

# Práctica 9 : Programación Lógica (Parte 1)

Tomás Felipe Melli

June 9, 2025

## Índice

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Ejercicio : Representar los <math>N_0</math></b>	<b>5</b>
<b>3</b>	<b>Sustitución y Unificación</b>	<b>5</b>
<b>4</b>	<b>Reversibilidad</b>	<b>6</b>
<b>5</b>	<b>Listas</b>	<b>7</b>
<b>6</b>	<b>Estructuras parcialmente instanciadas</b>	<b>9</b>

# 1 Introducción

**Prolog** es un lenguaje de programación lógica. En este lenguaje los programas se escriben en un **subconjunto de la LPO** (o sea, se basa en la LPO pero no implementa la totalidad de su expresividad). Es **declarativo**, y por tanto, tenemos que especificar los **hechos, reglas de inferencia y objetivos** sin indicar cómo se obtiene este último a partir de los primeros. El modelo de cómputo (motor lógico) de prolog está basado en **cláusulas de Horn y resolución SLD (Selective Linear Definite clause resolution)**. Es decir :

- Una cláusula de Horn es una fórmula lógica que tiene la forma  $A : - B_1, B_2, \dots, B_n$  que en lógica es equivalente a  $(B_1 \wedge B_2 \wedge \dots \wedge B_n) \rightarrow A$ . Una regla lógica de una única conclusión.
- La resolución SLD (Selective Linear Definite clause resolution) es el mecanismo de inferencia que utiliza prolog para buscar respuestas a las consultas :
  - Selective: selecciona una submeta a resolver (de izquierda a derecha).
  - Linear: trabaja con una única cadena de resoluciones (una a la vez).
  - Definite clause: las cláusulas de Horn son un subconjunto especial con una sola cabeza.

El algoritmo sería, dada una **base de conocimiento** y una consulta

1. Elige una submeta de la consulta.
2. Busca una cláusula cuya cabeza unifique con esa submeta.
3. Reemplaza la submeta por el cuerpo de esa cláusula.
4. Repite hasta:
  - Resolver todas las submetas (éxito), o
  - No encontrar coincidencias (falla  $\rightarrow$  backtracking).

Decimos que Prolog es un lenguaje de **mundo cerrado** (*Closed World Assumption (CWA)*), es decir, sólo se puede suponer lo que se declaró explícitamente, todo lo que no pueda deducirse a partir de la base de conocimiento, se supone falso. Si pensamos en LPO, en esta , *el hecho de que algo no esté demostrado como verdadero no implica que sea falso* – *podría ser desconocido*, es la *Open World Assumption*.

Sólo hay un tipo en Prolog, los **términos**.

## Base de conocimiento

Podemos pensar los programas como **bases de conocimiento** que describen el dominio del problema. Están formados por hechos y reglas de inferencia. La idea es consultar sobre esa base.

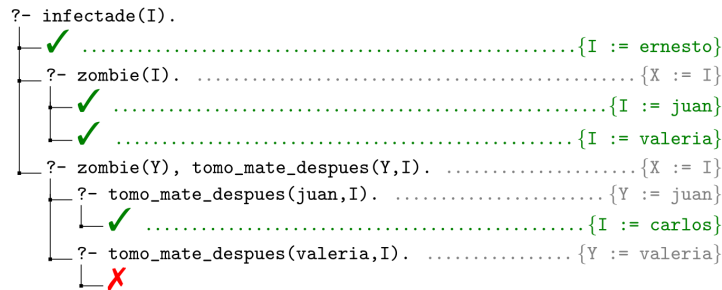
```
1 % base de conocimiento
2 zombie(juan).
3 zombie(valeria).
4
5 tomo_mate_despues(juan,carlos).
6 tomo_mate_despues(clara,juan).
7
8 infectado(ernesto).
9 infectado(X) :- zombie(X).
10 infectado(X) :- zombie(Y), tomo_mate_despues(Y,X).
```

## Cláusulas y Consultas

```
1 2 ?- zombie(juan).
2 true.
3
4 3 ?- tomo_mate_despues(juan,X).
5 X = carlos.
6
7 4 ?- infectado(I).
8 I = ernesto ;
9 I = juan ;
10 I = valeria ;
11 I = carlos ;
12 false.
```

Cuando escribimos la consulta por el infectado, si damos enter se termina, pero si presionamos ";" sigue consultado.

