Temas de “Programación lógica e I.A.”

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 21 de febrero de 2013

Índice general

Capítulo 1

El sistema deductivo de Prolog

Programación lógica (2008–09)

Tema 1: El sistema deductivo de Prolog

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 1: El sistema deductivo de Prolog

Objetivos del curso

- Lógica como:
  - sistema de especificación y
  - lenguaje de programación
- Principios:
  - Programas = Teorías
  - Ejecución = Búsqueda de pruebas
  - Programación = Modelización
- Prolog = Programming in Logic
- Relaciones con otros campos:
  - Inteligencia artificial
  - Sistemas basados en el conocimiento
  - Procesamiento del lenguaje natural

Tema 1: El sistema deductivo de Prolog

Declarativo vs. imperativo

- Paradigmas
  - Imperativo: Se describe *cómo* resolver el problema
  - Declarativo: Se describe *qué* es el problema
- Programas
  - Imperativo: Una sucesión de instrucciones
  - Declarativo: Un conjunto de sentencias
- Lenguajes
  - Imperativo: Pascal, C, Fortran
  - Declarativo: Prolog, Lisp puro, ML, Haskell
- Ventajas
  - Imperativo: Programas rápidos y especializados
  - Declarativo: Programas generales, cortos y legibles

Tema 1: El sistema deductivo de Prolog

Historia de la programación lógica

- 1960: Demostración automática de teoremas
- 1965: Resolución y unificación (Robinson)
- 1969: QA3, obtención de respuesta (Green)
- 1972: Implementación de Prolog (Colmerauer)
- 1974: Programación lógica (Kowalski)
- 1977: Prolog de Edimburgo (Warren)
- 1981: Proyecto japonés de Quinta Generación
- 1986: Programación lógica con restricciones
- 1995: Estándar ISO de Prolog

Tema 1: El sistema deductivo de Prolog

Deducción Prolog en lógica proposicional

- Base de conocimiento y objetivo:
  - Base de conocimiento:
    - Regla 1: Si un animal es ungulado y tiene rayas negras, entonces es una cebra.
    - Regla 2: Si un animal rumia y es mamífero, entonces es ungulado.
    - Regla 3: Si un animal es mamífero y tiene pezuñas, entonces es ungulado.
    - Hecho 1: El animal es mamífero.
    - Hecho 2: El animal tiene pezuñas.
    - Hecho 3: El animal tiene rayas negras.
  - Objetivo: Demostrar a partir de la base de conocimientos que el animal es una cebra.

Deducción Prolog en lógica proposicional

- Programa:

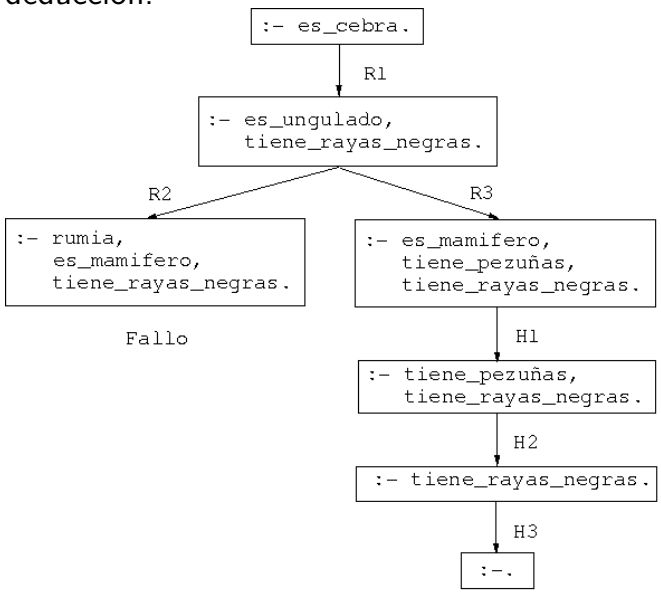
```
es_cebra    :- es_ungulado, tiene_rayas_negras. % R1
es_ungulado :- rumia, es_mamífero.              % R2
es_ungulado :- es_mamífero, tiene_pezuñas.      % R3
es_mamífero.                                    % H1
tiene_pezuñas.                                  % H2
tiene_rayas_negras.                             % H3
```

- Sesión:

```
> pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.20)
Copyright (c) 1990-2006 University of Amsterdam.
?- [animales].
Yes
?- es_cebra.
```

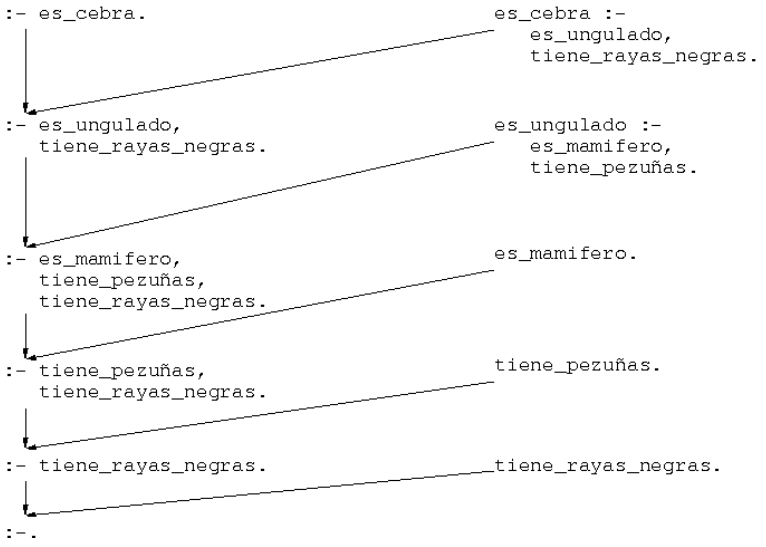

Deducción Prolog en lógica proposicional

- Árbol de deducción:



Deducción Prolog en lógica proposicional

- Demostración por resolución SLD:



Tema 1: El sistema deductivo de Prolog

Deducción Prolog en lógica relacional

- Base de conocimiento:
  - Hechos 1-4: 6 y 12 son divisibles por 2 y por 3.
  - Hecho 5: 4 es divisible por 2.
  - Regla 1: Los números divisibles por 2 y por 3 son divisibles por 6.
- Programa:

```
divide(2,6).                              % Hecho 1
divide(2,4).                              % Hecho 2
divide(2,12).                             % Hecho 3
divide(3,6).                              % Hecho 4
divide(3,12).                             % Hecho 5
divide(6,X) :- divide(2,X), divide(3,X). % Regla 1
```

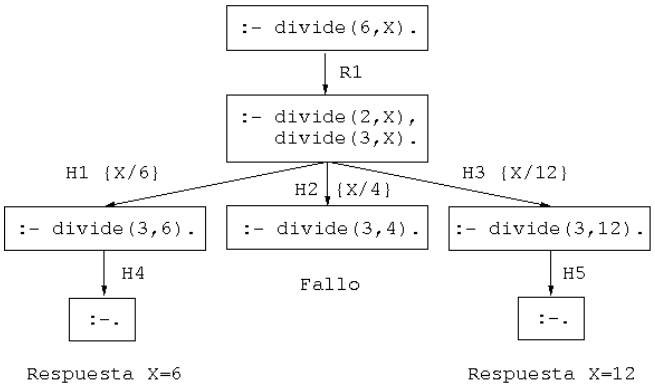

Deducción Prolog en lógica relacional

- Símbolos:
  - Constantes: 2, 3, 4, 6, 12
  - Relación binaria: `divide`
  - Variable: `X`
- Interpretaciones de la Regla 1:
  - `divide(6,X) :- divide(2,X), divide(3,X).`
  - Interpretación declarativa:
    $(\forall X)[divide(2, X) \wedge divide(3, X) \rightarrow divide(6, X)]$
  - Interpretación procedimental.
- Consulta: ¿Cuáles son los múltiplos de 6?

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

Deducción Prolog en lógica relacional

- Árbol de deducción:



- Comentarios:
  - Unificación.
  - Cálculo de respuestas.
  - Respuestas múltiples.

Tema 1: El sistema deductivo de Prolog

Deducción Prolog en lógica funcional

- Representación de los números naturales:
  0, s(0), s(s(0)), ...
- Definición de la suma:

```
0 + Y = Y
s(X) + Y = s(X+Y)
```

- Programa

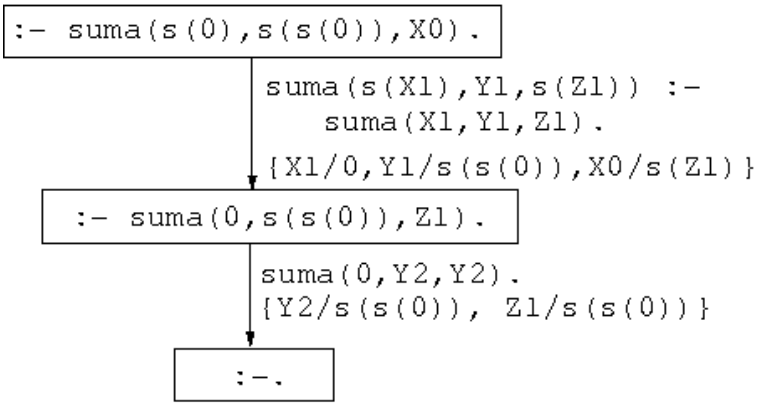
```
suma(0,Y,Y).                          % R1
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).     % R2
```

- Consulta: ¿Cuál es la suma de s(0) y s(s(0))?

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


Deducción Prolog en lógica funcional

- Árbol de deducción:



Resp.: X = X0 = s(Z1) = s(s(s(0)))

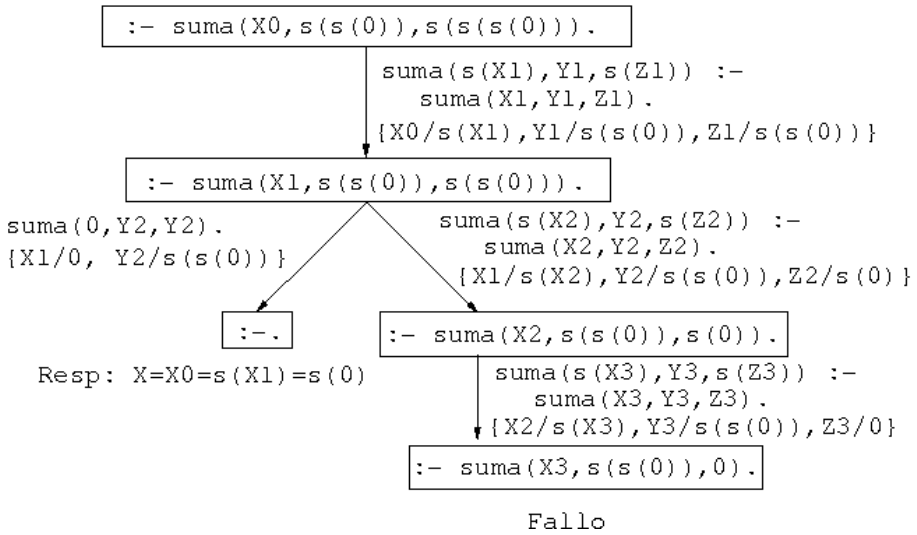
Deducción Prolog en lógica funcional

- Consulta:
  - ¿Cuál es la resta de `s(s(s(0)))` y `s(s(0))`?
  - Sesión:

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


Deducción Prolog en lógica funcional

- Árbol de deducción:



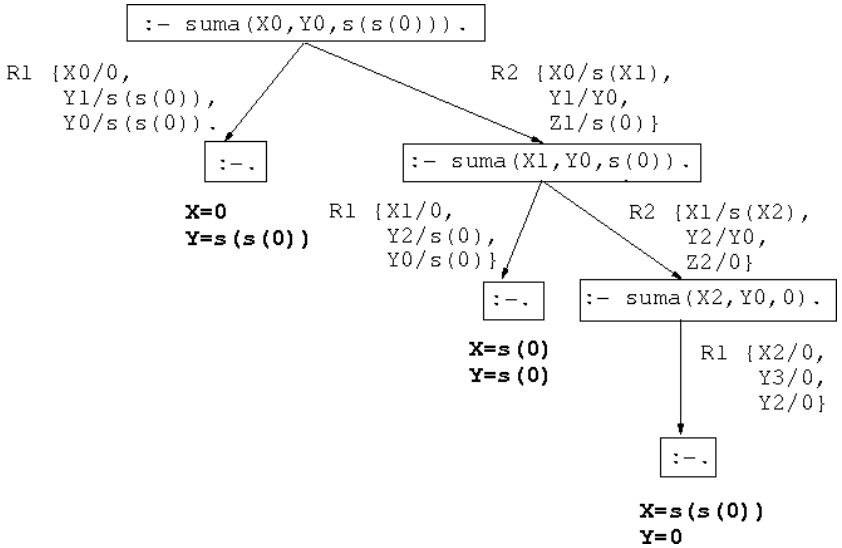
Deducción Prolog en lógica funcional

- Consulta:
  - Pregunta: ¿Cuáles son las soluciones de la ecuación
    `X + Y = s(s(0))`?
  - Sesión:

```
?- 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
```


Deducción Prolog en lógica funcional

- Árbol de deducción:



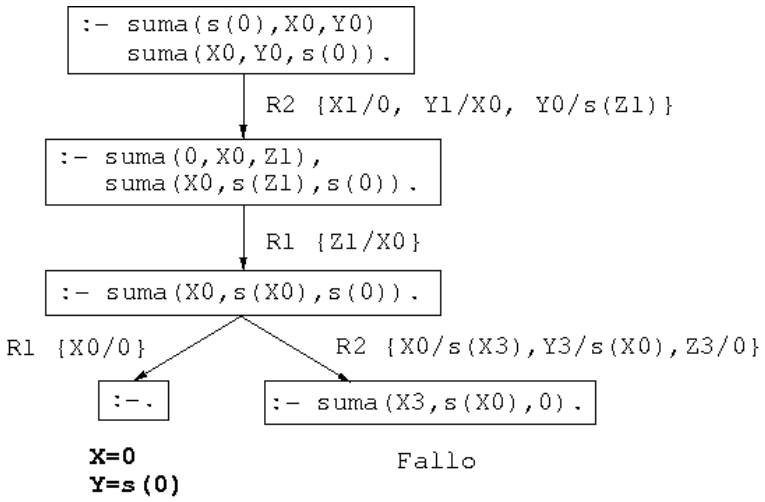
Deducción Prolog en lógica funcional

- Consulta:
  - Pregunta: resolver el sistema de ecuaciones
    $$1 + X = Y$$
    $$X + Y = 1$$
  - Sesión:

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


Deducción Prolog en lógica funcional

- Árbol de deducción:



Bibliografía

1. J.A. Alonso (2006) *Introducción a la programación lógica con Prolog.*
   - Cap. 0: "Introducción".
2. I. Bratko (1990) *Prolog Programming for Artificial Intelligence (2nd ed.)*
   - Cap. 1: "An overview of Prolog".
   - Cap. 2: "Syntax and meaning of Prolog programs".
3. W.F. Clocksin y C.S. Mellish (1994) *Programming in Prolog (Fourth Edition).*
   - Cap. 1: "Tutorial introduction".
   - Cap. 2: "A closer look".

Capítulo 2

Introducción a la programación lógica con Prolog

Programación lógica (2008–09)

Tema 2: Prolog

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 2: Prolog

Construcción de listas

- Definición de listas:
  - La lista vacía `[]` es una lista.
  - Si `L` es una lista, entonces `.(a,L)` es una lista.
- Ejemplos:

```
?- .(X,Y) = [a].
X = a
Y = []
?- .(X,Y) = [a,b].
X = a
Y = [b]
?- .(X,.(Y,Z)) = [a,b].
X = a
Y = b
Z = []
```

Escritura abreviada

- Escritura abreviada:

```
[X|Y] = .(X,Y)
```

- Ejemplos con escritura abreviada:

```
?- [X|Y] = [a,b].
X = a
Y = [b]
?- [X|Y] = [a,b,c,d].
X = a
Y = [b, c, d]
?- [X,Y|Z] = [a,b,c,d].
X = a
Y = b
Z = [c, d]
```


Tema 2: Prolog

Definición de concatenación (append)

- *Especificación:* `conc(A,B,C)` se verifica si `C` es la lista obtenida escribiendo los elementos de la lista `B` a continuación de los elementos de la lista `A`. Por ejemplo,

```
?- conc([a,b],[b,d],C).
C =[a,b,b,d]
```

- *Definición 1:*

```
conc(A,B,C) :- A=[], C=B.
conc(A,B,C) :- A=[X|D], conc(D,B,E), C=[X|E].
```

- *Definición 2:*

```
conc([],B,B).
conc([X|D],B,[X|E]) :- conc(D,B,E).
```


Consultas con la relación de concatenación

- Analogía entre la definición de `conc` y la de `suma`,
- ¿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]
```

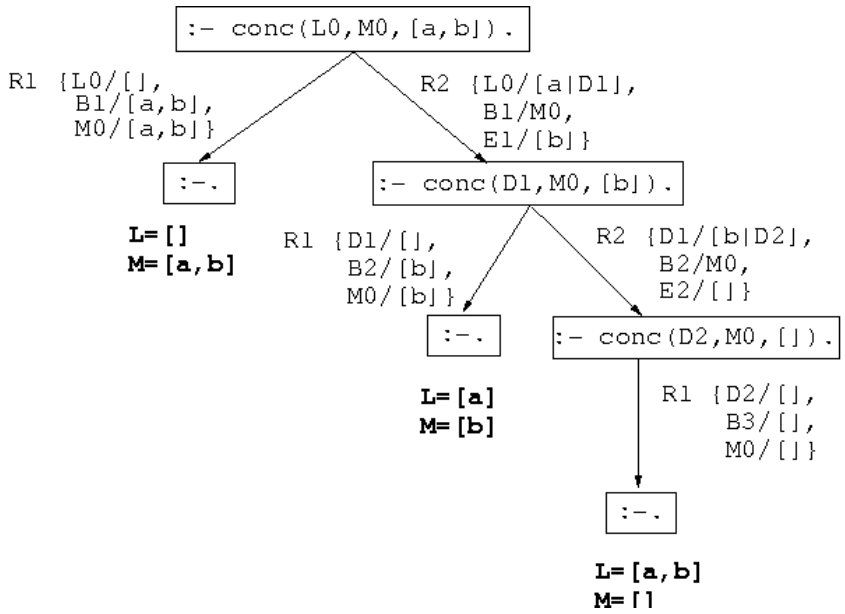
- ¿Qué lista hay que añadirle a la lista `[a,b]` para obtener `[a,b,c,d]`?

```
?- conc([a,b],L,[a,b,c,d]).
L = [c, d]
```

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

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

Árbol de deducción de ?- conc(L,M,[a,b]).



Definición de la relación de pertenencia (member)

- *Especificación:* `pertenece(X,L)` se verifica si `X` es un elemento de la lista `L`.
- *Definición 1:*

```
pertenece(X,[X|L]).
pertenece(X,[Y|L]) :- pertenece(X,L).
```

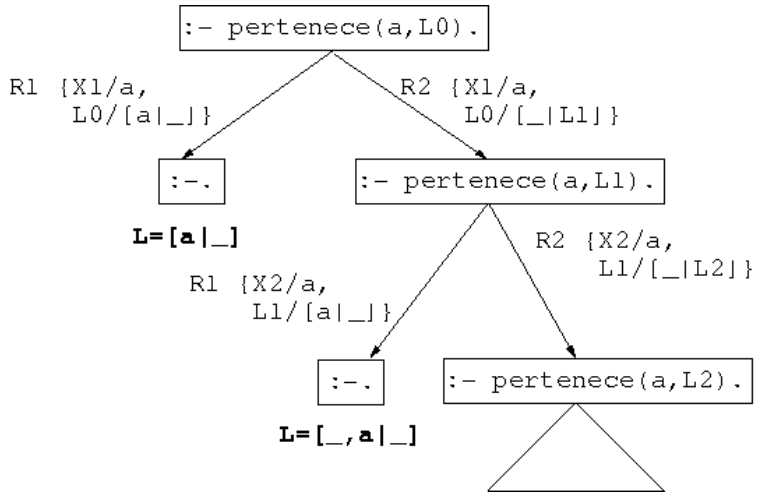
- *Definición 2:*

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

Consultas con la relación de pertenencia

```
?- pertenece(b,[a,b,c]).
Yes
?- pertenece(d,[a,b,c]).
No
?- pertenece(X,[a,b,a]).
X = a ;
X = b ;
X = a ;
No
?- pertenece(a,L).
L = [a|_G233] ;
L = [_G232, a|_G236] ;
L = [_G232, _G235, a|_G239]
Yes
```


Árbol de deducción de ?- pertenece(a,L).



Disyunciones

- Definición de `pertenece` con disyunción

```
pertenece(X,[Y|L]) :- X=Y ; pertenece(X,L).
```

- Definición equivalente sin disyunción

```
pertenece(X,[Y|L]) :- X=Y.
pertenece(X,[Y|L]) :- pertenece(X,L).
```


Tema 2: Prolog

Ejemplos de operadores aritméticos

- Ejemplos de notación infija y prefija en expresiones aritméticas:

```
?- +(X,Y) = a+b.
X = a
Y = b
?- +(X,Y) = a+b+c.
X = a+b
Y = c
?- +(X,Y) = a+(b+c).
X = a
Y = b+c
?- a+b+c = (a+b)+c.
Yes
?- a+b+c = a+(b+c).
No
```


Ejemplos de asociatividad y precedencia

- Ejemplos de asociatividad:

```
?- X^Y = a^b^c.
X = a        Y = b^c
?- a^b^c = a^(b^c).
Yes
```

- Ejemplo de precedencia

```
?- X+Y = a+b*c.
X = a        Y = b*c
?- X*Y = a+b*c.
No
?- X*Y = (a+b)*c.
X = a+b      Y = c
?- a+b*c = (a+b)*c.
No
```

Operadores aritméticos predefinidos

Precedencia	Tipo	Operadores	
500	`yfx`	`+,-`	Infijo asocia por la izquierda
500	`fx`	`-`	Prefijo no asocia
400	`yfx`	`*, /`	Infijo asocia por la izquierda
200	`xfy`	`^`	Infijo asocia por la derecha

Definición de operadores

- Definición (`ejemplo_operadores.pl`)

```
:-op(800,xfx,estudian).
:-op(400,xfx,y).

juan y ana estudian lógica.
```

- Consultas

```
?- [ejemplo_operadores].
?- Quienes estudian lógica.
```

```
Quienes = juan y ana
?- juan y Otro estudian Algo.
```

```
Otro = ana
Algo = lógica
```

Tema 2: Prolog

Evaluación de expresiones aritméticas

- Evaluación de expresiones aritmética con `is`.

