Programación Lógica - Parte 1

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

6 de junio de 2025

Sobre Prolog

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.
- Es declarativo: se especifican hechos, reglas de inferencia y objetivos, sin indicar cómo se obtiene este último a partir de los primeros.
- Cómputo basado en cláusulas de Horn y resolución SLD.
- Mundo cerrado: sólo se puede suponer lo que se declaró explícitamente, todo lo que no pueda deducirse a partir del programa se supone falso.
- Tiene un solo tipo: los términos.

Bases de conocimiento

Podemos pensar los programas en Prolog como bases de conocimiento que describen el dominio del problema. Están formados por hechos y reglas de inferencia. Se utilizan realizando consultas sobre dicha base.

Base de conocimiento

Juan y Valeria son zombies.
Carlos tomó mate después de Juan. Juan tomó mate después de Clara.
Ernesto está infectado, aunque no sabemos por qué.
Les zombies están infectades.
Si alguien (X) tomó mate después de otre (Y) zombie, entonces X está infectade.

¿Qué consultas podríamos hacer sobre esta base?

Consultas

¿Está Carlos infectado? ¿Está Clara infectada?
¿Quiénes están infectades?

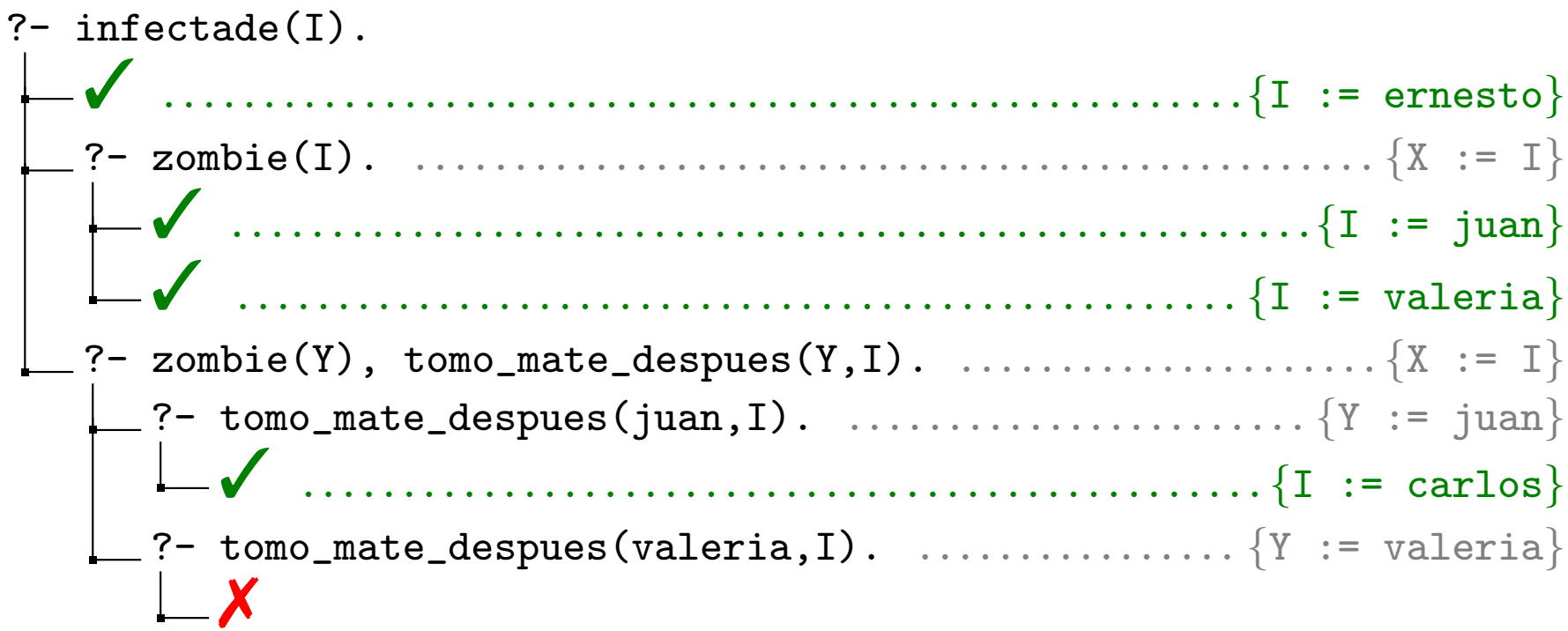
Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomo_mate_despues(juan,carlos).
tomo_mate_despues(clara,juan).

infectade(ernesto).
infectade(X) :- zombie(X).
infectade(X) :- zombie(Y), tomo_mate_despues(Y,X).
```

```
?- zombie(juan).
true.

?- tomo_mate_despues(juan,X).
X = carlos.

