Prolog

Programmation en Logique

Introducción

Prolog es un lenguaje de programación **declarativo**, lo que significa que es un lenguaje basado en la declaración de condiciones, proposiciones, afirmaciones... En el caso de Prolog se hace la declaración de hechos y reglas.

Para responder a las preguntas o consultas formuladas por el programador, Prolog consulta una **base de conocimiento**. Esta base conocimiento representa el programa como tal, programa que se compone únicamente de cláusulas, que con el uso de la lógica, expresan el **conocimiento** deseado.

La **base de conocimiento** (o programa) se guarda en un archivo con la extensión `'.pl'`. Archivo que puede ser abierto y, a partir de él, hacer consultas al programa.

Install Documentation The Project Bundles Support

The Ciao programming language

Ciao is a programming language that builds up from a logic-based simple kernel, and is designed to be extensible and modular. Its supports:

- **constraint** logic programming (and, in particular, **Prolog**),
- different levels of modularity (from small to large scale):
  - **modules** as (analysis-friendly) compilation units

ation options, compiler

o work cooperatively in a

unctions, etc.) and interfacing

n, global program analysis and
rocessor), a build automation

SWI Prolog

Robust, mature, free. **Prolog for the real world.**

HOME | DOWNLOAD | DOCUMENTATION | TUTORIALS | COMMUNITY | USERS | WIKI

SWI-Prolog offers a comprehensive free Prolog environment. Since its start in 1987, SWI-Prolog development has been driven by the needs of real world applications. SWI-Prolog is widely used in research and education as well as commercial applications. Join over a million users who have downloaded SWI-Prolog. more ...

Download SWI-Prolog Get Started Try SWI-Prolog online

Code Archivo Editar Selección Ver Ir Depurar Terminal Ventana Ayuda MEM 79%

07-lista1.pl

07-lista1.pl × | 05-map.pl | 02-mortal.pl | 03-fruta.pl

Users > cayetanoguerraartal > Propio > Docencia > Prolog > 07-lista1.pl

```
10      elemento_k(L, K1, Result).
11
12  % Identificar si una lista es un palíndromo.
13  es_palindromo(L) :- reverse(L,L).
14
15  % Hallar el valor numérico máximo de una lista
16  max_list([L], L).
17  max_list([Cabeza|Cola], Max) :-
18      max_lis
19      Max is
20
21
22  % Definir u
23  es_arbol_bi
24  es_arbol_bi
25
26  preorder(ni
27  preorder(t(
28
29
30
31  inorder(nil
32  inorder(t(I
33
34
35
36  postorder(n
37  postorder(t
38
39
40
41
42  mi_arbol_binario(t(t(t(n
```

PROBLEMAS SALIDA CONSOLA DE D

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

1 ?- ["02-mortal.pl"].
true.

2 ?- mortal(socrates).
true.

3 ?-
```

```python
from pyswip import Prolog
prolog = Prolog()
prolog.assertz("father(michael,john)")
prolog.assertz("father(michael,gina)")
list(prolog.query("father(michael,X)")) == [{'X': 'john'}, {'X': 'gina'}]
for soln in prolog.query("father(X,Y)"):
    print(soln["X"], "is the father of", soln["Y"])
# michael is the father of john
# michael is the father of gina
```

Hola Mundo

Ejecutemos nuestro primer programa sobre la línea de comandos de Prolog:

```
write(“Hola, Mundo!”). %Esto es un comentario. Todas los predicados en Prolog terminan en un punto.
```

Hechos

Ejecutemos nuestro primer programa en Prolog:

```
assert(fruta(manzana)).
assert(fruta(pera)).
assert(fruta(plátano)).
```

Ahora preguntaremos si la pera es una fruta:

```
fruta(pera).
```

Preguntemos ahora si el kiwi es una fruta:

```
fruta(kiwi).
```

Preguntemos por todas las frutas:

```
fruta(X). %Todas las variables comienzan por
mayúsculas
```

Primeras reglas

Veamos el principio de funcionamiento de Prolog:

```
assert(mortal(X) :- hombre(X)).

assert(hombre(socrates)).
```

Preguntemos si Sócrates es mortal.