```
?- X is 2+3^3.
X = 29
?- X is 2+3, Y is 2*X.
X = 5       Y = 10
```

- Relaciones aritméticas: `<`, `=<`, `>`, `>=`, `=:=` y `=/=`

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

Definición del factorial

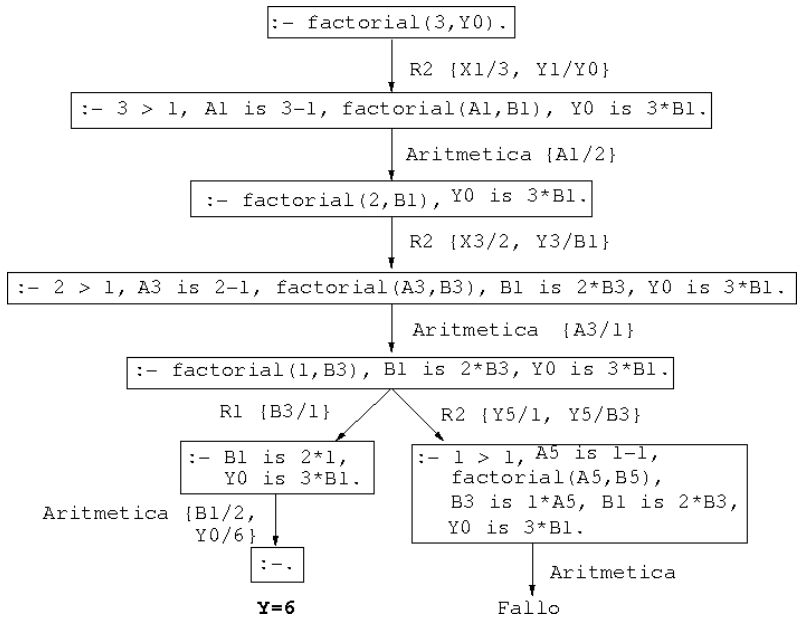
- `factorial(X,Y)` se verifica si `Y` es el factorial de `X`. Por ejemplo,

```
?- factorial(3,Y).
Y = 6 ;
No
```

- Definición:

```
factorial(1,1).
factorial(X,Y) :-
   X > 1,
   A is X - 1,
   factorial(A,B),
   Y is X * B.
```


Árbol de deducción de `?- factorial(3,Y).`



Tema 2: Prolog

Ejemplo de `nota` sin corte

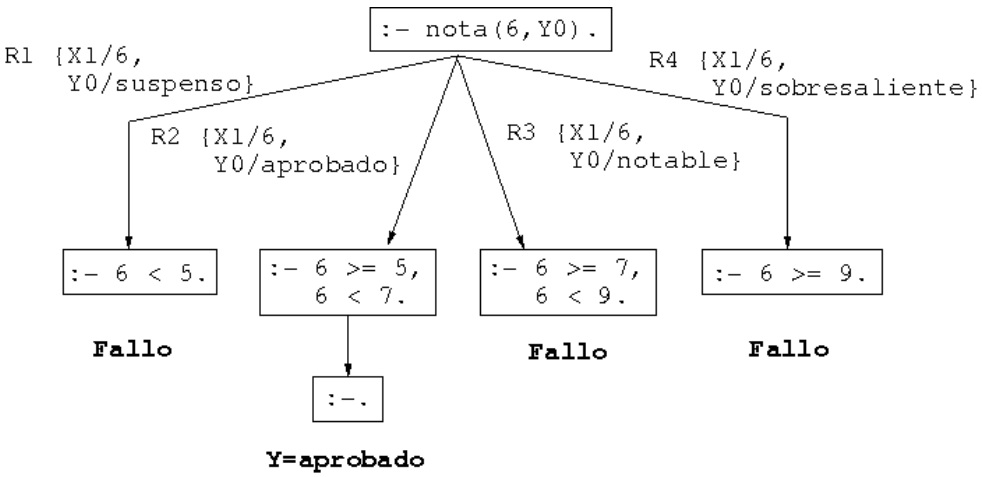
- `nota(X,Y)` se verifica si `Y` es la calificación correspondiente a la nota `X`; es decir, `Y` es `suspenso` si `X` es menor que 5, `Y` es `aprobado` si `X` es mayor o igual que 5 pero menor que 7, `Y` es `notable` si `X` es mayor que 7 pero menor que 9 e `Y` es sobresaliente si `X` es mayor que 9. Por ejemplo,

```
?- nota(6,Y).
Y = aprobado;
No
```

```
nota(X,suspenso)      :- X < 5.
nota(X,aprobado)      :- X >= 5, X < 7.
nota(X,notable)       :- X >= 7, X < 9.
nota(X,sobresaliente) :- X >= 9.
```

Deducción en el ejemplo sin corte

- Árbol de deducción de `?- nota(6,Y).`

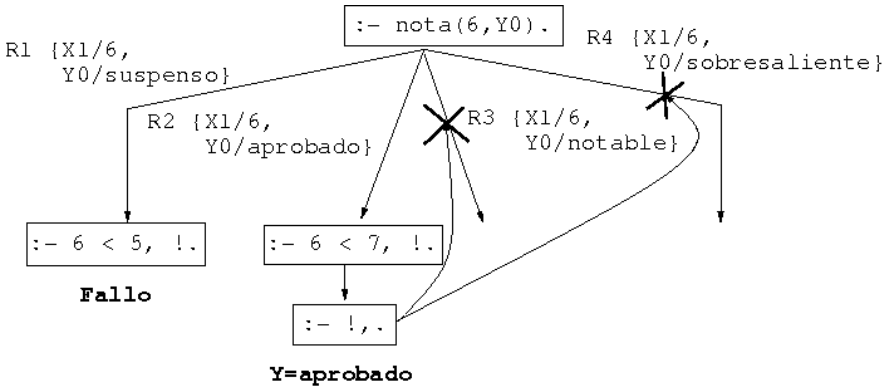


Ejemplo de `nota` con cortes

- Definición de `nota` con cortes

```
nota(X,suspenso)      :- X < 5, !.
nota(X,aprobado)      :- X < 7, !.
nota(X,notable)       :- X < 9, !.
nota(X,sobresaliente).
```

Deducción en el ejemplo con cortes



- ¿Un 6 es un sobresaliente?

```
?- nota(6,sobresaliente).
Yes
```


Uso de corte para respuesta única

- Diferencia entre `member` y `memberchk`

```
?- member(X,[a,b,a,c]), X=a.
X = a ;
X = a ;
No
?- memberchk(X,[a,b,a,c]), X=a.
X = a ;
No
```

- Definición de `member` y `memberchk`:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

memberchk(X,[X|_]) :- !.
memberchk(X,[_|L]) :- memberchk(X,L).
```

Tema 2: Prolog

Definición de la negación como fallo

- Definición de la negación como fallo `not`:

```
no(P) :- P, !, fail.          % No 1
no(P).                        % No 2
```

Programa con negación

- Programa:

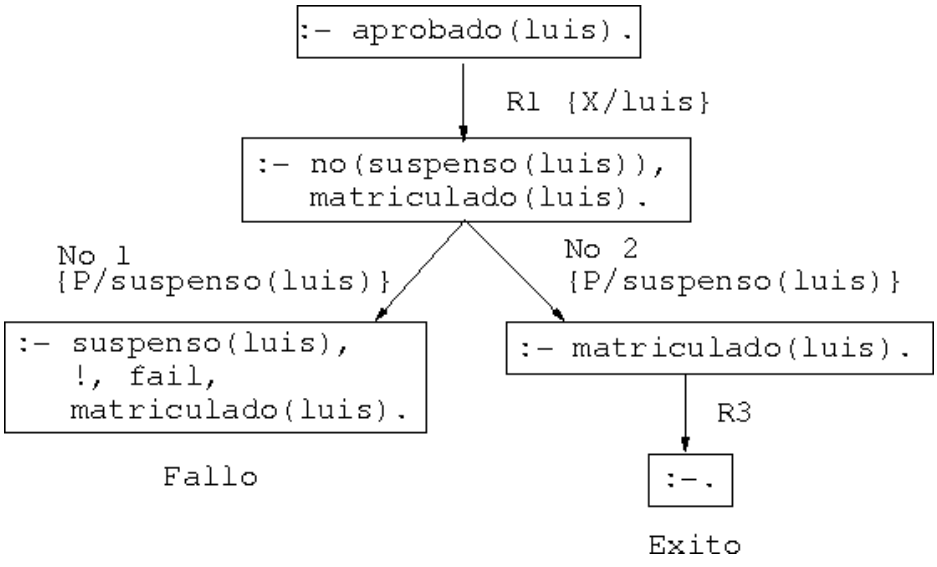
```
aprobado(X) :-                % R1
   no(suspenso(X)),
   matriculado(X).
matriculado(juan).            % R2
matriculado(luis).            % R3
suspenso(juan).               % R4
```

- Consultas:

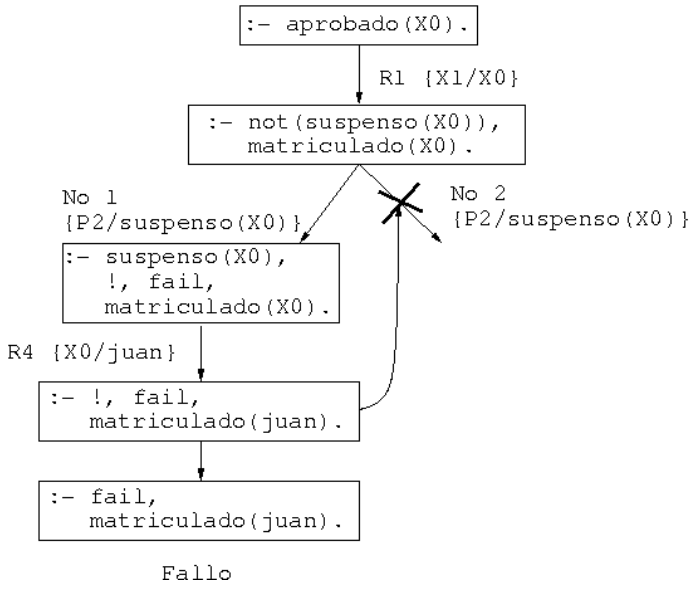
```
?- aprobado(luis).
Yes
```

```
?- aprobado(X).
No
```


Árbol de deducción de `?- aprobado(luis).`



Árbol de deducción de ?- aprobado(X).



Modificación del orden de los literales

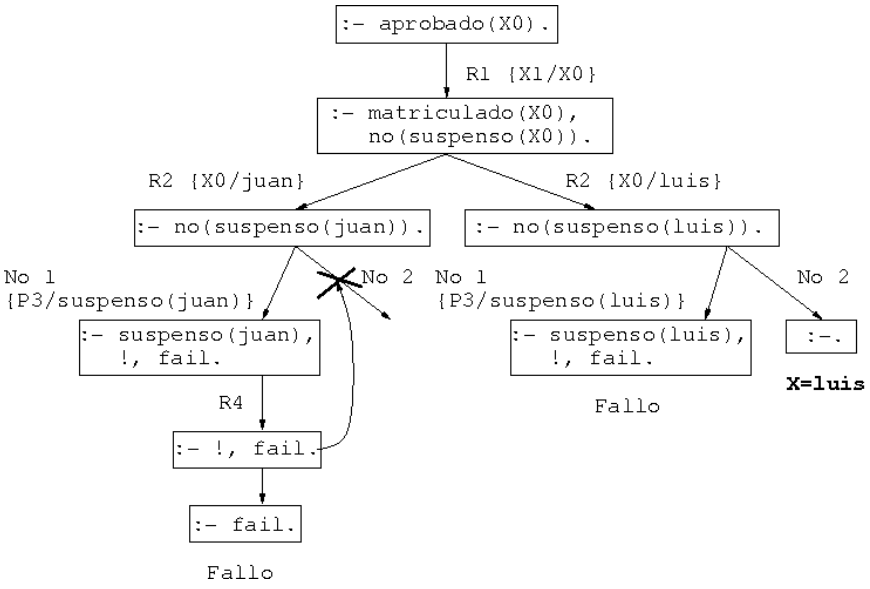
- Programa:

```
aprobado(X) :-          % R1
   matriculado(X),
   no(suspenso(X)).
matriculado(juan).      % R2
matriculado(luis).      % R3
suspenso(juan).         % R4
```

- Consulta:

```
?- aprobado(X).
X = luis
Yes
```

Árbol de deducción de ?- aprobado(X).



Ejemplo de definición con not y con corte

- borra(L1,X,L2) se verifica si L2 es la lista obtenida eliminando los elementos de L1 unificables simultáneamente con X. Por ejemplo,

```
?- borra([a,b,a,c],a,L).
L = [b, c] ;
No
?- borra([a,Y,a,c],a,L).
Y = a
L = [c] ;
No
?- borra([a,Y,a,c],X,L).
Y = a
X = a
L = [c] ;
No
```

Ejemplo de definición con not y con corte

- Definición con `not`:

```
borra_1([],_,[]).
borra_1([X|L1],Y,L2) :-
   X=Y,
   borra_1(L1,Y,L2).
borra_1([X|L1],Y,[X|L2]) :-
   not(X=Y),
   borra_1(L1,Y,L2).
```


Ejemplo de definición con not y con corte

- Definición con corte:

```
borra_2([],_,[]).
borra_2([X|L1],Y,L2) :-
   X=Y, !,
   borra_2(L1,Y,L2).
borra_2([X|L1],Y,[X|L2]) :-
   % not(X=Y),
   borra_2(L1,Y,L2).
```

Ejemplo de definición con `not` y con corte

- Definición con corte y simplificada

```
borra_3([],_,[]).
borra_3([X|L1],X,L2) :-
   !,
   borra_3(L1,Y,L2).
borra_3([X|L1],Y,[X|L2]) :-
   % not(X=Y),
   borra_3(L1,Y,L2).
```


Tema 2: Prolog

Definición de nota con el condicional

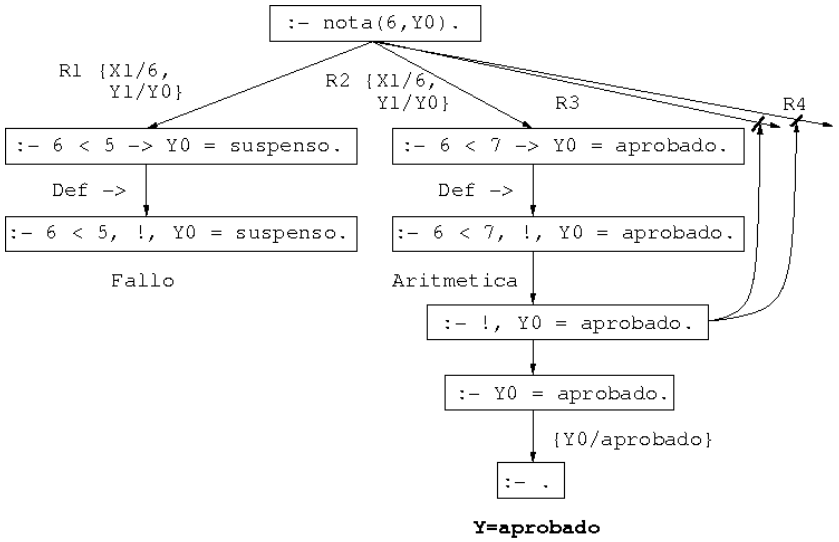
- Definición de nota con el condicional:

```
nota(X,Y) :-
   X < 5 -> Y = suspenso ;          % R1
   X < 7 -> Y = aprobado ;          % R2
   X < 9 -> Y = notable ;           % R3
   true  -> Y = sobresaliente.      % R4
```

- Definición del condicional y verdad:

```
P -> Q :- P, !, Q.                  % Def. ->
P -> Q.
true.                               % Def. true
```


Árbol de deducción de ?- nota(6,Y).



Tema 2: Prolog

Predicados sobre tipos de término

- `var(T)` se verifica si `T` es una variable.
- `atom(T)` se verifica si `T` es un átomo.
- `number(T)` se verifica si `T` es un número.
- `compound(T)` se verifica si `T` es un término compuesto.
- `atomic(T)` se verifica si `T` es una variable, átomo, cadena o número.

```
?- var(X1).              =>  Yes
?- atom(átomo).          =>  Yes
?- number(123).          =>  Yes
?- number(-25.14).       =>  Yes
?- compound(f(X,a)).     =>  Yes
?- compound([1,2]).      =>  Yes
?- atomic(átomo).        =>  Yes
?- atomic(123).          =>  Yes
```

Programa con predicados sobre tipos de término

- `suma_segura(X,Y,Z)` se verifica si `X` e `Y` son enteros y `Z` es la suma de `X` e `Y`. Por ejemplo,

```
?- suma_segura(2,3,X).
X = 5
Yes
?- suma_segura(7,a,X).
No
?- X is 7 + a.
% [WARNING: Arithmetic: ‘a’ is not a function]
```

```
suma_segura(X,Y,Z) :-
   number(X),
   number(Y),
   Z is X+Y.
```


Tema 2: Prolog

Relaciones de comparación de términos

- `T1 = T2` se verifica si `T1` y `T2` son unificables.
- `T1 == T2` se verifica si `T1` y `T2` son idénticos.
- `T1 \== T2` se verifica si `T1` y `T2` no son idénticos.

```
?- f(X) = f(Y).
X = _G164
Y = _G164
Yes
?- f(X) == f(Y).
No
?- f(X) == f(X).
X = _G170
Yes
```


Programa con comparación de términos

- `cuenta(A,L,N)` se verifique si `N` es el número de ocurrencias del átomo `A` en la lista `L`. Por ejemplo,

```
?- cuenta(a,[a,b,a,a],N).
N = 3
?- cuenta(a,[a,b,X,Y],N).
N = 1
```

```
cuenta(_,[],0).
cuenta(A,[B|L],N) :-
   A == B, !,
   cuenta(A,L,M),
   N is M+1.
cuenta(A,[B|L],N) :-
   % A \== B,
   cuenta(A,L,N).
```

Relaciones de ordenación de términos

- `T1 @< T2` se verifica si el término `T1` es anterior que `T2` en el orden de términos de Prolog.

```
?- ab @< ac.            =>  Yes
?- 21 @< 123.           =>  Yes
?- 12 @< a.             =>  Yes
?- g @< f(b).           =>  Yes
?- f(b) @< f(a,b).      =>  Yes
?- [a,1] @< [a,3].      =>  Yes
```

- `sort(+L1,-L2)` se verifica si `L2` es la lista obtenida ordenando de manera creciente los distintos elementos de `L1` y eliminando las repeticiones.

```
?- sort([c4,2,a5,2,c3,a5,2,a5],L).
L = [2, a5, c3, c4]
```


Tema 2: Prolog

Relación de transformación entre términos y listas

- `?T =.. ?L` se verifica si `L` es una lista cuyo primer elemento es el functor del término `T` y los restantes elementos de `L` son los argumentos de `T`. Por ejemplo,

```
?- padre(juan,luis) =.. L.
L = [padre, juan, luis]
?- T =.. [padre, juan, luis].
T = padre(juan,luis)
```


Programa con transformación entre términos y listas

- `alarga(+F1,+N,-F2)` se verifica si `F1` y `F2` son figuras del mismo tipo y el tamaño de `F1` es el de `F2` multiplicado por `N`,

```
?- alarga(triángulo(3,4,5),2,F).
F = triángulo(6, 8, 10)
?- alarga(cuadrado(3),2,F).
F = cuadrado(6)
```

```
alarga(Figura1,Factor,Figura2) :-
   Figura1 =.. [Tipo|Arg1],
   multiplica_lista(Arg1,Factor,Arg2),
   Figura2 =.. [Tipo|Arg2].

multiplica_lista([],_,[]).
multiplica_lista([X1|L1],F,[X2|L2]) :-
   X2 is X1*F, multiplica_lista(L1,F,L2).
```

