Programación lógica y Prolog

José A. Alonso

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Introducción

- **Significados de una cláusula definida**

  ```
  A :- B1,...,Bn
  ```

  - **Significado declarativo**
  - **Significado proocedimental**

- **Paradigma de la programación lógica: Ecuación de Kowalski:**

  algoritmo = lógica + control

- **Prolog**
  - **Lenguaje de programación lógica**
  - **Impureza declarativa de Prolog**

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:**

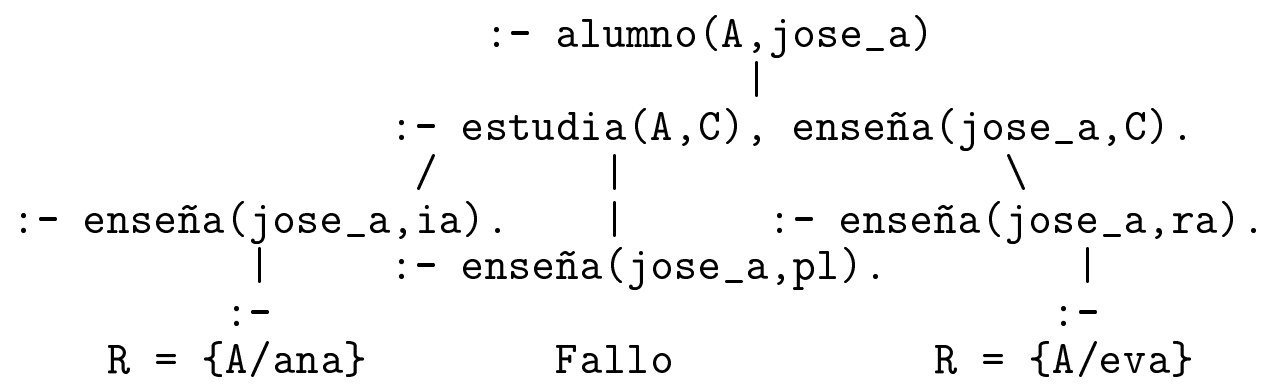
```
:- nieto(c,X0)
|    nieto(X1,Z1) :- hijo(X1,Y1), hijo(Y1,Z1)
|    {X1/c, Z1/X0}
:- hijo(c,Y1), hijo(Y1, X0)
|    hijo(X2,Y2) :- madre(Y2,X2).
|    {X2/c, Y2/Y1}
:- madre(Y1,c), hijo(Y1,X0)
|    madre(b,c).
|    {Y1/c}
:- hijo(c,X0)
|    hijo(X3,Y3) :- padre(Y3,X3)
|    {X3/c, Y3/X0}
:- padre(X0,c).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



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`

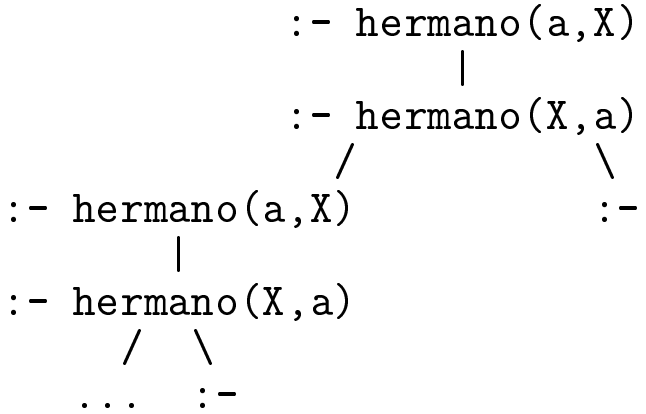
```
hermano(X,Y) :- hermano(Y,X).
hermano(b,a).
```

  - **Sesión:**

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

?- hermano(a,X).
[WARNING: Out of local stack]
Execution Aborted
```