?- infectade(I).
I = ernesto;
I = juan;
I = valeria;
I = carlos;
false.
```

Seguimiento de la consulta

```
zombie(juan).
zombie(valeria).

tomo_mate_despues(juan,carlos).
tomo_mate_despues(clara,juan).

infectade(ernesto).
infectade(X) :- zombie(X).
infectade(X) :- zombie(Y), tomo_mate_despues(Y,X).
```



Herramientas de Prolog

- Descargar https://www.swi-prolog.org (Download → SWI-Prolog → Stable release)
- `swipl archivo.pl` para iniciar el intérprete con cierto archivo cargado.
- `swipl` y luego `?- [archivo].` para cargarlo una vez dentro del intérprete.
- `?- make.` para recargar cambios de archivos.
- `?- consulta.` para evaluar una consulta.
- `;` para ver la siguiente solución.
- `.` para dejar de listar soluciones.
- `CTRL+D` o `?- halt.` para salir del intérprete.
- `?- E.` quizás para reírse.
- `:- use_module(archivo).` para importar un archivo dentro de otro.
- `% comentario` o `%! documentación`
- `?- help(predicado).` para ver la documentación de un predicado.

Sintaxis de Prolog

- **Variables:** `X, Persona, _var`
  Valores que todavía no fueron ligados. Después de ligarse ya no pueden ser modificadas. Empiezan con mayúscula o _.

- **Números:** `10, 15.6`

- **Átomos:** `zombie, 'hola␣mundo'`
  Constantes, texto, nombres de términos compuestos. Empiezan con minúscula o están entre comillas simples.

- **Términos compuestos:** `tomo_mate_despues(clara,juan)`
  También llamado estructura. Consiste en un nombre (átomo) seguido de *n* argumentos, cada uno de los cuales es un término. Decimos que *n* es la aridad del término compuesto.

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.
- **Cláusula:** es una línea del programa. Termina con punto. Puede ser:
  - **Hecho:** `zombie(juan).`
  - **Regla:** `infectade(X) :- zombie(Y), tomo_mate_despues(Y,X).`
    El símbolo `:-` se puede pensar como un $\Leftarrow$, y las comas como $\wedge$.
- **Predicado:** colección de cláusulas.
- **Objetivo (goal):** es el predicado que se consulta al motor de Prolog. Por ejemplo:
  `infectade(X), tomo_mate_despues(X,Y).`

Ejercicios

Vamos a representar a los naturales (incluyendo el 0) como `cero` y `suc(X)`. Consideremos el siguiente predicado:

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

Escribir los siguiente predicados

- `mayorA2(X)` que es verdadero cuando X es mayor que 2.
- `esPar(X)` que es verdadero cuando X es par.
- `menor(X,Y)` que es verdadero cuando X es menor que Y.

Indicar qué ocurre si efectuamos las siguientes consultas:

- `menor(cero,uno).`
- `menor(cero,X).`
- `menor(suc(cero),cero).`
- `menor(X,suc(suc(suc(cero)))).`
- `menor(X,Y).`

Sustitución y Unificación

Sea *Term* el conjunto formado por todos los posibles términos. Una **sustitución** es una función $S : \mathit{Variables} \rightarrow \mathit{Term}$. Podemos extender S a una función $\mathit{Term} \rightarrow \mathit{Term}$ de la siguiente manera:

$$S(c) = c$$

$$S(f(t_1, ..., t_n)) = f(S(t_1), ..., S(t_n))$$

Por ejemplo, si $S = \{X := a, Y := Z\}$, entonces:

$$S(b(X, Y, c)) = b(S(X), S(Y), S(c)) = b(a, Z, c)$$

Usamos sustituciones obtenidas con el **unificador más general (MGU)** para igualar literales y aplicar la regla de resolución.

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

La regla de resolución que se utiliza es:

$$\frac{G_1, \ldots, G_n \qquad H \ :- \ A_1, \ldots, A_k \qquad S \text{ es el MGU de } G_1 \text{ y } H}{S(A_1, \ldots, A_k, G_2, \ldots, G_n)}$$

La conclusión de la regla de resolución es el nuevo goal a resolver.

Prolog resuelve el goal empezando desde G_1, de izquierda a derecha y haciendo DFS. Para cada G_i, recorre el programa de arriba hacia abajo buscando unificar G_i con la cabeza de una cláusula.

Tener en cuenta que el orden de las cláusulas y sus literales en el programa influyen en el resultado.

En este link hay ejemplos sobre el proceso de reducción de Prolog.

Ejemplo de resolución

Veamos un ejemplo. Sea el siguiente programa:

```
gato(garfield).
tieneMascota(john,odie).
tieneMascota(john,garfield).
amaALosGatos(X) :- tieneMascota(X,Y), gato(Y).
```

Mostrar el seguimiento (árbol de ejecución) de la consulta: `amaALosGatos(Z).`

Ejemplo de resolución

¿Qué pasa si cambiamos el orden de algunas cláusulas?

