4.3. OPERADORES

En Prolog están predefinidos los **operadores aritméticos** y **relacionales** típicos, con la precedencia habitual entre ellos:

```
                ^
              mod
              *  /
              +  -
    =  \=  =<  >=  <  >
```

Para poder leer expresiones que contengan operadores necesitamos conocer los siguientes atributos:

* POSICIÓN: *Prefijo*: el operador va delante de sus argumentos.
Infijo: el operador se escribe entre los argumentos.
Postfijo: el operador se escribe detrás de sus argumentos.

* PRECEDENCIA: Nos indica el orden en que se realizan las operaciones. El operador más prioritario tendrá una precedencia 1 y el menos, 1201 (depende de la implementación).

© 1996-2001 Faraón Llorens

*ASOCIATIVIDAD: Sirve para quitar la ambigüedad en las expresiones en las que hay dos operadores, uno a cada lado del argumento, que tienen la misma precedencia.

Para conocer las propiedades de los operadores ya definidos podemos utilizar el predicado predefinido:

```
current_op(?Precedencia,?Especificador,?Nombre)
```

donde: *Precedencia* es un entero indicando la clase de precedencia,
Especificador es un átomo indicando la posición y la asociatividad, y
Nombre es un átomo indicando el nombre que queremos que tenga el operador.

Operador	Símbolo	Precedencia	Especificador
Potencia	^ ó **	200	xfx
Producto	*	400	yfx
División	/	400	yfx
División entera	//	400	yfx
Resto división entera	mod	400	yfx
Suma	+	500	yfx
Signo positivo	+	500	fx
Resta	-	500	yfx
Signo negativo	-	500	fx
Igualdad	=	700	xfx
Distinto	\=	700	xfx
Menor que	<	700	xfx
Menor o igual que	=<	700	xfx
Mayor que	>	700	xfx
Mayor o igual que	>=	700	xfx
Evaluación aritmética	is	700	xfx

Tabla 1: Operadores aritméticos y relacionales predefinidos en SWI-Prolog

Para la posición-asociatividad utilizamos átomos especiales del tipo:

xfx xfy yfy yfx xf yf fx fy

que nos ayudan a ver el uso del posible operador, representando **f** al operador, **x** un argumento indicando que cualquier operador del argumento debe tener una clase de precedencia estrictamente menor que este operador, e **y** un argumento indicando que puede contener operadores de la misma clase de precedencia o menor.

Ejemplo:

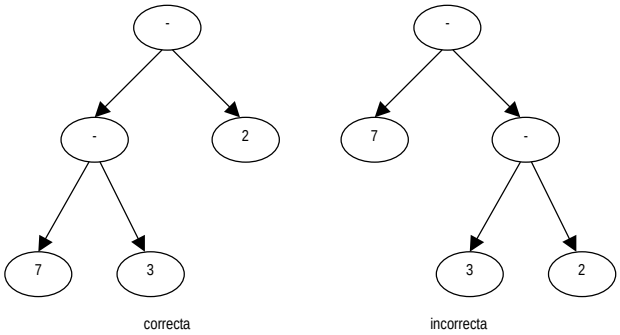
El operador - declarado como **yfx** determina que la expresión a - b - c sea interpretada como (a - b) - c, y no como a - (b - c) ya que la x tras la f exige que el argumento que va tras el primer - contenga un operador de precedencia estrictamente menor:

7 - 3 - 2 → (7 - 3) -2 = 4 - 2 = 2 correcta

© 1996-2001 Faraón Llorens

7 - (3 - 2) = 7 - 1 = 6 incorrecta

Gráficamente sería:



Si queremos declarar en Prolog un nuevo operador que sea reconocido por el sistema con una posición, clase de precedencia y asociatividad determinadas, utilizaremos el predicado predefinido *op*, cuya sintaxis es:

```
op(+Precedencia,+Especificador,+Nombre).
```

Al trabajar con expresiones aritméticas y relacionales, los argumentos de estas estructuras deben ser constantes numéricas o variables instanciadas a constantes numéricas.

Ejemplo (`ej02.pl`):

```
/* horoscopo(Signo,DiaIni,MesIni,DiaFin,MesFin) <-
           pertenecen al signo del horoscopo Signo los nacidos
           entre el DiaIni del MesIni y el DiaFin del MesFin   */
horoscopo(aries,21,3,21,4).
horoscopo(tauro,21,4,21,5).
horoscopo(geminis,21,5,21,6).
horoscopo(cancer,21,6,21,7).
horoscopo(leo,21,7,21,8).
horoscopo(virgo,21,8,21,9).
horoscopo(libra,21,9,21,10).
horoscopo(escorpio,21,10,21,11).
horoscopo(sagitario,21,11,21,12).
horoscopo(capricornio,21,12,21,1).
horoscopo(acuario,21,1,21,2).
horoscopo(piscis,21,2,21,3).

/* signo(Dia,Mes,Signo) <- los nacidos el Dia de Mes pertenecen al
                        signo del zodiaco Signo                  */
signo(Dia,Mes,Signo) :- horoscopo(Signo,D1,M1,D2,M2),
                  ( ( Mes=M1, Dia>=D1) ; ( Mes=M2, Dia=<D2) ).

?- signo(8, 5, tauro).
yes
?- signo(7, 8, acuario).
no
?- signo(7, 8, Signo).
```

© 1996-2001 Faraón Llorens

```
Signo=leo
More (y/n)? y
no
```

Ejercicio:

Piensa que ocurrirá si preguntamos `?- signo(7,X,Signo)`.

Y si preguntamos `?- signo(X,7,Signo)` . ¿ Por qué ?

El ejemplo contesta afirmativamente a preguntas del tipo `?- signo(74,4,tauro)`. Modifica el ejemplo para que trabaje con el número de días correcto para cada mes.

Los operadores no hacen que se efectúe ningún tipo de operación aritmética. El predicado de evaluación es el operador infijo *is*:

```
-Numero is +Expresion
```

donde: *Expresion* es un término que se interpreta como una expresión aritmética, con todos sus valores instanciados.
Numero puede ser una variable o una constante numérica.

Ejemplo (`ej03.pl`):

```
/* poblacion(Prov,Pob) <- la población, en miles de habitantes,
                      de la provincia Prov es Pob            */
poblacion(alicante,1149).
poblacion(castellon,432).
poblacion(valencia,2066).

/* superficie(Prov,Sup) <- la superficie, en miles de km², de la
                       provincia Prov es Sup                 */
superficie(alicante,6).
superficie(castellon,7).
superficie(valencia,11).

/* densidad(Prov,Den) <- la densidad de población, habitantes/km²,
                     de la provincia Prov es Den             */
densidad(X,Y) :- poblacion(X,P),
                 superficie(X,S),
                 Y is P/S.

?- densidad(alicante, X).
X=191'500

?- 6=4+2.
no
?- 6 is 4+2.
yes
?- N is 4+2.
N=6
?- N is N+1.
no
?- N=4, N is 4+1.
no
```


© 1996-2001 Faraón Llorens

Prolog también incorpora **funciones aritméticas** (`abs`, `sig`, `min`, `max`, `random`, `round`, `integer`, `float`, `sqrt`, `sin`, `cos`, `tan`, `log`, `log10`, `exp`, ...).

Ejemplos:

```
?- X is abs(-7.8).
X=7.8
?- X is min(9,2*3+1).
X=7
?- X is random(10).
X=5
?- X is random(10).
X=6
?- X is random(10).
X=8
?- X is sqrt(9).
X=3
?- X is sqrt(10).
X=3.16228
?- X is sin(2).
X=0.909297
?- X is sin(2*pi/4).
X=1
?- X is log(1000).
X=6.90776
?- X is log(e).
X=1
?- X is log(e**5).
X=5
?- X is log10(1000).
X=3
```

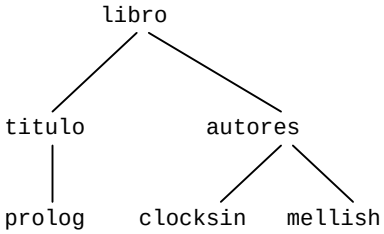
© 1996-2001 Faraón Llorens

5. ESTRUCTURAS DE DATOS

5.1. ÁRBOLES

Es más fácil entender la forma de una estructura complicada si la escribimos como un **árbol** en el que el nombre es un nodo y los componentes son las ramas.

Ejemplo: libro(titulo(prolog), autores(clocksin, mellish))



5.2. LISTAS

Las listas son unas estructuras de datos muy comunes en la programación no numérica. Una **lista** es una secuencia ordenada de elementos que puede tener cualquier longitud. Los elementos de una lista pueden ser cualquier término (constantes, variables, estructuras) u otras listas.

Las listas pueden representarse como un tipo especial de árbol. Una lista puede definirse recursivamente como:

* una **lista vacía []**, sin elementos, o
* una estructura con dos componentes:
  cabeza: primer argumento
  cola: segundo argumento, es decir, el resto de la lista.

El final de una lista se suele representar como una cola que contiene la lista vacía. La cabeza y la cola de una lista son componentes de una estructura cuyo nombre es "**.**".

Ejemplo:

Así, la lista que contiene un solo elemento a es

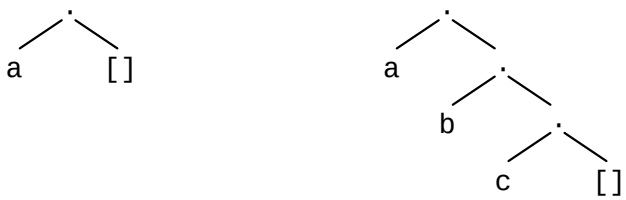
. (a, [])

y la lista de tres elementos [a, b, c] podría escribirse

. (a, . (b, . (c, [])))

siempre terminando con la lista vacía.

© 1996-2001 Faraón Llorens



En Prolog utilizaremos una notación más sencilla de las listas que dispone a los elementos de la misma separados por comas, y toda la lista encerrada entre corchetes. Así, las listas anteriores quedarían como *[a]* y *[a, b, c]*, que es un tipo de notación más manejable. Existe, también, una notación especial en Prolog para representar la lista con cabeza *X* (elemento) y cola *Y* (lista):

```
[ X | Y ]
```

Ejemplos:

Lista	Cabeza (elemento)	Cola (lista)
[a, b, c]	A	[b, c]
[a]	a	[ ]
[ ]	(no tiene)	(no tiene)
[[el, gato], maullo]	[el, gato]	[maullo]
[el, [gato, maullo]]	el	[[gato, maullo]]
[el, [gato, maullo], ayer]	el	[[gato, maullo], ayer]
[X+Y, x+y]	X+Y	[x+y]

Diseño de procedimientos de manejo de listas:

Vamos a utilizar la técnica de *refinamientos sucesivos* para el diseño de procedimientos de manejo de listas. Como no sabemos de antemano el tamaño de las listas deberemos utilizar la recursividad para recorrer las listas.

Ejemplo:

Procedimiento miembro que comprueba si un determinado elemento pertenece a una lista.

Esquema de la relación:

miembro(Elem, Lista) <- el término Elem pertenece a la lista Lista

Definición intuitiva:

Una determinada carta está en un mazo de cartas si es la primera o si está en el resto del mazo.

1ª aproximación: traducción literal

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

© 1996-2001 Faraón Llorens

2ª aproximación: recogida de parámetros

```
miembro(E,[X|Y]) :- X=E.
miembro(E,[X|Y]) :- miembro(E,Y).
```

3ª aproximación: unificación de variables

```
miembro(X,[X|Y]).
miembro(E,[X|Y]) :- miembro(E,Y).
```

4ª aproximación: ahorro de espacio en memoria (variable anónima)

```
miembro(X,[X|_]).
miembro(X,[_|Y]) :- miembro(X,Y).
```

Operaciones con listas: (ej04.pl)

```
/* miembro(Elem,Lista) <- el término Elem pertenece a Lista */
miembro(X,[X|_]).
miembro(X,[_|Y]) :- miembro(X,Y).

/* nel(Lista,N) <- el numero de elementos de la lista Lista es N */
nel([],0).
nel([X|Y],N) :- nel(Y,M),
                N is M+1.

/* es_lista(Lista) <- Lista es una lista */
es_lista([]).
es_lista([_|_]).

/* concatena(L1,L2,L3) <- concatenación de las listas L1 y L2
                       dando lugar a la lista L3 */
concatena([],L,L).
concatena([X|L1],L2,[X|L3]) :- concatena(L1,L2,L3).

/* ultimo(Elem,Lista) <- Elem es el ultimo elemento de Lista */
ultimo(X,[X]).
ultimo(X,[_|Y]) :- ultimo(X,Y).

/* inversa(Lista,Inver) <- Inver es la inversa de la lista Lista */
inversa([],[]).
inversa([X|Y],L) :- inversa(Y,Z),
                    concatena(Z,[X],L).

/* borrar(Elem,L1,L2) <- se borra el elemento Elem de la lista L1
                         obteniéndose la lista L2 */
borrar(X,[X|Y],Y).
borrar(X,[Z|L],[Z|M]) :- borrar(X,L,M).

/* subconjunto(L1,L2) <- la lista L1 es un subconjunto de lista L2 */
subconjunto([X|Y],Z) :- miembro(X,Z),
                        subconjunto(Y,Z).
subconjunto([],Y).

/* insertar(Elem,L1,L2) <- se inserta el elemento Elem en la lista L1
                           obteniéndose la lista L2 */
insertar(E,L,[E|L]).
insertar(E,[X|Y],[X|Z]) :- insertar(E,Y,Z).

/* permutacion(L1,L2) <- la lista L2 es una permutación de lista L1 */
permutacion([],[]).
permutación([X|Y],Z) :- permutacion(Y,L),
                        insertar(X,L,Z).
```


© 1996-2001 Faraón Llorens

Preguntas:

```
?- miembro(d,[a,b,c,d,e]).
yes

?- miembro(d,[a,b,c,[d,e]]).
no

?- miembro(d,[a,b,c]).
no

?- miembro(E,[a,b]).
E=a
More (y/n)? y
E=b
More (y/n)? y
no

?- nel([a,b,[c,d],e],N).
N=4

?- es_lista([a,b,[c,d],e]).
yes

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

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

?- concatena(L1,L2,[a,b]).
L1=[], L2=[a,b]
More (y/n)? y
L1=[a], L2=[b]
More (y/n)? y
L1=[a,b], L2=[]
More (y/n)? y
no
```

Ejercicios:

1.- Comprueba el funcionamiento de las restantes operaciones con listas y prueba con diferentes tipos de ejemplos.

2.- Escribe un procedimiento que obtenga el elemento que ocupa la posición n de una lista o la posición que ocupa el elemento e:

```
?- elemento(E,3,[a,b,c,d]).
E=c

?- elemento(c,N,[a,b,c,d]).
N=3
```

3.- Modifica el procedimiento borrar para que borre el elemento que ocupa la posición n de la lista:

```
?- borrar(3,[a,b,c,d],L).
L=[a,b,d]
```

© 1996-2001 Faraón Llorens

Prolog ya incorpora **procedimientos de manejo de listas** (`is_list`, `append`, `member`, `delete`, `select`, `nth0`, `nth1`, `last`, `reverse`, `flatten`, `length`, `merge`, ...). Hemos visto como se definían algunos de ellos para comprender mejor su funcionamiento y que en un futuro seamos capaces de crear nuevos procedimientos a la medida que necesitemos.

