# Introducción a Prolog

Flavio E. Spetale

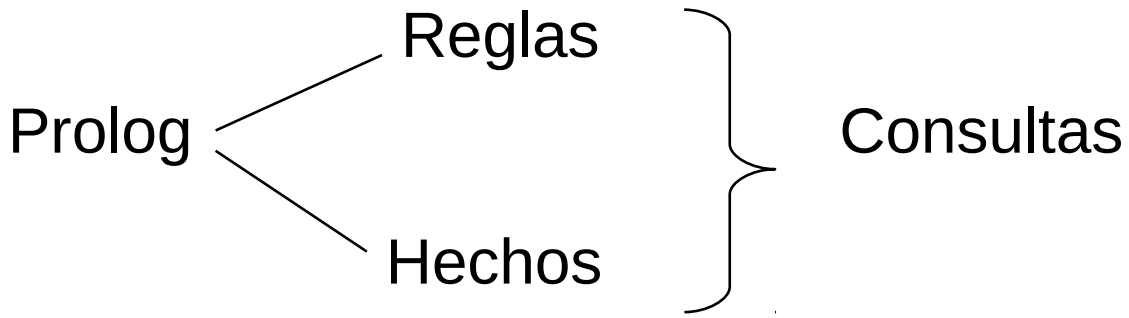
2019

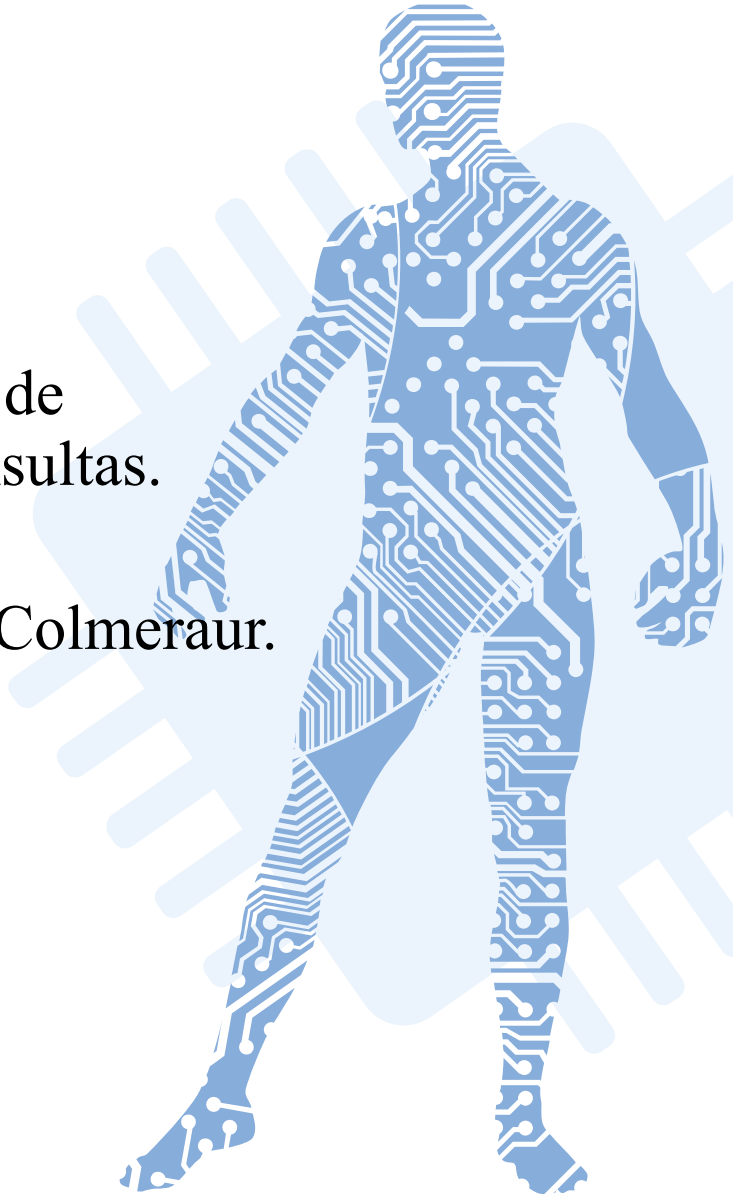
# ¿Qué es Prolog?

Prolog = PROgramming in LOGic.

Es un lenguaje 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.



# Prolog

Se basa en mecanismos básicos:

* Unificación
* Estructuras de datos basadas en árboles
* Backtracking automático

La sintaxis del lenguaje consiste en:

* Declarar hechos sobre objetos y sus relaciones
* Hacer preguntas sobre objetos y sus relaciones
* Definir reglas sobre objetos y sus relaciones

# 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 .

# Sintaxis

Los objetos simples pueden ser constantes o variables. Las constantes serán átomos o números.

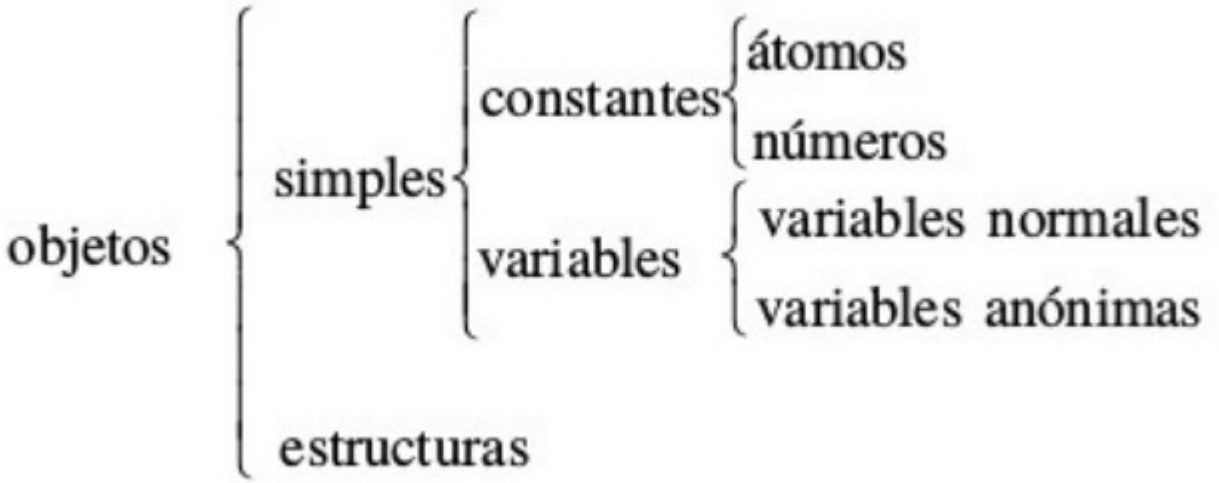
Los átomos empiezan con letra minúscula (nunca con números), pueden contener caracteres especiales y pueden ser nombres entre comillas simples.

