Tema 2: Prolog

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
http://www.cs.us.es/~jalonso

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Listas

- **Definición de listas**
  - **La lista vacía `[]` es una lista.**
  - **Si `L` es una lista, entonces `.(a,L)` es una lista.**

- **Ejemplos**

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

Listas

- **Escritura abreviada**

```
[X|Y] = .(X,Y)
```

- **Ejemplos con escritura abreviada**

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

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

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

Listas

- **Concatenación de listas** (`append`)
  - ***Especificación:*** `conc(A,B,C)` **se verifica si** `C` **es la lista obtenida escribiendo los elementos de la lista** `B` **a continuación de los elementos de la lista** `A`**. Por ejemplo,**

```
?- conc([a,b],[b,d],C).
C =[a,b,b,d]
```

  - ***Descripción:***

    1. **Si** `A` **es la lista vacía, entonces la concatenación de** `A` **y** `B` **es** `B`**.**
    2. **Si** `A` **es una lista cuyo primer elemento es** `X` **y cuyo resto es** `D`**, entonces la concatenación de** `A` **y** `B` **es una lista cuyo primer elemento es** `X` **y cuyo resto es la concatenación de** `D` **y** `B`**.**

  - ***Definición 1:***

```
conc(A,B,C) :- A=[], C=B.
conc(A,B,C) :- A=[X|D], conc(D,B,E), C=[X|E].
```

  - ***Definición 2:***

```
conc([],B,B).
conc([X|D],B,[X|E]) :- conc(D,B,E).
```

Listas

- ***Nota:*** **Analogía entre la definición de** `conc` **y la de** `suma`**,**
- ***Consulta 1:*** **¿Cuál es el resultado de concatenar las listas** `[a,b]` **y** `[c,d,e]`**?.**

```
?- conc([a,b],[c,d,e],L).
L = [a, b, c, d, e]
```

- ***Consulta 2:*** **¿Qué lista hay que añadirle al lista** `[a,b]` **para obtener** `[a,b,c,d]`**?.**

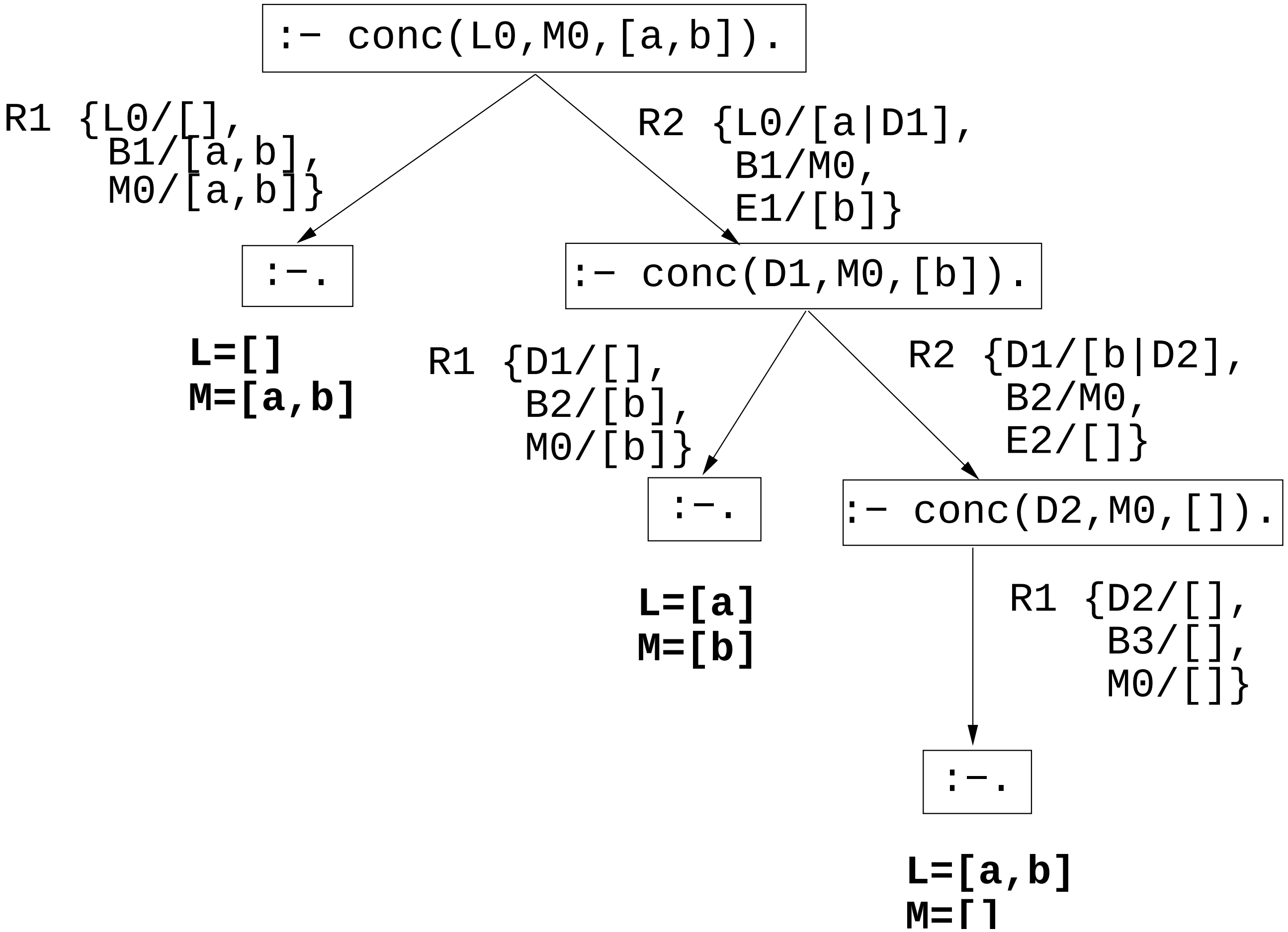
```
?- conc([a,b],L,[a,b,c,d]).
L = [c, d]
```

- ***Consulta 3:*** **¿Qué dos listas hay que concatenar para obtener** `[a,b]`**?.**

```
?- conc(L,M,[a,b]).
L = []
M = [a, b] ;
L = [a]
M = [b] ;
L = [a, b]
M = [] ;
No
```

Listas

- **Árbol de deducción correspondiente a** `?- conc(L,M,[a,b]).`



Listas

- **La relación de pertenencia (`member`)**
  - ***Especificación:*** `pertenece(X,L)` **se verifica si** `X` **es un elemento de la lista** `L`**.**
  - ***Definición 1:***

```
pertenece(X,[X|L]).
pertenece(X,[Y|L]) :- pertenece(X,L).
```