Podemos utilizar predicados predefinidos de ordenación de listas:

```
sort(+Lista,-ListaOrdenada)
```

```
msort(+Lista,-ListaOrdenada)
```

Ordenan los elementos de la lista *Lista* y la devuelven en *ListaOrdenada*. El primero elimina los elementos repetidos, mientras que el segundo no.

Ejemplos:

```
?- sort([b,f,a,r,t,r,h,a,r],L).
L = [a, b, f, h, r, t]
?- msort([b,f,a,r,t,r,h,a,r],L).
L = [a, a, b, f, h, r, r, r, t]
```


© 1996-2001 Faraón Llorens

6. ESTRUCTURAS DE CONTROL

6.1. RECURSIÓN

Las definiciones recursivas se encuentran frecuentemente en los programas Prolog. Fijémonos en cómo algunos predicados de manejo de listas vistos en el punto anterior están definidos **recursivamente**, es decir, el cuerpo de la cláusula se llama a sí mismo. En la recursividad debemos tener cuidado en que se cumplan las "condiciones de límite" (punto de parada cuando utilizamos recursividad). Podemos observar que en la llamada de la cláusula recursiva al menos uno de los argumentos crece o decrece, para así poder unificar en un momento dado con la cláusula de la condición de parada.

Ejemplo:

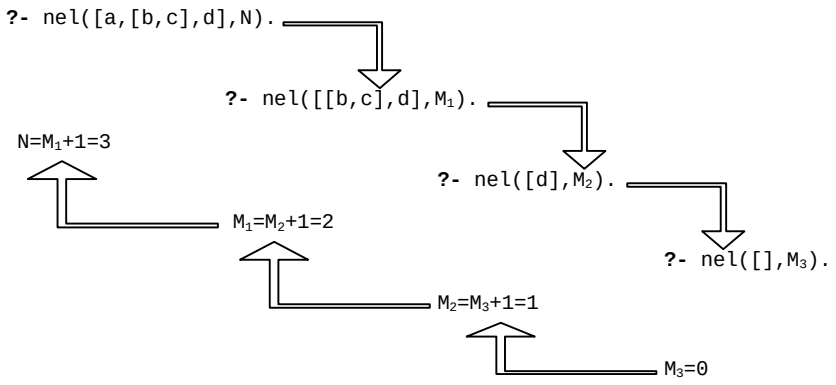
Procedimiento que calcula el número de elementos de una lista:

- Condición de parada: lista vacía
  ```
  nel([],0).
  ```
- Evolución de los parámetros: lista con un elemento menos
  ```
  nel([_|Y],N) :- nel(Y,M), N is M+1.
  ```

En la recursión encontramos dos partes:

- la primera parte en la que descendemos y construimos el árbol hasta encontrar el valor que unifica con la condición de parada
- una segunda parte en la que ascendemos por el árbol asignando valores a las variables que teníamos pendientes en las sucesivas llamadas.

Ejemplo:




```
?- trace,nel([a,[b,c],d],N),notrace.
   Call:  (  6) nel([a, [b, c], d], _G309) ? creep
   Call:  (  7) nel([[b, c], d], _L104) ? creep
   Call:  (  8) nel([d], _L117) ? creep
   Call:  (  9) nel([], _L130) ? creep
```

© 1996-2001 Faraón Llorens

```
   Exit:  (  9) nel([], 0) ? creep
 ^ Call:  (  9) _L117 is 0 + 1 ? creep
 ^ Exit:  (  9) 1 is 0 + 1 ? creep
   Exit:  (  8) nel([d], 1) ? creep
 ^ Call:  (  8) _L104 is 1 + 1 ? creep
 ^ Exit:  (  8) 2 is 1 + 1 ? creep
   Exit:  (  7) nel([[b, c], d], 2) ? creep
 ^ Call:  (  7) _G309 is 2 + 1 ? creep
 ^ Exit:  (  7) 3 is 2 + 1 ? creep
   Exit:  (  6) nel([a, [b, c], d], 3) ? creep

N = 3
Yes

?- trace(nel),nel([a,[b,c],d],N).
        nel/2: call redo exit fail
T Call:  (  7) nel([a, [b, c], d], _G292)
T Call:  (  8) nel([[b, c], d], _L241)
T Call:  (  9) nel([d], _L254)
T Call:  ( 10) nel([], _L267)
T Exit:  ( 10) nel([], 0)
T Exit:  (  9) nel([d], 1)
T Exit:  (  8) nel([[b, c], d], 2)
T Exit:  (  7) nel([a, [b, c], d], 3)

N = 3
Yes
```

Deberemos tener cuidado de no escribir *recursiones circulares* (entrada en un bucle que no terminaría nunca) y de evitar la *recursión a izquierdas* (cuando una regla llama a un objetivo que es esencialmente equivalente al objetivo original), que causarían que Prolog no terminara nunca.

Recursión circular:

```
padre(X,Y) :- hijo(Y,X).
hijo(A,B) :- padre(B,A).
```

Recursión a izquierdas:

```
persona(X) :- persona(Y), madre(X,Y).
persona(adán).
```

6.2. UNIFICACIÓN

La **unificación** («matching») aplicada junto con la *regla de resolución* es lo que nos permite obtener respuestas a las preguntas formuladas a un programa lógico. La unificación constituye uno de los mecanismos esenciales de Prolog, y consiste en buscar instancias comunes a dos átomos, uno de los cuales está en la cabeza de una cláusula y el otro en el cuerpo de otra cláusula. Prolog intentará hacerlos coincidir (*unificarlos*) mediante las siguientes reglas:

- Una variable puede instanciarse con cualquier tipo de término, y naturalmente con otra variable. Si *X* es una variable no instanciada, y si *Y* está instanciada a

© 1996-2001 Faraón Llorens

cualquier término, entonces *X* e *Y* son iguales, y *X* quedará *instanciada* a lo que valga *Y*. Si ambas están no instanciadas, el objetivo se satisface y las dos variables quedan *compartidas* (cuando una de ellas quede instanciada también lo hará la otra).

- Los números y los átomos sólo serán iguales a sí mismos. Evidentemente, también se pueden instanciar con una variable.

- Dos estructuras son iguales si tienen el mismo nombre y el mismo número de argumentos, y todos y cada uno de estos argumentos son unificables.

El predicado de igualdad (=) es un operador infijo que intentará unificar ambas expresiones. Un objetivo con el predicado no igual (\=) se satisface si el = fracasa, y fracasa si el = se satisface.

Ejemplos:

```
?- X=juan, X=Y.
X=juan, Y=juan

?- X=Y, X=juan.
X=juan, Y=juan

?- juan=juan.
yes

?- juan=pepe.
no

?- 1024=1024.
yes

?-
amigo(pepe,juan)=amigo(pepe,X).
X=juan

?- amigo(pepe,juan)=Y.
Y=amigo(pepe,juan)

?- 9 \= 8.
yes

?- letra(C)=palabra(C).
no

?- 'juan'=juan.
yes

?- “juan”=juan.
no

?- f(X,Y)=f(A).
no
```

6.3. REEVALUACIÓN

La **reevaluación** («backtracking») consiste en volver a mirar lo que se ha hecho e intentar resatisfacer los objetivos buscando una forma alternativa de hacerlo. Si se quieren obtener más de una respuesta a una pregunta dada, puede iniciarse la reevaluación pulsando la tecla «*y*» cuando Prolog acaba de dar una solución y pregunta «More (y/n) ?», con lo que se pone en marcha el proceso de *generación de soluciones múltiples*.

Vamos a ver dos conceptos ya utilizados y relacionados directamente con la reevaluación:

- *Satisfacer* un objetivo: cuando Prolog intenta satisfacer un objetivo, busca en la Base de Conocimientos desde su comienzo, y:

© 1996-2001 Faraón Llorens

- si encuentra un hecho o la cabeza de una regla que pueda unificar con el objetivo, marca el lugar en la base de conocimientos e instancia todas las variables previamente no instanciadas que coincidan. Si es una regla lo encontrado, intentará satisfacer los subobjetivos del cuerpo de dicha regla.

- si no encuentra ningún hecho o cabeza de regla que unifique, el objetivo ha *fallado*, e intentaremos resatisfacer el objetivo anterior.

- *Resatisfacer* un objetivo: Prolog intentará resatisfacer cada uno de los subobjetivos por orden inverso, para ello intentará encontrar una cláusula alternativa para el objetivo, en cuyo caso, se dejan sin instanciar todas las variables instanciadas al elegir la cláusula previa. La búsqueda la empezamos desde donde habíamos dejado el marcador de posición del objetivo.

Ejemplo (`ej05.pl`):

```
/* animal(Anim) <- Anim es un animal */
animal(mono).
animal(gallina).
animal(araña).
animal(mosca).
animal(cocodrilo).

/* gusta(X,Y) <- a X le gusta Y */
gusta(mono,banana).
gusta(araña,mosca).
Gusta(alumno,logica).
gusta(araña,hormiga).
gusta(cocodrilo,X) :- animal(X).
gusta(mosca,espejo).

/* regalo(X,Y) <- Y es un buen regalo para X */
regalo(X,Y) :- animal(X), gusta(X,Y).

?- regalo(X,Y).
X=mono, Y=banana
More (y/n)? y

X=araña, Y=mosca
More (y/n)? y

X=araña, Y=hormiga
More (y/n)? y

X=mosca, Y=espejo
More (y/n)? y

X=cocodrilo, Y=mono
More (y/n)? y

X=cocodrilo, Y=araña
More (y/n)? y

X=cocodrilo, Y=mosca
More (y/n)? y

X=cocodrilo, Y=cocodrilo
```


© 1996-2001 Faraón Llorens

```
More (y/n)? y
no
```

Ejercicio:

Alterar el orden de los hechos de la base de conocimientos anterior, añadir nuevos hechos y/o reglas y estudiar como se van generando las sucesivas respuestas. Intenta entender por qué Prolog genera estas respuestas y en este determinado orden (estrategias de resolución). Te puede ayudar si activas la opción de traza mediante el predicado predefinido trace, lo que te permitirá ir viendo los pasos que sigue prolog para obtener las distintas respuestas.

6.4. EL CORTE

El **corte** permite decirle a Prolog cuales son las opciones previas que no hace falta que vuelva a considerar en un posible proceso de reevaluación.

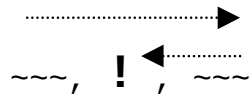
```
!
```

Es un mecanismo muy delicado, y que puede marcar la diferencia entre un programa que funcione y uno que no funcione. Su utilidad vienen dada por:

- *Optimización del tiempo de ejecución*: no malgastando tiempo intentando satisfacer objetivos que de antemano sabemos que nunca contribuirán a una solución.

- *Optimización de memoria*: al no tener que registrar puntos de reevaluación para un examen posterior.

El corte se representa por el objetivo "!" que se satisface inmediatamente y no puede resatisfacerse de nuevo. Es decir, se convierte en una valla que impide que la reevaluación la atraviese en su vuelta hacia atrás, convirtiendo en inaccesibles todos los marcadores de las metas que se encuentran a su izquierda.



Se trata de una impureza en lo referente al control. Esto es, el control en Prolog es fijo y viene dado de forma automática por la implementación del lenguaje: recursión, unificación y reevaluación automática. De esta manera Prolog nos proporciona la ventaja de centrarnos en los aspectos lógicos de la solución, sin necesidad de considerar aspectos del control. Además, el corte puede conducirnos a programas difíciles de leer y validar.

Los usos comunes del corte se pueden dividir en tres áreas según la intención con la que lo hayamos colocado:

© 1996-2001 Faraón Llorens

1).- CONFIRMACIÓN DE LA ELECCIÓN DE UNA REGLA: se informa a Prolog que va por buen camino, por tanto, si ha pasado el corte ha encontrado la regla correcta para resolver el objetivo. Si las metas siguientes fracasan, deberá fracasar el objetivo. El uso del corte con esta finalidad es semejante a simular un "if-then-else" (si *b* entonces *c* sino *d*):

```
a :- b, !, c.
a :- d.
```

que expresado sin utilizar el corte quedaría:

```
a :- b, c.
a :- not(b), d.
```

en la cual puede que Prolog acabe intentando satisfacer *b* dos veces. Así, debemos sopesar las ventajas de un programa claro sin cortes frente a uno que sea más rápido.

Ejemplo (`ej06.pl`):

```
/* sumatorio(Num,Sum) <- Sum es el sumatorio desde 1 hasta Num */
sumatorio(1,1) :- !.
sumatorio(N,S) :- N1 is N-1,
                  sumatorio(N1,S1),
                  S is N+S1.
```

Ejercicio:

Estudia que ocurriría si no hubiésemos puesto el corte en la primera cláusula.

2).- ADVERTENCIA DE QUE SE VA POR MAL CAMINO: se informa a Prolog que haga fracasar el objetivo sin intentar encontrar soluciones alternativas. Se utiliza en combinación con el predicado de fallo *fail*, el cual siempre fracasa como objetivo, lo que hace que se desencadene el proceso de reevaluación.

3).- TERMINACIÓN DE LA GENERACIÓN DE SOLUCIONES MÚLTIPLES: se informa a Prolog que deseamos finalizar la generación de soluciones alternativas mediante reevaluación.

Ejemplo (`ej06.pl`):

```
/* natural(Num) <- Num es un numero perteneciente a los Naturales */
natural(0).
natural(X) :- natural(Y),
              X is Y+1.

/* diventera(Dividendo,Divisor,Cociente) <- Cociente es el resultado
              de la division entera de Dividendo entre Divisor */
diventera(A,B,C) :- natural(C),
                    Y1 is C*B,
                    Y2 is (C+1)*B,
                    Y1 =< A, Y2 > A, !.
```


© 1996-2001 Faraón Llorens

Este programa Prolog va generando números naturales hasta encontrar un *C* que cumpla la condición:

$$B*C =< A < B*(C+1)$$

siendo *A* el dividendo y *B* el divisor de la división. La regla *natural* va generando infinitos números naturales, pero sólo uno de ellos es el que nos interesa y será el que pase la condición. Así, el corte al final de la regla impide que en un posible proceso de reevaluación entremos en un bucle infinito: hemos encontrado la única solución y no hay ninguna razón para volver a buscar otra.

Discusión del "corte"

Mientras que un corte puede ser inofensivo, o incluso beneficioso, cuando se utiliza una regla de una forma dada, el mismo corte puede provocar un comportamiento extraño si la regla se utiliza de otra forma. Así, el siguiente predicado (`ej06.pl`):

```
/* concatena(L1,L2,L3) <- concatenacion de las listas L1 y L2
                       dando lugar a la lista L3           */
concatena([],L,L) :- !.
concatena([X|L1],L2,[X|L3]) :- concatena(L1,L2,L3).
```

cuando consideramos objetivos de la forma

```
?- concatena([a,b,c],[d,e],X).
X=[a,b,c,d,e]
?- concatena([a,b,c],X,[a,b,c,d,e]).
X=[d,e]
```

el corte resulta muy adecuado, pero si tenemos el objetivo

```
?- concatena(X,Y,[a,b,c]).
X=[], Y=[a,b,c]
More (y/n)? y
no
```

ante la petición de nuevas soluciones, la respuesta es “no”, aunque de hecho existan otras posibles soluciones a la pregunta.