Los números serán enteros o reales, sin una definición explícita de tipos.

Las variables empiezan con mayúscula o con subrayado.

Las variables anónimas son aquellas cuyo nombre es sólo el carácter subrayado (_). Se usan cuando no es importante el nombre de la variable o cuando la variable no puede unificar con otra, dentro de la misma cláusula.

# Sintaxis



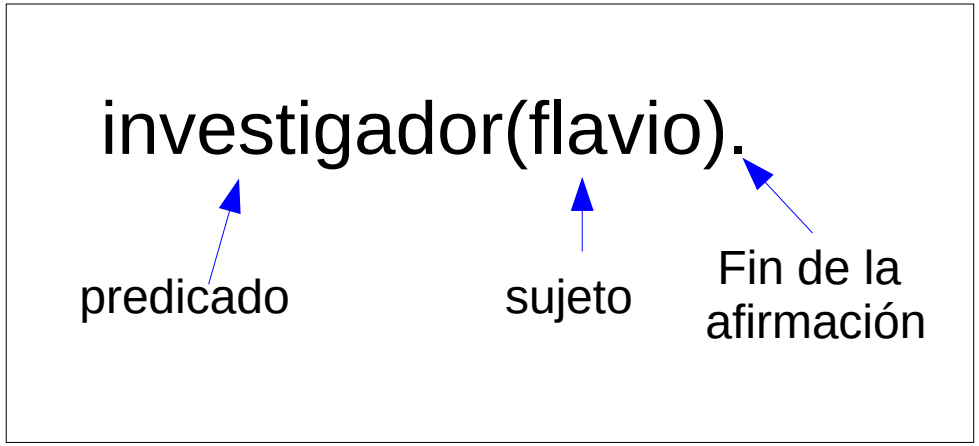

Nota: El alcance de una variable es la cláusula donde aparece, y el alcance de una constante es todo el programa.

# Hechos

**En la lengua natural**

Flavio es un investigador.

Gabriel es un programador.




**En Prolog**

investigador(flavio).

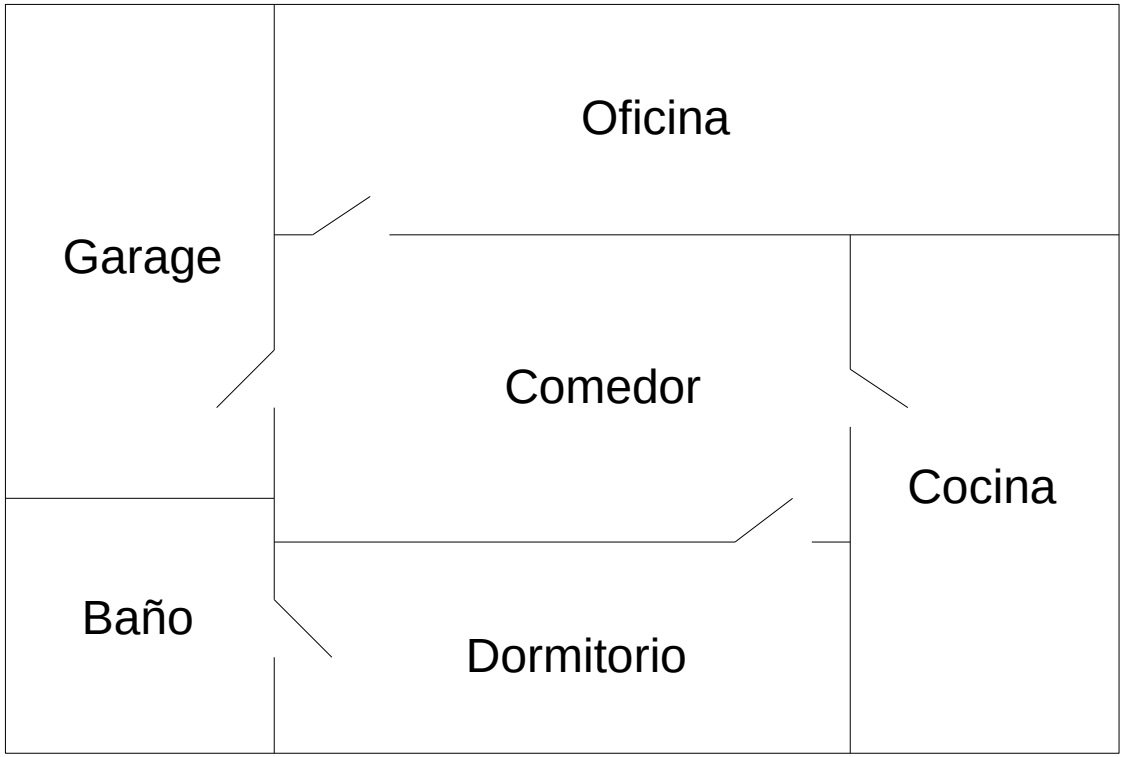
programador(gabriel).

**predicado ($arg_1$, $arg_2$, … , $arg_N$).**

# Juego Búsqueda

Desarrollaremos un juego de búsqueda definiendo los hechos básicos. Éstos incluyen:

- Habitaciones y sus conexiones
- Objetos y su lugar dentro de la casa
- Propiedades de varias cosas
- Donde el jugador comienza el juego



# Prolog

Definimos las habitaciones con el predicado habitacion/1 en base al layout de la casa.

```
habitacion(oficina).
habitacion(baño).
habitacion(comedor).
habitacion(dormitorio).
habitacion(garage).
habitacion(cocina).
```

Definimos las ubicaciones de los objetos con predicado ubicacion de dos argumentos/2. El primer argumento significará el objeto y el segundo significará su ubicación.

```
ubicacion(escritorio, oficina).
ubicacion(manzana, cocina).
ubicacion(luz_escritorio, escritorio).
ubicacion(lavadora, cocina).
ubicacion(control_remoto, ‘tv smart’).
ubicacion('tv smart', comedor).
ubicacion(cereales, cocina).
ubicacion(computadora, oficina).
```

# Prolog

**¿Los siguientes hechos son iguales?**

```
ubicacion(bacha_cocina, cocina).
ubicacion(cocina, bacha_cocina).
```

**NO, son diferentes.** Por lo tanto, mientras seamos coherentes en nuestro uso de argumentos, podemos representar con precisión nuestro significado y evitar la posible interpretación ambigua de que la cocina está en la bacha.

