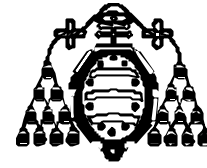




Universidad de Oviedo
Escuela Universitaria de Ingeniería
Técnica de Informática de Oviedo



Programación Práctica en Prolog

Jose E. Labra G.

<http://lsi.uniovi.es/~labra>

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática

Octubre - 1998




Tabla de Contenidos

1	Introducción	3
2	Hechos	3
3	Reglas	3
3.1	Reglas simples	3
3.2	Reglas con Variables	4
3.3	Reglas Recursivas	4
3.4	Utilización de funciones	4
3.5	Datos Compuestos	4
4	Unificación	4
5	Listas	6
5.1	Definición	6
5.2	Recorrer una lista	6
5.3	Recorrer una lista tomando elementos de dos en dos	7
5.4	Búsqueda de elementos en una lista	7
5.5	Generación de una lista mediante concatenación de otras dos	8
5.6	Descomposición de una lista en partes	8
5.7	Generación de una lista filtrando elementos de otra lista	9
5.8	Aplicación de un predicado a todos los elementos	9
5.9	Permutaciones de una lista	9
6	Aritmética	9
7	Aritmética con Listas	11
7.1	Obtener un valor a partir de una lista	11
7.2	Acumulación de Resultados	11
7.3	Combinación miembro a miembro de los elementos de dos listas	11
7.4	Generación de una lista a partir de un valor	11
7.5	Generación de listas por filtrado de elementos	12
7.6	Clasificación de listas	12
7.7	Creación de otra estructura a partir de una lista	13
7.8	Otras Estructuras Recursivas	13
8	Predicados Internos	14
8.1	Conversión de tipos	14
8.2	Chequeo de tipos	14
8.3	Inspección de estructuras	15
8.4	Predicados meta-lógicos	16
8.4.1	Chequeo de tipo	16
8.4.2	Comparación de términos no básicos	16
8.4.3	Conversión de Datos en Objetivos	17
8.5	Corte	18
8.5.1	Aplicaciones del Corte	18
8.6	Predicados Extra-lógicos	21
8.6.1	Entrada/Salida	21
8.6.2	Acceso a la Base de Datos	23
8.7	Predicados de Segundo Orden	24
9	Ejercicios Propuestos	25
10	Bibliografía Comentada	28
11	Índice	30

1 Introducción

Con estos apuntes se muestran las diferentes características del núcleo del lenguaje *Prolog* mediante sencillos ejemplos. Se ha realizado un tratamiento especial a las listas y a los diversos esquemas recursivos para trabajar con listas. El objetivo es ayudar al lector a identificar esquemas de programación que puedan servirle para resolver problemas similares. No se ha pretendido realizar una guía detallada sino una presentación práctica informal del lenguaje. El lector que esté interesado en profundizar en el lenguaje, deberá consultar los libros de la bibliografía.

Con el fin de facilitar la lectura, se utilizan los siguientes símbolos:

Clave: ✓	Punto importante a recordar
	Posibles ejercicios
	Comentarios avanzados
	Notas sobre portabilidad

En la presentación de ejemplos se utiliza una tabla escribiendo el código y las preguntas en la parte izquierda y los comentarios en la parte derecha.

2 Hechos

<pre>/* Relacion Progenitor */ progenitor(pilar,belen). progenitor(tomas,belen). progenitor(tomas,lucia). progenitor(belen,ana). progenitor(belen,pedro). progenitor(pedro,jose). progenitor(pedro,maria).</pre>	<p>Se describen una serie de hechos conocidos sobre una familia.</p> <p>✓ Sintaxis de Prolog: Constantes y predicados empiezan por minúscula. Los hechos acaban en punto. Variables comienzan por mayúscula.</p>
--	---

El programa anterior debe ser cargado en el sistema. Una vez cargado, es posible realizar preguntas:

<pre> ?- progenitor(pilar,belen). Yes</pre>	<p>El sistema puede responder:</p> <p>1.-<i>Yes</i>: Se puede deducir y el objetivo no tiene variables.</p>
<pre> ?- progenitor(pilar,lucia). No</pre>	<p>2.-<i>No</i>: No puede deducir a partir del programa La respuesta <i>No</i> indica que no se puede deducir. En la vida real, podría ocurrir que <i>pilar</i> fuese madre de <i>lucía</i> (parece lo más lógico) pero el sistema supone que todo lo que no está declarado es falso¹</p>
<pre> ?- progenitor(belen,X). X = ana ; X = pedro</pre>	<p>3.- <i>Substitución de Respuesta</i>: Se puede deducir y el objetivo tiene variables. Se indica el valor que toman las variables en la resolución del objetivo.</p> <p>En caso de que haya más de una solución. Cuando el sistema obtiene una solución espera a que el usuario introduzca <; > para solicitar más soluciones o <Enter> para finalizar. La búsqueda de otras soluciones se realiza mediante <i>backtracking</i></p>

3 Reglas

3.1 Reglas simples

<pre>/* Reglas */ cuida(belen,pedro):-paro(belen), bueno(pedro). /* 2 hechos más */ paro(belen). bueno(pedro).</pre>	<p>La regla equivale a: “<i>Belén cuida a Pedro</i> si <i>belén</i> está en <i>paro</i> y <i>Pedro</i> es bueno”. En lógica de predicados sería:</p> $\text{paro}(\text{belén}) \wedge \text{bueno}(\text{pedro}) \rightarrow \text{cuida}(\text{belén}, \text{pedro})$ <p>El símbolo :- podría interpretarse como: \leftarrow (si)</p>
---	---

¹ Presunción de mundo cerrado (*CWA-closed world assumption*)

3.2 Reglas con Variables

<pre>madre(X,Y):-mujer(X), progenitor(X,Y). mujer(pilar). mujer(belen). mujer(lucia). mujer(ana). mujer(maria). hombre(tomas). hombre(pedro). hombre(jose).</pre>	<p>Equivale a: <i>Para todo X e Y, si X es mujer y X es el progenitor de Y, entonces X es la madre de Y</i></p> <p>✓ En lógica de predicados: $\forall x \forall y (mujer(x) \wedge progenitor(x,y) \rightarrow madre(x,y))$</p>
<pre> ?- madre(belen,pedro). yes ?- madre(X,belen). X = pilar ; no ?- madre(belen,X). X = ana ; X = pedro no ?- madre(X,Y). X = pilar , Y = belen ; X = belen , Y = ana ; X = belen , Y = pedro ; no</pre>	<p>La programación lógica basa su modelo en la utilización de relaciones, lo cual permite que un mismo procedimiento sirva para diferentes propósitos dependiendo de qué variables están instanciadas²</p> <p>En el ejemplo, una misma regla <i>madre</i> sirve para:</p> <p>Comprobar si <i>belén</i> es madre de <i>pedro</i>. Calcular la <i>madre</i> de <i>belén</i> Calcular los hijos de <i>belén</i> Calcular parejas de madres/hijos</p> <p>✓ Obsérvese que las variables de los objetivos corresponden a cuantificadores existenciales, es decir:</p> <p>?- madre(belen,X) equivale a: ? $\exists x (madre(belen,x))$</p> <p>Téngase en cuenta que a la hora de resolver la pregunta <i>madre(belen,X)</i> con la cabeza de la regla <i>madre(X,Y)</i> es necesario renombrar la variable X</p> <p>El lenguaje Prolog realiza internamente un proceso de unificación que se describe en la página 4.</p>

3.3 Reglas Recursivas

<pre>antepasado(X,Y):-progenitor(X,Y). antepasado(X,Y):-progenitor(X,Z), antepasado(Z,Y).</pre>	<p>En general, en una definición recursiva, es necesario considerar 2 casos:</p> <p>Caso básico: Momento en que se detiene la computación</p> <p>Caso Recursivo: Suponiendo que ya se ha solucionado un caso más simple, cómo descomponer el caso actual hasta llegar al caso simple.</p> <p>✓ Tanto el caso básico como el caso recursivo no tienen porqué ser únicos (puede haber varios casos básicos y varios casos recursivos)</p>
<pre> ?- antepasado(belen,X). X = ana ; X = pedro ; X = jose ; X = maria ; no ?- antepasado(X,belen). X = pilar ; X = tomas ; no</pre>	<p>Las definiciones recursivas se resuelven de la misma forma que las reglas comunes. En la traza de este tipo de definiciones tiene especial importancia el renombramiento de variables.</p> <p>📖 Considerando la relación <i>progenitor</i> como un enlace entre dos nodos de un grafo. La relación <i>antepasado</i> indicaría si hay camino entre dos nodos del grafo dirigido acíclico formado por la relación <i>progenitor</i>. Este tipo de relaciones se utiliza en diversos contextos como la búsqueda de caminos entre ciudades, la simulación de movimientos de un autómatas, etc.</p> <p>🔗 Describir en <i>Prolog</i> una serie de caminos entre diversas ciudades y construir un predicado que indique</p>

² Se dice que una variable está instanciada si tiene un valor concreto (en realidad ya no sería una variable)

si 2 ciudades están conectadas.

3.4 Utilización de funciones

<pre>grande(pepe). grande(cabeza(juan)). grande(X):-mayor(X,Y). mayor(cabeza(X),cabeza(Y)):- progenitor(X,Y).</pre>	<p>Se utiliza la función: <i>cabeza(x)</i>="cabeza de x"</p> <p>El programa indica:</p> <p>"Pepe es grande, la cabeza de juan es grande, si X es mayor que Y, entonces X es grande, además: La cabeza de X es mayor que la de Y si X es el progenitor de Y"</p> <p>✓ Prolog no necesita declaraciones de tipos.</p>
<pre> ?- grande(X). X = pepe ; X = cabeza(juan) ; X = cabeza(pilar) ; X = cabeza(tomas) ; ...</pre>	<p>✓ Las variables en Prolog no tienen tipo, de ahí que la respuesta X puede ser una persona (<i>pepe</i>) o una cabeza (<i>cabeza(juan)</i>)</p>

3.5 Datos Compuestos

<pre>horizontal(seg(punto(X,Y), punto(X1,Y))). vertical(seg(punto(X,Y), punto(X,Y1))).</pre>	<p><i>punto(X,Y)</i> representa un punto de coordenadas (x,y)</p> <p><i>seg(p1,p2)</i> representa un segmento cuyos extremos son los puntos p1 y p2</p> <p>Los argumentos de una función pueden ser funciones</p>
<pre> ?- horizontal(seg(punto(1,2),punto(3,2))). Yes ?- horizontal(seg(punto(1,2),P)). P = punto(_47796,2) ?- horizontal(P),vertical(P). P=seg(punto(_29128,_29130),punto(_29128,_29130))</pre>	<p>P = punto(_47796,2) indica que P es un punto cuya primera coordenada es una variable sin instanciar³ y cuya segunda coordenada es 2</p> <p>La última respuesta indica que para que un segmento sea vertical y horizontal a la vez, sus coordenadas deben ser las mismas (los números de las variables X e Y coinciden)</p>

4 Unificación

Durante la resolución de objetivos, el sistema Prolog debe realizar la unificación entre los objetivos y las cabezas de las reglas o los hechos. De forma simplificada, el algoritmo de unificación consiste en:

- 1.- Inicializar σ = sustitución vacía
 - 2.- Si al aplicar σ a las 2 expresiones, éstas son iguales, finalizar y devolver σ
 - 3.- Buscar de izquierda a derecha las primeras subexpresiones diferentes:
 - Si dichas subexpresiones están formadas por una variable v y un término t (tal que $v \notin t$)⁴
 - Actualizar σ con el resultado de substituir v por t
 - Volver a 2
- En caso contrario Finalizar indicando que las expresiones no unifican

<pre> ?- f(X,X)=f(a,Y). X = a Y = a ?- f(X,X)\=f(a,Y). no ?- p(f(X),g(Z,X))=p(Y,g(Y,a)). X = a , Z = f(a)</pre>	<p>El operador '=' se cumple si sus argumentos unifican. El operador '\=' se cumple si sus argumentos no unifican.</p> <p>Puesto que Prolog no tiene chequeo de ocurrencias, se produce un error.</p>
--	---

³ El formato de las variables sin instanciar varía de un sistema Prolog a otro. En estos apuntes, se utiliza un número de 4 dígitos aleatorio precedido del carácter '_'

⁴ La condición $v \notin t$ se conoce como *chequeo de ocurrencias* (*occur check*) y no es implementada por la mayoría de los sistemas Prolog debido a su complejidad. Esto hace que en determinadas ocasiones, el sistema se meta en un bucle infinito tratando de unificar 2 términos

<pre>Y = f(a) ?- p(X,X)=p(Y,f(Y)). Error ..., System Stack Full</pre>	<p>✓ Se produce error porque Prolog no implementa chequeo de ocurrencias y el algoritmo de unificación entra en un bucle infinito. En la página 17 se implementa el algoritmo de unificación con chequeo de ocurrencias.</p>
---	--

5 Listas

5.1 Definición

Las listas son una de las estructuras de datos más utilizadas en los programas en lenguaje Prolog y responden a la siguiente definición:

[] es una lista

[] denota la lista vacía

Si Xs es una lista entonces $[X|Xs]$ es una lista

$[X|Xs]$ denota una lista de cabeza X y cola Xs

<pre> ?- [1,2,3] = [1 [2 [3 []]]]. yes ?- [X Xs]=[1,2,3]. X = 1 , Xs = [2,3] ?- [X Xs]=[1]. X = 1 , Xs = [] ?- [X,Y Ys]=[1,2,3]. X = 1 , Y = 2 , Ys = [3] ?- X = "abc". X = [97,98,99]</pre>	<p>El sistema convierte internamente entre: $[X,Y,Z]$ y $[X [Y [Z []]]]$ $[X,Y Z]$ y $[X [Y Z]]$</p> <p>✓ Se utiliza una sintaxis especial para las listas de caracteres del tipo "abc" que se convierten internamente en listas de los códigos ASCII correspondientes.</p>
---	---

<pre>lista([]). lista([X Xs]):-lista(Xs). ?- lista([1,2]). yes ?- lista(X). X = [] ; X = [_2440] ; X = [_2440,_4572] ; X = [_2440,_4572,_6708] ; X = [_2440,_4572,_6708,_8848] ...</pre>	<p>Puesto que las listas son una estructura de datos definida de forma recursiva. La mayoría de las definiciones identificarán el caso básico con la lista vacía y el caso recursivo con $[X Xs]$</p> <p>No todas las definiciones recursivas con listas utilizan esos dos casos</p> <p>Como convenio, los nombres de variables en singular (X, Y, \dots) indican elementos, mientras que en plural indican listas (Xs, Ys, \dots). Las listas de listas se suelen representar como Xss</p> <p>✓ La llamada <code>lista(X)</code> genera por <i>backtracking</i> listas de cualquier tipo de elementos.</p>
---	---

5.2 Recorrer una lista

<pre>hombres([]). hombres([X Xs]):-hombre(X), hombres(Xs). noPertenece(X, []). noPertenece(X, [Y Ys]):-X\=Y, noPertenece(X, Ys). ?- hombres([jose,tomas,pedro]). yes ?- hombres([jose,pilar,tomas]).</pre>	<p><i>hombres(Xs)</i>:- Todos los X de Xs cumplen <i>hombre(X)</i></p> <p><i>noPertenece(X,Xs)</i>:- El elemento X no pertenece a la lista Xs (se comprueba que no unifica con ningún elemento de Xs)</p> <p>En las definiciones recursivas, conviene escribir el caso básico antes que el caso recursivo ya que si la definición unifica con ambos casos y</p>
---	--

<pre>no ?- noPertenece(pilar,[jose,pilar,tomas]). no ?- noPertenece(luis,[jose,pilar,tomas]). yes</pre>	<p>estuviese pusiese el caso recursivo antes, el sistema entraría en un bucle infinito.</p> <p>De la misma forma, es conveniente que la llamada recursiva sea la última llamada de la definición recursiva (muchos sistemas optimizan las definiciones así escritas)</p>
---	--

5.3 Recorrer una lista tomando elementos de dos en dos

<pre>todosIguales([]). todosIguales([X]). todosIguales([X,X Xs]):-todosIguales([X Xs]). ?- todosIguales([1,2,3]). no ?- todosIguales([1,1,1]). yes ?- todosIguales(X). X = [] ; X = [_25576] ; X = [_27704,_27704] ; X = [_27704,_27704,_27704] ; X = [_27704,_27704,_27704,_27704]</pre>	<p><i>todosIguales(Xs):-</i> Los elementos de Xs son todos iguales</p> <p>Obsérvese que hay dos casos básicos y que el caso recursivo no es de la forma [X Xs].</p> <p>Obsérvese que los tres casos se excluyen. Como regla general, conviene que los casos de una definición se excluyan. Cuando los casos no se excluyen, al realizar <i>backtracking</i> el sistema puede obtener respuestas distintas de las esperadas.</p> <p>✎ Describir la respuesta ante: ?- todosIguales([2,X,Y]).</p>
--	---

5.4 Búsqueda de elementos en una lista

<pre>pertenece(X,[X Xs]). pertenece(X,[Y Ys]):-pertenece(X,Ys). ?- pertenece(pedro,[jose,pedro,tomas]). yes ?- pertenece(jose,[]). no ?- pertenece(X,[jose,pedro,tomas]). X = jose ; X = pedro ; X = tomas ; no ?- pertenece(jose,X). X = [jose _6617] ; X = [_8808,jose _8817] ; X = [_8808,_11012,jose _11021] ; X = [_8808,_11012,_13230,jose _13239] ...</pre>	<p><i>pertenece(X,Xs):-</i> X está en la lista Xs</p> <p>El caso básico no es la lista vacía. Además, los casos no se excluyen. Una lista que encaje con la primera definición, lo hará con la segunda.</p> <p>La relación <i>pertenece</i> es un claro ejemplo de la flexibilidad de las definiciones en Prolog ya que se puede utilizar para:</p> <ol style="list-style-type: none"> 1.- Chequear si un elemento pertenece a una lista 2.- Obtener todos los elementos de una lista por <i>backtracking</i> 3.- Obtener listas con un elemento X en primera, segunda, ... n-ésima posición.
<pre>elimina(X,[X Xs],Xs). elimina(X,[Y Ys],[Y Zs]):-elimina(X,Ys,Zs). ?- elimina(1,[1,2,1,3],V). V = [2,1,3] ; V = [1,2,3] ; no ?- elimina(1,V,[2,3]).</pre>	<p><i>eliminna(X,Ys,Zs):-</i> Zs contiene todas las listas resultantes de eliminar el elemento X de Ys</p> <p>✓ El predicado <i>selecciona</i> sirve también para diversos propósitos: Borrar elementos de una lista Insertar un elemento en diferentes posiciones de una lista</p>

<pre>V = [1,2,3] ; V = [2,1,3] ; V = [2,3,1] ; no</pre>	<p>☞ Describir el comportamiento del predicado si se añade $X \setminus Y$ en la segunda definición.</p>
<pre>algunHombre(Xs):-pertenece(X,Xs), hombre(X). ?- algunHombre([jose,pilar,tomas]). yes</pre>	<p><i>algunHombre(Xs)</i>:- alguno de los elementos X de Xs cumple la relación <i>hombre(X)</i></p>

5.5 Generación de una lista mediante concatenación de otras dos

<pre>concat([],Ys,Ys). concat([X Xs],Ys,[X Zs]):-concat(Xs,Ys,Zs). ?- concat([1,2],[3,4],V). V = [1,2,3,4] ?- concat([1,2],X,[1,2,3,4]). X = [3,4] ?- concat(X,Y,[1,2,3,4]). X = [] , Y = [1,2,3,4] ; X = [1] , Y = [2,3,4] ; X = [1,2] , Y = [3,4] ; X = [1,2,3] , Y = [4] ; X = [1,2,3,4] , Y = [] ; no</pre>	<p><i>concat(Xs,Ys,Zs)</i>:- Zs es el resultado de concatenar las listas Xs e Ys</p> <p>✓ Obsérvese la flexibilidad de la definición de <i>concat</i> que permite una gran variedad de usos dependiendo de qué variables están instanciadas en la llamada.</p>
---	--

5.6 Descomposición de una lista en partes

Gracias a la flexibilidad de las relaciones lógicas, el predicado *concat* se utiliza en la definición de una gran cantidad de predicados.

<pre>prefijo(Xs,Ys):- concat(Xs,Bs,Ys). sufijo(Xs,Ys):- concat(As,Xs,Ys). sublista(Xs,Ys):- concat(AsXs,Bs,Ys), concat(As,Xs,AsXs). ?- prefijo([1,2],[1,2,3,4]). yes ?- sufijo([3,4],[1,2,3,4]). yes ?- pertenece1(1,[2,1,3]). yes ?- sublista([2,3],[1,2,3,4]). yes</pre>	<p>Definir los predicados <i>prefijo</i>, <i>sufijo</i> y <i>sublista</i> de forma recursiva sin la ayuda del predicado <i>concat</i></p> <p>Definir mediante <i>concat</i> los predicados: <i>pertenece(X,Xs)</i>:- X es un elemento de Xs <i>reverse(Xs,Ys)</i>:- Ys es Xs con los elementos en orden inverso <i>adyacentes(X,Y,Xs)</i>:- X e Y están en posiciones consecutivas en Xs <i>ultimo(Xs,X)</i>:- X es el último elemento de Xs <i>primeros(Xs,Ys)</i>:- Ys es el resultado de eliminar el último elemento a Xs</p>
--	---

5.7 Generación de una lista filtrando elementos de otra lista

<pre> filtraHombres([],[]). filtraHombres([X Xs],[X Ys]):-hombre(X), filtraHombres(Xs,Ys). filtraHombres([X Xs],Ys):- mujer(X), filtraHombres(Xs,Ys). ?- filtraHombres([jose,pilar,tomas],V). V = [jose,tomas] sinDuplicados([],[]). sinDuplicados([X Xs],[X Ys]):-noPertenece(X,Xs), sinDuplicados(Xs,Ys). sinDuplicados([X Xs],Ys):- pertenece(X,Xs), sinDuplicados(Xs,Ys). ?- sinDuplicados([1,1,2,3,2,1,4],V). V = [3,2,1,4] ; </pre>	<p><i>filtraHombres(Xs,Ys)</i>:- Ys contiene todos los hombres de la lista Xs.</p> <p>☞ Obsérvese qué ocurre cuando se solicitan más soluciones por <i>backtracking</i> o cuando se pregunta: <code>?- filtraHombres(V,[jose,pilar]).</code></p> <p><i>sinDuplicados(Xs,Ys)</i>:-Ys contiene los elementos de Xs eliminando elementos duplicados.</p> <p>✓ Ambas definiciones son poco eficientes</p> <p>☞ Construir el predicado <i>sinDuplicados</i> para que la lista resultado contenga los elementos en el mismo orden en que aparecen</p>
--	--

5.8 Aplicación de un predicado a todos los elementos

<pre> edad(pilar,85). edad(tomas,90). edad(belen,67). edad(lucia,64). edad(ana,34). edad(pedro,36). edad(jose,10). edades([],[]). edades([X Xs],[Y Ys]):-edad(X,Y), edades(Xs,Ys). ?- edades([jose,pilar,tomas],V). V = [10,85,90] ?- edades([jose,dedo,tomas],V). no ?- edades(V,[10,85]). V = [jose,pilar] </pre>	<p><i>edades(Xs,Ys)</i>:- Ys contiene las edades de los elementos de Xs</p> <p>✓ Si alguno de los elementos no tiene <i>edad</i>, el predicado falla.</p>
--	---

5.9 Permutaciones de una lista

<pre> permutacion([],[]). permutacion(Xs,[X Ys]):-elimina(X,Xs,Zs), permutacion(Zs,Ys). ?- permutacion([1,2,3],V). V = [1,2,3] ; V = [1,3,2] ; V = [2,1,3] ; V = [2,3,1] ; V = [3,1,2] ; V = [3,2,1] ; no </pre>	<p><i>permutacion(Xs,Ys)</i>:- Ys es una permutación de la lista Xs</p>
---	---

6 Aritmética

Con el fin de aprovechar las posibilidades aritméticas de los computadores convencionales, el lenguaje *Prolog* contiene una serie de predicados de evaluación aritmética.