La conclusión sería que si se introducen cortes para obtener un comportamiento correcto cuando los objetivos son de una forma, no hay garantía de que los resultados sean razonables si empiezan a aparecer objetivos de otra forma. Sólo es posible utilizar el corte de una forma fiable si se tiene una visión clara de cómo se van a utilizar las reglas.

Ejemplos de uso del corte (`ej06.pl`):

```
/* borrar(Elem,L1,L2) <- L2 es la lista resultado de borrar todas las
                         ocurrencias del elemento Elem de la lista L1 */
borrar(_,[],[]).
borrar(E,[E|L1],L2) :- !, borrar(E,L1,L2).
borrar(E,[A|L1],[A|L2]) :- borrar(E,L1,L2).

/* sust(E1,E2,L1,L2) <- L2 es la lista resultado de sustituir en lista
                        L1 todas las ocurrencias del elemento E1 por E2 */
sust(_,_,[],[]).
```

© 1996-2001 Faraón Llorens

```
sust(E1,E2,[E1|L1],[E2|L2]) :- !, sust(E1,E2,L1,L2).
sust(E1,E2,[Y|L1],[Y|L2]) :- sust(E1,E2,L1,L2).

/* union(L1,L2,L3) <- L3 es la lista-conjunto unión de las
                    listas-conjuntos L1 y L2            */
union([],L,L).
union([X|L1],L2,L3) :- miembro(X,L2), !,
                       union(L1,L2,L3).
union([X|L1],L2,[X|L3]) :- union(L1,L2,L3).

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

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

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

6.5. PREDICADOS DE CONTROL

Existen una serie de predicados predefinidos que ayudan cuando queremos utilizar estructuras de control. Vamos a ver alguno de ellos.

`fail`

Siempre falla.

`true`

Siempre tiene éxito

`repeat`

Permite simular bucles junto con la combinación corte-fail.

Ejemplo (`ej08.pl`):

```
 /*-------------------------------------------------------------*/
 /*               PROGRAMA  PRINCIPAL                           */

menu :- cabecera,
        repeat,
          leer(Formula),
          hacer(Formula).

hacer(X) :- integer(X), X=0.
hacer('A') :- cabecera, ayuda, !, fail.
hacer('a') :- cabecera, ayuda, !, fail.
hacer(Formula) :- fbf(Formula),
                  formaNormal(Formula,FNC,FND),
                  escribirFormasNormales(FNC,FND),!, fail.
```


© 1996-2001 Faraón Llorens

```
 /*                 Fin Programa Principal                    */
 /*-----------------------------------------------------------*/
```

+Condicion -> +Accion

Permite simular la estructura condicional: *Si-Entonces* y *Si-Entonces-Sino*.

Ejemplo :

```
ejemplo :- write('Escribe s o n'),nl,
           read(Respuesta),
           ( (Respuesta='s') ->
                write('Has respondido afirmativamente')
             ;
                write('Has respondido negativamente')
           ).
```

© 1996-2001 Faraón Llorens

7. PREDICADOS DE ENTRADA Y SALIDA

Vamos a introducir una nueva ampliación sobre Prolog que no tiene nada que ver con la lógica. Hasta ahora, el único modo de comunicar información *a* y *desde* Prolog es a través del proceso de unificación entre nuestras preguntas y las cláusulas del programa. Pero puede que estemos interesados en una mayor interacción entre programa y usuario, aceptando entradas del usuario y presentando salidas al mismo. Así, aparecen los predicados de Entrada/Salida, que para mantener la consistencia del sistema cumplen:

- se evalúan siempre a verdad
- nunca se pueden resatisfacer: la reevaluación continua hacia la izquierda
- tiene un *efecto lateral* (efecto no lógico durante su ejecución): entrada o salida de un carácter, término, ...

7.1. LECTURA Y ESCRITURA DE TÉRMINOS

`write(+Termino)`

Si *Termino* está instanciada a un término, lo saca por pantalla. Si *Termino* no está instanciada, se sacará por pantalla una variable numerada de forma única (por ejemplo _69).

`nl`

Nueva Línea: la salida posterior se presenta en la siguiente línea de la pantalla.

`write_ln(+Termino)`

Equivalente a `write(+Termino),nl`

`writef(+Formato,+Argumentos)`

Escritura con formato

`tab(+X)`

Desplaza el cursor a la derecha *X* espacios. *X* debe estar instanciada a un entero (una expresión evaluada como un entero positivo).

© 1996-2001 Faraón Llorens

`display(+Termino)`

Se comporta como “write”, excepto que pasa por alto cualquier declaración de operadores que se haya hecho.

`read(-Termino)`

Lee el siguiente término que se teclee desde el ordenador. Debe ir seguido de un punto “.” y un “retorno de carro”. Instancia *Termino* al término leído, si *Termino* no estaba instanciada. Si *Termino* si que está instanciada, los comparará y se satisfará o fracasará dependiendo del éxito de la comparación.

Ejemplos (`ej07.pl`):

```
/* ibl(L) <- imprime la lista L de forma bonita, es decir
             saca por pantala los elementosa de la lista
             separados por un espacio y terminado en salto
             de línea                                      */
ibl([]) :- nl.
ibl([X|Y]) :-    write(X),
            tab(1),
            ibl(Y).

/* Diferencia entre write, display e ibl */
?- write(8+5*6), nl, display(8+5*6).
8+5*6
+(8,*(5,6))
yes

?- write(8*5+6), nl, display(8*5+6).
8*5+6
+(*(8,5),6)
yes

?- X=[esto,es,una,lista],write(X),nl,display(X),nl,ibl(X).
[esto,es,una,lista]
.(esto,.(es,.(una,.(lista,[]))))
esto es una lista
yes

?- writef('Atomo = %w\nNúmero = %w\n',[pepe,4.5]).
Atomo = pepe
Número = 4.5
yes
```

7.2. LECTURA Y ESCRITURA DE CARACTERES

El carácter es la unidad más pequeña que se puede leer y escribir. Prolog trata a los caracteres en forma de enteros correspondientes a su código ASCII.

© 1996-2001 Faraón Llorens

`put(+Caracter)`

Si *Caracter* está instanciada a un entero entre 0..255, saca por pantalla el carácter correspondiente a ese código ASCII. Si *Caracter* no está instanciada o no es entero, error.

`get(-Caracter)`

Lee caracteres desde el teclado, instanciando *Caracter* al primer carácter imprimible que se teclee. Si *Caracter* ya está instanciada, los comparará satisfaciéndose o fracasando.

`get0(-Caracter)`

Igual que el anterior, sin importarle el tipo de carácter tecleado.

`skip(+Caracter)`

Lee del teclado hasta el carácter *Caracter* o el final del fichero.

Ejemplos (`ej07.pl`):

```
/* escribe_cadena(L) <- escribe en pantalla la lista L de
                        códigos ASCII en forma de cadena de
                        caracteres                        */
escribe_cadena([]).
escribe_cadena([X|Y]) :- put(X),
                         escribe_cadena(Y).

?- escribe_cadena([80,114,111,108,111,103]).
Prolog
yes

?- put(80),put(114),put(111),put(108),put(111),put(103).
Prolog
yes
```

7.3. LECTURA Y ESCRITURA EN FICHEROS

Existe un fichero predefinido llamado *user*. Al leer de este fichero se hace que la información de entrada venga desde el teclado, y al escribir, se hace que los caracteres aparezcan en la pantalla. Este el modo normal de funcionamiento. Pero pueden escribirse términos y caracteres sobre ficheros utilizando los mismos predicados que se acaban de ver. La única diferencia es que cuando queramos escribir o leer en un fichero debemos cambiar el *canal de salida activo* o el *canal de entrada activo*, respectivamente.

© 1996-2001 Faraón Llorens

Posiblemente queramos leer y escribir sobre ficheros almacenados en discos magnéticos. Cada uno de estos tendrá un *nombre de fichero* que utilizamos para identificarlo. En Prolog los nombres de ficheros se representan como átomos. Los ficheros tienen una longitud determinada, es decir, contienen un cierto número de caracteres. Al final del fichero, hay una marca especial de *fin de fichero.* Cuando la entrada viene del teclado, puede generarse un fin de fichero tecleando el carácter de control ASCII 26 o control-Z. Se pueden tener abiertos varios ficheros, si conmutamos el canal de salida activo sin cerrarlos (`told`), permitiéndonos escribir sobre ellos en varios momentos diferentes, sin destruir lo escrito previamente.

`tell(+NomFichero)`

Si *NomFichero* está instanciada al nombre de un fichero, cambia el canal de salida activo. Crea un nuevo fichero con ese nombre. Si *NomFichero* no está instanciada o no es un nombre de fichero, producirá un error.

`telling(?NomFichero)`

Si *NomFichero* no está instanciada, la instanciará al nombre del fichero que es el canal de salida activo. Si *NomFichero* está instanciada, se satisface si es el nombre del fichero actual de salida.

`told`

Cierra el fichero para escritura, y dirige la salida hacia la pantalla.

`see(+NomFichero)`

Si *NomFichero* está instanciada al nombre de un fichero, cambia el canal de entrada activo al fichero especificado. La primera vez que se satisface, el fichero pasa a estar abierto y empezamos al comienzo del fichero. Si *NomFichero* no está instanciada o no es un nombre de fichero, producirá un error.

`seeing(?NomFichero)`

Si *NomFichero* no está instanciada, la instanciará al nombre del fichero que es el canal de entrada activo. Si *NomFichero* está instanciada, se satisface si es el nombre del fichero actual de entrada.

`seen`

Cierra el fichero para lectura, y dirige la entrada hacia el teclado.

© 1996-2001 Faraón Llorens

Ejemplos:

```
/* Secuencia normal de escritura */
... , tell(fichero), write(X), told, ...

/* Conmutación de ficheros */
... , tell(fichero), write(A), tell(user), write(B),
      tell(fichero), write(C), told.

/* Secuencia normal de lectura */
... , see(fichero), read(X), seen, ...
```


© 1996-2001 Faraón Llorens

8. MODIFICACIÓN DE LA BASE DE CONOCIMIENTOS

Vamos a estudiar una serie de predicados predefinidos en Prolog que nos permitirán manipular la Base de Conocimientos consultando, añadiendo, eliminando o modificando cláusulas.

8.1. ADICIÓN DE BASES DE CONOCIMIENTO EXTERNAS

En Prolog los ficheros se utilizan principalmente para almacenar programas y no perderlos al apagar el ordenador. Si queremos que Prolog lea nuevas cláusulas de un fichero que hemos preparado previamente, podemos utilizar los predicados **consult** y **reconsult**. Esto será conveniente cuando tengamos que trabajar con programas de tamaño considerable y no queramos teclear cada vez todas las cláusulas. Así, podremos crear el programa con un editor de textos y guardarlo para posteriores usos, con la ventaja añadida de que lo podremos modificar y ampliar en cualquier momento.

```
consult(+Fichero)
```

El predicado predefinido *consult* añade las cláusulas existentes en el fichero de nombre *Fichero* a la base de conocimientos de Prolog, al final de las ya existentes, sin destruir las anteriores. Como argumento pasamos un átomo con el nombre del fichero del que queremos leer las cláusulas. Recordemos que un átomo está formado por letras, dígitos y el carácter subrayado. Si queremos que contenga otros caracteres (los dos puntos «:» para la unidad, la barra invertida «\» para el directorio o el punto «.» para la extensión[6]) deberemos encerrarlo entre comillas simples.

```
reconsult(+Fichero)
```

El predicado predefinido *reconsult* actúa como el consult excepto que, de existir cláusulas para un mismo predicado (nombre y aridad), las nuevas sustituyen a las existentes[7]. Se suele utilizar para la corrección de errores mientras se está realizando el programa, ya que de esta manera la nueva versión sustituye a la anterior errónea. También es interesante cuando queremos asegurarnos que se utilizan las cláusulas que vamos a leer, y no otras posibles con el mismo nombre que tuviese el sistema de alguna consulta anterior.

[6] Por defecto, SWI-Prolog asume la extensión «.pl» y Prolog-2 toma la extensión «.pro». Pero esto dependerá del sistema Prolog utilizado y se puede personalizar.

[7] SWI-Prolog no tiene un predicado resonsult, ya que al consultar un fichero ya almacenado, automáticamente es reconsultado

© 1996-2001 Faraón Llorens

La mayor parte de las implementaciones permiten una notación especial que permite consultar una lista de ficheros, uno tras otro: la notación en forma de lista. Consiste en poner los nombres de los ficheros en una lista y darla como objetivo a satisfacer. Si se quiere consultar un fichero, se pone el nombre en la lista tal cual, mientras que si se quiere reconsultar se pone el nombre precedido de un signo "-".

Ejemplos:

```
?- consult(fichero).
fichero consulted

?- reconsult(fichero).
fichero reconsulted

?- [fichero1,-fichero2,-'prueba','a:prueba2'].
fichero1 consulted
fichero2 reconsulted
prueba reconsulted
a:prueba2 consulted

?- consult('prac1.pro').
prac1.pro consulted

?- reconsult('a:prac1').
a:prac1 reconsulted

?- ['lpo\prac1'].
lpo\prac1 consulted

?- [-'a:\lpo\prac1.pro'].
a:\lpo\prac1.pro reconsulted
```

8.2. MANIPULACIÓN DE LA BASE DE CONOCIMIENTOS

Veamos ahora los predicados predefinidos que podemos utilizar para ver (**listing**), obtener (**clause**), añadir (**assert**) o quitar (**retract** y **abolish**) cláusulas de nuestra base de conocimientos:

```
listing
```

```
listing(+Predicado)
```

Todas las cláusulas que tienen como predicado el átomo al que está instanciada la variable *Predicado* son mostradas por el fichero de salida activo (por defecto la pantalla). El predicado de aridad 0 listing (sin parámetro) muestra todas las cláusulas de la Base de Conocimientos.

Ejemplos :

```
?- listing(asignatura).
/* asignatura/1 */
asignatura(logica) .
```


© 1996-2001 Faraón Llorens

```
asignatura(programacion) .
asignatura(matematicas) .
/* asignatura/2 */
asignatura(logica,lunes) .
asignatura(programacion,martes) .
asignatura(matematicas,miercoles) .
yes

?- listing(asignatura/2).
asignatura(logica,lunes) .
asignatura(programacion,martes) .
asignatura(matematicas,miercoles) .
yes
```

`clause(?Cabeza,?Cuerpo)`

Se hace coincidir *Cabeza* con la cabeza y *Cuerpo* con el cuerpo de una cláusula existente en la Base de Conocimientos. Por lo menos *Cabeza* debe estar instanciada. Un *hecho* es considerado como una cláusula cuyo cuerpo es **`true`**.

`assert(+Clausula)`

`asserta(+Clausula)`

`assertz(+Clausula)`

Estos predicados permiten añadir nuevas cláusulas a la base de conocimientos. El predicado a*sserta* la añade al principio (letra «a») y *assertz* la añade al final (letra «z») de cualquier otra cláusula del mismo tipo que hubiese en la base de conocimientos. En todos los casos, *Clausula* debe estar previamente instanciada a una cláusula. Dicha cláusula queda incorporada a la base de conocimientos y no se pierde aunque se haga reevaluación.

Ejemplos :

