Práctica 8 : Programación Lógica

Tomás Felipe Melli

June 20, 2025

Índice

El motor de búsqueda y Prolog

1 Ejercicio 1

Consideremos la siguiente base de conocimiento :

```
padre(juan, carlos).
padre(juan, luis).
padre(carlos, daniel).
padre(carlos, diego).
padre(luis, pablo).
padre(luis, manuel).
padre(luis, ramiro).
abuelo(X,Y) :- padre(X,Z), padre(Z,Y).
```

1. Si consultamos `abuelo(X,manuel)` obtenemos :

```
?- abuelo(X,manuel).
X = juan ;
false.
```

2. Nos piden definir los predicados binarios `hijo, hermano` y `descendiente`

```
hijo(X,Y) :- padre(Y, X).

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

descendiente(X,Y) :- padre(Y,X).
descendiente(X,Y) :- padre(Y,Z), descendiente(X,Z).
```

3. Árbol de búsqueda para la consulta `descendiente(Alguien,juan)`

4. Para encontrar los nietos de `juan` podemos hacer `?- abuelo(juan,X).`

5. Para conocer los hermanos de `pablo` podemos hacer `?- hermano(X,pablo).`

6. Con el siguiente hecho y regla

```
ancestro(X,Y).
ancestro(X,Y) :- ancestro(Z,Y), padre(X,Z).
```

7. Lo que pasa con la consulta $ancestro(juan, X)$ es que Prolog primero intenta unificar con el hecho $ancestro(\_, \_)$, que dice que cualquiera es ancestro de cualquiera. Como ese hecho siempre es verdadero, Prolog responde true inmediatamente, aunque no da ningún valor concreto para X. Si luego pedimos más respuestas (;), Prolog usa la regla recursiva, que tiene un llamado a $ancestro(Z, X)$ antes de verificar hechos concretos como *padre*. Eso hace que siga llamando recursivamente sin fin, porque siempre intenta resolver una nueva llamada $ancestro(_, _)$antes de poder usar *padre*. Como resultado, Prolog entra en un ciclo infinito de llamadas recursivas, respondiendo true por siempre sin generar soluciones útiles.

8. Para corregirlo :

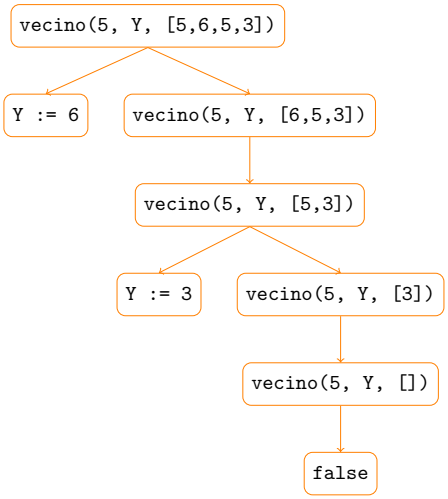
```
ancestro(X,Y) :- padre(X,Y).
ancestro(X,Y) :- padre(X,Z), ancestro(Z,Y).
```

2 Ejercicio 2

Con el siguiente programa

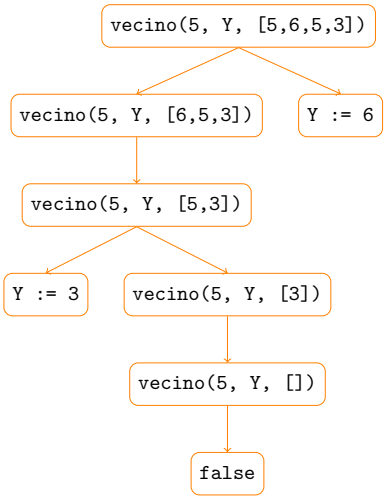
```
vecino(X, Y, [X|[Y|Ls]]).
vecino(X, Y, [W|Ls]) :- vecino(X, Y, Ls).
```

1. Árbol de búsqueda para la consulta `vecino(5, Y, [5,6,5,3])`



2. Si invertimos el orden de las reglas, qué pasa?

```
vecino(X, Y, [W|Ls]) :- vecino(X, Y, Ls).
vecino(X, Y, [X|[Y|Ls]]).
```



El orden de los resultados se invierte ya que Prolog utiliza DFS como método de búsqueda, es decir, busca en profundidad primero, eso hace que entre en la recursión primero hasta obtener el false para realizar el bactracking.

