title: "Práctica 4"

date: "2025-05-31"

summary: "Este informe presenta los conceptos básicos de Prolog, un lenguaje lógico orientado al razonamiento simbólico y se detalla el proceso de instalación de SWI-Prolog y el uso del intérprete con herramientas como trace. para depuración. Se probaron varios archivos (operadores.pl, option.pl, loop.pl, etc.) abordando estructuras como control de flujo, listas, bucles, conjunciones y disyunciones."

Universidad Autónoma de Baja California

Facultad de Ingeniería, Arquitectura y Diseño

Paradigmas de la programación

Práctica 4

Desarrollo e introducción a Prolog

Arturo Rafael Cornejo Escobar

31 de abril del 2025

INTRODUCCIÓN

¿Qué es la Programación Prolog?

A diferencia de los lenguajes tradicionales que se centran en la resolución de problemas paso a paso, la programación **Prolog** se basa en la **lógica y el razonamiento**. Los problemas se enuncian de forma declarativa, y el intérprete se encarga del proceso de solución. Esto lo hace especialmente potente para tareas que implican **razonamiento simbólico, representación del conocimiento e inteligencia artificial**.

Prolog es uno de los lenguajes de programación lógica más populares, desarrollado en la década de 1970. Aunque su uso en la IA convencional ha disminuido, sigue siendo valioso en áreas específicas como los **sistemas expertos** y el **procesamiento del lenguaje natural**. Su objetivo inicial era construir sistemas inteligentes capaces de razonar y resolver problemas como los humanos.

Conceptos Fundamentales de Prolog

La programación en Prolog se basa en cuatro pilares:

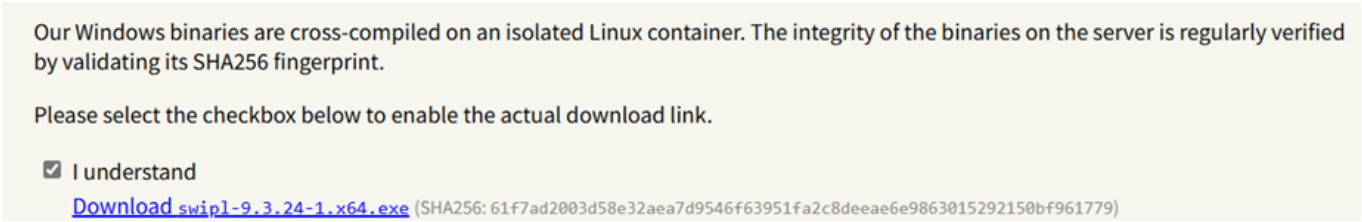
- **Hechos**: Son afirmaciones que se asumen como verdaderas y representan los bloques de construcción básicos del conocimiento. Son similares a oraciones simples en el lenguaje natural.
    - Ejemplo: `Padre(Juan, María).`
- **Reglas**: Expresan relaciones entre los hechos y permiten el razonamiento y la deducción. Constan de una cabeza (la conclusión) y un cuerpo (las condiciones).

    - Ejemplo: `mortal(X) :- humano(X).` (Si X es humano, entonces X es mortal)
- **Variables**: Son marcadores de posición para valores desconocidos en hechos y reglas. Facilitan la unificación, un proceso de coincidencia de variables con valores específicos.
    - Ejemplo: `le_gusta(X, música) :- género(X, Pop).` (Regla: A alguien le gusta la música si su género es Pop)
- **Consultas**: Se utilizan para pedirle al intérprete que resuelva problemas basándose en las reglas y hechos del programa.
    - Ejemplo: `?- Padre(Juan, X).`

