Introducción Prolog

Flavio E. Spetale

¿Que es Prolog?

Prolog = PROgramming in LOGic.

Es un lengauje declarativo basado en ***Reglas*** y ***Hechos*** de lógica cuya información es retribuida en forma de consultas.

Originado en Europa a principios de los 70's por Alain Colmeraur.

Ejemplo

Hechos

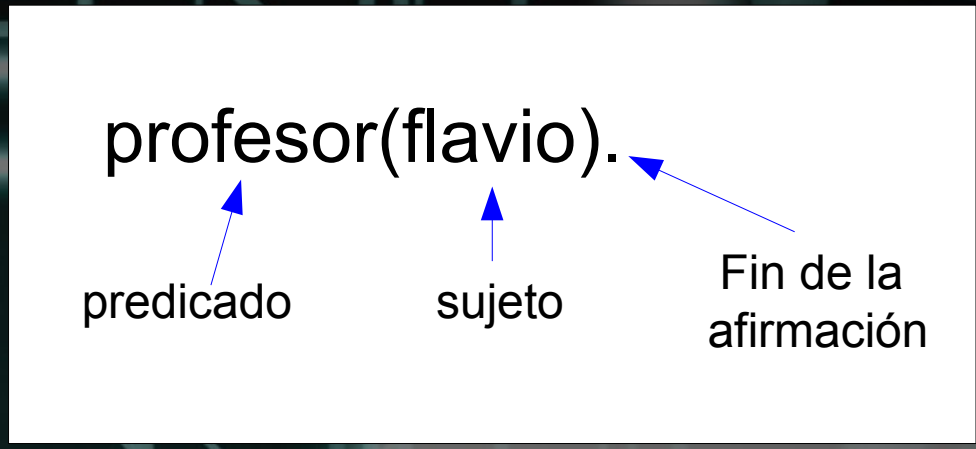
flavio es un profesor

gabriel es un programador

En Prolog

profesor(flavio).

programador(gabriel).



Prolog

Sintaxis

* Las variables se escriben en mayúscula
* Las constantes se escriben en minúscula
* Las afirmaciones terminan con punto (.)
* Los comentarios empiezan con %
* No se pueden dejar espacios entre los nombres de las constantes, se debe utilizar _

Operadores

* Conjunción ,
* Disjunción ;
* Regla o Condición :-
* Fin de la condición .

Ejemplo

Base de conocimiento

* Regla 1: Si está contento entonces escucha música
* Regla 2: Si tiene radio entonces escucha música
* Regla 3: Si escucha música y tiene una guitarra entonces toca la guitarra
* Hecho 1: Tiene una guitarra
* Hecho 2: Está contento

Consulta

* ¿Está tocando la guitarra Lucio?

Ejemplo

Programa

```
escucha_musica :- esta_contento.

escucha_musica :- tiene_radio.

toca_la_guitarra :- escucha_musica, tiene_guitarra.


tiene_guitarra.

esta_contento.
```

Programación

```
?- pwd.
/home/flavio
true.

?- [prueba].
% prueba compiled 0.00 sec, 6 clauses
true.

?- toca_la_guitarra.
true .
```

Razonamiento

Modus ponens:

Si el cuerpo de una regla se puede deducir de la base de conocimiento entonces la cabeza también se deduce de ella.

Deducción

Recorre la base de conocimiento buscando un hecho o la cabeza de una regla que coincida con la consulta

* Si coincide con un hecho entonces la consulta es cierta
* Si coincide con la cabeza de una regla y el cuerpo se deduce de la base de conocimiento entonces la consulta es cierta

Ejemplo

Base de conocimiento

* Regla 1: Si Santiago está contento entonces toca la guitarra
* Regla 2: Si Santiago escucha música entonces toca la guitarra
* Regla 3: Si Lucio está contento y escucha música entonces toca la guitarra
* Hecho 1: Lucio está contento
* Hecho 2: Santiago escucha música

Consulta

* ¿Está tocando la guitarra Lucio?
* ¿Quíen está tocando la guitarra?