Las relaciones `functor` y `arg`

- `functor(T,F,A)` se verifica si `F` es el functor del término `T` y `A` es su aridad.
- `arg(N,T,A)` se verifica si `A` es el argumento del término `T` que ocupa el lugar `N`.

```
?- functor(g(b,c,d),F,A).
F = g
A = 3
?- functor(T,g,2).
T = g(_G237,_G238)
?- arg(2,g(b,c,d),X).
X = c
?- functor(T,g,3),arg(1,T,b),arg(2,T,c).
T = g(b, c, _G405)
```


Tema 2: Prolog

Relación de transformación entre átomos y listas: name

- `name(A,L)` se verifica si `L` es la lista de códigos ASCII de los caracteres del átomo `A`. Por ejemplo,

```
?- name(bandera,L).
L = [98, 97, 110, 100, 101, 114, 97]
?- name(A,[98, 97, 110, 100, 101, 114, 97]).
A = bandera
```


Programa con transformación entre átomos y listas

- `concatena_átomos(A1,A2,A3)` se verifica si `A3` es la concatenación de los átomos `A1` y `A2`. Por ejemplo,

```
?- concatena_átomos(pi,ojo,X).
X = piojo
```

```
concatena_átomos(A1,A2,A3) :-
   name(A1,L1),
   name(A2,L2),
   append(L1,L2,L3),
   name(A3,L3).
```

Tema 2: Prolog

La relación apply

- `apply(T,L)` se verifica si es demostrable `T` después de aumentar el número de sus argumentos con los elementos de `L`; por ejemplo,

```
plus(2,3,X).                          =>  X=5
apply(plus,[2,3,X]).                  =>  X=5
apply(plus(2),[3,X]).                 =>  X=5
apply(plus(2,3),[X]).                 =>  X=5
apply(append([1,2]),[X,[1,2,3]]).     =>  X=[3]
```

```
n_apply(Término,Lista) :-
   Término =.. [Pred|Arg1],
   append(Arg1,Lista,Arg2),
   Átomo =.. [Pred|Arg2],
   Átomo.
```

Tema 2: Prolog

La relación `maplist`

- `maplist(P,L1,L2)` se verifica si se cumple el predicado `P` sobre los sucesivos pares de elementos de las listas `L1` y `L2`; por ejemplo,

```
?- succ(2,X).                  =>  3
?- succ(X,3).                  =>  2
?- maplist(succ,[2,4],[3,5]).  =>  Yes
?- maplist(succ,[0,4],[3,5]).  =>  No
?- maplist(succ,[2,4],Y).      =>  Y = [3,5]
?- maplist(succ,X,[3,5]).      =>  X = [2,4]
```

```
n_maplist(_,[],[]).
n_maplist(R,[X1|L1],[X2|L2]) :-
   apply(R,[X1,X2]),
   n_maplist(R,L1,L2).
```

Tema 2: Prolog

Lista de soluciones (`findall`)

- `findall(T,O,L)` se verifica si `L` es la lista de las instancias del término `T` que verifican el objetivo `O`.

```
?- assert(clase(a,voc)), assert(clase(b,con)),
   assert(clase(e,voc)), assert(clase(c,con)).
?- findall(X,clase(X,voc),L).
X = _G331   L = [a, e]
?- findall(_X,clase(_X,_Clase),L).
L = [a, b, e, c]
?- findall(X,clase(X,vocal),L).
X = _G355   L = []
?- findall(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
X = _G373   L = [c, b, c]
?- findall(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
X = _G373   L = []
```

Conjunto de soluciones (setof)

- `setof(T,O,L)` se verifica si `L` es la lista ordenada sin repeticiones de las instancias del término `T` que verifican el objetivo `O`.

```
?- setof(X,clase(X,Clase),L).
X = _G343   Clase = voc   L = [a, e] ;
X = _G343   Clase = con   L = [b, c] ;
No
?- setof(X,Y^clase(X,Y),L).
X = _G379   Y = _G380   L = [a, b, c, e]
?- setof(X,clase(X,vocal),L).
No
?- setof(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
X = _G361   L = [b, c]
?- setof(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
No
```


Multiconjunto de soluciones (bagof)

- `bagof(T,O,L)` se verifica si `L` es el multiconjunto de las instancias del término `T` que verifican el objetivo `O`.

```
?- bagof(X,clase(X,Clase),L).
X = _G343   Clase = voc   L = [a, e] ;
X = _G343   Clase = con   L = [b, c] ;
No
?- bagof(X,Y^clase(X,Y),L).
X = _G379   Y = _G380   L = [a, b, e, c]
?- bagof(X,clase(X,vocal),L).
No
?- bagof(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
X = _G361   L = [c, b, c]
?- bagof(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
No
```

Bibliografía

1. J.A. Alonso *Introducción a la programación lógica con Prolog.*
2. I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)*
3. T. Van Le *Techniques of Prolog Programming*
4. W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)

Capítulo 3

Programación con Prolog

Programación lógica (2008–09)

Tema 3: Programación con Prolog

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Acumuladores

- `inversa(+L1,-L2)`, `reverse(L1,L2)`, se verifica si `L2` es la lista inversa de `L1`. Por ejemplo,

```
?- inversa([a,b,c],L).
L = [c, b, a]
```

- Definición de `inversa` con `append` (no recursiva final):

```
inversa_1([],[]).
inversa_1([X|L1],L2) :-
   inversa_1(L1,L3),
   append(L3,[X],L2).
```


Acumuladores

- Definición de `inversa` con acumuladores (recursiva final):

```
inversa_2(L1,L2) :-
   inversa_2_aux(L1,[],L2).

inversa_2_aux([],L,L).
inversa_2_aux([X|L],Acum,L2) :-
   inversa_2_aux(L,[X|Acum],L2).
```

Comparación de eficiencia

```
?- findall(_N,between(1,1000,_N),_L1),
   time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).
501,501 inferences in 0.40 seconds
  1,002 inferences in 0.00 seconds

?- findall(_N,between(1,2000,_N),_L1),
   time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).
2,003,001 inferences in 1.59 seconds
    2,002 inferences in 0.00 seconds

?- findall(_N,between(1,4000,_N),_L1),
   time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).
8,006,001 inferences in 8.07 seconds
    4,002 inferences in 0.02 seconds
```


Combinaciones

- combinación(+L1,+N,-L2) se verifica si L2 es una combinación N–aria de L1. Por ejemplo,

```
?- combinación([a,b,c],2,L).
L = [a, b] ;   L = [a, c] ;   L = [b, c] ;  No
```

```
combinación_1(L1,N,L2) :-
   subconjunto(L2,L1),
   length(L2,N).

combinación_2(L1,N,L2) :-
   length(L2,N),
   subconjunto(L2,L1).

combinación(L1,N,L2) :-
   combinación_2(L1,N,L2).
```

Combinaciones

- `combinaciones(+L1,+N,-L2)` se verifica si `L2` es la lista de las combinaciones `N`–arias de `L1`. Por ejemplo,

```
?- combinaciones([a,b,c],2,L).
L = [[a, b], [a, c], [b, c]]
```

```
combinaciones_1(L1,N,L2) :-
   findall(L,combinación_1(L1,N,L),L2).

combinaciones_2(L1,N,L2) :-
   findall(L,combinación_2(L1,N,L),L2).

combinaciones(L1,N,L2) :-
   combinaciones_2(L1,N,L2).
```


Comparación de eficiencia de combinaciones

```
?- findall(_N,between(1,6,_N),_L1),
   time(combinaciones_1(_L1,2,_L2)),
   time(combinaciones_2(_L1,2,_L2)).
429 inferences in 0.00 seconds
 92 inferences in 0.00 seconds
?- findall(_N,between(1,12,_N),_L1),
   time(combinaciones_1(_L1,2,_L2)),
   time(combinaciones_2(_L1,2,_L2)).
28,551 inferences in 0.01 seconds
   457 inferences in 0.00 seconds
?- findall(_N,between(1,24,_N),_L1),
   time(combinaciones_1(_L1,2,_L2)),
   time(combinaciones_2(_L1,2,_L2)).
117,439,971 inferences in 57.59 seconds
      2,915 inferences in  0.00 seconds
```

Permutaciones

- `select(?X,?L1,?L2)` se verifica si `X` es un elemento de la lista `L1` y `L2` es la lista de los restantes elementos. Por ejemplo,

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

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


Permutaciones

- `permutación(+L1,-L2)` se verifica si `L2` es una permutación de `L1`. Por ejemplo,

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

```
permutación([],[]).
permutación(L1,[X|L2]) :-
   select(X,L1,L3),
   permutación(L3,L2).
```

Predefinida `permutation`.

Variaciones

- `variación(+L1,+N,-L2)` se verifica si `L2` es una variación `N`–aria de `L1`. Por ejemplo,

```
?- variación([a,b,c],2,L).
L=[a,b];L=[a,c];L=[b,a];L=[b,c];L=[c,a];L=[c,b];No
```

```
variación_1(L1,N,L2) :-
   combinación(L1,N,L3), permutación(L3,L2).

variación_2(_,0,[]).
variación_2(L1,N,[X|L2]) :-
   N > 0, M is N-1,
   select(X,L1,L3),
   variación_2(L3,M,L2).

variación(L1,N,L2) :- variación_2(L1,N,L2).
```


Variaciones

- `variaciones(+L1,+N,-L2)` se verifica si `L2` es la lista de las variaciones `N`–arias de `L1`. Por ejemplo,

```
?- variaciones([a,b,c],2,L).
L = [[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]
```

```
variaciones_1(L1,N,L2) :-
   setof(L,variación_1(L1,N,L),L2).

variaciones_2(L1,N,L2) :-
   setof(L,variación_2(L1,N,L),L2).

variaciones(L1,N,L2) :-
   variaciones_2(L1,N,L2).
```

Comparación de eficiencia de variaciones

```
?- findall(N,between(1,100,N),L1),
   time(variaciones_1(L1,2,L2)),
   time(variaciones_2(L1,2,L2)).
221,320 inferences in 0.27 seconds
 40,119 inferences in 0.11 seconds
?- findall(N,between(1,200,N),L1),
   time(variaciones_1(L1,2,L2)),
   time(variaciones_2(L1,2,L2)).
1,552,620 inferences in 2.62 seconds
  160,219 inferences in 0.67 seconds
?- findall(N,between(1,400,N),L1),
   time(variaciones_1(L1,2,L2)),
   time(variaciones_2(L1,2,L2)).
11,545,220 inferences in 19.02 seconds
   640,419 inferences in  2.51 seconds
```


Tema 3: Programación con Prolog

Ordenación por generación y prueba

- `ordenación(+L1,-L2)` se verifica si `L2` es la lista obtenida ordenando la lista `L1` en orden creciente. Por ejemplo,

```
?- ordenación([2,1,a,2,b,3],L).
L = [a,b,1,2,2,3]
```

```
ordenación(L,L1) :-
   permutación(L,L1),
   ordenada(L1).
```

```
ordenada([]).
ordenada([_]).
ordenada([X,Y|L]) :-
   X @=< Y,
   ordenada([Y|L]).
```


Ordenación por selección

```
ordenación_por_selección(L1,[X|L2]) :-
   selecciona_menor(X,L1,L3),
   ordenación_por_selección(L3,L2).
ordenación_por_selección([],[]).

selecciona_menor(X,L1,L2) :-
   select(X,L1,L2),
   not((member(Y,L2), Y @< X)).
```

Ordenación por divide y vencerás

```
ordenación_rápida([],[]).
ordenación_rápida([X|R],Ordenada) :-
   divide(X,R,Menores,Mayores),
   ordenación_rápida(Menores,Menores_ord),
   ordenación_rápida(Mayores,Mayores_ord),
   append(Menores_ord,[X|Mayores_ord],Ordenada).

divide(_,[],[],[]).
divide(X,[Y|R],[Y|Menores],Mayores) :-
   Y @< X, !,
   divide(X,R,Menores,Mayores).
divide(X,[Y|R],Menores,[Y|Mayores]) :-
   \% Y @>= X,
   divide(X,R,Menores,Mayores).
```


Ordenación: comparación de eficiencia

Comparación de la ordenación de la lista `[N,N-1,N-2,...,2,1]`

N	ordena	selección	rápida
1	5 inf 0.00 s	8 inf 0.00 s	5 inf 0.00 s
2	10 inf 0.00 s	19 inf 0.00 s	12 inf 0.00 s
4	80 inf 0.00 s	67 inf 0.00 s	35 inf 0.00 s
8	507,674 inf 0.33 s	323 inf 0.00 s	117 inf 0.00 s
16		1,923 inf 0.00 s	425 inf 0.00 s
32		13,059 inf 0.01 s	1,617 inf 0.00 s
64		95,747 inf 0.05 s	6,305 inf 0.00 s
128		732,163 inf 0.40 s	24,897 inf 0.01 s
256		5,724,163 inf 2.95 s	98,945 inf 0.05 s
512		45,264,899 inf 22.80 s	394,497 inf 0.49 s

Tema 3: Programación con Prolog

Cuadrado mágico por generación y prueba

- Enunciado: Colocar los números 1,2,3,4,5,6,7,8,9 en un cuadrado 3x3 de forma que todas las líneas (filas, columnas y diagonales) sumen igual.

A	B	C
D	E	F
G	H	I

```
cuadrado_1([A,B,C,D,E,F,G,H,I]) :-
   permutación([1,2,3,4,5,6,7,8,9],
               [A,B,C,D,E,F,G,H,I]),
   A+B+C =:= 15,  D+E+F =:= 15,
   G+H+I =:= 15,  A+D+G =:= 15,
   B+E+H =:= 15,  C+F+I =:= 15,
   A+E+I =:= 15,  C+E+G =:= 15.
```

Cuadrado mágico por generación y prueba

- Cálculo de soluciones:

```
?- cuadrado_1(L).
L = [6, 1, 8, 7, 5, 3, 2, 9, 4] ;
L = [8, 1, 6, 3, 5, 7, 4, 9, 2]
Yes
```

- Cálculo del número soluciones:

```
?- findall(_X,cuadrado_1(_X),_L),length(_L,N).
N = 8
Yes
```


Cuadrado mágico por comprobaciones parciales

- Programa 2:

```
cuadrado_2([A,B,C,D,E,F,G,H,I]) :-
   select(A,[1,2,3,4,5,6,7,8,9],L1),
   select(B,L1,L2),
   select(C,L2,L3),  A+B+C =:= 15,
   select(D,L3,L4),
   select(G,L4,L5),  A+D+G =:= 15,
   select(E,L5,L6),  C+E+G =:= 15,
   select(I,L6,L7),  A+E+I =:= 15,
   select(F,L7,[H]), C+F+I =:= 15, D+E+F =:= 15.
```

Cuadrado mágico por comprobaciones parciales

- Cálculo de soluciones:

```
?- cuadrado_2(L).
L = [2, 7, 6, 9, 5, 1, 4, 3, 8] ;
L = [2, 9, 4, 7, 5, 3, 6, 1, 8]
Yes
```

- Comprobación que las dos definiciones dan las *mismas* soluciones.

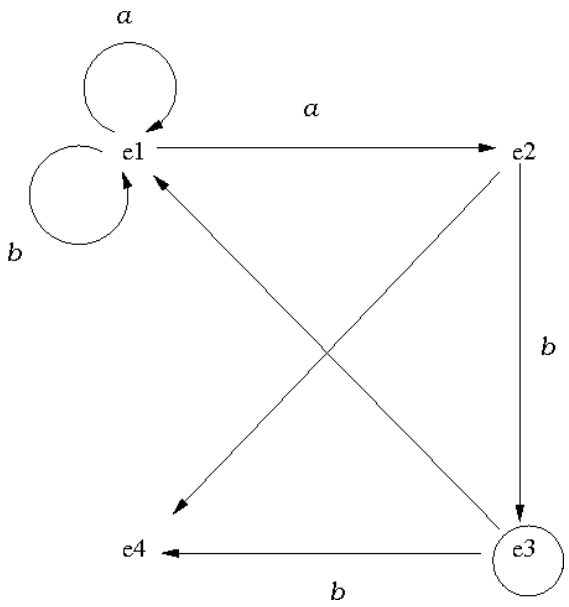
```
?- setof(_X,cuadrado_1(_X),_L),
   setof(_X,cuadrado_2(_X),_L).
Yes
```


Comparación de eficiencia del cuadrado mágico

```
?- time(cuadrado_1(_X)).
161,691 inferences in 0.58 seconds

?- time(cuadrado_2(_X)).
  1,097 inferences in 0.01 seconds

?- time(setof(_X,cuadrado_1(_X),_L)).
812,417 inferences in 2.90 seconds

?- time(setof(_X,cuadrado_2(_X),_L)).
  7,169 inferences in 0.02 seconds
```

Tema 3: Programación con Prolog

Ejemplo de autómata no determinista (con estado final e3)



Representación de un autómata (automata.pl)

- `final(E)` se verifica si `E` es el estado final.

```
final(e3).
```

- `trans(E1,X,E2)` se verifica si se puede pasar del estado `E1` al estado `E2` usando la letra `X`.

```
trans(e1,a,e1).   trans(e1,a,e2).   trans(e1,b,e1).
trans(e2,b,e3).
trans(e3,b,e4).
```

- `nulo(E1,E2)` se verifica si se puede pasar del estado `E1` al estado `E2` mediante un movimiento nulo.

```
nulo(e2,e4).
nulo(e3,e1).
```


Tema 3: Programación con Prolog

Simulación de los autómatas no deterministas

- `acepta(E,L)` se verifica si el autómata, a partir del estado `E` acepta la lista `L`. Por ejemplo,

```
acepta(e1,[a,a,a,b]) => Sí
acepta(e2,[a,a,a,b]) => No
```

```
acepta(E,[]) :-
   final(E).
acepta(E,[X|L]) :-
   trans(E,X,E1),
   acepta(E1,L).
acepta(E,L) :-
   nulo(E,E1),
   acepta(E1,L).
```


Tema 3: Programación con Prolog

Consultas al autómata

- Determinar si el autómata acepta la lista `[a,a,a,b]`

```
?- acepta(e1,[a,a,a,b]).
Yes
```

- Determinar los estados a partir de los cuales el autómata acepta la lista `[a,b]`

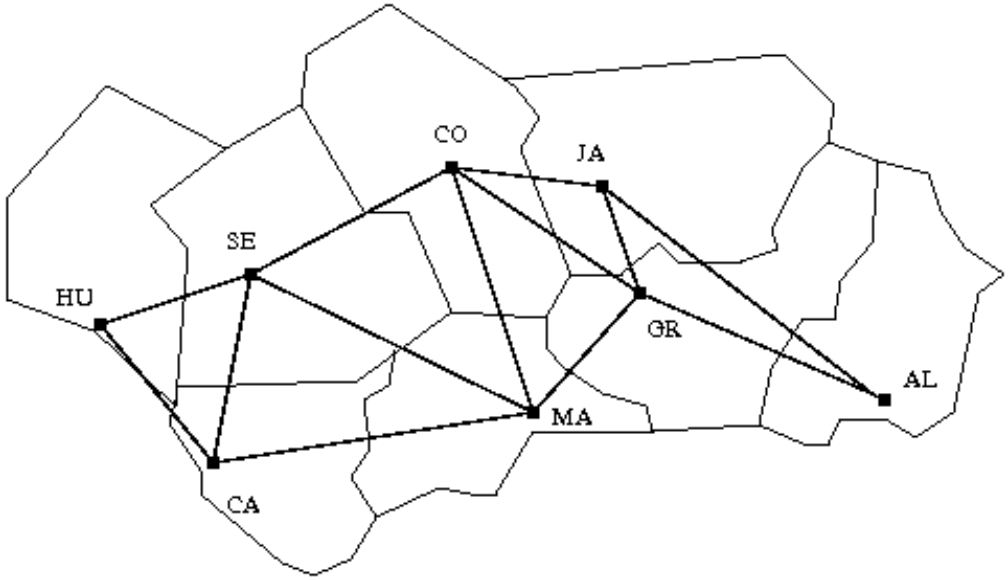
```
?- acepta(E,[a,b]).
E=e1 ;
E=e3 ;
No
```

- Determinar las palabras de longitud 3 aceptadas por el autómata a partir del estado `e1`

```
?- acepta(e1,[X,Y,Z]).
X = a   Y = a   Z = b ;
X = b   Y = a   Z = b ;
No
```


Tema 3: Programación con Prolog

Grafo de Andalucía



Representación del grafo

- `arcos(+L)` se verifica si `L` es la lista de arcos del grafo.

```
arcos([huelva-sevilla,  huelva-cádiz,
       cádiz-sevilla,   sevilla-málaga,
       sevilla-córdoba, córdoba-málaga,
       córdoba-granada, córdoba-jaén,
       jaén-granada,    jaén-almería,
       granada-almería]).
```

Adyacencia y nodos

- `adyacente(?X,?Y)` se verifica si `X` e `Y` son adyacentes.

```
adyacente(X,Y) :-
   arcos(L),
   (member(X-Y,L) ; member(Y-X,L)).
```

- `nodos(?L)` se verifica si `L` es la lista de nodos.

```
nodos(L) :-
   setof(X,Y^adyacente(X,Y),L).
```


