Tema 6: Representación lógica del conocimiento

José A. Alonso Jiménez
Miguel A. Gutiérrez Naranjo
Francisco J. Martín Mateos

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Un ejemplo simple: divisibilidad

- **Problema: Escribir un programa para declarar que 2 divide a 6**
- **Programa:** `divisibilidad-1.pl`

```
divide(2,6).
```

- **Sesión**
  - **¿2 divide a 6?.**

```
?- divide(2,6).
Yes
```

  - **¿3 divide a 12?.**

```
?- divide(3,12).
No
```

  - **¿Cuáles son los múltiplos de 2?.**

```
?- divide(2,X).
X = 6
Yes
```

  - **¿Cuáles son los elementos X e Y tales que X divide a Y?.**

```
?- divide(X,Y).
X=2
Y=6
Yes
```

Ampliación del programa

- **Problema: Ampliar el programa anterior, añadiéndole que 2 divide a 12 y que 3 divide a 6 y a 12**
- **Programa:** `divisibilidad-2.pl`

```
divide(2,6).
divide(2,12).
divide(3,6).
divide(3,12).
```

- **Sesión**
  - **¿Cuáles son los elementos X e Y tales que X divide a Y?**

```
?- divide(X,Y).
X = 2   Y = 6 ;
X = 2   Y = 12 ;
X = 3   Y = 6 ;
X = 3   Y = 12 ;
No
```

  - **¿Cuáles son los múltiplos de 2 y de 3?**

```
?- divide(2,X), divide(3,X).
X = 6 ;
X = 12 ;
No
```

Reglas

- **Problema: Ampliar el programa anterior añadiéndole que los números divisibles por 2 y por 3 son divisibles por 6**
- **Programa:** `divisibilidad-3.pl`

```
divide(2,6).
divide(2,12).
divide(3,6).
divide(3,12).
divide(6,X) :-
   divide(2,X),
   divide(3,X).
```

- **Interpretación de cláusulas**
  - **Cláusula:**
    `divide(6,X) :- divide(2,X), divide(3,X).`
  - **Fórmula:**
    $(\forall \boldsymbol{X})[\boldsymbol{divide}(\mathbf{2}, \boldsymbol{X}) \wedge \boldsymbol{divide}(\mathbf{3}, \boldsymbol{X}) \rightarrow \boldsymbol{divide}(\mathbf{6}, \boldsymbol{X})]$
  - **Interpretación declarativa**
  - **Interpretación procedimental**

Reglas

- **Sesión**
  - **¿Cuáles son los múltiplos de 6?**

```
?- divide(6,X).
X = 6 ;
X = 12 ;
No
```

  - **¿Cuáles son los elementos X e Y tales que X divide a Y?**

```
?- divide(X,Y).
X = 2  Y = 6 ;
X = 2  Y = 12 ;
X = 3  Y = 6 ;
X = 3  Y = 12 ;
X = 6  Y = 6 ;
X = 6  Y = 12 ;
No
```

Resolución en lógica proposicional

- **Programa:** `leche.pl`

```
es_leche :-
   parece_leche,
   lo_da_la_vaca.
parece_leche :-
   es_blanco,
   hay_una_vaca_en_la_etiqueta.
lo_da_la_vaca.
es_blanco.
hay_una_vaca_en_la_etiqueta.
```