Ejemplo

Programa

* toca_la_guitarra(santiago) :- esta_contento(santiago). % Regla 1
* toca_la_guitarra(santiago) :- escucha_musica(santiago). % Regla 2
* toca_la_guitarra(lucio) :- esta_contento(lucio) , escucha_musica(lucio). % Regla 3

* esta_contento(lucio). % Hecho 1
* escucha_musica(santiago). % Hecho 2

Programación

```
?- [prueba].
% prueba compiled 0.00 sec, 6 clauses
true.

?- toca_la_guitarra(lucio).
false.

?- toca_la_guitarra(X).
X = santiago ;
false.
```

SLD-Resolución

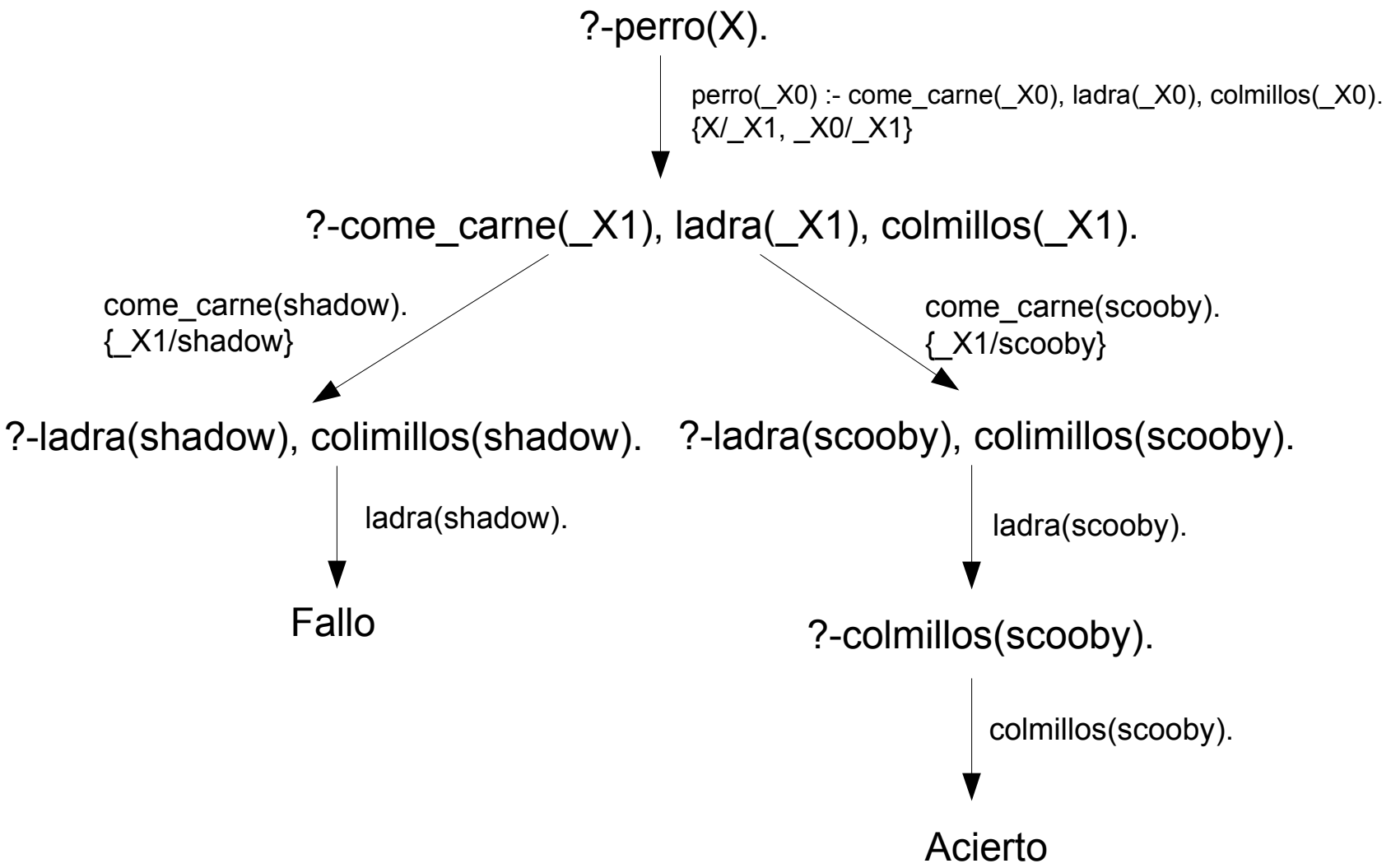
Base de conocimiento

* Regla 1: Si tiene colmillos, ladra y come carne entonces es un perro (perro(X) :- colmillos(X), ladra(X), come_carne(X).)
* Hecho 1: colmillos(shadow).
* Hecho 2: colmillos(scooby).
* Hecho 3: come_carne(shadow).
* Hecho 4: come_carne(scooby).
* Hecho 5: ladra(scooby).