Tema 3: Programación con Prolog

Caminos

- `camino(+A,+Z,-C)` se verifica si `C` es un camino en el grafo desde el nodo `A` al `Z`. Por ejemplo,

```
?- camino(sevilla,granada,C).
C = [sevilla, córdoba, granada] ;
C = [sevilla, málaga, córdoba, granada]
Yes
```

```
camino(A,Z,C) :-
   camino_aux(A,[Z],C).
```


Caminos

- `camino_aux(+A,+CP,-C)` se verifica si `C` es una camino en el grafo compuesto de un camino desde `A` hasta el primer elemento del camino parcial `CP` (con nodos distintos a los de `CP`) junto `CP`.

```
camino_aux(A,[A|C1],[A|C1]).
camino_aux(A,[Y|C1],C) :-
   adyacente(X,Y),
   not(member(X,[Y|C1])),
   camino_aux(A,[X,Y|C1],C).
```

Tema 3: Programación con Prolog

Caminos hamiltonianos

- `hamiltoniano(-C)` se verifica si `C` es un camino hamiltoniano en el grafo (es decir, es un camino en el grafo que pasa por todos sus nodos una vez). Por ejemplo,

```
?- hamiltoniano(C).
C = [almería, jaén, granada, córdoba, málaga, sevilla, hu
?- findall(_C,hamiltoniano(_C),_L), length(_L,N).
N = 16
```

- Primera definición de `hamiltoniano`

```
hamiltoniano_1(C) :-
    camino(_,_,C),
    nodos(L),
    length(L,N),
    length(C,N).
```

Caminos hamiltonianos

- Segunda definición de `hamiltoniano`

```
hamiltoniano_2(C) :-
   nodos(L),
   length(L,N),
   length(C,N),
   camino(_,_,C).
```

- Comparación de eficiencia

```
?- time(findall(_C,hamiltoniano_1(_C),_L)).
37,033 inferences in 0.03 seconds (1234433 Lips)
?- time(findall(_C,hamiltoniano_2(_C),_L)).
13,030 inferences in 0.01 seconds (1303000 Lips)
```

Capítulo 4

Resolución de problemas de espacios de estados

Programación lógica (2008–09)

Tema 4: Resolución de problemas de espacios de estados

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Enunciado del problema del 8–puzzle

Para el 8–puzzle se usa un cajón cuadrado en el que hay situados 8 bloques cuadrados. El cuadrado restante está sin rellenar. Cada bloque tiene un número. Un bloque adyacente al hueco puede deslizarse hacia él. El juego consiste en transformar la posición inicial en la posición final mediante el deslizamiento de los bloques. En particular, consideramos el estado inicial y final siguientes:

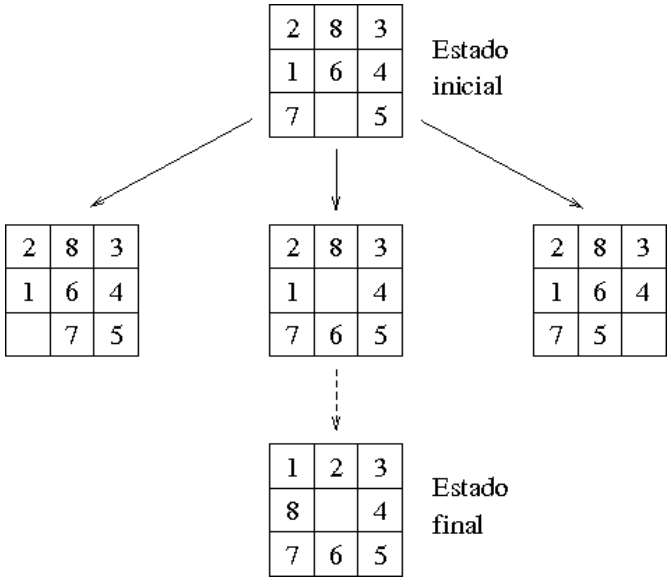
2	8	3
1	6	4
7		5

Estado inicial

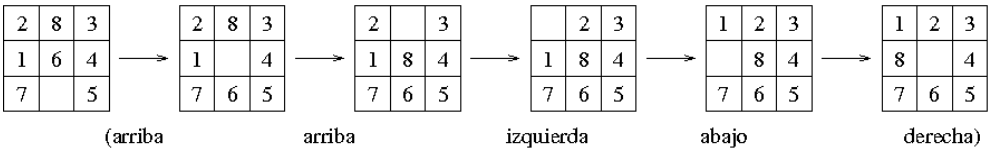
1	2	3
8		4
7	6	5

Estado final

Desarrollo del problema del 8–puzzle



Solución del problema del 8–puzzle



Especificación del problema del 8–puzzle

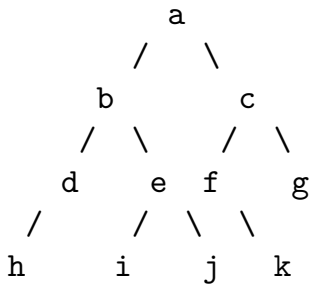
- Estado inicial: `[[2,8,3],[1,6,4],[7,h,5]]`
- Estado final: `[[1,2,3],[8,h,4],[7,6,5]]`
- Operadores:
  - Mover el hueco a la izquierda
  - Mover el hueco arriba
  - Mover el hueco a la derecha
  - Mover el hueco abajo

Número de estados $= 9! = 362.880$.

Tema 4: Resolución de problemas de espacios de estados

Enunciado del problema del árbol

- Árbol



- Estado inicial: `a`
- Estados finales: `f` y `j`

Representación del problema del árbol

Representación `arbol.pl`

- `estado_inicial(?E)` se verifica si `E` es el estado inicial.

```
estado_inicial(a).
```

- `estado_final(?E)` se verifica si `E` es un estado final.

```
estado_final(f).
estado_final(j).
```

- `sucesor(+E1,?E2)` se verifica si `E2` es un sucesor del estado `E1`.

```
sucesor(a,b).   sucesor(a,c).   sucesor(b,d).
sucesor(b,e).   sucesor(c,f).   sucesor(c,g).
sucesor(d,h).   sucesor(e,i).   sucesor(e,j).
sucesor(f,k).
```


Solución del problema del árbol

- `profundidad_sin_ciclos(?S)` se verifica si `S` es una solución del problema mediante búsqueda en profundidad sin ciclos. Por ejemplo,

```
?- [arbol].
?- profundidad_sin_ciclos(S).
S = [a, b, e, j]
?- trace(estado_final,+call), profundidad_sin_ciclos(S).
 T Call:  (9) estado_final(a)
 T Call: (10) estado_final(b)
 T Call: (11) estado_final(d)
 T Call: (12) estado_final(h)
 T Call: (11) estado_final(e)
 T Call: (12) estado_final(i)
 T Call: (12) estado_final(j)
S = [a, b, e, j]
```

Procedimiento de búsqueda en profundidad sin ciclos

```
profundidad_sin_ciclos(S) :-
   estado_inicial(E),
   profundidad_sin_ciclos(E,S).

profundidad_sin_ciclos(E,[E]) :-
   estado_final(E).
profundidad_sin_ciclos(E,[E|S1]) :-
   sucesor(E,E1),
   profundidad_sin_ciclos(E1,S1).
```


El problema de las 4 reinas

- Enunciado: Colocar 4 reinas en un tablero rectangular de dimensiones 4 por 4 de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- Estados: listas de números que representa las ordenadas de las reinas colocadas. Por ejemplo, [3,1] representa que se ha colocado las reinas (1,1) y (2,3).

Solución del problema de las 4 reinas por búsqueda en profundidad sin ciclos

- Soluciones:

```
?- ['4-reinas','b-profundidad-sin-ciclos'].
Yes
?- profundidad_sin_ciclos(S).
S = [[], [2], [4, 2], [1, 4, 2], [3, 1, 4, 2]] ;
S = [[], [3], [1, 3], [4, 1, 3], [2, 4, 1, 3]] ;
No
```


Representación del problema de las 4 reinas

`4-reinas.pl`

- `estado_inicial(?E)` se verifica si `E` es el estado inicial.

```
estado_inicial([]).
```

- `estado_final(?E)` se verifica si `E` es un estado final.

```
estado_final(E) :-
      length(E,4).
```

- `sucesor(+E1,?E2)` se verifica si `E2` es un sucesor del estado `E1`.

```
sucesor(E,[Y|E]) :-
   member(Y,[1,2,3,4]),
   not(member(Y,E)),
   no_ataca(Y,E).
```

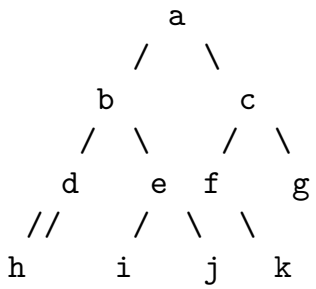
Representación del problema de las 4 reinas

- `no_ataca(Y,E)` se verifica si `E=[Yn,...,Y1]`, entonces la reina colocada en `(n+1,Y)` no ataca a las colocadas en `(1,Y1)`, ..., `(n,Yn)`.

```
no_ataca(Y,E) :-
   no_ataca(Y,E,1).
no_ataca(_,[],_).
no_ataca(Y,[Y1|L],D) :-
   Y1-Y =\= D,
   Y-Y1 =\= D,
   D1 is D+1,
   no_ataca(Y,L,D1).
```


Tema 4: Resolución de problemas de espacios de estados

Enunciado del problema de grafo con ciclos



- Estado inicial: `a`
- Estados finales: `f` y `j`
- Nota: el arco entre `d` y `h` es bidireccional

Representación del problema de grafo con ciclos

`grafo.pl`

- `estado_inicial(?E)` se verifica si `E` es el estado inicial.

```
estado_inicial(a).
```

- `estado_final(?E)` se verifica si `E` es un estado final.

```
estado_final(f).
estado_final(j).
```

- `sucesor(+E1,?E2)` se verifica si `E2` es un sucesor del estado `E1`.

```
sucesor(a,b).   sucesor(a,c).   sucesor(b,d).
sucesor(b,e).   sucesor(c,f).   sucesor(c,g).
sucesor(d,h).   sucesor(e,i).   sucesor(e,j).
sucesor(f,k).   sucesor(h,d).
```

Solución del problema de grafo con ciclos

Solución por búsqueda en profundidad con ciclos:

```
?- ['grafo','b-profundidad-sin-ciclos'].
?- trace(estado_final,+call), profundidad_sin_ciclos(S).
T Call:  ( 10) estado_final(a)
T Call:  ( 11) estado_final(b)
T Call:  ( 12) estado_final(d)
T Call:  ( 13) estado_final(h)
T Call:  ( 14) estado_final(d)
T Call:  ( 15) estado_final(h)
....
?- ['b-profundidad-con-ciclos'].
?- profundidad_con_ciclos(S).
S = [a, b, e, j] ;
S = [a, c, f] ;
No
```


Solución del problema de grafo con ciclos

```
?- trace(estado_final,+call), profundidad_con_ciclos(S).
 T Call: (10) estado_final(a)
 T Call: (11) estado_final(b)
 T Call: (12) estado_final(d)
 T Call: (13) estado_final(h)
 T Call: (12) estado_final(e)
 T Call: (13) estado_final(i)
 T Call: (13) estado_final(j)
S = [a, b, e, j] ;
 T Call: (11) estado_final(c)
 T Call: (12) estado_final(f)
S = [a, c, f] ;
 T Call: (13) estado_final(k)
 T Call: (12) estado_final(g)
No
```

Procedimiento de búsqueda en profundidad con ciclos

(`b-profundidad-con-ciclos.pl`)

- Un *nodo* es una lista de estados $[E_n, \ldots, E_1]$ de forma que E_1 es el estado inicial y E_{i+1} es un sucesor de E_i.
- `profundidad_con_ciclos(?S)` se verifica si `S` es una solución del problema mediante búsqueda en profundidad con ciclos.

```
profundidad_con_ciclos(S) :-
   estado_inicial(E),
   profundidad_con_ciclos([E],S).
```


Procedimiento de búsqueda en profundidad con ciclos

```
profundidad_con_ciclos([E|C],S) :-
   estado_final(E),
   reverse([E|C],S).
profundidad_con_ciclos([E|C],S) :-
   sucesor(E,E1),
   not(memberchk(E1,C)),
   profundidad_con_ciclos([E1,E|C],S).
```

Enunciado del problema de las jarras

- Se tienen dos jarras, una de 4 litros de capacidad y otra de 3.
- Ninguna de ellas tiene marcas de medición.
- Se tiene una bomba que permite llenar las jarras de agua.
- Averiguar cómo se puede lograr tener exactamente 2 litros de agua en la jarra de 4 litros de capacidad.

Especificación del problema de las jarras

- Estado inicial: `0-0`
- Estados finales: `2-y`
- Operadores:
  * Llenar la jarra de 4 litros con la bomba.
  * Llenar la jarra de 3 litros con la bomba.
  * Vaciar la jarra de 4 litros en el suelo.
  * Vaciar la jarra de 3 litros en el suelo.
  * Llenar la jarra de 4 litros con la jarra de 3 litros.
  * Llenar la jarra de 3 litros con la jarra de 4 litros.
  * Vaciar la jarra de 3 litros en la jarra de 4 litros.
  * Vaciar la jarra de 4 litros en la jarra de 3 litros.

Solución del problema de las jarras

```
?- ['jarras','b-profundidad-con-ciclos'].
?- profundidad_con_ciclos(S).
S = [0-0, 4-0, 4-3, 0-3, 3-0, 3-3, 4-2, 0-2, 2-0] ;
S = [0-0, 4-0, 4-3, 0-3, 3-0, 3-3, 4-2, 0-2, 2-0, 2-3]
Yes

?- findall(_S,profundidad_con_ciclos(_S),_L), length(_L,N).
N = 27
```


Representación del problema de las jarras

(jarras.pl)

- `estado_inicial(?E)` se verifica si `E` es el estado inicial.

```
estado_inicial(0-0).
```

- `estado_final(?E)` se verifica si `E` es un estado final.

```
estado_final(2-_).
```

Representación del problema de las jarras

- `sucesor(+E1,?E2)` se verifica si `E2` es un sucesor del estado `E1`.

```
sucesor(X-Y,4-Y)     :- X < 4.
sucesor(X-Y,X-3)     :- Y < 3.
sucesor(X-Y,0-Y)     :- X > 0.
sucesor(X-Y,X-0)     :- Y > 0.
sucesor(X1-Y1,4-Y2) :- X1 < 4, T is X1+Y1, T >= 4,
                       Y2 is Y1-(4-X1).
sucesor(X1-Y1,X2-3) :- Y1 < 3, T is X1+Y1, T >= 3,
                       X2 is X1-(3-Y1).
sucesor(X1-Y1,X2-0) :- Y1 > 0, X2 is X1+Y1, X2 < 4.
sucesor(X1-Y1,0-Y2) :- X1 > 0, Y2 is X1+Y1, Y2 < 3.
```


Tema 4: Resolución de problemas de espacios de estados

Enunciado y representación del problema del paseo

- Enunciado: Una persona puede moverse en línea recta dando cada vez un paso hacia la derecha o hacia la izquierda. Podemos representarlo mediante su posición `X`. El valor inicial de `X` es 0. El problema consiste en llegar a la posición -3.
- `estado_inicial(?E)` se verifica si `E` es el estado inicial.

```
estado_inicial(0).
```

- `estado_final(?E)` se verifica si `E` es un estado final.

```
estado_final(-3).
```

- `sucesor(+E1,?E2)` se verifica si `E2` es un sucesor del estado `E1`.

```
sucesor(E1,E2) :- E2 is E1+1.
sucesor(E1,E2) :- E2 is E1-1.
```


Solución del problema del paseo por búsqueda en profundidad con ciclos

```
?- ['paseo','b-profundidad-con-ciclos'].
?- trace(estado_final,+call), profundidad_con_ciclos(S).
 T Call: (9) estado_final(0)
 T Call: (10) estado_final(1)
 T Call: (11) estado_final(2)
...
```

Solución del problema del paseo por búsqueda en anchura

```
?- ['paseo','b-anchura'].
?- trace(estado_final,+call), anchura(S).
 T Call: (10) estado_final(0)
 T Call: (11) estado_final(1)
 T Call: (12) estado_final(-1)
 T Call: (13) estado_final(2)
 T Call: (14) estado_final(-2)
 T Call: (15) estado_final(3)
 T Call: (16) estado_final(-3)
S = [0, -1, -2, -3]
Yes
```


Procedimiento de búsqueda en anchura

- Un *nodo* es una lista de estados $[E_n, \ldots, E_1]$ de forma que E_1 es el estado inicial y E_{i+1} es un sucesor de E_i.
- *Abiertos* es la lista de nodos pendientes de analizar.
- `anchura(?S)` se verifica si `S` es una solución del problema mediante búsqueda en anchura.

```
anchura(S) :-
   estado_inicial(E),
   anchura([[E]],S).
```

Procedimiento de búsqueda en anchura

```
anchura([[E|C]|_],S) :-
   estado_final(E),
   reverse([E|C],S).
anchura([N|R],S) :-
   expande([N|R],Sucesores),
   append(R,Sucesores,NAbiertos),
   anchura(NAbiertos,S).
```


Procedimiento de búsqueda en anchura

- `expande(+Abiertos,?Sucesores)` se verifica si `Sucesores` es la lista de los sucesores del primer elemento de `Abiertos` que no pertenecen ni al camino que lleva a dicho elemento ni a `Abiertos`. Por ejemplo,

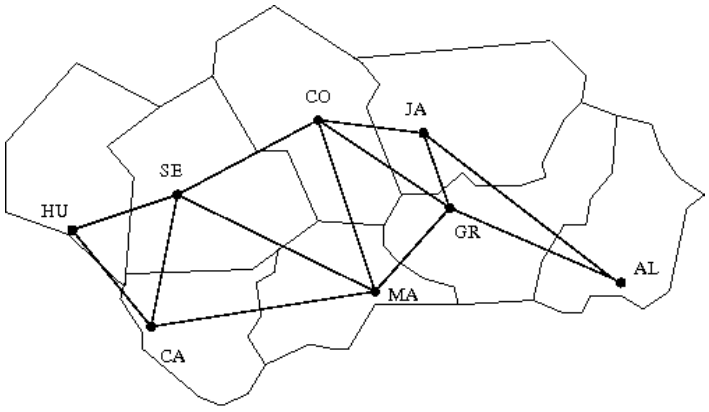
```
? [jarras], expande([[0-0]], L1).
L1 = [[4-0, 0-0], [0-3, 0-0]]
?- expande([[4-0, 0-0], [0-3, 0-0]], L2).
L2 = [[4-3, 4-0, 0-0], [1-3, 4-0, 0-0]]
```

```
expande([[E|C]|R],Sucesores) :-
   findall([E1,E|C],
           (sucesor(E,E1),
            not(memberchk(E1,C)),
            not(memberchk([E1|_],[[E|C]|R]))),
           Sucesores).
```

Tema 4: Resolución de problemas de espacios de estados

Enunciado del problema del viaje

Nos encontramos en una capital andaluza (p.e. Sevilla) y deseamos ir a otra capital andaluza (p.e. Almería). Los autobuses sólo van de cada capital a sus vecinas.