  - **Arbol:**



Resolución SLD

- **Programa** `ej-2.pl`

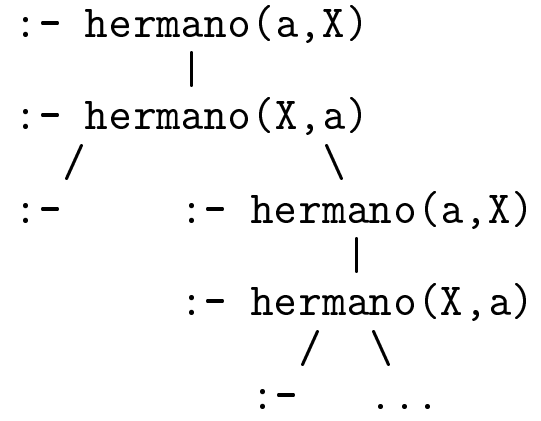
```
hermano(b,a).
hermano(X,Y) :- hermano(Y,X).
```

- **Sesión:**

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

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

- **Arbol:**



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
```

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
Yes
?- 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
Yes
```

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
```

Negación como fallo

- **Negación con corte**
  - **Programa:**

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

  - **Sesión:**

```
?- trace.
Yes

?- q(a).
   Call:  (  7) q(a) ?
   Call:  (  8) q(b) ?
   Fail:  (  8) q(b) ?
   Redo:  (  7) q(a) ?
   Call:  (  8) q(d) ?
   Exit:  (  8) q(d) ?
   Exit:  (  7) q(a) ?
Yes
```

- **Negación como fallo**
  - **Programa**

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

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

Negación como fallo

- **Sesión:**

```
?- q(a).
Yes

?- trace.
Yes

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

- **Comentarios: eficiencia y claridad**

Negación como fallo

- **Ejemplo con fallo de la negación**
  - **Programa**

```
q(a) :- no(q(b)), q(d).
q(a) :- q(b).
q(b).

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

  - **Sesión**

```
?- trace.
Yes

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

- **Metapredicado primitivo** `not`

Negación como fallo

- **Problemas con negación como fallo**
  - **Programa**

```
aprobado(X) :- no(suspenso(X)), matriculado(X).
matriculado(juan).
matriculado(luis).
suspenso(juan).

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

  - **Sesión**

```
?- trace.
Yes

?- aprobado(luis).
   Call:  (  8) aprobado(luis) ?
   Call:  (  9) no(suspenso(luis)) ?
   Call:  ( 10) suspenso(luis) ?
   Fail:  ( 10) suspenso(luis) ?
   Redo:  (  9) no(suspenso(luis)) ?
   Exit:  (  9) no(suspenso(luis)) ?
   Call:  (  9) matriculado(luis) ?
   Exit:  (  9) matriculado(luis) ?
   Exit:  (  8) aprobado(luis) ?
Yes
```

Negación como fallo

```
?- trace.
Yes

?- aprobado(X).
   Call:  (  8) aprobado(_G112) ?
   Call:  (  9) no(suspenso(_G112)) ?
   Call:  ( 10) suspenso(_G112) ?
   Exit:  ( 10) suspenso(juan) ?
   Call:  ( 10) fail ?
   Fail:  ( 10) fail ?
   Fail:  (  9) no(suspenso(_G112)) ?
   Fail:  (  8) aprobado(_G112) ?
No
```

- **Interpretación de cláusula**

  ```
  aprobado(X) :- no(suspenso(X)), matriculado(X).
  ```

  "X está aprobado si nadie está suspenso y X está matriculado"

- **Cambio del orden de literales**

```
aprobado(X) :- matriculado(X), no(suspenso(X)).
matriculado(juan).
matriculado(luis).
suspenso(juan).

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

Negación como fallo

- **Sesión**

```
?- trace.
Yes

?- aprobado(X).
   Call:  (  8) aprobado(_G112) ?
   Call:  (  9) matriculado(_G112) ?
   Exit:  (  9) matriculado(juan) ?
   Call:  (  9) no(suspenso(juan)) ?
   Call:  ( 10) suspenso(juan) ?
   Exit:  ( 10) suspenso(juan) ?
   Call:  ( 10) fail ?
   Fail:  ( 10) fail ?
   Fail:  (  9) no(suspenso(juan)) ?
   Redo:  (  9) matriculado(_G112) ?
   Exit:  (  9) matriculado(luis) ?
   Call:  (  9) no(suspenso(luis)) ?
   Call:  ( 10) suspenso(luis) ?
   Fail:  ( 10) suspenso(luis) ?
   Redo:  (  9) no(suspenso(luis)) ?
   Exit:  (  9) no(suspenso(luis)) ?
   Exit:  (  8) aprobado(luis) ?
X = luis
Yes
```

