

Tema 2: Ampliación de Prolog

José A. Alonso Jiménez
Miguel A. Gutiérrez Naranjo
Francisco J. Martín Mateos

Dpto. de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE SEVILLA

Contenido

- Ampliación de Prolog
 - Resolución SLD
 - Poda de la búsqueda mediante corte
 - Negación como fallo
 - Acumuladores
 - Modificación de la base de conocimiento
 - Modificación dinámica de la BC
 - Todas las soluciones
 - Construcción y descomposición de términos
 - Predicados de segundo orden
 - Bibliografía

Resolución SLD

- **Ejemplo de resolución SLD:**

- **Programa:**

```
nieto(X,Z) :- hijo(X,Y), hijo(Y,Z).
hijo(X,Y) :- madre(Y,X).
hijo(X,Y) :- padre(Y,X).
padre(a,b).
madre(b,c).
```

- **Pregunta:**

```
:- nieto(c,X).
```

- **Resolución SLD:**

```
:- nieto(c,X0).
|   nieto(X1,Z1) :- hijo(X1,Y1), hijo(Y1,Z1).
|   {X1/c, Z1/X0}
:- hijo(c,Y1), hijo(Y1, X0).
|   hijo(X2,Y2) :- madre(Y2,X2).
|   {X2/c, Y2/Y1}
:- madre(Y1,c), hijo(Y1,X0).
|   madre(b,c).
|   {Y1/b}
:- hijo(c,X0).
|   hijo(X3,Y3) :- padre(Y3,X3).
|   {X3/b, Y3/X0}
:- padre(X0,b).
|   padre(a,b).
|   {X0/a}
:-.
```

- **Respuesta: {X/a}**

Resolución SLD

- **Siglas SLD:**
 - S: regla de Selección
 - L: resolución Lineal
 - D: cláusulas Definidas
- **Reglas de selección de Prolog:**
 - Selección de objetivos: De izquierda a derecha
 - Selección de reglas: Como en el programa
- **Arbol SLD:**

```
alumno(A,P) :- estudia(A,C), enseña(P,C).
estudia(ana,ia).
estudia(ana,pl).
estudia(eva,ra).
enseña(jose_a,ia).
enseña(jose_a,ra).
enseña(rafael,pl).
```

```

                                     :- alumno(A,jose_a).
                                     |
                                     :- estudia(A,C), enseña(jose_a,C).
                                     /      |      \
:- enseña(jose_a,ia).      |      :- enseña(jose_a,ra).
    |      :- enseña(jose_a,pl).      |
    :- .                                :- .
R = {A/ana}      Fallo      R = {A/eva}
```

Resolución SLD

● Sesión

?- alumno(A,jose_a).

A = ana ;

A = eva ;

No

?- trace.

Yes

?- alumno(A,jose_a).

Call: (7) alumno(_G178, jose_a) ?

Call: (8) estudia(_G178, _L130) ?

Exit: (8) estudia(ana, ia) ?

Call: (8) enseña(jose_a, ia) ?

Exit: (8) enseña(jose_a, ia) ?

Exit: (7) alumno(ana, jose_a) ?

A = ana ;

Redo: (8) enseña(jose_a, ia) ?

Fail: (8) enseña(jose_a, ia) ?

Redo: (8) estudia(_G178, _L130) ?

Exit: (8) estudia(ana, pl) ?

Call: (8) enseña(jose_a, pl) ?

Fail: (8) enseña(jose_a, pl) ?

Redo: (8) estudia(_G178, _L130) ?

Exit: (8) estudia(eva, ra) ?

Call: (8) enseña(jose_a, ra) ?

Exit: (8) enseña(jose_a, ra) ?

Exit: (7) alumno(eva, jose_a) ?

A = eva ;

No

Resolución SLD

- **Notas:**
 - Ramas de éxito y de fallo
 - Búsqueda en profundidad
- **Ramas infinitas en Prolog por predicados simétricos:**

- Programa ej-1.pl

```
hermano(X,Y) :- hermano(Y,X).  
hermano(b,a).
```

- **Sesión:**

```
?- [ej-1].
```

```
Yes
```

```
?- hermano(a,X).
```

```
[WARNING: Out of local stack]
```

```
Execution Aborted
```

- **Arbol:**

```
                :- hermano(a,X).  
                |  
                :- hermano(X,a).  
                /      \  
:- hermano(a,X).      :-.  
    |  
:- hermano(X,a).  
    /  \  
... :-.
```

Resolución SLD

- Programa ej-2.pl

```
hermano(b,a).  
hermano(X,Y) :- hermano(Y,X).
```

- Sesión:

```
?- [ej-2].  
Yes
```

```
?- hermano(a,X).  
X = b ;  
X = b ;  
Yes
```

- Arbol:

```
:- hermano(a,X).  
  |  
:- hermano(X,a).  
 /      \  
:-      :- hermano(a,X).  
                |  
                :- hermano(X,a).  
                /  \  
                :-  ...
```