```
mortal(socrates).
```

Primer programa

En general, escribiremos nuestros programas en un editor, no en la línea de comandos de Prolog. Por tanto, escribamos nuestro primer programa:

```
padre_de(juan, ana).
padre_de(juan, pedro).
padre_de(juan, juan).

hermanos(X,Y) :- padre_de(Z,X), padre_de(Z,Y), X\==Y.
```

Guardamos el código con el nombre ‘misreglas.pl', entramos en ‘swipl’ y escribimos:

```
[‘misreglas.pl'].
```

Y preguntamos si:

```
hermanos(ana, juan).
```

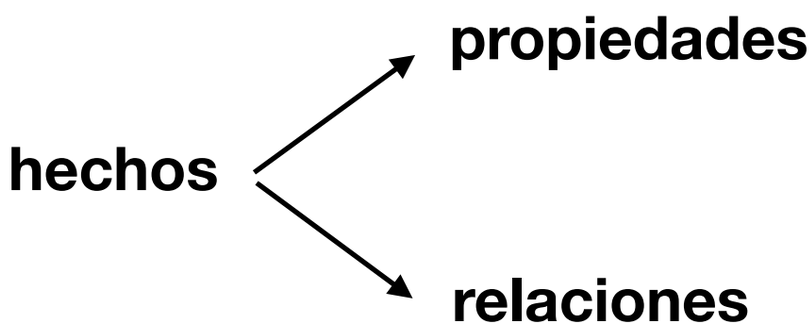
Formalismos

Hechos

Un **hecho** es un mecanismo para **representar propiedades o relaciones** de los objetos. Los hechos declaran los valores que van a ser verdaderos o afirmativos para un predicado en todo el programa.

Los hechos poseen la siguiente sintaxis:
nombre_predicado(argumentos).

Los hechos se dividen en dos tipos: **propiedades** y **relaciones**.



SWI Prolog

Hechos

Propiedades: llevan un solo argumento y de esta manera expresan alguna característica de los objetos. Por ejemplo:

```
color(azul). %azul es color - Denota la propiedad del
azul de ser un color

color(verde). %verde es color
```

```
padre(juan).%Juan es padre - Denota la propiedad que
tiene Juan de ser padre.

padre(pablo). % Pablo es padre
```

Hechos

Relaciones: se caracterizan por llevar más de un argumento y, de esta manera, expresan la vinculación que existe entre varios objetos. Por ejemplo:

```
padrede(juan, maria). % Juan es padre de maria - Este
hecho expresa una relacion de padre-hijo

padrede(pablo, juan). % Pablo es padre de juan

edad(juan, 30). % Juan tiene la edad de 30 años

edad(pablo, 50).
```

Consultas

Es el mecanismo para extraer conocimiento del programa. Una **consulta** está constituida por una o más **metas** u **objetivos** que Prolog debe resolver.

Hecho:

```
?- assert(amigos(pedro, antonio)).
```

Consulta:

```
?- amigos(pedro, antonio). %Consulta
true
```

Reglas

Cuando la verdad de un hecho depende de la verdad de otro hecho o de un grupo de hechos se usa una **regla.**

Las reglas permiten establecer relaciones más elaboradas entre objetos, donde se declaran las condiciones para que un predicado sea cierto.

La sintaxis para una regla es la siguiente:

CABEZA :- CUERPO

La forma de leer esta sintaxis es: “La cabeza es verdad si el cuerpo es verdad”.

Reglas

De esta manera, se obtendrá el valor de verdad de la cabeza con el valor que se obtenga en el cuerpo. Si el cuerpo resulta falso, la cabeza será falsa.

Por ejemplo:

```
hombre(socrates).

mortal(socrates) :- hombre(socrates).
```

Reglas

Las reglas así expuestas no tendrían una gran utilidad. Sin embargo, si podemos sustituir símbolos por variables su capacidad expresiva se amplía notablemente.

Por ejemplo:

```
hombre(socrates).
mortal(X) :- hombre(X).
```

Ahora cualquier hombre es, a su vez, mortal. En este caso, a la pregunta de si Sócrates es mortal, Prolog respondería `true`. Prolog ha asignado a la variable X el valor “socrates” en un proceso llamado **unificación**, que veremos un poco más adelante.

Reglas

Variables:

Las variables en Prolog no son variables en el sentido habitual, por eso las llamamos **variables lógicas.** Se escriben como una secuencia de caracteres alfabéticos comenzando siempre por mayúscula o subrayado.