Solución del problema del viaje

```
?- ['viaje', 'b-profundidad-con-ciclos', 'b-anchura'].

?- profundidad_con_ciclos(S).
S = [sevilla, córdoba, jaén, granada, almería]

?- anchura(S).
S = [sevilla, córdoba, granada, almería]
```


Representación del problema del viaje

`viaje.pl`

- `estado_inicial(?E)` se verifica si `E` es el estado inicial.

```
estado_inicial(sevilla).
```

- `estado_final(?E)` se verifica si `E` es un estado final.

```
estado_final(almería).
```

- `sucesor(+E1,?E2)` se verifica si `E2` es un sucesor del estado `E1`.

```
sucesor(E1,E2) :-
   ( conectado(E1,E2) ; conectado(E2,E1) ).
```

Representación del problema del viaje

- `conectado(?E1,?E2)` se verifica si `E1` y `E2` están conectados.

```
conectado(huelva,sevilla).
conectado(huelva,cádiz).
conectado(sevilla,córdoba).
conectado(sevilla,málaga).
conectado(sevilla,cádiz).
conectado(córdoba,jaén).
conectado(córdoba,granada).
conectado(córdoba,málaga).
conectado(cádiz,málaga).
conectado(málaga,granada).
conectado(jaén,granada).
conectado(granada,almería).
```


Representación del problema del viaje

- `coste(+E1,+E2,?C)` se verifica si `C` es la distancia entre `E1` y `E2`.

```
coste(E1,E2,C) :-
   ( distancia(E1,E2,C) ; distancia(E2,E1,C) ).
```

Representación del problema del viaje

- `distancia(+E1,+E2,?D)` se verifica si `D` es la distancia de `E1` a `E2`.

```
distancia(huelva,sevilla,94).
distancia(huelva,cádiz,219).
distancia(sevilla,córdoba,138).
distancia(sevilla,málaga,218).
distancia(sevilla,cádiz,125).
distancia(córdoba,jaén,104).
distancia(córdoba,granada,166).
distancia(córdoba,málaga,187).
distancia(cádiz,málaga,265).
distancia(málaga,granada,129).
distancia(jaén,granada,99).
distancia(granada,almería,166).
```


Procedimiento de búsqueda óptima

`b-optima-1.pl`

- `óptima_1(?S)` se verifica si `S` es una solución óptima del problema; es decir, `S` es una solución del problema y no hay otra solución de menor coste.

```
óptima_1(S) :-
   profundidad_con_ciclos(S),
   coste_camino(S,C),
   not((profundidad_con_ciclos(S1),
        coste_camino(S1,C1),
        C1 < C)).
```

Procedimiento de búsqueda óptima

- `coste_camino(+L,?C)` se verifica si `C` es el coste del camino `L`.

```
coste_camino([E2,E1],C) :-
   coste(E2,E1,C).
coste_camino([E2,E1|R],C) :-
   coste(E2,E1,C1),
   coste_camino([E1|R],C2),
   C is C1+C2.
```


2º procedimiento de búsqueda óptima

`b-optima-2.pl`

- Un *nodo* es un término de la forma $C - [E_n, \ldots, E_1]$ tal que E_1 es el estado inicial y E_{i+1} es un sucesor de E_i y C es el coste de dicho camino.
- `óptima(?S)` se verifica si `S` es una solución del problema mediante búsqueda óptima; es decir, `S` es la solución obtenida por búsqueda óptima a partir de `[0-[E]]`, donde `E` el estado inicial.

```
óptima(S) :-
   estado_inicial(E),
   óptima([0-[E]],S).
```

2º procedimiento de búsqueda óptima

```
óptima([_C-[E|R]|_RA],S) :-
   estado_final(E),
   reverse([E|R],S).
óptima([N|RAbiertos],S) :-
   expande(N,Sucesores),
   append(RAbiertos,Sucesores,Abiertos1),
   sort(Abiertos1,Abiertos2),
   óptima(Abiertos2,S).
```


2º procedimiento de búsqueda óptima

- `expande(+N,?Sucesores)` se verifica si `Sucesores` es la lista de sucesores del nodo `N` (es decir, si `N=C-[E|R]`, entonces `Sucesores` son los nodos de la forma `C1-[E1,E|R]` donde `E1` es un sucesor de `E` que no pertenece a `[E|R]` y `C1` es la suma de `C` y el coste de `E` a `E1`).

```
expande(C-[E|R],Sucesores) :-
   findall(C1-[E1,E|R],
           (sucesor(E,E1),
            not(member(E1,[E|R])),
            coste(E,E1,C2),
            C1 is C+C2),
           Sucesores).
```

2º procedimiento de búsqueda óptima

- Comparaciones

```
?- time(óptima_1(S)).
% 1,409 inferences in 0.00 seconds (Infinite Lips)
S = [sevilla, córdoba, granada, almería]

?-  time(óptima(S)).
% 907 inferences in 0.00 seconds (Infinite Lips)
S = [sevilla, córdoba, granada, almería]
```


Bibliografía

1. Bratko, I. *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
   - Cap. 11 "Basic problem–solving strategies"
2. Flach, P. *Simply Logical (Intelligent Reasoning by Example)* (John Wiley, 1994)
   - Cap. 5: "Seaching graphs"
3. Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)
   - Cap. 4: "Searching"
4. Shoham, Y. *Artificial Intelligence Techniques in Prolog* (Morgan Kaufmann, 1994)
   - Cap. 2 "Search"

Capítulo 5

Procesamiento del lenguaje natural

Programación lógica (2008–09)

Tema 5: Procesamiento del lenguaje natural

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 5: Procesamiento del lenguaje natural

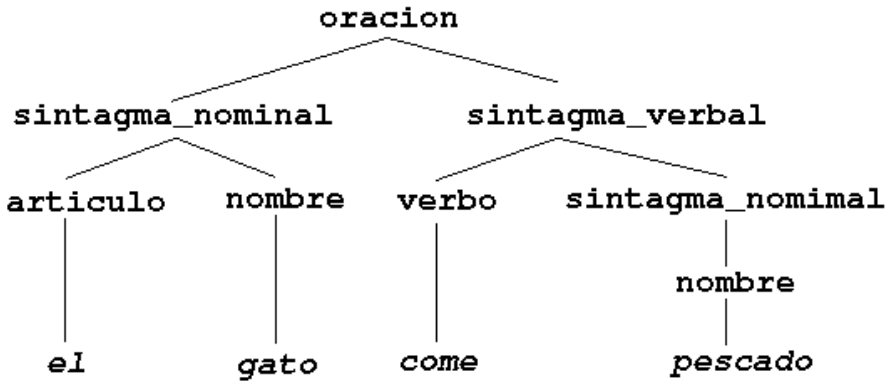
Ejemplo de gramática libre de contexto

- Ejemplos de frases
  - El gato come pescado
  - El perro come carne
- Ejemplo de gramática libre de contexto (GLC)

```
oración           -->  sintagma_nominal, sintagma_verbal
sintagma_nominal  -->  nombre
sintagma_nominal  -->  artículo, nombre
sintagma_verbal   -->  verbo, sintagma_nominal
artículo          -->  [el]
nombre            -->  [gato]
nombre            -->  [perro]
nombre            -->  [pescado]
nombre            -->  [carne]
verbo             -->  [come]
```

Árbol de análisis en gramáticas libres de contexto

- Árbol de análisis



Definiciones de gramáticas libres de contexto

- Concepto de gramática: G = (N,T,P,S)
  - N: vocabulario no terminal (categorías sintácticas)
  - T: vocabulario terminal
  - P: reglas de producción
  - S: símbolo inicial
- Vocabulario: $V = N \cup T$ es el vocabulario con $N \cap T = \emptyset$
- Derivaciones
  - $xAy \Longrightarrow xwy$ mediante $A \Longrightarrow w$
  - $x \overset{*}{\Longrightarrow} y$ si existen $x_1, x_2, \ldots, x_n$ tales que $x = x_1 \Longrightarrow x_2 \cdots \Longrightarrow x_{n-1} \Longrightarrow x_n = y$
- Lenguaje definido por una gramática: $L(G) = \{x \in T^* : S \overset{*}{\Longrightarrow} x\}$
- Gramáticas libres de contextos (GLC): $A \Longrightarrow w$ con $A \in N$ y $w \in V^*$

Tema 5: Procesamiento del lenguaje natural

Reconocedor de GLC mediante append

- Representación de oraciones en Prolog

```
[el, gato, come, pescado]   [el, perro, come, carne]
```

- Sesión con el reconocedor de GLC en Prolog mediante append

```
?- time(oración([el,gato,come,pescado])).
% 178 inferences in 0.00 seconds (Infinite Lips)
Yes

?- time(oración([el,come,pescado])).
% 349 inferences in 0.00 seconds (Infinite Lips)
No
```

Reconocedor de GLC mediante append

- Definición del reconocedor de GLC en Prolog mediante append

```
oración(O) :-
   sintagma_nominal(SN), sintagma_verbal(SV), append(SN,SV
sintagma_nominal(SN) :-
   nombre(SN).
sintagma_nominal(SN) :-
   artículo(A), nombre(N), append(A,N,SN).
sintagma_verbal(SV) :-
   verbo(V), sintagma_nominal(SN), append(V,SN,SV).

artículo([el]).
nombre([gato]).      nombre([perro]).
nombre([pescado]).   nombre([carne]).
verbo([come]).
```


Reconocedor de GLC mediante append

Otros usos de la gramática

- Generación de las oraciones

```
?- oración(O).
O = [gato, come, gato] ;
O = [gato, come, perro] ;
O = [gato, come, pescado]
Yes

?- findall(_O,oración(_O),_L),length(_L,N).
N = 64
```

Reconocedor de GLC mediante append

Otros usos de la gramática (cont.)

- Reconocedor de las categorías gramaticales

```
?- sintagma_nominal([el,gato]).
Yes

?- sintagma_nominal([un,gato]).
No
```

- Generador de las categorias gramaticales

```
?- findall(_SN,sintagma_nominal(_SN),L).
L = [[gato],[perro],[pescado],[carne],
     [el,gato],[el,perro],[el,pescado],[el,carne]]
```


Tema 5: Procesamiento del lenguaje natural

Reconocedor de GLC mediante listas de diferencia

- Sesión (y ganancia en eficiencia)

```
?- time(oración([el,gato,come,pescado]-[])).
% 9 inferences in 0.00 seconds (Infinite Lips)
Yes

?- time(oración([el,come,pescado]-[])).
% 5 inferences in 0.00 seconds (Infinite Lips)
No
```


Reconocedor de GLC mediante listas de diferencia

- Definición

```
oración(A-B) :-
   sintagma_nominal(A-C), sintagma_verbal(C-B).
sintagma_nominal(A-B) :-
   nombre(A-B).
sintagma_nominal(A-B) :-
   artículo(A-C), nombre(C-B).
sintagma_verbal(A-B) :-
   verbo(A-C), sintagma_nominal(C-B).

artículo([el|A]-A).
nombre([gato|A]-A).      nombre([perro|A]-A).
nombre([pescado|A]-A).   nombre([carne|A]-A).
verbo([come|A]-A).
```

Reconocedor de GLC mediante listas de diferencia

Otros usos de la gramática

- Generación de las oraciones

```
?- oración(O-[]).
O = [gato, come, gato] ;
O = [gato, come, perro] ;
O = [gato, come, pescado]
Yes

?- findall(_O,oración(_O-[]),_L),length(_L,N).
N = 64
```


Reconocedor de GLC mediante listas de diferencia

Otros usos de la gramática (cont.)

- Reconocedor de las categorías gramaticales

```
?- sintagma_nominal([el,gato]-[]).
Yes
?- sintagma_nominal([un,gato]-[]).
No
```

- Generador de las categorias gramaticales

```
?- findall(_SN,sintagma_nominal(_SN-[]),L).
L = [[gato],[perro],[pescado],[carne],
     [el,gato],[el,perro],[el,pescado],[el,carne]]
```

Tema 5: Procesamiento del lenguaje natural

Ejemplo de gramática de cláusulas definidas

- Definición

```
oración          -->  sintagma_nominal, sintagma_verbal.
sintagma_nominal -->  nombre.
sintagma_nominal -->  artículo, nombre.
sintagma_verbal  -->  verbo, sintagma_nominal.
artículo         -->  [el].
nombre           -->  [gato].
nombre           -->  [perro].
nombre           -->  [pescado].
nombre           -->  [carne].
verbo            -->  [come].
```

Usos de gramática de cláusulas definidas

- Reconocimiento de oraciones

```
?- oración([el,gato,come,pescado],[]).
Yes
?- oración([el,come,pescado]-[]).
No
```

- Generación de las oraciones

```
?- oración(O,[]).
O = [gato, come, gato] ;
O = [gato, come, perro] ;
O = [gato, come, pescado]
Yes
?- findall(_O,oración(_O,[]),_L),length(_L,N).
N = 64
```


Usos de gramáticas de cláusulas definidas

- Reconocedor de las categorías gramaticales

```
?- sintagma_nominal([el,gato],[]).
Yes
?- sintagma_nominal([un,gato],[]).
No
```

- Generador de las categorias gramaticales

```
?- findall(_SN,sintagma_nominal(_SN,[]),L).
L = [[gato],[perro],[pescado],[carne],
     [el,gato],[el,perro],[el,pescado],[el,carne]]
```

Usos de gramáticas de cláusulas definidas

- Determinación de elementos

```
?- oración([X,gato,Y,pescado],[]).
X = el
Y = come ;
No
```

- La relación phrase

```
?- phrase(oración,[el,gato,come,pescado]).
Yes
?- phrase(sintagma_nominal,L).
L = [gato] ;
L = [perro]
Yes
```


Compilación de gramáticas de cláusulas definidas

```
?- listing([oración,sintagma_nominal,
            sintagma_verbal,
            artículo, nombre, verbo]).
oración(A,B) :- sintagma_nominal(A,C), sintagma_verbal(C,B).
sintagma_nominal(A,B) :- nombre(A,B).
sintagma_nominal(A,B) :- artículo(A,C), nombre(C,B).
sintagma_verbal(A,B) :- verbo(A,C), sintagma_nominal(C,B).
artículo([el|A], A).
nombre([gato|A], A).
nombre([perro|A], A).
nombre([pescado|A], A).
nombre([carne|A], A).
verbo([come|A], A).
Yes
```

Eficiencia de gramáticas de cláusulas definidas

```
?- time(oración([el,gato,come,pescado],[])).
% 9 inferences in 0.00 seconds (Infinite Lips)
Yes

?- time(oración([el,come,pescado]-[])).
% 5 inferences in 0.00 seconds (Infinite Lips)
No
```


Tema 5: Procesamiento del lenguaje natural

Reglas recursivas en GCD

Problema: Extender el ejemplo de GCD para aceptar oraciones como `[el,gato,come,pescado,o,el,perro,come,pescado]`

- Primera propuesta (GCD)

```
oración          -->  oración, conjunción, oración.
oración          -->  sintagma_nominal, sintagma_verbal.
sintagma_nominal -->  nombre.
sintagma_nominal -->  artículo, nombre.
sintagma_verbal  -->  verbo, sintagma_nominal.
artículo         -->  [el].
nombre           -->  [gato].
nombre           -->  [perro].
nombre           -->  [pescado].
nombre           -->  [carne].
verbo            -->  [come].
conjunción       -->  [y].
conjunción       -->  [o].
```


Reglas recursivas en GCD

- Problema con la primera propuesta:

```
?- oración([el,gato,come,pescado,o,el,perro,come,pescado]
ERROR: Out of local stack

?- listing(oración).
oración(A, B) :-
   oración(A, C),
   conjunción(C, D),
   oración(D, B).
oración(A, B) :-
   sintagma_nominal(A, C),
   sintagma_verbal(C, B).
Yes
```

Reglas recursivas en GCD

- Segunda propuesta

```
oración           -->  sintagma_nominal, sintagma_verbal.
oración           -->  oración, conjunción, oración.
sintagma_nominal  -->  nombre.
sintagma_nominal  -->  artículo, nombre.
sintagma_verbal   -->  verbo, sintagma_nominal.
artículo          -->  [el].
nombre            -->  [gato].
nombre            -->  [perro].
nombre            -->  [pescado].
nombre            -->  [carne].
verbo             -->  [come].
conjunción        -->  [y].
conjunción        -->  [o].
```


Reglas recursivas en GCD

- Problema con la segunda propuesta:

```
?- oración([el,gato,come,pescado,o,el,perro,come,pescado]
Yes

?- oración([un,gato,come],[]).
ERROR: Out of local stack
```

Reglas recursivas en GCD

- Tercera propuesta

```
oración           -->  oración_simple.
oración           -->  oración_simple, conjunción, oración.
oración_simple    -->  sintagma_nominal, sintagma_verbal.
sintagma_nominal  -->  nombre.
sintagma_nominal  -->  artículo, nombre.
sintagma_verbal   -->  verbo, sintagma_nominal.
artículo          -->  [el].
nombre            -->  [gato].
nombre            -->  [perro].
nombre            -->  [pescado].
nombre            -->  [carne].
verbo             -->  [come].
conjunción        -->  [y].
conjunción        -->  [o].
```


Reglas recursivas en GCD

- Solución con la tercera propuesta:

```
?- oración([el,gato,come,pescado,o,el,perro,come,pescado]
Yes

?- oración([un,gato,come],[]).
No
```

Tema 5: Procesamiento del lenguaje natural

GCD para el lenguaje formal $\{a^n b^n : n \in \mathbb{N}\}$

- Sesión

```
?- s([a,a,b,b],[]).
Yes

?- s([a,a,b,b,b],[]).
No

?- s(X,[]).
X = [] ;
X = [a, b] ;
X = [a, a, b, b] ;
X = [a, a, a, b, b, b]
Yes
```

GCD para el lenguaje formal $\{a^n b^n : n \in \mathbb{N}\}$

- GCD

```
s --> [].
s --> i,s,d.
i --> [a].
d --> [b].
```


Tema 5: Procesamiento del lenguaje natural

Árbol de análisis con GCD

- Ejemplo de cálculo del árbol de análisis:

```
?- oración(A,[el,gato,come,pescado],[]).
A = o(sn(art(el),n(gato)),sv(v(come),sn(n(pescado))))
Yes
```


Árbol de análisis con GCD

- Definición de GCD con árbol de análisis

```
oración(o(SN,SV))            -->  sintagma_nominal(SN),
                                  sintagma_verbal(SV).
sintagma_nominal(sn(N))      -->  nombre(N).
sintagma_nominal(sn(Art,N))  -->  artículo(Art),
                                  nombre(N).
sintagma_verbal(sv(V,SN))    -->  verbo(V),
                                  sintagma_nominal(SN).
artículo(art(el))            -->  [el].
nombre(n(gato))              -->  [gato].
nombre(n(perro))             -->  [perro].
nombre(n(pescado))           -->  [pescado].
nombre(n(carne))             -->  [carne].
verbo(v(come))               -->  [come].
```

Árbol de análisis con GCD: Compilación

```
?- listing([oración,sintagma_nominal,sintagma_verbal,
            artículo,nombre,verbo]).
oración(o(A, B), C, D) :-
   sintagma_nominal(A, C, E), sintagma_verbal(B, E, D).
sintagma_nominal(sn(A), B, C) :- nombre(A, B, C).
sintagma_nominal(sn(A, B), C, D) :-
   artículo(A, C, E), nombre(B, E, D).
sintagma_verbal(sv(A, B), C, D) :-
   verbo(A, C, E), sintagma_nominal(B, E, D).
artículo(art(el), [el|A], A).
nombre(n(gato), [gato|A], A).
nombre(n(perro), [perro|A], A).
nombre(n(pescado), [pescado|A], A).
nombre(n(carne), [carne|A], A).
verbo(v(come), [come|A], A).
Yes
```


Tema 5: Procesamiento del lenguaje natural

Concordancia de género en GCD

Ejemplos de concordancia de género en GCD

```
?- phrase(oración,[el,gato,come,pescado]).
Yes

?- phrase(oración,[la,gato,come,pescado]).
No

?- phrase(oración,[la,gata,come,pescado]).
Yes
```


Concordancia de género en GCD

Definición de GCD con concordancia en género:

```
oración              -->  sintagma_nominal, sintagma_verbal.
sintagma_nominal     -->  nombre(_).
sintagma_nominal     -->  artículo(G), nombre(G).
sintagma_verbal      -->  verbo, sintagma_nominal.
artículo(masculino)  -->  [el].
artículo(femenino)   -->  [la].
nombre(masculino)    -->  [gato].
nombre(femenino)     -->  [gata].
nombre(masculino)    -->  [pescado].
verbo                -->  [come].
```

Concordancia de número en GCD

Ejemplo de concordancia de número en GCD

```
?- phrase(oración,[el,gato,come,pescado]).
Yes

?- phrase(oración,[los,gato,come,pescado]).
No

?- phrase(oración,[los,gatos,comen,pescado]).
Yes
```


Concordancia de número en GCD

Definición de GCD con concordancia de número:

```
oración              -->  sintagma_nominal(N), sintagma_verbal(N).
sintagma_nominal(N)  -->  nombre(N).
sintagma_nominal(N)  -->  artículo(N), nombre(N).
sintagma_verbal(N)   -->  verbo(N), sintagma_nominal(_).
artículo(singular)   -->  [el].
artículo(plural)     -->  [los].
nombre(singular)     -->  [gato].
nombre(plural)       -->  [gatos].
nombre(singular)     -->  [perro].
nombre(plural)       -->  [perros].
nombre(singular)     -->  [pescado].
nombre(singular)     -->  [carne].
verbo(singular)      -->  [come].
verbo(plural)        -->  [comen].
```

Tema 5: Procesamiento del lenguaje natural

Ejemplo de GCD no GCL

GCD para el lenguaje formal $\{a^n b^n c^n : n \in \mathbb{N}\}$

- Sesión

```
?- s([a,a,b,b,c,c],[]).
Yes

?- s([a,a,b,b,b,c,c],[]).
No

?- s(X,[]).
X = [] ;
X = [a, b, c] ;
X = [a, a, b, b, c, c]
Yes
```

Ejemplo de GCD no GCL

- GCD

```
s                --> bloque_a(N),
                     bloque_b(N),
                     bloque_c(N).
bloque_a(0)      --> [].
bloque_a(suc(N)) --> [a], bloque_a(N).
bloque_b(0)      --> [].
bloque_b(suc(N)) --> [b], bloque_b(N).
bloque_c(0)      --> [].
bloque_c(suc(N)) --> [c], bloque_c(N).
```


GCD con llamadas a Prolog

GCD para el lenguaje formal $L = \{a^{2n}b^{2n}c^{2n} : n \in \mathbb{N}\}$.