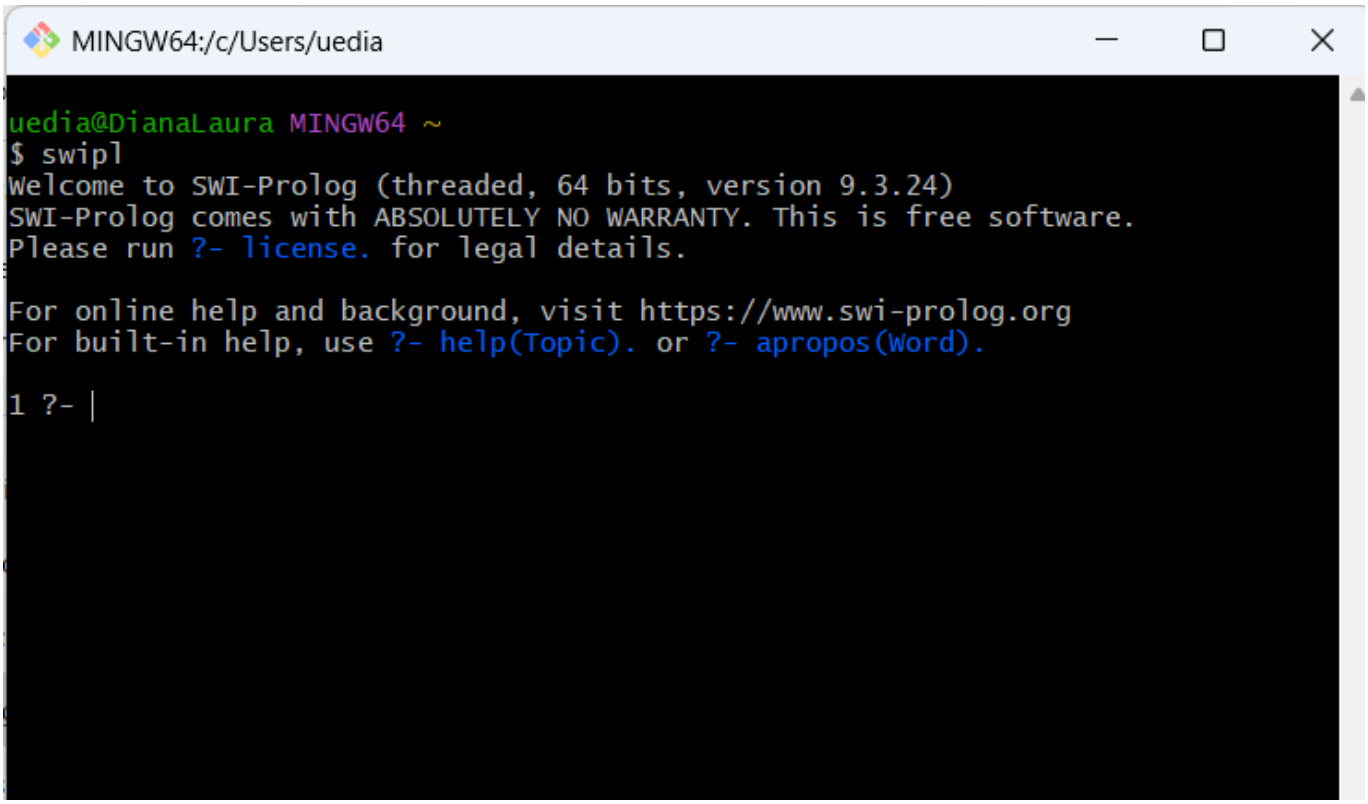
DESARROLLO

Instalación del entorno de desarrollo

Para instalar prolog, lo haremos desde la página de SWI prolog, durante la instalación, marcaremos la opción para designar el PATH en el usuario actual.



Verificamos esto con el comando `swipl` en bash.



Consulta

Lo primero es dar el comando de `trace.`, ya qué nos ayudara a seguir todas las llamadas del programa.

```
uedia@DianaLaura MINGW64 /c/portafolio (main)
$ swipl swipl/operadores.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.3.24)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

1 ?- trace.
true.

[trace] 1 ?-
```

Consultaremos primero, el archivo de los **operadores de datos** (operadores.pl).

```
% nl significa New Line

calc :- X is 100 + 200,write('100 + 200 is '),write(X),nl,
        Y is 400 - 150,write('400 - 150 is '),write(Y),nl,
        Z is 10 * 300,write('10 * 300 is '),write(Z),nl,
        A is 100 / 30,write('100 / 30 is '),write(A),nl,
        B is 100 // 30,write('100 // 30 is '),write(B),nl,
        C is 100 ** 2,write('100 ** 2 is '),write(C),nl,
        D is 100 mod 30,write('100 mod 30 is '),write(D),nl.
```