```
X
_Hola
_
```

Pero no son variables:

```
variable
$Hola
p__
```

Reglas

Variables:

Existen variables sin nombre (anónimas), y se representan mediante el símbolo de subrayado _. Pero cuidado, aunque todas las variables anónimas se escriben igual, son todas distintas. Es decir, mientras que dos apariciones de la secuencia de caracteres `Hola` se refieren a la misma variable, dos apariciones de la secuencia `_` se refieren a variables distintas.

Reglas

Las reglas se pueden dividir en dos tipos, dependiendo de cómo se calcula el valor de verdad del cuerpo:

Conjunciones: Se usa una coma para relacionar los hechos del cuerpo de la regla con un **AND** lógico. Por ejemplo:

```
hermano(X, Y) :- padrede(Z, X), padrede(Z, Y), X\==Y.
```

Disyunciones: Se usa un punto y coma para relacionar los hechos del cuerpo con un **OR** lógico. Por ejemplo:

```
familiarde(A,B) :-
   padrede(A,B);
   hijode(A,B);
   hermanode(A,B).
```

Reglas recursivas

Prolog permite el uso de la **recursividad** cuando se están definiendo reglas. Esto es útil para definir reglas generales y más flexibles. Por ejemplo, si se quiere definir la regla `predecesor_de` se puede realizar, de forma **iterativa**, como se muestra a continuación:

```
antecesor_de(X,Y) :- padrede(X, Y).
antecesor_de(X,Y) :- padrede(X, Z), padrede(Z, Y).
antecesor_de(X,Y) :- padrede(X, Z1), padrede(Z1, Z2),
padrede(Z2, Y). %Bisabuelo
```

Pero como se puede ver en el anterior ejemplo, encontrar el antecesor de una persona genera mucho código. Una forma más adecuada sería mediante reglas **recurrentes**:

```
antecesor_de(X, Y) :- padrede(X, Y). % base
antecesor_de(X, Y) :- padrede(X, Z), antecesor_de(Z,
Y). % Paso recursivo
```

Reglas recursivas

Otro ejemplo clásico de **recursividad** es el cálculo del factorial de un numero:

```
factorial(0, 1). % caso base.
factorial(N, F) :-
 N>0,
 N1 is N - 1,
 factorial(N1, F1),
 F is N * F1. % Paso recursivo.
```

Resolución de consultas: Unificación

Para resolver consultas, Prolog intenta **unificar** algún **Hecho** o **Regla** con igual **predicado.** Si es posible, se realiza lo mismo con el Cuerpo de la Regla sustituyendo en cada objetivo también lo que se logró unificar.

Si no se puede resolver un **objetivo**, se retrocede mediante **backtracking** y se intenta con la siguiente alternativa. Si no existen alternativas disponibles, el objetivo de partida falla. Si se vacía la lista de objetivos, el objetivo queda resuelto.

Resolución de consultas: Unificación

Ejemplo 1:

```
padre(pablo, juan).
padre(pablo, andres).
hermano(A, B) :- padre(C, A), padre(C, B).
```

“Pablo es padre de Juan y de Andrés, ¿Juan es hermano de Andrés?”

Si consultamos `hermano(juan, andres)`, el proceso es el siguiente:

1. Se unifica con la regla `hermano(A, B)` y obtenemos: `hermano(juan, andres) :- padre(C, juan), padre(C, andres)`.
2. Ahora se tendrá que hallar C para completar el primer objetivo, para lo cual unificamos con el primer hecho, `padre(pablo, juan)`. Por tanto, **C = pablo**.
3. Ya que C está definido debemos evaluar si y `padre(pablo, andres)` es verdadero.
4. Se acaban los objetivos y todos fueron verdaderos, por tanto, Juan es hermano de Andrés.

Resolución de consultas: Unificación

Ejemplo:

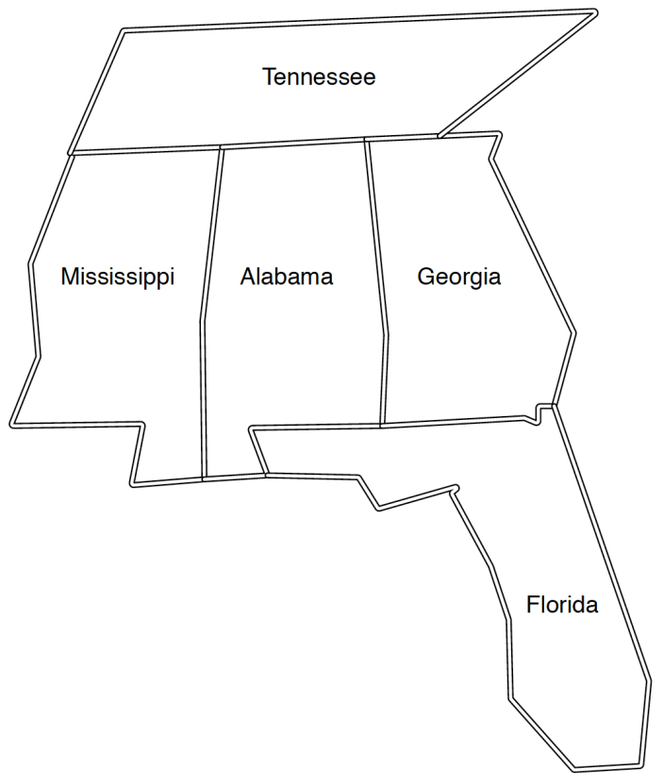
```
padre(juan,alberto).
padre(luis,alberto).
padre(alberto,leoncio).
padre(geronimo,leoncio).
padre(luisa,geronimo).
```