Al encontrar un objetivo de la forma “*? X is E*” el sistema:

Evalúa la expresión *E* hasta obtener un valor aritmético *v* (si no puede, devuelve un error)

El objetivo se cumple si *X* unifica con *v*

La expresión E puede contener los operadores aritméticos clásicos (+, -, *, mod, etc.) y valores numéricos. Si contiene variables, éstas deben estar instanciadas a un valor numérico en el momento de la evaluación.

<pre> ?- X is 3+5. X = 8 ?- X is pepe. Error al evaluar ?- 8 is 3+5. yes ?- 4 is 3+5. no ?- X is 4/0. Error aritmético ?- X is X + 1. Error al evaluar ?- X = 3, Y is X + 5. X = 3 , Y = 8 ?- X=0, X is X + 1. no ?- X = 3 + 5. X = 3 + 5</pre>	<p>La evaluación de 3+5 se realiza internamente en una instrucción del procesador.</p> <p>Si la expresión no resulta en un valor aritmético, se obtiene un error.</p> <p>Al evaluar pueden producirse errores.</p> <p>El operador “is” no es nunca un operador de asignación como el := de Pascal. En general, una expresión del tipo X is X+1 no tiene sentido en <i>Prolog</i>.</p> <p>Describir en qué situaciones “X is X + 1” es un error y en qué situaciones falla sin más.</p> <p>✓ El operador = unifica sus argumentos pero no evalúa.</p>
--	---

<pre>par(X) :- 0 is X mod 2. impar(X) :- 1 is X mod 2. suma(X,Y,Z):- Z is X + Y. ?- par(3). no ?- par(4). yes ?- suma(2,3,V). V = 5 ?- suma(2,V,5). Error al evaluar</pre>	<p>Con el operador <i>is</i> se pierde la flexibilidad de las relaciones lógicas. Lo ideal es que al evaluar suma(2,V,5), el sistema devolviese V=3</p> <p>☞ ¿Por qué se obtiene error al evaluar suma(2,V,5) y no se obtiene error al evaluar par(4)?</p>
--	--

Además del predicado *is*, los predicados de comparación realizan una evaluación aritmética de sus argumentos y se cumplen si los valores obtenidos cumplen las relaciones correspondientes:

<pre> ?- 3+5 > 2+6. no ?- 3+5 >= 2+6. yes ?- 3+5 < 2+6. yes ?- 3+5 <= 2+6. yes ?- 3+5 := 2+6. yes ?- 3+5 \= 2+6. no</pre>	<p>📖 Para la comparación se utiliza el operador “<=” en lugar del más habitual “<” debido a que éste último se asemeja a una doble flecha y se reserva para otros propósitos</p> <p>☞ Describir la diferencia entre el comportamiento de los operadores: “=”, “is” y “:=”</p>
<pre>fact(0,1). fact(N,F):-N > 0, N1 is N - 1, fact(N1,F1), F is N * F1.</pre>	<p>fact(N,F):-F es el factorial de N</p> <p>✓ La versión de factorial aquí presentada es poco eficiente.</p>

```
| ?- fact(5,V).
V = 120
```

7 Aritmética con Listas

7.1 Obtener un valor a partir de una lista

```
sum([],0).
sum([X|Xs],S):-sum(Xs,Sc), S is Sc + X.

long([],0).
long([X|Xs],L):-long(Xs,Lc), L is Lc + 1.

prod([],1).
prod([X|Xs],P):-prod(Xs,Pc), P is Pc * X.

| ?- sum([1,2,3,4],V).
V = 10

| ?- long([1,2,3,4],V).
V = 4

| ?- prod([1,2,3,4],V).
V = 24

%% Version optimizada de sum
sum1(Xs,S):-sumAux(Xs,0,S).

sumAux([],S,S).
sumAux([X|Xs],Sa,S):-Sn is X + Sa,
sumAux(Xs,Sn,S).
```

📖 Obsérvese la similitud entre las tres definiciones. En algunos lenguajes se utilizan construcciones de orden superior que permiten utilizar una única definición parametrizada por las operaciones y constantes⁵.

✓ Las definiciones ofrecidas no aprovechan la *optimización de la recursividad de cola*. Consiste en que el último objetivo de una definición recursiva sea el predicado que se está definiendo. Los sistemas con dicha optimización permiten que las definiciones recursivas se comporten de forma similar a un bucle en un lenguaje imperativo.

sum1 es una versión optimizada de *sum*

✓ La definición de *sum1* sigue el patrón de *acumulación de resultados* que se ve a continuación

7.2 Acumulación de Resultados

A diferencia de los lenguajes imperativos, *Prolog* utiliza variables lógicas. En el momento en que una variable lógica es instanciada, dicho valor no puede modificarse. De esta forma, no es posible utilizar variables globales cuyo valor se modifique durante la resolución del objetivo. Existen ciertos algoritmos que requieren la utilización de un estado que almacena resultados intermedios. Para implementar dichos algoritmos es necesario utilizar un predicado auxiliar con un argumento extra que almacenará el estado que se modifica.

La definición de *sum1* de la sección anterior sigue el patrón mencionado.

```
sumAcum(Xs,Ys):-sumAc(Xs,0,Ys).

sumAc([],S,[]).
sumAc([X|Xs],Sa,[Sp|Ys]):-Sp is X + Sa,
sumAc(Xs,Sp,Ys).

| ?- sumAcum([1,2,3,4],V).
V = [1,3,6,10]
```

$\text{sumAcum}(Xs,Ys) :- y_j = \sum_{i=1}^j x_i \text{ para cada } y_j \in Ys$

🔧 Construir la definición del predicado *maximo(Xs,M)* que se cumple si *M* es el máximo de los elementos de *Xs*

7.3 Combinación miembro a miembro de los elementos de dos listas

```
prodEscalar(Xs,Ys,P):-pEsc(Xs,Ys,0,P).

pEsc([],[],P,P).
pEsc([X|Xs],[Y|Ys],Pa,Pr):-Pn is Pa + X * Y,
pEsc(Xs,Ys,Pn,Pr).

| ?- prodEscalar([1,2,3],[4,5,6],P).
P = 32
```

$\text{prodEscalar}(Xs,Ys,P) :- P$ es el producto escalar de los vectores *Xs* e *Ys* ($P = \sum x_i y_i$)

✓ En esta definición se utiliza además el patrón de *acumulación de resultados* anterior.

7.4 Generación de una lista a partir de un valor

Es posible generar una lista mediante la descomposición de un valor. En el primer ejemplo, se descompone un número natural hasta llegar a cero, en el segundo, se descompone un intervalo hasta que los extremos son iguales.

⁵ En este sentido, los lenguajes funcionales se caracterizan por utilizar funciones de orden superior, permitiendo una mayor reutilización de código.

<pre> repite(0,X,[]). repite(N,X,[X Xs]):-N > 0, N1 is N - 1, repite(N1,X,Xs). intervalo(X,X,[X]). intervalo(X,Y,[X Xs]):-X < Y, Z is X + 1, intervalo(Z,Y,Xs). ?- repite(3,a,V). V = [a,a,a] ; ?- intervalo(1,5,V). V = [1,2,3,4,5] ; </pre>	<p><i>repite(N,X,Xs)</i>:- Xs con N elementos de valor X</p> <p><i>intervalo(X,Y,Xs)</i>:-Xs es una lista creciente cuyo primer valor es X y su último valor Y</p>
---	--

7.5 Generación de listas por filtrado de elementos

Los siguientes ejemplos muestran cómo se puede generar una o varias listas filtrando los elementos de otras listas.

- En *pares*, la condición de filtrado (ser par) indica si se inserta o no el elemento
- En *inserta* la condición de filtrado (ser menor o mayor que el valor X) indica si se selecciona el elemento Y de la lista que se recorre o el elemento X a insertar
- En *particion* la condición de filtrado indica en qué lista se inserta
- En el último caso, se recorren dos listas y la condición indica de cuál de las listas seleccionar el valor

<pre> pares([],[]). pares([X Xs],[X Ys]):-par(X), pares(Xs,Ys). pares([X Xs],Ys):-impar(X),pares(Xs,Ys). ?- intervalo(1,5,V), pares(V,P). V = [1,2,3,4,5] , P = [2,4] ; </pre>	<p><i>pares(Xs,Ys)</i>:-Ys contiene los elementos pares de Xs. Se filtran los elementos pares</p>
<pre> inserta(X,[],[X]). inserta(X,[Y Ys],[X,Y Ys]):-X < Y. inserta(X,[Y Ys],[Y Zs]):-X >= Y, inserta(X,Ys,Zs). ?- inserta(3,[1,2,4],V). V = [1,2,3,4] ; </pre>	<p><i>inserta(X,Xs,Ys)</i>:-Ys es el resultado de insertar X en la posición adecuada de la lista ordenada Xs de forma que Ys se mantenga ordenada.</p>
<pre> particion(X,[Y Ys],[Y Ps],Gs):- Y < X, particion(X,Ys,Ps,Gs). particion(X,[Y Ys],Ps,[Y Gs]):- Y >= X, particion(X,Ys,Ps,Gs). particion(X,[],[],[]). ?- particion(3,[2,4,1,5],X,Y). X = [2,1] , Y = [4,5] ; </pre>	<p><i>particion(X,Xs,Ps,Gs)</i>:-Ps contiene los elementos de Xs más pequeños que X y Gs los elementos de Xs más grandes o iguales que X.</p>
<pre> mezcla(Xs,[],Xs). mezcla([],Y Ys,[Y Ys]). mezcla([X Xs],[Y Ys],[X Zs]):-X < Y, mezcla(Xs,[Y Ys],Zs). mezcla([X Xs],[Y Ys],[Y Zs]):-X >= Y, mezcla([X Xs],Ys,Zs). ?- mezcla([1,3,5],[2,4,6],V). V = [1,2,3,4,5,6] ; </pre>	<p><i>mezcla(Xs,Ys,Zs)</i>:-Zs es una lista ordenada formada a partir de las listas ordenadas Xs e Ys</p> <p>✓ Obsérvese que esta definición consta de dos casos básicos y dos recursivos. El segundo caso básico podría haber sido: <i>mezcla([],Ys,Ys)</i> pero no sería excluyente con el primero cuando Xs e Ys están vacías.</p>