# Prolog

Definimos las puertas que unen dos habitaciones de la casa con predicado puerta de dos argumentos/2.

```
puerta(comedor, oficina).
puerta(comedor, garage).
puerta(comedor, cocina).
puerta(dormitorio, comedor).
puerta(baño, dormitorio).
```

Definimos propiedades de los objetos con el predicado comida/1 y la posiscion donde se encuentra el jugador.

```
comida(manzana).
comida(cereales).
here(oficina).
```

# Consultas

Las consultas de Prolog funcionan por coincidencia de patrones.

El patrón de consultas se llama **objetivo/meta (goal)**.

Si hay un hecho que coincide con el objetivo, entonces la consulta es exitosa y el intérprete responde con un **sí (yes)**.

Si no hay un hecho coincidente, entonces la consulta falla y el intérprete responde con **no (false)**.

PROLOG busca automáticamente en la base de conocmiento si existe un hecho que se puede

**unificar** (es decir, tiene el mismo nombre de predicado, el mismo número de argumentos y cada uno

de los argumentos tiene el mismo nombre, uno a uno) con el hecho que aparece en la consulta.

# Prolog

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

?- [intro_Prolog2].
true.

?- [Intro_Prolog2].
ERROR: Arguments are not sufficiently instantiated
...
ERROR: Re-run your program in debug mode (:- debug.) to get more detail.

?- ubicacion(escritorio, oficina).
true.

?- ubicacion(escritorio, dormitorio).
false.
```

La respuesta “**FALSE**” no implica que el hecho sea falso sino que no se puede probar (en general) que sea verdadero con el conocimiento almacenado en la base de conocimiento.

# Consultas

Las metas se pueden generalizar mediante el uso de variables Prolog.

Estas variable no se comportan como las variables en otros lenguajes, y se denominan variables lógicas (aunque Prolog no se corresponde precisamente con la lógica).

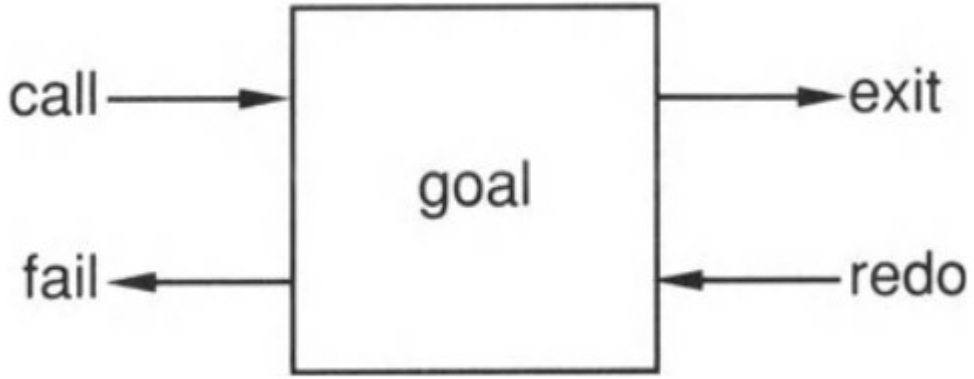
Las variables lógicas reemplazan uno o más de los argumentos en el objetivo.

```
?- habitacion(X).
X = oficina ;
X = baño ;
X = comedor ;
X = dormitorio ;
X = garage.

?- ubicacion(Objeto,cocina).
Objeto = manzana ;
Objeto = lavadora ;
Objeto = cereales.
```

# Consultas

Una meta tiene cuatro puertos que representan el flujo de control a través del objetivo:
Call, Exit, Redo y Fail.




Call: Comienza la búsqueda de cláusulas que unifican con el objetivo.

Exit:Indica que el objetivo es satisfecho, establece un marcador de lugar en la cláusula y une las variables apropiadamente.

Redo: Reintenta el objetivo, desenlaza las variables y reanuda la búsqueda en el marcador de lugar.

Fail: Indica que no hay más cláusulas que coincidan con el objetivo.

# Consultas

Objetivos simples se pueden combinar para formar consultas compuestas.

?- ubicacion(X,cocina), comida(X).
X = manzana ;
X = cereales.

```
[trace]  ?- ubicacion(X,cocina), comida(X).
    Call: (9) ubicacion(_11566, cocina) ? creep
    Exit: (9) ubicacion(manzana, cocina) ? creep
    Call: (9) comida(manzana) ? creep
    Exit: (9) comida(manzana) ? creep
X = manzana ;
    Redo: (9) ubicacion(_11566, cocina) ? creep
    Exit: (9) ubicacion(lavadora, cocina) ? creep
    Call: (9) comida(lavadora) ? creep
    Fail: (9) comida(lavadora) ? creep
    Redo: (9) ubicacion(_11566, cocina) ? creep
    Exit: (9) ubicacion(cereales, cocina) ? creep
    Call: (9) comida(cereales) ? creep
    Exit: (9) comida(cereales) ? creep
X = cereales.
```

# Consultas

Objetivos simples se pueden combinar para formar consultas compuestas.

```
?- ubicacion(manzana,X),habitacion(X).
X = cocina.

?- habitacion(X), ubicacion(manzana,X).
X = cocina ;
false.

?- habitacion(Lugar), ubicacion(Objeto, Lugar).
Lugar = oficina,
Objeto = escritorio ;
Lugar = oficina,
Objeto = computadora ;
Lugar = comedor,
Objeto = 'tv smart' ;
Lugar = cocina,
Objeto = manzana ;
Lugar = cocina,
Objeto = lavadora ;
Lugar = cocina,
Objeto = cereales.
```

# Predicados intrínsecos

Un predicado intrínseco está predefinido por Prolog.

No hay cláusulas en la base de conocimieto para predicados intrínsecos.

Cuando el intérprete encuentra un objetivo que coincide con un predicado intrínseco, se trata de un procedimiento predefinido.

Pueden realizar funciones que no tienen nada que ver con la prueba del teorema lógico, como escribir en la consola. Por esta razón, a veces se les llama predicados extra-lógicos.

write/1: Este predicado siempre tiene éxito cuando se le llama, y tiene el efecto secundario de escribir su argumento en la consola. Siempre falla en Backtracking. Backtracking no deshace el efecto secundario.

nl/0: Tiene éxito, y comienza una nueva línea. Al igual que escribir, siempre tiene éxito cuando se le llama, y falla en Backtracking.

tab/1: Su argumento es un entero que tabula la cantidad de espacios. Tiene éxito cuando se le llama y falla en Backtracking.

# Predicados intrínsecos

read/1: es un predicado que lee un término desde el teclado y lo instancia en una variable.

fail/0: es un predicado predefinido que siempre falla.