3 Ejercicio 3

Consideremos :

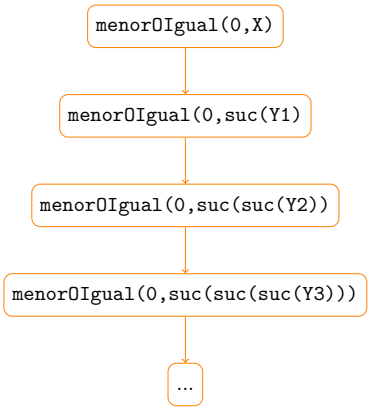
```
natural(0).
natural(suc(X)) :- natural(X).

menorOIgual(X,suc(Y)) :- menorOIgual(X,Y).
menorOIgual(X,X) :- natural(X).
```

1. Si realizamos la consulta `?- menorOIgual(0,X)`

```
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 87.8Mb, trail: 43.9Mb
ERROR:   Stack depth: 5,751,781, last-call: 0%, Choice points: 5,751,774
ERROR:   Possible non-terminating recursion:
ERROR:     [5,751,781] user:menorOIgual(0, _23012478)
ERROR:     [5,751,780] user:menorOIgual(0, <compound suc/1>)
```

Este error sucede ya que el caso base es inalcanzable y como mencionamos antes, la búsqueda es al estilo DFS. Miremos el árbol



2. Se puede colgar si el caso base de la recursión no es alcanzable, o si en ese llamado no hay un avance hacia la terminación.
3. Corrección

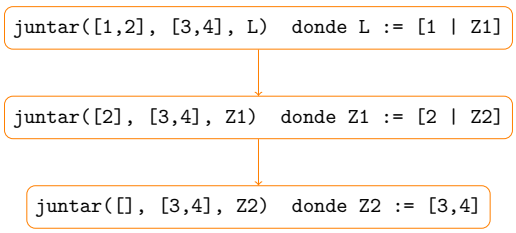
```
menorOIgual(X,X) :- natural(X).
menorOIgual(X,suc(Y)) :- menorOIgual(X,Y).
```

Operaciones sobre Listas

4 Ejercicio 4 : `juntar(?Lista1, ?Lista2, ?Lista3)`

```
juntar([],L,L).
juntar([X|Xs], Ys, [X|Zs]) :- juntar(Xs, Ys, Zs).
```

Ejemplo :



5 Ejercicio 5

5.1 `last(?L, ?U)`

```
last([X|[]],X).
last([_|Xs], Z) :- last(Xs, Z).

last2(X, Y) :- append(_, [Y],X).
```

5.2 reverse(+L, ?R)

```
invertir([],[]).
invertir([X|Xs], L) :- invertir(Xs,L2), append(L2, [X], L).
```

5.3 prefijo(?P, +L)

```
prefijo(L, P) :- append(P, _, L).
```

5.4 sufijo(?S, +L)

```
sufijo(L, S) :- append(_, S, L).
```

5.5 sublista(?S, +L)

```
sublista(_, []).
sublista(L, SL) :- prefijo(L, P), sufijo(P, SL), SL \= [].
```

5.6 pertenece(?X, +L)

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

6 Ejercicio 6 : aplanar

```
aplanar([], []).
aplanar([X|Xs], L) :- not(is_list(X)), aplanar(Xs,L2), append([X],L2, L).
aplanar([X|Xs], L) :- (is_list(X)), aplanar(X,L3), aplanar(Xs,L4), append(L3,L4, L).
```

7 Ejercicio 7

7.1 intersección(+L1, +L2, -L3)

```
interseccion([],_,[]).
interseccion(L1,L2,L) :- interseccion2(L1,L2,[],L3), reverse(L3,L).

interseccion2([],_,LAcc,LAcc).
interseccion2([X|Xs], L2,LAcc, L) :- not(member(X, L2)), interseccion2(Xs, L2, LAcc,L).
interseccion2([X|Xs], L2,LAcc, L) :- member(X, L2), member(X, LAcc), interseccion2(Xs, L2, LAcc,L).
interseccion2([X|Xs], L2,LAcc, L) :- member(X, L2), not(member(X, LAcc)), interseccion2(Xs, L2, [X|LAcc],L
    ).
```