```
hermano(A,B) :-
  padre(A,P),
  padre(B,P),
  A \== B. % A y B no deben ser la misma persona

nieto(A,B) :-
  padre(A,P),
  padre(P,B).
```

Resolución de consultas: Unificación

Veamos la utilidad de la unificación con el problema del coloreado de mapas. En este problema nos encontramos con un conjunto de países o estados que tienen fronteras comunes y que debemos, por tanto, colorear con colores diferentes. También nos encontramos con la restricción de un número de colores disponibles limitado.

La forma de proceder en Prolog, a diferencia de otros lenguajes de paradigma imperativo, es que no vamos a "**resolver**" el problema sino a "**decribirlo**".



SWI Prolog

Resolución de consultas: Unificación

```
different(red, green). different(green, red).
different(red, blue). different(blue, red).
different(green, blue). different(blue, green).


coloring(Alabama, Mississippi, Georgia, Tennessee,
Florida) :-
    different(Alabama, Mississippi),
    different(Alabama, Tennessee),
    different(Alabama, Georgia),
    different(Alabama, Florida),
    different(Florida, Georgia),
    different(Tennessee, Georgia),
    different(Tennessee, Mississippi).
```

Resolución de consultas: Unificación

Una de las posibles soluciones sería:

```
?- coloring(Alabama, Mississippi, Georgia, Tennessee,
Florida).
Alabama = blue,
Mississippi = Georgia, Georgia = red,
Tennessee = Florida, Florida = green
```

Resolución de consultas

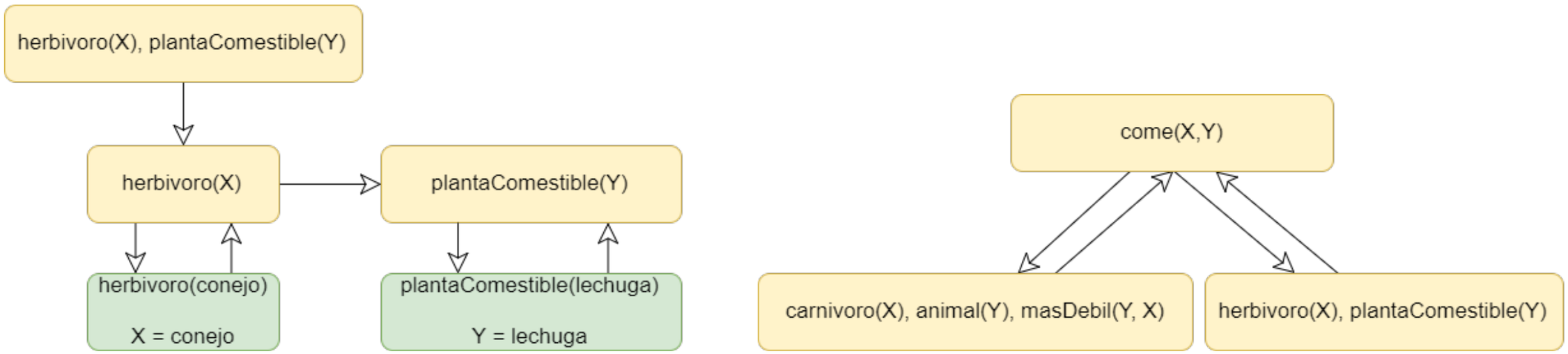
Ejemplo 2:

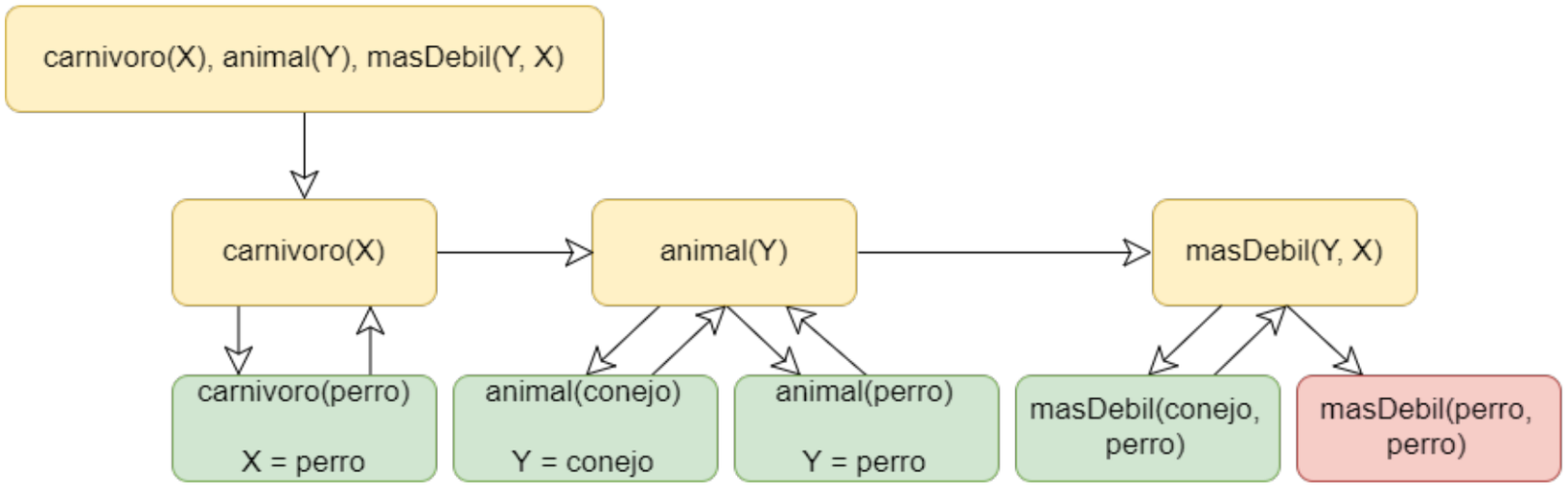
```
animal(conejo).
animal(perro).
carnivoro(perro).
masDebil(conejo, perro).

herbivoro(conejo).
plantaComestible(lechuga).

