Primeros pasos con `prolog`

Todos estos ejemplos están extraídos de Learn prolog now.

Sólo hechos

Nuestro primer ejemplo es con una base de datos con hechos.

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

Supongamos que tenemos un fichero `kb1.pl` con estas afirmaciones. Para cargarlo en `prolog` tenemos que teclear `['kbl1.pl']` en la línea de comandos. Y obtendremos lo siguiente.

```
?- ['kb1.pl']
true.
```

Podemos a partir de entonces hacerle peticiones a `prolog` relativas a la base de conocimiento que hemos cargado.

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

```
?- woman(mia).
true.

?- woman(pepe).
false.

?- party.
true.

?- fiesta.
ERROR: Undefined procedure: fiesta/0 (DWIM could not cor
```

Hechos y reglas

Veamos ahora un ejemplo con hechos y reglas.

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2Music(mia).
playsAirGuitar(yolanda):- listens2Music(yolanda).
```

```
?- playsAirGuitar(mia).
true.

?- playsAirGuitar(yolanda).
true.
```

Modifiquemos un poco las reglas para introducir variables.

```
happy(yolanda).
listens2Music(mia).
listens2Music(X):- happy(X).
playsAirGuitar(X):- listens2Music(X).
```

Podemos preguntar quién toca la guitarra.

```
?- ['kb2-1.1.pl'].
true.

?- playsAirGuitar(X).
X = mia ;
X = yolanda.
```

Unificación

El predicado `=` se utiliza para unificar términos. Se puede usar tanto de forma infija como prefija.

```
?- X=mia.
X = mia.

?- =(X,mia).
X = mia.

?- f(X,a)=f(b,Y).
X = b,
Y = a.
```

En principio `prolog` no se va a preocupar de si existen posibles bucles infinitos, pero podemos hacer que detecte ocurrencias de variables en términos.

```
?- X=f(X).
X = f(X).

?- unify_with_occurs_check(X,f(X)).
false.
```

En GNU `prolog` obtenemos

```
| ?- X=f(X).

cannot display cyclic term for X
```

O uno un poco más elaborado:

```
?- f(X,f(Y))=f(Y,X).
X = Y, Y = f(Y).

?- unify_with_occurs_check(f(X,f(Y)),f(Y,X)).
false.
```

Unificación y programación

Se puede utilizar la unificación para definir predicados que tengan un significado especial en caso de que los términos sean unificables.

```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).
```

Si cargamos esta base de conocimiento en un `prolog` moderno, obtendremos una serie de advertencias diciendo que ni `Y` ni `Z` son usadas después de la línea 1, y lo mismo ocurre con la línea 2.

Para evitar esto, podemos hacer uso de variables anónimas.

```
vertical(line(point(X,_),point(X,_))).
horizontal(line(point(_,Y),point(_,Y))).
```

```
?- vertical(line(point(1,2),point(1,3))).
true.
?- vertical(line(point(3,2),point(1,3))).
false.
?- vertical(line(point(1,2),point(1,X))).
true.
?- vertical(line(point(1,2),point(X,Y))).
X = 1.
```

Pero también obtenemos

```
?- vertical(line(point(1,2),point(1,2))).
true.
```

Por lo que podemos cambiar nuestra base de conocimiento por

```
vertical(line(point(X,Y),point(X,T))):-Y\==T.
```

```
?- vertical(line(point(1,2),point(1,2))).
false.

?- vertical(line(point(1,2),point(X,Y))).
X = 1.
```

Árbol de búsqueda

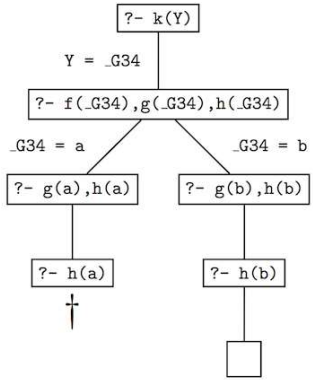
Tomemos ahora la base de conocimiento

```
f(a).
f(b).

g(a).
g(b).

h(b).

k(X) :- f(X), g(X), h(X).
```

y hagamos la pregunta

```
?- k(Y).
Y = b.
```

Éste sería el árbol de búsqueda.

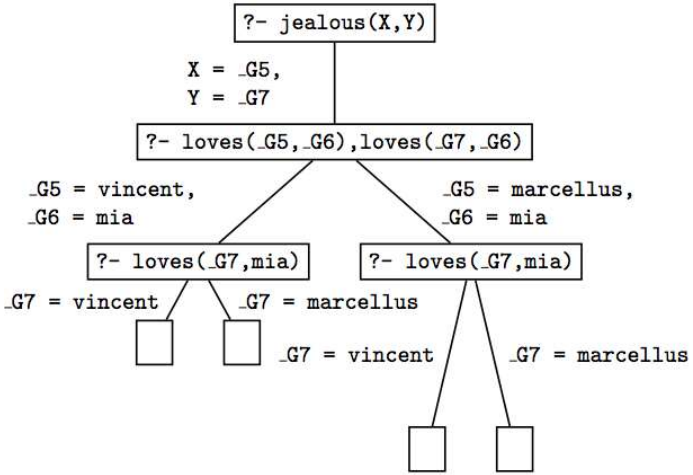


Pongamos ahora la base