- Ejemplos

```
?- s([a,a,b,b,c,c],[]).
Yes

?- s([a,b,c],[]).
No

?- s(X,[]).
X = [] ;
X = [a,a,b,b,c,c] ;
X = [a,a,a,a,b,b,b,b,c,c,c,c] ;
X = [a,a,a,a,a,a,b,b,b,b,b,b,c,c,c,c,c,c]
Yes
```

GCD con llamadas a Prolog

- GCD

```
s               --> bloque_a(N),
                    bloque_b(N),
                    bloque_c(N),
                    {par(N)}.
bloque_a(0)     --> [].
bloque_a(s(N)) --> [a],bloque_a(N).
bloque_b(0)     --> [].
bloque_b(s(N)) --> [b],bloque_b(N).
bloque_c(0)     --> [].
bloque_c(s(N)) --> [c],bloque_c(N).

par(0).
par(s(s(N))) :- par(N).
```


Tema 5: Procesamiento del lenguaje natural

Separación de reglas y lexicón

- Lexicón

```
lex(el,artículo).
lex(gato,nombre).
lex(perro,nombre).
lex(pescado,nombre).
lex(carne,nombre).
lex(come,verbo).
```


Separación de reglas y lexicón

- Regla

```
oración           -->  sintagma_nominal, sintagma_verbal.
sintagma_nominal  -->  nombre.
sintagma_nominal  -->  artículo, nombre.
sintagma_verbal   -->  verbo, sintagma_nominal.
artículo          -->  [Palabra], {lex(Palabra,artículo)}.
nombre            -->  [Palabra], {lex(Palabra,nombre)}.
verbo             -->  [Palabra], {lex(Palabra,verbo)}.
```

- Sesión

```
?- oración([el,gato,come,pescado],[]).
Yes
?- oración([el,come,pescado],[]).
No
```

Separación de reglas y lexicón con concordancia

- Sesión

```
?- oración([el,gato,come,pescado],[]).     ==>  Yes
?- oración([los,gato,come,pescado],[]).    ==>  No
?- oración([los,gatos,comen,pescado],[]).  ==>  Yes
```

- Lexicón

```
lex(el,artículo,singular).      lex(los,artículo,plural).
lex(gato,nombre,singular).      lex(gatos,nombre,plural).
lex(perro,nombre,singular).     lex(perros,nombre,plural).
lex(pescado,nombre,singular).   lex(pescados,nombre,plural)
lex(carne,nombre,singular).     lex(carnes,nombre,plural).
lex(come,verbo,singular).       lex(comen,verbo,plural).
```


Separación de reglas y lexicón con concordancia

- Reglas

```
oración                --> sintagma_nominal(N),
                           sintagma_verbal(N).
sintagma_nominal(N)    --> nombre(N).
sintagma_nominal(N)    --> artículo(N),
                           nombre(N).
sintagma_verbal(N)     --> verbo(N),
                           sintagma_nominal(_).
artículo(N)            --> [Palabra],
                           {lex(Palabra,artículo,N)}.
nombre(N)              --> [Palabra],
                           {lex(Palabra,nombre,N)}.
verbo(N)               --> [Palabra],
                           {lex(Palabra,verbo,N)}.
```

Lexicón con género y número

- Sesión

```
?- oración([la,profesora,lee,un,libro],[]).     ==>  Yes
?- oración([la,profesor,lee,un,libro],[]).      ==>  No
?- oración([los,profesores,leen,un,libro],[]). ==>  Yes
?- oración([los,profesores,leen],[]).           ==>  Yes
?- oración([los,profesores,leen,libros],[]).    ==>  Yes
```


Lexicón con género y número

- Lexicón

```
lex(el,determinante,masculino,singular).
lex(los,determinante,masculino,plural).
lex(la,determinante,femenino,singular).
lex(las,determinante,femenino,plural).
lex(un,determinante,masculino,singular).
lex(una,determinante,femenino,singular).
lex(unos,determinante,masculino,plural).
lex(unas,determinante,femenino,plural).
lex(profesor,nombre,masculino,singular).
lex(profesores,nombre,masculino,plural).
lex(profesora,nombre,femenino,singular).
lex(profesoras,nombre,femenino,plural).
lex(libro,nombre,masculino,singular).
lex(libros,nombre,masculino,plural).
lex(lee,verbo,singular).
lex(leen,verbo,plural).
```

Lexicón con género y número

- Reglas

```
oración              --> sintagma_nominal(N),
                         verbo(N),
                         complemento.
complemento          --> [].
complemento          --> sintagma_nominal(_).
sintagma_nominal(N)  --> nombre(_,N).
sintagma_nominal(N)  --> determinante(G,N), nombre(G,N).
determinante(G,N)    --> [P],{lex(P,determinante,G,N)}.
nombre(G,N)          --> [P],{lex(P,nombre,G,N)}.
verbo(N)             --> [P],{lex(P,verbo,N)}.
```


Bibliografía

- P. Blackburn, J. Bos y K. Striegnitz *Learn Prolog Now!* [http://www.coli.uni-sb.de/~kris/learn-prolog-now]
  - Cap. 7 "Definite Clause Grammars"
  - Cap. 8 "More Definite Clause Grammars"
- I. Bratko *Prolog Programming for Artificial Intelligence (Third ed.)* (Prentice–Hall, 2001)
  - Cap 21: "Language Processing with Grammar Rules"
- P. Flach *Simply Logical (Intelligent Reasoning by Example)* (John Wiley, 1994)
  - Cap. 7: "Reasoning with natural languaje"
- U. Nilsson y J. Maluszynski *Logic, Programming and Prolog (2nd ed.)* (Autores, 2000) [http://www.ida.liu.se/~ulfni/lpp]
  - Cap. 10 "Logic and grammars"
- L. Sterling y E. Shapiro *The Art of Prolog (2nd edition)* (The MIT Press, 1994)
  - Cap. 19: "Logic grammars"

Capítulo 6

Ingeniería del conocimiento y metaintérpretes

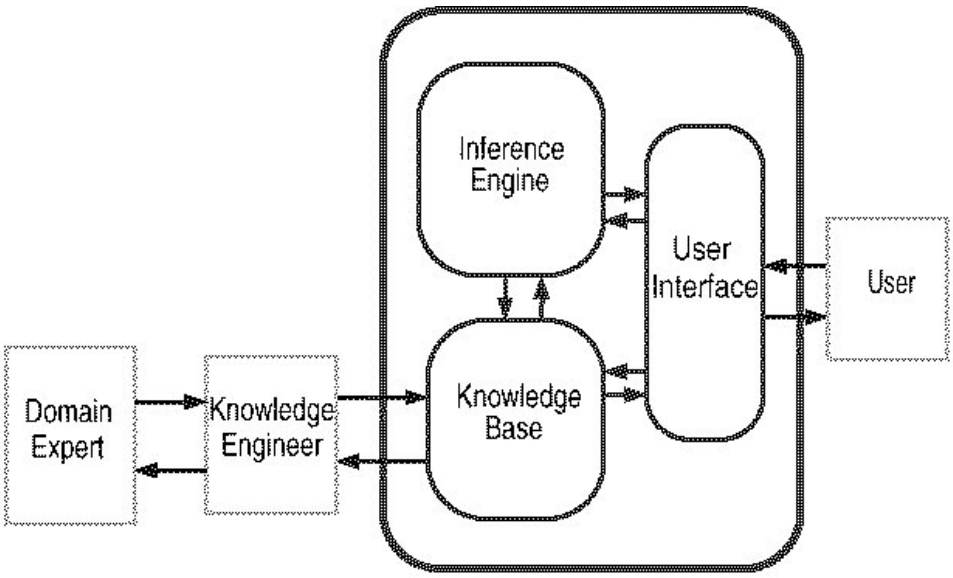
Programación lógica (2008–09)

Tema 6: Ingeniería del conocimiento y metaintérpretes

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

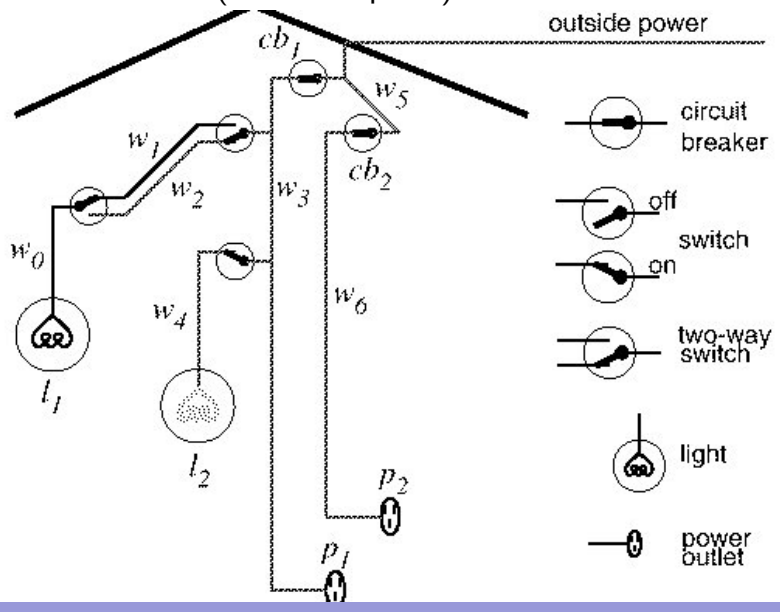
Arquitectura de los SBC (Poole–98 p. 200)



Tema 6: Ingeniería del conocimiento y metaintérpretes

Ejemplo de BC objeto

- El sistema eléctrico (Poole–98 p. 16)



Ejemplo de BC objeto: `i_electrica.pl`

- Operadores

```
:- op(1100, xfx, <-).
:- op(1000, xfy, &).
```

- `luz(?L)` se verifica si `L` es una luz

```
luz(l1) <- verdad.
luz(l2) <- verdad.
```

- `abajo(?I)` se verifica si el interruptor `I` está hacia abajo

```
abajo(i1) <- verdad.
```

Ejemplo de BC objeto: i_electrica.pl

- `arriba(?I)` se verifica si el interruptor `I` está hacia arriba

```
arriba(i2) <- verdad.
arriba(i3) <- verdad.
```

- `esta_bien(?X)` se verifica si la luz (o cortacircuito) `X` está bien.

```
esta_bien(l1) <- verdad.
esta_bien(l2) <- verdad.
esta_bien(cc1) <- verdad.
esta_bien(cc2) <- verdad.
```


Ejemplo de BC objeto: i_electrica.pl

- `conectado(?D1,?D2)` se verifica si los dispositivos `D1` y `D2` está conectados (de forma quela corriente eléctrica va de `D2` a `D1`)

```
conectado(l1,c0) <- verdad.
conectado(c0,c1) <- arriba(i2).
conectado(c0,c2) <- abajo(i2).
conectado(c1,c3) <- arriba(i1).
conectado(c2,c3) <- abajo(i1).
conectado(l2,c4) <- verdad.
conectado(c4,c3) <- arriba(i3).
conectado(e1,c3) <- verdad.
conectado(c3,c5) <- esta_bien(cc1).
conectado(e2,c6) <- verdad.
conectado(c6,c5) <- esta_bien(cc2).
conectado(c5,entrada) <- verdad.
```

Ejemplo de BC objeto: `i_electrica.pl`

- `tiene_corriente(?D)` se verifica si el dispositivo `D` tiene corriente

```
tiene_corriente(D)        <- conectado(D,D1) &
                             tiene_corriente(D1).
tiene_corriente(entrada)  <- verdad.
```

- `esta_encendida(?L)` se verifica si la luz `L` está encendida

```
esta_encendida(L) <-  luz(L) &
                      esta_bien(L) &
                      tiene_corriente(L).
```


Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintérprete simple

- Sesión

```
?- prueba(esta_encendida(X)).
X = l2 ;
No
```

- Metaintérprete simple

  `prueba(+O)` se verifica si el objetivo `O` se puede demostrar a partir de la BC objeto

```
prueba(verdad).
prueba((A & B)) :-
    prueba(A),
    prueba(B).
prueba(A) :-
    (A <- B),
    prueba(B).
```


Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintéprete ampliado

- Ampliación del lenguaje base:
  - Disyunciones: `A ; B`
  - Predicados predefinidos: `is`, `<`, ...
- Operadores

```
:- op(1100, xfx, <-).
:- op(1000, xfy, [&,;]).
```

- Ejemplo de BC ampliada

```
vecino(X,Y) <- Y is X-1 ; Y is X+1.
```

- Sesión

```
?- prueba(vecino(2,3)).
Yes
?- prueba(vecino(3,2)).
Yes
```


Metaintéprete ampliado

- `prueba(+O)` se verifica si el objetivo `O` se puede demostrar a partir de la BC objeto (que puede contener disyunciones y predicados predefinidos)

```
prueba(verdad).
prueba((A & B)) :- prueba(A), prueba(B).
prueba((A ; B)) :- prueba(A).
prueba((A ; B)) :- prueba(B).
prueba(A)       :- predefinido(A), A.
prueba(A)       :- (A <- B), prueba(B).
```

- `predefinido(+O)` se verifica si `O` es un predicado predefinido

```
predefinido((X is Y)).
predefinido((X < Y)).
```

Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintérprete con profundidad acotada

- `prueba_pa(+O,+N)` es verdad si el objetivo `O` se puede demostrar con profundidad `N` como máximo

```
prueba_pa(verdad,_N).
prueba_pa((A & B),N) :-
   prueba_pa(A,N),
   prueba_pa(B,N).
prueba_pa(A,N) :-
   N >= 0,
   N1 is N-1,
   (A <- B),
   prueba_pa(B,N1).
```

Metaintérprete con profundidad acotada

- Ejemplo

```
numero(0) <- verdad.
numero(s(X)) <- numero(X).
```

- Sesión

```
?- prueba_pa(numero(N), 3).
N = 0 ;
N = s(0) ;
N = s(s(0)) ;
N = s(s(s(0))) ;
No
?- prueba_pa(numero(s(s(0))),1).
No
?- prueba_pa(numero(s(s(0))),2).
Yes
```


Metaintérprete con profundidad acotada

- Segundo ejemplo
  - Programa

```
hermano(X,Y) <- hermano(Y,X).
hermano(b,a) <- verdad.
```

  - Sesión

```
?- prueba(hermano(a,X)).
ERROR: Out of local stack
?- prueba_pa(hermano(a,X),1).
X = b ; No
?- prueba_pa(hermano(X,Y),1).
X = a  Y = b ; X = b  Y = a ; No
?- prueba_pa(hermano(a,X),2).
X = b ; No
?- prueba_pa(hermano(a,X),3).
X = b ; X = b ; No
```

Tema 6: Ingeniería del conocimiento y metaintérpretes

Consulta al usuario

- Planteamiento del problema: Aportación del conocimiento de los usuarios cuando:
  - No conocen las interioridades del sistema
  - No son expertos en el dominio
  - No concocen qué información es importante
  - No conocen la sintaxis del sistema
  - Poseen información esencial del caso particular del problema
- Funciones del sistema:
  - Determinar qué información es importante
  - Preguntar al usuario sobre dicha información
- Tipos de objetivos:
  - No preguntable
  - Preguntable no–preguntado
  - Preguntado

Consulta al usuario

- Preguntas elementales:
  - Son objetivos básicos (sin variables)
  - Las respuestas son "sí" o "no"
  - Se plantean si son importantes, preguntables y no preguntadas
  - El sistema almacena la respuesta

Tema 6: Ingeniería del conocimiento y metaintérpretes

Ejemplo de consulta al usuario

- Ejemplo: Modificación de `i_electrica.pl`

```
preguntable(arriba(_)).
preguntable(abajo(_)).
```

- Sesión

```
?- prueba_p(esta_encendida(L)).
¿Es verdad arriba(i2)? (si/no)
|: si.
¿Es verdad arriba(i1)? (si/no)
|: no.
¿Es verdad abajo(i2)? (si/no)
|: no.
¿Es verdad arriba(i3)? (si/no)
|: si.

L = l2 ;
No
```


Ejemplo de consulta al usuario (cont.)

```
?- listing(respuesta).
respuesta(arriba(i2), si).
respuesta(arriba(i1), no).
respuesta(abajo(i2), no).
respuesta(arriba(i3), si).
Yes

?- retractall(respuesta(_,_)).
Yes

?- prueba_p(esta_encendida(L)).
¿Es verdad arriba(i2)? (si/no)
|: si.
¿Es verdad arriba(i1)? (si/no)
|: si.
L = l1 ;
¿Es verdad abajo(i2)? (si/no)
|:
```

Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintérprete con preguntas

- `prueba_p(+O)` se verifica si el objetivo `O` se puede demostrar a partir de la BC objeto y las respuestas del usuario

```
prueba_p(verdad).
prueba_p((A & B)) :- prueba_p(A), prueba_p(B).
prueba_p(G)       :- preguntable(G), respuesta(G,si).
prueba_p(G) :-
   preguntable(G),
   no_preguntado(G),
   pregunta(G,Respuesta),
   assert(respuesta(G,Respuesta)),
   Respuesta=si.
prueba_p(A) :-
   (A <- B),
   prueba_p(B).
```

Metaintérprete con preguntas (cont.)

- `respuesta(?O,?R)` se verifica si la respuesta al objetivo `O` es `R`. [Se añade dinámicamente a la base de datos]

```
:- dynamic respuesta/2.
```

- `no_preguntado(+O)` es verdad si el objetivo `O` no se ha preguntado

```
no_preguntado(O) :-
   not(respuesta(O,_)).
```


Metaintérprete con preguntas (cont.)

- `pregunta(+O, -Respuesta)` pregunta `O` al usuario y éste responde la `Respuesta`

```
pregunta(O,Respuesta) :-
   escribe_lista(['¿Es verdad ',O,'? (si/no)']),
   read(Respuesta).
```

- `escribe_lista(+L)` escribe cada uno de los elementos de la lista `L`

```
escribe_lista([]) :- nl.
escribe_lista([X|L]) :-
   write(X),
   escribe_lista(L).
```

Tema 6: Ingeniería del conocimiento y metaintérpretes

Otros tipos de consultas

- Consulta sobre relaciones funcionales
  - Ejemplo: `edad(P,N)`
- Forma de la consulta:
  - Menús
  - Texto de formato libre
- Preguntas generales

Pregunta	¿Preguntable?	Respuesta
p(X)	Sí	p(f(Z))
p(f(c))	No	
p(a)	Sí	Sí
p(X)	Sí	No
P(X)	No	

Tema 6: Ingeniería del conocimiento y metaintérpretes

Explicación

- Necesidad del sistema de justificar sus respuestas
- Uso en explicación y depuración
- Tipos de explicaciones:
  - Preguntar CÓMO se ha probado un objetivo
  - Preguntar PORQUÉ plantea una consulta
  - Preguntar PORQUÉ_NO se ha probado un objetivo
- Preguntas CÓMO
  - El usuario puede preguntar CÓMO ha probado el objetivo `q`
  - El sistema muestra la instancia de la regla usada

    ```
    q :- p1,...,pn.
    ```

  - El usuario puede preguntar CÓMO i para obtener la regla usada para probar `pi`
- El comando CÓMO permite descender en el árbol de prueba

Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintérprete con árbol de prueba

```
?- [meta_con_explicacion_arbol, i_electrica].
Yes
?- prueba_con_demostracion(esta_encendida(L),T).
L = l2
T = si(esta_encendida(l2),
        (si(luz(l2), verdad) &
         si(esta_bien(l2), verdad) &
         si(tiene_corriente(l2),
             (si(conectado(l2, c4), verdad) &
              si(tiene_corriente(c4),
                  (si(conectado(c4, c3),
                       si(arriba(i3), verdad)) &
                   si(tiene_corriente(c3),
                       (si(conectado(c3, c5),
                            si(esta_bien(cc1), verdad)) &
                        si(tiene_corriente(c5),
                            (si(conectado(c5, entrada), verdad) &
                            si(tiene_corriente(entrada), verdad)))))))))) ;

No
```

Metaintérprete con árbol de prueba

- Metaintérprete con árbol de prueba
  `prueba_con_demostracion(+O,?A)` es verdad si `A` es un árbol de prueba del objetivo `O`

```
prueba_con_demostracion(verdad,verdad).
prueba_con_demostracion((A & B),(AA & AB)) :-
   prueba_con_demostracion(A,AA),
   prueba_con_demostracion(B,AB).
prueba_con_demostracion(O,si(O,AB)) :-
   (O <- B),
   prueba_con_demostracion(B,AB).
```


Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintérprete con Cómo

```
?- [meta_con_explicacion_como, i_electrica].
Yes
?- prueba_con_como(esta_encendida(L)).
esta_encendida(l2) :-
   1: luz(l2)
   2: esta_bien(l2)
   3: tiene_corriente(l2)
|: 3.
tiene_corriente(l2) :-
   1: conectado(l2, c4)
   2: tiene_corriente(c4)
|: 2.
tiene_corriente(c4) :-
   1: conectado(c4, c3)
   2: tiene_corriente(c3)
|: 1.
conectado(c4, c3) :-
   1: arriba(i3)
|: 1.
arriba(i3) es un hecho
L = l2 ;
No
```


Metaintérprete con Cómo

- `prueba_con_como(+O)` significa probar el objetivo `O` a partir de la BC objeto y navegar por su árbol de prueba mediante preguntas Cómo

```
prueba_con_como(O) :-
   prueba_con_demostracion(O,A),
   navega(A).
```

- `navega(+A)` significa que se está navegando en el árbol `A`

```
navega(si(A,verdad)) :-
    escribe_lista([A,' es un hecho']).
navega(si(A,B)) :-
    B \== verdad,
    escribe_lista([A,' :-']),
    escribe_cuerpo(B,1,_),
    read(Orden),
    interpreta_orden(Orden,B).
```

Metaintérprete con CÓMO

- `escribe_lista(+L)` escribe cada uno de los elementos de la lista `L`

```
escribe_lista([]) :- nl.
escribe_lista([X|L]) :-
   write(X),
   escribe_lista(L).
```