```
?- ubicacion(X, cocina),write(X),nl,fail.
manzana
lavadora
cereales
false.
```

```
[trace]  ?- ubicacion(X, comedor),write(X),nl,fail.
   Call: (9) ubicacion(_9212, comedor) ? creep
   Exit: (9) ubicacion('tv smart', comedor) ? creep
   Call: (9) write('tv smart') ? creep
tv smart
   Exit: (9) write('tv smart') ? creep
   Call: (9) nl ? creep

   Exit: (9) nl ? creep
   Call: (9) fail ? creep
   Fail: (9) fail ? creep
false.
```

# Predicados intrínsecos

El final **false** significa que la consulta falló debido al  fail/0.

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.

# Reglas

Un predicado se define por cláusulas, que pueden ser hechos o reglas.

Una regla no es más que una consulta almacenada.

**Cabecera :- Cuerpo**

**Cabecera**: una definición de predicado (como un hecho)

**:-** El símbolo del cuello, a veces leído como "si"

**Cuerpo**: uno o más objetivos (una consulta)

# Formato de Reglas

La cabeza de la regla sólo puede tener un único predicado.

Por lo tanto, la regla $\forall x\ (p(x) \rightarrow (q(x) \vee r(x)))$ no puede expresarse en PROLOG

$\forall x\ (persona(x) \rightarrow (esRica(x) \vee esHonrada(x))$ Error

La regla $\forall x\ (p(x) \rightarrow (q(x) \wedge r(x)))$ sí puede expresarse y puede dividirse en dos:

$$\forall x\ (p(x) \rightarrow q(x)) \text{ y } \forall x\ (p(x) \rightarrow r(x))$$

# Formato de Reglas

$\forall x\ (persona(x) \rightarrow (tienePadre(x) \wedge tieneMadre(x)))$

se divide en: $\forall x\ (persona(x) \rightarrow tienePadre(x))$

$\forall x\ (persona(x) \rightarrow tieneMadre(x))$

PROLOG:

```
tienePadre(X) :- persona(X).

tieneMadre(X) :- persona(X).
```

# Formato de Reglas

Si tenemos antecedentes conectados con ∨, también puede descomponerse la regla.

"Cualquier persona es feliz si es observada por Ana o por Federico"

$$\forall x\ (obs(federico,x) \vee obs(ana,x) \rightarrow feliz(x))$$

equivalente a:

$$\forall x\ (obs(federico,x) \rightarrow feliz(x))$$
$$\forall x\ (obs(ana,x) \rightarrow feliz(x))$$

PROLOG:

```
feliz(X) :- obs(federico,X).
feliz(X) :- obs(ana,X).
```

Nota: Las variables X de las dos reglas son diferentes.

# Formato de Reglas

La consulta compuesta que descubre dónde están las cosas para comer se puede almacenar como una regla con el nombre del predicado donde_comer/2

donde_comer (X, Y) :- ubicacion(X, Y), comida(X, Y).

Indica que "Hay algo de X para comer en la habitación Y si X se encuentra en Y, y X es comida".

```
?- donde_comer(cereales,X).
X = cocina.

?- donde_comer(Y,X).
Y = manzana,
X = cocina ;
Y = cereales,
X = cocina ;
false.
```

# Reglas

Prólogo unifica el patrón de objetivos con el encabezado de la cláusula. Si la unificación tiene éxito, Prolog inicia una nueva consulta con los objetivos en el cuerpo de la cláusula.

Las reglas, en efecto, nos dan múltiples niveles de consultas. El primer nivel está compuesto por los objetivos originales. El siguiente nivel es una nueva consulta compuesta por objetivos encontrados en el cuerpo de una cláusula del primer nivel.

Cada nivel puede crear niveles aún más profundos. Teóricamente, esto podría continuar para siempre. En la práctica puede continuar hasta que el intérprete se quede sin espacio.

# Prolog

```
list_objetos(Lugar) :- ubicacion(X, Lugar), tab(2), write(X), nl, fail.

?- list_objetos(X).
  escritorio
  manzana
  luz_escritorio
  lavadora
  control_remoto
  tv smart
  cereales
  computadora
false.
```

# Prolog

list_puertas(Lugar) :- puerta(Lugar, X), tab(2), write(X), nl, fail.

Definimos una regla que informe la habitación que estamos, que objetos hay y hacia donde podemos ir.

info :- here(Lugar), write('Estas en la habitación: '), write(Lugar), nl, write('Cosas dentro de la habitación: '), nl, list_objetos(Lugar), nl, write('Puedes ir a:'), nl, list_puertas(Lugar).

```
?- info.
Estas en la habitación: oficina
Cosas dentro de la habitación:
  escritorio
  computadora
false.
```

# Prolog

Problema: No se muestra las habitaciones que puedo ir.

Causa: Solo esta definida una dirección para ir de una habitación a otra.

Solución: Definir dos reglas que permitan cada dirección utilizando la misma cabecera, es decir, su cuerpo proviene de una disyunción (OR).

```
?- info.
Estas en la habitación: oficina
Cosas dentro de la habitación:
  escritorio