- `calc.`: Este es el predicado (la función o regla) que el intérprete está a punto de ejecutar.
- `?`: Este signo de interrogación indica que el depurador está esperando una acción tuya. Te está preguntando qué quieres hacer a continuación con la ejecución de calc.
- `creep`: Significa "avanzar paso a paso". Le estás diciendo al depurador que continúe ejecutando el código línea por línea (o subobjetivo por subobjetivo), y que se detenga y te pregunte de nuevo en el siguiente puerto de traza. Es el modo más granular de depuración.

```
[trace] 1 ?- calc.
   Call: (12) calc ? creep
   Call: (13) _16042 is 100+200 ? creep
   Exit: (13) 300 is 100+200 ? creep
   Call: (13) write('100 + 200 is ') ? creep
100 + 200 is
   Exit: (13) write('100 + 200 is ') ? creep
   Call: (13) write(300) ? creep
300
   Exit: (13) write(300) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
   Call: (13) _23098 is 400-150 ? creep
   Exit: (13) 250 is 400-150 ? creep
   Call: (13) write('400 - 150 is ') ? creep
400 - 150 is
   Exit: (13) write('400 - 150 is ') ? creep
   Call: (13) write(250) ? creep
250
   Exit: (13) write(250) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
   Call: (13) _30154 is 10*300 ? creep
   Exit: (13) 3000 is 10*300 ? creep
   Call: (13) write('10 * 300 is ') ? creep
10 * 300 is
   Exit: (13) write('10 * 300 is ') ? creep
   Call: (13) write(3000) ? creep
3000
   Exit: (13) write(3000) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
   Call: (13) _5954 is 100/30 ? creep
   Exit: (13) 3.3333333333333335 is 100/30 ? creep
   Call: (13) write('100 / 30 is ') ? creep
100 / 30 is
   Exit: (13) write('100 / 30 is ') ? creep
   Call: (13) write(3.3333333333333335) ? creep
3.3333333333333335
   Exit: (13) write(3.3333333333333335) ? creep
   Call: (13) nl ? creep
```

```
   Exit: (13) nl ? creep
   Call: (13) _20072 is 100**2 ? creep
   Exit: (13) 10000 is 100**2 ? creep
   Call: (13) write('100 ** 2 is ') ? creep
100 ** 2 is
   Exit: (13) write('100 ** 2 is ') ? creep
   Call: (13) write(10000) ? creep
10000
   Exit: (13) write(10000) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
```

```
    Call: (13) _27128 is 100 mod 30 ? creep
    Exit: (13) 10 is 100 mod 30 ? creep
    Call: (13) write('100 mod 30 is ') ? creep
100 mod 30 is
    Exit: (13) write('100 mod 30 is ') ? creep
    Call: (13) write(10) ? creep
10
    Exit: (13) write(10) ? creep
    Call: (13) nl ? creep

    Exit: (13) nl ? creep
    Exit: (12) calc ? creep
true.
```

Ahora, seguiremos con los **controles de flujos** (option.pl)

```
% If-Then-Else statement
gt(X,Y) :- X >= Y,write('X is greater or equal').
gt(X,Y) :- X < Y,write('X is smaller').
% If-Elif-Else statement
gte(X,Y) :- X > Y,write('X is greater').
gte(X,Y) :- X =:= Y,write('X and Y are same').
gte(X,Y) :- X < Y,write('X is smaller').
```

```
[trace] 1 ?- gt(2,3).
   Call: (12) gt(2, 3) ? creep
   Call: (13) 2>=3 ? creep
   Fail: (13) 2>=3 ? creep
   Redo: (12) gt(2, 3) ? creep
   Call: (13) 2<3 ? creep
   Exit: (13) 2<3 ? creep
   Call: (13) write('X is smaller') ? creep
X is smaller
   Exit: (13) write('X is smaller') ? creep
   Exit: (12) gt(2, 3) ? creep
true.
```

```
[trace] 2 ?- gt(10,9).
   Call: (12) gt(10, 9) ? creep
   Call: (13) 10>=9 ? creep
   Exit: (13) 10>=9 ? creep
   Call: (13) write('X is greater or equal') ? creep
X is greater or equal
   Exit: (13) write('X is greater or equal') ? creep
   Exit: (12) gt(10, 9) ? creep
true .
```