  - ***Definición 2:***

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

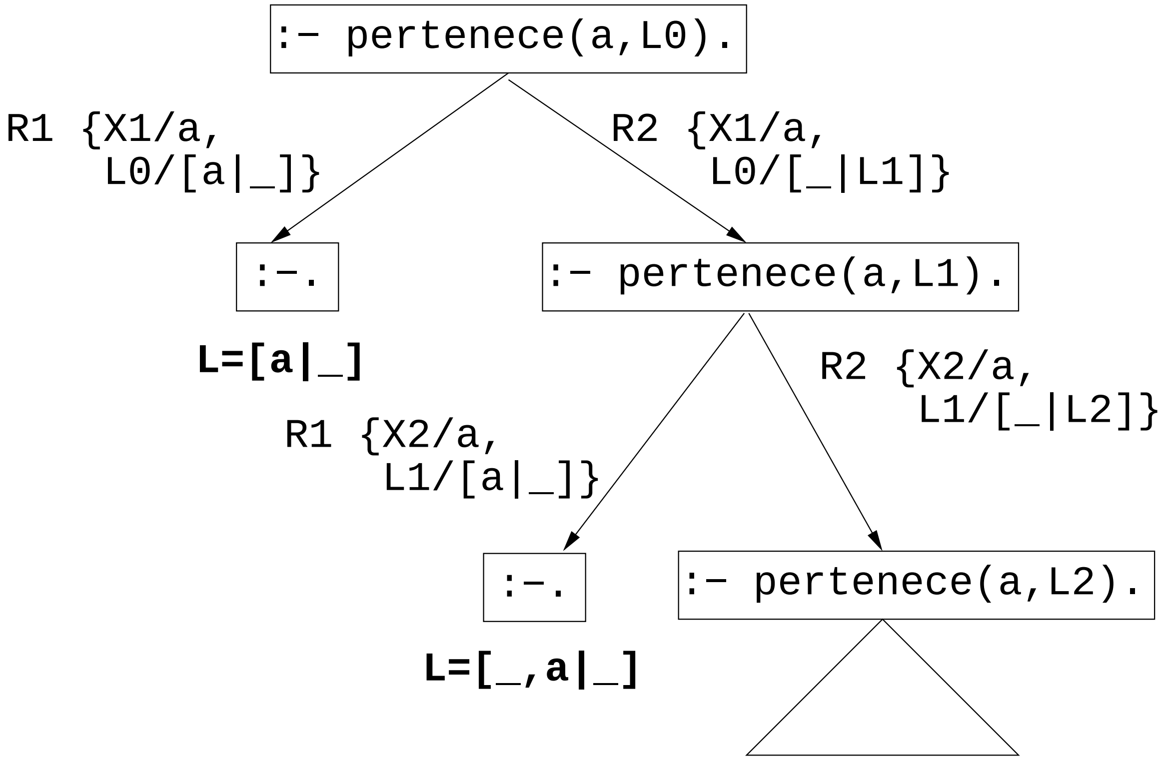
Listas

- ***Consultas:***

```
?- pertenece(b,[a,b,c]).
Yes

?- pertenece(d,[a,b,c]).
No

?- pertenece(X,[a,b,a]).
X = a ;
X = b ;
X = a ;
No

?- pertenece(a,L).
L = [a|_G233] ;
L = [_G232, a|_G236] ;
L = [_G232, _G235, a|_G239]
Yes
```

Listas

- **Árbol de deducción de** `?- pertenece(a,L).`



Disyunciones

- **Definición de `pertenece` con disyunción**

```
pertenece(X,[Y|L]) :- X=Y ; pertenece(X,L).
```

- **Definición equivalente sin disyunción**

```
pertenece(X,[Y|L]) :- X=Y.
pertenece(X,[Y|L]) :- pertenece(X,L).
```

Operadores

- **Ejemplos de notación infija y prefija en expresiones aritméticas**

```
?- +(X,Y) = a+b.
X = a
Y = b
?- +(X,Y) = a+b+c.
X = a+b
Y = c
?- +(X,Y) = a+(b+c).
X = a
Y = b+c
?- a+b+c = (a+b)+c.
Yes
?- a+b+c = a+(b+c).
No
```

Operadores

- **Operadores aritméticos predeclarados:**

Precedencia	Tipo	Operadores	
500	`yfx`	`+,-`	Infijo asocia por la izquierda
500	`fx`	`-`	Prefijo no asocia
400	`yfx`	`*, /`	Infijo asocia por la izquierda
200	`xfy`	`^`	Infijo asocia por la derecha

- **Ejemplos de asociatividad**

```
?- X^Y = a^b^c.
X = a
Y = b^c
?- a^b^c = (a^b)^c.
No
?- a^b^c = a^(b^c).
Yes
```

Operadores

- **Ejemplo de precedencia**

```
?- X+Y = a+b*c.
X = a
Y = b*c
?- X*Y = a+b*c.
No
?- X*Y = (a+b)*c.
X = a+b
Y = c
?- a+b*c = a+(b*c).
Yes
?- a+b*c = (a+b)*c.
No
```

Operadores

- **Definición de operadores**

  - **Definición** (`ejemplo_operadores.pl`)

```
:-op(800,xfx,estudian).
:-op(400,xfx,y).

juan y ana estudian lógica.
```

  - **Consultas**

```
?- [ejemplo_operadores].
Yes

?- Quienes estudian lógica.
Quienes = juan y ana

?- juan y Otro estudian Algo.
Otro = ana
Algo = lógica
```

Aritmética

- **Evaluación de expresiones aritmética con `is`**

```
?- X is 2+3^3.
X = 29
?- X is 2+3, Y is 2*X.
X = 5
Y = 10
```

- **Relaciones aritméticas:**
  - `<`, `=<`, `>`, `>=`, `=:=`

Aritmética

- **Definición de procedimientos aritméticos**
  - `factorial(X,Y)` **se verifica si** `Y` **es el factorial de** `X`**. Por ejemplo,**