- **Consejo: Asegurar que `not` se llame con átomos básicos**

Otros usos del corte

- **Programa con alternativas**
  - **Programa**

```
a(p) :- a(q), a(r), a(s), !, a(t).
a(p) :- a(q), a(r), a(u).
a(q).
a(r).
a(u).
```

  - **Sesión**

```
?- trace.
Yes

?- a(p).
   Call:  (  8) a(p) ?
   Call:  (  9) a(q) ?
   Exit:  (  9) a(q) ?
   Call:  (  9) a(r) ?
   Exit:  (  9) a(r) ?
   Call:  (  9) a(s) ?
   Fail:  (  9) a(s) ?
   Redo:  (  8) a(p) ?
   Call:  (  9) a(q) ?
   Exit:  (  9) a(q) ?
   Call:  (  9) a(r) ?
   Exit:  (  9) a(r) ?
   Call:  (  9) a(u) ?
   Exit:  (  9) a(u) ?
   Exit:  (  8) a(p) ?
Yes
```

Otros usos del corte

- **Programa con `if_then_else`**
  - **Programa**

```
a(p) :- a(q), a(r), if_then_else(a(s),a(t),a(u)).
a(q).
a(r).
a(u).

if_then_else(X,Y,Z) :- X, !, Y.
if_then_else(X,Y,Z) :- Z.
```

  - **Sesión**

```
?- trace.
Yes

?- a(p).
   Call:  (  8) a(p) ?
   Call:  (  9) a(q) ?
   Exit:  (  9) a(q) ?
   Call:  (  9) a(r) ?
   Exit:  (  9) a(r) ?
   Call:  (  9) if_then_else(a(s), a(t), a(u)) ?
   Call:  ( 10) a(s) ?
   Fail:  ( 10) a(s) ?
   Redo:  (  9) if_then_else(a(s), a(t), a(u)) ?
   Call:  ( 10) a(u) ?
   Exit:  ( 10) a(u) ?
   Exit:  (  9) if_then_else(a(s), a(t), a(u)) ?
   Exit:  (  8) a(p) ?
Yes
```

Otros usos del corte

- **Programa con meta–predicado ->**
  - **Programa**

```
a(p) :- a(q), a(r),
        (a(s) -> a(t)
        ;        a(u)).
a(q).
a(r).
a(u).
```

  - **Sesión**

```
?- trace.
Yes

?- a(p).
   Call:  (  8) a(p) ?
   Call:  (  9) a(q) ?
   Exit:  (  9) a(q) ?
   Call:  (  9) a(r) ?
   Exit:  (  9) a(r) ?
   Call:  (  9) a(s) ?
   Fail:  (  9) a(s) ?
   Call:  (  9) a(u) ?
   Exit:  (  9) a(u) ?
   Exit:  (  8) a(p) ?
Yes
```

Otros usos del corte

- **Programa con -> múltiple**
  - **Programa**

```
calificacion(X,Y) :-
        nota(X,N),
        ( N < 5 -> Y=suspenso
        ; N < 7 -> Y=aprobado
        ; N < 9 -> Y=notable
        ; true  -> Y=sobresaliente).

nota(juan,6).
```

  - **Sesión**

```
?- trace.
Yes

?- calificacion(juan,X).
   Call:  (  8) calificacion(juan, _G167) ?
   Call:  (  9) nota(juan, _L128) ?
   Exit:  (  9) nota(juan, 6) ?
   Call:  (  9) 6 < 5 ?
   Fail:  (  9) 6 < 5 ?
   Call:  (  9) 6 < 7 ?
   Exit:  (  9) 6 < 7 ?
   Call:  (  9) _G167 = aprobado ?
   Exit:  (  9) aprobado = aprobado ?
   Exit:  (  8) calificacion(juan, aprobado) ?
X = aprobado ;
No
```