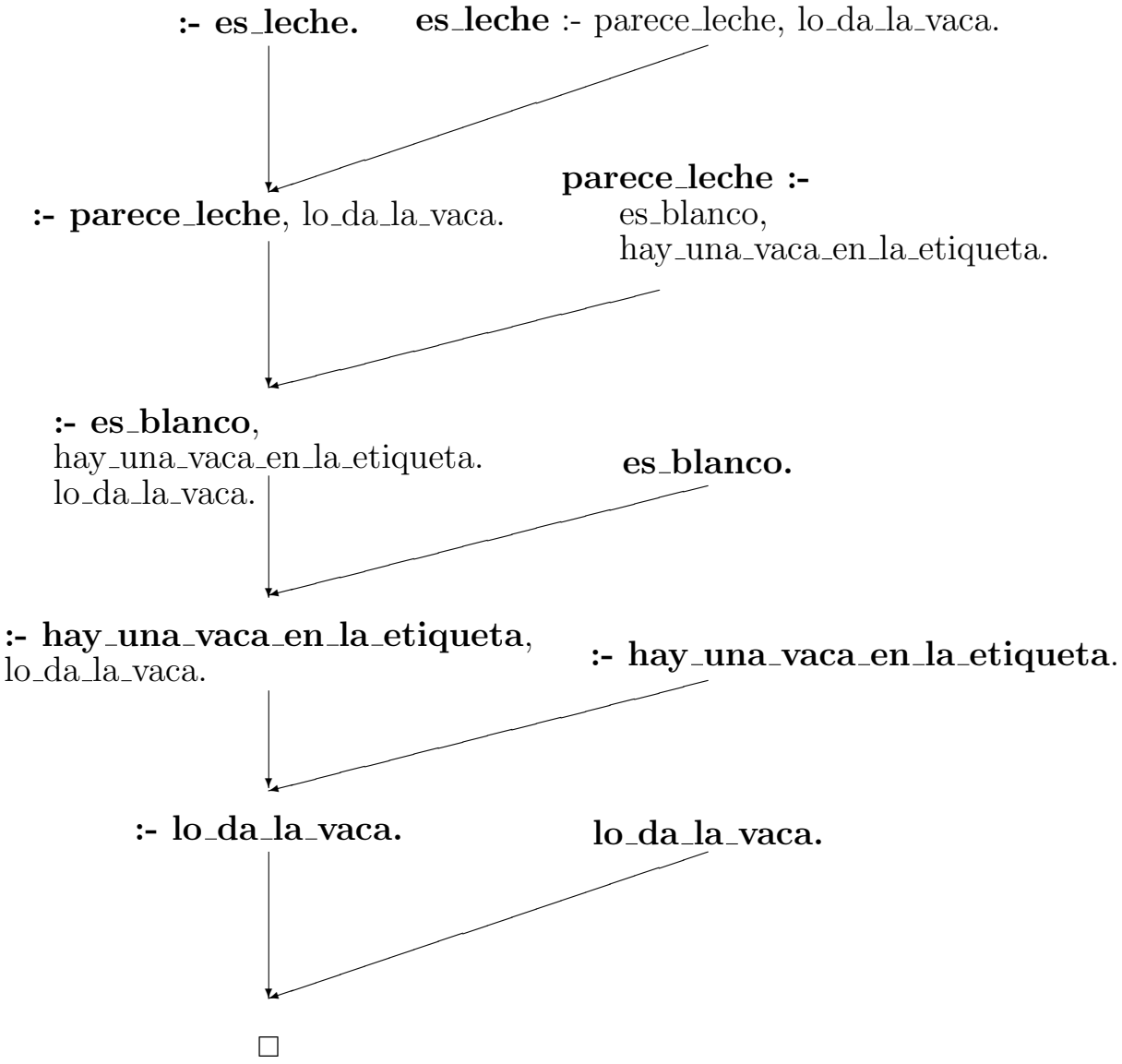
- **Sesión**

```
?- es_leche.
yes
```

- **Traza**

```
(1) 0 CALL es_leche?
(2) 1 CALL parece_leche?
(3) 2 CALL es_blanco?
(3) 2 EXIT es_blanco
(4) 2 CALL hay_una_vaca_en_la_etiqueta?
(4) 2 EXIT hay_una_vaca_en_la_etiqueta
(2) 1 EXIT parece_leche
(5) 1 CALL lo_da_la_vaca?
(5) 1 EXIT lo_da_la_vaca
(1) 0 EXIT es_leche
```

Demostración SLD



- **SLD:**
  - **S: regla de Selección**
  - **L: resolución Lineal**
  - **D: cláusulas Definidas**

Traza

- **Problema: Utilizar el programa anterior para calcular los divisores de 6 con el dispositivo trace y construir el árbol de deducción.**