Resolución SLD

- **Ramas infinitas en Prolog por predicados transitivos:**

- **Programa**

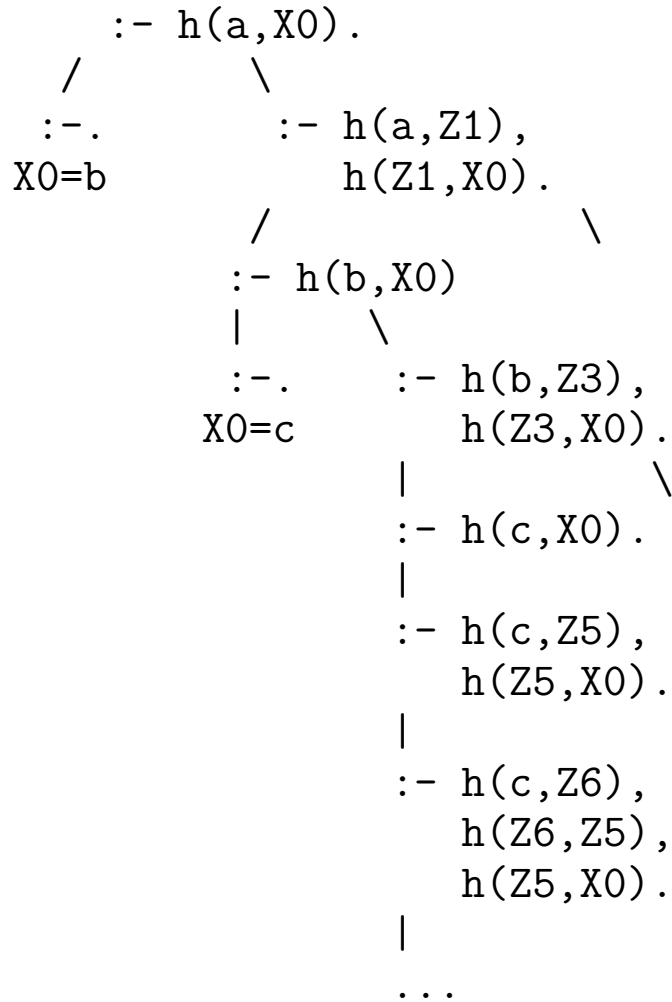
```
hermano(a,b).  
hermano(b,c).  
hermano(X,Y) :-  
    hermano(X,Z),  
    hermano(Z,Y).
```

- **Pregunta**

```
?- hermano(a,X).  
X = b ;  
X = c ;  
[WARNING: Out of local stack]  
Exception: (42407) hermano(c, _L508920) ? a  
Execution Aborted
```


Resolución SLD

- Arbol SLD



Poda de la búsqueda mediante corte

- Ejemplo nota

- Programa sin corte

```
nota(X,suspenso)      :- X < 5.  
nota(X,aprobado)     :- X >= 5, X < 7.  
nota(X,notable)      :- X >= 7, X < 9.  
nota(X,sobresaliente) :- X >= 9.
```

- Traza

```
?- trace.
```

```
Yes
```

```
?- nota(3,Y).
```

```
Call: ( 7) nota(3, _G98) ?
```

```
Call: ( 8) 3 < 5 ?
```

```
Exit: ( 8) 3 < 5 ?
```

```
Exit: ( 7) nota(3, suspenso) ?
```

```
Y = suspenso ;
```

```
Redo: ( 7) nota(3, _G98) ?
```

```
Call: ( 8) 3 >= 5 ?
```

```
Fail: ( 8) 3 >= 5 ?
```

```
Redo: ( 7) nota(3, _G98) ?
```

```
Call: ( 8) 3 >= 7 ?
```

```
Fail: ( 8) 3 >= 7 ?
```

```
Redo: ( 7) nota(3, _G98) ?
```

```
Call: ( 8) 3 >= 9 ?
```

```
Fail: ( 8) 3 >= 9 ?
```

```
Fail: ( 7) nota(3, _G98) ?
```

```
No
```

Poda de la búsqueda mediante corte

- Programa con corte

```
nota(X,suspenso)      :- X < 5, !.  
nota(X,aprobado)     :- X < 7, !.  
nota(X,notable)      :- X < 9, !.  
nota(X,sobresaliente).
```

- Traza

```
?- trace.
```

```
Yes
```

```
?- nota(3,Y).
```

```
    Call: ( 7) nota(3, _G101) ?
```

```
    Call: ( 8) 3 < 5 ?
```

```
    Exit: ( 8) 3 < 5 ?
```

```
    Exit: ( 7) nota(3, suspenso) ?
```

```
Y = suspenso;
```

```
No
```

```
?- trace.
```

```
Yes
```

```
?- nota(6,Y).
```

```
    Call: ( 7) nota(6, _G101) ?
```

```
    Call: ( 8) 6 < 5 ?
```

```
    Fail: ( 8) 6 < 5 ?
```

```
    Redo: ( 7) nota(6, _G101) ?
```

```
    Call: ( 8) 6 < 7 ?
```

```
    Exit: ( 8) 6 < 7 ?
```

```
    Exit: ( 7) nota(6, aprobado) ?
```

```
Y = aprobado;
```

```
No
```