Bucles (loop.pl)

- Contador a 10

```
count_to_10(10) :- write(10),nl.
count_to_10(X) :-
    write(X),nl,
    Y is X + 1,
    count_to_10(Y).
```

```
[trace] 2 ?- count_to_10(8).
   Call: (12) count_to_10(8) ? creep
   Call: (13) write(8) ? creep
8
   Exit: (13) write(8) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
   Call: (13) _5358 is 8+1 ? creep
   Exit: (13) 9 is 8+1 ? creep
   Call: (13) count_to_10(9) ? creep
   Call: (14) write(9) ? creep
9
   Exit: (14) write(9) ? creep
   Call: (14) nl ? creep

   Exit: (14) nl ? creep
   Call: (14) _11532 is 9+1 ? creep
   Exit: (14) 10 is 9+1 ? creep
   Call: (14) count_to_10(10) ? creep
   Call: (15) write(10) ? creep
10
   Exit: (15) write(10) ? creep
   Call: (15) nl ? creep

   Exit: (15) nl ? creep
   Exit: (14) count_to_10(10) ? creep
   Exit: (13) count_to_10(9) ? creep
   Exit: (12) count_to_10(8) ? creep
true .
```

- Rangos

```
count_down(L, H) :-
    between(L, H, Y),
    Z is H - Y,
    write(Z), nl.

count_up(L, H) :-
    between(L, H, Y),
    Z is L + Y,
    write(Z), nl.
```

```
[trace] 5 ?-
count_down(1,5).
   Call: (12) count_down(1, 5) ? creep
   Call: (13) between(1, 5, _1854) ? creep
   Exit: (13) between(1, 5, 1) ? creep
   Call: (13) _3636 is 5-1 ? creep
   Exit: (13) 4 is 5-1 ? creep
   Call: (13) write(4) ? creep
4
   Exit: (13) write(4) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
   Exit: (12) count_down(1, 5) ? creep
true .
```

```
[trace] 7 ?- count_up(3,6).
   Call: (12) count_up(3, 6) ? creep
   Call: (13) between(3, 6, _1848) ? creep
   Exit: (13) between(3, 6, 3) ? creep
   Call: (13) _3630 is 3+3 ? creep
   Exit: (13) 6 is 3+3 ? creep
   Call: (13) write(6) ? creep
6
   Exit: (13) write(6) ? creep
   Call: (13) nl ? creep

   Exit: (13) nl ? creep
   Exit: (12) count_up(3, 6) ? creep
true .

[trace] 8 ?- count_up(6,3).
   Call: (12) count_up(6, 3) ? creep
   Call: (13) between(6, 3, _1848) ? creep
   Fail: (13) between(6, 3, _1848) ? creep
   Fail: (12) count_up(6, 3) ? creep
false.
```