```
?- asignatura(X)
no

?- assert(asignatura(logica)).
yes

?- asserta(asignatura(matematicas)).
yes

?- assertz(asignatura(programacion)).
yes

?- asignatura(X).
X = matematicas
More (y/n)? y
X = logica
More (y/n)? y
X = programacion
More (y/n)? y
```

© 1996-2001 Faraón Llorens

```
no
```

`retract(+Clausula)`

`retractall(+Clausula)`

Este predicado nos permite eliminar una cláusula de nuestra base de conocimientos. Para ello, *Clausula* debe estar instanciada y se quitará la primera cláusula de la base de conocimientos que empareje con ella. Si se resatisface el objetivo, se irán eliminando, sucesivamente, las cláusulas que coincidan. Con *retractall* se eliminarán todas.

Ejemplos :

```
?- listing(asignatura).
/* asignatura/1 */
asignatura(matematicas) .
asignatura(logica) .
asignatura(programacion) .
yes

?- retract(asignatura(logica)).
yes

?- listing(asignatura).
/* asignatura/1 */
asignatura(matematicas) .
asignatura(programacion) .
yes

?- retract(asignatura(X)).
X = matematicas
More (y/n)? n
yes

?- listing(asignatura).
/* asignatura/1 */
asignatura(programacion) .
yes
```

`abolish(+Predicado/Aridad)`

`abolish(+Predicado,+Aridad)`

Retira de la Base de Conocimientos todas las cláusulas del predicado *Predicado*. Debe estar identificado completamente el predicado : nombre y aridad.

Ejemplos :

```
?- listing(asignatura).
/* asignatura/1 */
asignatura(logica) .
asignatura(programacion) .
```


© 1996-2001 Faraón Llorens

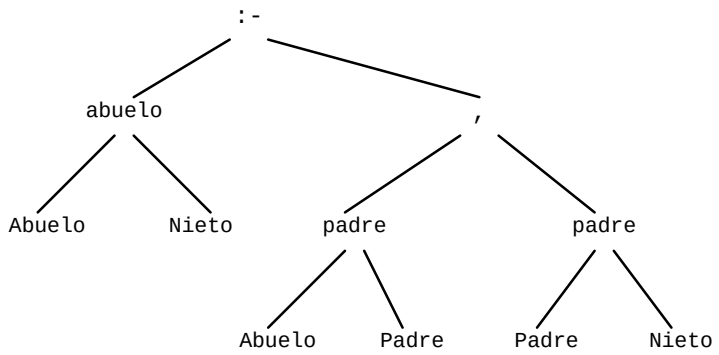
```
asignatura(matematicas) .
/* asignatura/2 */
asignatura(logica,lunes) .
asignatura(programacion,martes) .
asignatura(matematicas,miercoles) .
yes

?- abolish(asignatura/1).
yes

?- listing(asignatura).
/* asignatura/2 */
asignatura(logica,lunes) .
asignatura(programacion,martes) .
asignatura(matematicas,miercoles) .
yes

?- abolish(asignatura,2).
yes

?- listing(asignatura).
yes
```

8.3. COMPONENTES DE ESTRUCTURAS

Como ya hemos comentado anteriormente Prolog considera los hechos y las reglas, e incluso los mismos programas, como estructuras. Veamos ahora una de sus ventajas : el considerar a las propias cláusulas como estructuras nos permite manipularlas (construir, modificar, eliminar, añadir, ...) con los predicados de construcción y acceso a los componentes de las estructuras **functor**, **arg**, **=..** y **name**.

Así, la siguiente regla la considera Prolog constituida como la estructura de la figura :

```
abuelo(Abuelo, Nieto) :- padre(Abuelo, Padre), padre(Padre, Nieto).
```



`functor(?Estructura,?Nombre,?Aridad)`

Estructura es una estructura con nombre *Nombre* y aridad (número de argumentos) *Aridad*. Se puede utilizar básicamente de dos maneras :

© 1996-2001 Faraón Llorens

- teniendo *Estructura* instanciada, y por tanto hace corresponder a *Nombre* con el nombre de dicha estructura y *Aridad* con un entero que indica el número de argumentos.
- cuando *Nombre* y *Aridad* están instanciados, por lo que *Estructura* queda instanciada a una estructura con el nombre y el número de argumentos dados.

Ejemplos :

```
?- functor(logica,F,N).
F = logica, N = 0

?- functor(3+2,F,N).
F = +, N = 2

?- functor([a,b,c],F,N).
F = . , N = 2

?- functor(E,libro,2).
E = libro(_,_)

?- functor(3+2+1,+,3).
no

?- functor(3+2+1,+,2).
yes
```

arg(?Posicion,?Estructura,?Argumento)

El argumento de la estructura *Estructura* que ocupa la posición *Posicion* es *Argumento*. Se debe utilizar con, por lo menos, los dos primeros argumentos instanciados, y sirve para acceder a un determinado argumento de una estructura.

Ejemplos :

```
?- arg(2,1+2+3,Arg).
Arg = 1+2

?- arg(1,[a,b,c],Arg).
Arg = a

?- arg(2,p(x,f(y)),Arg).
Arg = f(y)

?- arg(1,asignatura(logica,lunes),logica).
yes
```

?Estructura =.. ?Lista

El predicado *univ* (=..) nos permite acceder a argumentos de estructuras de las que no conocemos de antemano el número de componentes. Si *Estructura* está instanciada, *Lista* será una lista cuyo primera componente (cabeza) es el functor de la estructura y el resto (cola) los distintos argumentos de dicha estructura. Si *Lista* está instanciada, creará una estructura cuyo functor será la cabeza de la lista, y cuyos argumentos serán los distintos elementos de la lista.

© 1996-2001 Faraón Llorens

Ejemplos :

```
?- p(x,f(y),z) =.. L.
L = [p,x,f(y),z]

?- E =.. [+,2,3].
E = 2+3

?- Clausula =.. [asignatura,logica,lunes,11].
Clausula = asignatura(logica,lunes,11)
```

name(?Atomo,?Lista)

Atomo es un átomo formado por los caracteres de la lista *Lista* (códigos ASCII). Puede, tanto crear un átomo con los caracteres de la lista de códigos ASCII, como crear la lista de caracteres correspondientes al átomo.

Ejemplos :

```
?- name(logica,L).
L = "logica"

?- name(Atomo,[80,114,111,108,111,103]).
Prolog

?- name(hola,"hola").
yes
```

© 1996-2001 Faraón Llorens

9. DEPURACIÓN DE PROGRAMAS PROLOG

Por último, veremos una serie de predicados predefinidos para supervisar el funcionamiento de los programas Prolog durante su ejecución que nos permitirán la depuración y corrección de nuestros programas, ya que es raro que nuestros programas funcionen a la primera. Muchos de estos errores son fácilmente identificables y por tanto se pueden corregir sin dificultad. En cambio hay tipos de errores difíciles de detectar, y siempre es de agradecer cualquier tipo de ayuda que nos pueda prestar el interprete. Estos predicados son : **trace**, **notrace**, **spy**, **nospy**, **debugging**, **debug** y **nodebug**.

Los errores en programación pueden ser clasificados en dos grandes grupos : errores *sintácticos* y errores *semánticos*.

- Los errores sintácticos hacen referencia a la estructura gramatical de las sentencias (falta del punto al final de un hecho o una regla, no cerrar comillas en una cadena, ...), y Prolog nos informa de ellos inmediatamente que consultamos el programa. La gran mayoría de los errores cometidos son de este tipo, pero sin embargo son fáciles de localizar y corregir.

  Ejemplo :

```
hombre[jorge).
```

- Por otro lado, los errores semánticos se refieren a incorrecciones en el razonamiento o la interpretación. Son, por contra, complicados de descubrir ya que el ordenador no puede saber lo que nosotros pretendemos, y por ello nos proporciona una limitada ayuda. Solamente pueden ser descubiertos siguiendo el proceso de inferencia paso a paso (*traza* de la ejecución). Y aquí es donde entran en juego los predicados predefinidos que veremos a continuación y que nos facilitarán el seguimiento de la traza.

  Ejemplos :

interpretación
```
padre(felix,borja)
```
¿quién es el padre?

razonamiento
```
hermanos(X,Y) :- padres(X,P,M),
                 padres(Y,P,M).
```
¡una persona es hermano de sí mismo !

Pero antes de continuar debemos repasar algunos conceptos ya vistos :

- *Call* : se intenta satisfacer dicho predicado
- *Exit* : el predicado ha tenido éxito y se ha satisfecho
- *Fail* : el objetivo ha fracasado
- *Redo* : se intenta resatisfacer el objetivo al haber fracasado un predicado posterior

© 1996-2001 Faraón Llorens

`trace`

`trace(+Predicado)`

Activa el seguimiento exhaustivo de la traza de la ejecución de las metas posteriores a la consecución de este objetivo o del predicado *Predicado*, viendo los distintos objetivos que intenta satisfacer y el resultado de esta acción. A partir de la consecución de este objetivo, hay una interacción entre el programador y la máquina Prolog, pidiéndonos que confirmemos cada paso. Podemos utilizar las siguientes opciones del *modo traza* :

Opción	Nombre	Descripción
<return>,<espacio>	*avanzar*	continua con la traza
«a»	*abort*	retorna a Prolog
«b»	*break*	inicia un nuevo nivel de Prolog
«c»	*continue*	continua con la traza
«e»	*exit*	abandona el sistema Prolog
«f»	*fail*	hace fallar a ese objetivo
«g»	*goal stack*	muestra la pila de objetivos cumplidos que nos han permitido llegar al objetivo actual
«i»	*ignore*	ignora el actual objetivo (como si se hubiera cumplido)
«l»	*leap*	realiza la pausa solamente en los puntos espía
«n»	*nodebug*	continua ejecución en modo “no debug”
«r»	*retry*	intenta de nuevo satisfacer el objetivo
«s»	*skip*	salta el modo interactivo hasta que se cumple o falla ese objetivo

`notrace`

`notrace(+Objetivo)`

Su efecto es el inverso del predicado anterior, es decir, desactiva la traza. Con el argumento *Objetivo*, intenta satisfacer el objetivo desactivando la traza mientras dure su ejecución

`spy(+Predicado)`

Fija un punto espía en el predicado *Predicado* para poder seguir su funcionamiento. *Predicado* debe ser :

© 1996-2001 Faraón Llorens

- un átomo : se ponen puntos espía en todos los predicados con ese átomo, independientemente del número de argumentos que tengan.
- una estructura (functor/aridad) : sólo establecería puntos espía en aquellos predicados con dicho functor y número de argumentos.
- una lista : situaría puntos espía en todos los lugares especificados por cada uno de los elementos de la lista.

Ejemplos :

```
?- spy(iniciar).
?- spy(iniciar/0).
?- spy(iniciar/1).
?- spy([iniciar, pertenece]).
```

nospy(+Predicado)

nospyall

Elimina los puntos espía especificados en *Predicado* (determinado de manera análoga a la referida en el punto anterior) o retira todos los puntos espía activos en ese momento.

debugging

Permite ver una lista con los puntos espía que se han establecido hasta el momento.

Ejemplo :

```
?- debugging.
There are spypoints set on :
iniciar/0
iniciar/1
pertenece/2
```

debug

nodebug

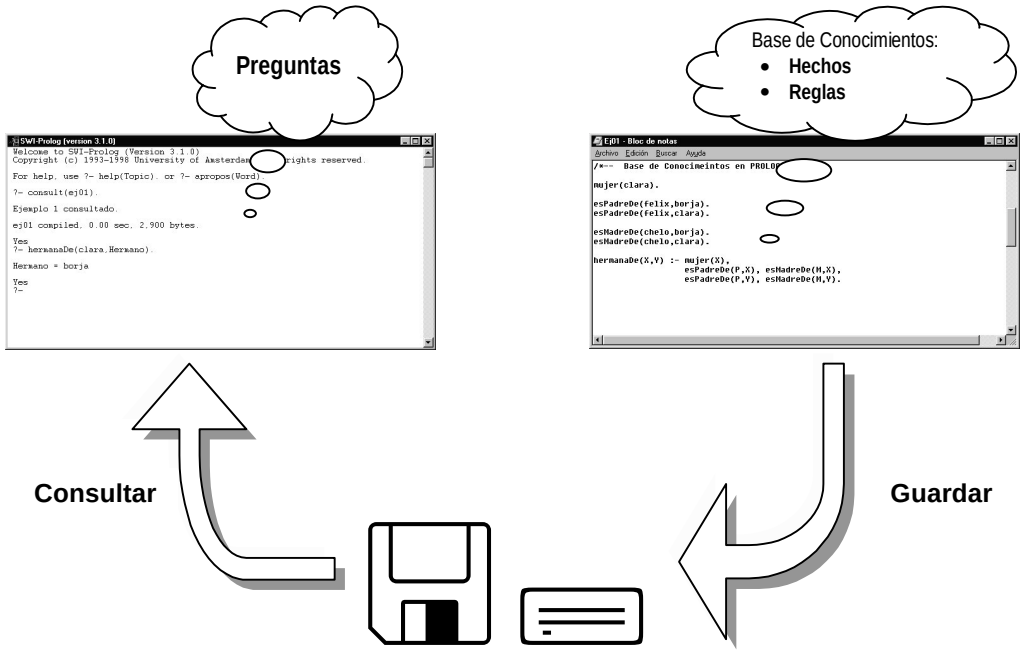
Inicia y finaliza el depurador.

© 1996-2001 Faraón Llorens

10. PROGRAMACIÓN EN PROLOG

10.1. ENTORNO DE TRABAJO

En la siguiente figura podemos ver un esquema de la forma de trabajo en Prolog. Mediante un editor de texto escribiremos nuestra *Base de Conocimientos* compuesta por distintos *hechos* y *reglas* que reflejan nuestro conocimiento acerca del problema a formalizar. Para poder utilizarlo deberemos guardarlo en disco (disco duro o disquete). Posteriormente desde Prolog deberemos consultar el fichero deseado. Una vez consultada la base de conocimientos correspondiente, estaremos en condiciones de hacer las *preguntas* de las que queramos obtener respuesta o que resuelvan el problema. Este ciclo se repetirá continuamente (editar-guardar-consultar-preguntar) hasta que demos por finalizado nuestro trabajo.



10.2. ESTILO DE PROGRAMACIÓN EN PROLOG

Los lenguajes de Programación Lógica, al igual que los lenguajes procedimentales, necesitan de una metodología para construir y mantener programas largos, así como un buen estilo de programación. Prolog ofrece mecanismos de control típicos de cualquier lenguaje imperativo y no puras propiedades o relaciones entre objetos. Dichos mecanismos contribuyen a dar mayor potencia y expresividad al lenguaje, pero violan su naturaleza lógica. Adoptando un estilo apropiado de programación podemos beneficiarnos de esta doble naturaleza de Prolog.

© 1996-2001 Faraón Llorens

- *Metodología de Diseño "top-down"*: descomponemos el problema en subproblemas y los resolvemos. Para ello solucionamos primero los pequeños problemas: *implementación "bottom-up"*. Esto nos va a permitir una mejor depuración («debugged») de nuestros programas, ya que cada pequeño trozo de programa puede ser probado inmediatamente y, si lo necesita, corregido.

- Una vez analizado el problema y separado en trozos, el siguiente paso es decidir como representar y manipular tanto los objetos como las relaciones del problema. Debemos elegir los nombres de los predicados, variables, constantes y estructuras de manera que aporten claridad a nuestros programas (palabras o nombres mnemónicos que estén relacionados con lo que hacen).

- El siguiente paso es asegurarse de que la organización y la sintaxis del programa sea clara y fácilmente legible:

  - Llamamos *procedimiento* al conjunto de cláusulas para un predicado dado. Agruparemos por bloques los procedimientos de un mismo predicado (mismo nombre y mismo número de argumentos). Así, cada cláusula de un procedimiento comenzará en una línea nueva, y dejaremos una línea en blanco entre procedimientos.

  - Si el cuerpo de una regla es lo bastante corto, lo pondremos en una línea; sino, se escriben los objetivos de las conjunciones indentados y en líneas separadas.

  - Primero se escriben los hechos, y luego las reglas.

  - Es recomendable añadir comentarios y utilizar espacios y líneas en blanco que hagan el programa más legible. El listado del programa debe estar "autodocumentado". Debemos incluir para cada procedimiento y antes de las cláusulas el *esquema de la relación*.

  - Los argumentos deben ir precedidos por el signos `+', `-' o `?'. `+' indica que el argumento es de entrada al predicado (debe estar instanciado cuando se hace la llamada), `-' denota que es de salida y `?' indica que puede ser tanto de entrada como de salida

  - Agrupar los términos adecuadamente. Dividir el programa en partes razonablemente autocontenidas (por ejemplo, todos los procedimientos de procesado de listas en un mismo fichero).

  - Evitar el uso excesivo del corte.

  - Cuidar el orden de las cláusulas de un procedimiento. Siempre que sea posible escribiremos las condiciones limite de la recursividad antes de las demás cláusulas. Las cláusulas "recogetodo" tienen que ponerse al final.

© 1996-2001 Faraón Llorens

10.3. INTERPRETACIÓN PROCEDIMENTAL DE LOS PROGRAMAS PROLOG

Podemos realizar una interpretación de los programas Prolog desde un punto de vista más cercano a los lenguajes de programación. Si estamos intentando resolver un determinado problema, tenemos que:

- las preguntas son los *objetivos* o problemas a resolver. Si las preguntas P_i contienen variables x_i, la interpretación que hacemos es que debemos hallar los valores de x_i que resuelvan los problemas P_i.

  ?- P_1 , ..., P_m.

- una regla es interpretada como un *procedimiento* o *método* para resolver un subproblema. Así, ante un problema B que empareje con A con μ=UMG(A,B), el problema original se reduce a resolver los subproblemas $A_1\mu$, ..., $A_m\mu$.

  A :- A_1 , ..., A_m.

- un hecho es interpretado como un procedimiento que resuelve de forma directa un determinado problema.

  A.

De forma general, cuando tenemos que resolver un determinado problema B, buscamos un procedimiento, directo (A.) o no (A :- A_1 , ..., A_m.), que empareje con él (μ=UMG(A,B)). Esto tiene lugar en dos fases:

1. Una primera fase que afecta a las variables del procedimiento (A_1 , ..., A_m), de forma que se transfiere la *entrada* del problema B al procedimiento que trata de resolverlo. Tendremos por tanto un *unificador de entrada* μ_e.
2. Una segunda fase que afecta a las variables del problema a resolver (B), que transfiere la *salida* del procedimiento una vez resuelto. Tendremos por ello un *unificador de salida* μ_s.

Un *programa lógico* se compone de un conjunto de procedimientos (directos o hechos y compuestos o reglas). Su ejecución consistirá en su activación por medio de una pregunta objetivo o problema a resolver. Dicha pregunta irá activando distintos procedimientos para su resolución. Los programas lógicos son más abstractos que los programas convencionales ya que no incorporan control sobre los procedimientos invocados ni el orden en que deben realizarse. Los programas lógicos expresan únicamente la lógica de los métodos que resuelven el problema y son, por tanto, más fáciles de comprender, de verificar y de modificar.

Otra característica destacable de los programas lógicos viene dada por los parámetros de los procedimientos, que en este caso son argumentos de predicados y que por tanto no tienen dirección, es decir, igual pueden servir como parámetros de entrada como de salida dependiendo del contexto en el cual se haya invocado el procedimiento. Así, los parámetros dados en forma de constante en la llamada son considerados como parámetros de entrada y el resto de parámetros son computados como parámetros de salida.

© 1996-2001 Faraón Llorens

Ejemplo:

Sea el siguiente programa en prolog:

```
padresDe(clara,felix,chelo).
padresDe(borja,felix,chelo).
hermanos(X,Y) :- padresDe(X,P,M), padresDe(Y,P,M).
```

Si lanzamos el objetivo

```
?- hermanos(clara,borja).
```

nos contestará que sí. Aquí, los dos argumentos, tanto "clara" como "borja", son considerados como parámetros de entrada. Por otra lado, si lanzamos el objetivo

```
?- hermanos(clara,X).
```

nos responde que si X=borja el problema queda solucionado. En este caso el primer argumento, "clara", es considerado como parámetro de entrada y el segundo, X, es considerado como parámetro de salida, obteniéndose como respuesta si X=borja.

En comparación con los programas convencionales, los programas lógicos son *no deterministas*, en el sentido de que:
- la estrategia de resolución por medio de la cual se van probando procedimientos alternativos no está determinada
- el orden de ejecución dentro de los subobjetivos no está determinada.

Esta propiedad la perdemos al tomar decisiones concretas a la hora de realizar la resolución (SLD-Resolución) y es lo que ocurre con Prolog.

Para conseguir repeticiones de la misma orden, *estructura repetitiva* de los lenguajes de programación, mediante Prolog deberemos utilizar procedimientos *recursivos* (como se ha visto detalladamente al manejar listas).

10.4. VENTAJAS DE PROLOG

Prolog, y en general los lenguajes de programación lógica, tienen las siguientes ventajas frente a los lenguajes clásicos (procedimentales):

- **Expresividad**: un programa (base de conocimiento) escrito en prolog puede ser leído e interpretado intuitivamente. Son, por tanto, más entendibles, manejables y fáciles de mantener.
- **Ejecución y búsqueda incorporada en el lenguaje**: dada una descripción prolog válida de un problema, automáticamente se obtiene cualquier conclusión válida.
- **Modularidad**: cada predicado (procedimiento) puede ser ejecutado, validado y examinado independiente e individualmente. Prolog no tiene variables globales, ni asignación. Cada relación está autocontenida, lo que permite una mayor *modularidad*, *portabilidad* y *reusabilidad* de relaciones entre programas.

© 1996-2001 Faraón Llorens

- **Polimorfismo**: se trata de un lenguaje de programación sin tipos, lo que permite un alto nivel de abstracción e independencia de los datos (objetos).
- **Manejo dinámico y automático de memoria**.

© 1996-2001 Faraón Llorens

11. EJEMPLOS

11.1. FORMAS NORMALES

El siguiente programa comprueba si una fórmula del Cálculo de Proposiciones es una *fórmula bien formada* (fbf) y la transforma a *Forma Normal Conjuntiva* (FNC) y a *Forma Normal Disyuntiva* (FND): (`ej08.pl`)

```
 /*************************************************************/
 /*                                                           */
 /*  Departamento de Ciencia de la Computación e I.A.         */
 /*  Universidad de Alicante                                  */
 /*                                                           */
 /*              LOGICA DE PRIMER ORDEN                       */
 /*                     Prolog                                */
 /*                                                           */
 /*  M. Jesús Castel                          Faraón Llorens  */
 /*                                                           */
 /*   S. O.      :  MS-DOS (Windows)                          */
 /*   Interprete :  SWI-Prolog                                */
 /*   Fichero    :  EJ08.PL                                   */
 /*   Programa   :  " FORMAS NORMALES "                       */
 /*                    Lee  una fórmula del  Cálculo  de      */
 /*                  Proposiciones,  comprueba si  es  una    */
 /*                  Fórmula Bien Formada, y la transforma    */
 /*                  a  Forma Normal Conjuntiva (FNC) y  a    */
 /*                  Forma Normal Disyuntiva (FND).           */
 /*                                                           */
 /*                  Se lanza con :                           */
 /*                    ?- menu.                               */
 /*                                                           */
 /*************************************************************/

 /*         Declaración de conectivas lógicas                 */
?- op(10,fx,no).
?- op(100,yfx,o).
?- op(100,yfx,y).
?- op(200,yfx,imp).
?- op(200,yfx,coimp).

 /*-----------------------------------------------------------*/
 /*               PROGRAMA  PRINCIPAL                         */

menu :- cabecera,
        repeat,
          leer(Formula),
          hacer(Formula).

hacer(X) :- integer(X), X=0.
hacer('A') :- cabecera, ayuda, !, fail.
hacer('a') :- cabecera, ayuda, !, fail.
hacer(Formula) :- fbf(Formula),
                  formaNormal(Formula,FNC,FND),
                  escribirFormasNormales(FNC,FND),!, fail.
```


© 1996-2001 Faraón Llorens

```
 /*                 Fin Programa Principal                    */
 /*-----------------------------------------------------------*/

 /*-----------------------------------------------------------*/
 /*        Determinar si es una Fórmula Bien Formada          */

/* fbf(Form) <- Form es una fórmula bien formada (fbf) del
                Cálculo de Proposiciones                      */
fbf((no P)) :- !,fbf(P).
fbf((P y Q)) :- !,fbf(P),fbf(Q).
fbf((P o Q)) :- !,fbf(P),fbf(Q).
fbf((P imp Q)) :- !,fbf(P),fbf(Q).
fbf((P coimp Q)) :- !,fbf(P),fbf(Q).
fbf(P) :- atom(P),!,proposicion(P).
fbf(_) :- nl,
          write('¡ERROR! : Formula erronea. Vuelve a introducirla.'),
          nl, nl, !, fail.

/* proposicion(Prop) <- Prop es una variable proposicional    */
proposicion(X) :- miembro(X,[p,q,r,s,t,u,v]),!.
proposicion(_) :- nl,
                  write('¡ERROR! no es simbolo de var.proposicional'),
                  nl,nl,!,fail.


 /*-----------------------------------------------------------*/
 /*               Transformar a Forma Normal                  */


/* formaNormal(Form,Fnc,Fnd) <- Fnc es la Forma Normal Conjuntiva
                      y Fnd es la Forma Normal Disyuntiva de Form */
formaNormal(Formula,FNC,FND) :- quitarImplicadores(Formula,F1),
                                normalizarNegadores(F1,F2),
                                exteriorizarConjuntor(F2,FNC),
                                exteriorizarDisyuntor(F2,FND).

/* quitarImplicadores(Form,Form1) <- Form1 es la fbf resultado de
             eliminar implicadores y coimplicadores a la fbf Form */
quitarImplicadores((P imp Q),((no P1) o Q1)) :-
                               !,quitarImplicadores(P,P1),
                               quitarImplicadores(Q,Q1).
quitarImplicadores((P coimp Q),(((no P1) o Q1) y ((no Q1) o P1))) :-
                               !,quitarImplicadores(P,P1),
                               quitarImplicadores(Q,Q1).
quitarImplicadores((P o Q),(P1 o Q1)) :-
                               !,quitarImplicadores(P,P1),
                               quitarImplicadores(Q,Q1).
quitarImplicadores((P y Q),(P1 y Q1)) :-
                               !,quitarImplicadores(P,P1),
                               quitarImplicadores(Q,Q1).
quitarImplicadores((no P),(no P1)) :- !,quitarImplicadores(P,P1).
quitarImplicadores(P,P).

/* normalizarNegadores(Form,Form1) <- Form1 es la fbf resultado de
                           normalizar los negadores a la fbf Form */
normalizarNegadores((no P),P1) :- !,negar(P,P1).
normalizarNegadores((P y Q),(P1 y Q1)) :- !,normalizarNegadores(P,P1),
                                          normalizarNegadores(Q,Q1).
normalizarNegadores((P o Q),(P1 o Q1)) :- !,normalizarNegadores(P,P1),
                                          normalizarNegadores(Q,Q1).
normalizarNegadores(P,P).

/* negar(Form,Form1) <- Form1 es la fbf resultado de negar la fbf
                        Form, teniendo en cuenta la Eliminación del
                        Doble Negador y  las leyes de De Morgan    */
```

© 1996-2001 Faraón Llorens

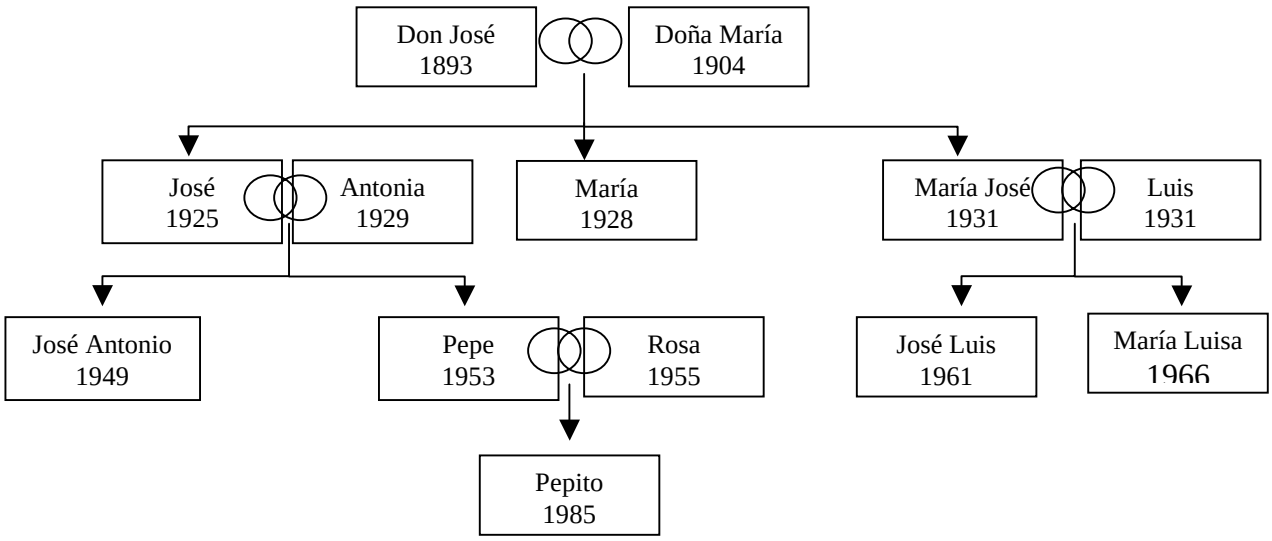
```
negar((no P),P1) :- !,normalizarNegadores(P,P1).
negar((P y Q),(P1 o Q1)) :- !,negar(P,P1), negar(Q,Q1).
negar((P o Q),(P1 y Q1)) :- !,negar(P,P1), negar(Q,Q1).
negar(P,(no P)).

/* exteriorizarConjuntor(Form,Form1) <- Form1 es la fbf resultado
                  de exteriorizar los conjuntores de la fbf Form */
exteriorizarConjuntor((P o Q),R) :- !, exteriorizarConjuntor(P,P1),
                                       exteriorizarConjuntor(Q,Q1),
                                       fnc1((P1 o Q1),R).
exteriorizarConjuntor((P y Q),(P1 y Q1)) :-
                                       !, exteriorizarConjuntor(P,P1),
                                       exteriorizarConjuntor(Q,Q1).
exteriorizarConjuntor(P,P).

fnc1(((P y Q) o R),(P1 y Q1)) :- !, exteriorizarConjuntor((P o R),P1),
                                    exteriorizarConjuntor((Q o R),Q1).
fnc1((R o (P y Q)),(P1 y Q1)) :- !, exteriorizarConjuntor((R o P),P1),
                                    exteriorizarConjuntor((R o Q),Q1).
fnc1(P,P).