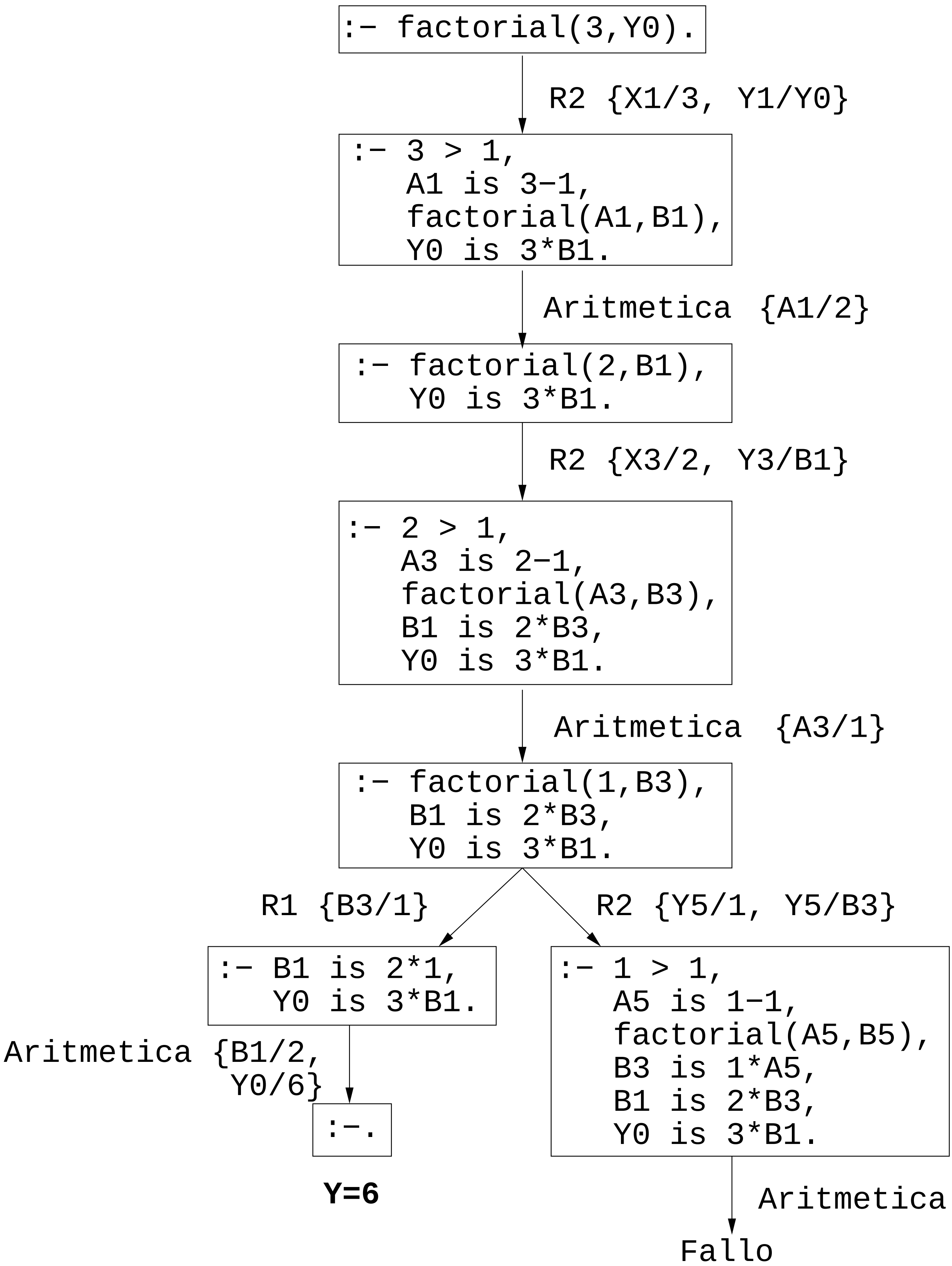
    ```
    ?- factorial(3,Y).
    Y = 6 ;
    No
    ```

  - **Definición**

    ```
    factorial(1,1).
    factorial(X,Y) :-
       X > 1,
       A is X - 1,
       factorial(A,B),
       Y is X * B.
    ```

Aritmética

- **Árbol de deducción de** `?- factorial(3,Y).`



Control mediante corte

- **Ejemplo de `nota` sin corte**
  - `nota(X,Y)` se verifica si `Y` es la calificación correspondiente a la nota `X`; es decir, `Y` es `suspenso` si `X` es menor que `5`, `Y` es `aprobado` si `X` es mayor o igual que `5` pero menor que `7`, `Y` es `notable` si `X` es mayor que `7` pero menor que `9` e `Y` es sobresaliente si `X` es mayor que `9`.
  - Definición:

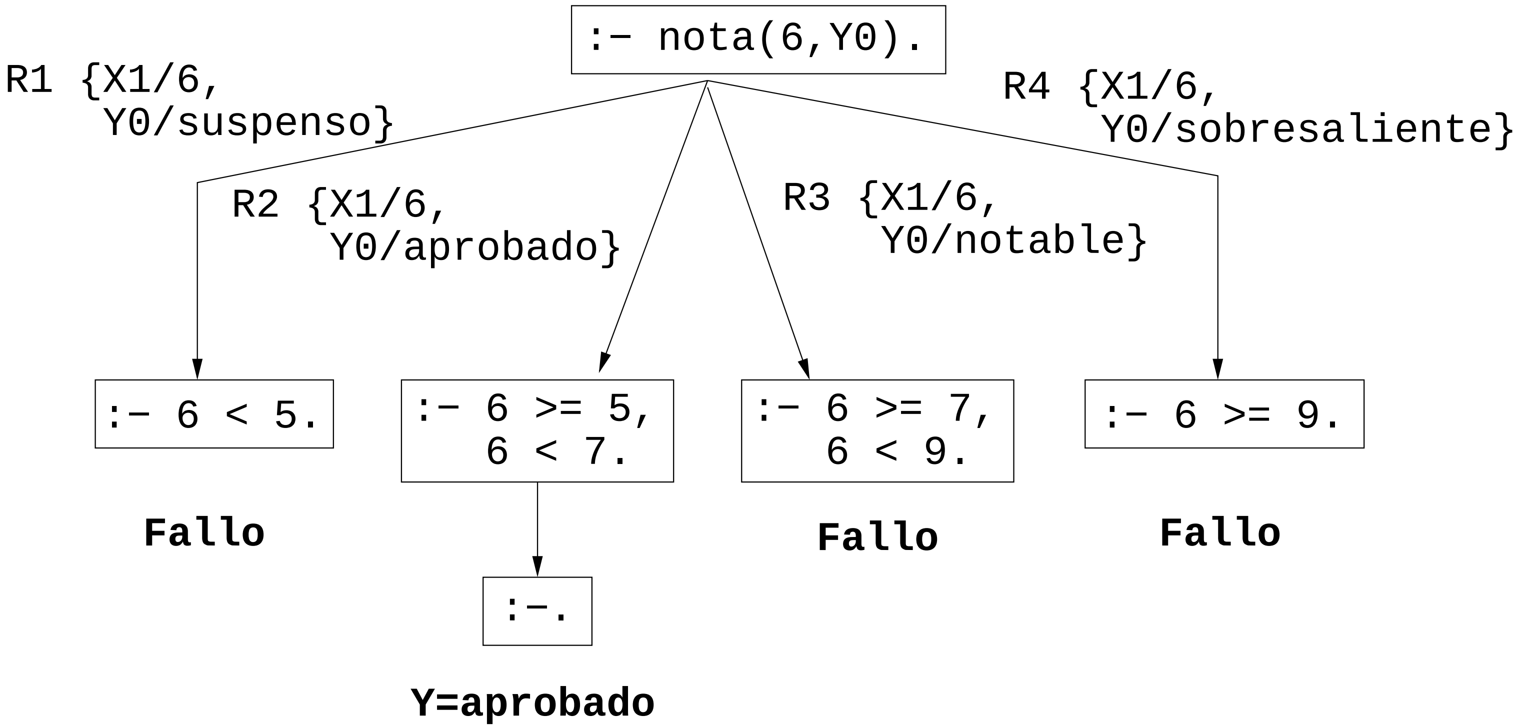
```
nota(X,suspenso)      :- X < 5.
nota(X,aprobado)      :- X >= 5, X < 7.
nota(X,notable)       :- X >= 7, X < 9.
nota(X,sobresaliente) :- X >= 9.
```