- `escribe_cuerpo(+B,+N1,?N2)` es verdad si `B` es un cuerpo que se va a escribir, `N1` es el número de átomos antes de la llamada a `B` y `N2` es el número de átomos después de la llamada a `B`

```
escribe_cuerpo(verdad,N,N).
escribe_cuerpo((A & B),N1,N3) :-
   escribe_cuerpo(A,N1,N2),
   escribe_cuerpo(B,N2,N3).
escribe_cuerpo(si(H,_),N,N1) :-
   escribe_lista(['   ',N,': ',H]),
   N1 is N+1.
```


Metaintérprete con CÓMO

- `interpreta_orden(+Orden,+B)` interpreta la `Orden` sobre el cuerpo `B`

```
interpreta_orden(N,B) :-
   integer(N),
   nth(B,N,E),
   navega(E).
```

- `nth(+E,+N,?A)` es verdad si `A` es el `N`–ésimo elemento de la estructura `E`

```
nth(A,1,A) :-
   not((A = (_,_))).
nth((A&_),1,A).
nth((_&B),N,E) :-
   N>1,
   N1 is N-1,
   nth(B,N1,E).
```

Tema 6: Ingeniería del conocimiento y metaintérpretes

Metaintérprete con Porqué

- Ejemplo: Modificación de `i_electrica.pl`

```
preguntable(arriba(_)).
preguntable(abajo(_)).
```

- Sesión

```
?- [meta_con_explicacion_porque, i_electrica_con_preguntas].
Yes

?- prueba_con_porque(esta_encendida(L)).
¿Es verdad arriba(i2)? (si/no/porque)
|: porque.
Se usa en: conectado(c0, c1) <- arriba(i2).
¿Es verdad arriba(i2)? (si/no/porque)
|: porque.
Se usa en: tiene_corriente(c0) <- conectado(c0, c1) & tiene_corriente(c1).
¿Es verdad arriba(i2)? (si/no/porque)
|: porque.
Se usa en: tiene_corriente(l1) <- conectado(l1, c0) & tiene_corriente(c0).
```

Metaintérprete con PORQUÉ

```
¿Es verdad arriba(i2)? (si/no/porque)
|: porque.
Se usa en: esta_encendida(l1) <- luz(l1) & esta_bien(l1) & tiene_corriente(l1).
¿Es verdad arriba(i2)? (si/no/porque)
|: porque.
Porque esa fue su pregunta!
¿Es verdad arriba(i2)? (si/no/porque)
|: si.
¿Es verdad arriba(i1)? (si/no/porque)
|: porque.
Se usa en: conectado(c1, c3) <- arriba(i1).
¿Es verdad arriba(i1)? (si/no/porque)
|: no.
¿Es verdad abajo(i2)? (si/no/porque)
|: no.
¿Es verdad arriba(i3)? (si/no/porque)
|: porque.
Se usa en: conectado(c4, c3) <- arriba(i3).
¿Es verdad arriba(i3)? (si/no/porque)
|: si.

L = l2 ;
No
```


Metaintérprete con PORQUÉ

- Ejemplo 2
  - Base de conocimiento

```
a   <- a1 & a2 & a3.
a1  <- a11 & a12.
a11 <- verdad.
a12 <- verdad.
a3  <- a31 & a32.
a31 <- verdad.
a32 <- verdad.

preguntable(a2).
```

  - Sesión

```
% prueba_cpa = prueba_con_porque_aux
?- trace(prueba_cpa,[call,exit]).
Yes
```

Metaintérprete con PORQUÉ

```
?- prueba_con_porque(a).
Call: 8) prueba_cpa(a, [])
Call: 9) prueba_cpa((a1&a2&a3), [(a<-a1&a2&a3)])
Call:10) prueba_cpa(a1, [(a<-a1&a2&a3)])
Call:11) prueba_cpa((a11&a12), [(a1<-a11&a12), (a<-a1&a2&a3)])
Call:12) prueba_cpa(a11, [(a1<-a11&a12), (a<-a1&a2&a3)])
Call:13) prueba_cpa(verdad, [(a11<-verdad), (a1<-a11&a12), (a<-a1&a2&a3)])
Exit:13) prueba_cpa(verdad, [(a11<-verdad), (a1<-a11&a12), (a<-a1&a2&a3)])
Exit:12) prueba_cpa(a11, [(a1<-a11&a12), (a<-a1&a2&a3)])
Call:12) prueba_cpa(a12, [(a1<-a11&a12), (a<-a1&a2&a3)])
Call:13) prueba_cpa(verdad, [(a12<-verdad), (a1<-a11&a12), (a<-a1&a2&a3)])
Exit:13) prueba_cpa(verdad, [(a12<-verdad), (a1<-a11&a12), (a<-a1&a2&a3)])
Exit:12) prueba_cpa(a12, [(a1<-a11&a12), (a<-a1&a2&a3)])
Exit:11) prueba_cpa((a11&a12), [(a1<-a11&a12), (a<-a1&a2&a3)])
Exit:10) prueba_cpa(a1, [(a<-a1&a2&a3)])
Call:10) prueba_cpa((a2&a3), [(a<-a1&a2&a3)])
Call:11) prueba_cpa(a2, [(a<-a1&a2&a3)])
```


Metaintérprete con PORQUÉ

```
¿Es verdad a2? (si/no/porque)
|: porque.
Se usa en:
   a <-
       a1 &
       a2 &
       a3.
¿Es verdad a2? (si/no/porque)
|: porque.
Porque esa fue su pregunta!
¿Es verdad a2? (si/no/porque)
|: si.
```

Metaintérprete con PORQUÉ

```
Exit:11) prueba_cpa(a2, [(a<-a1&a2&a3)])
Call:11) prueba_cpa(a3, [(a<-a1&a2&a3)])
Call:12) prueba_cpa((a31&a32), [(a3<-a31&a32), (a<-a1&a2&a3)])
Call:13) prueba_cpa(a31, [(a3<-a31&a32), (a<-a1&a2&a3)])
Call:14) prueba_cpa(verdad, [(a31<-verdad), (a3<-a31&a32), (a<-a1&a2&a3)])
Exit:14) prueba_cpa(verdad, [(a31<-verdad), (a3<-a31&a32), (a<-a1&a2&a3)])
Exit:13) prueba_cpa(a31, [(a3<-a31&a32), (a<-a1&a2&a3)])
Call:13) prueba_cpa(a32, [(a3<-a31&a32), (a<-a1&a2&a3)])
Call:14) prueba_cpa(verdad, [(a32<-verdad), (a3<-a31&a32), (a<-a1&a2&a3)])
Exit:14) prueba_cpa(verdad, [(a32<-verdad), (a3<-a31&a32), (a<-a1&a2&a3)])
Exit:13) prueba_cpa(a32, [(a3<-a31&a32), (a<-a1&a2&a3)])
Exit:12) prueba_cpa((a31&a32), [(a3<-a31&a32), (a<-a1&a2&a3)])
Exit:11) prueba_cpa(a3, [(a<-a1&a2&a3)])
Exit:10) prueba_cpa((a2&a3), [(a<-a1&a2&a3)])
Exit: 9) prueba_cpa((a1&a2&a3), [(a<-a1&a2&a3)])
Exit: 8) prueba_cpa(a, [])

Yes
```


Metaintérprete con PORQUÉ

- `prueba_con_porque(+O)` significa probar el objetivo `O`, con las respuestas del usuario, permitiéndole preguntar PORQUÉ se le plantean preguntas

```
prueba_con_porque(O) :-
   prueba_con_porque_aux(O,[]).
```

Metaintérprete con PORQUÉ

- `prueba_con_porque_aux(+O,+Antecesores)` se verifica si `O` es probable con la lista de `Antecesores`
  [Pregunta al usuario y le permite preguntar PORQUÉ]

```
prueba_con_porque_aux(verdad,_).
prueba_con_porque_aux((A & B), Antecesores) :-
   prueba_con_porque_aux(A,Antecesores),
   prueba_con_porque_aux(B,Antecesores).
prueba_con_porque_aux(O,_) :-
   preguntable(O),
   respuesta(O,si).
prueba_con_porque_aux(O,Antecesores) :-
   preguntable(O),
   no_preguntado(O),
   pregunta(O,Respuesta,Antecesores),
   assert(respuesta(O,Respuesta)),
   Respuesta=si.
prueba_con_porque_aux(A,Antecesores) :-
   (A <- B),
   prueba_con_porque_aux(B,[(A <- B)|Antecesores]).
```


Metaintérprete con PORQUÉ

- `pregunta(+O,-Respuesta,+Antecesores)` pregunta al usuario, en el contexto dado por los `Antecesores`, la cuestión `O` y éste responde la `Respuesta`

```
pregunta(O,Respuesta,Antecesores) :-
   escribe_lista(['¿Es verdad ',O,'? (si/no/porque)']),
   read(Replica),
   interpreta(O,Respuesta,Replica,Antecesores).
```

Metaintérprete con PORQUÉ

- `interpreta(+O,?Respuesta,+Replica,+Antecesores)`

```
interpreta(_,Replica,Replica,_) :-
   Replica \== porque.
interpreta(O,Respuesta,porque,[Regla|Reglas]):-
   write('Se usa en:'), nl,
   escribe_regla(Regla),
   pregunta(O,Respuesta,Reglas).
interpreta(O,Respuesta,porque,[]):-
   write('Porque esa fue su pregunta!'), nl,
   pregunta(O,Respuesta,[]).
```


Metaintérprete con PORQUÉ

- `escribe_regla(Regla)`

```
escribe_regla((A <- B)) :-
   escribe_lista(['   ',A, ' <- ']),
   escribe_cuerpo(B).
```

- `escribe_cuerpo(C)`

```
escribe_cuerpo((A & B)) :-
   escribe_lista(['       ',A, ' &']),
   escribe_cuerpo(B).
escribe_cuerpo(A) :-
   not(A = (_B & _C)),
   escribe_lista(['       ',A,'.']).
```

Metaintérprete con PORQUÉ

- Utilidad de saber porqué el sistema plantea una pregunta:
- Aumenta la confianza del usuario
- Ayuda al ingeniero del conocimiento en la optimización de las preguntas realizadas por el sistema
- Una pregunta irrelevante puede manifestar la existencia de problemas en el sistema
- El usuario puede usar el sistema para aprender

Tema 6: Ingeniería del conocimiento y metaintérpretes

Depuración de bases de conocimiento:

- Tipos de errores no sintácticos
  - Respuestas incorrectas
  - Respuestas perdidas
  - Bucle infinito
  - Preguntas irrelevantes

Tema 6: Ingeniería del conocimiento y metaintérpretes

Depuración de bases de conocimiento

- Depuración de respuestas incorrectas
  - Una respuesta incorrecta es una respuesta computada que es falsa en la interpretación deseada
  - Una respuesta incorrecta implica la existencia de una cláusula de la BC que es falsa en la interpretación deseada (o que el sistema de razonamiento es inadecuado)
  - Si q es falsa en la interpretación deseada, existe una demostración de q usando `q :- p1,...,pn.` Entonces
    1. alguna `pi` es falsa (y se depura), o
    2. todas las `pi` son verdaderas (y la cláusula es falsa)
- Ejemplo
  - BC errónea: en `i_electrica.pl` cambiar la cláusula
    `conectado(c1,c3) <- arriba(i1).`
    por la cláusula errónea
    `conectado(c1,c3) <- arriba(i3).`

Depuración de respuestas incorrectas

- Depuración de respuesta errónea con el metaintérprete con Cómo

```
?- prueba_con_como(esta_encendida(L)).
esta_encendida(l1) :-
   1: luz(l1)
   2: esta_bien(l1)
   3: tiene_corriente(l1)
|: 3.
tiene_corriente(l1) :-
   1: conectado(l1, c0)
   2: tiene_corriente(c0)
|: 2.
tiene_corriente(c0) :-
   1: conectado(c0, c1)
   2: tiene_corriente(c1)
|: 2.
tiene_corriente(c1) :-
   1: conectado(c1, c3)
   2: tiene_corriente(c3)
|: 1.
conectado(c1, c3) :-
   1: arriba(i3)
```

Bibliografía

1. Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).
   - Cap. 1: "Introduction"
   - Cap. 3: "Production rules and inference"
   - Cap. 4: "Tools for knowledge and inference inspection"
2. Merritt, D. *Building Expert Systems in Prolog* (Springer Verlag, 1989).
   - `www.amzi.com/ExpertSystemsInProlog`
3. Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)
   - Cap. 6: "Knowledge engineering"

Capítulo 7

Razonamiento por defecto y razonamiento abductivo

Tema 7: Razonamiento por defecto y razonamiento abductivo

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
http://www.cs.us.es/~jalonso

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Razonamiento por defecto

- **Ejemplo de razonamiento por defecto**

```
El animal_1 es un pájaro
Normalmente, los pájaros vuelan.
Por tanto, el animal_1 vuela.
```

- **Programa** `P1`
  - **Programa** `P1`

```
pájaro(animal_1).
vuela(X) :-
   pájaro(X),
   normal(X).
```

  - **Modelos del programa** `P1`

```
{pájaro(animal_1)}
{pájaro(animal_1), vuela(animal_1)}
{pájaro(animal_1), normal(animal_1), vuela(animal_1)}
```

  - **Consecuencia**

```
P1 |=/= vuela(animal_1)
```

Razonamiento por defecto

- **Programa `P2` con `anormal/1`**
  - **Programa `P2`**

```
:- dynamic anormal/1.

pájaro(animal_1).
vuela(X) :-
   pájaro(X),
   not anormal(X).
```

  - **Sesión**

```
?- vuela(animal_1).
Yes
```


Razonamiento por defecto

- **Traza**

```
?- vuela(animal_1).
   Call:  (  7) vuela(animal_1) ?
   Call:  (  8) pájaro(animal_1) ?
   Exit:  (  8) pájaro(animal_1) ?
 ^ Call:  (  8) not anormal(animal_1) ?
   Call:  (  9) anormal(animal_1) ?
   Fail:  (  9) anormal(animal_1) ?
 ^ Exit:  (  8) not anormal(animal_1) ?
   Exit:  (  7) vuela(animal_1) ?

Yes
```

Razonamiento por defecto

- **Extensión del conocimiento**
  - **Nuevo conocimiento**

```
El animal_1 es un avestruz.
Los avestruces son pájaros que no vuelan.
```

  - **Programa extendido**

```
:- dynamic anormal/1.

pájaro(animal_1).

avestruz(animal_1).

vuela(X) :-
   pájaro(X),
   not anormal(X).

anormal(X) :- avestruz(X).
```


Razonamiento por defecto

- **Traza**

```
?- vuela(animal_1).
   Call:  (  7) vuela(animal_1) ?
   Call:  (  8) pájaro(animal_1) ?
   Exit:  (  8) pájaro(animal_1) ?
 ^ Call:  (  8) not anormal(animal_1) ?
   Call:  (  9) anormal(animal_1) ?
   Call:  ( 10) avestruz(animal_1) ?
   Exit:  ( 10) avestruz(animal_1) ?
   Exit:  (  9) anormal(animal_1) ?
 ^ Fail:  (  8) not anormal(animal_1) ?
   Fail:  (  7) vuela(animal_1) ?

No
```

Razonamiento por defecto

- **Cancelación reglas por defectos mediante reglas específicas**
  - **Regla por defecto: "Normalmente, los pájaros vuelan"**
  - **Regla específica: "Los avestruces no vuelan"**
- **Razonamiento monótono y no–monótono**
  - **Razonamiento monótono**
    $P_1 \models C$ y P_2 extiende a P_1, entonces $P_2 \models C$.
  - **Razonamiento monótono**
    $P_1 \models C$ y P_2 extiende a P_1, entonces $P_2 \not\models C$.

Razonamiento por defecto

- **Programa con reglas y reglas con excepciones (defectos)**
  - **Programa objeto**

```
:- op(1100,xfx,<-).

defecto(vuela(X) <- pájaro(X)).

regla(pájaro(X) <- avestruz(X)).
regla(not(vuela(X)) <- avestruz(X)).
regla(avestruz(animal_1) <- verdad).
regla(pájaro(animal_2) <- verdad).
```

  - **Sesión**

```
?- explica(vuela(X),E).
X = animal_2
E = [defecto((vuela(animal_2) <- pájaro(animal_2))),
     regla((pájaro(animal_2) <- verdad))] ;
No
?- explica(not(vuela(X)),E).
X = animal_1
E = [regla((not(vuela(animal_1)) <- avestruz(animal_1))),
     regla((avestruz(animal_1) <- verdad))] ;
No
```

Razonamiento por defecto

- **Metaprograma para explicaciones**

```
explica(A,E) :- explica(A,[],E).

explica(verdad,E,E)               :- !.
explica((A,B),E0,E)               :- !, explica(A,E0,E1), explica(B,E1,E).
explica(A,E0,E)                   :- prueba(A,E0,E).
explica(A,E0,[defecto(A<-B)|E])   :- defecto(A<-B),
                                     explica(B,E0,E),
                                     \+ contradicción(A,E).

prueba(verdad,E,E)             :- !.
prueba((A,B),E0,E)             :- !, prueba(A,E0,E1), prueba(B,E1,E).
prueba(A,E0,[regla(A<-B)|E])   :- regla(A<-B), prueba(B,E0,E).

contradicción(not(A),E) :- !, prueba(A,E,_E1).
contradicción(A,E)      :- prueba(not(A),E,_E1).
```


Razonamiento por defecto

- **Explicaciones de hechos contradictorios**
  - **Programa objeto**

```
defecto(not(vuela(X))  <- mamífero(X)).
defecto(vuela(X)       <- vampiro(X)).
defecto(not(vuela(X))  <- muerto(X)).

regla(mamífero(X)      <- vampiro(X)).
regla(vampiro(dracula) <- verdad).
regla(muerto(dracula)  <- verdad).
```

Razonamiento por defecto

- **Sesión**

```
?- explica(vuela(dracula),E).
E = [defecto(vuela(dracula) <- vampiro(dracula)),
     regla(vampiro(dracula) <- verdad)] ;
No

?- explica(not(vuela(dracula),E)).
E = [defecto(not(vuela(dracula)) <- mamífero(dracula)),
     regla(mamífero(dracula) <- vampiro(dracula)),
     regla(vampiro(dracula) <- verdad)] ;
E = [defecto(not(vuela(dracula)) <- muerto(dracula)),
     regla(muerto(dracula) <- verdad)] ;
No
```


Razonamiento por defecto

- **Cancelación entre defectos mediante nombres**
  - **Programa objeto**

```
defecto(mamíferos_no_vuelan(X), (not(vuela(X)) <- mamífero(X))).
defecto(vampiros_vuelan(X),     (vuela(X) <- vampiro(X))).
defecto(muertos_no_vuelan(X),   (not(vuela(X)) <- muerto(X))).

regla(mamífero(X)      <- vampiro(X)).
regla(vampiro(dracula) <- verdad).
regla(muerto(dracula)  <- verdad).

regla(not(mamíferos_no_vuelan(X)) <- vampiro(X)).
regla(not(vampiros_vuelan(X))     <- muerto(X)).
```

Razonamiento por defecto

- **Modificación de** `explica`

```
explica(A,E0,[defecto(A<-B)|E]) :-
   defecto(Nombre,(A<-B)),
   explica(B,E0,E),
   \+ contradicción(Nombre,E),
   \+ contradicción(A,E).
```

- **Sesión**

```
?- explica(vuela(dracula),E).
No

?- explica(not vuela(dracula),E).
E = [defecto((not(vuela(dracula))<-muerto(dracula))),
     regla((muerto(dracula) <- verdad))]
Yes
```


Razonamiento abductivo

- **Problema de la abducción**

```
Dados     P un programa lógico y
          O una observación (un hecho básico en el lenguaje de P)
Encontrar E una abducción (una lista de hechos atómicos en el lenguaje de P
            cuyos predicados no son cabezas de cláusulas de P) tal que
            P U E |- O)
```

- **Abducción para programas definidos**
  - **Programa objeto**

```
europeo(X) <- español(X).
español(X) <- andaluz(X).
europeo(X) <- italiano(X).
```

  - **Sesión**

```
?- abducción(europeo(juan),E).
E = [andaluz(juan)] ;
E = [italiano(juan)] ;
No
```

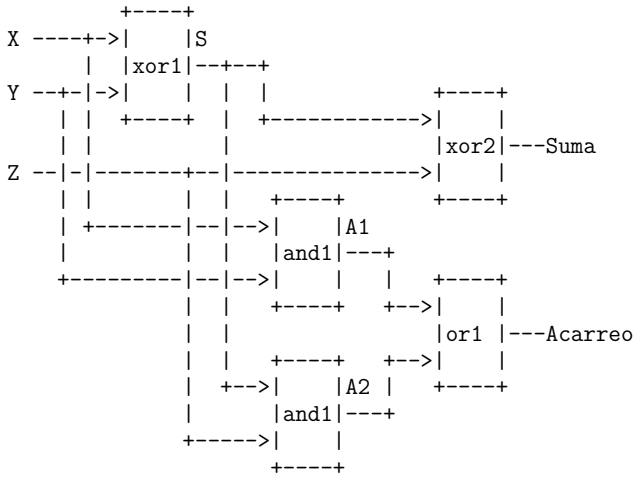
Razonamiento abductivo

- **Programa**

```
:- op(1200,xfx,<-).

abducción(O,E) :-
   abducción(O,[],E).

abducción(verdad,E,E) :- !.
abducción((A,B),E0,E) :- !,
   abducción(A,E0,E1),
   abducción(B,E1,E).
abducción(A,E0,E) :-
   (A <- B),
   abducción(B,E0,E).
abducción(A,E,E) :-
   member(A,E).
abducción(A,E,[A|E]) :-
   not member(A,E),
   explicable(A).

explicable(A) :-
   not (A <- _B).
```