Consulta

* ¿Quién es un perro? (perro(X).)

SLD-Resolución



SLD-Resolución trace

```
?- trace.
true.

[trace]  ?- perro(X).
    Call: (6) perro(_G337) ? creep
    Call: (7) come_carne(_G337) ? creep
    Exit: (7) come_carne(shadow) ? creep
    Call: (7) ladra(shadow) ? creep
    Fail: (7) ladra(shadow) ? creep
    Redo: (7) come_carne(_G337) ? creep
    Exit: (7) come_carne(scooby) ? creep
    Call: (7) ladra(scooby) ? creep
    Exit: (7) ladra(scooby) ? creep
    Call: (7) colmillos(scooby) ? creep
    Exit: (7) colmillos(scooby) ? creep
    Exit: (6) perro(scooby) ? creep
X = scooby.
```

Recursión

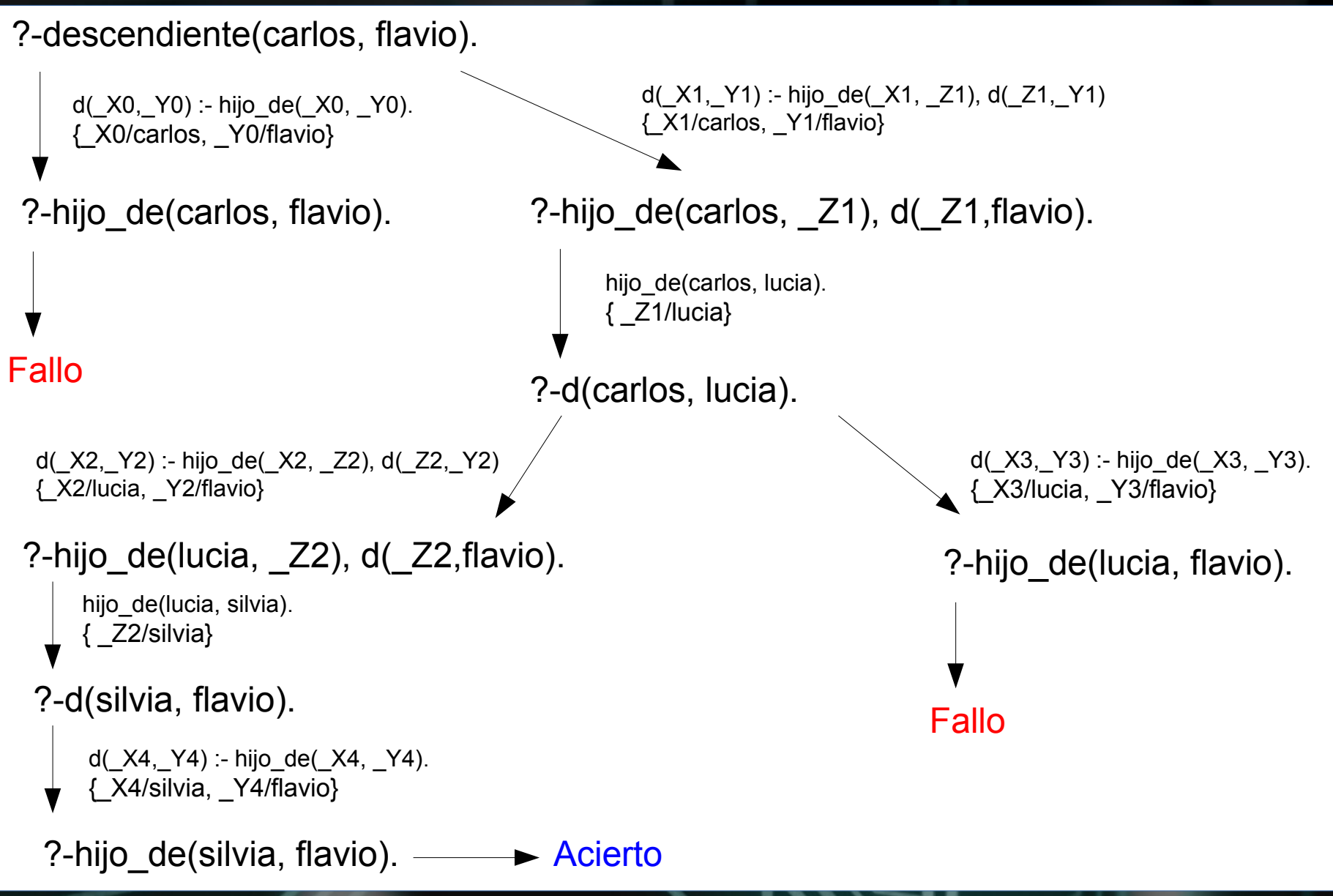
Base de conocimiento

* Regla 1: descendiente(X, Y) :- hijo_de(X, Y).
* Regla 2: descendiente(X, Y) :- hijo_de(X, Z), descendiente(Z, Y).
* Hecho 1: hijo_de(carlos, lucia).
* Hecho 2: hijo_de(lucia, silvia).
* Hecho 3: hijo_de(silvia, flavio).

Consulta

* descendiente(carlos, flavio).

Recursión


?-descendiente(carlos, flavio).
d(_X0,_Y0) :- hijo_de(_X0, _Y0).
{_X0/carlos, _Y0/flavio}
d(_X1,_Y1) :- hijo_de(_X1, _Z1), d(_Z1,_Y1)
{_X1/carlos, _Y1/flavio}
?-hijo_de(carlos, flavio).
?-hijo_de(carlos, _Z1), d(_Z1,flavio).
Fallo
hijo_de(carlos, lucia).
{ _Z1/lucia}
?-d(carlos, lucia).