  - Ejemplo: ¿cuál es la calificación correspondiente a un `6`?:

```
?- nota(6,Y).
Y = aprobado;
No
```

Control mediante corte

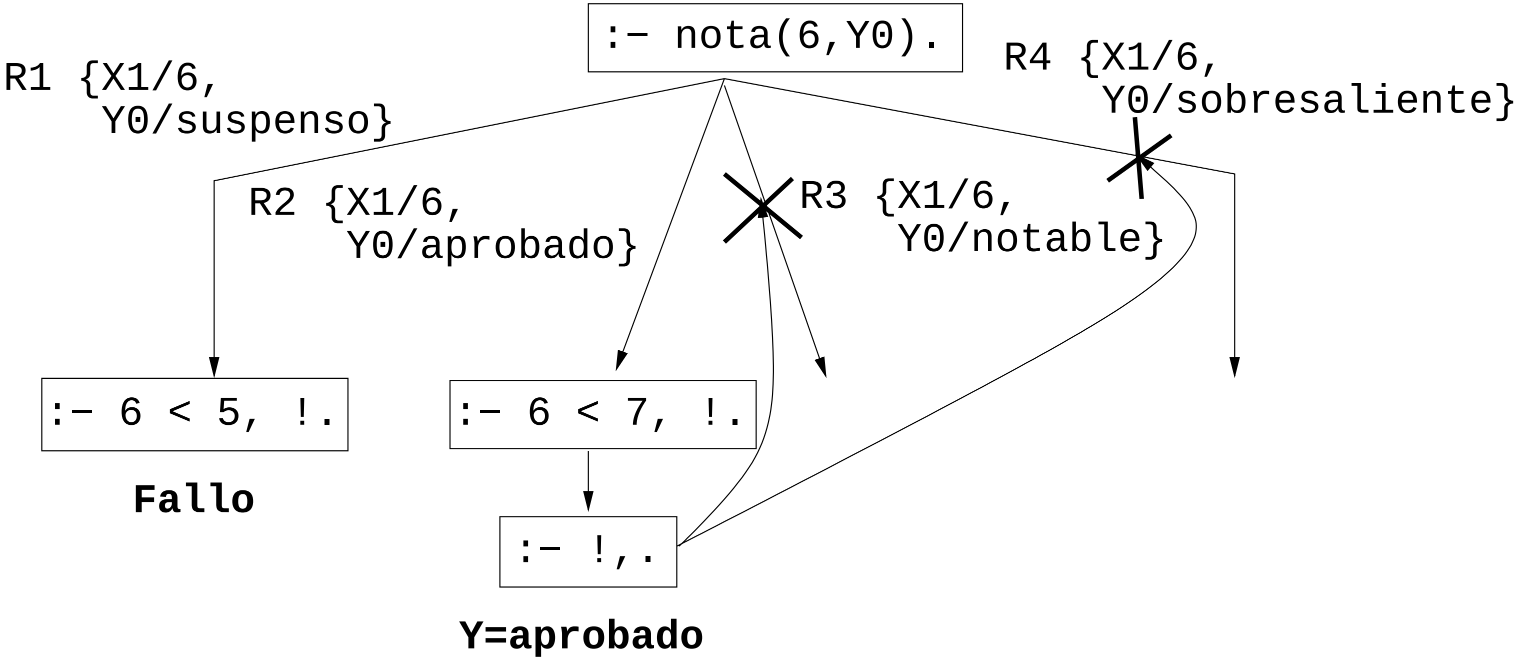
- **Árbol de deducción de** `?- nota(6,Y).`



Control mediante corte

- **Ejemplo de `nota` con cortes**

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




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

Control mediante corte

- **Uso de corte para respuesta única**
  - **Diferencia entre `member` y `memberchk`**

```
?- member(X,[a,b,a,c]), X=a.
X = a ;
X = a ;
No

?- memberchk(X,[a,b,a,c]), X=a.
X = a ;
No
```

  - **Definición de `member` y `memberchk`**

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

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

Negación como fallo

- **Negación como fallo**
  - **Definición de la negación como fallo (`not`)**

```
no(P) :- P, !, fail.          % No 1
no(P).                        % No 2
```