Poda de la búsqueda mediante corte

- **Ventajas e inconvenientes del uso del corte**

- Aumento de eficiencia

- Pérdida de sentido declarativo. Ejemplo:

```
?- nota(6,sobresaliente).
```

```
Yes
```

- **Ejemplo maximo**

- Programa sin corte

```
maximo_1(X,Y,X) :- Y =< X.
```

```
maximo_1(X,Y,Y) :- X =< Y.
```

- Programa con corte

```
maximo_2(X,Y,X) :- Y =< X, !.
```

```
maximo_2(X,Y,Y).
```

- Sesión

```
?- maximo_1(3,5,X).
```

```
X = 5 ;
```

```
No
```

```
?- maximo_2(3,5,X).
```

```
X = 5 ;
```

```
No
```

```
?- maximo_1(3,2,2).
```

```
No
```

```
?- maximo_2(3,2,2).
```

```
Yes
```

Poda de la búsqueda mediante corte

- Ejemplo pertenece

- Programa sin corte

```
pertenece_1(X, [X|_]).  
pertenece_1(X, [_|L]) :- pertenece_1(X,L).
```

- Programa con corte

```
pertenece_2(X, [X|_]) :- !.  
pertenece_2(X, [_|L]) :- pertenece_2(X,L).
```

- Sesión

```
?- pertenece_1(a, [a,b,a]). => Yes  
?- pertenece_1(c, [a,b,a]). => No  
?- pertenece_1(X, [a,b,a]). => X = a; X = b; X = a; No  
?- pertenece_2(a, [a,b,a]). => Yes  
?- pertenece_2(c, [a,b,a]). => No  
?- pertenece_2(X, [a,b,a]). => X = a; No
```

- Predicados member y memberchk

Negación como fallo

- Introducción de la negación como fallo

- Programa 1

```
q1(a) :- q1(b), !, q1(c).
```

```
q1(a) :- q1(d).
```

```
q1(d).
```

- Traza

```
?- q1(a).
```

```
Call: ( 7) q1(a) ?
```

```
Call: ( 8) q1(b) ?
```

```
Fail: ( 8) q1(b) ?
```

```
Redo: ( 7) q1(a) ?
```

```
Call: ( 8) q1(d) ?
```

```
Exit: ( 8) q1(d) ?
```

```
Exit: ( 7) q1(a) ?
```

```
Yes
```

Negación como fallo

- Programa 2

```
q2(a) :- q2(b), !, q2(c).  
q2(a) :- q2(d).  
q2(d).  
q2(b).
```

- Traza

```
?- q2(a).  
  Call: ( 7) q2(a) ?  
  Call: ( 8) q2(b) ?  
  Exit: ( 8) q2(b) ?  
  Call: ( 8) q2(c) ?  
  Fail: ( 8) q2(c) ?  
  Fail: ( 7) q2(a) ?
```

No

- Comentario: No monotonía

Negación como fallo

- Programa 3: Def. de negación

```
q3(a) :- q3(b), q3(c).  
q3(a) :- no(q3(b)), q3(d).  
q3(d).
```

```
no(P) :- P, !, fail.  
no(P).
```

- Traza

```
?- q3(a).  
  Call: ( 7) q3(a) ?  
  Call: ( 8) q3(b) ?  
  Fail: ( 8) q3(b) ?  
  Redo: ( 7) q3(a) ?  
  Call: ( 8) no(q3(b)) ?  
  Call: ( 9) q3(b) ?  
  Fail: ( 9) q3(b) ?  
  Redo: ( 8) no(q3(b)) ?  
  Exit: ( 8) no(q3(b)) ?  
  Call: ( 8) q3(d) ?  
  Exit: ( 8) q3(d) ?  
  Exit: ( 7) q3(a) ?
```

Yes

Negación como fallo

- Programa 4

```
q4(a) :- q4(b), q4(c).  
q4(a) :- no(q4(b)), q4(d).  
q4(d).  
q4(b).
```

```
no(P) :- P, !, fail.  
no(P).
```

- Traza

```
?- q4(a).  
  Call: ( 7) q4(a) ?  
  Call: ( 8) q4(b) ?  
  Exit: ( 8) q4(b) ?  
  Call: ( 8) q4(c) ?  
  Fail: ( 8) q4(c) ?  
  Redo: ( 7) q4(a) ?  
  Call: ( 8) no(q4(b)) ?  
  Call: ( 9) q4(b) ?  
  Exit: ( 9) q4(b) ?  
  Call: ( 9) fail ?  
  Fail: ( 9) fail ?  
  Fail: ( 8) no(q4(b)) ?  
  Fail: ( 7) q4(a) ?
```