d(_X2,_Y2) :- hijo_de(_X2, _Z2), d(_Z2,_Y2)
{_X2/lucia, _Y2/flavio}
d(_X3,_Y3) :- hijo_de(_X3, _Y3).
{_X3/lucia, _Y3/flavio}
?-hijo_de(lucia, _Z2), d(_Z2,flavio).
?-hijo_de(lucia, flavio).
hijo_de(lucia, silvia).
{ _Z2/silvia}
?-d(silvia, flavio).
Fallo
d(_X4,_Y4) :- hijo_de(_X4, _Y4).
{_X4/silvia, _Y4/flavio}
?-hijo_de(silvia, flavio).
Acierto

Recursión

Base de conocimiento

* Regla 1: descendiente(X, Y) :- hijo_de(X, Z), descendiente(Z, Y).
* Regla 2: descendiente(X, Y) :- hijo_de(X, Y).
* Hecho 1: hijo_de(carlos, lucia).
* Hecho 2: hijo_de(lucia, silvia).
* Hecho 3: hijo_de(silvia, flavio).

Consulta

```
?- descendiente(carlos, flavio).
ERROR: Out of local stack
   Exception: (1,675,218) descendiente(_G2330, flavio) ? abort
% Execution Aborted
```

Aritmética

OPERADORES

Operador	Descripción
<	Menor
>	Mayor
=<	Menor que
>=	Mayor que
=\=	Diferente
is	Evalúa si un numero equivale a una expresión
=:=	Igual
mod	Modulo

FUNCIONES

Función	Descripción
Abs	Valor absoluto
sign	Signo de un numero
Min	Valor minimo
Max	Valor maximo
Random	Numero aleatorio.
round	Redondeo
Floor	Redondeo hacia arria
ceiling	Redondeo hacia abajo
sqrt	Raiz
powm	Potencia
pi	Valor de pi

Estructura básica - Listas

* En prolog, una lista es una representación de un conjunto de elementos.

