

Metodología de Programación I

Tutorial Prolog 1/3

Dr. Alejandro Guerra-Hernández

Departamento de Inteligencia Artificial
Facultad de Física e Inteligencia Artificial
aguerra@uv.mx
<http://www.uv.mx/aguerra>

Maestría en Inteligencia Artificial 2009



Universidad Veracruzana

Hechos

- ▶ Se describen una serie de **hechos** conocidos sobre geografía. **Constantes** y **predicados** inician con minúsculas. Las **variables** empiezan con mayúsculas. Los hechos acaban en punto. Los **comentarios** inician con “%”.

```
1   %%% loc_en/2
2   %%% loc_en(+X,+Y)
3   %%% La ciudad X se localiza en el estado Y
4
5   loc_en(atlanta,georgia).
6   loc_en(huston,texas).
7   loc_en(austin,texas).
8   loc_en(boston,massachussets).
9   loc_en(xalapa,veracruz).
10  loc_en(veracruz,veracruz).
```



Consultas

- ▶ **true.** Se puede deducir y la meta no tiene variables.

```
?- loc_en(xalapa,veracruz).  
true.
```

- ▶ **false.** No se puede deducir. Se asume la **NAF**.

```
?- loc_en(xalapa,texas).  
false.
```

- ▶ **Substitución de respuesta.** Se puede deducir y la meta tiene variables.

```
?- loc_en(Ciudad,texas).  
Ciudad = houston ;  
Ciudad = austin ;  
false.
```



Reglas

- ▶ Extienden la **semántica** de localizado en:

```
1  %% loc_en(+X,+Y)
2  %% La ciudad X está localizada en el país Y
3
4  loc_en(X,usa) :- loc_en(X,georgia).
5  loc_en(X,usa) :- loc_en(X,texas).
6  loc_en(X,usa) :- loc_en(X,massachussets).
7  loc_en(X,mex) :- loc_en(X,veracruz).
8
9  %% loc_en(+X,norteamerica)
10 %% La ciudad X está en norteamérica
11
12 loc_en(X,norteamerica) :- loc_en(X,usa).
13 loc_en(X,norteamerica) :- loc_en(X,mexico).
```



Consultas

- ▶ El mismo predicado *loc_en/2* nos sirve para **computar** si xalapa está en USA, o en Norte América; ¿Qué está en México? ¿Donde se localiza Xalapa?

```
?- loc_en(xalapa,usa).
false.
?- loc_en(xalapa,norteamerica).
true
?- loc_en(C,mexico).
C = xalapa ;
C = veracruz ;
false.
?- loc_en(xalapa,Loc).
Loc = veracruz ;
Loc = mexico ;
Loc = norteamerica ;
false.
```



Consultas negativas

- ▶ Se puede usar el predicado `not` (`\+`) para preguntar ¿Es verdad que Xalapa no está en USA? Las consultas pueden dar sorpresas ¿Porqué?

```
?- \+ loc_en(xalapa,usa).  
true.  
?- \+ loc_en(C,usa).  
false.
```

- ▶ Pero:

```
?- loc_en(C,norteamerica), \+ loc_en(C,usa).  
C = xalapa ;  
C = veracruz ;  
false.
```



Definición de listas

- ▶ `[]` es una lista **vacía**. Si `Xs` es una lista, entonces `[X|Xs]` es una lista. `X` denota **cabeza** y `Xs` **cola**. **Recursividad!**

```

1  ?- [1,2,3] = [1 | [2 | [3 | []]]].
2  Yes
3  ?- [X|Xs] = [1,2,3].
4  X = 1
5  Xs = [2, 3] ;
6  No
7  ?- [X|Xs] = [1].
8  X = 1
9  Xs = [] ;
10 No
11 ?- [X,Y|Xs] = [1,2,3].
12 X = 1
13 Y = 2
14 Xs = [3]
15 Yes
16 ?- X = "abcd".
17 X = [97, 98, 99, 100]
18 Yes

```



Recorriendo listas

- ▶ Todas las ciudades de la lista son gringas:

```
1 %% gringas/1
2 %% Las ciudades en la lista X son gringa
3 gringas([X]) :- loc_en(X,usa).
4 gringas([X|Xs]) :- loc_en(X,usa),gringas(Xs).
```

- ▶ Consultas:

```
?- gringas([boston]).
true
?- gringas([]).
false.
?- gringas([boston,huston,georgia]).
false.
?- gringas([boston,huston,atlanta]).
true
```



Otro recorrido: miembro

► La definición:

```
1  %%% miembro(X,Ys)
2  %%% El elemento X es miembro de la lista Ys
3  miembro(X,[X|_]).
4  miembro(X,[_|Ys]) :- miembro(X,Ys).
```

► Las consultas:

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



Longitud de una lista

► La definición:

```
1 long([],0).  
2 long([_ | Xs],L) :- long(Xs,L1), L is L1+1.
```

► Las consultas:

```
?- long([1,2,3,4],L).  
L = 4;  
No  
?- long([],L).  
L = 0;  
No
```



Recursividad no a la cola

► La traza:

```
?- trace.  
[trace] ?- long([1,2],L).  
  Call: (7) long([1, 2], _G309) ?  
  Call: (8) long([2], _L350) ?  
  Call: (9) long([], _L369) ?  
  Exit: (9) long([], 0) ?  
  ^ Call: (9) _L350 is 0+1 ?  
  ^ Exit: (9) 1 is 0+1 ?  
  Exit: (8) long([2], 1) ?  
  ^ Call: (8) _G309 is 1+1 ?  
  ^ Exit: (8) 2 is 1+1 ?  
  Exit: (7) long([1, 2], 2) ?  
L = 2
```