7.2 partir(N,L,L1,L2)

```
partir(0,L,[],L).
partir(N,[X|Xs],[X|L1],L2) :-N > 0, N1 is N-1, partir(N1,Xs, L1,L2).
```

7.3 borrar(+ListaOriginal, +X, -ListaSinXs)

```
borrar(_, [], []).
borrar(X, [X|Xs], L) :- borrar(X, Xs, L).
borrar(X, [H|Xs], [H|L]) :- X \= H, borrar(X, Xs, L).
```

7.4 sacarDuplicados(+L1, -L2)

```
sacarDuplicados([],[]).
sacarDuplicados(X,R) :- sacarDuplicados2(X,[],L), reverse(L, R).


sacarDuplicados2([],LAcc,LAcc).
sacarDuplicados2([X|Xs], LAcc, L) :- member(X, LAcc), sacarDuplicados2(Xs,LAcc,L).
sacarDuplicados2([X|Xs], LAcc, L) :- not(member(X, LAcc)), sacarDuplicados2(Xs,[X|LAcc],L).
```

7.5 permutación(+L1, ?L2)

```
permutacion([], []).
permutacion(L, [X|P]) :-borrar(X, L, R), permutacion(R, P).
```

7.6 reparto(+L, +N, -LListas)

```
reparto([],0,[]).
reparto(L,N,[X|Xs]) :- N > 0, N1 is N-1, append(X, Xs, L), reparto(L, N1, Xs).
```

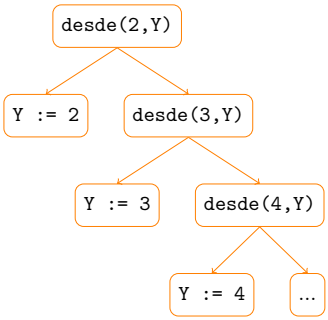
Instanciación y Reversibilidad

8 Ejercicio 9

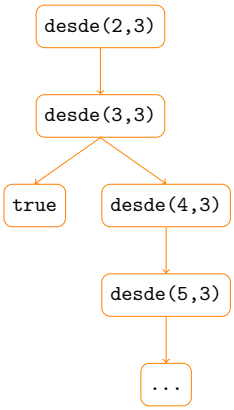
Consideremos

```
desde(X,X).
desde(X,Y) :- N is X+1, desde(N,Y).
```

1. X debe estar instanciado porque sino cómo resuelve : N is X+1. Y no debe estar instanciado, porque se rompe si lo instancias con algo menor a X. Si se cumple el patrón de instanciación, se generan todos los numeros mayores o iguales a X.



Pero si no se cumple el patrón de instanciación :



```
desdeReversible(X,X).
desdeReversible(X,Y) :- var(Y), N is X+1, desdeReversible(N,Y).
desdeReversible(X,Y) :- nonvar(Y), Y > X.
```

9 Ejercicio 11 : árbol binario

En Prolog definimos un AB como `nil` si es vacío, `bin(izq, v, der)` donde v es el valor del nodo. Nos piden definir

9.1 vacío

```
vacio(nil).
```

9.2 raíz

```
raiz(bin(izq, V, der), V).
```

9.3 altura

```
altura(nil,0).
altura(bin(Izq,_,Der), Altura) :- altura(Izq, AltIzq), altura(Der, AltDer), AltDer >= AltIzq,  Altura is AltDer+1.
altura(bin(Izq,_,Der), Altura) :- altura(Izq, AltIzq), altura(Der, AltDer), AltDer < AltIzq,  Altura is AltIzq+1.
```

9.4 cantidadDeNodos

```
cantDeNodos(nil,0).
cantDeNodos(bin(Izq, _, Der), CantNodos) :- cantDeNodos(Izq, CantNodosIzq),  cantDeNodos(Der, CantNodosDer), SumSubarboles is CantNodosDer + CantNodosIzq, CantNodos is  SumSubarboles + 1.
```