[manzana, pera, banana] lista vacia: [ ]

*Se pueden utilizar cómo elementos de la lista cualquier tipo de dato de prolog, incluyendo listas:

[[a,b,c],[d, e, f]]

*También estructuras prolog:

[vehiculo(ale, [bici, moto, auto]), vehiculo(ariel,[bici, auto, helicoptero])]

Listas

En su forma más básica, una lista se puede ver como un predicado que tiene 2 partes: lista(cabeza, cola)

Prolog: [Cabeza | Cola]

Ejemplos

[a|T]: Lista con “a” en la cabeza y el resto en la variable T (cola)

[a, b| T]: Lista con los elementos “a” y “b” en la cabeza y el resto en la variable T (cola)

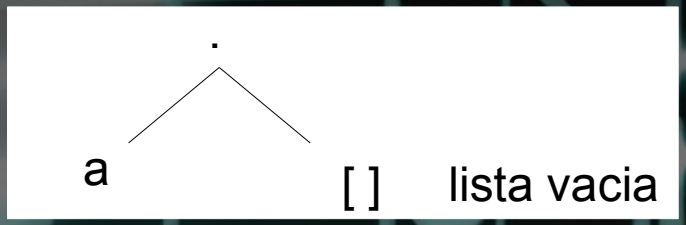
[X| T]: Lista con el primer elemento instanciado en la variable X y el resto en la variable T (cola)

[X, Y| T]: Lista con dos elementoe instanciados por las variables X e Y y el resto en la variable T (cola)

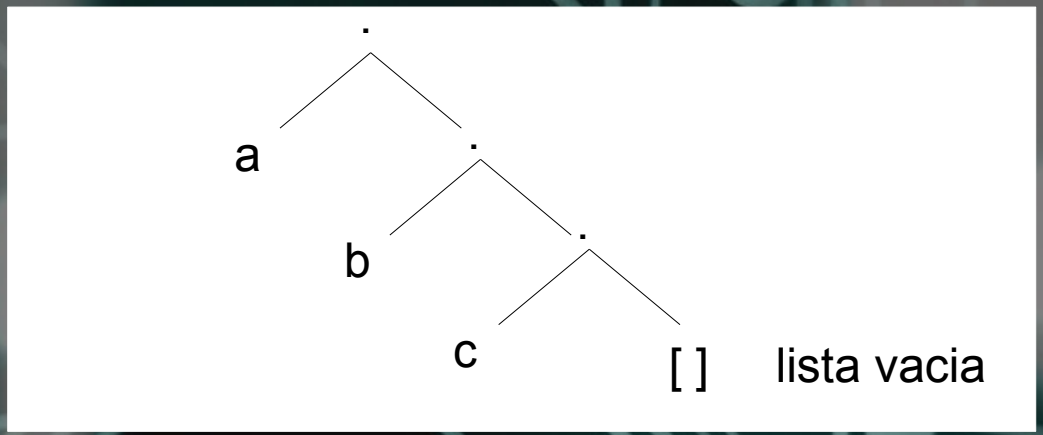
Listas

Ejemplos

Lista con un único elemento “a” . En Prolog ,(a,[ ])



Lista con tres átomos “a, b, c”. En Prolog .(a, . (b, . (c, [ ])))



Ejemplo

Base de conocimiento

ficha(identidad (primer_nombre, segundo_nombre, apellido),
fecha_nacimiento(dia, mes, año),
direccion(calle(nombre, numero), ciudad(nombre)),
datos(dni, tel, titulo, profesion)).

Consulta

```
?- ficha(identidad(Nombre, Seg_Nombre, Apellido),_, direccion(_,ciudad('rosario'
)), _).
Nombre = 'Flavio',
Seg_Nombre = 'Ezequiel',
Apellido = 'Spetale' ;
Nombre = 'Santiago',
Seg_Nombre = 'Wilfredo',
Apellido = 'Rodriguez'.
```

Ejemplo

A efectos de eficacia, resulta conveniente simplificar la estructura de representación

Base de conocimiento

ficha(id, identidad (primer_nombre, segundo_nombre, apellido)).
nacimiento(id, fecha(dia, mes, año)).
direccion(id, calle(nombre, numero), ciudad(nombre)).
datos(id, dni, tel, titulo, profesion).

Consulta

```
?- ficha(X, identidad(Primer_nombre, Segundo_nombre, Apellido)), direccion(X, _,
ciudad('rosario')), datos(X, _, _, 'doctor', _).
X = 1,
Primer_nombre = 'Flavio',
Segundo_nombre = 'Ezequiel',
Apellido = 'Spetale' ;
false.
```

Operaciones con Listas

Base de conocimiento

fruto([manzana, pera, naranja, uva]).