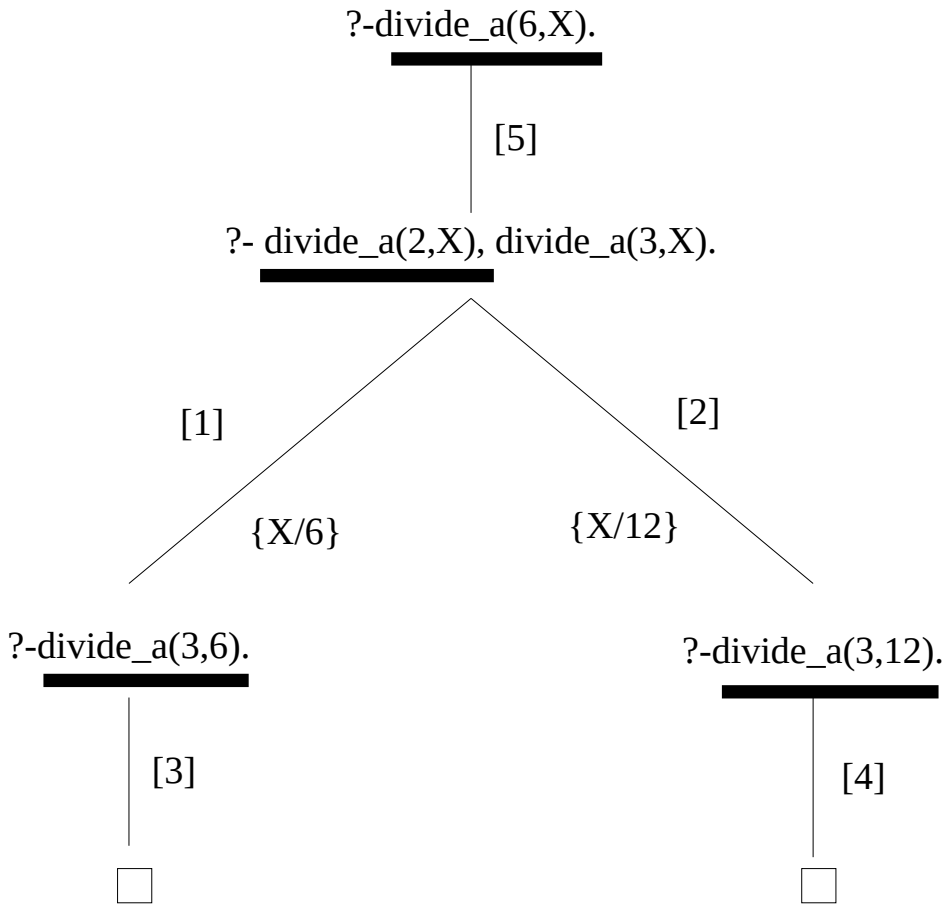
```
?- trace(divide).
        divide/2: call redo exit fail

Yes
?- divide(6,X).
T Call:  (  7) divide(6, _G260)
T Call:  (  8) divide(2, _G260)
T Exit:  (  8) divide(2, 6)
T Call:  (  8) divide(3, 6)
T Exit:  (  8) divide(3, 6)
T Exit:  (  7) divide(6, 6)
X = 6 ;
T Redo:  (  8) divide(3, 6)
T Fail:  (  8) divide(3, 6)
T Redo:  (  8) divide(2, _G260)
T Exit:  (  8) divide(2, 12)
T Call:  (  8) divide(3, 12)
T Exit:  (  8) divide(3, 12)
T Exit:  (  7) divide(6, 12)
X = 12 ;
No
```

Traza

- **Arbol de resolución SLD**

```
[1]     divide(2,6).
[2]     divide(2,12).
[3]     divide(3,6).
[4]     divide(3,12).
[5]     divide(6,X) :-
           divide(2,X),
           divide(3,X).
```



Reglas recursivas: suma.pl

- **Problema: Definir el predicado `suma(X,Y,X)` de forma que si `X` e `Y` son dos números naturales, con la representación basada en `0` (cero) y `s` (sucesor), entonces `Z` es el resultado de sumar `X` e `Y`. Por ejemplo,**