7.6 Clasificación de listas

<pre> ordenada([]). ordenada([X]). </pre>	<p><i>ordenada(Xs)</i>:- Los elementos de Xs están en</p>
---	---

<pre>ordenada([X,Y Ys]):-X<Y, ordenada([Y Ys]). ordenaBruto(Xs,Ys):-permutacion(Xs,Ys), ordenada(Ys).</pre>	<p>orden ascendente</p> <p>☞ ¿qué patrón sigue la definición de <i>ordenada</i>?</p> <p><i>ordenaBruto(Xs,Ys)</i>:-Ys contiene los elementos de Xs en orden ascendente (utiliza el famoso algoritmo de la <i>fuerza bruta</i>)</p>
<pre>ordenaIns([],[]). ordenaIns([X Xs],Ys):-ordenaIns(Xs,XsO), inserta(X,XsO,Ys).</pre>	<p><i>ordenaIns(Xs,Ys)</i>:-Ys contiene los elementos de Xs en orden ascendente (utiliza el algoritmo de inserción)</p>
<pre>quicksort([],[]). quicksort([X Xs],Ys):-particion(X,Xs,Ps,Gs), quicksort(Ps,PsO), quicksort(Gs,GsO), concat(PsO,[X GsO],Ys). ?- quicksort([3,1,2,4],V). V = [1,2,3,4] ;</pre>	<p><i>quicksort(Xs,Ys)</i>:-Ys contiene los elementos de Xs en orden ascendente (algoritmo <i>quicksort</i>⁶)</p> <p>☞ Escribir la definición del algoritmo de clasificación <i>mergesort</i> (de forma similar a <i>quicksort</i>, divide la lista en dos mitades de igual tamaño, las clasifica y las mezcla).</p>

7.7 Creación de otra estructura a partir de una lista

En muchas ocasiones puede ser necesario convertir una lista en otra estructura.

Supóngase que se desea comprimir una lista con muchos valores repetidos en una estructura que informe del tipo de valor y del número de veces que aparece⁷. Para ello, se utiliza un término estructurado de la forma “cod(X,N)” indicando que el término *X* aparece *N* veces.

<pre>comprime([],[]). comprime([X Xs],Ys):-comp(Xs,X,1,Ys). comp([],C,N,[cod(C,N)]). comp([X Xs],X,N,Ys):-N1 is N+1, comp(Xs,X,N1,Ys). comp([X Xs],Y,N,[cod(Y,N) Ys]):- X\=Y, comp(Xs,X,1,Ys).</pre> <p> ?- comprime([a,a,b,b,b,c,b,b],V).</p> <p>V = [cod(a,2),cod(b,3),cod(c,1),cod(b,2)]</p>	<p><i>comprime(Xs,Ys)</i>:-Ys es una lista cuyos elementos tienen la forma <i>cod(X,N)</i> donde <i>X</i> indica un elemento de Xs y <i>N</i> indica el número de veces que se repite.</p> <p>☞ Construir un predicado que calcule la frecuencia de cada elemento de una lista</p>
---	--

7.8 Otras Estructuras Recursivas

Las estructuras recursivas forman una parte fundamental de los lenguajes de programación⁸.

A continuación se definen unos predicados de conversión entre una lista y árbol binario de búsqueda.

Los árboles binarios se definen por inducción de la siguiente forma:

vacio ∈ *ArbolesBinarios*

Si *A1* y *A2* ∈ *ArbolesBinarios* entonces *rama(X,A1,A2)* ∈ *ArbolesBinarios*

Ejemplo: *rama(2,rama(1,vacio,vacio),vacio)*.

Los árboles de búsqueda se caracterizan porque el valor de cualquier nodo del árbol es mayor que el valor de los nodos a su izquierda y menos que el valor de los nodos a su derecha.

<code>insertArbol(X,vacio,rama(X,vacio,vacio)).</code>	<code>insertArbol(X,A,An):-An es el árbol de</code>
--	---

⁶ El algoritmo *quicksort* se caracteriza porque, aunque en el caso peor tiene complejidad $O(n^2)$, en el caso medio tiene complejidad $O(n \log_2 n)$

⁷ Este esquema de codificación se conoce como *run-length encoded* y es muy utilizado para comprimir imágenes

⁸ Las listas son un caso especial de estructura recursiva

<pre> insertArbol(X,rama(X,A1,A2),rama(X,A1,A2)). insertArbol(X,rama(Y,A1,A2),rama(Y,A1n,A2)):- X < Y, insertArbol(X,A1,A1n). insertArbol(X,rama(Y,A1,A2),rama(Y,A1,A2n)):- X > Y, insertArbol(X,A2,A2n). listArbol(Xs,A):-creaArbol(Xs,vacio,A). creaArbol([],A,A). creaArbol([X Xs],Ao,Ar):-insertArbol(X,Ao,An), creaArbol(Xs,An,Ar). ?- listArbol([2,1,3],V). V = rama(2,rama(1,vacio,vacio),rama(3,vacio,vacio)) busca(X,rama(X,_,_)). busca(X,rama(Y,A1,A2)):-X < Y, busca(X,A1). busca(X,rama(Y,A1,A2)):-X > Y, busca(X,A2). nodos(vacio,[]). nodos(rama(X,A1,A2),Xs):-nodos(A1,Xs1), nodos(A2,Xs2), concat(Xs1,[X Xs2],Xs). ordenArbol(Xs,XsO):-listArbol(Xs,A), nodos(A,XsO). ?- ordenArbol([2,1,3,4],V). V = [1,2,3,4] ; </pre>	<p>búsqueda resultante de insertar X en A</p> <p><i>listaArbol(Xs,A)</i>:- A es el árbol de búsqueda creado a partir de la lista Xs</p> <p>☞ Indicar qué patrón de recursividad se utiliza en la definición de <i>listArbol</i></p> <p><i>busca(X,A)</i>:-Se cumple si X está en el árbol A</p> <p><i>nodos(A,Xs)</i>:- Xs son los nodos del árbol A.</p> <p>📖 El predicado <i>nodos</i> realiza el recorrido del árbol empezando por los nodos de la izquierda, luego el nodo central y finalmente los de la derecha. Este recorrido se conoce como recorrido <i>inorden</i>. Existen otros recorridos como <i>preorden</i> y <i>postorden</i></p> <p>✓ Obsérvese que con los árboles podrían definirse patrones recursivos similares a los definidos para las listas⁹.</p>
--	---

8 Predicados Internos

8.1 Conversión de tipos

En general, *Prolog* no contiene chequeo de tipos en tiempo de compilación, siendo necesario utilizar una serie de predicados predefinidos que chequean el tipo de sus argumentos.

Predicado	Condición para que se cumpla	Preguntas Simples
<i>atom_chars(A,Cs)</i>	Cs es la lista de caracteres que representa el átomo A	<pre> ?- atom_chars(abc,V). V = [97,98,99] ?- atom_chars(V,"abc"). V = abc </pre>
<i>number_chars(N,Cs)</i>	Cs es la lista de caracteres que representa el número N	<pre> ?- number_chars(123,V). V = [49,50,51] ?- number_chars(V,"123"). V = 123 </pre>
<i>number_atom(N,A)</i>	A es el átomo que representa el número N	<pre> ?- number_atom(123,V). V = '123' ?- number_atom(V,'123'). V = 123 </pre>

8.2 Chequeo de tipos

En general, *Prolog* no contiene chequeo de tipos en tiempo de compilación, siendo necesario utilizar una serie de predicados predefinidos que chequean el tipo de sus argumentos.

Predicado	Condición para que se cumpla	Ejemplos
<i>integer(X)</i>	X es un entero.	4

⁹ Actualmente, se investiga la posibilidad de utilizar lenguajes que definan automáticamente dichos patrones de recursividad para los diferentes tipos de datos definidos por el usuario.

<i>float(X)</i>	<i>X</i> es un flotante.	4.5
<i>number(X)</i>	<i>X</i> es un número.	4 4.5
<i>atom(X)</i>	<i>X</i> es un átomo.	pepe []
<i>atomic(X)</i>	<i>X</i> es un átomo o un número.	pepe 4 4,5 []
<i>compound(X)</i>	<i>X</i> es un término compuesto.	padre(luis,juan) [3]
<i>list(X)</i>	<i>X</i> es una lista.	[] [3] "3,2"

<pre> sumaLogica(X,Y,Z):-integer(X), integer(Y), Z is X + Y. sumaLogica(X,Y,Z):-integer(Y), integer(Z), X is Z - Y. sumaLogica(X,Y,Z):-integer(X), integer(Z), Y is Z - X. ?- sumaLogica(2,3,V). V = 5 ; ?- sumaLogica(3,V,5). V = 2 </pre>	<p><i>sumaLogica(X,Y,Z):-Z</i> es igual a <i>X + Y</i> Define un predicado aritmético que admite la flexibilidad de las relaciones lógicas.</p>
<pre> alisar([],[]). alisar([X Xs],Ls):-list(X), alisar(X,Ls1), alisar(Xs,Ls2), concat(Ls1,Ls2,Ls). alisar([X Xs],[X Ls]):-atomic(X), alisar(Xs,Ls). ?- alisar([[1,2],[3],[[4]]],V). V = [1,2,3,4] </pre>	<p><i>alisar(Xs,Ys):-Ys</i> contiene una lista con los elementos de las listas de <i>Xs</i></p> <p>✓ Obsérvese que se utiliza el patrón de generación de una lista por filtrado de elementos.</p>

8.3 Inspección de estructuras

Predicado	Condición para que se cumpla	Ejemplos
<i>functor(T,F,A)</i>	<i>T</i> es un término compuesto de functor <i>F</i> y aridad <i>A</i>	<pre> ?- functor(f(a,b),F,A). F = f , A = 2 ?- functor(T,f,2). T = f(_4648,_4650) </pre>
<i>arg(N,T,A)</i>	<i>A</i> es el argumento <i>N</i> -ésimo del término <i>T</i>	<pre> ?- arg(2,f(a,b,c),A). A = b ?- arg(2,f(a,X,c),b). X = b </pre>
<i>T =.. Ls</i>	<i>Ls</i> es una lista cuyo primer elemento es el functor de <i>T</i> y cuyo resto de elementos son los argumentos de <i>T</i>	<pre> ?- f(a,b) =.. Ls. Ls = [f,a,b] ?- T =.. [f,a,b]. T = f(a,b) </pre>

<pre> subTer(T,T). subTer(S,T):-compound(T), T =.. [F Args], subTerL(S,Args). </pre>	<p><i>subTer(S,T):-</i> Se cumple si <i>S</i> es un subtérmino de <i>T</i></p> <p>📖 El predicado '=<i>..</i>' también se conoce</p>
--	---

<pre> subTerL(S,[A As]):- subTer(S,A). subTerL(S,[A As]):- subTerL(S,As). ?- subTer(g(b),f(a,g(b),c)). yes ?- subTer(X,f(a,g(b))). X = f(a,g(b)) ; X = g(b) ; X = b ; X = a ; subTerm(T,T). subTerm(S,T):-compound(T), functor(T,F,A), subTermA(A,S,T). subTermA(N,S,T):-arg(N,T,Ta), subTerm(S,Ta). subTermA(N,S,T):-N > 1, N1 is N - 1, subTermA(N1,S,T). </pre>	<p>como 'univ'</p> <p><i>subTerm(S,T)</i>:-Se cumple si <i>S</i> es un subtérmino de <i>T</i>.</p> <p>✓ En general el predicado '=' proporciona la misma expresividad que los predicados 'functor' y 'arg' juntos. Sin embargo, aunque los programas con '=' son más legibles, también son menos eficientes, pues necesitan construir una lista auxiliar.</p>
---	---

8.4 Predicados meta-lógicos

Los predicados meta-lógicos permiten controlar el algoritmo de resolución facilitando la meta-programación. Ésta consiste en construir programas que manipulan otros programas proporcionando una mayor expresividad al lenguaje.

