UNIDAD 5 – PROGRAMACIÓN LÓGICA

1 Introducción

La programación lógica, junto con la funcional, forma parte de lo que se conoce como programación declarativa. En los lenguajes tradicionales, la programación consiste en indicar cómo resolver un problema mediante sentencias; en la programación lógica, se trabaja de una forma descriptiva, estableciendo relaciones entre entidades, indicando no cómo, sino qué hacer.

La ecuación de Robert Kowalski (Universidad de Edimburgo) establece la idea esencial de la programación lógica: algoritmos = lógica + control. Es decir, un algoritmo se construye especificando conocimiento en un lenguaje formal (lógica de primer orden), y el problema se resuelve mediante un mecanismo de inferencia (control) que actúa sobre aquél.

La programación lógica es un paradigma de los lenguajes de programación en el cual los programas se consideran como una serie de aserciones lógicas. De esta forma, el conocimiento se representa mediante reglas, tratándose de sistemas *declarativos*. Una representación declarativa es aquella en la que el conocimiento está especificado, pero en la que la manera en que dicho conocimiento debe ser usado no viene dado. El más popular de los sistemas de programación lógica es el PROLOG.

1.1 El lenguaje PROLOG

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, si bien uno de los principales protagonistas de su desarrollo y promoción fue Robert Kowalski 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, 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 computadores 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 mujeres y tienen los mismos padres". Como

veremos más adelante, esto es lo que hacemos en PROLOG.

1.2 La programación lógica en PROLOG

Un programa escrito en PROLOG puro, es un conjunto de cláusulas de Horn. Sin embargo, PROLOG, como lenguaje de programación moderno, incorpora más cosas, como instrucciones de Entrada/Salida, etc.

Una cláusula de Horn puede ser ó bien una conjunción de hechos positivos ó una implicación con un único consecuente (un único termino a la derecha). La negación no tiene representación en PROLOG, y se asocia con la falta de una afirmación (negación por fallo), según el modelo de *suposición de un mundo cerrado* (CWA); solo es cierto lo que aparece en la base de conocimiento ó bien se deriva de esta.

Las diferencias sintácticas entre las representaciones lógicas y las representaciones PROLOG son las siguientes:

1. En PROLOG todas las variables están implícitamente cuantificadas universalmente.

2. En PROLOG existe un símbolo explicito para la conjunción "y" (,), pero no existe uno para la disyunción "o", que se expresa como una lista de sentencias alternativas.

3. En PROLOG, las implicaciones p $\rightarrow$ q se escriben al revés q :- p, ya que el intérprete siempre trabaja hacia atrás sobre un objetivo, como se verá más adelante.

2 ESTRUCTURAS BÁSICAS

PROLOG cuenta con dos tipos de estructuras: términos y sentencias. Los términos pueden ser constantes, variables o functores:

- Las constantes, representadas por una cadena de caracteres, pueden ser números o cualquier cadena que comience en minúscula.

- Las variables son cadenas que comienzan con una letra mayúscula.

- Los functores son identificadores que empiezan con minúscula, seguidos de una lista de parámetros (términos) entre paréntesis, separados por comas.

Las sentencias son reglas o cláusulas. Hay hechos, reglas con cabeza y cola, y consultas.

2.1 PREDICADOS O HECHOS

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:

/* Predicados monádicos: PROPIEDADES */

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

/* 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 */

```
/* 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(chela,borja).
es_madre_de(chela, clara).

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

Supongamos que queremos expresar el hecho de que "un coche tiene ruedas". Este hecho, consta de dos objetos, "coche" y "ruedas", y de una relación llamada "tiene". La forma de representarlo en PROLOG es:

```
tiene(coche,ruedas).
```

El orden de los objetos dentro de la relación es arbitrario, pero debemos ser coherentes a lo largo de la base de hechos.

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 computador y la implementación.
  - *Enteros*: en la implementación de PROLOG-2 puede utilizarse cualquier entero que el intervalo [-223,223-1]=[-8.388.608,8.388.607].
  - *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

2.3. VARIABLES

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. Estará instanciada cuando existe un objeto determinado representado por la variable. De este modo, cuando preguntamos "¿Un coche tiene X?", PROLOG busca en los hechos cosas que tiene un coche y respondería:

X = ruedas. (instanciando la variable X con el objeto ruedas)

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

- 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 cual 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 $\forall x\ \neg gusta(jorge,x)$ $\neg\exists x\ gusta(jorge,x)$

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

Un caso particular es la variable anónima ("_"), es una especie de comodín que utilizaremos en aquellos lugares que debería aparecer una variable, pero no nos interesa darle un nombre concreto ya que no vamos a utilizarla posteriormente.

2.4. 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.

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

2.5. REGLAS

Las reglas se utilizan en PROLOG para significar que un hecho depende de uno ó más hechos. Son la representación de las implicaciones lógicas del tipo p $\rightarrow$ q (p implica q).

- Una regla consiste en una cabeza y un cuerpo, unidos por el signo ":-".
- La cabeza está formada por un único hecho.
- El cuerpo puede ser uno o más hechos (conjunción de hechos), separados por una coma (","), que actúa como el "y" lógico.
- Las reglas finalizan con un punto (".").

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 :- O1, O2, ..., On.

puede ser leída declarativamente como:

"La demostración de la cláusula *C* se sigue de la demostración
de los objetivos *O1, O2, ..., On.*"

o procedimentalmente como:

"Para ejecutar el procedimiento *C*, se deben llamar para
su ejecución los objetivos *O1,O2,...,On.*"

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

cuerpo_de_la_regla $\rightarrow$ 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*).

Llamaremos cláusulas de un predicado tanto a los hechos como a las reglas. Una colección de cláusulas forma una Base de Conocimientos.

La cabeza en una regla PROLOG corresponde al consecuente de una implicación lógica, y el cuerpo al antecedente. Este hecho puede conducir a errores de representación. Supongamos el siguiente razonamiento lógico:

tiempo(lluvioso) → suelo(mojado)
suelo(mojado)

Que el suelo esté mojado, es una condición suficiente de que el tiempo sea lluvioso, pero no necesaria. Por lo tanto, a partir de ese hecho, no podemos deducir mediante la implicación, que esté lloviendo (pueden haber regado las calles). La representación **correcta** en PROLOG, sería:

```
suelo(mojado) :- tiempo(lluvioso).
suelo(mojado).
```

Adviértase que la regla esta "al revés". Esto es así por el mecanismo de deducción hacia atrás que emplea PROLOG. Si cometiéramos el **error** de representarla como:

```
tiempo(lluvioso) :- suelo(mojado).
suelo(mojado).
```

PROLOG, partiendo del hecho de que el suelo está mojado, deduciría incorrectamente que el tiempo es lluvioso.

Para generalizar una relación entre objetos mediante una regla, utilizaremos variables. Por ejemplo:

Representación lógica	**Representación PROLOG**
es_un_coche(X) →	tiene(X,ruedas) :-
tiene(X,ruedas)	es_un_coche(X).

Con esta regla generalizamos el hecho de que cualquier objeto que sea un coche, tendrá ruedas. Al igual que antes, el hecho de que un objeto tenga ruedas, no es una condición suficiente de que sea un coche. Por lo tanto la representación inversa sería incorrecta.

El ámbito de las variables

Cuando en una regla aparece una variable, el *ámbito* de esa variable es únicamente esa regla. Supongamos las siguientes reglas:

```
(1)  hermana_de(X, Y) :- mujer(X), padres(X, M, P), padres(Y, M, P).
(2)  puede_robar(X, P) :- ladron(X), le_gusta_a(X, P), valioso(P).
```

Aunque en ambas aparece la variable X (y la variable P), no tiene nada que ver la X de la regla (1) con la de la regla (2), y por lo tanto, la instanciación de la X en (1) no implica la instanciación en (2). Sin embargo todas las X de **una misma regla** sí que se instanciarán con el mismo valor.

2.6. OPERADORES

Son predicados predefinidos en PROLOG para las operaciones matemáticas básicas. Su sintaxis depende de la posición que ocupen, pudiendo ser infijos ó prefijos. Por ejemplo el operador suma ("+"), podemos encontrarlo en forma prefija '+(2,5)' ó bien infija, '2 + 5'.

También disponemos de predicados de igualdad y desigualdad.

X = Y	igual
X \= Y	distinto
X < Y	menor
X > Y	mayor
X =< Y	menor ó igual
X >= Y	mayor ó igual

Al igual que en otros lenguajes de programación es necesario tener en cuenta la *precedencia* y la *asociatividad* de los operadores antes de trabajar con ellos.

En cuanto a precedencia, es la típica. Por ejemplo, 3+2*6 se evalúa como 3+(2*6). En lo referente a la asociatividad, PROLOG es asociativo por la izquierda. Así, 8/4/4 se interpreta como (8/4)/4. De igual forma, 5+8/2/2 significa 5+((8/2)/2).

El operador 'is'

Es un operador infijo, que en su parte derecha lleva un término que se interpreta como una expresión aritmética, contrastándose con el término de su izquierda.

Por ejemplo, la expresión '6 is 4+3.' es falsa. Por otra parte, si la expresión es 'X is 4+3.', el resultado será la instanciación de X:

X = 7

Una regla PROLOG puede ser esta:

densidad(X,Y) :- poblacion(X,P), area(X,A), Y is P/A.

3 PROGRAMACION BASICA EN PROLOG.

3.1 UN EJEMPLO SENCILLO.

Con los datos que conocemos, ya podemos construir un programa en PROLOG. Necesitaremos un editor de textos para escribir los hechos y reglas que lo componen. Un ejemplo sencillo de programa PROLOG es el siguiente:

```
quiere_a(maria,enrique).
quiere_a(juan,jorge).
quiere_a(maria,susana).
quiere_a(maria,ana).
quiere_a(susana,pablo).
quiere_a(ana,jorge).
hombre(juan).
hombre(pablo).
hombre(jorge).
hombre(enrique).
mujer(maria).
mujer(susana).
mujer(ana).
teme_a(susana,pablo).
teme_a(jorge,enrique).
teme_a(maria,pablo).
/* Esta linea es un comentario */
quiere_pero_teme_a(X,Y) :- quiere_a(X,Y), teme_a(X,Y).
querido_por(X,Y) :- quiere_a(Y,X).
puede_casarse_con(X,Y) :- quiere_a(X,Y), hombre(X), mujer(Y).
puede_casarse_con(X,Y) :- quiere_a(X,Y), mujer(X), hombre(Y).
```

Una vez creado, lo salvaremos para su posterior consulta desde el interprete PROLOG. Un programa PROLOG tiene como extensión por defecto '.PRO'. Le daremos el nombre 'relacion.pro'.

3.2 INTRODUCCION AL PROLOG DE A.D.A. (AUTOMATA DESIGN ASSOCIATES).

Este apunte se basa en el A.D.A. PROLOG. Al ejecutarlo nos aparecerá la información referente al autor y versión del programa y a continuación aparece el símbolo "?-"

Se trata del prompt de PROLOG, que nos indica que está dispuesto para recibir comandos. Un comando *siempre* debe finalizar con un punto (".").

ALGUNOS COMANDOS BASICOS:

consult: El predicado *consult* está pensado para leer y compilar un programa PROLOG ó bien para las situaciones en las que se precise añadir las clausulas existentes en un determinado fichero a las que ya están almacenadas y compiladas en la base de datos. Su sintaxis puede ser una de las siguientes:

```
consult(fichero).
consult('fichero.ext').
consult('c:\dsp\prolog\fichero').
```

recon: El predicado *recon* es muy parecido a *consult*, con la salvedad de que las cláusulas existentes en el fichero consultado, reemplazan a las existentes en la base de hechos. Puede ser útil para sustituir una única cláusula sin consultar todas las demás, situando esa cláusula en un fichero. Su sintaxis es la misma que la de *consult*.

[NOTA: El predicado *recon* puede encontrarse como *reconsult* en otras implementaciones de PROLOG]

forget: Tiene como fin eliminar de la base de datos actual aquellos hechos consultados de un fichero determinado. Su sintaxis es:

```
forget(fichero).
```

exitsys: Este predicado nos devuelve al sistema operativo.

3.3 LA RESOLUCION DE OBJETIVOS.

Ya hemos creado un programa PROLOG [relacion.pro] y lo hemos compilado en nuestro interprete PROLOG [consult(relacion)]. A partir de este momento podemos interrogar la base de datos, mediante consultas.

Una consulta tiene la misma forma que un hecho. Consideremos la pregunta:

```
?- quiere_a(susana,pablo).
```

PROLOG buscará por toda la base de datos hechos que *coincidan* con el anterior. Dos hechos coinciden si sus predicados son iguales, y cada uno de sus correspondientes argumentos lo son entre sí. Si PROLOG encuentra un hecho que coincida con la pregunta, responderá *yes*. En caso contrario responderá *no*.

Además, una pregunta puede contener variables. En este caso PROLOG buscara por toda la base de hechos aquellos objetos que pueden ser representado por la variable. Por ejemplo:

```
?- quiere_a(maria, Alguien).
```

[NOTA: Alguien es un nombre perfectamente válido de variable, puesto que empieza por una letra mayúscula.]

El resultado de la consulta es:

```
Alguien = enrique
More (Y/N):
```

El hecho 'quiere_a(maria,enrique).' coincide con la pregunta al instanciar la variable Alguien con el objeto 'enrique'. Por lo tanto es una respuesta válida, pero no la única. Por eso se nos pregunta si queremos obtener más respuestas. En caso afirmativo, obtendríamos:

```
Alguien = Susana
More (Y/N):y
Alguien = ana
More (Y/N):y
No.
```

Cuando Prolog no tenga más respuestas, nos dirá que No; esa negativa no significa que lo que preguntemos sea falso, sino que no lo conoce o no puede demostrarlo porque nuestro programa no cuenta con el conocimiento suficiente. Esto se denomina negación por falla.

Las consultas a una base de datos se complican cuando estas están compuestas por conjunciones ó bien intervienen reglas en su resolución.

Conviene, por lo tanto, conocer cuál es el mecanismo de control del PROLOG, con el fin de comprender el porqué de sus respuestas.

4 EL MECANISMO DE CONTROL DEL PROLOG

El mecanismo empleado por PROLOG para satisfacer las cuestiones que se le plantean, es el de *razonamiento hacia atrás* (backward) complementado con la *búsqueda en profundidad* (depth first) y la *vuelta atrás* ó *reevaluación* (backtracking).

Razonamiento hacia atrás: Partiendo de un objetivo a probar, busca las aserciones que pueden probar el objetivo. Si en un punto caben varios caminos, se recorren en el orden que aparecen en el programa, esto es, de arriba a abajo y de izquierda a derecha.

Reevaluación: Si en un momento dado una variable se instancia con determinado valor con el fin de alcanzar una solución, y se llega a un camino no satisfactorio, el mecanismo de control retrocede al punto en el cual se instanció la variable, la des-instancia y si es posible, busca otra instanciación que supondrá un nuevo camino de búsqueda.

Se puede ilustrar esta estrategia sobre el ejemplo anterior. Supongamos la pregunta:

```
?- puede_casarse_con(maria,X).
```

PROLOG recorre la base de datos en busca de un hecho que coincida con la cuestión planteada. Lo que haya es la regla