## Seguimiento de consulta



## Útil

- `swipl nombre-archivo.pl` para iniciar el intérprete con ese archivo
- `?- make.` para recargarlo
- `%` comentarios

## Syntax

- **Variables :** tenemos 3 tipos:

1. Normal : comienzan con mayúscula y se usan para obtener o comparar valores. Prolog las unifica y se pueden reutilizar. Ejemplo **Hijo**

```
1 padre(juan, maria).
2 padre(juan, carlos).
3
4 ?- padre(juan, Hijo).
5 Hijo = maria ;
6 Hijo = carlos.
```

Que guarda el valor y podría reutilizarse.

2. Anónima (-): Se usan cuando no importa el valor que puede tomar una variable. Sirven para ignorar argumentos.
- Ejemplo

```
1 padre(juan, maria).
2 padre(juan, carlos).
3
4 ?- padre(juan, _).
5 true.
```

No nos importa de quién, sólo si es padre.

3. Variables que comienzan con `_` : se utilizan para mostrar que no será reutilizada pero sí se comporta como una variable normal ya que guarda el valor.

- **Átomo** : un nombre constante, se usa para representar entidades, propiedades, entre otras cosas. Básicamente son valores indivisibles (sin estructura interna). Ejemplos

```
1 juan
2 'ejemplo con espacio'
```

- **Término Compuesto (Estructura)** : Consiste en un objeto con componentes. Tiene estructura interna. Consiste en

```
1 atomo(arg1, arg2, ..., argN)
```

Donde cada *arg* es un término y decimos que *n* es la **aridad** del término compuesto. Ejemplo:

```
1 tomo_mate_despues(clara,juan)
```

- **Término:** un término en prolog es la unidad básica de datos, es decir, cualquier cosa que puede aparecer como parte de una cláusula, un argumento de un predicado o una expresión lógica. (variable, número, átomo, término compuesto). Estos nos permitirán **construir conocimiento** y expresar **hechos y reglas**.

Reglas: hermano(X, Y) :- padre(Z, X), padre(Z, Y).

- **Números** : En Prolog, los números son **términos atómicos (átomo)** que pueden representar valores enteros, flotantes o expresiones aritméticas. Pero ojo! No podemos hacer **comparaciones numéricas** como estamos acostumbrados...

```
1 ?- 2 + 2 = 4.
2 false.
```

Ya que en prolog, se comparan estructuras, y en este caso  $2 + 2$  es una estructura que no fue evaluada. Para realizar comparaciones contamos con **comparadores evaluativos**

- **==** : igualdad numérica
- **\=** : distinto numéricamente
- **Comparaciones estandar** :  $<, >, =, <=, >=$  (fijarse que en prolog no podemos hacer aparecer una flecha acá)

Contamos con **operadores aritméticos** como

- $+, -, *, /$
- **div** , **mod**
- **abs**
- **max(X,Y)** , **min(X,Y)**
- **sqrt(X)**
- **round(X)** , **floor(X)** , **ceiling(X)** , **truncate(X)**

Algo interesante de los números es el **predicado is**. Este se usa para asignar resultados numéricos. Este predicado existe ya que prolog no es simbólico por defecto, es decir, no hace cuentas automáticamente ya que trata a los términos como estructuras y es evaluativo si se le indica (con **is** o los comparadores evaluativos).

```
1 ?- X = 3 + 4.
2 X = 3+4.    % No lo eval a
3
4 ?- X is 3 + 4.
5 X = 7.      % Ahora s
6
7 ?- X is Y + 1.
8 ERROR: Arguments are not sufficiently instantiated
```

Falla si Y no tiene valor aún. **is** necesita que la derecha esté completamente instanciada.