Aritmética en Prolog

- **Evaluación de expresiones aritméticas básicas: `is`**

```
?- display(2+3*4).
+(2, *(3, 4))

?- X is 2+3*4.
X = 14

?- Y is (2+3)/4.
Y = 1.25

?- 5 is 2+3.
Yes

?- 5 is X+3.
[WARNING: Unbound variable in arithmetic expression]
```

- **Comparación de `is` con `=`**

```
?- X = 2+3.
X = 2 + 3

?- X is 2+3.
X = 5

?- 5 = 2+3.
No

?- 5 is 2+3.
Yes
```

Aritmética en Prolog

```
?- 2+X = 2+3.
X = 3

?- X = 2+Y.
X = 2 + _G89
Y = _G89

?- X = 2+X.
```

- **Inadecuación por eliminación del test de ocurrencia**
  - **Programa**

    ```
    falso :- X = X+1.
    ```

  - **Sesión**

    ```
    ?- falso.
    Yes
    ```

Listas en Prolog

```
?- display([a]).
.(a, [])
Yes

?- display([a,b]).
.(a, .(b, []))
Yes

?- .(X,Y) = [a].
X = a
Y = []
Yes

?- X = .(a,[]).
X = [a]
Yes

?- .(X,Y) = [a,b].
X = a
Y = [b]
Yes

?- .(X,Y) = [a,b,c].
X = a
Y = [b, c]
Yes
```

Listas en Prolog

```
?- [X|Y] = [a,b,c].
X = a
Y = [b, c]
Yes

?- [X,Y|Z] = [a,b,c].
X = a
Y = b
Z = [c]
Yes

?- [X,Y,Z] = [a,b,c].
X = a
Y = b
Z = c
Yes

?- [X,Y,Z] = [a,b].
No

?- [X,Y|Z] = [a,b].
X = a
Y = b
Z = []
Yes
```

Acumuladores

- **Procedimiento `longitud` recursivo**

```
longitud([],0).
longitud([X|R], N) :-
        longitud(R,M),
        N is M+1.
```

- **Procedimiento `longitud` recursivo con acumuladores**

```
longitud(L,N) :-
        longitud_ac(L,0,N).

longitud_ac([],N,N).
longitud_ac([X|R],N0,N) :-
        N1 is N0+1,
        longitud_ac(R,N1,N).
```

- **Comentarios:**
  - **Acumuladores**
  - **Ventaja de procedimientos recursivos finales**
  - **Procediminento primitivo `length`**

Acumuladores

- **Procedimiento `inversa`**

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

conc([],L,L).
conc([X|L1],L2,[X|L3]) :-
        conc(L1,L2,L3).
```

- **Procedimiento `inversa` con acumuladores**

```
inversa(L1,L2) :-
        inversa(L1,[],L2).

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

- **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] - []
```

Acumuladores

- **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
```

- **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]).
```

Acumuladores

- **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
```

Acumuladores

- **Procedimientos** `reverse` **y** `append`

```
?- help(reverse).
reverse(+List1, -List2)
     Reverse the order of the elements in List1 and
     unify the result with the elements of List2.
Yes

?- help(append).
append(?List1, ?List2, ?List3)
     Succeeds when List3 unifies with the concatenation
     of List1 and List2. The predicate can be used with
     any instantiation pattern (even three variables).
Yes

?- listing(append).
append([], A, A).
append([A|B], C, [A|D]) :-
        append(B, C, D).
Yes
```

Predicados de segundo orden

- **El predicado de segundo orden `call`**

```
?- call(reverse,[a,b,c],X).
X = [c, b, a]
Yes
```

- **Definición del predicado de segundo orden `map`**

```
% ?- map(reverse,[[a,b],[1,2,3]],X).
% X = [[b, a], [3, 2, 1]]
% Yes
map(R,[],[]).
map(R,[X|Xs],[Y|Ys]) :-
        call(R,X,Y),
        map(R,Xs,Ys).
```