  `suma(s(0),s(s(0)),X) => X=s(s(s(0)))`

- **Programa: `suma.pl`**

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

- **Sesión**
  - **¿Cuál es la suma de `s(0)` y `s(s(0))`?**

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

  - **¿Cuál es la resta de `s(s(s(0)))` y `s(0)`**

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

Reglas recursivas: `suma.pl`

- **¿Cuáles son las soluciones de la ecuación**

  `X + Y = 2`**?**

```
?- suma(X,Y,s(s(0))).
X = 0
Y = s(s(0)) ;
X = s(0)
Y = s(0) ;
X = s(s(0))
Y = 0 ;
No
```

- **¿Cuáles son las soluciones del sistema de ecuaciones**

  `X + Y = 5`

  `Y + 1 = X`**?**

```
?- suma(X,Y,s(s(s(s(s(0)))))), suma(Y,s(0),X).
X = s(s(s(0)))
Y = s(s(0)) ;
No
```

- **¿Cuáles son las soluciones de la ecuación**

  `X + Y = Z`**?**

```
?- suma(X,Y,Z).
X = 0       Y = _G110  Z = _G110 ;
X = s(0)    Y = _G110 Z = s(_G110) ;
X = s(s(0)) Y = _G110 Z = s(s(_G110))
Yes
```

Representación de listas

- **Símbolos:**
  - **Una constante:** []
  - **Un símbolo de función binario:** .
- **Ejemplos de listas (notación de términos)**
  - []
  - .(a,[])
  - .(1,.(2,.(3,.(4,[]))))
  - .([],[])
- **Ejemplos de listas (notación reducida)**
  - []
  - [a]
  - [1,2,3,4]
  - [[]]

Representación de listas

- **El predicado `display`**

```
?- display([]).
[]
Yes
?- display([a]).
.(a, [])
Yes
?- display([1,2,3,4]).
.(1, .(2, .(3, .(4, []))))
Yes
?- display([[]]).
.([], [])
Yes
```

- **El predicado de unificación: =**

```
?- X = .(a,.(1,[])).
X = [a, 1]
Yes
?- .(X,Y) = [1].
X = 1
Y = []
Yes
?- .(X,Y) = [a,b,c].
X = a
Y = [b, c]
Yes
?- .(X,.(2,.(Z,[]))) = [1,Y,3].
X = 1
Z = 3
Y = 2
Yes
```

Unificación con listas

- **¿Son unificables las siguientes listas?**
  - `[X|Y]` **y** `[1,2,3]`

```
?- [X|Y] = [1,2,3].
X = 1
Y = [2, 3]
Yes
```

  - `[X,Y|Z]` **y** `[1,2,3]`

```
?- [X,Y|Z] = [1,2,3].
X = 1
Y = 2
Z = [3]
Yes
```

  - `[X,Y]` **y** `[1,2,3]`

```
?- [X,Y] = [1,2,3].
No
```

  - `[X,Y|Z]` **y** `[1,2]`

```
?- [X,Y|Z] = [1,2].
X = 1
Y = 2
Z = []
Yes
```

Unificación con listas

- `[X,a,b|[]]` **y** `[[b,L],a|L]`

```
?- [X,a,b|[]] = [[b,L],a|L].
X = [b, [b]]
L = [b]
Yes
```

- `[X,Y,Z]` **y** `[Y,Z,X]`

```
?- [X,Y,Z] = [Y,Z,X].
X = _G171
Y = _G171
Z = _G171
Yes
```

- `[X,Y,Z]` **y** `[Y,Z|[]]`

```
?- [X,Y,Z] = [Y,Z|[]].
No
```

- `p([X|R],h(X,[a|[R|L]]))` **y** `p([a],h(a,[L]))`

```
?- p([X|R],h(X,[a|[R|L]])) = p([a],h(a,[L])).
No
```