  - **Programa con negación**

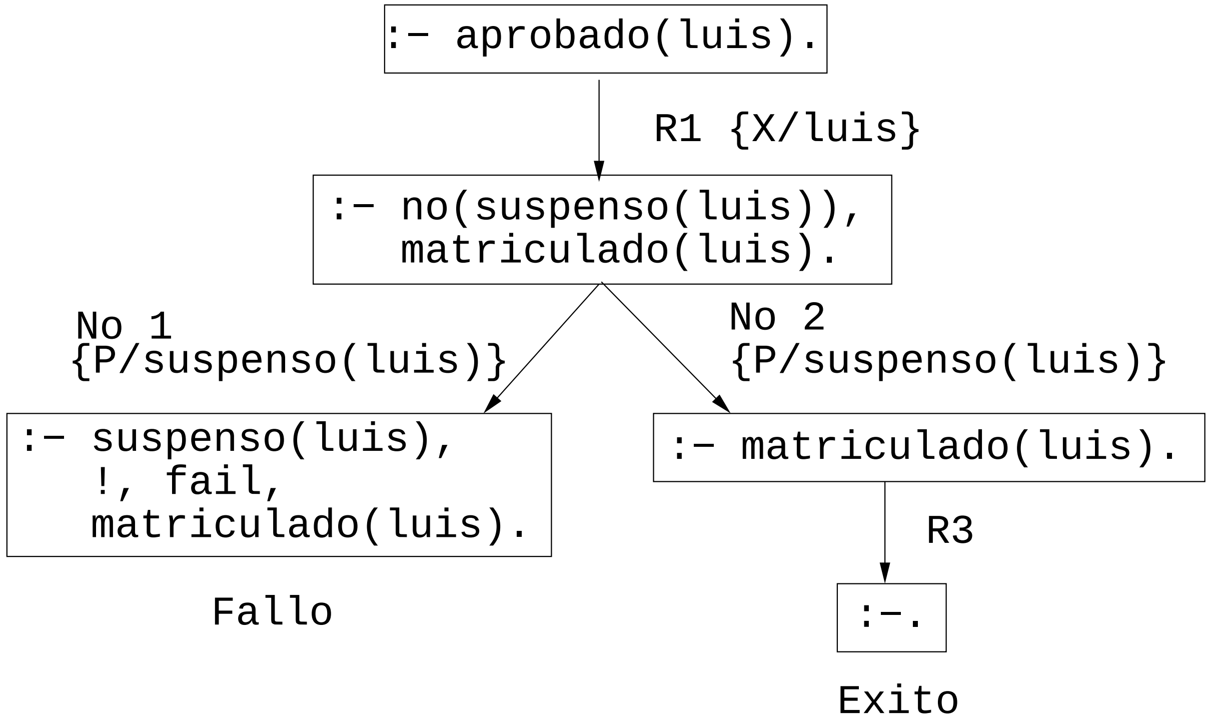
```
aprobado(X) :- no(suspenso(X)), matriculado(X).    % R1
matriculado(juan).                                 % R2
matriculado(luis).                                 % R3
suspenso(juan).                                    % R4
```