Puedes ir a:
  comedor
  computadora

Puedes ir a:
  comedor
false.
```

```
conexion(X, Y) :- puerta(X, Y).
conexion(X, Y) :- puerta(Y, X).
list_objetos(Lugar) :- ubicacion(X, Lugar), tab(2), write(X), nl.
list_puertas(Lugar) :- conexion(Lugar,  X), tab(2), write(X), nl, fail.
```

# 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.

# SLD-Resolución

Prolog utiliza un mecanismo conocido como Resolución lineal con unificación para resolver las preguntas que se le plantean.

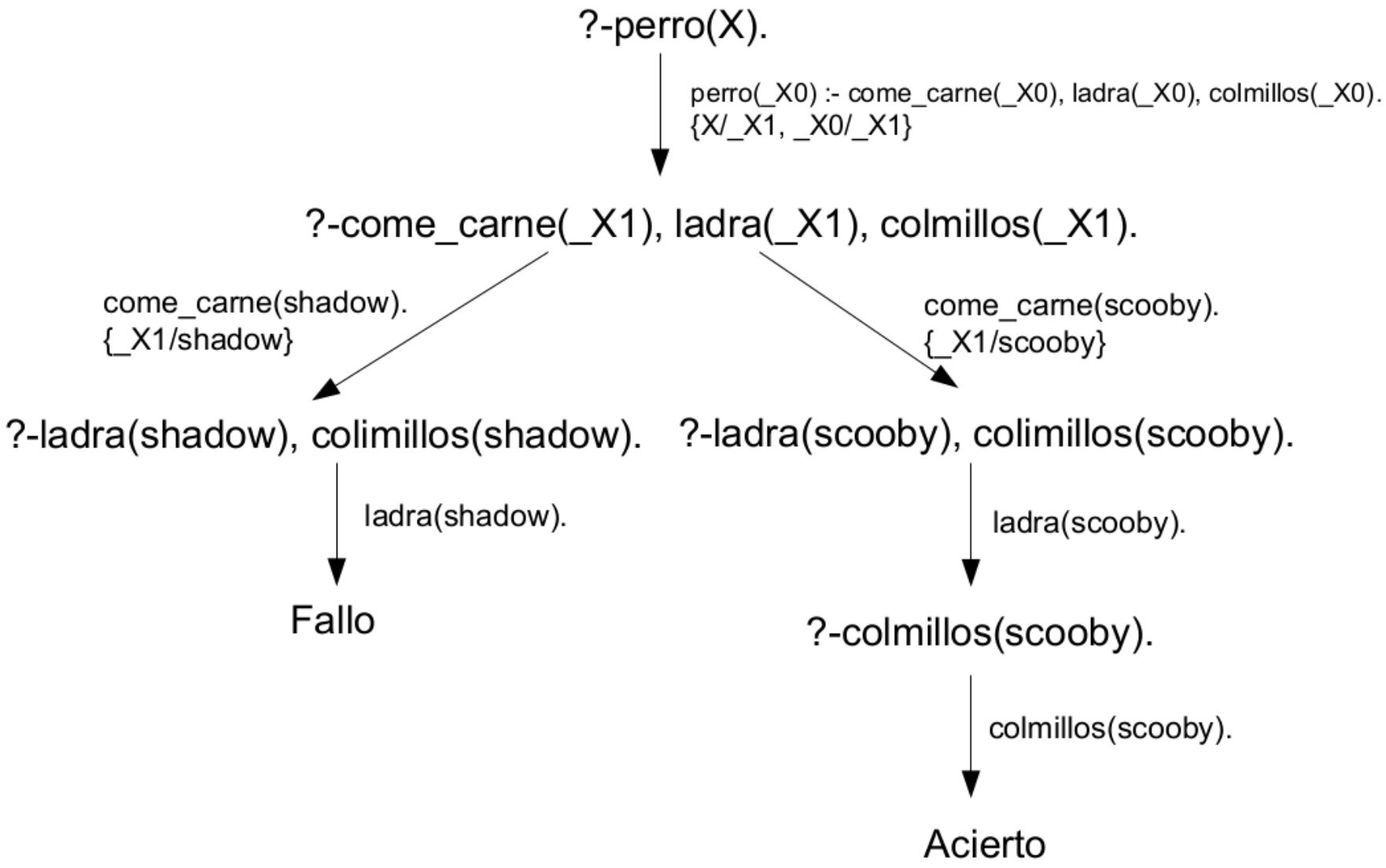
El mecanismo consiste en realizar una búsqueda en profundidad y retroceso (backtracking) tratando de unificar la cláusula objetivo con las contenidas en la base de hechos, hasta lograr alcanzar la cláusula vacía.

**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



# Aritmética

Prolog debe ser capaz de manejar la aritmética para ser un útil lenguaje de programación de propósito general. Sin embargo, la aritmética no encaja bien en el esquema lógico de las cosas.

Prolog proporciona que el predicado intrínseco **is** que evalúa expresiones aritméticas.

X **is** <expresión aritmética>

```
?- X is 3 + 2.
X = 5.

?- X is (30/2 + 5)*3.
X = 60.
```

```
?-  X is 2 + 2, X > 3.
X = 4.

?- X is 2 + 2,  X =< 3.
false.
```

Usando reglas con operaciones aritméticas Convertir la temperatura de grados Celcius a Fahrenheit.

```
cel_a_far(C, F) :- F is C * 9 / 5 + 32.

?- cel_a_far(22,Temp).
Temp = 71.6.
```

# 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 |

# Comparación de expresiones

| | |
|---|---|
| X==Y | la expresión X es igual que la expresión Y |
| X\==Y | la expresión X es distinta que la expresión Y |
| X@<Y | la expresión X es menor que la expresión Y |
| X@>Y | la expresión X es mayor que la expresión Y |
| X@=<Y | la expresión X es menor o igual que la expresión Y |
| X@>=Y | la expresión X es mayor o igual que la expresión Y |

# Manejo de datos

| | |
|---|---|
| asserta(X) | Agrega la cláusula X como primer dato para su predicado. Al igual que los otros predicados de E / S, siempre falla en Backtracking y no deshace su trabajo. |
| assertz(X) | Igual que **asserta**/1, solo agrega la cláusula X como la última cláusula para su predicado. |
| retract(X) | Elimina la cláusula X de la base de conocimiento, nuevamente con un efecto permanente que no se deshace en Backtracking. |

# Prolog

Definimos la regla moverse/1, que mueve al jugador de una habitación a otra.

moverse(Lugar) :- puedo_ir(Lugar), mover(Lugar), info.

Luego, definimos la regla puedo_ir si una habitación se conecta a otra por medio de una puerta

puedo_ir(Lugar) :- here(X), conexion(X, Lugar).

Ahora, sería bueno que nos diera un mensaje cuando no es posible ir a otra habitación.

puedo_ir(Lugar) :- write('No puedes ir hasta la habitación'), tab(2), write(Lugar), tab(2), write('desde aquí'), nl, fail.

```
?- puedo_ir(cocina).
No puedes ir hasta la habitación  cocina  desde aquí
false.
```

# Prolog

Definimos la regla mover/1, que realiza el trabajo de actualizar dinámicamente la base de conocimiento para reflejar la nueva ubicación del jugador.

De esta manera siempre habrá una solo cláusula here/1 que representa el lugar actual. Debido a que moverse/1 llama a puedo_ir/1 antes de mover/1, el nuevo here/1 siempre será un lugar posible en el juego.

mover(Lugar) :- retract(here(_X)), asserta(here(Lugar)).

```
?- moverse(comedor).
Puedes ir hasta la habitación  comedor  desde aquí
Estas en la habitación: comedor
Cosas dentro de la habitación:
  tv smart

Puedes ir a:
  oficina
  garage
  cocina
  dormitorio
false.
```

# Prolog

El predicado dynamic permite modificar la base de conocimiento.