No

- Comentarios:

- Eficiencia y claridad
- Metapredicado primitivo not

Negación como fallo

- Problemas con negación como fallo

- Programa 1

```
aprobado(X) :- not(suspenso(X)), matriculado(X).  
matriculado(juan).  
matriculado(luis).  
suspenso(juan).
```

- Traza

```
?- aprobado(luis).  
  Call: ( 8) aprobado(luis) ?  
  Call: ( 9) not(suspenso(luis)) ?  
  Call: (10) suspenso(luis) ?  
  Fail: (10) suspenso(luis) ?  
  Redo: ( 9) not(suspenso(luis)) ?  
  Exit: ( 9) not(suspenso(luis)) ?  
  Call: ( 9) matriculado(luis) ?  
  Exit: ( 9) matriculado(luis) ?  
  Exit: ( 8) aprobado(luis) ?
```

Yes

```
?- aprobado(X).  
  Call: ( 8) aprobado(_G112) ?  
  Call: ( 9) not(suspenso(_G112)) ?  
  Call: (10) suspenso(_G112) ?  
  Exit: (10) suspenso(juan) ?  
  Call: (10) fail ?  
  Fail: (10) fail ?  
  Fail: ( 9) not(suspenso(_G112)) ?  
  Fail: ( 8) aprobado(_G112) ?
```

No

Negación como fallo

- Programa 2

```
aprobado(X) :- matriculado(X), not(suspenso(X)).
matriculado(juan).
matriculado(luis).
suspenso(juan).
```

- Traza

```
?- aprobado(X).
  Call: ( 8) aprobado(_G112) ?
  Call: ( 9) matriculado(_G112) ?
  Exit: ( 9) matriculado(juan) ?
  Call: ( 9) not(suspenso(juan)) ?
  Call: (10) suspenso(juan) ?
  Exit: (10) suspenso(juan) ?
  Call: (10) fail ?
  Fail: (10) fail ?
  Fail: ( 9) not(suspenso(juan)) ?
  Redo: ( 9) matriculado(_G112) ?
  Exit: ( 9) matriculado(luis) ?
  Call: ( 9) not(suspenso(luis)) ?
  Call: (10) suspenso(luis) ?
  Fail: (10) suspenso(luis) ?
  Redo: ( 9) not(suspenso(luis)) ?
  Exit: ( 9) not(suspenso(luis)) ?
  Exit: ( 8) aprobado(luis) ?
X = luis
Yes
```

- Consejo: Asegurar que not se llame con átomos básicos

Acumuladores

- Longitud de una lista

- Versión 1:

```
longitud_1([],0).
longitud_1(_|L,N):-
    longitud_1(L,N1),
    N is N1 +1.
```

- Traza 1:

```
?- trace(longitud_1).
    longitud_1/2: call redo exit fail
```

Yes

```
?- longitud_1([a,b,c],N).
```

```
T Call: ( 8) longitud_1([a, b, c], _G179)
```

```
T Call: ( 9) longitud_1([b, c], _L131)
```

```
T Call: (10) longitud_1([c], _L144)
```

```
T Call: (11) longitud_1([], _L157)
```

```
T Exit: (11) longitud_1([], 0)
```

```
T Exit: (10) longitud_1([c], 1)
```

```
T Exit: ( 9) longitud_1([b, c], 2)
```

```
T Exit: ( 8) longitud_1([a, b, c], 3)
```

```
N = 3
```

Yes

Acumuladores

- **Versión 2:**

```
longitud_2(L,N):-  
    longitud_2_aux(L,0,N).
```

```
longitud_2_aux([],N,N).  
longitud_2_aux(_|L,N,Long):-  
    N1 is N+1,  
    longitud_2_aux(L,N1,Long).
```

- **Traza**

```
?- trace([longitud_2,longitud_2_aux]).
```

```
?- longitud_2([a,b,c],N).
```

```
T Call: ( 8) longitud_2([a, b, c], _G179)  
T Call: ( 9) longitud_2_aux([a, b, c], 0, _G179)  
T Call: (10) longitud_2_aux([b, c], 1, _G179)  
T Call: (11) longitud_2_aux([c], 2, _G179)  
T Call: (12) longitud_2_aux([], 3, _G179)  
T Exit: (12) longitud_2_aux([], 3, 3)  
T Exit: (11) longitud_2_aux([c], 2, 3)  
T Exit: (10) longitud_2_aux([b, c], 1, 3)  
T Exit: ( 9) longitud_2_aux([a, b, c], 0, 3)  
T Exit: ( 8) longitud_2([a, b, c], 3)  
N = 3  
Yes
```

Acumuladores

- Inversa de una lista

- Versión 1:

```
inversa_1([], []).
inversa_1([X|L1],L2):-
    inversa_1(L1,L3),
    append(L3,[X],L2).
```