- **Cláusula** : una cláusula es una unidad básica de conocimiento. **Hechos** (cláusulas sin cuerpo), **reglas**(cláusulas con cuerpo) y **negación**(cláusulas vacías) son unidades de conocimiento. Ejemplos

```
1 gato(felix).    % hecho
2
3 hermano(X, Y) :- padre(Z, X), padre(Z, Y).    % regla
4
5 :- false.    % negacion
```

- **Predicado** : una colección de cláusulas.
- **Objetivo (goal)** : es una expresión lógica que le pedimos a Prolog que pruebe.

```
1 ?- objetivo.
```

con cierta base de conocimiento, prolog intentará demostrarlo y si lo logra nos da

```
1 true.
```

Podemos usar varios objetivos separados por coma también

```
1 ?- padre(juan, maria), padre(juan, pedro).
```

Podemos consultar con variables también

```
1 ?- padre(juan, X).
```

## 2 Ejercicio : Representar los $N_0$

Queremos representar los naturales con el `cero` y `suc(X)`

```

1 % Ejercicio - Naturales con el 0
2 natural(cero).
3 natural(suc(M)) :- natural(M).
4
5
6 mayorA2(suc(suc(suc(_)))) . % podemos hacerlo expl citamente
7 % mayorA2(suc(suc(suc(X)))) :- natural(X). de manera de generar todos
8
9 esPar(cero).
10 esPar(suc(suc(X))) :- esPar(X). % si X es par, entonces el sucesor del sucesor tambi n lo es
11
12 menor(cero, suc(_)). % cero es m s chico que todos
13 menor(suc(X),suc(Y)) :- menor(X,Y).

1 3 ?- menor(cero,uno).
2 false.
3
4 4 ?- menor(cero,X).
5 X = suc(_).
6
7 5 ?- menor(suc(cero),cero).
8 false.
9
10 6 ?- menor(X,suc(suc(suc(cero)))).
11 X = cero .
12
13 7 ?- menor(X,Y).
14 X = cero,
15 Y = suc(_).

```

## 3 Sustitución y Unificación

Una sustitución es una función  $S : Var \rightarrow Term$  donde  $Term$  es el conjunto de todos los términos. Usamos estas sustituciones con el **mgu** para igualar literales y aplicar la regla de resolución. En particular, Prolog lo que hace es: recibe un programa **P** y un **goal**  $G_1, \dots, G_n$  (por ejemplo : `?- padre(juan, X), hermano(X, Y).`); queremos saber si el goal es consecuencia lógica de P. La regla que se utiliza es :

$$\frac{G_1, \dots, G_n \quad H : - A_1, \dots, A_k \quad S \text{ es el MGY de } G_1 \text{ y } H}{S(A_1, \dots, A_k, G_2, \dots, G_n)}$$

Es decir, si  $G_1$  **unifica con la cabeza H de una cláusula del programa, entonces sustituye  $G_1$  por el cuerpo  $A_1, \dots, A_k$  de esa cláusula y aplica la sustitución S a todo (incluido  $G_2, \dots, G_n$ )**. Hagamos un ejemplo para visualizar cómo Prolog aplica esta regla :

Sea la base de conocimiento :

```

1 padre(juan, maria).
2 padre(juan, pedro).
3 abuelo(X, Y) :- padre(X, Z), padre(Z, Y).

```

Queremos consultar :

```

1 ?- abuelo(juan, Y).

```

Queremos saber de quién es abuelo juan, o sea,  $G_1 = abuelo(juan, Y)$ . Entonces, buscamos la cláusula para aplicar, que en este caso es  $abuelo(X, Y)$  que llamaremos  $H$  y por tanto,  $A_1 = padre(X, Z)$  y  $A_2 = padre(Z, Y)$ . Unificamos  $G_1$  con  $H$  y nos queda

$$\begin{aligned}
 G_1 &\stackrel{?}{=} H \\
 abuelo(juan, Y) &\stackrel{?}{=} abuelo(X, Y) \\
 abuelo(juan, Y) &\stackrel{?}{=} abuelo(juan, Y) \text{ con } S = \{ X := juan \}
 \end{aligned}$$

Con esto...

$$\frac{abuelo(juan, Y) \quad abuelo(X, Y) : padre(X, Z), padre(Z, Y)}{padre(juan, Z), padre(Z, Y)}$$

Ahora tenemos nuevos objetivos :  $G'_1 = padre(juan, Z)$  y  $G'_2 = padre(Z, Y)$

Tratamos de unificar por un lado con...  $S2 := \{ Z := maria \}$

$$\frac{\frac{abuelo(juan, Y) \quad abuelo(X, Y) : padre(X, Z), padre(Z, Y)}{padre(juan, Z), padre(Z, Y)} \quad padre(juan, maria)}{padre(maria, Y)}$$