```
gato(garfield).
tieneMascota(john,garfield).
tieneMascota(john,odie).
amaALosGatos(X) :- tieneMascota(X,Y), gato(Y).
```

y el goal: `amaALosGatos(Z).`

Resolución

Al evaluar un goal, los resultados posibles son los siguientes:

- `true`: la resolución terminó en la cláusula vacía.
- `false`: la resolución terminó en una cláusula que no unifica con ninguna regla del programa.
- El proceso de aplicación de la regla de resolución no termina.

Reversibilidad

Un predicado define una relación entre elementos. No hay parámetros de "entrada" ni de "salida".

Conceptualmente, cualquier argumento podría cumplir ambos roles dependiendo de cómo se consulte.

Un predicado podría estar implementado asumiendo que ciertas variables ya están instanciadas, por diversas cuestiones prácticas.

Patrones de instanciación

El modo de instanciación esperado por un predicado se comunicará en los comentarios.

```
%! pow(+B,+E,-P)
% Es verdadero si P es igual a B elevado a E.
pow(...) :- ...
```

+X	−X	?X
debe estar instanciado	**no** debe estar instanciado	puede o no estar instanciado

Se debe tener en cuenta que el usuario no puede suponer más cosas de las que se especificaron. En caso de llamar a un predicado con argumentos instanciados de otra manera, el resultado puede no ser el esperado.

Aritmética

El motor de operaciones aritméticas de Prolog es independiente del motor lógico (es extra-lógico).

Expresión aritmética:

- Un número.
- Una variable ya instanciada en una expresión aritmética.
- `E1+E2`, `E1-E2`, `E1*E2`, `E1/E2`, etc, siendo `E1` y `E2` expresiones aritméticas.

Aritmética

Algunos operadores aritméticos:

- `E1 < E2`, `E1 =< E2`, `E1 >= E2`, `E1 =:= E2`, `E1 =\= E2`: evalúa ambas expresiones aritméticas y realiza la comparación indicada. ($\geq$ y $\leq$ se escriben de forma que **no** formen flechas).

- `X is E`: tiene éxito sí y sólo si `X` unifica con el resultado de evaluar la expresión aritmética `E`.

Algunos operadores no aritméticos:

- `X = Y`: tiene éxito si y sólo si `X` unifica con `Y`.

- `X \= Y`: `X` no unifica con `Y`. Ambos términos deben estar instanciados.

Aritmética

```
?- 1+1 =:= 2.
true.

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
true.

?- 1+1 is 2.
false.

?- 1+1 is 1+1.
false.
```

Ejercicio

Definir el predicado `entre(+X,+Y,-Z)` que sea verdadero cuando el número entero `Z` esté comprendido entre los números enteros `X` e `Y` (inclusive).

Notar que lo que se nos pide es un predicado capaz de instanciar sucesivamente `Z` en cada número entero entre `X` e `Y`:

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

Listas

Sintaxis:

- `[]`
- `[H | T]`
- `[X,Y,...,Z | L]`

Ejemplos:

- `[1,2], [1,2 | []], [1 | [2]], [1 | [2 | []]]`
- `[1,cero,'hola mundo',[3,5]]`

Ejercicios sobre listas

- Definir el predicado `long(+L,-N)` que relaciona una lista con su longitud.
- ¿Es `long/2` reversible? ¿Por qué? ¿Cómo lo notamos entonces?
- Definir el predicado `sacar(+X,+XS,-YS)`, que relaciona un elemento y una lista con otra lista como la original pero sin el elemento dado. Por ejemplo:
  ```
  ?- sacar(2,[1,2,2,3,2],L).
  L = [1,3].
  ```
- Definir el predicado `sinConsecRep(+XS,-YS)` que relaciona una lista con otra que contiene los mismos elementos sin las repeticiones consecutivas. Por ejemplo:
  ```
  ?- sinConsecRep([1,2,2,3,2],L).
  L = [1,2,3,2].
  ```

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

- `prefijo(+L,?P)`: que tiene éxito si `P` es un prefijo de la lista `L`.
- `sufijo(+L,?S)`: que tiene éxito si `S` es un sufijo de la lista `L`.
- `sublista(+L,?SL)`: que tiene éxito si `SL` es una sublista de `L`.
- `insertar(?X,+L,?LX)`: que tiene éxito si `LX` puede obtenerse insertando a `X` en alguna posición de `L`.
- `permutacion(+L,?P)`: que tiene éxito si `P` es una permutación de la lista `L`.

Estructuras parcialmente instanciadas

`capicua(?Lista)` es verdadero si `Lista` es capicúa. Por ejemplo: `[1,2,1]`

```
capicua([]).
capicua([_]).
capicua([H|T]) :- append(M, [H], T), capicua(M).
```

Pero puede no estar instanciada, ¿Cómo se comporta en tal caso?

```
?- capicua(L).
L = [] ;
L = [_] ;
L = [_A, _A] ;
L = [_A, _, _A] ;
L = [_A, _B, _B, _A] ;
L = [_A, _B, _, _B, _A] ;
```

Solución alternativa:

```
capicua(L) :- reverse(L, L).
```

Ejercicios sobre listas

Considerando el siguiente predicado:

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

Realizar un seguimiento de las siguientes consultas:

- `?- member(2,[1,2]).`
- `?- member(X,[1,2]).`
- `?- member(5,[X,3,X]).`
- `?- member(2,[1,2,2]).`
- `?- length(L,2), member(5,L), member(2,L).`

Fin

¿¿¿¿¿Preguntas?????