  - **Consultas**

```
?- aprobado(luis).
Yes

?- aprobado(X).
No
```

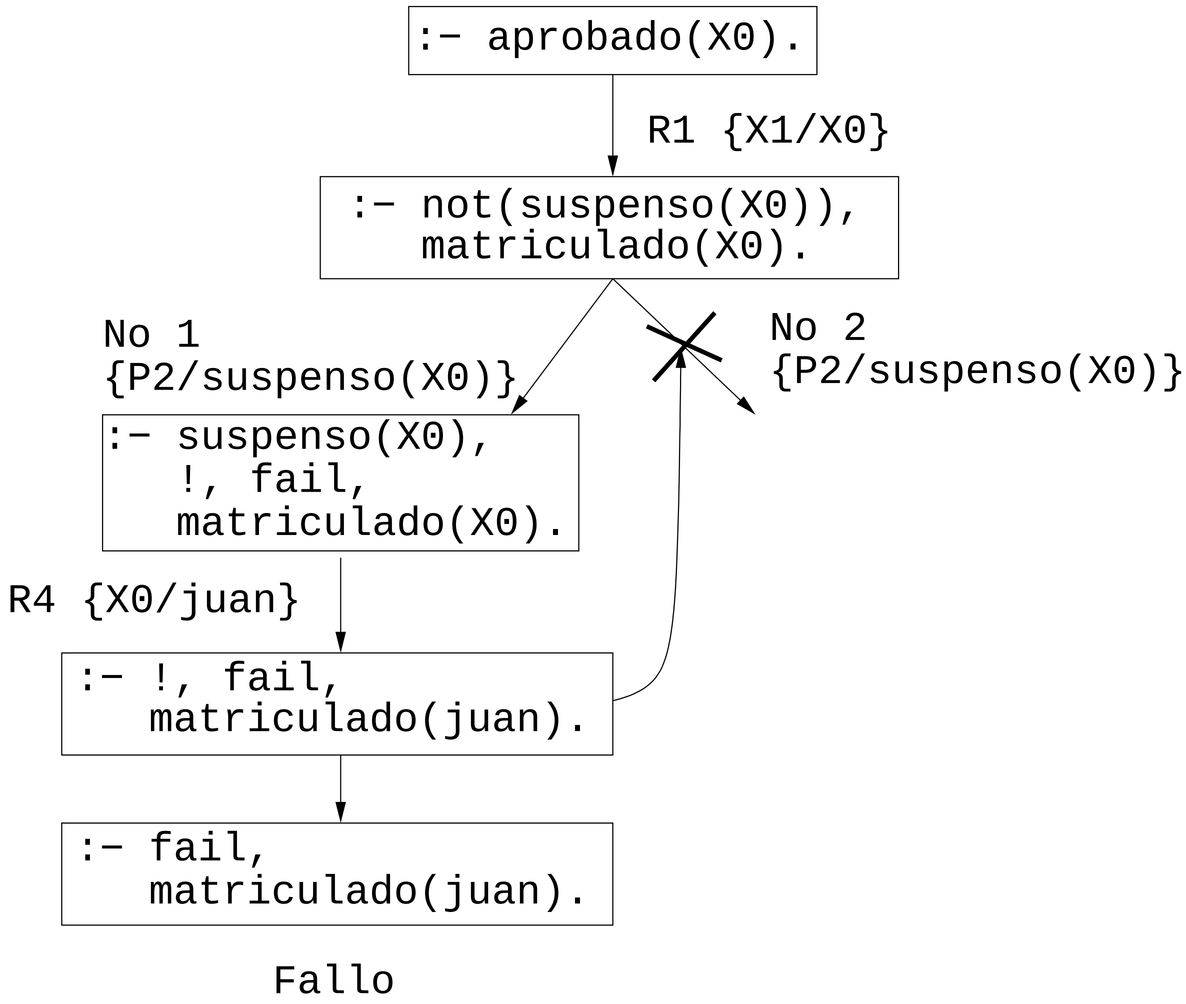
Negación como fallo

- **Árbol de deducción de** `?- aprobado(luis).`



Negación como fallo

- **Árbol de deducción de** `?- aprobado(X).`



Negación como fallo

- **Modificación del orden de los literales**
  - **Programa**

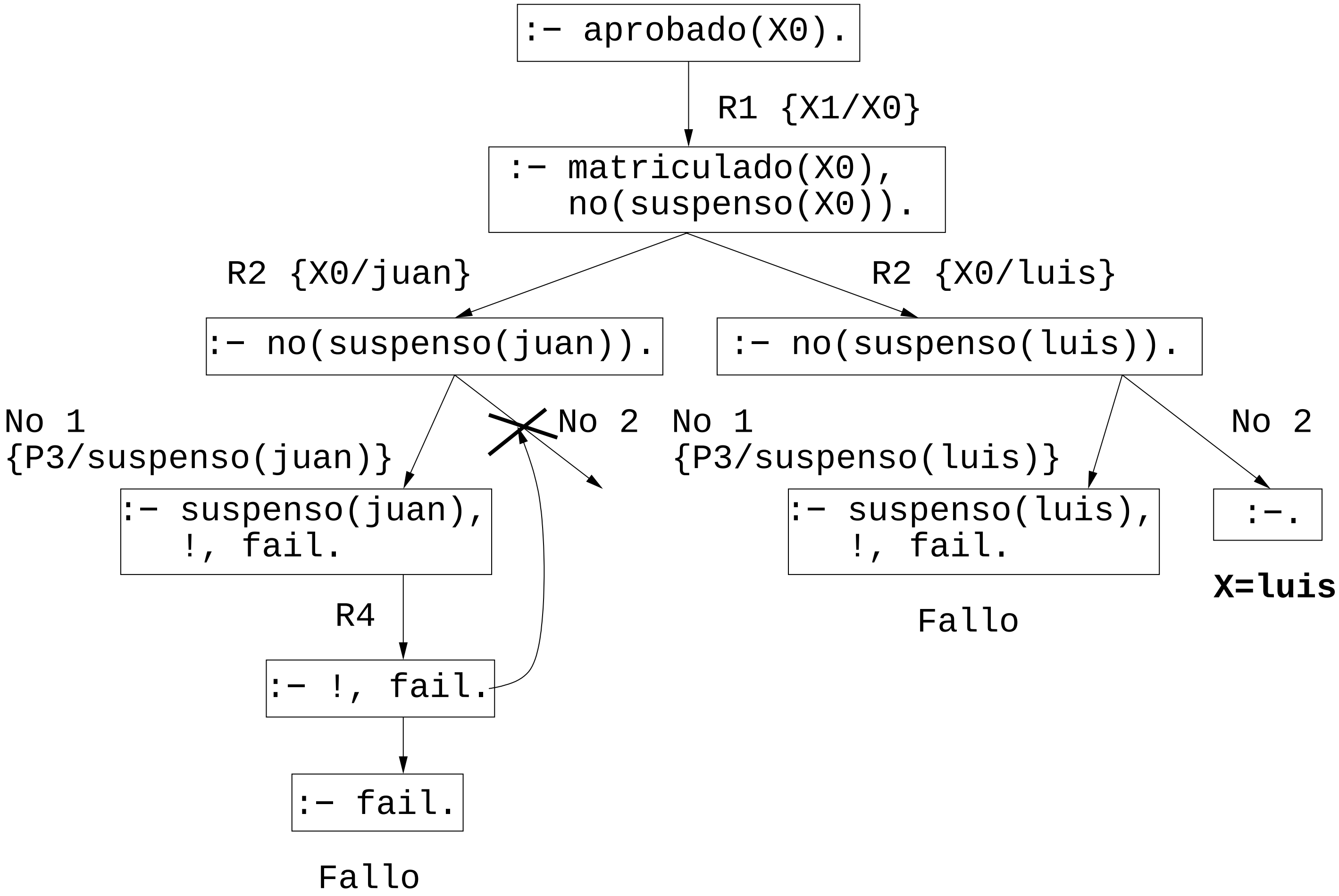
```
aprobado(X) :- matriculado(X), no(suspenso(X)).    % R1
matriculado(juan).                                  % R2
matriculado(luis).                                  % R3
suspenso(juan).                                     % R4
```

  - **Consulta**

```
?- aprobado(X).
X = luis
Yes
```

Negación como fallo

- **Árbol de deducción de** `?- aprobado(X).`



Negación como fallo

- **Ejemplo de definición con `not` y con corte**
  - `borra(L1,X,L2)` se verifica si `L2` es la lista obtenida eliminando los elementos de `L1` unificables simultáneamente con `X`; por ejemplo,

```
?- borra([a,b,a,c],a,L).
L = [b, c] ;
No
?- borra([a,Y,a,c],a,L).
Y = a
L = [c] ;
No
?- borra([a,Y,a,c],X,L).
Y = a
X = a
L = [c] ;
No
```

Negación como fallo

- **Definición con not**

```
borra_1([],_,[]).
borra_1([X|L1],Y,L2) :-
   X=Y,
   borra_1(L1,Y,L2).
borra_1([X|L1],Y,[X|L2]) :-
   not(X=Y),
   borra_1(L1,Y,L2).
```

- **Definición con corte**

```
borra_2([],_,[]).
borra_2([X|L1],Y,L2) :-
   X=Y, !,
   borra_2(L1,Y,L2).
borra_2([X|L1],Y,[X|L2]) :-
   % not(X=Y),
   borra_2(L1,Y,L2).
```

Negación como fallo

- **Definición con corte y simplificada**

```
borra_3([],_,[]).
borra_3([X|L1],X,L2) :-
   !,
   borra_3(L1,Y,L2).
borra_3([X|L1],Y,[X|L2]) :-
   % not(X=Y),
   borra_3(L1,Y,L2).
```