/* exteriorizarDisyuntor(Form,Form1) <- Form1 es la fbf resultado
                  de exteriorizar los Disyuntores de la fbf Form */
exteriorizarDisyuntor((P y Q),R) :- !, exteriorizarDisyuntor(P,P1),
                                       exteriorizarDisyuntor(Q,Q1),
                                       fnd1((P1 y Q1),R).
exteriorizarDisyuntor((P o Q),(P1 o Q1)) :-
                                       !, exteriorizarDisyuntor(P,P1),
                                       exteriorizarDisyuntor(Q,Q1).
exteriorizarDisyuntor(P,P).

fnd1(((P o Q) y R),(P1 o Q1)) :- !, exteriorizarDisyuntor((P y R),P1),
                                    exteriorizarDisyuntor((Q y R),Q1).
fnd1((R y (P o Q)),(P1 o Q1)) :- !, exteriorizarDisyuntor((R y P),P1),
                                    exteriorizarDisyuntor((R y Q),Q1).
fnd1(P,P).

 /*-------------------------------------------------------------*/
 /*              Pantalla de Ayuda                              */
ayuda :- nl,nl,write('      El programa lee una formula del Calculo
de'),
         write('Proposiciones, comprueba que  es'),nl,
         write('una  Formula  Bien Formada (fbf), y la transforma a'),
         write(' Forma   Normal  Conjuntiva'),nl,
         write('(FNC) y a Forma Normal Disyuntiva (FND)'),nl,nl,
         write('    Las conectivas usadas son:'),nl,
         tab(20),write('no       negacion'),nl,
         tab(20),write('y        conjuncion'),nl,
         tab(20),write('o        disyuncion'),nl,
         tab(20),write('imp      implicador'),nl,
         tab(20),write('coimp    coimplicador'),nl,nl,
         nl.

 /*-------------------------------------------------------------*/
 /*              Procedimientos diversos                        */

cabecera :- nl,nl,write('Logica de Primer Orden             Dpto.'),
            write(' Tecnologia Informatica y Computacion'),nl,nl,
            write('Practicas PROLOG_____________________________'),
            write('_________________FORMAS NORMALES'),nl,nl.

/****   LEER lee una formula del Calculo de proposicions   ****/
leer(F) :- nl,write('  Introduce la formula terminada en punto (.)'),
           nl,write('        (  0  Terminar   /   A  Ayuda  )'),nl,
           nl,write('  FORMULA : '),
           read(F).
```


© 1996-2001 Faraón Llorens

```
 /****   ESCRIBIR la formula en FNC y FND                  ****/
escribirFormasNormales(FNC,FND) :- write('FORMA NORMAL CONJUNTIVA:'),
                                   nl,write(FNC),nl,nl,
                                   write('FORMA NORMAL DISYUNTIVA:'),
                                    nl,write(FND).

/* miembro(Elem,Lista) <- el término Elem pertenece a lista Lista */
miembro(X,[X|_]).
miembro(X,[_|Y]) :- miembro(X,Y).

 /*****************  FIN DEL PROGRAMA  ************************/
```

11.2. ÁRBOL GENEALÓGICO

El siguiente programa representa información acerca del *árbol genealógico* de una persona así como algunas relaciones familiares: (`ej09.pl`)




```
/*******************************************************/
/*                                                     */
/*  Departamento de Ciencia de la Computación e        */
/*               Inteligencia Artificial               */
/*  Universidad de Alicante                            */
/*                                                     */
/*                LOGICA DE PRIMER ORDEN               */
/*                      Prolog                         */
/*                                                     */
/*  M. Jesús Castel                    Faraón Lloréns  */
/*                                                     */
/*       S.O.       :  MS-DOS (Windows)                */
/*       Interprete :  SWI-Prolog                      */
/*       Fichero    :  EJ09.PL                         */
/*                                                     */
/*******************************************************/

/* ÁRBOL GENEALÓGICO                                   */
```

© 1996-2001 Faraón Llorens

```
mensaje :- nl,write('Ejemplo "Árbol Genealógico" cargado.'),
           nl,nl.

/*----              PROGRAMA PRINCIPAL                  ----*/

/*------ Hechos ------*/

/* padres(H,P,M,A) <- H tiene como padre a P y como madre
                      a M, y nació el año A */
padres('Don Jose',p1,m1,1893).
padres('Doña Maria',p2,m2,1904).
padres('Jose','Don Jose','Doña Maria',1925).
padres('Maria','Don Jose','Doña Maria',1928).
padres('Antonia',p3,m3,1929).
padres('Maria Jose','Don Jose','Doña Maria',1931).
padres('Luis',p4,m4,1931).
padres('Jose Antonio','Jose','Antonia',1949).
padres('Pepe','Jose','Antonia',1953).
padres('Rosa',p5,m5,1955).
padres('Jose Luis','Luis','Maria Jose',1961).
padres('Maria Luisa','Luis','Maria Jose',1966).
padres('Pepito','Pepe','Rosa',1985).

/* casados(H,M) <- El hombre H está casado con la mujer M */
casados('Don Jose','Doña Maria').
casados('Jose','Antonia').
casados('Luis','Maria Jose').
casados('Pepe','Rosa').

/* hombre(P) <- la persona P es del género masculino */
hombre('Don Jose').
hombre('Jose').
hombre('Luis').
hombre('Jose Antonio').
hombre('Pepe').
hombre('Jose Luis').
hombre('Pepito').

/* mujer(P) <- la persona P es del género femenino */
mujer('Doña Maria').
mujer('Antonia').
mujer('Maria').
mujer('Maria Jose').
mujer('Rosa').
mujer('Maria Luisa').

/*------ Reglas ------*/

/* edad(P,E) <- la persona P tiene E años */
edad(P,E) :- padres(P,_,_,A),
             E is 1998-A.

/* mayor(P1,P2) <- la persona P1 es mayor que P2 */
mayor(P1,P2) :- padres(P1,_,_,A1),
                padres(P2,_,_,A2),
                A1<A2.

/* niño(P1) <- P1 es un niño (menos de 14 años) */
ninyo(P) :- edad(P,E),
            E=<14.

/* joven(P1) <- P1 es una persona joven (entre 14 y 25 años) */
joven(P) :- edad(P,E),
            14<E,E=<25.
```


© 1996-2001 Faraón Llorens

```
/* adulto(P1) <- P1 es un adulto (entre 25 y 50 años) */
adulto(P) :- edad(P,E),
             25<E,E=<50.

/* viejo(P1) <- P1 es una persona vieja (más de 50 años) */
viejo(P) :- edad(P,E),
            E>50.

/* hermanos(H1,H2) <- H1 es hermano/a de H2 */
hermanos(H1,H2) :- padres(H1,P,M,_),
                   padres(H2,P,M,_),
                   H1\=H2.

/* tio(T,S) <- T es el tio de S */
tio(T,S) :- hombre(T),
            padres(S,P,_,_),
            hermanos(T,P).
tio(T,S) :- hombre(T),
            padres(S,_,M,_),
            hermanos(T,M).
tio(T,S) :- hombre(T),
            padres(S,P,_,_),
            hermanos(T1,P),
            casados(T,T1).
tio(T,S) :- hombre(T),
            padres(S,_,M,_),
            hermanos(T1,M),
            casados(T,T1).

/* tia(T,S) <- T es la tia de S */
tia(T,S) :- mujer(T),
            padres(S,P,_,_),
            hermanos(T,P).
tia(T,S) :- mujer(T),
            padres(S,_,M,_),
            hermanos(T,M).
tia(T,S) :- mujer(T),
            padres(S,P,_,_),
            hermanos(T1,P),
            casados(T1,T).
tia(T,S) :- mujer(T),
            padres(S,_,M,_),
            hermanos(T1,M),
            casados(T1,T).

/* primos(P1,P2) <- P1 es primo/a de P2 */
primos(P1,P2) :- padres(P1,PA1,MA1,_),
                 padres(P2,PA2,MA2,_),
                 (hermanos(PA1,PA2);
                  hermanos(PA1,MA2);
                  hermanos(MA1,PA2);
                  hermanos(MA1,MA2)).

/* abuelo(A,N) <- A es el abuelo de N */
abuelo(A,N) :- padres(N,P,M,_),
               (padres(P,A,_,_);
                padres(M,A,_,_)).

/* abuela(A,N) <- A es la abuela de N */
abuela(A,N) :- padres(N,P,M,_),
               (padres(P,_,A,_);
                padres(M,_,A,_)).

/* antepasado(A,P) <- A es antepasado de P */
antepasado(A,P) :- padres(P,A,_,_).
antepasado(A,P) :- padres(P,_,A,_).
```

© 1996-2001 Faraón Llorens

```
antepasado(A,P) :- padres(P,PA,_,_),
                   antepasado(A,PA).
antepasado(A,P) :- padres(P,_,MA,_),
                   antepasado(A,MA).

:- mensaje.
```

11.3. JUEGO LÓGICO

El siguiente programa sirve para resolver los típicos juegos lógico s de relaciones que aparecen en las revistas de pasatiempos: (`ej10.pl`)

> *Un alumno de Informática, debido al nerviosismo del primer día de clase, ha anotado solamente el nombre de sus profesores (María, Jesús y Faraón), las asignatura que se imparten (Lógica, Programación y Matemáticas) y el día de la semana de las distintas clases (lunes, miércoles y jueves), pero sólo recuerda que :*
>
> *- La clase de Programación, impartida por María, es posterior a la de Lógica.*
>
> *- A Faraón no le gusta trabajar los lunes, día en el que no se imparte Lógica.*
>
> *¿ Serías capaz de ayudarle a relacionar cada profesor con su asignatura, así como el día de la semana que se imparte ?*
>
> *(Sabemos que cada profesor imparte una única asignatura, y que las clases se dan en días diferentes)*

Hay unos pasos comunes a todos los problemas, y otros que dependen del enunciado concreto a resolver. Por tanto, se ha separado el código Prolog en dos ficheros:

- **Juego**: esquema general del juego : (`ej10.pl`)

```
/*********************************************************/
/*                                                       */
/*  Departamento de Ciencia de la Computación e          */
/*               Inteligencia Artificial                 */
/*  Universidad de Alicante                              */
/*                                                       */
/*                LOGICA DE PRIMER ORDEN                 */
/*                      Prolog                           */
/*                                                       */
/*  M. Jesús Castel                    Faraón Lloréns    */
/*                                                       */
/*       S.O.       :  MS-DOS (Windows)                  */
/*       Interprete :  SWI-Prolog                        */
/*       Fichero    :  EJ10.PL                           */
/*                                                       */
/*********************************************************/

/* JUEGO DE LÓGICA:
   Vamos a implementar en Prolog juegos de lógica que se tienen
   que resolver por deducción a partir de un conjunto de pistas. El
   objetivo  es correlacionar una serie de propiedades que  cumplen
   distintos elementos de nuestro Dominio  (Universo del Discurso).
   La restricción a la que está sujeto este juego es que no  pueden
   tener  dos  elementos  distintos de un mismo Universo  la  misma
   característica.                                              */
```


© 1996-2001 Faraón Llorens

```
mensaje :- nl,write('Ejemplo "Juego Lógico" cargado.'),nl,
           write('Se lanza con ?- iniciar.'),
           nl,nl.

/*----              PROGRAMA PRINCIPAL                 ----*/

/* iniciar <- llamada inicial del programa */
iniciar :- write('Base de Conocimientos: '),
           read(BC),
           consult(BC),!,
           nl,write('Base de Conocimientos '),write(BC),
           write(' consultada'),nl,nl,
           numeroPropiedades(N),
           objetosUniverso(M),
           iniciar(N,M).

iniciar :- nl,write('ERROR: Base de Conocimientos no encontrada'),nl.

iniciar(2,M) :- !,ini(M,[],[]).

iniciar(3,M) :- !,ini(M,[],[],[]).

iniciar(4,M) :- !,ini(M,[],[],[],[]).

iniciar(5,M) :- !,ini(M,[],[],[],[],[]).

iniciar(N,_) :- nl,write('ERROR: Número Propiedades incorrecto = '),
                write(N),nl.

/* ini(Sol1,Sol2,...) <- Sol1 es lista con los objetos del dominio 1,
           Sol2 la lista con los objetos del dominio 2, ...
           con las soluciones respectivamente relacionadas. */

/* Correlacionar 2 propiedades */

ini(M,L1,L2) :- nel(L1,M),escribir(L1,L2),nl,pausa,fail.

ini(M,L1,L2) :- r1(Obj1,Obj2),
                nopertenece(Obj1,L1),
                nopertenece(Obj2,L2),
                ini(M,[Obj1|L1],[Obj2|L2]).

/* Correlacionar 3 propiedades */

ini(M,L1,L2,L3) :- nel(L1,M),escribir(L1,L2,L3),nl,pausa,fail.

ini(M,L1,L2,L3) :- r1(Obj1,Obj2),
                   nopertenece(Obj1,L1),
                   nopertenece(Obj2,L2),
                   r2(Obj1,Obj3),
                   nopertenece(Obj3,L3),
                   r3(Obj2,Obj3),
                   ini(M,[Obj1|L1],[Obj2|L2],[Obj3|L3]).

/* Correlacionar 4 propiedades */

ini(M,L1,L2,L3,L4) :- nel(L1,M),escribir(L1,L2,L3,L4),nl,pausa,fail.

ini(M,L1,L2,L3,L4) :- r1(Obj1,Obj2),
                      nopertenece(Obj1,L1),
                      nopertenece(Obj2,L2),
                      r2(Obj1,Obj3),
                      nopertenece(Obj3,L3),
                      r3(Obj1,Obj4),
                      nopertenece(Obj4,L4),
                      r4(Obj2,Obj3),
```

© 1996-2001 Faraón Llorens

```
                      r5(Obj2,Obj4),
                      r6(Obj3,Obj4),
                      ini(M,[Obj1|L1],[Obj2|L2],[Obj3|L3],[Obj4|L4]).

/* Correlacionar 5 propiedades */

ini(M,L1,L2,L3,L4,L5)                                                 :-
nel(L1,M),escribir(L1,L2,L3,L4,L5),nl,pausa,fail.

ini(M,L1,L2,L3,L4,L5) :- r1(Obj1,Obj2),
                         nopertenece(Obj1,L1),
                         nopertenece(Obj2,L2),
                         r2(Obj1,Obj3),
                         nopertenece(Obj3,L3),
                         r3(Obj1,Obj4),
                         nopertenece(Obj4,L4),
                         r4(Obj1,Obj5),
                         nopertenece(Obj5,L5),
                         r5(Obj2,Obj3),
                         r6(Obj2,Obj4),
                         r7(Obj2,Obj5),
                         r8(Obj3,Obj4),
                         r9(Obj3,Obj5),
                         r10(Obj4,Obj5),

ini(M,[Obj1|L1],[Obj2|L2],[Obj3|L3],[Obj4|L4],[Obj5|L5]).