8.4.1 Chequeo de tipo

Predicado	Condición para que se cumpla	Ejemplos
<i>var(X)</i>	<i>X</i> es una variable no instanciada	<pre> ?- var(X). X = _ ?- X = 1, var(X). no </pre>
<i>nonvar(X)</i>	<i>X</i> no es una variable o es una variable instanciada	<pre> ?- nonvar(X). no ?- X = 1, nonvar(X). X = 1 </pre>

La utilización de estos predicados permite al programador chequear si una variable está instanciada o no para proporcionar programas más flexibles y eficientes.

<pre> abuelo(X,Y):-nonvar(X), hombre(X), progenitor(X,Z), progenitor(Z,Y). abuelo(X,Y):-nonvar(Y), progenitor(Z,Y), progenitor(X,Z), hombre(X). </pre>	<i>abuelo(X,Y)</i> :- <i>X</i> es abuelo de <i>Y</i>
--	--

8.4.2 Comparación de términos no básicos

Predicado	Condición para que se cumpla	Ejemplos
<i>X==Y</i>	<i>X</i> e <i>Y</i> son iguales (no unifica las variables)	<pre> ?- f(X,2) = f(1,Y). X = 1 , Y = 2 ?- f(X,2) == f(1,Y). no </pre>
<i>X \== Y</i>	<i>X</i> e <i>Y</i> no son iguales (no unifica las variables)	<pre> ?- f(X,2) \== f(1,Y). yes </pre>

	unifica las variables)	
--	------------------------	--

<pre> unifica(X,Y):-var(X),var(Y), X = Y. unifica(X,Y):-var(X), nonvar(Y), noOcorre(X,Y),X=Y. unifica(X,Y):-var(Y), nonvar(X), noOcorre(Y,X),Y=X. unifica(X,Y):-nonvar(X), nonvar(Y), atomic(X), atomic(Y), X = Y. unifica(X,Y):-nonvar(X), nonvar(Y), compound(X), compound(Y), unifTerm(X,Y). unifTerm(X,Y):- functor(X,F,A), functor(Y,F,A), unifArgs(A,X,Y). unifArgs(N,X,Y):- N > 0, unifArg(N,X,Y), N1 is N - 1, unifArgs(N1,X,Y). unifArgs(0,X,Y). unifArg(N,X,Y):- arg(N,X,Ax), arg(N,Y,Ay), unifica(Ax,Ay). noOcorre(X,Y):-var(Y), X == Y. noOcorre(X,Y):-nonvar(Y), atomic(Y). noOcorre(X,Y):-nonvar(Y), compound(Y), functor(Y,F,A), noOcorreArgs(A,X,Y). noOcorreArgs(N,X,Y):- N > 0, arg(N,Y,An), noOcorre(X,An), N1 is N - 1, noOcorreArgs(N1,X,Y). noOcorreArgs(0,X,Y). </pre>	<pre> unifica(X,Y):- Se cumple si X e Y son unificables utilizando chequeo de ocurrencias. ?- unifica(f(1,X),f(Y,2)). X = 2 , Y = 1 ; ?- f(X,X)=f(Y,g(Y)). Error ... Stack Full, ?- unifica(f(X,X),f(Y,g(Y))). no noOcorre(X,T):- se cumple si la variable X no aparece en el término T </pre>
--	--

8.4.3 Conversión de Datos en Objetivos

El predicado *call(X)* se cumple si se cumple el objetivo *X*.

<pre> o(X,Y):-call(X). o(X,Y):-call(Y). ?- o(progenitor(belen,tomas), progenitor(tomas,belen)). yes ?- progenitor(belen,tomas) ; progenitor(tomas,belen). yes for(0,X). for(N,X):-call(X),N1 is N - 1, for(N1,X). ?- for(5,write('*')). ***** yes ?- T =.. [progenitor, tomas, belen], T. T = progenitor(tomas,belen) ; </pre>	<pre> o(X,Y):- se cumple si se cumple X o si se cumple Y ✓ El predicado o(X,Y) está predefinido como el operador ‘;’ ✓ El <i>Prolog Standard</i> sustituye automáticamente una objetivo en forma de variable X por <i>call(X)</i> </pre>
---	--

8.5 Corte

El corte es uno de los predicados internos más polémicos del lenguaje *Prolog*. Se utiliza para “podar” ramas del árbol de resolución consiguiendo que el sistema vaya más rápido. Un mal uso del corte puede podar ramas del árbol de resolución que contengan soluciones impidiendo que el sistema encuentre algunas soluciones (o todas) a un problema dado. De esta forma se aconseja utilizar el corte con precaución y **únicamente en el lugar necesario**, Ni antes ni después.

El predicado corte se representa mediante el símbolo ‘!’ y su efecto es:

- El predicado siempre se cumple.
- Si se intenta *re-ejecutar* (al hacer backtracking) elimina las alternativas restantes de los objetivos que hay desde su posición hasta la cabeza de la regla donde aparece.

```
q(X):-p(X).
q(0).
```

```
p(X):-a(X),!,b(X).
p(1).
```

```
a(2). a(3).
```

```
b(2). b(2). b(3).
```

<Con el corte>

```
| ?- q(X).
X = 2 ;
```

```
X = 2 ;
```

```
X = 0
```

<Sin se quita el corte>

```
| ?- q(X).
X = 2 ;
```

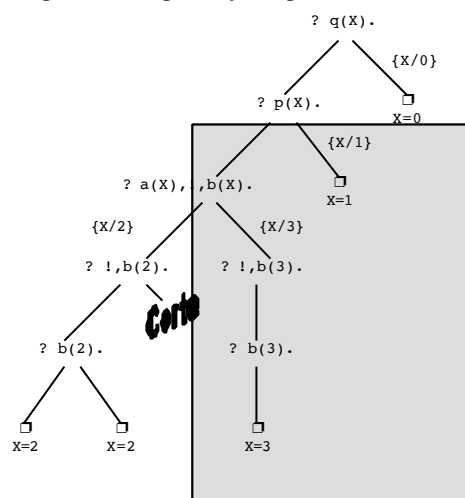
```
X = 2 ;
```

```
X = 3 ;
```

```
X = 1 ;
```

```
X = 0
```

En la figura puede observarse que el corte afecta a las soluciones alternativas del predicado ‘p(X)’ y del predicado ‘a(X)’.



```
padre(X,Y):-progenitor(X,Y),hombre(X).
```

```
| ?- padre(X,belen).
X = tomas ;
```

```
no
```

```
| ?-
```

```
padre(X,Y):-progenitor(X,Y), hombre(X), !.
```

```
| ?- padre(X,belen).
X = tomas
```

```
| ?-
```

padre(X,Y):-X es el padre de Y.

Se encuentra el primer progenitor hombre de belen (tomas) y el sistema busca más progenitores mediante backtracking. Como no los encuentra, devuelve *no*.

En este ejemplo particular, suponiendo que cualquier persona tiene un único padre, podría ser interesante podar las alternativas que se encuentren.

Sería como declarar:

“X es padre de Y si es su progenitor, es hombre y además, es único”:

✓ Un buen uso del corte puede permitir la construcción de programas *Prolog* eficientes.

```
orgulloso(X):-padre(X,Y), recién_nacido(Y).
```

```
recién_nacido(maria).
```

```
? orgulloso(X).
```

```
no
```

orgulloso(X):-X está orgulloso (cuando es padre de un recién nacido).

Si se suprime el corte en la definición de *padre* devolvería a *Pedro*. Sin embargo, tal y como se ha definido, en cuanto se encuentra que *pedro* es padre de

	<p><i>ana</i> y que <i>ana</i> no es un recién nacido, el <i>Prolog</i> poda la alternativa de que <i>Pedro</i> también es padre de <i>María</i></p> <p>✓ La inclusión de cortes puede hacer que el sistema no encuentre algunas soluciones.</p> <p>✓ El exceso de cortes en un programa dificulta la legibilidad.</p>
--	--

8.5.1 Aplicaciones del Corte

En general, la utilización del corte debe ser examinada con precaución. En muchas ocasiones conviene sustituir el corte por construcciones que encapsulen su utilización.

8.5.1.1 Indicar al Sistema que ha escogido la regla correcta (if-then-else)

<pre>soloMujeres([], []). soloMujeres([X Xs],[X Ys]) :- mujer(X), soloMujeres(Xs,Ys). soloMujeres([X Xs],Ys) :- hombre(X), soloMujeres(Xs,Ys). ?- soloMujeres3([ana,pedro,maria],V). V = [ana,maria] ; no</pre>	<p><i>soloMujeres(Xs,Ys)</i>:-<i>Ys</i> contiene las mujeres de <i>Xs</i></p> <p>Si se hace backtracking, el predicado chequeará más alternativas comprobando si las mujeres son, a su vez, hombres.</p>
<pre>soloMujeres1([], []). soloMujeres1([X Xs],[X Ys]) :- mujer(X), !, soloMujeres1(Xs,Ys). soloMujeres1([X Xs],Ys) :- hombre(X), soloMujeres1(Xs,Ys).</pre>	<p><i>soloMujeres1</i>: Si se inserta un corte tras la condición <i>mujer</i> se le indica al sistema que no busque más alternativas cuando encuentra una mujer.</p>
<pre>soloMujeres2([], []). soloMujeres2([X Xs],[X Ys]) :- mujer(X), !, soloMujeres2(Xs,Ys). soloMujeres2([X Xs],Ys) :- soloMujeres2(Xs,Ys).</pre>	<p><i>soloMujeres2</i>: Utilizando el corte, si el sistema encuentra una mujer, no va a chequear si es un hombre. La condición de que sea un hombre, podría suprimirse.</p>
<pre>ifThenElse(Cond,X,Y):-call(Cond)!,call(X). ifThenElse(Cond,X,Y):-call(Y). soloMujeres3([], []). soloMujeres3([X Xs],Ys1) :- ifThenElse(mujer(X), Ys1 = [X Ys], Ys1 = Ys), soloMujeres3(Xs,Ys).</pre>	<p>El esquema conseguido puede generalizarse.</p> <p><i>ifthenelse</i> (Cond,X,Y) :-Si <i>Cond</i> entonces ejecuta <i>X</i> sino, ejecuta <i>Y</i>.</p>
<pre>soloMujeres4([], []). soloMujeres4([X Xs],Ys1) :- (mujer(X) -> Ys1 = [X Ys] ; Ys1 = Ys), soloMujeres4(Xs,Ys).</pre>	<p>El predicado <i>ifThenElse</i> está predefinido mediante el operador '<i>-></i>'.</p>

8.5.1.2 Negación por fallo

El lenguaje *Prolog* utilice un subconjunto de la lógica de predicados de primer orden (cláusulas *Horn*) lo que impide modelizar ciertas situaciones con conocimiento negativo. La *negación por fallo* es una aproximación a las técnicas de representación de conocimiento negativo. La idea intuitiva es, si se detecta que algo no se cumple, entonces, se supone que es falso.