```
:- dynamic here/1.
:- dynamic tengo/1.
:- dynamic ubicacion/2.
```

Definiremos un nuevo predicado, tengo/1, que tiene una cláusula para cada objeto que tiene el jugador. Inicialmente, temgo/1 no está definido porque el jugador no lleva nada.

```
tomar(X) :- puedo_tomar(X), tomar_objeto(X).

puedo_tomar(Objeto) :- here(Lugar), ubicacion(Objeto, Lugar).

tomar_objeto(X) :- retract(ubicacion(X ,_)), asserta(tengo(X)), write('agarrado'), nl.
```

# Prolog

```
?- ubicacion(X, oficina).
X = escritorio ;
X = computadora.

?- tomar(escritorio).
agarrado
true ;
false.

?- ubicacion(X, oficina).
X = computadora.
```

La base de conocimiento permite que todas las cláusulas compartan información sobre una base más amplia, reemplazando la necesidad de variables globales.

Los predicados retract y assert son las herramientas utilizadas para manipular estos datos globales.

# Manejo de datos

El predicado tell/1 abre un archivo y lo establece como fuente de los comandos de escritura. Para cerrarlo se utiliza told/0.

*tell(nombre del archivo).*

El predicado see/1 abre un archivo y lo establece como fuente de los comandos de lectura. Para cerrarlo se utiliza seen/0.

*see(nombre del archivo).*

# Recursividad

Para definir reglas más generales y flexibles, es necesario un mecanismo adicional. Para ello se utilizará el concepto de recursividad.

Este mecanismo no es eficiente, dado que no nos permite generalizar fácilmente el concepto de antecesor. Prolog permite utilizar definiciones recursivas, que resuelven el problema de forma elegante.

Definimos un nuevo predicado, esta_dentro_de/2, que explorará capas de objetos anidados de modo que responderá sí si se le pregunta si la computadora está en la oficina.

Para que el problema sea más interesante, agregaremos algunos elementos más anidados al juego.

```
ubicacion('lampara led',luz_escritorio).
ubicacion('lampara fria', 'lampara led').
ubicacion(interruptor, luz_escritorio).
```

# Recursividad

Si generalizamos una habitación para que sea simplemente otra cosa, podemos establecer una regla de dos partes que se puede usar para deducir si algo está dentro (anidado) de otro objeto.

- Un elemento, E1, está dentro de otro elemento, E2, si E1 está ubicado directamente en E2. (Esta es la condición límite).

- Un elemento, E1, está dentro de otro elemento, E2, si algun elemento intermedio, E, está ubicado en E2 y E1 está dentro de E. (Aquí es donde simplificamos y repetimos).

```
esta_dentro_de(E1,E2) :- ubicacion(E1,E2).
esta_dentro_de(E1,E2) :- esta_dentro_de(E,E2),  ubicacion(E1,E).
```

**1° las más específicas**
**2°  las más generales (con recursividad)**

```
?- esta_dentro_de(X, oficina).
X = escritorio ;
X = computadora ;
X = luz_escritorio ;
X = 'lampara led' ;
X = interruptor ;
X = 'lampara fria';
ERROR: Out of local stack
```

# Recursividad

esta_dentro_de(E1,E2) :- ubicacion(E1,E2).

**1° los términos más específicos.**
**2° los términos más generales (recursivos).**
esta_dentro_de(E1,E2) :- ubicacion(E,E2), esta_dentro_de(E1,E).

?- esta_dentro_de(X, oficina).
X = escritorio ;
X = computadora ;
X = luz_escritorio ;
X = 'lampara led' ;
X = interruptor ;
X = 'lampara fria' ;
false.

?- esta_dentro_de(manzana, comedor).
false.

?- esta_dentro_de(control_remoto, comedor).
true.

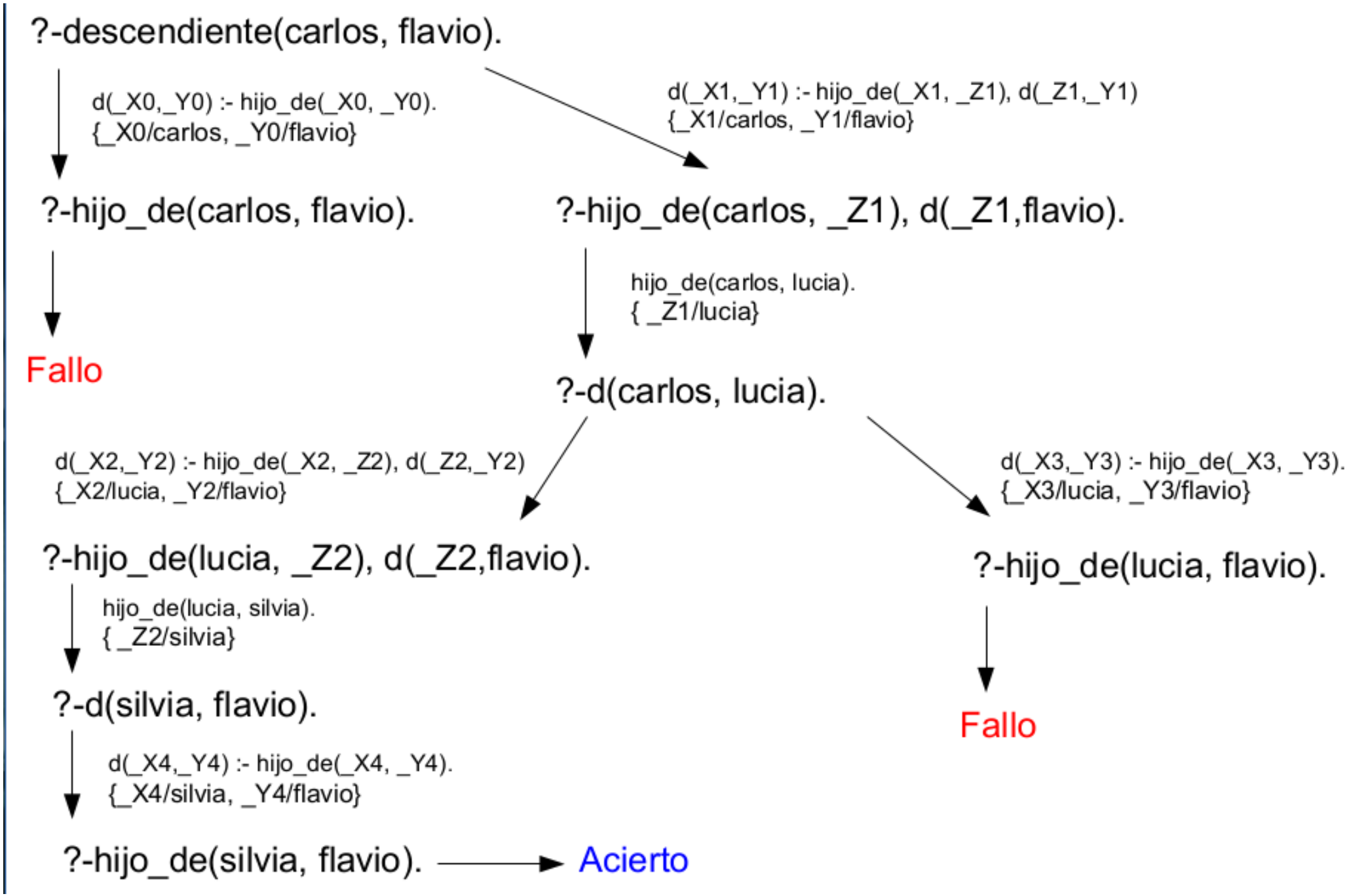
# Recursividad

**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).


?-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

# Problemas de Recursividad

Uno de los peligros que conlleva la recursividad, es la de realizar definiciones circulares o que el intérprete de Prolog no sea capaz de resolver.

esta_dentro_de(E1, E2) :- esta_dentro_de(E1, E2).

Esta cláusula es declarativamente correcta, pero el intérprete de Prolog no podrá resolverla nunca y se quedará atrapado en un bucle infinito.

Si consideramos que el concepto conexion entre dos habitaciones (como vimos anteriormente) tiene una relación conmutativa, entonces posiblemente nos interesaría definir la relación de amistad inversa o complementaria:

puerta(X, Y) :- puerta(Y, X)

Esta definición aparentemente lógica provoca un error. La forma correcta de definirlo sería a través de un nuevo predicado:

conexion(X, Y) :- puerta(X, Y).
conexion(X, Y) :- puerta(Y, X).

# Estructura de datos

Una estructura se compone de un functor, función, y un número fijo de argumentos. Su forma es igual a la de los objetivos y hechos que ya hemos visto.

$$\mathbf{functor(arg_1, arg_2, \ldots, arg_N).}$$

Cada uno de los argumentos de la estructura puede ser un tipo de datos primitivo u otra estructura.

Definiremos estructuras que describan el color, el tamaño y la cantidad de los objetos en la casa.