```
loves(vincent,mia).
loves(marcellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

y planteamos la pregunta

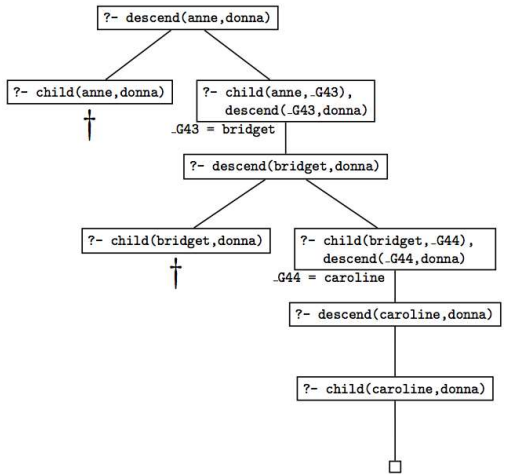
```
?- jealous(X,Y).
X = Y, Y = vincent ;
X = vincent,
Y = marcellus ;
X = marcellus,
Y = vincent ;
X = Y, Y = marcellus.
```



Recursividad

```
child(anne,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

```
?- descend(anne,donna).
true ;
false.
```



Ejemplo: naturales

```
natural(0).
natural(s(X)) :- natural(X).

suma(0,X,X).
suma(s(X),Y,s(Z)):-suma(X,Y,Z).

traduce(0,0).
traduce(s(X),N) :- traduce(X, M),
                   N is M+1.
```

```
?- traduce(s(0),X).
X = 1.

?- natural(s(s(1))).
false.

?- natural(s(s(0))).
true.

?- suma(s(0),s(s(0)),X).
X = s(s(s(0))).

?- suma(s(0),X,s(0)).
X = 0.

?- suma(X,s(0),s(s(0))).
X = s(0) ;
false.

?- suma(s(0),s(s(0)),X), traduce(X,Y).
X = s(s(s(0))),
Y = 3.
```

Listas

Una lista en `prolog` es una secuencia de términos. Tenemos patrones para recuperar cabeza y cola de una lista.

```
?- [H|T] = [1,2,pepe,juan].
H = 1,
T = [2, pepe, juan].
```

O elementos en particular

```
?- [Primero,Segundo|Resto] = [1,2,pepe,juan].
Primero = 1,
Segundo = 2,
Resto = [pepe, juan].

?- [Primero,_,Tercero|Resto] = [1,2,pepe,juan,maria].
Primero = 1,
Tercero = pepe,
Resto = [juan, maria].
```

Predicados con Listas

Implementemos un predicado pertenece.

```
pertenece(X,[X|_]).
pertenece(X,[_|T]) :- pertenece(X,T).
```

```
?- pertenece(1,[1,2,3,1]).
true ;
true ;
false.

?- pertenece(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
false.
```

Más ejemplos con Listas

```
concat([],Y,Y).
concat([H|T],Y,[H|TT]) :- concat(T,Y,TT).

conc([],Y,Y).
conc([H|T],Y,Z) :- conc(T,Y,U), Z=[H|U].

rev([],[]).
rev([H|T],X):-rev(T,Y), concat(Y,[H],X).

rever([],X,X).
rever([H|T],A,R):- rever(T,[H|A],R).
vuelta(X,V):- rever(X,[],V).
```

Aritmética entera

```
?- 8 is 6+2.
yes
?- 12 is 6*2.
yes
?- -2 is 6-8.
yes
?- 3 is 6/2.
yes
?- 1 is mod(7,2).
yes
?- X is 6+2.
X=8
?- R is mod(7,2).
R=1
```

Cuidado, una consulta como `3+2 is X` daría un error de instanciación

Además expresiones enteras no se evalúan si no se pide explícitamente (con un `is`)

```
?- X=3+2
X=3+2
yes
```

Predicados con aritmética entera

```
len([], 0).
len([_|T], N) :- len(T, X), N is X+1.

maxl([H], H).
maxl([H|T], H) :- maxl(T, Y), H>Y.
maxl([H|T], Y) :- maxl(T, Y), H=<Y.
```

Y con acumuladores

```
accLen([_|T], A, L) :- Anew is A+1, accLen(T, Anew, L).
accLen([], A, A).

leng(List, Length) :- accLen(List, 0, Length).

accMax([H|T],A,Max) :- H>A, accMax(T,H,Max).
accMax([H|T],A,Max) :- H=<A, accMax(T,A,Max).
accMax([],A,A).
max(List,Max) :- List = [H|_], accMax(List,H,Max).
```