Conjunciones y disyunciones (conj_disj.pl)

```
parent(jhon,bob).
parent(lili,bob).

male(jhon).
female(lili).

% Conjunction Logic
father(X,Y) :- parent(X,Y),male(X).
mother(X,Y) :- parent(X,Y),female(X).

% Disjunction Logic
child_of(X,Y) :- father(X,Y);mother(X,Y).
```

```
[trace] 2 ?- mother(X, bob).
   Call: (12) mother(_238, bob) ? creep
   Call: (13) parent(_238, bob) ? creep
   Exit: (13) parent(jhon, bob) ? creep
   Call: (13) female(jhon) ? creep
   Fail: (13) female(jhon) ? creep
   Redo: (13) parent(_238, bob) ? creep
   Exit: (13) parent(lili, bob) ? creep
   Call: (13) female(lili) ? creep
   Exit: (13) female(lili) ? creep
   Exit: (12) mother(lili, bob) ? creep
X = lili.
```

```
[trace] 4 ?- child_of(lili, X).
   Call: (12) child_of(lili, _240) ? creep
   Call: (13) father(lili, _240) ? creep
   Call: (14) parent(lili, _240) ? creep
   Exit: (14) parent(lili, bob) ? creep
   Call: (14) male(lili) ? creep
   Fail: (14) male(lili) ? creep
   Fail: (13) father(lili, _240) ? creep
   Redo: (12) child_of(lili, _240) ? creep
   Call: (13) mother(lili, _240) ? creep
   Call: (14) parent(lili, _240) ? creep
   Exit: (14) parent(lili, bob) ? creep
   Call: (14) female(lili) ? creep
   Exit: (14) female(lili) ? creep
   Exit: (13) mother(lili, bob) ? creep
   Exit: (12) child_of(lili, bob) ? creep
X = bob.
```

Listas (_basic.pl, _misc.pl, _repost.pl)

- list_basic.pl

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).

list_length([],0).
list_length([_|TAIL],N) :- list_length(TAIL,N1), N is N1 + 1.

list_concat([],L,L).
list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).

list_append(A,T,T) :- list_member(A,T),!.
list_append(A,T,[A|T]).

list_delete(X, [X], []).
list_delete(X,[X|L1], L1).
list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X,L2,L1).

list_insert(X,L,R) :- list_delete(X,R,L).
```

```
[trace] 1 ?- list_member(6, [2,4,6,8]).
   Call: (12) list_member(6, [2, 4, 6, 8]) ? creep
   Call: (13) list_member(6, [4, 6, 8]) ? creep
   Call: (14) list_member(6, [6, 8]) ? creep
   Exit: (14) list_member(6, [6, 8]) ? creep
   Exit: (13) list_member(6, [4, 6, 8]) ? creep
   Exit: (12) list_member(6, [2, 4, 6, 8]) ? creep
true .
```

- list_misc.pl

```
list_even_len([]).
list_even_len([Head|Tail]) :- list_odd_len(Tail).

list_odd_len([_]).
list_odd_len([Head|Tail]) :- list_even_len(Tail).

list_divide([],[],[]).
list_divide([X],[X],[]).
list_divide([X,Y|Tail], [X|List1],[Y|List2]) :- list_divide(Tail,List1,List2).

max_of_two(X,Y,X) :- X >= Y.
max_of_two(X,Y,Y) :- X < Y.

list_max_elem([X],X).
list_max_elem([X,Y|Rest],Max) :- list_max_elem([Y|Rest],MaxRest), max_of_two(X,MaxRest,Max).

list_sum([],0).
list_sum([Head|Tail], Sum) :- list_sum(Tail,SumTemp), Sum is Head + SumTemp.
```

```
[trace] 4 ?- list_even_len([1,2,3,5]).
   Call: (12) list_even_len([1, 2, 3, 5]) ? creep
   Call: (13) list_odd_len([2, 3, 5]) ? creep
   Call: (14) list_even_len([3, 5]) ? creep
   Call: (15) list_odd_len([5]) ? creep
   Exit: (15) list_odd_len([5]) ? creep
   Exit: (14) list_even_len([3, 5]) ? creep
   Exit: (13) list_odd_len([2, 3, 5]) ? creep
   Exit: (12) list_even_len([1, 2, 3, 5]) ? creep
true .
```

```
[trace] 7 ?- max_of_two(10,9, 10).
   Call: (12) max_of_two(10, 9, 10) ? creep
   Call: (13) 10>=9 ? creep
   Exit: (13) 10>=9 ? creep
   Exit: (12) max_of_two(10, 9, 10) ? creep
true .
```

Referencias

United States Artificial Intelligence Institute (USAII®). (s. f.). What is Prolog Programming Language: An Overview. https://www.usaii.org/ai-insights/what-is-prolog-programming-language-an-overview.

Dirección del repositorio:

https://github.com/roixarturo/portafolio1

Dirección de la página de GitHub:

https://roixarturo.github.io/portafolio1/