Otra longitud (acumuladores)

- ▶ Consideren la definición alternativa de longitud:

```
1 longTR(Xs,L) :- long(Xs,0,L).
2 long([],Acc,Acc).
3 long([_ | Xs],Acc,L) :-
4     Acc1 is Acc + 1,
5     long(Xs,Acc1,L).
```

- ▶ Las consultas:

```
?- longTR([1,2,3,4],L).
L = 4;
No
?- longTR([],L).
L = 0;
No
```



Recursividad a la cola

- Observen la traza:

```
[trace] ?- longTR([1,2,3,4],L).  
  Call: (7) longTR([1, 2, 3, 4], _G315) ?  
  Call: (8) long([1, 2, 3, 4], 0, _G315) ?  
  ^ Call: (9) _L368 is 0+1 ?  
  ^ Exit: (9) 1 is 0+1 ?  
  Call: (9) long([2, 3, 4], 1, _G315) ?  
  ^ Call: (10) _L388 is 1+1 ?  
  ^ Exit: (10) 2 is 1+1 ?  
  Call: (10) long([3, 4], 2, _G315) ?  
  ^ Call: (11) _L408 is 2+1 ?  
  ^ Exit: (11) 3 is 2+1 ?  
  Call: (11) long([4], 3, _G315) ?  
  ^ Call: (12) _L428 is 3+1 ?  
  ^ Exit: (12) 4 is 3+1 ?  
  Call: (12) long([], 4, _G315) ?  
  Exit: (12) long([], 4, 4) ? ...
```



A pensar

- ▶ ¿Cómo probamos si esto es cierto (recursivo a la cola vs recursivo)?
- ▶ Función misterio:

```
1  mist(0, []).  
2  mist(N, [N|Ns]) :- N1 is N-1, mist(N1, Ns).
```



Resultados

Desempeño de long y longTR (time/1)

Método	N	infs	CPU	secs	Lips
long	1000	2001	0.01	0.00	200100
longTR	1000	2002	0.00	0.00	Infinite
long	10000	20001	0.02	0.02	1000050
longTR	10000	20002	0.01	0.02	2000200
long	100000	200001	0.17	0.22	1176476
longTR	100000	200202	0.14	0.16	1428586
long	1000000	348726	0.29	1.85	out stack
longTR	1000000	2000002	2.09	2.31	956939



Concatenación (append)

► La definición:

```
1 append([], Ys, Ys).  
2 append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys,  
    Zs).
```

► Las consultas:

```
?- append([1,2,3], [4,5,6], L).  
L = [1, 2, 3, 4, 5, 6].  
?- append(X, Y, [1,2,3]).  
X = [],  
Y = [1, 2, 3] ;  
X = [1],  
Y = [2, 3] ;  
X = [1, 2],  
Y = [3] ;  
X = [1, 2, 3],  
Y = [] ;  
false.
```



Partes de una lista

► Las definiciones:

```
1 prefijo(Xs,Ys) :- append(Xs,Bs,Ys).
2 sufijo(Xs,Ys) :- append(As,Xs,Ys).
3 sublista(Xs,Ys):- prefijo(Aux,Ys), sufijo(Xs,
    Aux).
```

► Las consultas:

```
?- prefijo([1,2],[1,2,3]).
true
?- sufijo([2,3],[1,2,3]).
true
?- sublista([2,3],[1,2,3,4]).
true
```



Reverso de una lista

- ▶ Es posible definir versiones recursivas (naif) y recursivas a la cola:

```
1  reverso([], []).
2  reverso([X|Xs], Res) :-
3      reverso(Xs, XsReverso),
4      append(XsReverso, [X], Res).
5
6  reversoTR(L, Res) :-
7      nonvar(L),
8      reversoTRaux(L, [], Res).
9  reversoTRaux([], Acc, Acc).
10 reversoTRaux([X|Xs], Acc, Res) :-
11     reversoTRaux(Xs, [X|Acc], Res).
```



Datos compuestos

- ▶ $punto(X, Y)$ representa un punto con sus coordenadas.
- ▶ $seg(P1, P2)$ representa un segmento cuyos extremos son los puntos $P1$ y $P2$.
- ▶ entonces, podemos definir vertical y horizontal:

```
1 horizontal(seg(punto(X,Y),
2             punto(X1,Y))).
3
4 vertical(seg(punto(X,Y),
5             punto(X,Y1))).
```



Consulta

- ▶ Los datos pueden estructurarse como términos compuestos.
- ▶ La segunda consulta indica que P es un punto cuya primera coordenada no está instanciada y la segunda es 2.

```
1  ?- horizontal(seg(punto(1,2),
2                        punto(3,2))).
3  true
4
5  ?- horizontal(seg(punto(1,2),P)).
6  P = punto(_G240, 2) ;
7  false
```



Consultas

- ▶ `=` verifica la unificación; `\=` verifica la no unificación.
- ▶ Prolog no tiene chequeo de ocurrencias, por eso la última meta produce un ciclo infinito (**).

```
?- f(X,X) = f(a,Y).  
X = a  
Y = a.  
?- f(X,X) \= f(a,Y).  
false.  
?- p(f(X),g(Z,X)) = p(Y, g(Y,a)).  
X = a  
Z = f(a)  
Y = f(a).  
?- p(X,X) = p(Y,f(Y)).  
X = f(**)  
Y = f(**).
```