Operaciones con listas

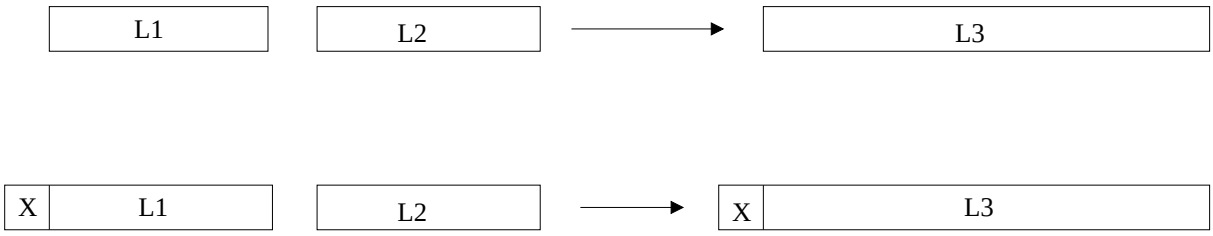
- **Concatenación de listas**
  - `conc(L1,L2,L3)` **se verifica si** `L3` **es la lista obtenida escribiendo los elementos de** `L2` **a continuación de los elementos de** `L1`**.**
  - **Ejemplo**

```
conc([a,b],[c,d],L3) => L3 = [a,b,c,d]
```

  - **Programa** `conc.pl`

```
conc([],L,L).
conc([X|L1],L2,[X|L3]) :-
   conc(L1,L2,L3).
```

  - **Esquema**



Operaciones con listas

- **¿Cuál es el resultado de concatenar las listas `[a,b]` y `[c,d,e]`?**

```
?- conc([a,b],[c,d,e],L).
L = [a, b, c, d, e] ;
No
```

- **¿Qué lista hay que añadirle al lista `[a,b]` para obtener `[a,b,c,d]`?.**

```
?- conc([a,b],L,[a,b,c,d]).
L = [c, d] ;
No
```

- **¿Qué dos listas hay que concatenar para obtener `[a,b]`?.**

```
?- conc(L1,L2,[a,b]).
L1 = []
L2 = [a, b] ;
L1 = [a]
L2 = [b] ;
L1 = [a, b]
L2 = [] ;
No
```

Operaciones con listas

- **¿Pertenece `b` a la lista `[a,b,c]`?**

```
?- conc(L1,[b|L2],[a,b,c]).
L1 = [a]
L2 = [c] ;
No
?- conc(_,[b|_],[a,b,c]).
Yes
```

- **¿Es `[b,c]` una sublista de `[a,b,c,d]`?**

```
?- conc(_,[b,c|_],[a,b,c,d]).
Yes
```

- **¿Es `[b,d]` una sublista de `[a,b,c,d]`?**

```
?- conc(_,[b,d|_],[a,b,c,d]).
No
```

- **¿Cuál es el último elemento de `[b,a,c,d]`?.**

```
?- conc(_,[X],[b,a,c,d]).
X = d ;
No
```

- **Predicado predefinido:** `append(L1,L2,L3)`

Aritmética

- **Operadores prefijos e infijos**
  - `(a + b) * (5 / c)`

```
?- display((a + b) * (5 / c)).
*(+(a, b), /(5, c))
Yes
```

  - `a + b * 5 / c`

```
?- display(a + b * 5 / c).
+(a, /(*(b, 5), c))
Yes
```

- **Precedencia y tipo de operadores predefinidos:**

Precedencia	Tipo	Operadores
500	yfx	+, -
500	fx	-
400	yfx	*, /
200	xfy	^

  - `fx`, `fy`: **Prefijo**
  - `xfx`: **Infijo no asociativo**
  - `yfx`: **Infijo asocia por la izquierda**
  - `xfy`: **Infijo asocia por la derecha**
  - `xf`, `yf`: **Postfijo**

Aritmética

- **Análisis de expresiones con operadores**

```
?- display(2+3+4).
+(+(2, 3), 4)
Yes

?- display(2+3*4).
+(2, *(3, 4))
Yes

?- display((2+3)*4).
*(+(2, 3), 4)
Yes

?- display(2^3^4).
^(2, ^(3, 4))
Yes
```