- Traza 1:

```
?- trace(inversa_1).
    inversa_1/2: call redo exit fail

?- inversa_1([a,b,c],L).
T Call: ( 7) inversa_1([a, b, c], _G212)
T Call: ( 8) inversa_1([b, c], _L172)
T Call: ( 9) inversa_1([c], _L186)
T Call: (10) inversa_1([], _L200)
T Exit: (10) inversa_1([], [])
T Exit: ( 9) inversa_1([c], [c])
T Exit: ( 8) inversa_1([b, c], [c, b])
T Exit: ( 7) inversa_1([a, b, c], [c, b, a])
L = [c, b, a]
Yes
```

- Versión 2:

```
inversa_2(L1,L2):-
    inversa_2_aux(L1,[],L2).

inversa_2_aux([],L,L).
inversa_2_aux([X|L],Acum, Sal):-
    inversa_2_aux(L,[X|Acum],Sal).
```

Acumuladores

- Traza 2:

```
?- trace([inversa_2, inversa_2_aux]).
      inversa_2/2: call redo exit fail
      inversa_2_aux/3: call redo exit fail
```

Yes

```
?- inversa_2([a,b,c],L).
T Call: ( 7) inversa_2([a, b, c], _G212)
T Call: ( 8) inversa_2_aux([a, b, c], [], _G212)
T Call: ( 9) inversa_2_aux([b, c], [a], _G212)
T Call: (10) inversa_2_aux([c], [b, a], _G212)
T Call: (11) inversa_2_aux([], [c, b, a], _G212)
T Exit: (11) inversa_2_aux([], [c, b, a], [c, b, a])
T Exit: (10) inversa_2_aux([c], [b, a], [c, b, a])
T Exit: ( 9) inversa_2_aux([b, c], [a], [c, b, a])
T Exit: ( 8) inversa_2_aux([a, b, c], [], [c, b, a])
T Exit: ( 7) inversa_2([a, b, c], [c, b, a])
L = [c, b, a]
Yes
```

- Comparación:

```
?- setof(_N,between(1,1000,_N),_L1),
   time(inversa_1(_L1,_)).
501,503 inferences in 6.80 seconds (73750 Lips)
Yes
```

```
?- setof(_N,between(1,1000,_N),_L1),
   time(inversa_2(_L1,_)).
1,004 inferences in 0.01 seconds (100400 Lips)
Yes
```

Listas de diferencia

- Representaciones de $[a,b,c]$ como listas de diferencias

$[a,b,c,d] - [d]$

$[a,b,c,1,2,3] - [1,2,3]$

$[a,b,c|X] - [X]$

$[a,b,c] - []$

- Concatenación de listas de diferencias

- Programa

```
conc_ld(X-RX,Y-RY,X-RY) :- RX=Y.
```

- Sesión

```
?- conc_ld([a,b|RX]-RX,[c,d|RY]-RY,Z-[]).
```

```
RX = [c, d]
```

```
RY = []
```

```
Z = [a, b, c, d]
```

```
Yes
```

```
?- conc_ld([a,b|_RX]-_RX,[c,d|_RY]-_RY,Z-[]).
```

```
Z = [a, b, c, d]
```

```
Yes
```


Listas de diferencia

- Procedimiento inversa con listas de diferencias

- Programa

```
inversa(L1,L2) :-  
    inversa_ld(L1,L2-[]).  
  
inversa_ld([],L2-L2).  
inversa_ld([X|L1],L2-L3) :-  
    inversa_ld(L1,L2-[X|L3]).
```

- Traza

```
?- trace.  
Yes
```

```
?- inversa([a,b],L).  
Call: ( 8) inversa([a, b], _G149) ?  
Call: ( 9) inversa_ld([a, b], _G149 - []) ?  
Call: (10) inversa_ld([b], _G149 - [a]) ?  
Call: (11) inversa_ld([], _G149 - [b, a]) ?  
Exit: (11) inversa_ld([], [b, a] - [b, a]) ?  
Exit: (10) inversa_ld([b], [b, a] - [a]) ?  
Exit: ( 9) inversa_ld([a, b], [b, a] - []) ?  
Exit: ( 8) inversa([a, b], [b, a]) ?
```

```
L = [b, a]  
Yes
```

Modificación de la B.C.

- Predicados assert y retract
 - assert(+Term) inserta un hecho o una cláusula en la base de conocimientos. Term es insertado como última cláusula del predicado correspondiente.
 - retract(+Term) elimina la primera cláusula de la base de conocimientos que unifica con Term
- Ejemplos

```
?- hace_frio.  
[WARNING: Undefined predicate: 'hace_frio/0']  
No  
?- assert(hace_frio).  
Yes  
?- hace_frio.  
Yes  
?- retract(hace_frio).  
Yes  
?- hace_frio.  
No
```

Modificación de la B.C.