come(A,B) :-
    carnivoro(A), animal(B), masDebil(B, A);
    herbivoro(A), plantaComestible(B).
```

Si consultamos `come(X, Y)`, estaríamos preguntando para qué X e Y se cumple que X come a Y. El proceso es el siguiente:

Resolución de consultas

Ejemplo 2:





SWI Prolog

Expresiones

Los operadores nos permiten manipular diferentes tipos de datos.

Operadores aritméticos.
Con estos podemos llevar a cabo operaciones aritméticas entre números de tipo entero o real.

Suma	+
Resta	-
Multiplicación	*
División real y entera	/ y //
Potencia	^ y **
Positivo	+
Negativo	-

Expresiones

Operadores relacionales.

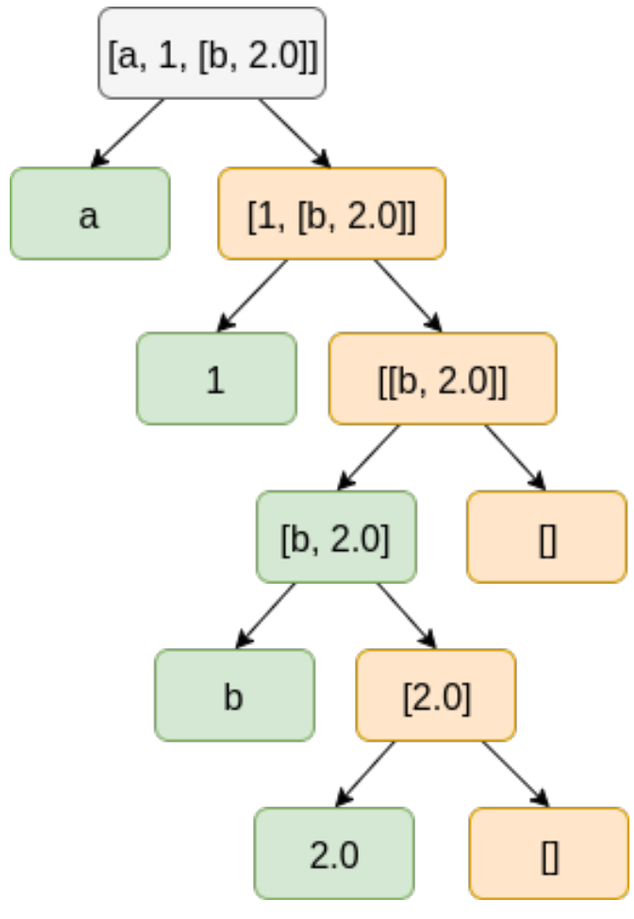
Este tipo de operadores reciben valores numéricos y/o expresiones antes de realizar unificaciones o comparaciones.

Operador	Significado	Ejemplo
is	Unificación	X is 10 + 2
=:=	Igualdad	10 + 2 =:= 5 + 7
=\=	Desigualdad	10 + 2 =\= 5 + 8
>	Mayor que	11 * 3 > 3 ^ 2
<	Menor que	2 ** 10 < 5 * 2
>=	Mayor o igual que	99.0 >= 0
=<	Menor o igual que	-15 =< 15

Listas

Las **listas** son estructuras que contienen una secuencia ordenada de cualquier tipo de términos. Está formada **recursivamente** por una **cabeza**, que es el primer elemento de la lista, y una **cola**, que es una lista con el resto de elementos. Por tanto, podemos definir una lista como un predicado `Lista(Cabeza, Cola)`.

La notación para las componentes de una lista es: [A | B], donde A es la cabeza y B es el cuerpo.



SWI Prolog

Listas: operadores

Las operaciones en listas nos permiten consultar alguna propiedad de una lista, así como realizar modificaciones.

Operador	Significado	Ejemplo
=	Unificación	[X, Y, Z] = [a, 1, 2.0]. [X, Y | Z] = [b, 2, 3.0].
member(term, list)	term ∈ list	member(4.0, [c, 3, 4.0]). member(X, [c, 3, 4.0]).
append(list1, list2, result)	Une list1 con list2	append([h, o], [l, a], X). append([h, o], X, [h, o, l, a]). append(X, [l, a], [h, o, l, a]). append(X, Y, [h, o, l, a]).
length(list, result)	Longitud de la lista	length([3, 0.0, x], X).
sort(list, result)	Ordena la lista	sort([4, a, 3], X).
is_list(term)	Comprueba si term es lista	is_list([1,2,3]).

Listas

Ejemplo 1: Extraer el último elemento de una lista.

```
ultimo([Result], Result). % Base
ultimo([_|L], Result) :- ultimo(L, Result).
```

Resultado:

```
?- ultimo([a, [b,c], 2], Ultimo).
Ultimo = 2 ;
```

Listas

Ejemplo 2: Extraer un elemento de una lista.

```
elemento_k([Result|_], 0, Result). % Base
elemento_k([_|L], K, Result) :-
   K > 0,
   K1 is K -1,
   elemento_k(L, K1, Result).
```

Resultado:

```
?- elemento_k([a, [b,c], 2], 1, Elemento).
Elemento = [b, c] ;
```

Listas

Ejemplo 3: Identificar si una lista es un palíndromo.

```
es_palindromo(L) :- reverse(L,L).
```

Resultado:

```
?- es_palindromo([a, b, a]).
true.
```

Listas

Ejemplo 4: Hallar el valor numérico máximo de una lista.