- **El predicado `maplist`**

```
?- maplist(reverse,[[a,b],[1,2,3]],X).
X = [[b, a], [3, 2, 1]]
Yes
```

- **Transformación entre términos y listas con `=..`**

```
?- T =.. [suma,2,3,5].
T = suma(2, 3, 5)
Yes

?- suma(2,3,5) =.. L.
L = [suma, 2, 3, 5]
Yes
```

Predicados de segundo orden

- **Definición del predicado de segundo orden `map` con `=..`**

```
% ?- map(reverse,[[a,b],[1,2,3]],X).
% X = [[b, a], [3, 2, 1]]
% Yes
map(R,[],[]).
map(R,[X|Xs],[Y|Ys]) :-
        P =.. [R,X,Y],
        P,
        map(R,Xs,Ys).
```

- **Base de datos interna**

```
?- assert(clase(a,vocal)).
Yes

?- listing(clase).
clase(a, vocal).
Yes

?- assert(clase(b,consonante)).
Yes

?- listing(clase).
clase(a, vocal).
clase(b, consonante).
Yes

?- retract(clase(a,vocal)).
Yes
```

Predicados de segundo orden

```
?- listing(clase).
clase(b, consonante).
Yes

?- asserta(clase(a,vocal)).
Yes

?- listing(clase).
clase(a, vocal).
clase(b, consonante).
Yes

?- abolish(clase,2).
Yes

?- listing(clase).
Correct to 'close'? n
Correct to 'clause'? n
[WARNING: No predicates for 'clase']
No
```

Predicados de segundo orden

- **Predicados que recogen todas las soluciones**
  - **El predicado `findall`**

```
?- assert(clase(a,vocal)),
   assert(clase(b,consonante)),
   assert(clase(e,vocal)),
   assert(clase(c,consonante)).
Yes

?- findall(X,clase(X,vocal),L).
X = _G331
L = [a, e]
Yes

?- findall(_X,clase(_X,vocal),L).
L = [a, e]
Yes

?- findall(X,clase(X,C),L).
X = _G244
C = _G245
L = [a, b, e, c] ;
No
```

Predicados de segundo orden

- **El predicado** `bagof`

```
?- bagof(X,clase(X,vocal),L).
X = _G196
L = [a, e]
Yes

?- bagof(X,clase(X,C),L).
X = _G196
C = vocal
L = [a, e] ;
X = _G196
C = consonante
L = [b, c] ;
No

?- bagof(X,C^clase(X,C),L).
X = _G232
C = _G233
L = [a, b, e, c] ;
No
```

- **El predicado** `setof`

```
?- setof(X,C^clase(X,C),L).
X = _G232
C = _G233
L = [a, b, c, e]
Yes
```

Operadores

- **Operadores predefinidos**

Precedencia	Tipo	Operadores
1200	xfx	:-
1200	fx	:-
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	not
700	xfx	<, =<, >, >=, is, =, =..
500	yfx	+, -
500	fx	-
400	yfx	*, /, //, mod, rem
200	xfy	^

- **Tipo de operadores**
  - `fx`, `fy`: **Prefijo**
  - `xfx`: **Infijo no asociativo**
  - `yfx`: **Infijo asocia por la izquierda**
  - `xfy`: **Infijo asocia por la derecha**
  - `xf`, `yf`: **Postfijo**

Operadores

- **Análisis de expresiones con operadores**

```
?- display(2+3+4).
+(+(2, 3), 4)
Yes

?- display(2+3*4).
+(2, *(3, 4))
Yes

?- display((2+3)*4).
*(+(2, 3), 4)
Yes

?- display(2^3^4).
^(2, ^(3, 4))
Yes
```

- **Declaración de operadores**

```
?- op(900,fx,si).
Yes

?- op(800,xfx,entonces).
Yes

?- op(700,yfx,y).
Yes

?- display(si a y b y c entonces c).
si(entonces(y(a, y(b, c)), c))
Yes
```

Metaprogramas

• Metaprograma 1 (con operadores)