```
objeto(vaso,transparente,chico,12).
objeto(manzana,verde,chico,6).
objeto(manzana,roja,chico,3).
objeto(lavadora,blanco,grande,1).
```

# Prolog

Estas estructuras podrían usarse directamente en el primer argumento de ubicacion/2, pero para la experimentación, en cambio, definiremos un nuevo predicado, ubicacion_detallada/2.

```
ubicacion_detallada(objeto(vaso,transparente,chico,12), cocina).
ubicacion_detallada(objeto(manzana,verde,chico,6), cocina).
ubicacion_detallada(objeto(manzana,roja,chico,3), cocina).
ubicacion_detallada(objeto(lavadora,blanco,grande,1),cocina).
```

```
?- ubicacion_detallada(X, cocina).
X = objeto(vaso, transparente, chico, 12) ;
X = objeto(manzana, verde, chico, 6) ;
X = objeto(manzana, roja, chico, 3) ;
X = objeto(lavadora, blanco, grande, 1).

?- ubicacion_detallada(objeto(Elemento, verde, chico, Cantidad), cocina).
Elemento = manzana,
Cantidad = 6 ;
false.
```

# Prolog

Si no nos importara el tamaño y la cantidad, podríamos reemplazar las variables de tamaño y cantidad con variables anónimas (_).

```
?- ubicacion_detallada(objeto(X, blanco,_,_), cocina).
X = lavadora.
```

Usaremos estructuras para agregar más realismo al juego. Modificaremos el predicado puede_tomar/1 por puede_tomar_detallada/1

```
puede_tomar_detallada(Elemento) :- here(Habitacion),
ubicacion_detallada(objecto(Elemento ,_,chico,_), Habitacion).
```

# Negación not

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

La negacion **not/1** no corresponde a una negación lógica, sino al hecho de que no hay evidencia para demostrar lo contrario.

**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.

# Negación not

```
?- not(habitacion(oficina)).
false.

?- not(ubicacion(auto,garage)).
true.
```

Semánticamente, el **not/1** en Prolog significa que el objetivo no se puede resolver con éxito con la base de conocimientos actual de hechos y reglas.

```
puede_tomar_detallada(Elemento):- here(Lugar), ubicacion_detallada(objeto(Elemento,_,grande,_), Lugar), write('El '), write(Elemento), write(' es demasiado grande para llevar'), nl, fail.

puede_tomar_detallada(Elemento) :- here(Lugar), not(ubicacion_detallada(objeto(Elemento ,_,_,_),Lugar)), write('No hay '), write(Elemento), write(' aquí'), nl, fail.
```

```
?- puede_tomar_detallada(computadora).
true ;
false.

?- puede_tomar_detallada(mouse).
No hay mouse aquí
false.
```

# Unificación

Una de las características más poderosas de Prolog es su algoritmo intrinseco de combinación de patrones, **unificación**.

Variable y cualquier término: La variable se unificará y estará unida a cualquiera de los términos, incluyendo otra variable.

Primitiva y primitiva: Dos términos primitivos (átomos o números) se unifican solo si son idénticos.

Estructura y estructura: Dos estructuras se unifican si tienen la misma functor y aridad y si cada par de argumentos unifican.

El signo igual (=) no causa asignación como en la mayoría de los lenguajes de programación, ni causa evaluación aritmética sino que causa la unificación del prólogo.

# Prolog

```
?- ubicacion(manzana, cocina) = ubicacion(manzana, cocina).
true.

?- ubicacion(manzana, cocina) = ubicacion(pera, cocina).
false.

?- ubicacion(manzana, cocina) = ubicacion(X, cocina).
X = manzana.

?- ubicacion(manzana, cocina) = ubicacion(X, Y).
X = manzana,
Y = cocina.