```
maximo_lista([L], L).
maximo_lista([Cabeza|Cola], Max) :-
   maximo_lista(Cola, MaxRec),
   Max is max(MaxRec, Cabeza).
```

Resultado:

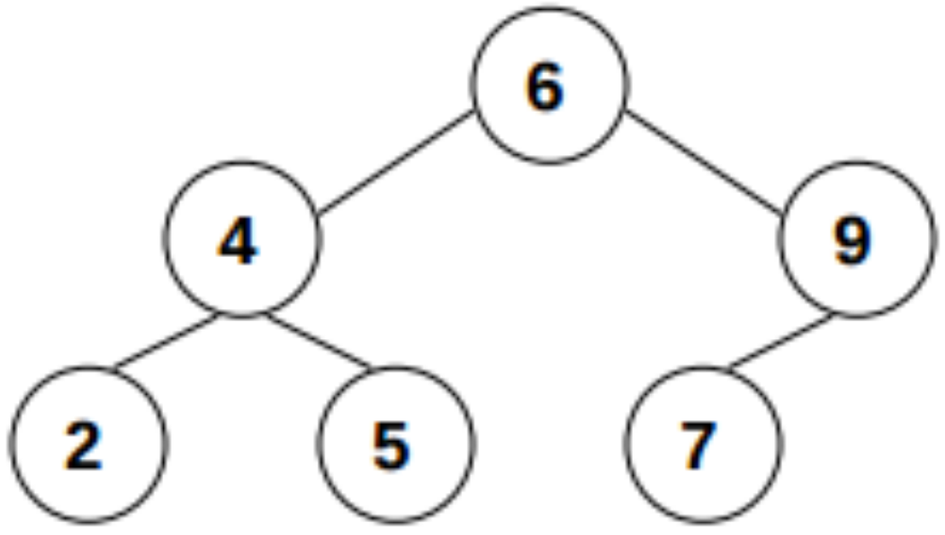
```
?- maximo_lista([4,5,3,7,8,4,3],Max).
Max = 8 ;
```

¡Ojo! *max_list* ya existe en Prolog

Listas

Ejemplo 5: Definir un árbol binario y sus recorridos **preorder**, **inorder** y **postorder**.

```
mi_arbol_binario(t(t(t(nil, 2, nil), 4, t(nil, 5,
nil)), 6, t(t(nil, 7, nil), 9, nil))).
```




```
preorder(nil, []).
preorder(t(Izquierda, Padre, Derecha), Lista):-
 preorder(Izquierda,ListaL),
 preorder(Derecha, ListaR),
 append([Padre | ListaL], ListaR, Lista).
```

SWI Prolog

Listas

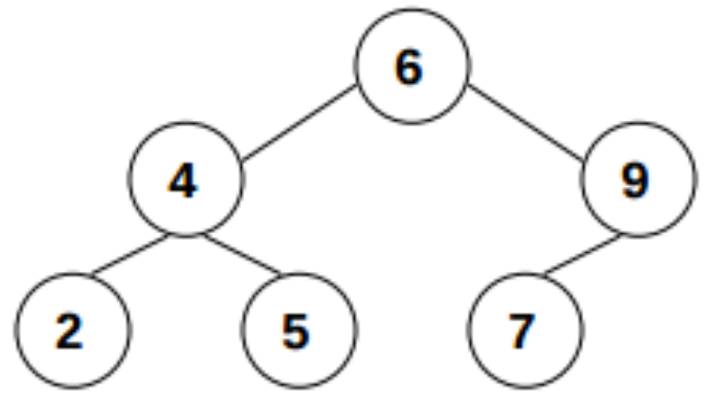
Ejemplo 5: Definir un árbol binario y sus recorridos **preorder**, **inorder** y **postorder**.

```
inorder(nil, []).
inorder(t(Izquierda, Padre, Derecha), Lista) :-
 inorder(Izquierda, ListaL),
 inorder(Derecha, ListaR),
 append(ListaL, [Padre | ListaR],Lista).
```

```
postorder(nil, []).
postorder(t(Izquierda, Padre, Derecha), Lista) :-
 postorder(Izquierda, ListaL),
 postorder(Derecha, ListaR),
 append(ListaL, ListaR, R1),
 append(R1, [Padre], Lista).
```

Listas

```
mi_arbol_binario(Arbol),
es_arbol_binario(Arbol),
preorder(Arbol, RecorridoPre),
inorder(Arbol, RecorridoIn),
postorder(Arbol, RecorridoPost).
```



Resultado:

```
?- Arbol = t(t(t(nil, 2, nil), 4, t(nil, 5, nil)), 6,
t(t(nil, 7, nil), 9, nil)),
RecorridoPre = [6, 4, 2, 5, 9, 7],
RecorridoIn = [2, 4, 5, 6, 7, 9],
RecorridoPost = [2, 5, 4, 7, 9, 6].
```

SWI Prolog

Listas: findall

```
person(pepe, 30, hombre).
person(ana, 24, mujer).
person(jose, 54, hombre).
person(pedro, 13, hombre).
person(maria, 38, mujer).

personas(Arg, LIST) :- findall(Nombre, person(Nombre,
_, Arg), LIST).
```

Acción:

```
?- personas(mujer, L).
```

Resultado:

```
L = [ana, maria]
```

Control

El programador puede afectar la ejecución de un programa cambiando el orden de las cláusulas, y también frenar el proceso de **backtracking** mediante el predicado "**cut**" (!).

La función principal del "**cut**" es reducir el espacio de búsqueda de soluciones cortándolo dinámicamente.

Para entender su uso fácilmente, consideremos los siguientes hechos:

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

Control

Si preguntamos:

```
?- teaches(dr_fred, Course), studies(Student, Course).

Course = english
Student = alice ;

Course = english
Student = angus ;

Course = drama
Student = amelia ;

false.
```

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

SWI Prolog

Control

El **backtracking** no se inhibe aquí. Inicialmente, el curso está vinculado a la historia, pero no hay estudiantes de historia, por lo que el segundo objetivo falla, se produce un retroceso. El curso se vuelve a vincular ahora al inglés, se intenta el segundo objetivo y se encuentran dos soluciones (Alice y Angus), luego se produce el backtracking de nuevo, y curso se une a drama, y se encuentra finalmente una estudiante, amelia.

```
?- teaches(dr_fred, Course), studies(Student, Course).

Course = english
Student = alice ;

Course = english
Student = angus ;

Course = drama
Student = amelia ;

false.
```

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

Control

Si ahora preguntamos:

```
?- teaches(dr_fred, Course),!,studies(Student,Course).

false.
```

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

Control

Si ahora preguntamos:

```
?- teaches(dr_fred, Course),!,studies(Student,Course).

false.
```

Esta vez, el curso está vinculado inicialmente a la historia, luego se ejecuta el corte (!), pero el objetivo *studies* falla (porque nadie estudia historia). Debido al corte, no podemos dar marcha atrás al objetivo de enseñanza para encontrar otro enlace para el curso, por lo que toda la consulta falla.

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

Control

¿Qué pasaría ahora?

```
?- teaches(dr_fred,Course),studies(Student,Course),!.

Course = english
Student = alice ;

false.
```

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

Control

¿Qué pasaría ahora?

```
teaches(dr_fred, history).
teaches(dr_fred, english).
teaches(dr_fred, drama).
teaches(dr_fiona, physics).

studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).
```

```
?- teaches(dr_fred,Course),studies(Student,Course),!.

Course = english
Student = alice ;

false.
```

Aquí el objetivo *teaches* se intenta como de costumbre, y *Course* se vincula a *history*, como de costumbre. Luego, el objetivo *studies* se prueba y falla, por lo que no llegamos al corte al final de la consulta en este punto y puede ocurrir un retroceso. Por lo tanto, el objetivo *teaches* se vincula ahora a *english*. Luego, *studies* se vuelve a intentar, y tiene éxito con *Student = alice*. Después de eso, se intenta el objetivo de corte y, por supuesto, tiene éxito, por lo que no es posible retroceder más y, por lo tanto, solo se encuentra una solución.

SWI Prolog

Negación

`not(X)` es la forma de implementar la negación en Prolog; sin embargo, `not(X)` no significa que **X** sea falso, significa que **X** no se puede probar como verdadero.

Por ejemplo, con la base de datos:

```
man('Adam').
woman('Eve').
```

preguntar `not(man('Adam')).` devolvería falso. Pero `not(man('Isaac')).` devolvería verdadero puesto que no se puede demostrar que Isaac sea hombre.

If… Then…

```
hombre(socrates).

go(X) :-
    (hombre(X) ->
        write(X), write(' es un hombre'), nl
        ;
        write(X), write(' no es un hombre'), nl
    ).
```