• Especificación

```
* Base de reglas:
  * R1: Si el animal tiene pelos es un mamífero.
  * R2: Si el animal da leche es un mamífero.
  * R3: Si el animal es un mamífero y tiene pezuñas
        es un ungulado.
  * R4: Si el animal es un mamífero y rumia
        es un ungulado.
  * R5: Si el animal es un ungulado y tiene cuello largo
        es una jirafa.
  * R6: Si el animal es un ungulado y tiene rayas negras
        es una cebra.

* Conclusión:
  * Si el animal tiene pelos, pezuñas y rayas negras
    entonces es una cebra.
```

• Programa objeto

```
:- op(900,fx,si).
:- op(800,xfx,entonces).
:- op(700,xfy,y).

si tiene_pelos entonces es_mamifero.
si da_leche entonces es_mamifero.
si es_mamifero y tiene_pezuñas entonces es_ungulado.
si es_mamifero y rumia entonces es_ungulado.
si es_ungulado y tiene_cuello_largo entonces es_jirafa.
si es_ungulado y tiene_rayas_negras entonces es_cebra.
```

Metaprogramas

- **Sesión**

```
?- deriva(si tiene_pelos
             y tiene_pezuñas
             y tiene_rayas_negras
          entonces es_cebra).
Yes

?- deriva(si tiene_pelos
             y tiene_pezuñas
             y tiene_rayas_negras
          entonces es_jirafa).
No
```

- **Metaprograma**

```
deriva(si Condiciones entonces Conclusion) :-
        si Cuerpo entonces Conclusion,
        deriva(si Condiciones entonces Cuerpo).
deriva(si Condiciones entonces Concl1 y Concl2) :-
        deriva(si Condiciones entonces Concl1),
        deriva(si Condiciones entonces Concl2).
deriva(si Condiciones entonces Conclusion) :-
        asumida(Conclusion,Condiciones).

asumida(A,A).
asumida(A,A y As).
asumida(A,B y As) :-
        asumida(A,As).
```

Metaprogramas

- **Metaprograma 2 (con `clause`)**
  - **Programa objeto**

```
es_mamifero :- tiene_pelos.
es_mamifero :- da_leche.
es_ungulado :- es_mamifero, tiene_pezuñas.
es_ungulado :- es_mamifero, rumia.
es_jirafa   :- es_ungulado, tiene_cuello_largo.
es_cebra    :- es_ungulado, tiene_rayas_negras.

tiene_pelos.
tiene_pezuñas.
tiene_rayas_negras.
```

  - **El predicado `clause`**

```
?- clause(es_ungulado,C).
C = es_mamifero, tiene_pezuñas ;
C = es_mamifero, rumia ;
No
```

  - **Metaprograma 2**

```
prueba(true).
prueba((A,B)) :-
        prueba(A),
        prueba(B).
prueba(A) :-
        clause(A,B),
        prueba(B).
```

Metaprogramas

- **Sesión**

```
?- prueba(es_cebra).
Yes

?- prueba(es_jirafa).
Action (h for help) ? a
abort
Execution Aborted
```

- **Metaprograma 3 (con `clause` y corte)**

```
prueba(true) :- !.
prueba((A,B)) :- !,
        prueba(A),
        prueba(B).
prueba(A) :-
        % no A=true ni A=(X,Y)
        clause(A,B),
        prueba(B).
```

- **Sesión**

```
?- prueba(es_cebra).
Yes

?- prueba(es_jirafa).
No
```

Metaprogramas

- **Metaprograma 4 (con variables)**
  - **Programa objeto**

```
c(X) :- a(X), b(X).
a(X) :- f(X).
b(X) :- f(X).
f(p).
```

  - **Metaprograma 4**

```
prueba(true):-!.
prueba((A,B)):-!,
        prueba(A),
        prueba(B).
prueba(A):-
        clause(Cabeza,Cuerpo),
        A=Cabeza,
        prueba(Cuerpo).
```

  - **Sesión**

```
?- prueba(c(X)).
X = p ;
No
```

- **Variaciones de meta–intérpretes:**
  - **Estrategias de búsqueda**
  - **Procedimientos de prueba**
  - **Lenguaje de cláusulas**
  - **Información adicional**