El predicado *member* devuelve los elementos de una lista

member(Elemento, Lista).

```
?- member(pera, [manzana, pera, naranja, uva]).
true ;
false.

?- member(X, [manzana, pera, naranja, uva]).
X = manzana ;
X = pera ;
X = naranja ;
X = uva.
```

El predicado *append* concatena las dos primeras lista en la tercera

append(Lista1, Lista2, Lista3).

Operaciones con Listas

El predicado *append* concatena las dos primeras lista en la tercera

append(Lista1, Lista2, Lista3).

```
?- append([manzana, pera], [naranja, uva, banana], Fruto).
Fruto = [manzana, pera, naranja, uva, banana].

?- append([manzana, pera], banana, Fruto).
Fruto = [manzana, pera|banana].

?- append([manzana, pera], [banana], Fruto).
Fruto = [manzana, pera, banana].
```

Operaciones con Listas

El predicado *length* encuentra la longitud de una lista

length(Lista, Longitud).

```
?- length([a,b,95,[1,1,1,1]],X).
X = 4.

?- length([a,XYZ,59,1,1,1,1],X).
X = 7.
```

El predicado *reverse* devuelve la lista inversa de una dada.

reverse(Lista, Lista_Res).

```
?- reverse([a,b,c,e,'flavio'], T).
T = [flavio, e, c, b, a].
```

Operaciones con Listas

El predicado *eliminar* elimina todas las ocurrencias de un elemento en una lista simple (sin estructura).

eliminar(Elemento, Lista, Lista_Res).

Definición:

```
eliminar( _ , [ ], [ ]).
eliminar( X, [X | Y], R) :- eliminar( X, Y, R).
eliminar( X, [W| Y], [W | R]) :- X \== W, eliminar( X, Y, R).
```

```
?- eliminar( i, [e,l,i,m,i,n,a], R).
R = [e, l, m, n, a] ;
false.
```

Operaciones con Listas

El predicado *sustituir* todas las ocurrencias de un elemento (1er arg.) por el 2do argumento en una lista simple (sin estructura)

sustituir(Elemento1, Elemento2, Lista, Lista_Res).

Definición:

```
sustituir( _ , _ , [ ], [ ]).
sustituir( X, Y, [X | U], [Y | V]) :- sustituir( X, Y, U, V).
sustituir( X, Y, [Z | U], [Z | V]) :- X \== Z, sustituir( X, Y, U, V).
```

```
?- sustituir( w, k, [e,l,w,m,w,n,a], T).
T = [e, l, k, m, k, n, a] ;
false.
```

El predicado cut !

El corte es un predicado predefinido que no recibe argumentos. Se representa mediante un signo de admiración (!). El corte tiene la propiedad de eliminar los puntos de elección del predicado que lo contiene.

Es decir, cuando se ejecuta el corte, el resultado del objetivo (no sólo la cláusula en cuestión) queda comprometido al éxito o fallo de los objetivos que aparecen a continuación.

Otra forma de ver el efecto del corte es pensar que solamente tiene la propiedad de detener el backtracking cuando éste se produce.

El predicado cut !

Los motivos por los que se usa el corte son:

Para optimizar la ejecución. El corte sirve para evitar que por culpa del backtracking se exploren puntos de elección que no llevan a otra solución (fallan). Esto es podar el árbol de búsqueda de posibles soluciones.

Para implementar algoritmos diferentes según la combinación de argumentos de entrada.

Para conseguir que un predicado solamente tenga una solución. Una vez que el programa encuentra una solución ejecutamos un corte. Así evitamos que Prolog busque otras soluciones aunque sabemos que éstas existen.

El predicado cut !

Resolución (control de ejecución)

* Si a lo largo de la resolución no se alcanza el lugar de corte, se procede de forma normal.

* Si se alcanza el corte, se marca el predicado que define la regla, las condiciones de la regla anteriores al corte y la sustitución obtenida hasta el momento. A lo largop del resto de la resolución no se permite una nueva resolución de de dicho predicado, ni las condiciones, ni las sustituciones obtenidas.