El condicional

- **Definición de `nota` con el condicional**

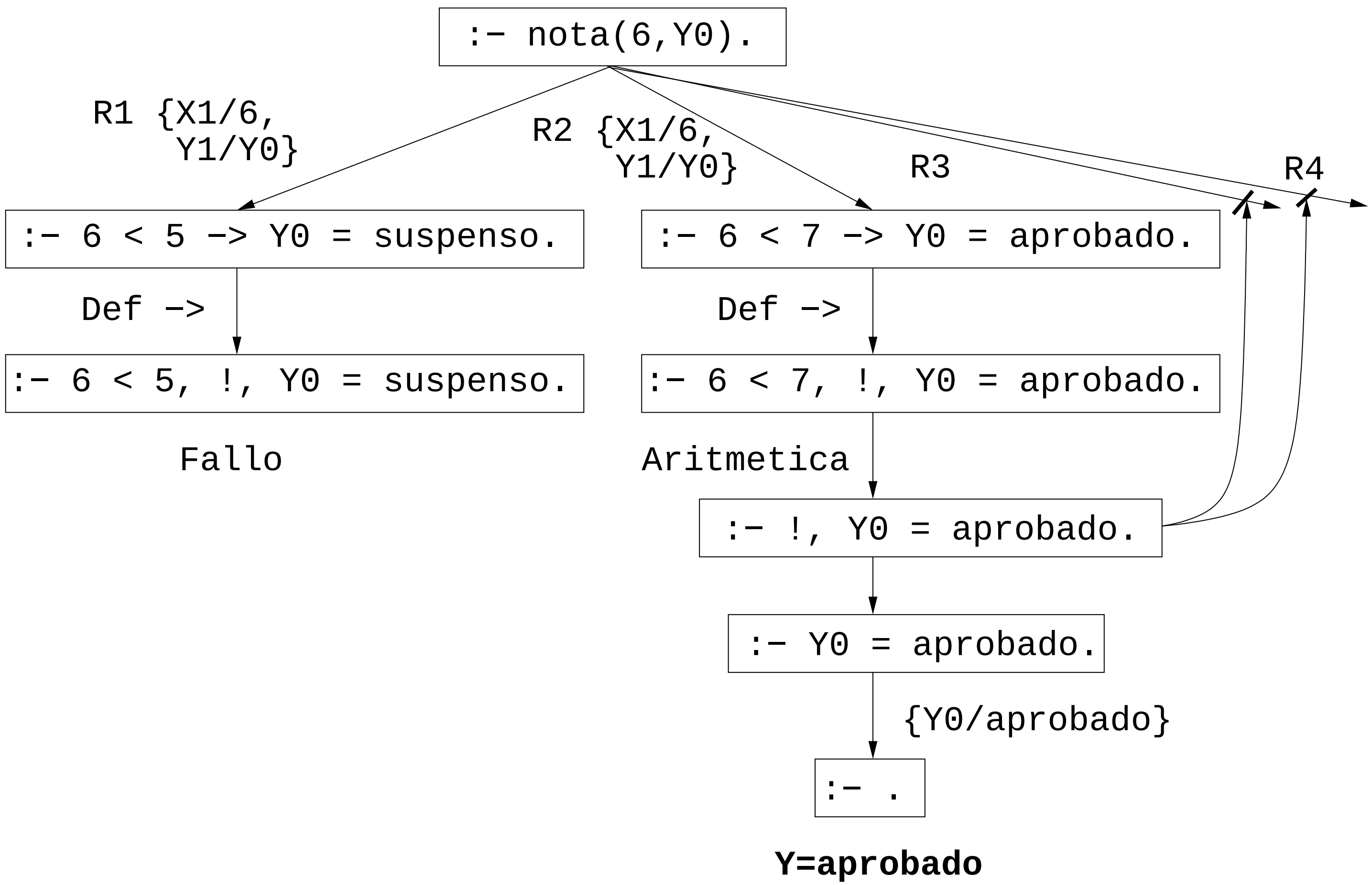
```
nota(X,Y) :-
   X < 5 -> Y = suspenso ;              % R1
   X < 7 -> Y = aprobado ;              % R2
   X < 9 -> Y = notable ;               % R3
   true  -> Y = sobresaliente.          % R4
```

- **Definición del condicional y verdad**

```
P -> Q :- P, !, Q.                      % Def. ->
true.
```

El condicional

- **Árbol de deducción correspondiente a la pregunta** `?- nota(6,Y).`



Predicados sobre tipos de término

- **Predicados sobre tipos de término**
  - `var(T)` **se verifica si** `T` **es una variable.**

```
?- var(X1).              =>  Yes
?- var(_).               =>  Yes
?- var(_X1).             =>  Yes
?- X1=a, var(X1).        =>  No
```

  - `atom(T)` **se verifica si** `T` **es un átomo.**

```
?- atom(átomo).                  =>  Yes
?- atom('átomo').                =>  Yes
?- atom([]).                     =>  Yes
?- atom('esto es un átomo').     =>  Yes
?- atom(3).                      =>  No
?- atom(1+2).                    =>  No
```

Predicados sobre tipos de término

- `number(T)` se verifica si `T` es un número.

```
?- number(123).            =>  Yes
?- number(-25.14).         =>  Yes
?- number(1+3).            =>  No
?- X is 1+3, number(X).    =>  X=4  Yes
```

- `compound(T)` se verifica si `T` es un término compuesto.

```
?- compound(1+2).          =>  Yes
?- compound(f(X,a)).       =>  Yes
?- compound([1,2]).        =>  Yes
?- compound([]).           =>  No
```

- `atomic(T)` se verifica si `T` es un término atómico (i.e. variable, átomo, cadena o número).

```
?- atomic(átomo).          =>  Yes
?- atomic(123).            =>  Yes
?- atomic(X).              =>  No
?- atomic(f(1,2)).         =>  No
```

Comparación y ordenación de términos

- **Comparación de términos**
  - `T1 = T2` **se verifica si** `T1` **y** `T2` **son unificables.**
  - `T1 == T2` **se verifica si** `T1` **y** `T2` **son idénticos.**

```
?- f(X) = f(Y).
X = _G164
Y = _G164
Yes
?- f(X) == f(Y).
No
?- f(X) == f(X).
X = _G170
Yes
```

Comparación y ordenación de términos

- **Ordenación de términos**
  - `T1 @< T2` **se verifica si el término** `T1` **es anterior que** `T2` **en el orden de términos de Prolog**

```
?- X @< 3.               =>  Yes
?- ab @< ac.             =>  Yes
?- 21 @< 123.            =>  Yes
?- 12 @< a.              =>  Yes
?- g @< f(b).            =>  Yes
?- f(b) @< f(a,b).       =>  Yes
?- [a,1] @< [a,3].       =>  Yes
?- [a] @< [a,3].         =>  Yes
```

- **Ordenación con** `sort`
  - `sort(+L1,-L2)` **se verifica si** `L2` **es la lista obtenida ordenando de manera creciente los distintos elementos de** `L1` **y eliminando las repeticiones.**

```
?- sort([c4,2,a5,2,c3,a5,2,a5],L).
L = [2, a5, c3, c4]
```

Comparación y ordenación de términos

- **Procedimiento de ordenación**
  - `ordenación(+L1,-L2)` **se verifica si** `L2` **es la lista obtenida ordenando de manera creciente los distintos elementos de** `L1`**.**

```
?- ordenación([c4,2,a5,2,c3,a5,2,a5],L).
L = [2, 2, 2, a5, a5, a5, c3, c4]
```