Pero como **falla**, hace **backtracking** y tratamos de unificar por el otro con...  $S3 := \{ Z := pedro \}$

$$\frac{\frac{abuelo(juan, Y) \quad abuelo(X, Y) : padre(X, Z), padre(Z, Y)}{padre(juan, Z), padre(Z, Y)} \quad padre(juan, pedro)}{padre(pedro, Y)}$$

Pero en ambos casos, **falla**. Como exploró todas las ramas en ninguna pudo derivar exitosamente, **concluye false**. (“No hay ninguna manera de satisfacer el goal con las reglas y hechos dados”). Esta forma de explorar las ramas sigue el esquema del **DFS**, es por ello que el orden de las cláusulas y los literales dentro de cada regla afecta a la eficiencia. Prolog intenta resolver **los goals de izquierda a derecha**.

## 4 Reversibilidad

La reversibilidad en prolog es la capacidad que tiene para ejecutar un predicado en múltiples direcciones, es decir,

- Usar un predicado para generar soluciones a partir de variables no instanciadas.
- Usar el mismo predicado para verificar si una solución cumple ciertas condiciones.

O sea, se puede usar un predicado para consultar, generar o probar, dependiendo de qué variables estén instanciadas o libres. Veamos un ejemplo claro

```
1 append([], Ys, Ys).
2 append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Podemos consultar, es decir, le damos los argumentos y queremos saber el resultado....

```
1 3 ?- append([1,2], [3], R).
2 R = [1, 2, 3].
```

O, podemos pasarle el resultado y queremos saber qué argumentos cumplen...

```
1 4 ?- append(A, B, [1,2,3]).
2 A = [],
3 B = [1, 2, 3] ;
4 A = [1],
5 B = [2, 3] ;
6 A = [1, 2],
7 B = [3] ;
8 A = [1, 2, 3],
9 B = [] ;
```

Recién hablamos de variables instanciadas o libres, esto se convierte en un problema para la reversibilidad por ejemplo en el siguiente predicado

```
1 suma(X, Y, Z) :- Z is X + Y.
```

Ya que, podemos consultar por

```
1 6 ?- suma(2,3,Z).
2 Z = 5.
```

Pero si queremos usar **is** con variables no instanciadas, falla. Esto es un claro ejemplo de un predicado no reversible (ya que evalúa, no unifica)

```
1 7 ?- suma(2,Z,3).
2 ERROR: Arguments are not sufficiently instantiated
```

Tenemos una notación especial a la hora de leer una especificación de un predicado (modo de instanciación):

- **+X** : debe estar instanciado
- **-X** : no debe estar instanciado
- **?X** : puede o no debe estar instanciado

Con esta notación nos piden definir el predicado **entre(+X,+Y,-Z)** que sea verdadero cuando el número  $\in Z$  esté comprendido entre los enteros X e Y (inclusive). O sea nos piden algo como

```
1 ?- entre(1,3,Z).
2 Z = 1;
3 Z = 2;
4 Z = 3;
5 false.
```

Entonces, definimos el predicado

```
1 entre(X,Y,X) :- X =< Y.
```

Que verifica si  $X \leq Y$  con lo cual,  $Z = X$ , si no lo es, tenemos que definir una regla

```
1 entre(X,Y,Z) :- X < Y, X1 is X + 1, entre(X1, Y, Z).
```

que nos permite incrementar X para explorar el siguiente número en el rango. En particular, poder generar lo pedido :

```
1 17 ?- entre(1,3,Z).
2 Z = 1 ;
3 Z = 2 ;
4 Z = 3 ;
5 false.
6
7 18 ?- entre(1,5,3).
8 true .
9
10 19 ?- entre(1,2,3).
11 false.
```

## 5 Listas

Las listas en Prolog son una colección ordenada de elementos, se definen usando corchetes `[ ]`. La estructura interna es `[Head | Tail]`. Un ejemplo

```
1 [1, 2, 3] = [1 | [2 | [3 | [ ]]]]
```

Podemos por ejemplo definir la función *longitud*

```
1 longitud([], 0).
2 longitud([_|T], N) :- longitud(T, N1), N is N1 + 1.
```

Nos piden definir:

1. **long(+L,-N)** que relaciona una lista con su longitud.
2. **sacar(+X, +XS, -YS)** que elimina todas las ocurrencias de X.
3. **sinConsecRep(+XS, -YS)** que elimina repeticiones consecutivas

```
1 long([], 0).
2 long([_|T], N) :- long(T, N1), N is N1 + 1.
3
4 sacarTodas(_, [], []).
5 sacarTodas(X, [X|T], R) :- sacarTodas(X, T, R).
6 sacarTodas(X, [H|T], [H|R]) :- X \= H, sacarTodas(X, T, R).
7
8 sinConsecRep([], []).
9 sinConsecRep([X], [X]).
10 sinConsecRep([X,X|T], R) :- sinConsecRep([X|T], R).
11 sinConsecRep([X,Y|T], [X|R]) :- X \= Y, sinConsecRep([Y|T], R).
```

Obtenemos entonces ...