Ejemplo

Base de conocimiento

```
s1(X,Y) :- q1(X, Y).
q1(X, Y) :- i(X), j(Y).
i(6). i(2).
j(1). j(3).
```

Consulta

```
?- s1(X, Y).
X = 6,
Y = 1 ;
X = 6,
Y = 3 ;
X = 2,
Y = 1 ;
X = 2,
Y = 3.
```

Ejemplo

Base de conocimiento

```
s1(X,Y) :- q1(X, Y).
q1(X, Y) :- i(X), !,j(Y).
i(6). i(2).
j(1). j(3).
```

Consulta

```
?- s1(X, Y).
X = 6,
Y = 1 ;
X = 6,
Y = 3.
```

Ejemplo

Base de conocimiento

```
s1(X,Y) :- q1(X, Y).
q1(X, Y) :- i(X),j(Y),!.
i(6). i(2).
j(1). j(3).
```

Consulta

```
?- s1(X, Y).
X = 6,
Y = 1.
```

El predicado fail

Es muy habitual encontrar la secuencia de objetivos corte-fallo: !,fail. El predicado fail/0 es un predicado predefinido que siempre falla.

Se utiliza para detectar prematuramente combinaciones de los argumentos que no llevan a solución, evitando la ejecución de un montón de código que al final va a fallar de todas formas.

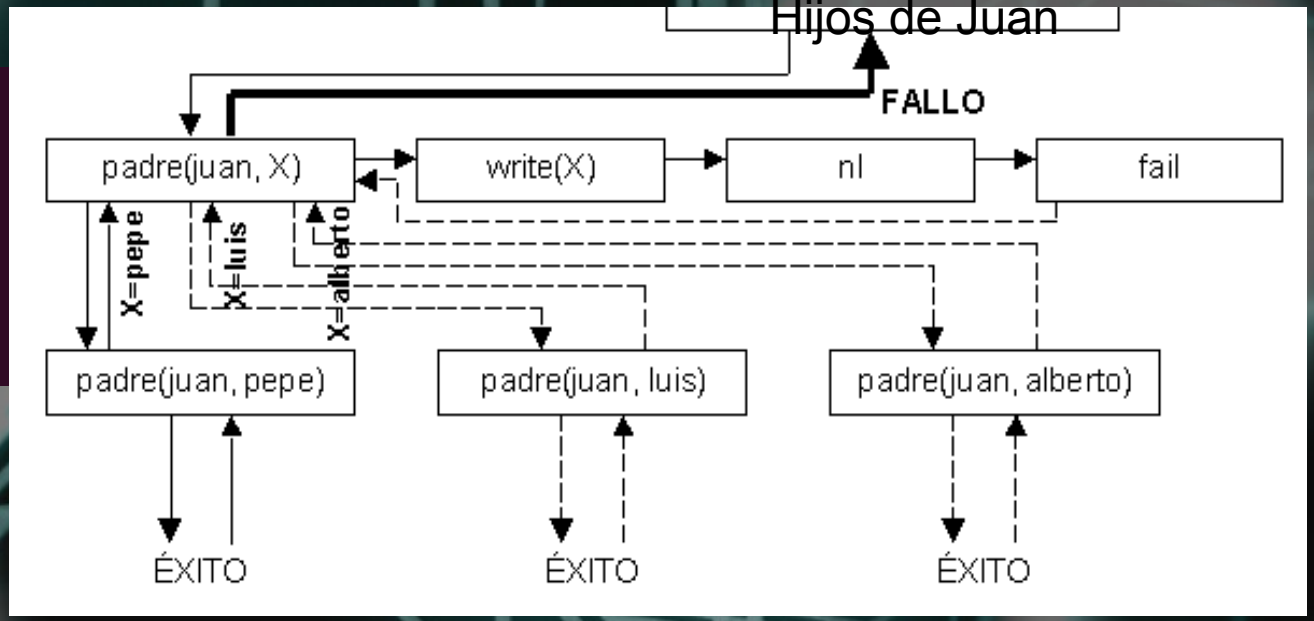
Ejemplo

Base de conocimiento

```
padre(juan, pepe).
padre(juan, luis).
padre(juan, alberto).
hijos_de_Juan:-padre(juan,X), write(X), nl, fail.
```

Consulta

```
?- hijos_de_Juan.
pepe
luis
alberto
false.
```



Negación not

Evalúa como falso cualquier cosa que Prolog sea incapaz de verificar que su predicado argumento es cierto.