En el `solve.pl` hay un test `T = bin(bin(nil, a, nil), b, bin(nil, c, nil)), altura(T, A), cantDeNodos(T, N).`

10 Ejercicio 12

10.1 inorder(+AB, -Lista)

```
inorder(nil, []).
inorder(bin(Izq, V, Der), Lista) :- inorder(Izq, LadoIzq), inorder(Der, LadoDer), append(LadoIzq, [V|LadoDer], Lista).
```

10.2 arbolConInorder(+Lista, -AB)

```
arbolConInorder([], nil).
arbolConInorder(Lista, bin(Izq, V, Der)) :-  append(LadoIzq, [V | LadoDer], Lista), arbolConInorder(LadoIzq, Izq), arbolConInorder(LadoDer, Der).
```

Podemos testearlos con `arbolConInorder([1,2,3], T), inorder(T, L).`

10.3 aBB(+T)

```
aBB(nil).
aBB(bin(Izq, V, Der)) :- aBB(Izq),aBB(Der), inorder(Izq, LadoIzq), mayorATodos(V, LadoIzq), inorder(Der, LadoDer), menorIgual(V, LadoDer).

mayorATodos(X, Lista) :- forall(member(E, Lista), X > E).
menorIgual(X, Lista) :- forall(member(E, Lista), X =< E).
```

Testeamos por **true** con `aBB(bin(bin(bin(nil, 2, nil), 3, nil), 5, bin(bin(nil, 6, nil), 7, bin(nil, 8, nil)))).` y **false** con `aBB(bin(bin(bin(nil, 2, nil), 3, nil), 5, bin(bin(nil, 8, nil), 7, bin(nil, 6, nil)))).`

10.4 aBBInsertar(+X, +T1, -T2)

```
aBBInsertar(X, nil, bin(nil, X, nil)).
aBBInsertar(X, bin(Izq, V, _), bin(IzqRes, V, _)) :- X < V, aBBInsertar(X, Izq, IzqRes).
aBBInsertar(X, bin(_, V, Der), bin(_, V, DerRes)) :- X >= V, aBBInsertar(X, Der, DerRes).
```

Tenemos 4 tests, están en el `solve.pl`.

Generate and Test

11 Ejercicio 13 : coprimos(-X, -Y

```
coprimos(X, Y) :- generarPares(X,Y), X > 0, Y > 0, 1 =:= gcd(X,Y).

generarPares(X,Y) :- desde(0, N), paresQueSuman(N, X, Y).

paresQueSuman(N, X, Y) :- between(0, N, X), Y is N-X.
```

12 Ejercicio 15 : triángulos

12.1 esTriángulo(+T)

```
esTriangulo(tri(A,B,C)) :- valido(A,B,C), valido(B,C,A),valido(C,A,B).
% valido(+A, +B, +C)
valido(A, B, C) :- A < B + C.
```

12.2 perímetro(?T, ?P

```
perimetro(tri(A,B,C), P) :- ground(tri(A,B,C)), esTriangulo(tri(A,B,C)), P is A + B + C.
perimetro(tri(A,B,C), P) :- not(ground(tri(A,B,C))), triplasQueSuman(P, A, B, C), esTriangulo(tri(A,B,C)).

%triplasQueSuman(?P, -A, -B, -C)
triplasQueSuman(P, A, B, C) :- desdeReversible(0,P),between(1,P,A),between(1,P,B),C is P - A - B, C > 0.
```