```
puede_casarse_con(X,Y) :- quiere_a(X,Y), hombre(X), mujer(Y).
```

produciéndose una coincidencia con la cabeza de la misma, y una instanciación de la variable X de la regla con el objeto 'maria'. Tendremos por lo tanto:

(1) puede_casarse_con(maria,Y) :- quiere_a(maria,Y), hombre(maria), mujer(Y).

A continuación, se busca una instanciación de la variable Y que haga cierta la regla, es decir, que verifique los hechos del cuerpo de la misma. La nueva meta será:

(2) quiere_a(maria,Y).

De nuevo PROLOG recorre la base de datos. En este caso encuentra un hecho que coincide con el objetivo:

quiere_a(maria,enrique).

Instanciando la variable Y con el objeto 'enrique'. Siguiendo el orden dado por la regla (1), quedan por probar dos hechos una vez instanciada la variable Y:

hombre(maria), mujer(enrique).

Se recorre de nuevo la base de datos, no hallando en este caso ninguna coincidencia con el hecho 'hombre(maria)'. Por lo tanto, PROLOG recurre a la vuelta atrás, desistanciando el valor de la variable Y, y retrocediendo con el fin de encontrar una nueva instanciación de la misma que verifique el hecho (2). Un nuevo recorrido de la base de hechos da como resultado la coincidencia con:

quiere_a(maria,susana).

Se repite el proceso anterior. La variable Y se instancia con el objeto 'susana' y se intentan probar los hechos restantes:

hombre(maria), mujer(susana).

De nuevo se produce un fallo que provoca la desinstanciación de la variable Y, así como una vuelta atrás en busca de nuevos hechos que coincidan con (2).

Una nueva reevaluación da como resultado la instanciación de Y con el objeto 'ana' (la última posible), y un nuevo fallo en el hecho 'hombre(maria)'.

Una vez comprobadas sin éxito todas las posibles instanciaciones del hecho (2), PROLOG da por imposible la regla (1), se produce de nuevo la vuelta atrás y una nueva búsqueda en la base de datos que tiene como resultado la coincidencia con la regla:

(3) puede_casarse_con(maria,Y) :- quiere_a(maria,Y), mujer(maria), hombre(Y).

Se repite todo el proceso anterior, buscando nuevas instanciaciones de la variable Y que verifiquen el cuerpo de la regla. La primera coincidencia corresponde al hecho

quiere_a(maria,enrique).

que provoca la instanciación de la variable Y con el objeto 'enrique'. PROLOG tratará de probar ahora el resto del cuerpo de la regla con las instanciaciones actuales:

mujer(maria), hombre(enrique).

Un recorrido de la base de datos, da un resultado positivo en ambos hechos, quedando probado en su totalidad el cuerpo de la regla (3) y por lo tanto su cabeza, que no es m s que una de las soluciones al objetivo inicial.

```
X = enrique
More (Y/N): y
```

De esta forma se generarán el resto de las soluciones, si las hubiera.

[NOTA: Algunas implementaciones PROLOG incorporan los comandos 'trace' y 'notrace' que activan y desactivan la salida por pantalla del proceso de búsqueda.]

PROLOG utiliza un mecanismo de búsqueda independiente de la base de datos. Aunque pueda parecer algo retorcido, es una buena estrategia puesto que garantiza el proceso de todas las posibilidades. Es útil para el programador conocer dicho mecanismo a la hora de depurar y optimizar los programas.

5 PROGRAMACION AVANZADA

Hasta ahora hemos visto los aspectos fundamentales de PROLOG necesarios para crear sencillos programas (ó sistemas expertos). A continuación se comentan algunos mecanismos que nos permitirán crear programas más avanzados.

5.1 OPERADORES ESPECIALES

El operador "corte": El operador corte, representado por el símbolo "!" nos da un cierto control sobre el mecanismo de deducción del PROLOG. Su función es la de controlar el proceso de reevaluación, limitándolo a los hechos que nos interesen. Supongamos la siguiente regla:

```
regla :- hecho1, hecho2, !, hecho3, hecho4, hecho5.
```

PROLOG efectuará reevaluaciones entre los hechos 1, 2 sin ningún problema, hasta que se satisface el hecho2. En ese momento se alcanza el hecho3, pudiendo haber a continuación reevaluaciones de los hechos 3, 4 y 5. Sin embargo, si el hecho3 fracasa, no se intentara de ninguna forma reevaluar el hecho2.

Una interpretación práctica del significado del corte en una regla puede ser que "si has llegado hasta aquí es que has encontrado la única solución a este problema y no hay razón para seguir buscando alternativas".

Aunque se suele emplear como una herramienta para la optimización de programas, en muchos casos marca la diferencia entre un programa que funciona y otro que no lo hace.

El operador "not": Se define de tal forma que el objetivo not(X) se satisface solo si fracasa la evaluación de X. En muchos casos, puede sustituir al operador corte, facilitando la lectura de los programas. Por ejemplo:

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

Equivale a:

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

Sin embargo, en términos de coste, el operador corte es más eficiente, ya que en el primer caso PROLOG intentará satisfacer b dos veces, y en el segundo, solo una.

5.2 ESTRUCTURAS DE DATOS: LISTAS

La lista es una estructura de datos muy común en la programación no numérica. Es una secuencia ordenada de elementos que puede tener cualquier longitud. Ordenada significa que el orden de cada elemento es significativo. Un elemento puede ser cualquier término e incluso otra lista. Se representa como una serie de elementos separados por comas y encerrados entre corchetes. Para procesar una lista, la dividimos en dos partes: la cabeza y la cola. Por ejemplo:

Lista	**Cabeza**	**Cola**
[a,b,c,d]	a	[b,c,d]
[a]	a	[ ] (lista vacía)
[ ]	no tiene	no tiene
[[a,b],c]	[a,b]	[c]
[a,[b,c]]	a	[[b,c]]
[a,b,[c,d]]	a	[b,[c,d]]

Para dividir una lista, utilizamos el símbolo "|". Una expresión con la forma [X | Y] instanciará X a la cabeza de una lista e Y a la cola.

Por ejemplo:

```
p([1,2,3]).
p([el,gato,estaba,[en,la,alfombra]]).
?- p([X|Y]).
X = 1,
Y = [2,3]
More (Y/N):y
X = el,
Y = [gato,estaba,[en,la,alfombra]]
```

5.3 LA RECURSIVIDAD

La recursividad es un mecanismo que da bastante potencia a cualquier lenguaje de programación. Veamos un ejemplo de programación recursiva que nos permitir determinar si un tomo es miembro de una lista dada:

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

La regla (1) es el caso base de la recursión. Se evaluará como cierta siempre que coincida la variable X con la cabeza de la lista que se pasa como argumento. En la regla (2) está la definición recursiva. X es miembro de una lista si lo es de la cola de esa lista (la cabeza se comprueba en la regla (1)).

La regla (1) es una simplificación de la regla: miembro(X,[Y|_]) :- X = Y.

La traza para el caso de 'miembro(b,[a,b,c]).' es la siguiente:

(1) miembro(b,[a,b,c]) :- b = a. → no.
(2) miembro(b,[a,b,c]) :- miembro(b,[b,c]).
(1) Miembro(b,[b,c]) :- b = b. → yes.

Si necesitamos conocer la longitud de una lista, emplearemos una función recursiva como la siguiente:

```
longitud([],0).
longitud([_|Y],L1) :- longitud(Y,L2), L1 is L2 + 1.
```

Otro ejemplo muy típico de función recursiva es el del factorial de un número:

```
factorial(0,1) :- !.
factorial(X,Y) :- X1 is X-1, factorial(X1,Y1), Y is X*Y1.
```

5.4 ENTRADA/SALIDA

PROLOG, al igual que la mayoría de lenguajes de programación modernos incorpora predicados predefinidos para la entrada y salida de datos. Estos son tratados como reglas que siempre se satisfacen.

write: Su sintaxis es:

```
write('Hello world.').
```

Las comillas simples encierran constantes, mientras que todo lo que se encuentra entre comillas dobles es tratado como una lista.

También podemos mostrar el valor de una variable, siempre que esté instanciada:

write(X).

nl: El predicado nl fuerza un retorno de carro en la salida. Por ejemplo:

write('linea 1'), nl, write('linea 2').

tiene como resultado:
linea 1
linea 2

read: Lee un valor del teclado. La lectura del comando read no finaliza hasta que se introduce un punto ".". Su sintaxis es:

read(X). (Instancia la variable X con el valor leído del teclado)

read(ejemplo).

Se evalúa como cierta siempre que lo tecleado coincida con la constante entre paréntesis (en este caso 'ejemplo').

6 ALGUNOS EJEMPLOS

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

```
/* 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)).
/* 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).
```