```
?- len([a,b,c,d,e,[a,b],g],X).
X=7

?- maxl([3,1,2],X).
X = 3 .
```

Comparación `prolog`

$x < y$	x<y
$x \leq y$	x=<y
$x > y$	x>y
$x \geq y$	x>=y
$x = y$	x=:=y
$x \neq y$	x=\=y

Estos predicados fuerzan la evaluación de sus argumentos

Igualdades y desigualdades

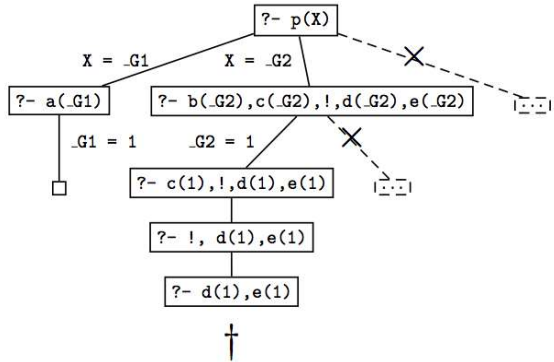
En `prolog` hay varios símbolos para denotar igualdad o desigualdad

Símbolo	Significado
=	unificador
\=	no es unificable
==	identidad
\==	no son idénticos
=:=	igualdad para aritmética
=\=	desigualdad para aritmética

Cortes

El predicado `!` siempre es cierto. Además, si algún objetivo hace uso de la cláusula que lo contiene (objetivo padre), el corte restringe la búsqueda a las sustituciones de variables hechas para unificar la cabecera de la regla con el objetivo padre, y hace que `prolog` use sólo esa regla.

```
p(X):- a(X).
p(X):- b(X), c(X), !, d(X), e(X).
p(X):- f(X).
a(1). b(1). c(1). d(2). e(2). f(3). b(2). c(2).
```



Un ejemplo con corte

```
max(X,Y,Y) :- X=<Y,!.
max(X,Y,X) :- X> Y.
```

```
max(X,Y,Z) :- X =< Y,!, Y = Z.
max(X,Y,X).
```

Corte y `fail`

El predicado `fail` es siempre falso.

```
enjoys(vincent,X) :- big_kahuna_burger(X),!,fail.
enjoys(vincent,X) :- burger(X).
burger(X) :- big_mac(X).
burger(X) :- big_kahuna_burger(X).
burger(X) :- whopper(X).
big_mac(a).
big_kahuna_burger(b).
big_mac(c).
whopper(d).
```

```
?- enjoys(vincent,a).
yes
?- enjoys(vincent,b).
no
?- enjoys(vincent,c).
yes
?- enjoys(vincent,d).
yes
```

```
enjoys(vincent,X) :- burger(X),
                     \+ big_kahuna_burger(X).
```

Analizando términos

En `prolog` tenemos una serie de predicados que nos facilitan el estudio de la estructura de un término.

predicado	**función**
atom/1	detecta si es un átomo
integer/1	detecta si es un entero
float/1	detecta si es un real
number/1	detecta si es un número (entero o real)
atomic/1	determina si es una constante
var/1	determina si es una variable no instanciada
unvar/1	determina si no es una variable o una variable instanciada

Más información sobre términos

Además, `functor/3` devuelve el nombre de una función y la ariedad.

```
?- functor(f(a,b),F,A).
A = 2
F = f
yes
?- functor([a,b,c],X,Y).
X = '.'
Y = 2
yes
```

El predicado `arg/3` sirve para seleccionar o instanciar argumentos de un término.

```
?- arg(2,loves(vincent,mia),X).
X = mia
yes

?- arg(2,loves(vincent,X),mia).
X = mia
yes

?- arg(2,happy(yolanda),X).
no
```

Un ejemplo

Podemos definir un predicado `complexterm/1` que determine si un término es simple o compuesto.

O una función `sumandos` que extraiga los sumandos de una suma de términos.

```
complexterm(X):- nonvar(X), functor(X,_,A), A > 0.

sumandos(A,[A]):-var(A),!.
sumandos(C,[C]):-not(functor(C,+,_)).
sumandos(A+B,W):-sumandos(A,U),
                 sumandos(B,V),
                 append(U,V,W).
```

```
?- complexterm(a).
false.

?- complexterm(f(a)).
true.

?- complexterm(1+2).
true.

?- sumandos(A+b*C+Y,L).
L = [A, b*C, Y].
```

De términos a Listas

El predicado `'=..'/2` se usa para aplanar un termino en una lista, y al revés.

```
?- cause(vincent,dead(zed)) =.. X.
X = [cause, vincent, dead(zed)]
yes

?- X =.. [a,b(c),d].
X = a(b(c), d)
yes
```

Definiendo nuevos operadores

Para definir nuevmos operadores podemos usar

```
:-op(precedencia, tipo, nombre).
```

Tenemos los siguientes tipos

- xfx, infijo sin asociatividad
- xfy, infijo asociativo derecha
- yfx, infijo asociativo izquierda
- fx, fy, prefijo
- xf, yf, sufijo

Así para `:-op(100,xf,t).` la entrada `1 t t.` da un error de paréntesis, mientras que para `:-op(100,yf,t).`, la entrada `1 t t.` se interpreta como `1 (1 t)`.