12.3 triángulo(-T)

```
triangulo(T) :- perimetro(T, _).
```

Negación por Falla y Cut

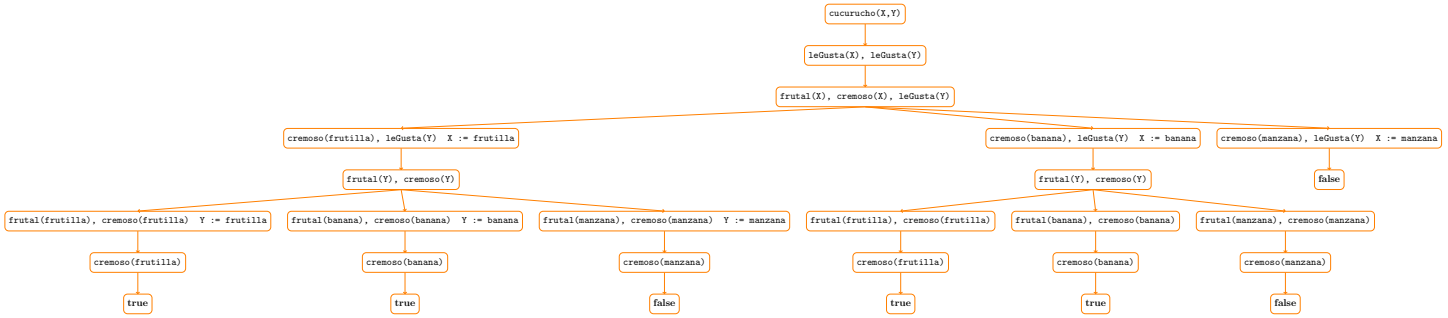
13 Ejercicio 16

Tenemos la siguiente base de conocimiento con estos predicados :

```
frutal(frutilla).
frutal(banana).
frutal(manzana).
cremoso(banana).
cremoso(americana).
cremoso(frutilla).
cremoso(dulceDeLeche).

leGusta(X) :- frutal(X), cremoso(X).
cucurucho(X,Y) :- leGusta(X), leGusta(Y).
```

1. Árbol de búsqueda para la consulta `?- cucurucho(X,Y)`



Se ve poco, pero la consulta devuelve :

```
X = Y, Y = frutilla ;
X = frutilla,
Y = banana ;
X = banana,
Y = frutilla ;
X = Y, Y = banana ;
```

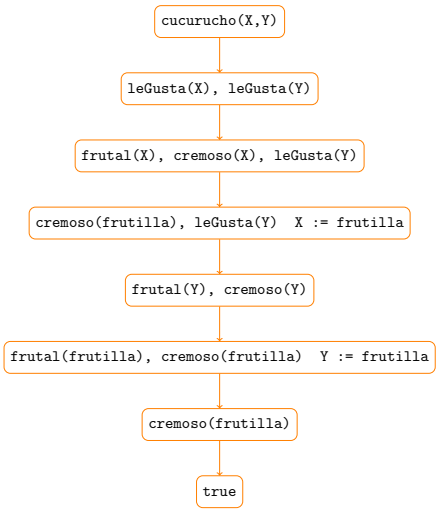
Esto como vemos en el árbol genera resultados repetidos, y por ello nos piden ...

2. Colocar una poda con `cut (!)` , es decir, decirle a Prolog que deje de meterse en una rama cuando ya encontró una solución (o sea, corta el backtracking hacia decisiones anteriores en ese predicado). Por ejemplo si modificamos esta regla y consultamos por `cucurucho(X,Y)`:

```
leGusta(X) :- frutal(X), cremoso(X), !.

?- cucurucho(X,Y).
X = Y, Y = frutilla.
```

Miremos el árbol



En este escenario Prolog **no va a explorar otras frutas**. Si consideramos por ejemplo algo como

```
leGusta(X) :- frutal(X), cremoso(X).
cucurucho(X,Y) :- leGusta(X), leGusta(Y), !.

?- cucurucho(X,Y).
X = Y, Y = frutilla.
```

Prolog una vez que encuentra el par que cumple, frena. (Para otras combinaciones mirar `solve.pl`)

14 Ejercicio 17

Consultar

15 Ejercicio 18 : corteMásParejo(+L, -L1, -L2)

```
corteMasParejo(L, I, D) :- append(I, D, L), not(hayUnCorteMasParejo(I,D,L)).

hayUnCorteMasParejo(I,D,L) :- append(I2, D2, L), esMasParejo(I2, D2, I, D).

esMasParejo(I2, D2, I, D) :-
    sum_list(I2, SI2), sum_list(D2, SD2),
    sum_list(I, SI), sum_list(D, SD),
    abs(SI - SD) >  abs(SI2 - SD2).
```