Predicados aritméticos

- **Evaluador:** `is`
- **Predicados aritméticos:**

Precedencia	Tipo	Operadores
700	xfx	<, =<, >, >=, =:=, =\=

- **Ejemplos**

```
?- X is 2+3^3.
X = 29
Yes
?- 29 is X+3^3.
[WARNING: Arguments are not sufficiently instantiated]
?- X = 2+3^3.
X = 2+3^3
Yes
?- 2+3^Y = 2+3^3.
Y = 3
Yes
?- 3 =< 5.
Yes
?- 3 > X.
[WARNING: Arguments are not sufficiently instantiated]
?- 2+5 = 10-3.
No
?- 2+5 =:= 10-3.
Yes
?- 2+5 =\= 10-3.
No
?- 2+5 =\= 10^3.
Yes
```

Factorial

- **Factorial de un número**
  - `factorial(X,Y)` **se verifica si** `Y` **es el factorial de** `X`
  - **Programa:** `factorial.pl`

```
factorial(1,1).
factorial(X,Y) :-
   X > 1,
   X1 is X - 1,
   factorial(X1,Y1),
   Y is X * Y1.
```

  - **Sesión**

```
?- factorial(4,Y).
Y = 24
Yes
```

  - **Cálculo de** `factorial(4,Y)`

```
X = 4
X1 is X-1 => X1 = 3
factorial(X1,Y1) => Y1 = 6
Y is X*Y1 => Y = 24
```

Longitud

- **Longitud de una lista**
  - `longitud(L,N)` **se verifica si** `N` **es la longitud de la lista** `L`
  - **Ejemplos**

```
longitud([],N)          =>  N = 0
longitud([a,b,c],N)     =>  N = 3
longitud([a,[b,c]],N)   =>  N = 2
```

  - **Programa:** `longitud.pl`

```
longitud([],0).
longitud([_X|L],N) :-
   N is M + 1.
```

  - **Sesión**

```
?- longitud([a,b,c],N).
N = 3
Yes
```

  - **Predicado predefinido:** `length(L,N)`

Todas las soluciones y negación

• Todas las soluciones con `findall`

```
?- append(L1,L2,[a,b]).
L1 = []       L2 = [a, b] ;
L1 = [a]      L2 = [b] ;
L1 = [a, b]   L2 = [] ;
No

?- findall(L1+L2,append(L1,L2,[a,b]),L).
L1 = _G259
L2 = _G260
L = [[]+[a, b], [a]+[b], [a, b]+[]]
Yes

?- findall(N+L1+L2,(append(L1,L2,[a,b]),length(L1,N)),L).
N = _G361
L1 = _G362
L2 = _G365
L = [0+[]+[a, b], 1+[a]+[b], 2+[a, b]+[]]
Yes
```

• Negación

```
?- member(X,[a,b,c,e]),not(member(X,[b,c,d])).
X = a ;
X = e ;
No

?- findall(X,(member(X,[a,b,e]),not(member(X,[b,d]))),L).
X = _G400
L = [a, e]
Yes
```

Bibliografía

- Alonso, J.A. y Gutiérrez, M.A. *Programación declarativa* (`http://www-cs.us.es/ jalonso/pd`)
- Bratko, I. *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
- Clocksin, W.F. y Mellish, C.S. *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
- Luger, G.F. y Stubblefield, W.A. *Artificial Intelligence (Structures and Strategies for Complex Problem Solving (3 edition)* (Addison–Wesley, 1997)
  - Cap. 9 "An introduction to Prolog"
- Kowalski, R. *Lógica, programación e inteligencia artificial"* (Díaz de Santos, 1979)

Bibliografía

- **Rich, E. y Knight, K. *Inteligencia artificial (segunda edición)* (McGraw–Hill Interamericana, 1994)**
  - **Cap. 5 "La lógica de predicados"**
  - **Cap. 6 "Representación del conocimiento mediante reglas"**
- **Russell, S. y Norvig, P. *Inteligencia artificial: un enfoque moderno* (Prentice–Hall Hispanoamericana, 1996)**
  - **Cap. 10: "Sistemas de razonamiento lógico"**