  - **Definición**

```
ordenación([],[]).
ordenación([X|R],Ordenada) :-
   divide(X,R,Menores,Mayores),
   ordenación(Menores,Menores_ord),
   ordenación(Mayores,Mayores_ord),
   append(Menores_ord,[X|Mayores_ord],Ordenada).

divide(_,[],[],[]).
divide(X,[Y|R],Menores,[Y|Mayores]) :-
   X @< Y, !,
   divide(X,R,Menores,Mayores).
divide(X,[Y|R],[Y|Menores],Mayores) :-
   % not(X @< Y),
   divide(X,R,Menores,Mayores).
```

Procesamiento de términos

- **Transformación entre términos y listas**
  - `?T =.. ?L` se verifica si `L` es una lista cuyo primer elemento es el functor del término `T` y los restantes elementos de `L` son los argumentos de `T`. Por ejemplo,

```
?- padre(juan,luis) =.. L.
L = [padre, juan, luis]
?- T =.. [padre, juan, luis].
T = padre(juan,luis)
```

  - `alarga(+F1,+N,-F2)` se verifica si `F1` y `F2` son figuras geométricas del mismo tipo y el tamaño de la `F1` es el de la `F2` multiplicado por `N`, donde las figuras geométricas se representan como términos en los que el functor indica el tipo de figura y los argumentos su tamaño; por ejemplo,

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

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

Procesamiento de términos

- **Definición**

```
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).
```

• Los procedimientos `functor` y `arg`

- `functor(T,F,A)` **se verifica si** `F` **es el functor del término** `T` **y** `A` **es su aridad.**
- `arg(N,T,A)` **se verifica si** `A` **es el argumento del término** `T` **que ocupa el lugar** `N`**.**

```
?- functor(g(b,c,d),F,A).                 => F = g     A = 3
?- functor(T,g,2).                        => T = g(_G237,_G238)
?- arg(2,g(b,c,d),X).                     => X = c
?- functor(T,g,3),arg(1,T,b),arg(2,T,c). => T = g(b, c, _G405)
```

Procedimientos aplicativos

- **El procedimiento `apply`**
  - `apply(T,L)` **se verifica si es demostrable `T` después de aumentar el número de sus argumentos con los elementos de `L`; por ejemplo,**

```
?- plus(2,3,X).                           =>  X = 5
?- apply(plus,[2,3,X]).                   =>  X = 5
?- apply(plus(2),[3,X]).                  =>  X = 5
?- apply(plus(2,3),[X]).                  =>  X = 5
?- apply(append([1,2]),[X,[1,2,3,4,5]]).  =>  X = [3, 4, 5]
```

  - **Definición de `apply`**

```
apply(Termino,Lista) :-
   Termino =.. [Pred|Arg1],
   append(Arg1,Lista,Arg2),
   Átomo =.. [Pred|Arg2],
   Átomo.
```

Procedimientos aplicativos

- **El procedimiento `maplist`**
  - `maplist(P,L1,L2)` **se verifica si se cumple el predicado** `P` **sobre los sucesivos pares de elementos de las listas** `L1` **y** `L2`**; por ejemplo,**

```
?- succ(2,X).                    =>  3
?- succ(X,3).                    =>  2
?- maplist(succ,[2,4],[3,5]).    =>  Yes
?- maplist(succ,[0,4],[3,5]).    =>  No
?- maplist(succ,[2,4],Y).        =>  Y = [3, 5]
?- maplist(succ,X,[3,5]).        =>  X = [2, 4]
```

  - **Definición de** `maplist`

```
maplist(_,[],[]).
maplist(R,[X1|L1],[X2|L2]) :-
   apply(R,[X1,X2]),
   maplist(R,L1,L2).
```

  - **Definición de** `multiplica_lista` **con** `maplist`

```
multiplica_lista(L1,F,L2) :-
   maplist(producto(F),L1,L2).

producto(X,Y,Z) :- Z is X*Y.
```

Agrupación de todas las soluciones en listas

- **Los procedimientos `findall` y `setof`**
  - `findall(T,O,L)` **se verifica si** `L` **es la lista de las instancias del término** `T` **que verifican el objetivo** `O`**.**
  - `setof(T,O,L)` **se verifica si** `L` **es la lista ordenada sin repeticiones de las instancias del término** `T` **que verifican el objetivo** `O`**.**

```
?- findall(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).
L = [4, 3, 4, 2, 3]
?- setof(X,(member(X,[d,4,a,3,d,4,2,3]),number(X)),L).
L = [2, 3, 4]
?- setof(X,member(X,[d,4,a,3,d,4,2,3]),L).
L = [2, 3, 4, a, d]
?- findall(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).
L = []
Yes
?- setof(X,(member(X,[d,4,a,3,d,4,2,3]),compound(X)),L).
No
```

Agrupación de todas las soluciones en listas

```
?- findall(X,member([X,Y],[[5,0],[3,0],[4,1],[2,1]]),L).
L = [5, 3, 4, 2]
?- setof(X,member([X,Y],[[5,0],[3,0],[4,1],[2,1]]),L).
Y = 0
L = [3, 5] ;
X = _G398
Y = 1
L = [2, 4] ;
No
?- setof(X,Y^member([X,Y],[[5,0],[3,0],[4,1],[2,1]]),L).
L = [2, 3, 4, 5]
```

Agrupación de todas las soluciones en listas

- `setof0(T,O,L)` es como `setof` salvo en el caso en que ninguna instancia de `T` verifique `O`, en cuyo caso `L` es la lista vacía

```
setof0(X,O,L) :- setof(X,O,L), !.
setof0(_,_,[]).
```

- Operaciones conjuntistas

```
intersección(S,T,U) :-
   setof0(X, (member(X,S), member(X,T)), U).
unión(S,T,U) :-
   setof0(X, (member(X,S); member(X,T)), U).
diferencia(S,T,U) :-
   setof0(X, (member(X,S), not(member(X,T))), U).
```

Bibliografía

- **Alonso, J.A. y Borrego, J.** ***Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)*** **(Ed. Kronos, 2002)**
- **Bratko, I.** ***Prolog Programming for Artificial Intelligence (2nd ed.)*** **(Addison–Wesley, 1990)**
- **Clocksin, W.F. y Mellish, C.S.** ***Programming in Prolog (Fourth Edition)*** **(Springer Verlag, 1994)**
- **Covington, M.A.; Nute, D. y Vellino, A.** ***Prolog Programming in Depth*** **(Prentice Hall, 1997)**
- **Sterling, L. y Shapiro, E.** ***L'art de Prolog*** **(Masson, 1990)**
- **Van Le, T.** ***Techniques of Prolog Programming*** **(John Wiley, 1993)**