<pre>animal(X):-perro(X). animal(X):-serpiente(X). serpiente(kika). perro(kiko).</pre>	<p>Se desea representar el siguiente conocimiento:</p> <p>“A Ana le gustan todos los animales salvo las serpientes”</p> <p><i>fail</i> es un predicado interno que siempre falla.</p>
---	---

<pre>le_gusta(ana,X):-serpiente(X),!,fail. le_gusta(ana,X):-animal(X). ?- le_gusta(ana,kika). no ?- le_gusta(ana,kiko). yes</pre>	<p>Se puede generalizar el patrón de <i>negación por fallo</i>.</p>
<pre>falla_si(X):-call(X),!,fail. falla_si(X). le_gusta2(ana,X):-falla_si(serpiente(X)), animal(X). ?- le_gusta2(ana,kika). no ?- le_gusta2(ana,kiko). yes</pre>	<p><i>falla_si(X)</i>:- falla si el objetivo <i>X</i> se cumple y se cumple si <i>X</i> falla.</p> <p>✓ El predicado <i>falla_si</i> está predefinido en <i>Prolog Standard</i> como el operador: ‘\+’.</p> <p>📖 En muchos sistemas, el predicado ‘\+’ se conoce como <i>not</i>. En la definición Standard se decidió evitar el predicado <i>not</i> para que los usuarios no lo confundiesen con una negación común.</p> <p>El símbolo ‘\+’ pretende ser una representación del símbolo matemático \neg (no demostrable)</p>
<pre> ?- le_gusta2(ana,X). no ?- le_gusta2(ana,X). no</pre>	<p>Con variables sin instanciar no funciona. En cuanto encuentra una serpiente, falla y no busca más alternativas.</p>
<pre>le_gusta3(ana,X):- animal(X), falla_si(serpiente(X)).</pre>	<p>En esta caso, puede corregirse cambiando de orden los objetivos.</p> <p>✓ Al introducir <i>negación por fallo</i> el orden de los objetivos influye en los resultados.</p>

📖 La raíz del problema es que la frase “A Ana le gustan todos los animales salvo las serpientes”, se formalizaría en lógica como

$$\forall x (\text{animal}(x) \wedge \neg \text{serpiente}(x) \rightarrow \text{le_gusta}(\text{ana},x))$$

y al pasarla a forma clausal se obtendría una cláusula con dos literales positivos (no *Horn*)

8.5.1.3 Reglas por defecto

<pre>pension(X,invalides):- invalido(X),!. pension(X,jubilacion):- jubilado(X),!. pension(X,nada). invalido(pedro). jubilado(tomas). jubilado(pilar). ?- pension(tomas,X). X = jubilado ?- pension(ana,X). X = nada</pre>	<p><i>pension(P,Tp)</i>:- <i>Tp</i> es el tipo de pension que le corresponde a la persona <i>P</i></p> <p>☺ Con preguntas sencillas funciona</p>
<pre> ?- pension(tomas,nada). yes ?- pension(X,jubilacion). X = tomas ?-</pre>	<p>☹ No funciona, debería responder que no!</p> <p>☹ Tampoco funciona, le ha quitado la pensión a <i>Pilar</i></p> <p>✓ La introducción de cortes puede hacer que al plantear al sistema preguntas no contempladas, las respuestas sean erróneas.</p>
<pre>pension2(X,invalides) :- invalido(X). pension2(X,jubilacion):- jubilado(X).</pre>	<p>La solución sería quitar el corte y utilizar dos niveles.</p>

<pre> asigna(X,P) :- pension2(X,P). asigna(X,nada):- \+ pension2(X,P). ?- asigna(X,jubilacion). X = tomas ; X = pilar ; no </pre>	<p><i>pension2(P,Tp):- Tp es el tipo de pension que le corresponde a la persona P</i></p> <p><i>asignacion(P,Ta):-Ta es el tipo de asignación que le corresponde a P</i></p>
---	--

8.5.1.4 Finalizar generación de alternativas (once)

En algunas ocasiones, puede ser necesario finalizar la generación de alternativas devolviendo únicamente la primera solución encontrada. Las razones pueden ser, porque se conoce que el problema sólo va a tener una solución, o porque no tiene importancia devolver una solución u otra y se opta por devolver la primera. Mediante un corte, puede finalizarse la generación de otras soluciones.

<pre> hayProgJubilados:-progenitor(X,Y),jubilado(X),!. once(X):-call(X),!. hayProgenitoresJubilados:-once((progenitor(X,Y), jubilado(X))). </pre>	<p><i>hayPadresJubilados:-</i> Se cumple si en la base de conocimiento existe algun progenitor jubilado.</p> <p><i>once(X):-</i> Se cumple si se <i>cumple</i> el objetivo X, pero no intenta re-ejecutar X.</p> <p>✓ Los dobles paréntesis son necesarios para que el sistema reconozca la sintaxis.</p> <p>➔ El predicado <i>once</i> está predefinido en el Prolog <i>Standard</i>, sin embargo, muchos sistemas comerciales no lo utilizan o utilizan el predicado <i>one</i></p>
---	---

8.6 Predicados Extra-lógicos

8.6.1 Entrada/Salida

En *Prolog* la Entrada/Salida (E/S) se realiza mediante efectos laterales. Los predicados de E/S tienen un escaso valor lógico, pero, al ser encontrados por el sistema, ejecutan las acciones correspondientes.

➔ El principal cambio producido al standarizar el lenguaje *Prolog* ha sido la redefinición de los predicados clásicos de E/S. Sin embargo, muchas implementaciones continúan ofreciendo el repertorio anterior. Se indica mediante el símbolo ➔ aquellos predicados que no son *standard*.

8.6.1.1 E/S a nivel de términos

Los siguientes predicados leen y escriben términos *Prolog* completos

Predicado	Se cumple cuando:	Efecto lateral	Ejemplos Sencillos	Standard
<i>write(X)</i>	Siempre	Escribe el valor del término X	?- write(f(x,3+4)). f(x,3+4). yes	<i>write</i>
<i>read(X)</i>	Si es posible unificar X con el valor leído ✓ No se hace backtracking. Analiza los caracteres de entrada hasta encontrar un término y, si unifica, se cumple, sino, falla.	Lee un término <i>Prolog</i> ✓ Los términos <i>Prolog</i> deben acabar en punto	?- read(X). :<usuario> f(x,3+4). X = f(x,3 + 4) ?- read(X), X = 2. :<usuario> 3. no	<i>read</i>
<i>display(X)</i>	Siempre	Escribe el valor del término X en notación functor	?- display(f(x,3+4)). f(x,(+)(3,4)) yes	<i>write_canonical</i>
<i>nl</i>	Siempre	Escribe un salto de línea		<i>nl</i>

<pre>saludo:-write('Tu nombre?'), read(N), write('Hola '), write(N). ?- saludo. Tu nombre? :<usuario> juan. Hola juan yes</pre>	<p><i>saludo</i>: Pide al usuario un nombre (término <i>Prolog</i>), lo lee y lo muestra por pantalla.</p>
<pre>writeln([X Xs]):-write(X), writeln(Xs). writeln([]):-nl.</pre>	
<pre>% Predefinido repeat. repeat:-repeat. cuadrados:- repeat, leeNumero(X), procesa(X), !. leeNumero(X):-repeat, write('Numero?'), read(X), number(X), !. procesa(0):-!. procesa(X):- R is X*X, writeln([X, '^2 = ',R]), fail.</pre>	<p><i>repeat</i>: Predicado que siempre se cumple y que tiene un infinito número de soluciones.,</p> <p><i>cuadrados</i>: Solicita numeros al usuario hasta que éste teclea 0. Para cada número, imprime su cuadrado.</p> <p>✓ Los bucles Prolog con <i>repeat</i> tienen siempre un mismo esquema:</p> <pre>bucle:- repeat, <cuerpo del bucle>, <condición de salida, si se cumple => fin>, !.</pre> <p>El corte al final es necesario para que el predicado no intente re-ejecutarse de nuevo.</p>

8.6.1.2 E/S a nivel de caracteres

<i>Predicado</i>	<i>Se cumple cuando:</i>	<i>Efecto lateral</i>	<i>Standard</i>
<i>get0(X)</i>	Si es posible unificar el caracter leído con <i>X</i>	Lee un caracter del teclado	<i>get_char</i>
<i>get(X)</i>	Si es posible unificar el caracter leído con <i>X</i>	Lee el siguiente caracter distinto de blanco.	
<i>put(X)</i>	Siempre	Escribe el caracter <i>X</i>	<i>put_char</i>

<pre>leeLsPals(Pals):- once((get0(C), leePals(C,Pals))). leePals(C,[]):-punto(C),!. leePals(C,[Pal Pals]):-alfaNu(C),!, leePal(C,Pal,Csig), leePals(Csig,Pals). leePals(C,Pals):- get0(Csig), leePals(Csig,Pals). leePal(C1,Pal,Csig):-cogerCodsPal(C1,Cods,Csig), atom_chars(Pal,Cods). cogerCodsPal(C1,[C1 Cars],Cfin):-alfaNu(C1), get0(Csig), cogerCodsPal(Csig,Cars,Cfin). cogerCodsPal(C,[],C):- \+ alfaNu(C). alfaNu(C):- (C>=0'a, C<=0'z) ; (C>=0'A, C<=0'Z) ; (C>=0'0, C<=0'9). punto(0'.). espacio(0').</pre>	<p><i>leeLsPals(X)</i>:- Obtiene una lista de Atomos de la entrada</p> <pre> ?- leeLsPals(X). : Esta es una cadena con 1 numero. X = ['Esta',es,una,cadena,con,'1',numero]</pre> <p>➔ La forma de obtener el código ASCII de un caracter varía de un sistema a otro. En LPA, 0'a representa el código ASCII del caracter 'a'.</p>
--	---

8.6.1.3 *E/S con ficheros*

Esta parte es una de las que más difiere con el *Prolog Stándar* donde se utilizan manejadores para trabajar con *streams* o flujos de E/S.

<i>Predicado</i>	<i>Se cumple cuando:</i>	<i>Efecto lateral</i>	<i>Standard</i>
<i>tell(F)</i>	Si no hay errores de apertura	Abre <i>F</i> como <i>stream</i> de salida actual	open
<i>see(F)</i>	Si no hay errores de apertura	Abre <i>F</i> como <i>stream</i> actual de entrada.	open
<i>told</i>	Siempre	Cierra <i>stream</i> de salida actual	close
<i>seen</i>	Siempre	Cierra <i>stream</i> de entrada actual	close
<i>telling(F)</i>	Unifica <i>F</i> con el nombre del <i>stream</i> de salida actual	Ninguno	
<i>seeing(F)</i>	Unifica <i>F</i> con el nombre del <i>stream</i> de salida actual	Ninguno	

<pre> verFich:-write('Nombre fichero?'), read(N), seeing(Antes), see(N), bucle, seen, see(Antes). bucle:-repeat, get0(C), minMay(C,Cm), put(Cm), at_end_of_file, !. minMay(C,Cm):-minuscule(C),!, Cm is C - 0'a + 0'A. minMay(C,C). minuscule(C):- C >= 0'a, C <= 0'z.</pre>	<p><i>verFich</i>:-Pregunta un nombre de fichero al usuario, lo abre y visualiza su contenido en mayúsculas.</p>
---	--

8.6.2 Acceso a la Base de Datos

Los sistemas *Prolog* ofrecen la posibilidad de modificar en tiempo de ejecución el contenido de la base de conocimiento.