La negacion not no corresponde a una negación lógica, sino al hecho de que no hay evidencia para demostrar lo contrario.

Definición:

\+ (P) :- P, !, fail.

Negación not

Base de conocimiento

```
saldo_cuenta(maria,1000).
saldo_cuenta(flora,3000000).
saldo_cuenta(antonio,2000000).
millonario(X) :- saldo_cuenta(X, Y), Y > 1000000.
pobre(X) :- \+ millonario(X).
```

Consulta

```
[trace]  ?- pobre(X).
   Call: (6) pobre(_G4895) ? creep
   Call: (7) millonario(_G4895) ? creep
   Call: (8) saldo_cuenta(_G4895, _G4970) ? creep
   Exit: (8) saldo_cuenta(maria, 1000) ? creep
   Call: (8) 1000>1000000 ? creep
   Fail: (8) 1000>1000000 ? creep
   Redo: (8) saldo_cuenta(_G4895, _G4970) ? creep
   Exit: (8) saldo_cuenta(flora, 3000000) ? creep
   Call: (8) 3000000>1000000 ? creep
   Exit: (8) 3000000>1000000 ? creep
   Exit: (7) millonario(flora) ? creep
   Fail: (6) pobre(_G4895) ? creep
false.
```

Negación not

Consulta

```
[trace]  ?- pobre(maria).
   Call: (6) pobre(maria) ? creep
   Call: (7) millonario(maria) ? creep
   Call: (8) saldo_cuenta(maria, _G4958) ? creep
   Exit: (8) saldo_cuenta(maria, 1000) ? creep
   Call: (8) 1000>1000000 ? creep
   Fail: (8) 1000>1000000 ? creep
   Fail: (7) millonario(maria) ? creep
   Redo: (6) pobre(maria) ? creep
   Exit: (6) pobre(maria) ? creep
true.
```

Negación not

Problemas:

* Una consulta con una variable no instanciada se satisface si hay al menos una asignación de valores a la variable que la cumpla.:

* Al usar la negación, esa consulta pasa a ser cierta si el argumento de la negación fue falso.

Manipulación de la Base de datos y Archivos

Base de conocimiento

```
:- dynamic capital_of/2.

capital_of(suiza, berna).
capital_of(argentina, buenos_aires).
capital_of(estados_unidos, washington).
capital_of(italia, roma).
capital_of(francia, paris).
capital_of(alemania, berlin).
```

El predicado *dynamic* permite modificar la base de conocimiento.

Manipulación de la Base de datos y Archivos

El predicado *assert* permite incluir más cláusulas la base de conocimiento.

assert(predicado).

```
?- capital_of(hawaii, X).
false.

?- assert(capital_of(hawaii, honolulu)).
true.

?- capital_of(hawaii, X).
X = honolulu.
```

Manipulación de la Base de datos y Archivos

El predicado retract permite eliminar más cláusulas la base de conocimiento.

retract(predicado).

```
?- retract(capital_of(hawaii, honolulu)).
true.

?- capital_of(hawaii, X).
false.
```

Manipulación de la Base de datos y Archivos

El predicado *tell abre un archivo y lo establece como fuente de los comandos de escritura. Para cerrarlo se utiliza told.*

tell(nombre del archivo).

```
?- tell('mi_archivo'), write('fiorela.'), nl, write('flavio.'), nl, told.
true.

?- tell('mi_archivo'), write('hermana(fiorela,flavio).'), nl, write('flavio.'),
nl, told.
true.
```

Manipulación de la Base de datos y Archivos

El predicado *see abre un archivo y lo establece como fuente de los comandos de lectura. Para cerrarlo se utiliza seen.*

see(nombre del archivo).

```
?- see('mi_archivo'), read(X), read(Y), seen.
X = fiorela,
Y = flavio.
```

Manipulación de la Base de datos y Archivos

El predicado *consult* añade las cláusulas existentes en el fichero de nombre a la base de conocimientos de Prolog, al final de las ya existentes, sin destruir las anteriores.

consult(nombre del archivo).

```
?- consult('mi_archivo').
% mi_archivo compiled 0.00 sec, 1 clauses
true.

?- hermana(fiorela, X).
X = flavio.
```