- El predicado listing

- listing(+Pred) lista las cláusulas en cuya cabeza aparece el predicado Pred

- Ejemplos

```
?- listing(member).
member(A, [A|B]).
member(A, [B|C]) :- member(A, C).
Yes
?- assert( (gana(X,Y) :- rapido(X), lento(Y))).
X = _G445
Y = _G446
Yes
?- listing(gana).
gana(A, B) :-
    rapido(A),
    lento(B).
Yes
?- assert(rapido(juan)),assert(lento(jose)),
    assert(lento(luis)).
Yes
?- gana(X,Y).
X = juan   Y = jose ;
X = juan   Y = luis ;
No
?- retract(lento(X)).
X = jose ;
X = luis ;
No
?- gana(X,Y).
No
```

Modificación de la B.C.

- Los predicados `asserta` y `assertz`
 - `asserta(+Term)` equivale a `assert/1`, pero `Term` es insertado como primera cláusula del predicado correspondiente
 - `assertz(+Term)` equivale a `assert/1`
 - Ejemplos

```
?- assert(p(a)), assertz(p(b)), asserta(p(c)).
```

```
Yes
```

```
?- p(X).
```

```
X = c ;
```

```
X = a ;
```

```
X = b ;
```

```
No
```

```
?- listing(p).
```

```
p(c).
```

```
p(a).
```

```
p(b).
```

```
Yes
```

Modificación de la B.C.

- Los predicados `retractall` y `abolish`
 - `retractall(+Head)` elimina de la base de conocimientos todas las cláusulas cuya cabeza unifica con `Head`
 - `abolish(+SimbPred/+Aridad)` elimina de la base de conocimientos todas las cláusulas que en su cabeza aparece el símbolo de predicado `SimbPred/Aridad`
 - `abolish(+SimbPred, +Aridad)` es equivalente a `abolish(+SimbPred/+Aridad)`

- Ejemplo

```
?- assert(p(a)), assert(p(b)).
Yes
?- retractall(p(_)).
Yes
?- p(a).
No
?- assert(p(a)), assert(p(b)).
Yes
?- abolish(p/1).
Yes
?- p(a).
[WARNING: Undefined predicate: 'p/1']
No
```

Modificación de la B.C.

```
?- assert(f(a,b)).
Yes
?- f(X,Y).
X = a    Y = b ;
?- asserta(f(a,a)),assertz(f(b,b)).
Yes
?- f(X,Y).
X = a    Y = a ;
X = a    Y = b ;
X = b    Y = b ;
?- listing(f).
f(a, a).
f(a, b).
f(b, b).
?- retract(f(_,a)).
Yes
?- listing(f).
f(a, b).
f(b, b).
?- assert(f(b,a)).
Yes
?- listing(f).
f(a, b).
f(b, b).
f(b, a).
?- retractall(f(b,_)).
Yes
?- listing(f).
f(a, b).
?- abolish(f/2).
Yes
?- listing(f).
[WARNING: No predicates for 'f']
No
```

Modificación de la B.C.

- **Multiplicaciones (tabla.pl)**

- `crea_tabla` añade los hechos `producto(X,Y,Z)` donde X e Y son números de 0 a 9 y Z es el producto de X e Y.

```
crea_tabla :-  
    L = [0,1,2,3,4,5,6,7,8,9],  
    member(X,L),  
    member(Y,L),  
    Z is X*Y,  
    assert(producto(X,Y,Z)),  
    fail.  
crea_tabla.
```

- **Sesión:**

```
?- crea_tabla.  
Yes  
?- listing(producto).  
producto(0, 0, 0). producto(0, 1, 0). ...  
... producto(9, 8, 72). producto(9, 9, 81).  
Yes
```

- **Determinar las descomposiciones de 6 en producto de dos números.**

```
?- producto(A,B,6).  
A = 1      A = 2      A = 3      A = 6  
B = 6 ;    B = 3 ;    B = 2 ;    B = 1 ;  
No
```

Modificación dinámica de la BC

- Programa

```
:- dynamic r/2.
```

```
relacionados(X,Y) :-  
    assert(r(X,Y)).
```

```
no_relacionados(X,Y) :-  
    retract(r(X,Y)).
```

- Sesión

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

No

```
?- relacionados(a,b).
```

Yes

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

X = a

Y = b ;

No

```
?- no_relacionados(a,b).
```

Yes

```
?- r(a,b).
```

No

```
?- relacionados(a,b), relacionados(c,d).
```

Yes

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

X = a

Y = b ;

X = c

Y = d ;

No

Todas las soluciones

- El predicado `findall`

```
?- assert(clase(a,voc)),
    assert(clase(b,con)),
    assert(clase(e,voc)),
    assert(clase(c,con)).
```

Yes

```
?- findall(X,clase(X,voc),L).
```

```
X = _G331
```

```
L = [a, e]
```