```
1 39 ?- sacarTodas(2,[1,2,2,3,2],L).
2 L = [1, 3] ;
3
4 40 ?- sinConsecRep([1,2,2,3,2],L).
5 L = [1, 2, 3, 2] ;
```

A continuación :

1. prefijo(+L, ?P)
2. sufijo(+L, ?S)
3. sublista(+L, ?SL)
4. insertar(?X, +L, ?LX)
5. permutacion(+L, ?P)

```
1 prefijo(L, P) :- append(P, _, L).
2
3 sufijo(L, S) :- append(_, S, L).
4
5 sublista(L, SL) :- append(_, L2, L), append(SL, _, L2).
6
7 insertar(X, L, LX) :- append(P, S, L), append(P, [X|S], LX).
8
9 sacarUna(X, [X|T], T).
10 sacarUna(X, [H|T], [H|R]) :- sacarUna(X, T, R).
11
12 permutacion([], []).
13 permutacion(L, [X|P]) :-sacarUna(X, L, R), permutacion(R, P).
```

Con esto,

```
1 25 ?- prefijo([1,2,3], P).
2 P = [] ;
3 P = [1] ;
4 P = [1, 2] ;
5 P = [1, 2, 3] ;
6 false.
7
8 25 ?- sufijo([1,2,3], S).
9 S = [1, 2, 3] ;
10 S = [2, 3] ;
11 S = [3] ;
12 S = [] ;
13 false.
14
15 26 ?- sublista([1,2,3], SL).
16 SL = [] ;
17 SL = [1] ;
18 SL = [1, 2] ;
19 SL = [1, 2, 3] ;
20 SL = [] ;
21 SL = [2] ;
22 SL = [2, 3] ;
23 SL = [] ;
24 SL = [3] ;
25 SL = [] ;
26 false.
27
28 27 ?- insertar(9, [1,2,3], LX).
29 LX = [9, 1, 2, 3] ;
30 LX = [1, 9, 2, 3] ;
31 LX = [1, 2, 9, 3] ;
32 LX = [1, 2, 3, 9] ;
33 false.
34
35 33 ?- permutacion([1,2,3], P).
36 P = [1, 2, 3] ;
37 P = [1, 3, 2] ;
38 P = [2, 1, 3] ;
39 P = [2, 3, 1] ;
40 P = [3, 1, 2] ;
41 P = [3, 2, 1] ;
42 false.
```



## 6 Estructuras parcialmente instanciadas

```
1 capicua([]).
2 capicua([_]).
3 capicua([H|T]) :- append(M, [H], T), capicua(M).
4
5 capicuaB ( L ) : - reverse (L , L ).
```

Qué pasa si L no está instanciada ?

```
1 42 ?- capicua(L).
2 L = [] ;
3 L = [_] ;
4 L = [_A, _A] ;
5 L = [_A, _, _A] ;
6 L = [_A, _B, _B, _A] ;
7 L = [_A, _B, _, _B, _A] ;
8 L = [_A, _B, _C, _C, _B, _A] ;
9 L = [_A, _B, _C, _, _C, _B, _A] .
```

Consideremos ...

```
1 % member
2 member(X,[X|_]).
3 member(X,[_|L]) :- member(X,L).
```

Miremos las siguientes consultas

```
1 44 ?- member(2,[1,2]).
2 true .
3
4 45 ?- member(X,[1,2]).
5 X = 1 ;
6 X = 2 ;
7 false.
8
9 46 ?- member(5,[X,3,X]).
10 X = 5 ;
11 X = 5 ;
12 false.
13
14 1 ?- member(2,[1,2,2]).
15 true .
16
17 2 ?- length(L,2), member(5,L), member(2,L).
18 L = [5, 2] ;
19 L = [2, 5] ;
20 false.
```