?- a(b, X) = a(b, c(d, e)).
X = c(d, e).
?- mesa(X,Y) = Z, write(Z), nl, bebida(X),comida(Y), write(Z).
mesa(_11330,_11332)
mesa(leche,manzana)
X = leche,
Y = manzana,
Z = mesa(leche, manzana) ;
mesa(leche,cereales)
X = leche,
Y = cereales,
Z = mesa(leche, cereales).
```

# Listas

Las listas son estructuras de datos poderosas para sostener y manipular grupos de cosas.

En Prolog, una lista es simplemente una colección de términos.

Los términos pueden ser cualquier tipo de datos, incluidas las estructuras y otras listas.

Sintácticamente, una lista se denota entre corchetes con los términos separados por comas.

**[elemento$_1$, elemento$_2$, … , elemento$_n$ ]**

Hay una lista especial, llamada lista vacía, que está representada por un conjunto de corchetes vacíos ([]). También se conoce como nulo.

```
ubicacion_lista([manzana, vaso, cereales], cocina).
ubicacion_lista([escritorio, computadora], oficina).
ubicacion_lista([luz_escritorio, papeles, lapices], escritorio).
ubicacion_lista(['lampara led', interruptor], luz_escritorio).
ubicacion_lista(['tv smart'], comedor).
```

# Listas

```
?- ubicacion_lista(X, cocina).
X = [manzana, vaso, cereales].

?- [_,X,_] = [manzana, vaso, cereales].
X = vaso.
```

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

**[Cabeza | Cola]**

[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)

# Prolog

```
?- [H|T]=[manzana, vaso, cereales].
H = manzana,
T = [vaso, cereales].

?- [H|T]=[manzana, cereales].
H = manzana,
T = [cereales].

?- [H|T]=[manzana, [vaso, cereales]].
H = manzana,
T = [[vaso, cereales]].

?- [H|T]=[manzana].
H = manzana,
T = [].

?- [Elemento1, Elemento2|T]=[manzana, vaso, cereales, leche].
Elemento1 = manzana,
Elemento2 = vaso,
T = [cereales, leche].
```

# Listas

El predicado member/2 devuelve los elementos de una lista

*member(Elemento, Lista).*

```
?- member(leche, [manzana, vaso, cereales, leche]).
true.
```

El predicado append/3 concatena las dos primeras lista en la tercera.

*append(Lista1, Lista2, Lista3).*

```
?- append([manzana, pera],[banana, durazno, sandia], Frutas).
Frutas = [manzana, pera, banana, durazno, sandia].
```

# Listas

El predicado length/2 encuentra la longitud de una lista.

*length(Lista, Longitud).*

```
?- length([[manzana, pera],[banana, durazno, sandia]], Frutas).
Frutas = 2.

?- length([manzana, pera,banana, durazno, sandia], Frutas).
Frutas = 5.
```

El predicado reverse/2 devuelve la lista inversa de una dada.

*reverse( Lista, Lista_Res).*

```
?- reverse([manzana, pera,banana, durazno, sandia], Frutas).
Frutas = [sandia, durazno, banana, pera, manzana].
```

# Listas

El predicado eliminar/3 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).
```

# Listas

El predicado sustituir todas las ocurrencias de un elemento (1er argumento) 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).
```

# Prolog

En el juego, será necesario añadir cosas a ubicacion_lista cuando algo se pone en una habitación. Podemos escribir sumar_objeto/3 que utiliza append/3.

```
sumar_objeto(Nuevo_Objeto, Contenedor, NuevaLista) :-
ubicacion_lista(Lista_Antigua,Contenedor), append([Nuevo_Objeto],Lista_Antigua,NuevaLista).

sumar_objeto2(Nuevo_Objeto, Contenedor, [Nuevo_Objeto|Lista_Antigua]) :-
ubicacion_lista(Lista_Antigua,Contenedor).

sumar_objeto3(Nuevo_Objeto, Contenedor, NuevaLista) :-
ubicacion_lista(Lista_Antigua,Contenedor), NuevaLista = [Nuevo_Objeto|Lista_Antigua].
```

Alternativa:

```
:- dynamic ubicacion_lista/2.
poner_objeto(Objeto, Lugar) :- retract(ubicacion_lista(Lista ,Lugar)),
asserta(ubicacion_lista([Objeto|Lista],Lugar)).
```

# Prolog

```
?- sumar_objeto(miel, cocina, Objetos_Cocina).
Objetos_Cocina = [miel, manzana, vaso, cereales].

?- sumar_objeto2(miel, cocina, Objetos_Cocina).
Objetos_Cocina = [miel, manzana, vaso, cereales].

?- sumar_objeto3([miel, cafe], cocina, Objetos_Cocina).
Objetos_Cocina = [[miel, cafe], manzana, vaso, cereales].
```

```
?- poner_objeto(miel, cocina).
true.

?- ubicacion_lista(X, cocina).
X = [miel, manzana, vaso, cereales].
```

# Listas

Transformar múltiples hechos en una lista es más difícil. Para realizar esta tarea existe un predicado intrínseco llamado findall/3 cuyos argumentos son:

$arg_1$: Un patrón para los términos en la lista resultante.

$arg_2$: Un patrón objetivo.

$arg_3$: una lista resultante.

findall/3 realiza automáticamente un Backtraking completo del patrón de objetivos y almacena cada resultado en la lista.

```
?- findall(X, conexion(cocina,X), L).
L = [comedor].

?- findall(X, conexion(comedor,X), L).
L = [oficina, garage, cocina, dormitorio].
```

# 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.

# 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.

# 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 largo del resto de la resolución no se permite una nueva resolución de dicho predicado, ni las condiciones, ni las sustituciones obtenidas.

# Cut

```
donde_comer(X, Y) :- ubicacion(X, Y), comida(X).
```

```
?- donde_comer(X,Y).
X = manzana,
Y = cocina ;
X = cereales,
Y = cocina ;
X = miel,
Y = comedor.
```

```
donde_comer(X, Y) :- ubicacion(X, Y), !, comida(X).
```

```
?- donde_comer(X,Y).
false.
```

```
donde_comer(X, Y) :- ubicacion(X, Y), comida(X), !.
```

```
?- donde_comer(X,Y).
X = manzana,
Y = cocina.
```

# Cut

```
ubicacion(cereales, cocina).
ubicacion(cereales, comedor).
comida(cereales).
comida(manzana).
donde_comer(X, Y) :-  comida(X), !,ubicacion(X, Y).
```

```
?- donde_comer(X,Y).
X = cereales,
Y = cocina ;
X = cereales,
Y = comedor.
```

```
donde_comer(X, Y) :-comida(X), ubicacion(X, Y),!.
```

```
?- donde_comer(X,Y).
X = cereales,
Y = cocina.
```