Yes

```
?- findall(_X,clase(_X,voc),L).
```

```
L = [a, e]
```

Yes

```
?- findall(_X,clase(_X,_Clase),L).
```

```
L = [a, b, e, c]
```

Yes

```
?- findall(X,clase(X,vocal),L).
```

```
X = _G355
```

```
L = []
```

Yes

```
?- findall(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
```

```
X = _G373
```

```
L = [c, b, c]
```

Yes

```
?- findall(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
```

```
X = _G373
```

```
L = []
```

Yes

Todas las soluciones

- El predicado bagof

```
?- bagof(X, clase(X, voc), L).
```

```
X = _G331
```

```
L = [a, e]
```

```
Yes
```

```
?- bagof(X, clase(X, Clase), L).
```

```
X = _G343   Clase = voc   L = [a, e] ;
```

```
X = _G343   Clase = con   L = [b, c] ;
```

```
No
```

```
% L = {X: (existe Y)[clase(X,Y)]}
```

```
?- bagof(X, Y^clase(X,Y), L).
```

```
X = _G379   Y = _G380   L = [a, b, e, c] ;
```

```
No
```

```
?- bagof(_X, _Y^clase(_X, _Y), L).
```

```
L = [a, b, e, c] ;
```

```
No
```

```
?- bagof(letra(_X), _Y^clase(_X, _Y), L).
```

```
L = [letra(a), letra(b), letra(e), letra(c)]
```

```
Yes
```

```
?- bagof(X, clase(X, vocal), L).
```

```
No
```

```
?- bagof(X, (member(X, [c, b, c]), member(X, [c, b, a]))), L).
```

```
X = _G361
```

```
L = [c, b, c] ;
```

```
No
```

```
?- bagof(X, (member(X, [c, b, c]), member(X, [1, 2, 3]))), L).
```

```
No
```

Todas las soluciones

- El predicado setof

```
?- setof(X, clase(X, voc), L).
```

```
X = _G331
```

```
L = [a, e]
```

```
Yes
```

```
?- setof(X, clase(X, Clase), L).
```

```
X = _G343   Clase = voc   L = [a, e] ;
```

```
X = _G343   Clase = con   L = [b, c] ;
```

```
No
```

```
% L = {X: (existe Y)[clase(X,Y)]}
```

```
?- setof(X, Y^clase(X,Y), L).
```

```
X = _G379   Y = _G380   L = [a, b, c, e] ;
```

```
No
```

```
?- setof(_X, _Y^clase(_X, _Y), L).
```

```
L = [a, b, c, e] ;
```

```
No
```

```
?- setof(letra(_X), _Y^clase(_X, _Y), L).
```

```
L = [letra(a), letra(b), letra(c), letra(e)]
```

```
Yes
```

```
?- setof(X, clase(X, vocal), L).
```

```
No
```

```
?- setof(X, (member(X, [c, b, c]), member(X, [c, b, a]))), L).
```

```
X = _G361
```

```
L = [b, c]
```

```
Yes
```

```
?- setof(X, (member(X, [c, b, c]), member(X, [1, 2, 3]))), L).
```

```
No
```

Todas las soluciones

• Operaciones conjuntistas

- `interseccion(S,T,U)` se verifica si `U` es la intersección de `S` y `T`. Por ejemplo,

```
?- interseccion([1,4,2],[2,3,4],U).  
U = [2,4]
```

```
interseccion(S,T,U) :-  
    setof(X, (member(X,S), member(X,T)), U).
```

- `diferencia(S,T,U)` se verifica si `U` es la diferencia de los conjuntos de `S` y `T`. Por ejemplo,

```
?- diferencia([5,1,2],[2,3,4],U).  
U = [1,5]
```

```
diferencia(S,T,U) :-  
    setof(X, (member(X,S), not(member(X,T))), U).
```

- `n_union(S,T,U)` se verifica si `U` es la unión de `S` y `T`. Por ejemplo,

```
?- n_union([1,2,4],[2,3,4],U).  
U = [1,2,3,4]
```

```
n_union(S,T,U) :-  
    setof(X, (member(X,S); member(X,T)), U).
```

Todas las soluciones

- `partes(X,L)` se verifica si `L` es el conjunto de las partes de `X`. Por ejemplo,

```
?- partes([a,b],L).  
L = [[], [a], [a, b], [b]]
```

```
partes(X,L) :-  
    setof(Y,subconjunto(Y,X),L).
```

```
subconjunto([],[]).  
subconjunto([X|L1],[X|L2]) :-  
    subconjunto(L1,L2).  
subconjunto(L1,[_|L2]) :-  
    subconjunto(L1,L2).
```

Manipulación de términos

- El predicado =..
 - ?T =.. ?L se verifica si L es una lista cuya primer elemento es el functor del término T y los restantes elementos de L son los argumentos de T
 - Ejemplos
 - ?- padre(juan,luis) =.. L.
L = [padre, juan, luis]
 - ?- T =.. [padre, juan, luis].
T = padre(juan,luis)
- Figuras proporcionales
 - Las figuras geométricas se representan como términos en los que el functor indica el tipo de figura y los argumentos su tamaño. Por ejemplo,
 - cuadrado(3)
 - triangulo(3,4,5)
 - circulo(2)
 - alarga(Figura1,Factor,Figura2) se verifica si Figura1 y Figura2 son figuras geométricas del mismo tipo y el tamaño de la Figura2 es el de la Figura1 multiplicado por Factor.