<i>Predicado</i>	<i>Se cumple cuando:</i>	<i>Efecto lateral</i>
<i>asserta(T)</i>	Siempre	Añade al principio de la base de conocimiento el término <i>T</i>
<i>assertz(T)</i>	Siempre	Añade al final de la base de conocimiento el término <i>T</i> .
<i>retract(T)</i>	Si <i>T</i> unifica con el término eliminado.	Elimina de la base de conocimiento el término <i>T</i>

<pre> :-dynamic(fib/2). fib(0,1). fib(1,1). fib(X,Y):-X>1, X1 is X - 1, fib(X1,Y1), X2 is X - 2, fib(X2,Y2), Y is Y1 + Y2, asserta((fib(X,Y):-!)).</pre>	<p><i>fib(X,Y)</i>:- <i>Y</i> es el <i>X</i>-ésimo número de la sucesión de Fibonacci. Se memorizan resultados intermedios.</p> <p>✓ Para poder insertar dinámicamente un predicado en la base de conocimiento, debe utilizarse la directiva</p> <p><i>:-dynamic(P/A).</i></p> <p>que indica al sistema que el predicado <i>P</i> de aridad <i>A</i> es dinámico.</p>
---	---

8.7 Predicados de Segundo Orden

Supóngase que se desea construir un programa que busque todas las soluciones para un objetivo dado. Con los predicados considerados hasta ahora, la tarea no es sencilla. El problema es que Prolog devuelve las soluciones al realizar *backtracking*. Sería necesario forzar el *backtracking* para cada solución e ir recogiendo las soluciones encontradas. El problema es que la pregunta está fuera del modelo lógico (toda la información de una computación, se pierde al realizar *backtracking*). La solución es utilizar los siguientes predicados:

Predicado	Se cumple cuando:
<i>findall</i> (<i>T,C,L</i>)	<p><i>L</i> es la lista de todas las instancias del término <i>T</i> tales que el objetivo <i>C</i> se cumple.</p> <p>La lista de soluciones no se ordena (se insertan en el orden en que aparecen) y puede contener elementos repetidos.</p> <p>Si no hay soluciones, se obtiene la lista vacía</p> <p>Tras el objetivo, las variables de <i>T</i> y <i>C</i> permanecen sin instanciar.</p>
<i>bagof</i> (<i>T,C,L</i>)	<p>Se cumple si <i>L</i> es la lista de todas las instancias de <i>T</i> tales que el objetivo <i>C</i> se cumple.</p> <p><i>C</i> puede tener la forma: $V_1^{\wedge}V_2^{\wedge}...\wedge V_n^{\wedge}$Objetivo indicando que las variables V_1, V_2, \dots, V_n están cuantificadas existencialmente.</p> <p>Si no se cumple <i>C</i> el predicado falla</p> <p><i>L</i> no se clasifica y puede contener elementos duplicados.</p>
<i>setof</i> (<i>T,C,L</i>)	Similar a <i>bagof</i> salvo que la lista de resultados se clasifica y se eliminan elementos duplicados.

<pre>hijos(Xs):-findall(X,progenitor(Y,X),Xs). ?- hijos(V). V = [belen,belen,lucia,ana,pedro,jose,maria]</pre>	<p><i>hijos(L)</i>:- se obtiene la lista de hijos</p> <p>Con <i>findall</i> la lista puede contener duplicados y no está ordenada.</p>
<pre>hijos1(Xs):-bagof(X,progenitor(Y,X),Xs). ?- hijos1(V). V = [ana,pedro] ; V = [jose,maria] ; V = [belen] ; V = [belen,lucia] ; no</pre>	<p>Con <i>bagof</i> las variables no cuantificadas, indican posibles soluciones. Para cada posible valor de la variable (en el ejemplo, para cada progenitor), se devuelve la lista de sus hijos.</p>
<pre>hijos2(Xs):-bagof(X,Y^progenitor(Y,X),Xs). ?- hijos2(V). V = [belen,belen,lucia,ana,pedro,jose,maria]</pre>	<p>Las variables pueden cuantificarse existencialmente mediante \wedge, actuando de forma similar a <i>findall</i></p>
<pre>hijos3(Xs):-setof(X,Y^progenitor(Y,X),Xs). ?- hijos3(V). V = [ana,belen,jose,lucia,maria,pedro]</pre>	<p><i>setof</i> elimina ordena la lista y elimina duplicados.</p>





El efecto de estos predicados se podría simular mediante efectos laterales, almacenando las soluciones en la base de datos con los predicados definidos en la sección 8.6.2.

8.8 Directivas

El *Prolog Standard* define una serie de directivas que indican al sistema tareas a realizar u opciones de compilación.

<code>:-ensure_loaded(F).</code>	Indica al sistema que cargue el fichero <i>F</i>
<code>:-multifile(P/N).</code>	Indica que el predicado <i>P</i> de aridad <i>N</i> puede definirse en varios ficheros. Por defecto, las definiciones deben estar en un solo fichero.
<code>:-dynamic(P/N).</code>	Indica al sistema que la definición del predicado <i>P</i> de aridad <i>N</i> puede modificarse de forma dinámica
<code>:-initialization(C).</code>	Declara que el objetivo <i>C</i> debe ser ejecutado tras cargar el fichero.
<code>:-op(Prioridad,Asociatividad,Atomo).</code>	Define <i>Atomo</i> como un operador con la <i>Prioridad</i> y <i>Asociatividad</i> dadas

9 Ejercicios Propuestos

Clave:		Muy Fácil
		Fácil
		Medio
		Largo (proyectos de programación)

1.- `cambia(Xs,Ys):-` Si $Xs=[x_1,x_2,\dots,x_n]$, entonces $Ys=[y_1,y_2,\dots,y_n]$ de forma que $y_i = x_i + \sum_{t=1}^n x_t$ para $i = 1..n$ (👉)

```
? cambia ([1,3,2],Xs).
Xs = [7,9,8]
```

2.- `sucesion(N,Xs):-` Xs es de la forma $[x_1,x_2,\dots,x_n]$ donde $x_1 = 0$, $x_2 = 1$ y $x_{i+2} = 2*x_i + x_{i+1}$ (👉)

```
Ejemplo: ? sucesion(8,V).
V = [0,1,1,3,5,11,21,43]
```

3.- `diferentes(Xs,N):-` N es el numero de elementos diferentes de la lista Xs (👉)

```
Ejemplo: ? diferentes([1,2,2,1,1,3,2,1],V).
V = 4
```

4.- Supóngase que se ha cargado el siguiente programa en un sistema *Prolog*.

```
a(X,Y):-b(X,Y).
a(X,Y):-c(X,Y).
b(X,Y):-d(X),!,e(X,Y).
b(X,Y):-f(X,Y).
c(1,2). c(1,3).
d(1). d(2).
e(2,3).
f(3,4). f(3,5).
```

Indicar cuales son las respuestas del sistema (suponer que se solicitan todas por *backtracking*) al ejecutar: (👉)

```
a. ? a(1,X).
b. ? a(2,X).
c. ? a(3,X).
```

5.- `rep(Xs,V) :-` V es el representante decimal equivalente a la lista de enteros Xs . (👉)

```
Ejemplo:
? rep([1,4,6],V).
V = 146
```

6.- `monte(Xs):-` Se cumple si Xs tiene forma de monte. Es decir, es creciente hasta un elemento y decreciente desde ese elemento hasta el final. (👉)

```
Ejemplo: ? monte([1,3,5,6,4,3,0]).
yes
```

7.- `cambio(X,Xs) :-` Xs es la lista monedas de 1,5,25 ó 100 necesarias para alcanzar la cantidad X (👉)

```
? cambio 156
[100,25,25,5,1]
```

8.- `decBin(D,B):-` B es el valor binario de D (👉👉)

```
? decBin(13,V).
V = [1,1,0,1]
? decBin(V,[1,0,1,1]).
13
```

9.- `combs(Xss,Ys):-` Se cumple si Ys es una lista en la que el primer elemento es alguno de los elementos de la primera lista de Xss , el segundo elemento es alguno de los elementos de la segunda lista de Xss y...el n -ésimo elemento de cada sublista es alguno de los elementos de la n -ésima lista de Xss . Mediante *backtracking*, debe devolver todas las posibles listas Ys que satisfagan dicha condición. (👉👉)

Ejemplo:

```
? combs([ [1,2,3], [4,5], [6] ],V).
V = [1,4,6] ;
V = [1,5,6] ;
V = [2,4,6] ;
V = [2,5,6] ;
```

```
V = [3,4,6] ;
V = [3,5,6] ;
No
```

10.- `triangulo(N)`:- muestra por pantalla un triangulo de N filas (🔗🔗)

Ejemplo: `? triangulo(4).`

```
  *
 ***
*****
*****
```

11.- `sumCombs(Xss,V)`:- V es la suma de todas las representaciones decimales de las listas obtenidas mediante las combinaciones del ejercicio (🔗🔗)

```
? sumCombs([ [1,2,3], [4,5], [6] ],V).
V = 1506
```

12.- `decBin(D,B)`:- B es el valor binario de D (🔗🔗)

```
? decBin(13,V).
V = [1,1,0,1]
? decBin(V,[1,0,1,1]).
V = 13
```

13.- `varsRep(N,Xs,Vs)`:- Vs son las variaciones con repetición de los elementos de Xs tomados de N en N

`? varsRep(3,[0,1],Vs).` (🔗🔗)

```
Vs = [[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]
```

```
? varsRep(2,[0,1,2],Vs).
Vs = [[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,0],[2,1],[2,2]]
```

14.- `palabras(Cad,Pals)`:- $Pals$ es una lista de estructuras de la forma: `c(P,N)`, donde P es una palabra de Cad y N es el número de veces que se repite dicha palabra en Cad . Se deben ignorar las diferencias entre mayúsculas y minúsculas y los caracteres de puntuación. (🔗🔗🔗)

```
? palabras ("¿No duerme nadie por el mundo?. No, no duerme nadie.",Ps).
Ps = [c(no,3), c(duerme,2), c(nadie, 2), c(por,1), c(el,1), c(mundo,1)]
```

15.- `busca(Exp, Cad, Ps)`:- Ps es una lista de palabras de Cad que encajen con la expresión regular Exp . Se deben ignorar las diferencias entre mayúsculas y minúsculas y los caracteres de puntuación. (🔗🔗🔗)

```
? busca ("*e", "¿No duerme nadie por el mundo?. No, no duerme nadie.",Ps).
Ps = [duerme, nadie, duerme, nadie]
```

16.- `tartaglia(N)`:- muestra por pantalla las N primeras filas del triángulo de Tartaglia. (🔗🔗🔗)