/*----          RUTINAS GENERALES                    ----*/

/* escribir(Lista1,Lista2,...) <- escribe las soluciones
           correlacionadas de las listas: Lista1, Lista2 ...  */
escribir([],[]).
escribir([Obj1|Resto1],[Obj2|Resto2]) :-
              write(Obj1), write(' - '),write(Obj2),nl,
              escribir(Resto1,Resto2).

escribir([],[],[]).
escribir([Obj1|Resto1],[Obj2|Resto2],[Obj3|Resto3]) :-
              write(Obj1), write(' - '),write(Obj2),
              write(' - '), write(Obj3),nl,
              escribir(Resto1,Resto2,Resto3).

escribir([],[],[],[]).
escribir([Obj1|Resto1],[Obj2|Resto2],[Obj3|Resto3],[Obj4|Resto4]) :-
              write(Obj1), write(' - '),write(Obj2),
              write(' - '), write(Obj3),write(' - '),write(Obj4),nl,
              escribir(Resto1,Resto2,Resto3,Resto4).

escribir([],[],[],[],[]).
escribir([Obj1|Resto1],[Obj2|Resto2],[Obj3|Resto3],[Obj4|Resto4],[Obj5
|Resto5]) :-
              write(Obj1), write(' - '),write(Obj2),write(' - '),
              write(Obj3),write(' - '),write(Obj4),write(' - '),
              write(Obj5),nl,
              escribir(Resto1,Resto2,Resto3,Resto4,Resto5).

/* pausa <- detiene la ejecución del programa hasta pulsar una tecla*/
pausa :- write('Pulsa <return> para buscar otra solucion'),
         skip(10),nl.

/*----          RUTINAS DE MANEJO DE LISTAS          ----*/

/*  nopertenece(Elem,Lista)) <-  el elemento  Elem  no  pertenece  a  la
lista Lista */
nopertenece(_,[]).
nopertenece(E,[X|L]) :- E\=X,
```


© 1996-2001 Faraón Llorens

```
                        nopertenece(E,L).

/* nel(Lista,N) <- el número de elementos de la lista Lista es N */
nel([],0).
nel([_|L],N) :- nel(L,M),
                N is M+1.

:- mensaje.
```

- **Base de Conocimientos**: dependiente del enunciado concreto del problema. : (`ej10bc.pl`)

```
/*********************************************************/
/*                                                       */
/*  Departamento de Ciencia de la Computación e          */
/*               Inteligencia Artificial                 */
/*  Universidad de Alicante                              */
/*                                                       */
/*                LOGICA DE PRIMER ORDEN                 */
/*                       Prolog                          */
/*                                                       */
/*  M. Jesús Castel                      Faraón Lloréns  */
/*                                                       */
/*       S.O.       :  MS-DOS (Windows)                  */
/*       Interprete :  SWI-Prolog                        */
/*       Fichero    :  EJ10BC.PL                         */
/*                                                       */
/*                     Se debe cargar desde el programa  */
/*                     EJ10.PL                           */
/*                                                       */
/*********************************************************/

/*   Un alumno de Informática, debido al nerviosismo del primer
   día de clase, ha anotado el nombre de sus profesores (María,
   Jesús y Faraón ), las asignaturas  que se imparten ( Lógica,
   Programación  y Matemáticas ) y el día  de la semana de  las
   distintas  clases ( lunes, miércoles y jueves ),  pero  sólo
   recuerda que:
      - La  clase  de  Programación,  impartida por  María,  es
        posterior a la de Lógica
      - A Faraón  no le gusta trabajar los lunes, día en el que
        no se imparte Lógica

     ¿ Serías capaz de ayudarle a relacionar cada profesor con
   su asignatura, así como el día de la semana que se imparte ?

     ( Sabemos que cada profesor imparte una única asignatura y
   que las clases se dan en días diferentes)                */

/*----           BASE DE CONOCIMIENTOS               ----*/

numeroPropiedades(3).
objetosUniverso(3).

/*-                     PROPIEDADES                     -*/

/* prof(Profesor) <- Profesor es el nombre de un profesor */
prof(maria).
prof(jesus).
prof(faraon).

/* asig(Asignatura) <- Asignatura es el nombre de una asignatura */
asig(logica).
```

```
asig(programacion).
asig(matematicas).

/* dia(Dia) <- Dia es un dia de la semana que hay alguna clase */
dia(lunes).
dia(miercoles).
dia(jueves).

/*-                      RELACIONES                        -*/

/* r1(Profesor,Asignatura) <- Profesor imparte la Asignatura */
r1(maria,programacion).
r1(Profesor,Asignatura) :- prof(Profesor), Profesor\=maria,
                           asig(Asignatura).

/* r2(Profesor,Dia) <- Profesor imparte clases el Dia de la semana */
r2(faraon,Dia) :- dia(Dia), Dia\=lunes.
r2(Profesor,Dia) :- prof(Profesor), Profesor\=faraon,
                    dia(Dia).

/* r3(Asignatura,Dia) <- Asignatura se imparte el Dia de la semana */
r3(logica,Dia) :- dia(Dia), Dia\=lunes, Dia\=jueves.
r3(programacion,Dia) :- dia(Dia), Dia\=lunes.
r3(Asignatura,Dia) :- asig(Asignatura), Asignatura\=logica,
                      Asignatura\=programacion, dia(Dia).
```


© 1996-2001 Faraón Llorens

12. PREDICADOS PREDEFINIDOS

Vamos a ver algunos predicados predefinidos que ofrecen la mayoría de los sistemas Prolog.

1.- Adición de Bases de Conocimiento externas:

`consult(X)`	Añadir las cláusulas del fichero *X* a la base de conocimiento
`reconsult(Y)`	Las cláusulas leídas de fichero *Y* sustituyen a las existentes para el mismo predicado
`[X,-Y,Z]`	Notación más cómoda para consult y reconsult
`halt`	Salir del sistema Prolog

2.- Construcción de objetivos compuestos (conectivas lógicas):

`X,Y`	Conjunción de objetivos
`X;Y`	Disyunción de objetivos
`\+ X`	Se cumple si fracasa el intento de satisfacer *X*
`not(X)`	Se cumple si fracasa el intento de satisfacer *X*

3.- Clasificación de términos:

`var(X)`	Se cumple si *X* es en ese momento una variable no instanciada
`nonvar(X)`	Se cumple si *X* no es una variable sin instanciar en ese momento
`atomic(X)`	Se cumple si *X* representa en ese momento un número o un átomo
`atom(X)`	Se cumple si *X* representa en ese momento un átomo de Prolog
`numeric(X)`	Se cumple si *X* representa en ese momento un número
`integer(X)`	Se cumple si *X* representa en ese momento un número entero
`real(X)`	Se cumple si *X* representa en ese momento un número real

4.- Control del programa:

`true`	Objetivo que siempre se cumple
`fail`	Objetivo que siempre fracasa
`!`	Corte. Fuerza al sistema Prolog a mantener ciertas elecciones
`repeat`	Sirve para generar soluciones múltiples mediante el mecanismo de reevaluación
`X -> Y`	Si *X* entonces *Y*. En combinación con ; actúa como Si-Entonces-Sino.
`call(X)`	Se cumple si tiene éxito el intento de satisfacer *X* (instanciada a un término)

5.- Operadores aritméticos y relacionales:

`X=Y`	Se cumple si puede hacer *X* e *Y* iguales (unificarlos)
`X\=Y`	Se cumple si *X=Y* fracasa
`X==Y`	Igualdad más estricta
`X\==Y`	Se cumple si fracasa ==
`X < Y`	Predicado menor que
`X > Y`	Predicado mayor que
`X >= Y`	Predicado mayor o igual que

© 1996-2001 Faraón Llorens

X =< Y	Predicado menor o igual que
X @< Y	Ordenación de términos. Predicado menor que
X @> Y	Ordenación de términos. Predicado mayor que
X @>= Y	Ordenación de términos. Predicado mayor o igual que
X @=< Y	Ordenación de términos. Predicado menor o igual que
X is Y	Se evalúa la expresión a la que esta instanciada *Y* para dar como resultado un número, que se intentará hacer coincidir con *X*
X is_string Y	Se evalúa la expresión a la que está instanciada *Y* para dar como resultado una cadena, que se intentará hacer coincidir con *X*
X & Y	Operador concatenación de cadenas de caracteres
X + Y	Operador suma
X - Y	Operador resta
X * Y	Operador de multiplicación
X / Y	Operador de división
X // Y	Operador de división entera
X mod Y	Operador de resto de la división entera
X ** Y	Operador de exponenciación
X ^ Y	Operador de exponenciación
op(X,Y,Z)	Declara el operador de nombre *Z* con clase de precedencia *X* y posición-asociatividad *Y*
current_op(X,Y,Z)	Busca el operador ya declarado de nombre *Z* con clase de precedencia *X* y posición-asociatividad *Y*

6.- Funciones aritméticas:

abs(X)	Devulve el valor absoluto de la expresión *X*
sign(X)	Devuelve -1 si $X < 0$, 1 si $X > 0$ y 0 si $X = 0$
min(X,Y)	Devuelve el menor de *X*e *Y*
max(X,Y)	Devuelve el mayor de *X*e *Y*
random(X)	Devuelve un entero aleatorio i ($0 =< i < X$); la es determinada por el reloj del sistema cuando se arranca SWI-Prolog
round(X)	Evalua la expresión *X* y la redondea al entero más cercano
integer(X)	Evalua la expresión *X* y la redondea al entero más cercano
float(X)	Evalua la expresión *X* en un número en coma flotante
truncate(X)	Trunca la expresión *X* en un número entero
floor(X)	Devuleve el mayor número entero menor o igual que el resultado de la expresión *X*
ceiling(X)	Devuleve el menor número entero mayor o igual que el resultado de la expresión *X*
sqrt(X)	Raíz cuadrada de la expresión *X*
sin(X)	Seno de la expresión *X* (angulo en radianes)
cos(X)	Coseno de la expresión *X* (angulo en radianes)
tan(X)	Tangente de la expresión *X* (angulo en radianes)
asin(X)	Arcoseno (angulo en radianes) de la expresión *X*
acos(X)	Arcocoseno (angulo en radianes) de la expresión *X*
atan(X)	Arcotangente (angulo en radianes) de la expresión *X*
log(X)	Logaritmo neperiano de la expresión *X*
log10(X)	Logaritmo en base 10 de la expresión *X*
exp(X)	e elevado al resultado de la expresión *X*
pi	Constante matemática *pi* (3.141593)

© 1996-2001 Faraón Llorens

e	Constante matemática *e* (2.718282)

7.- Manejo de listas:

[X|Y]	*X* es la cabeza de *Y* la cola de la lista
is_list(X)	Se cumple si *X* es una lista o la lista vacía []
append(X,Y,Z)	*Z* es la concatenación de las listas *X* e *Y*
member(X,Y)	*X* es uno de los elementos de la lista *Y*
delete(X,Y,Z)	Borra el elemento *Y* de la lista *X* y da como resultado la lista *Z*
select(X,Y,Z)	Selecciona el elemento *X* de la lista *Y* y da como resultado la lista *Z*
nth0(X,Y,Z)	El elemento *X*-ésimo de la lista *Y* es *Z* (empezando en 0)
nth1(X,Y,Z)	El elemento *X*-ésimo de la lista *Y* es *Z* (empezando en 1)
last(X,Y)	*X* es el último elemento de la lista *Y*
reverse(X,Y)	*Y* es la lista invertida de *X*
length(X,Y)	*Y* es la longitud de la lista *X*
merge(X,Y,Z)	Siendo *X* e *Y* listas ordenadas, *Z* es la lista ordenada con los elementos de ámbas
sort(X,Y)	*Y* es la lista ordenada de *X* (sin elementos duplicados)
msort(X,Y)	*Y* es la lista ordenada de *X*

8.- Predicados de Entrada y Salida:

get0(X)	Se cumple si *X* puede hacerse corresponder con el siguiente carácter encontrado en el canal de entrada activo (en código ASCII)
get(X)	Se cumple si *X* puede hacerse corresponder con el siguiente carácter imprimible encontrado en el canal de entrada activo
skip(X)	Lee y pasa por alto todos los caracteres del canal de entrada activo hasta que encuentra un carácter que coincida con *X*
read(X)	Lee el siguiente término del canal de entrada activo
put(X)	Escribe el entero *X* como un carácter en el canal de salida activo
nl	Genera una nueva línea en el canal de salida activo
tab(X)	Escribe *X* espacios en el canal de salida activo
write(X)	Escribe el término *X* en el canal de salida activo
write_ln(X)	Escribe el término *X* en el canal de salida activo y salta la línea
writef(X,Y)	Escribe en el canal de salida activo, siendo *X* el formato e *Y* la lista con los argumentos a escribir
display(X)	Escribe el término *X* en el canal de salida activo, pasando por alto las declaraciones de operadores

9.- Manejo de ficheros:

see(X)	Abre el fichero *X* y lo define como canal de entrada activo
seeing(X)	Se cumple si el nombre del canal de entrada activo coincide con *X*
seen	Cierra el canal de entrada activo, y retorna al teclado
tell(X)	Abre el fichero *X* y lo define como canal de salida activo
telling(X)	Se cumple si *X* coincide con el nombre del canal de salida activo
told	Cierra con un fin de fichero el canal de salida activo, que pasa a ser la pantalla

10.- Manipulación de la Base de Conocimiento:

© 1996-2001 Faraón Llorens

listing	Se escriben en el fichero de salida activo todas las cláusulas
listing(X)	Siendo *X* un átomo, se escriben en el fichero de salida activo todas las cláusulas que tienen como predicado dicho átomo
clause(X,Y)	Se hace coincidir *X* con la cabeza e *Y* con el cuerpo de una cláusula existente en la base de conocimiento
assert(X)	Permite añadir la nueva cláusula *X* a la base de conocimiento
asserta(X)	Permite añadir la nueva cláusula *X* al principio de la base de conocimiento
assertz(X)	Permite añadir la nueva cláusula *X* al final de la base de conocimiento
retract(X)	Permite eliminar la primera cláusula de la base de conocimiento que empareje con *X*
retractall(X)	Permite eliminar todas las cláusulas de la base de conocimiento que emparejen con *X*
abolish(X)	Retira de la Base de Conocimiento todas las cláusulas del predicado *X*
abolish(X,Y)	Retira de la Base de Conocimiento todas las cláusulas del predicado de nombre *X* y número de argumentos *Y*

11.- Construcción y acceso a componentes de estructuras:

functor(E,F,N)	*E* es una estructura con nombre *F* y número de argumentos *N*
arg(N,E,A)	El argumento número *N* de la estructura *E* es *A*
E=..L	Predicado *univ*. *L* es la lista cuya cabeza es el functor de la estructura *E* y la cola sus argumentos
name(A,L)	Los caracteres del átomo *A* tienen por ASCII los números de la lista *L*

12.- Depuración de programas Prolog:

trace	Activación de un seguimiento exhaustivo, generando la traza de la ejecución de las metas siguientes
trace(X)	Activación de un seguimiento del predicado *X*
notrace	Desactiva el seguimiento exhaustivo
notrace(X)	Desactiva el seguimiento mientras dure la llamada al objetivo *X*
spy(X)	Fijación de puntos espía en el predicado *X*
nospy(X)	Elimina los puntos espía especificados
nospyall	Elimina todos los puntos espía
debugging	Ver el estado del depurador y los puntos espía se han establecido hasta el momento
debug	Inicia el depurador
nodebug	Desactiva el depurador

© 1996-2001 Faraón Llorens

13. SISTEMAS PROLOG

Existen gran cantidad de implementaciones del lenguaje de programación lógica Prolog, tanto en versiones comerciales como de libre distribución. Por razones de facilitar el acceso a dichos sistemas, vamos a comentar dos sistemas Prolog de libre distribución para ordenadores PC, de forma que el lector puede acceder a ellos, instalarlos en su máquina y así ejecutar los ejemplos de este anexo (y lo más importante, probar todo aquello que se le ocurra).

Si se quiere una lista más extensa de implementaciones del lenguaje Prolog, con las direcciones donde localizarlas, se puede visitar la página Web:

http://gruffle.comlab.ox.ac.uk/archive/logic-prog.html

13.1. PROLOG-2

Prolog-2 es un sistema Prolog de Dominio Público de "Expert Systems Ltd". Es una implementación del Prolog de Edinburgh descrito por Clocksin y Mellish en el libro "Programación en Prolog", convertido ya en un estandar. Encontraremos mayor información sobre esta implementación en los libros "The Prolog-2 User Guide" y "The Prolog-2 Encyclopaedia" de Tony Dodd publicados por Ablex Publishing Corporation. El software lo podemos encontrar via ftp en la siguiente dirección:

ftp://ai.uga.edu/pub/prolog

Su principal ventaja es que se puede trabajar sobre sistemas DOS muy sencillos. Se puede ejecutar tanto desde disco duro como desde disquete, tecleando:

```
prolog2
```

Si da problemas durante el arranque, comprobar que el fichero CONFIG.SYS contenga una línea que nos permita tener abiertos al menos 20 archivos. Si no, añadir la línea:

```
FILES=20
```

El sistema Prolog-2 posee un editor integrado, al que accedemos con el predicado predefinido *edit*. El fichero es automáticamente reconsultado después de su edición. Así:

```
?- edit(fichero).
```

edita el fichero *fichero.pro* y lo reconsulta. Sólo están libres para la edición 128.000 bytes. Si necesitamos más, la llamada al editor deberá ser del tipo:

```
?- edit(fichero,200000).
```

© 1996-2001 Faraón Llorens

que dejara libres para edición 200.000 bytes de memoria. El editor que utiliza por defecto el sistema Prolog-2 es PrEd ("Program Editor"), un editor de ficheros de textos en ASCII especialmente diseñado para entornos con disponibilidad de memoria limitada (necesita únicamente 55 K). Incorpora una iluminación automática para el emparejamiento de paréntesis y otros símbolos delimitadores, así como de un sistema de ayuda que muestra las distintas combinaciones de teclas, que se puede activar con la pulsación de la tecla F3. Para más detalles de este editor consultar el archivo "pred.doc". Podemos utilizar cualquier otro editor que nosotros queramos, simplemente debemos modificar del archivo "edit.bat" la llamada a *pred*, ya que el predicado predefinido *edit* invoca al fichero de lotes "edit.bat" .

Para añadir cláusulas a Prolog podemos editar el fichero mediante edit, y al salir, como lo reconsulta automáticamente, quedan incorporadas a Prolog. También podemos añadir cláusulas mediante el predicado:

```
[user].
```

De esta manera entramos en el *modo de consulta*, indicado por el prompt "C:". Una vez tecleadas las cláusulas podemos volver al *modo de pregunta*, indicado por el prompt "?-", tecleando ^Z.

Las *teclas de función*, dentro del sistema Prolog-2, tienen el siguiente significado:

F1	Ayuda
F2	Menú
F3	Línea de comandos introducida anteriormente
F4	Línea siguiente
F5	Borrar
F6	Información sobre las teclas de función
F7	Ocultar
F8	Ocultar Debugger
F9	Ocultar todo
F10	Mostrar Debugger

13.2. SWI-PROLOG

SWI-Prolog es un compilador Prolog de Dominio Público, que como el anterior sigue el estándar de Edinburgh. Ha sido diseñado e implementado por Jan Wielemaker, del departamento Social Science Informatics (SWI) de la Universidad de Amsterdam. El software lo podemos encontrar en la siguiente dirección:

http://www.swi.psy.uva.nl/projects/SWI-Prolog

© 1996-2001 Faraón Llorens

La plataforma original sobre la que se desarrollo fue para Unix, aunque en la actualidad podemos encontrar versiones para Linux y para PC que se ejecutan bajo el entorno Windows (Windows 3.1., Windows 95 y Windows NT). Es potente y flexible (permite integración con C), y tiene las ventajas sobre el anterior de que, de momento, se mantiene actualizado y se ejecuta sobre un entorno gráfico más agradable para el usuario.

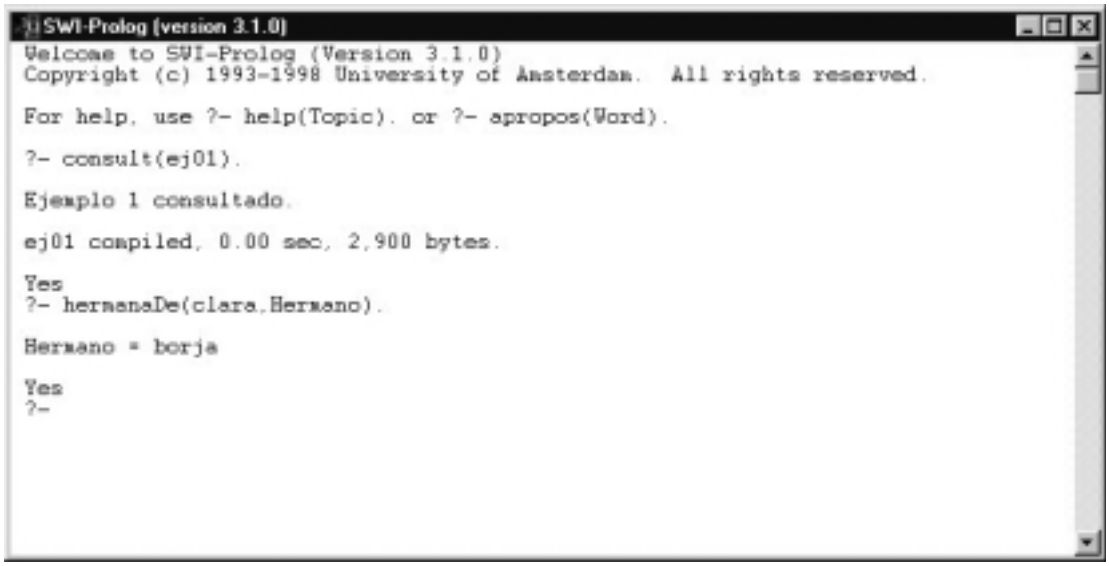
Posee un sistema de ayuda en línea que se puede consultar mediante los predicados:

```
?- help.
?- help(Termino).
?- apropos(Cadena).
```

También se puede conseguir una versión escrita del manual de referencia, tanto en formato PostScript como en html, en la misma dirección.

Para ejecutarlo, una vez iniciado Windows, basta con hacer doble click sobre el icono correspondiente (plwin.exe).

Abriéndose la ventana de SWI-Prolog:



Al estar desarrollado sobre plataforma Unix, reconoce algunas órdenes del sistema operativo que nos pueden ser de utilidad para la gestión de los archivos:

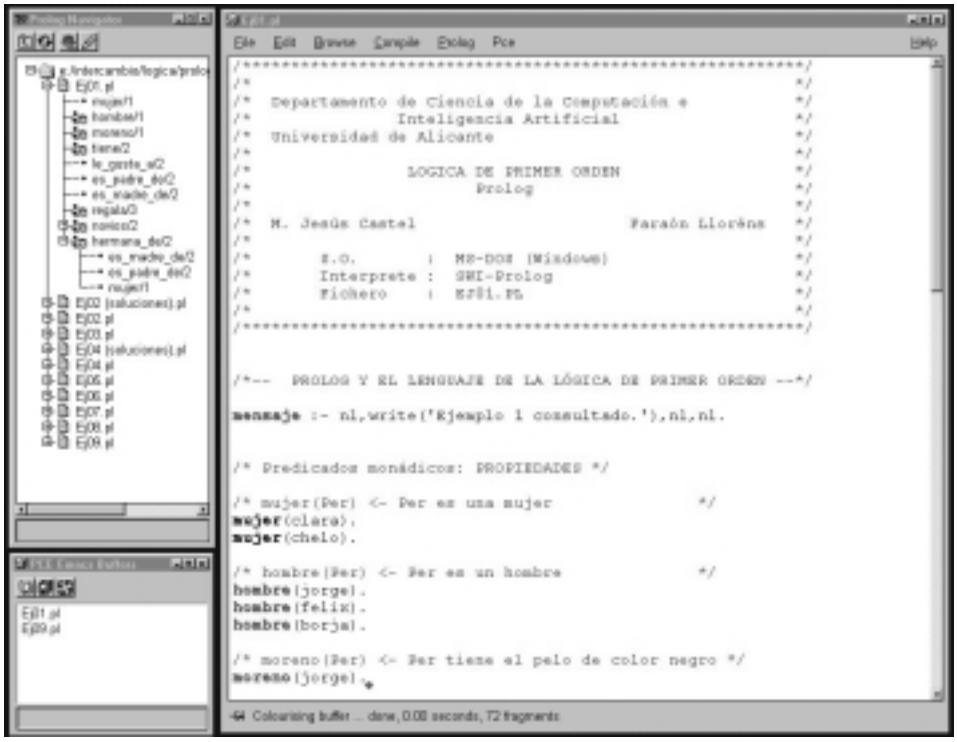
`cd`	cambiar el directorio de trabajo
`pwd`	muestra el directorio de trabajo
`ls`	muestra el contenido del directorio de trabajo
...	

© 1996-2001 Faraón Llorens

A partir de la versión 4 incorpora un sistema GUI (“Graphical User Interface”) que nos permite, entre otras cosas, tener un frontal gráfico para la depuración. Se activa con `guitracer` y se desactiva con `noguitracer`.

También incorpora el editor PCE Emacs, que reconoce la sintaxis de colores para Prolog, dispone de un navegador Prolog, ...

```
?- emacs(Fichero).
```



© 1996-2001 Faraón Llorens

ANEXO: CUADRO COMPARATIVO DE LAS DIFERENTES NOTACIONES PARA PROGRAMACIÓN LÓGICA

Como resumen final veamos un cuadro comparativo de las diferentes notaciones usadas para las sentencias, desde las fbf del cálculo de predicados hasta la sintaxis de Prolog, pasando por la notación clausal y la de la programación lógica:

	Fórmula	Cláusulas de Horn	Program. Lógica	Prolog
REGLA	$\forall (A \vee \neg N_1 \vee ... \vee \neg N_m)$ $\forall ((N_1 \wedge ... \wedge N_m) \rightarrow A)$	$\{A, \neg N_1 , ... , \neg N_m \}$	$A \leftarrow N_1 , ... , N_m$	A :- N_1 , ..., N_m .
HECHO	$\forall$ (A)	{ A }	$A \leftarrow$	A.
OBJETIVO	$\forall (\neg N_1 \vee ... \vee \neg N_m)$ $\neg\exists (N_1 \wedge ... \wedge N_m)$	$\{ \neg N_1 , ... , \neg N_m \}$	$\leftarrow N_1 , ... , N_m$	?- N_1 , ... , N_m .

Ejemplo:

	Fórmula	Cláusulas de Horn	Program. Lógica	Prolog
REGLA	$\forall x (\exists y$ marido(x,y) $\rightarrow$ casado(x))	{ casado(x), ¬marido(x,y) }	casado(x) $\leftarrow$ marido(x,y)	casado(x) :- marido(x,y).
HECHO	marido(jordi,ana)	{ marido(jordi,ana) }	marido(jordi,ana) $\leftarrow$	marido(jordi,ana).
OBJETIVO (negado)	$\neg\exists$x casado(x)	{ ¬ casado(x) }	$\leftarrow$ casado(x)	?- casado(x) .

© 1996-2001 Faraón Llorens

BIBLIOGRAFÍA

Amble, 1987
Amble, Tore
Logic Programming and Knowledge Engineering
Ed. Addison-Wesley, 1987

Berk, 1986
Berk, A.A.
PROLOG. Programación y aplicaciones en Inteligencia Artificial
Ed. ANAYA Multime., S.A., 1986

Bratko, 1990
Bratko, Ivan
PROLOG Programming for Artificial Intelligence
Addison-Wesley, second ed., 1990

Clocksin, 1997
Clocksin, William F.
Clause and Effect. Prolog Programming for the Working Programmer
Springer-Verlag, 1997

Clocksin y Mellish, 1987
Clocksin, W. F. y Mellish, C. S.
Programación en PROLOG
Ed. Gustavo Gili, S.A., 1987

Covington, 1993
Covington, Michael A.
ISO Prolog. A Summary of the Draft Proposed Standard
A. I. Prog. Univ. of Georgia 1993

Dahl y Sobrino,, 1996
Dahl, Veronica y Sobrino, Alejandro
Estudios sobre programación lógica y sus aplicaciones
Universidade de Santiago de Compostela, Servicio de Publicacións e Intercambio Científico, 1996

Deransart y otros., 1996
Deransart, P., Ed-Dbali, A. y Cervoni, L.
Prolog: The Standard
Springer-Verlag, New York, 1996.

Dodd
Dodd, Tony
The Prolog-2 Encyclopaedia
Ablex Publishing Corporation

Dodd
Dodd, Tony
The Prolog-2 User Guide
Ablex Publishing Corporation

Dodd, 1990
Dodd, Tony
Prolog. A Logical Approach
Oxford University Press, 1990

© 1996-2001 Faraón Llorens

Giannesini, y otros, 1989
Giannesini, F., Kanoui, H., Pasero, R. y van Caneghem, M.
Prolog
Addison-Wesley Iberoamericana, 1989

Kim, 1991
Kim, Steven H.
Knowledge Systems Through PROLOG
Oxford University Press, 1991

Kowalski, 1986
Kowalski, Robert A.
Lógica, Programación e Inteligencia Artificial
Ed. Díaz de Santos, 1986.

Kowalski y Kriwaczek, 1986
Kowalski, Bob y Kriwaczek, Frank
Logic Programming. Prolog and its applications
Addison-Wesley Pub. Co., 1986

Lloyd, 1993
Lloyd, J. W.
Foundations of Logic Programming
Ed. Springer-Verlag, 1993

Maier y Warren, 1988
Maier, David y Warren, David S.
Computing with Logic. Logic programming with Prolog
The Benjamin/Cummings, 1988

O'Keefe, 1990
O'Keefe, Richard A.
The craft of Prolog
The MIT Press, Cambridge, 1990

Sterling y Shapiro, 1994
Sterling, L. y Shapiro, E.
The art of PROLOG. Advanced Programming Techniques
The MIT Press, second ed., 1994

Wielemaker, 2001
Wielemaker, Jan
SWI-Prolog 4.0. Reference Manual
Dept. of Social Science Informatics (SWI), University of Amsterdam, 2001

JLP, 1994
Special Issue: *Ten Years of Logic Programming*
Journal of Logic Programming, vol. 19/20, may/july 1994

© 1996-2001 Faraón Llorens