Manipulación de términos

- Ejemplo

```
?- alarga(triangulo(3,4,5),2,F).  
F = triangulo(6, 8, 10)
```

```
?- alarga(cuadrado(3),2,F).  
F = cuadrado(6)
```

- Programa alarga.pl

```
alarga(Figura1,Factor,Figura2) :-  
    Figura1 =.. [Tipo|Argumentos1],  
    multiplica_lista(Argumentos1,Factor,Argumentos2),  
    Figura2 =.. [Tipo|Argumentos2].
```

```
multiplica_lista([],_,[]).  
multiplica_lista([X1|L1],F,[X2|L2]) :-  
    X2 is X1*F,  
    multiplica_lista(L1,F,L2).
```

Predicados de segundo orden

- El predicado `apply`

- `n_apply(+Term,+Lista)` se verifica si es demostrable `Term` después de aumentar el número de sus argumentos con los elementos de `Lista`

- Ejemplo

```
?- plus(1,2,X).
```

```
X = 3
```

```
Yes
```

```
?- n_apply(plus,[1,2,X]).
```

```
X = 3 ;
```

```
No
```

```
?- n_apply(plus(1),[2,X]).
```

```
X = 3
```

```
Yes
```

```
?- n_apply(plus(1,2),[X]).
```

```
X = 3
```

```
Yes
```

```
?- n_apply(append([1,2]),[X,[1,2,3,4,5]]).
```

```
X = [3, 4, 5] ;
```

```
No
```

- Programa `n_apply(+Term,+List)`

```
n_apply(Term,List):-
```

```
    Term =.. [Pred|Arg1],
```

```
    append(Arg1,List,Arg2),
```

```
    Atomo =.. [Pred|Arg2],
```

```
    Atomo.
```

- El predicado predefinido `apply`

Predicados de segundo orden

- Patrones aplicativos y maplist

- padre(X,P) se verifica si P es el padre de X

```
padre(beatriz, andres).  
padre(david, carlos).  
padre(elisa, ernesto).
```

- madre(X,M) se verifica si M es la madre de X

```
madre(beatriz, maria).  
madre(david, eva).  
madre(elisa, carmen).
```

- padres(L1,L2) se verifica si cada elemento de L2 es el padre del correspondiente elemento de L1. Por ejemplo,

```
?- padres([beatriz,david,elisa],L).  
L = [andres, carlos, ernesto]  
Yes
```

```
padres([], []).  
padres([X|R1], [P|R2]):-  
    padre(X,P),  
    padres(R1,R2).
```

Predicados de segundo orden

- `madres(L1,L2)` se verifica si cada elemento de `L2` es la madre del correspondiente elemento de `L1`. Por ejemplo,

```
?- madres([beatriz,david,elisa],L).  
L = [maria, eva, carmen]  
Yes
```

```
madres([], []).  
madres([X|R1], [M|R2]):-  
    madre(X,M),  
    madres(R1,R2).
```

Predicados de segundo orden

- El predicado `maplist`

- `maplist(+Predicado, ?Lista1, ?Lista2)` se verifica si se verifica el Predicado sobre los sucesivos pares de elementos de la `Lista1` y la `Lista2`. Por ejemplo,

```
?- maplist(succ,[2,4],[3,5]).
```

```
Yes
```

```
?- maplist(succ,[2,4],Y).
```

```
Y = [3, 5]
```

```
Yes
```

```
?- maplist(succ,X,[3,5]).
```

```
X = [2, 4]
```

```
Yes
```

```
?- maplist(succ,[0,4],[3,5]).
```

```
No
```

- Preguntas con `maplist`

```
?- maplist(padre,[beatriz,david,elisa],L).
```

```
L = [andres, carlos, ernesto]
```

```
Yes
```

```
?- maplist(madre,[beatriz,david,elisa],L).
```

```
L = [maria, eva, carmen]
```

```
Yes
```

- Definición de `maplist`

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

```
n_maplist(R, [X1|L1], [X2|L2]) :-
```

```
    apply(R, [X1,X2]),
```

```
    n_maplist(R,L1,L2).
```

Bibliografía

- Bratko, I. *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
- Clocksin, W.F. y Mellish, C.S. *Programming in Prolog (Fourth ed.)* (Springer, 1994)
- Flach, P. *Simply Logical (Intelligent Reasoning by Example)* (John Wiley, 1994)
 - Cap. 3: “Logic programming and Prolog”.
- Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)
 - Ap. B: “The Prolog programming language”