Ejemplo: `? tartaglia(5)`

```
  1
 1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

17.- Desarrollar un programa que calcule y simplifique derivadas de funciones de forma simbólica. (🔗🔗🔗)

18.- Construir un programa de gestión de una agencia matrimonial. El programa debe almacenar datos de personas, así como sus preferencias y, buscar las mejores combinaciones de parejas. (🔗🔗🔗)

19.- Construir un programa que maneje polinomios. El programa será capaz de leer polinomios a a partir de la entrada del usuario, sumarlos, multiplicarlos y restarlos. Además, el programa buscará raíces de forma analítica o numérica. El programa no se restringirá a polinomios cuadráticos y se puede considerar el cálculo de raíces complejas. (🔗🔗🔗🔗)

20.- Escribir un programa interactivo que juegue finales de partidas de ajedrez. Se pueden tomar finales de una reina contra dos torres, por ejemplo. (🔗🔗🔗🔗)

21.- Escribir un programa que tome una representación de una matriz de puntos cuadrada (de tamaño 10x10, por ejemplo) y decida qué letra del alfabeto representa. El patrón podría tener alguna distorsión o ambigüedad que el programa debe ser capaz de afrontar. (🔗🔗🔗🔗)

22.- Escribir un algoritmo que tome como entrada un mapa dividido en regiones (se almacena información de qué regiones tienen frontera en común entre sí) y asigne un color a cada región de forma que no haya dos regiones adyacentes con frontera común. (🔗🔗🔗🔗)

23.- Construir un programa interactivo que muestre un *prompt* por pantalla y reconozca los siguientes comandos:

Fin	Fin del programa
Ver Var	Muestra el valor de 'Var' en pantalla
Var = Expresión	Asigna a Var el valor de 'Expresión'
Expresión	Evalúa la 'expresión' y muestra el resultado en pantalla

Donde Expresión es una expresión aritmética formada por sumas, restas, multiplicaciones, divisiones, constantes numéricas y variables. Un ejemplo de sesión podría ser: (☞☞☞☞)

```
$ x = 3 * 2 + 4 * (6 + 2).  
OK  
$ x + 1.  
39  
$ y = x + 1.  
OK  
$ y + 2.  
40  
$ x = 2 * 3.  
OK  
$ y + 2.  
8  
$ V(y).  
x+1  
$ fin.  
Adios!
```

10 Bibliografía Comentada

A continuación se presenta una bibliografía comentada de los libros sobre el lenguaje *Prolog* o la programación lógica más destacables por el autor. Se incluye un signo que resume el principal motivo de inclusión del libro en la bibliografía. El significado de dichos símbolos es:

✓	Lectura Fundamental o referencia clásica
🔧	Destacable por las aplicaciones prácticas
📖	Destacable por las presentaciones teóricas
📄	Destacable para consulta o referencia

- [Bird, 97] R. Bird, O. de Moor, *Algebra of Programming*. Prentice-Hall Intl. Series in Computer Science, ISBN: 0-13-507245-X, 1997

Presenta la programación desde un punto de vista algebraico (siguiendo la teoría de la categoría) y centrándose en el paradigma funcional. Aunque el libro no menciona el lenguaje *Prolog*, presenta una metodología de resolución de problemas mediante relaciones lógicas y generaliza el tratamiento de funciones de orden superior recursivas de una forma similar a los patrones recursivos utilizados. Reservado para programadores con inquietudes matemáticas. 📖

- [Bratko, 90] I. Bratko, *Prolog, Programming for Artificial Intelligence*. Addison-Wesley, ISBN: 0-201-41606-9, 1990

El libro se divide en dos partes, una primera de introducción al lenguaje y una segunda de presentación de aplicaciones a la inteligencia artificial. Ejemplos comprensibles y aplicaciones convencionales. 🔧

- [Covington, 97] M. A. Covington, D. Nute, A. Vellino, *Prolog Programming in Depth*. Prentice Hall, ISBN: 0-13-138645-X, 1997

Resolución de aplicaciones de Inteligencia Artificial mediante *Prolog*. No separa claramente la componente declarativa de la procedural. Contiene un sugestivo capítulo dedicado a expresar algoritmos procedurales en lenguaje *Prolog* y unos anexos resumiendo la norma ISO de Prolog y diferencias entre diversas implementaciones. 🔧

- [Clocksin, 81] W.F. Clocksin, C.S. Mellish, *Programación en Prolog*, Colección Ciencia-Informática, Ed. Gustavo Gili, ISBN:84-252-1339-8, 1981

Es el libro de introducción al lenguaje *Prolog* clásico. A pesar de su edad, el libro sigue siendo una buena fuente de obtención de ejemplos sencillos de funcionamiento del lenguaje. Además, la traducción mantiene el tipo. ✓

- [Clocksin, 97] W.F. Clocksin, *Clause and Effect*. Springer-Verlag, ISBN: 3-540-62971-8, 1997

Con un planteamiento similar al de los presentes apuntes (aunque con un nivel algo más elevado), presenta una introducción práctica, concisa y entretenida a la programación en Prolog. Está a mitad de camino entre [Clocksin, 81] y [O'Keefe 91] ✓

- [Deransart, 96] P. Deransart, A. Ed-Dbali, L. Cervoni, *Prolog: The Standard. Reference Manual*. Springer-Verlag, ISBN: 3-540-59304-7, 1996

Manual de Referencia del Prolog Standard. Para aquellos que deseen construir implementaciones *standard* de *Prolog* o para programadores muy preocupados por la portabilidad. Los ejemplos no están pensados para enseñar a programar en *Prolog*, sino para distinguir características del lenguaje. 📖

- [Flach, 94] P. Flach, *Simply Logical, Intelligent Reasoning by Example*. John Wiley & Sons. ISBN: 0-471-94152-2, 1994

Comienza con una perfecta introducción a las principales técnicas de programación *Prolog* para presentar de una forma clara y concisa aplicaciones del lenguaje en sistemas de inteligencia artificial. 🔧

- [Fitting, 96] M. Fitting, *First Order Logic and Automated Teorem Proving*. Springer-Verlag, ISBN: 0-387-94593-8, 1996

Introducción a las técnicas de demostración automática en lógica de proposiciones y predicados utilizando *Prolog* como lenguaje de implementación de ejemplos. 📖

- [Lloyd, 93] J. W. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, ISBN: 3-540-18199-7, 1993

Recomendable para el lector interesado en profundizar en las bases teóricas de la programación lógica 📖

- [O'Keefe, 90] R. A. O'Keefe, *The Craft of Prolog*. MIT Press, ISBN: 0-262-15039-5, 1990
Con un planteamiento avanzado, el libro es fundamental para aquél que se considere un programador serio en *Prolog*. El capítulo de tratamiento de secuencias proporciona una idea de los patrones de recursividad ofrecidos en estos apuntes. ✓
- [Ross, 89] P. Ross, *Advanced Prolog, techniques and examples*. Addison-Wesley, ISBN: 0-201-17527-4, 1989
Presenta varias aplicaciones con un planteamiento riguroso y avanzado. 🐭
- [Sterling, 94] L. Sterling, E. Shapiro. *The Art of Prolog*. The MIT Press, ISBN: 0-262-19338-8, 1994
Referencia clásica e imprescindible del lenguaje Prolog. Realiza la presentación partiendo del núcleo declarativo del lenguaje y ofreciendo gradualmente las características menos declarativas. Contiene otras dos partes interesantes de métodos avanzados y aplicaciones. ✓
- [Shoham, 94] Y. Shoham, *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann Publishers, Inc. ISBN: 1-55860-319-0, 1994
Se centra directamente en la presentación de aplicaciones de inteligencia artificial en *Prolog* partiendo de que el lector conoce el lenguaje. Destacable por las aplicaciones presentadas. 🐭

11 Índice

A

abuelo, 16
 algunHombre, 8
 alisar, 15
 animal, 19, 20
 antepasado, 4
 árbol, 13, 18
 árbol binario de búsqueda, 13
 arg, 15, 17
 Aritmética, 9, 11
 asserta, 23
 assertz, 23
 atom, 14, 15, 22
 atom_chars, 14, 22
 atomic, 15, 17

B

backtracking, 3, 6, 7, 9, 18, 19, 21, 24
 bagof, 24
 bueno, 3

C

cabeza, 4, 5, 6, 18
 call, 17, 19
Caso básico, 4
Caso Recursivo, 4
 chequeo de ocurrencias, 5, 17
 Chequeo de tipos, 14
 Clasificación, 12
 comp, 13
 compound, 15, 17
 comprime, 13
 concat, 8, 13, 15
 Conversión de tipos, 14
 Corte, 18, 19
 creaArbol, 13
 cuadrados, 22
 cuida, 3

D

display, 21

E

edad, 9, 27
 elimina, 7, 18, 24
 Entrada/Salida, 21
 estructura recursiva, 13

F

fact, 10
 fail, 19, 20, 22
 falla_si, 20
 fib, 23
 filtrado, 12, 15
 filtraHombres, 9
 findall, 24

float, 15
 función, 5, 11, 27
 functor, 15, 17, 21

G

Generación, 8, 9, 11, 12
 get, 22
 get0, 22, 23
 grafo, 4
 grande, 5

H

hayProgJubilados, 21
 Hecho, 3
 hijos, 4, 24
 hombre, 4, 6, 8, 16, 18, 19
 hombres, 6, 9, 19
 horizontal, 5

I

ifThenElse, 19
 inserta, 12, 13, 19
 insertArbol, 13
 integer, 14, 15
 intervalo, 11, 12
 invalido, 20
 is, 9, 10, 11, 12, 13, 15, 17, 22, 23

J

jubilado, 20, 21

L

le_gusta, 19, 20
 leeLsPals, 22
 leeNumero, 22
 list, 15
 lista, 6, 7, 8, 9, 11, 12, 13, 14, 15, 22, 24
 Lista, 6, 11
 listArbol, 13
 long, 11

M

madre, 3, 4
 mayor, 5, 11, 12, 13, 16
 mezcla, 12, 13
 mujer, 4, 9, 19

N

Negación por fallo, 19
 nl, 22
 nonvar, 16, 17
 noOcurre, 17
 noPertenece, 6
 not, 20
 number, 14, 15, 22

number_atom, 14
number_chars, 14

O

objetivo, 3, 9, 11, 17, 20, 21, 24
once, 21, 22
ordenaBruto, 12
ordenada, 12, 24
ordenaIns, 13
orgulloso, 18

P

padre, 15, 18
par, 10, 12
pares, 12
paro, 3
particion, 12
pension, 20
permutacion, 9
Permutación, 9
perro, 19
pertenece, 6, 7, 8, 9
Predicados Internos, 14
Predicados meta-lógicos, 16
prefijo, 8
prod, 11
prodEscalar, 11
progenitor, 3, 4, 5, 16, 17, 18, 21, 24
put, 22, 23

Q

quicksort, 13

R

read, 21, 22, 23
recien_nacido, 18
Recursividad, 4, 6, 11
Regla, 3, 4, 20
Reglas por defecto, 20
repeat, 22
repite, 12, 13
resolución, 3, 5, 11, 16, 18, 27
retract, 23

S

saludo, 22
see, 23
seeing, 23
seen, 23
Segundo Orden, 24
serpiente, 19, 20
setof, 24
sinDuplicados, 9
Sintaxis, 3
soloMujeres, 19
sublista, 8
Substitución de Respuesta, 3
substitución vacía, 5
subTer, 15
subTerm, 15
sufijo, 8
sum1, 11
suma, 10
sumAcum, 11
sumaLogica, 15

T

tell, 23
telling, 23
todosIguales, 7
told, 23

U

unifica, 6, 9, 10, 16, 17, 21, 23
Unificación, 5

V

var, 16, 17
verFich, 23
vertical, 5

W

write, 17, 21, 22
writeln, 22