Metaprogramas

- **Meta–intérprete con negación**

```
prueba(true):-!.
prueba((A,B)):-!,
        prueba(A),
        prueba(B).
prueba(not A) :- !,
        not prueba(A).
prueba(A):-
        clause(A,Cuerpo),
        prueba(Cuerpo).
```

- **Meta–intérprete con escritura de la demostración**
  - **Programa objeto**

```
c(X) :- a(X), b(X).
a(X) :- f(X).
b(X) :- f(X).
f(p).
```

Metaprogramas

- **Sesión**

```
?- prueba_y_escribe_demostracion(c(X)).
:- c(p)
     +-------- c(p) :- a(p), b(p)
     |
:- a(p), b(p)
     +-------- a(p) :- f(p)
     |
:- f(p), b(p)
     +-------- f(p) :- true
     |
:- b(p)
     +-------- b(p) :- f(p)
     |
:- f(p)
     +-------- f(p) :- true
     |
               []

X = p ;

No
```

Metaprogramas

- **Meta–intérprete**

```
prueba_y_escribe_demostracion(A) :-
        prueba(A,P),
        escribe_demostracion(P).

prueba(true,[]) :- !.
prueba((A,B),[p((A,B),(A:-C))|Demostracion]) :- !,
        clause(A,C),
        concatena_conjunciones(C,B,D),
        prueba(D,Demostracion).
prueba(A,[p(A,(A:-B))|Demostracion]) :-
        clause(A,B),
        prueba(B,Demostracion).

concatena_conjunciones(true,Ys,Ys).
concatena_conjunciones(X,Ys,(X,Ys)) :-
        not X=true,
        not X=(_A,_B).
concatena_conjunciones((X,Xs),Ys,(X,Zs)):-
        concatena_conjunciones(Xs,Ys,Zs).

escribe_demostracion([]) :-
        tab(15),write('[]'),nl.
escribe_demostracion([p(A,B)|Demostracion]) :-
        write((:-A)),nl,
        tab(5),write('+-------- '),write(B),nl,
        tab(5),write('|'),nl,
        escribe_demostracion(Demostracion).
```

Metodogía de programación lógica

- **Enunciado del problema:**

```
?- ord_quicksort([3,2,1,5],L).
L = [1, 2, 3, 5]
```

- **Procedimiento quicksort: Para ordenar la lista no vacía `L`:**
  - **Elegir un elemento `X` de `L` y dividir el resto en las dos listas `Menores` (que contiene los elementos de `L` menores que `X`) y `Mayores` (que contiene los elementos de `L` mayores o iguales que `X`)**
  - **Ordenar los `Menores` obteniendo `MenoresOrdenados`**
  - **Ordenar los `Mayores` obteniendo `MayoresOrdenados`**
  - **La lista `L` ordenada es la concatenación de `MenoresOrdenados` y `[X|MayoresOrdenados]`**

Metodogía de programación lógica

- **Programa**

```
ord_quicksort([],[]).
ord_quicksort([X|R],Ordenada) :-
        divide(X,R,Menores,Mayores),
        ord_quicksort(Menores,MenoresOrdenados),
        ord_quicksort(Mayores,MayoresOrdenados),
        append(MenoresOrdenados,
               [X|MayoresOrdenados],
               Ordenada).

divide(_,[],[],[]).
divide(X,[Y|R],[Y|Menores],Mayores) :-
        menor(Y,X), !,
        divide(X,R,Menores,Mayores).
divide(X,[Y|R],Menores,[Y|Mayores]) :-
        divide(X,R,Menores,Mayores).
```

Bibliografía

- Bratko, I. "Prolog Programming for Artificial Intelligence (2nd ed.)" (Addison–Wesley, 1990)
- Clocksin, W.F. y Mellish, C.S. "Programación en Prolog" (Gustavo Gili, 1987)
- 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".
- Kowalski, R. "Lógica, programación e inteligencia artificial" (Díaz de Santos, 1979)
- Sterling,, L. y Shapiro, E. "The Art of Prolog (2nd ed.)" (The MIT Press, 1994)