Diagnóstico mediante abducción

- **Representación de un sumador**



Diagnóstico mediante abducción

- **Definición del sumador**

```
sumador(X,Y,Z,Acarreo,Suma) :-
   xor(X,Y,S),
   xor(Z,S,Suma),
   and(X,Y,A1),
   and(Z,S,A2),
   or(A1,A2,Acarreo).

and(1,1,1).  and(1,0,0). and(0,1,0).  and(0,0,0).
or(1,1,1).   or(1,0,1).  or(0,1,1).   or(0,0,0).
xor(1,0,1).  xor(0,1,1). xor(1,1,0).  xor(0,0,0).

tabla :-
   format('X Y Z A S~n'),
   tabla2.
tabla2 :-
   member(X,[0,1]), member(Y,[0,1]), member(Z,[0,1]),
   sumador(X,Y,Z,A,S),
   format('~a ~a ~a ~a ~a~n',[X,Y,Z,A,S]),
   fail.
tabla2.
```


Diagnóstico mediante abducción

- **Sesión con el sumador**

```
?- tabla.
X Y Z A S
0 0 0 0 0
0 0 1 0 1
0 1 0 0 1
0 1 1 1 0
1 0 0 0 1
1 0 1 1 0
1 1 0 1 0
1 1 1 1 1
Yes
```

Diagnóstico mediante abducción

- **Modelo de fallo del sumador**

```
sumador(X,Y,Z,Acarreo,Suma) <-
   xorg(xor1,X,Y,S),
   xorg(xor2,Z,S,Suma),
   andg(and1,X,Y,A1),
   andg(and2,Z,S,A2),
   org(or1,A1,A2,Acarreo).

xorg(_N,X,Y,Z) <- xor(X,Y,Z).
xorg(N,1,1,1)  <- fallo(N=f1).   xorg(N,0,0,1)  <- fallo(N=f1).
xorg(N,1,0,0)  <- fallo(N=f0).   xorg(N,0,1,0)  <- fallo(N=f0).
andg(_N,X,Y,Z) <- and(X,Y,Z).
andg(N,0,0,1)  <- fallo(N=f1).   andg(N,1,0,1)  <- fallo(N=f1).
andg(N,0,1,1)  <- fallo(N=f1).   andg(N,1,1,0)  <- fallo(N=f0).
org(_N,X,Y,Z)  <- or(X,Y,Z).
org(N,0,0,1)   <- fallo(N=f1).   org(N,1,0,0)   <- fallo(N=f0).
org(N,0,1,0)   <- fallo(N=f0).   org(N,1,1,0)   <- fallo(N=f0).
```


Diagnóstico mediante abducción

- **Diagnóstico mediante abducción**

```
diagnóstico(Observacion,Diagnóstico) :-
   abducción(Observacion,Diagnóstico).

:- abolish(explicable,2).
explicable(fallo(_X)).
```

- **Sesión de diagnóstico**

```
?- diagnóstico(sumador(0,0,1,1,0),D).
D = [fallo(or1 = f1),fallo(xor2 = f0)] ;
D = [fallo(and2 = f1),fallo(xor2 = f0)] ;
D = [fallo(and1 = f1),fallo(xor2 = f0)] ;
D = [fallo(and2 = f1),fallo(and1 = f1),fallo(xor2 = f0)] ;
D = [fallo(xor1 = f1)] ;
D = [fallo(or1 = f1),fallo(and2 = f0),fallo(xor1 = f1)] ;
D = [fallo(and1 = f1),fallo(xor1 = f1)] ;
D = [fallo(and2 = f0),fallo(and1 = f1),fallo(xor1 = f1)] ;
No
```

Diagnóstico mediante abducción

- **Diagnóstico mínimo**

```
diagnóstico_mínimo(O,D) :-
   diagnóstico(O,D),
   not((diagnóstico(O,D1),
       subconjunto_propio(D1,D))).

subconjunto_propio([],Ys):-
   Ys \= [].
subconjunto_propio([X|Xs],Ys):-
   select(Ys,X,Ys1),
   subconjunto_propio(Xs,Ys1).
```

- **Diagnóstico mínimo del sumador**

```
?- diagnóstico_mínimo(sumador(0,0,1,1,0),D).
D = [fallo(or1 = f1),fallo(xor2 = f0)] ;
D = [fallo(and2 = f1),fallo(xor2 = f0)] ;
D = [fallo(and1 = f1),fallo(xor2 = f0)] ;
D = [fallo(xor1 = f1)] ;
No
```


Bibliografía

- **Flach, P. "Simply Logical (Intelligent Reasoning by Example)" (John Wiley, 1994)**
  - **Cap. 8: "Reasoning incomplete information"**

- **Peischl, B. y Wotawa, F. "Model–Based Diagnosis or Reasonoing from First Principles" (IEEE Intelligent Systems, Vol 18, No. 3 (2003) p. 32–37)**

- **Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)**
  - **Cap. 9: "Assumption–Based Reasoning"**

Capítulo 8

Programación lógica con restricciones

Tema 8: Programación lógica con restricciones

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
http://www.cs.us.es/~jalonso

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sistemas de programación lógica con restricciones

- **Descripción del sistema que usaremos en el tema**
  - **ECLiPSe (ECLiPSe Common Logic Programming System)**
  - **es un sistema basado en Prolog**
  - **su objetivo es de servir como plataforma para integrar varias extensiones de la programación lógica, en particular la programación lógica con restricciones (CLP)**
  - **Llamada al sistema**

```
> eclipse
ECLiPSe Constraint Logic Programming System [kernel]
Copyright Imperial College London and ICL
Certain libraries copyright Parc Technologies Ltd
GMP library copyright Free Software Foundation
Version 5.7 #28, Mon Dec 22 00:13 2003
```

CLP sobre números reales: CLP(R)

- **Uso de la biblioteca CLP(R)**

```
[eclipse 1]: lib(clpr).
Yes (0.32s cpu)
```

- **Diferencia entre objetivo y restricción:**

```
[eclipse 2]: 1+X=5.
No (0.00s cpu)

[eclipse 3]: {1+X=5}.
X = 4.0
Yes (0.00s cpu)
```

- **Restricciones aritméticas:**
  - `<restricciones>` := {`<restricción 1>`, `<restricción 2>`, ...}
  - `<restricción>` := `<expresión 1>` `<operador>` `<expresión 2>`
    `<expresión 1>` y `<expresión 2>` son expresiones aritméticas
    `<operador>` es uno de los siguientes: `=`, `=\=`, `<`, `=<`, `>`, `>=`

CLP sobre números reales: CLP(R)

- **Ejemplo de programa en Prolog y en CLP(R)**
  - **Especificación:**
    `convierte(-C,+F)` se verifica si `C` son los grados centígrados correspondientes a `F` grados Fahrenheit; es decir, $C = \frac{(F-32)*5}{9}$.
  - **Programa Prolog**

```
convierte_1(C,F) :-
   C is (F-32)*5/9.
```

  - **Sesión con el programa Prolog**

```
[eclipse 4]: convierte_1(C,95).
C = 35
Yes (0.00s cpu)

[eclipse 5]: convierte_1(35,F).
instantiation fault in -(F, 32, _192) in module eclipse
Abort
```

CLP sobre números reales: CLP(R)

- **Programa CLP(R)**

```
:- lib(clpr).

convierte_2(C,F) :-
   {C = (F-32)*5/9}.
```

- **Sesión con el programa CLP(R)**

```
[eclipse 6]: convierte_2(C,95).
C = 35.0
Yes (0.00s cpu)

[eclipse 7]: convierte_2(35,F).
F = 95.0
Yes (0.00s cpu)

[eclipse 8]: convierte_2(C,F).
% Linear constraints:
{F = 32.0 + 1.7999999999999998 * C}
Yes (0.00s cpu)
```


CLP sobre números reales: CLP(R)

- **Ejemplo de ecuaciones e inecuaciones:**

```
[eclipse 9]: {3*X-2*Y=6, 2*Y=X}.
Y = 1.5
X = 3.0
Yes (0.00s cpu)

[eclipse 10]: {Z=<X-2, Z=<6-X, Z+1=2}.
X = X
Z = 1.0
% Linear constraints:
{X =< 5.0, X >= 3.0}
Yes (0.00s cpu)

[eclipse 11]: {X>0, X+2<0}.
No (0.00s cpu)
```

CLP sobre números reales: CLP(R)

- **Optimización con `minimize/1` y `maximize/1`:**

```
[eclipse 12]: {X=<5}, maximize(X).
X = 5.0
Yes (0.00s cpu)

[eclipse 13]: {X=<5, 2=<X}, minimize(2*X+3).
X = 2.0
Yes (0.00s cpu)

[eclipse 14]: {3=<X, X+1=<Y+2, Y=<9, Z=X+Y}, minimize(Z).
X = 3.0
Y = 2.0
Z = 5.0
Yes (0.00s cpu)

[eclipse 15]: {X+Y=<4}, maximize(X+Y).
% Linear constraints: {Y = 4.0 - X}
Yes (0.00s cpu)

[eclipse 16]: {X=<5}, minimize(X).
No (0.00s cpu)
```


CLP sobre números reales: CLP(R)

- **Optimización con `sup/2` e `inf/2`:**

```
[eclipse 17]: {2=<X, X=<5}, inf(X,I), sup(X,S).
I = 2.0
X = X
S = 5.0
% Linear constraints:
{X =< 5.0, X >= 2.0}
Yes (0.00s cpu)

[eclipse 18]: {3=<X, X+1=<Y+2, Y=<9, Z=X+Y}, inf(Z,I), sup(Z,S), minimize(Z).
X = 3.0
Y = 2.0
I = 5.0
S = 19.0
Z = 5.0
Yes (0.00s cpu)
```

CLP sobre números reales: CLP(R)

- **Planificación de tareas:**
  - **Problema: Calcular el menor tiempo necesario para realizar las tareas A, B, C y D teniendo en cuenta que los tiempos empleados en cada una son 2, 3, 5 y 4, respectivamente, y además que A precede a B y a C y que B precede a D.**
  - **Plan óptimo:**

```
[eclipse 19]: {Ta>=0, Ta+2=<Tb, Ta+2=<Tc, Tb+3=<Td, Tc+5=<Tf, Td+4=<Tf},
              minimize(Tf).
Ta = 0.0
Tb = 2.0
Tc = Tc
Td = 5.0
Tf = 9.0
% Linear constraints:
{Tc =< 4.0, Tc >= 2.0}
Yes (0.01s cpu)
```


CLP sobre números reales: CLP(R)

- **Sucesión de Fibonacci:**
  - **Especificación:**
    `fib(+N,-F)` **se verifica si** `F` **es el** `N`**-ésimo término de la sucesión de Fibonacci; es decir, 0, 1, 1, 2, 3, 5, 8, 13, 21, . . .**
  - **Programa Prolog**

```
fib_1(N,F) :-
   ( N=0, F=1
   ; N=1, F=1
   ; N>1,
     N1 is N-1, fib_1(N1,F1),
     N2 is N-2, fib_1(N2,F2),
     F is F1+F2 ).
```

  - **Sesión con el programa Prolog**

```
[eclipse 20]: fib_1(6,F).
F = 13
Yes (0.00s cpu, solution 1, maybe more) ?
[eclipse 21]: fib_1(N,13).
instantiation fault in N > 1 in module eclipse
Abort
```

CLP sobre números reales: CLP(R)

- **Programa CLP(R)**

```
:- lib(clpr).

fib_2(N,F) :-
   ( {N=0, F=1}
   ; {N=1, F=1}
   ; {N>1, F=F1+F2, N1=N-1, N2=N-2},
     fib_2(N1,F1),
     fib_2(N2,F2) ).
```

- **Sesión con el programa CLP(R)**

```
[eclipse 22]: fib_2(6,F).
F = 13.0
Yes (0.01s cpu, solution 1, maybe more) ?
[eclipse 23]: fib_2(N,13).
N = 6.0
Yes (0.02s cpu, solution 1, maybe more) ?
[eclipse 24]: fib_2(N,4).
interruption: type a, b, c, e, or h for help : ? a
abort
Aborting execution ...
Abort
```


CLP sobre números reales: CLP(R)

- **Modificación de `fib_2` para determinar los números que no son términos de la sucesión**

```
fib_3(N,F) :-
   ( {N=0, F=1}
   ; {N=1, F=1}
   ; {N>1, F=F1+F2, N1=N-1, N2=N-2, F1>=N1, F2>=N2},
     fib_3(N1,F1),
     fib_3(N2,F2) ).
```

- **Sesión**

```
[eclipse 25]: fib_3(6,F).
F = 13.0
Yes (0.02s cpu, solution 1, maybe more) ?
[eclipse 25]: fib_3(N,13).
N = 6.0
Yes (0.04s cpu, solution 1, maybe more) ?
[eclipse 27]: fib_3(N,4).
No (0.01s cpu)
```

CLP sobre números racionales: CLP(Q)

- **CLP sobre números racionales: CLP(Q)**

```
[eclipse 1]: lib(clpq).
Yes (0.66s cpu)

[eclipse 2]: {X=2*Y, Y=1-X}.
Y = 1 / 3
X = 2 / 3
Yes (0.00s cpu)
```


Planificación de tareas con CLP(Q)

- **Especificación de un problema mediante `tareas` y `precede`**
  - `tareas(+LTD)` **se verifica si** `LTD` **es la lista de los pares** `T/D` **de las tareas y sus duraciones.**

```
tareas([t1/5,t2/7,t3/10,t4/2,t5/9]).
```

  - `precede(+T1,+T2)` **se verifica si la tarea** `T1` **tiene que preceder a la** `T2`**.**

```
precede(t1,t2).
precede(t1,t4).
precede(t2,t3).
precede(t4,t5).
```

Planificación de tareas con CLP(Q)

- **Planificador**
  - **Biblioteca CLP(Q)**

```
:- lib(clpq).
```

  - `planificación(P,TP)` **se verifica si** `P` **es el plan (esto es una lista de elementos de la forma tarea/inicio/duración) para realizar las tareas en el menor tiempo posible y** `TP` **es dicho tiempo. Por ejemplo,**

```
[eclipse 3]: planificación(P,TP).
P = [t1/0/5, t2/5/7, t3/12/10, t4/I/2, t5/I/9]
TP = 22
% Linear constraints:
{ I =< 13, I >= 5, I - I >= 2 }
Yes (0.01s cpu)
```

```
planificación(P,TP) :-
   tareas(LTD),
   restricciones(LTD,P,TP),
   minimize(TP).
```


Planificación de tareas con CLP(Q)

  - `restricciones(LTD,P,TP)` **se verifica si** `P` **es un plan para realizar las tareas de** `LTD` **cumpliendo las restricciones definidas por** `precedencia/2` **y** `TP` **es el tiempo que se necesita para ejecutar el plan** `P`**.**

```
restricciones([],[],_TP).
restricciones([T/D | RLTD],[T/I/D | RTID],TP) :-
   {I >= 0, I + D =< TP},
   restricciones(RLTD,RTID,TP),
   restricciones_aux(T/I/D,RTID).
```

  - `restricciones_aux(TID,LTID)` **se verifica si el triple tarea–inicio–duración** `TID` **es consistente con la lista de triples tarea–inicio–duración** `LTID`**.**

```
restricciones_aux(_,[]).
restricciones_aux(T/I/D, [T1/I1/D1 | RTID]) :-
   ( precede(T,T1),!,{I+D =< I1}
   ; precede(T1,T),!,{I1+D1 =< I}
   ; true ),
   restricciones_aux(T/I/D,RTID).
```

Restricciones sobre dominios finitos: CLP(FD)

- **Restricciones de dominio y aritméticas**
  - **Ejemplos:**

```
[eclipse 1]: lib(fd).

[eclipse 2]: X :: 1..5, Y :: 0..4, X #< Y, Z #= X+Y+1.
Z = Z{[4..8]}    X = X{[1..3]}    Y = Y{[2..4]}
Delayed goals:  Y{[2..4]} - X{[1..3]}#>=1
                -1 - X{[1..3]} - Y{[2..4]} + Z{[4..8]}#=0

[eclipse 3]: [X,Y] :: 1..3, Z #=X+Y.
Z = Z{[2..6]}    X = X{[1..3]}    Y = Y{[1..3]}
Delayed goals:  0 - X{[1..3]} - Y{[1..3]} + Z{[2..6]}#=0
```

  - **Tipos de restricciones**
    * **de dominio de variables:** `<variable> :: <Mínimo>..<Máximo>`
    * **de dominio de lista:** `<lista> :: <Mínimo>..<Máximo>`
    * **aritmética:** `<expresión 1> <relación> <expresión 2>` **con** `<relación>` **en** `#=, #\=, #<, #>, #=<, #>=`.

Restricciones sobre dominios finitos: CLP(FD)

- **Relaciones de enumeración:**
  - `indomain(X)` **asigna un valor a la variable de dominio acotado** `X`**, en orden creciente. Por ejemplo,**

```
[eclipse 4]: X :: 1..3, indomain(X).
X = 1   Yes (0.00s cpu, solution 1, maybe more) ? ;
X = 2   Yes (0.01s cpu, solution 2, maybe more) ? ;
X = 3   Yes (0.01s cpu, solution 3)
```

  - `labeling(L)` **se verifica si existe una asignación que verifica las restricciones de las variables de la lista** `L`**. Por ejemplo,**

```
[eclipse 5]: L=[X,Y], L :: 1..2, labeling(L).
X = 1   Y = 1   L = [1, 1]   Yes (0.00s cpu, solution 1, maybe more) ? ;
X = 1   Y = 2   L = [1, 2]   Yes (0.00s cpu, solution 2, maybe more) ? ;
X = 2   Y = 1   L = [2, 1]   Yes (0.00s cpu, solution 3, maybe more) ? ;
X = 2   Y = 2   L = [2, 2]   Yes (0.00s cpu, solution 4)
```

  - `alldifferent(L)` **se verifica si todos las variables de la lista** `L` **tienen valores distintos. Por ejemplo,**

```
[eclipse 5]: L=[X,Y], L :: 1..2, alldifferent(L), labeling(L).
X = 1   Y = 2   L = [1, 2]   Yes (0.00s cpu, solution 1, maybe more) ? ;
X = 2   Y = 1   L = [2, 1]   Yes (0.00s cpu, solution 2)
```

Restricciones sobre dominios finitos: CLP(FD)

- **Problema de criptoaritmética**
  - **Especificación:** `solución([S,E,N,D],[M,O,R,E],[M,O,N,E,Y])` **se verifica si cada una de las letras se sustituye por un dígito distinto de forma que** `SEND+MORE=MONEY`**.**
  - **Programa**

```
:- lib(fd).

solución([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
   Vars = [S,E,N,D,M,O,R,Y], Vars :: 0..9,
   alldifferent(Vars),
           1000*S+100*E+10*N+D
         + 1000*M+100*O+10*R+E #=
   10000*M+1000*O+100*N+10*E+Y,
   M #\= 0, S #\= 0,
   labeling(Vars).
```

  - **Solución**

```
[eclipse 6]: solución(L1,L2,L3).
L1 = [9,5,6,7]   L2 = [1,0,8,5]   L3 = [1,0,6,5,2]
Yes (0.00s cpu, solution 1, maybe more) ? ;
No (0.00s cpu)
```


Restricciones sobre dominios finitos: CLP(FD)

- **Problema de las N reinas**
  - `solución(N,L)` **se verifica si** `L` **es una solución del problema de las** `N` **reinas. Por ejemplo,**

```
[eclipse 7]: solución(4,L).
L = [2, 4, 1, 3]   Yes (0.00s cpu, solution 1, maybe more) ? ;
L = [3, 1, 4, 2]   Yes (0.00s cpu, solution 2, maybe more) ? ;
No (0.00s cpu)
[eclipse 8]: findall(L,solución(8,L),LS), length(LS,N).
N = 92
Yes (0.02s cpu)
```

```
:- lib(fd).

solución(N,L) :-
   length(L,N),              % Hay N reinas
   L :: 1..N,                % Las ordenadas están en el intervalo 1..N
   alldifferent(L),          % Las ordenadas son distintas (distintas filas)
   segura(L),                % No hay en más de una en las diagonales
   labeling(L)    .          % Buscar los valores de L
```

Restricciones sobre dominios finitos: CLP(FD)

- `segura(L)` **se verifica si las reinas colocadas en las ordenadas** `L` **no se atacan diagonalmente.**

```
segura([]).
segura([Y|L]) :-
   no_ataca(Y,L,1),
   segura(L).
```

- `no_ataca(Y,L,D)` **se verifica si** `Y` **es un número,** `L` **es una lista de números** $[n_1, \ldots, n_m]$ **y** `D` **es un número tales que la reina colocada en la posición** `(x,Y)` **no ataca a las colocadas en las posiciones** $(x + d, n_1), \ldots, (x + d + m, n_m)$**.**

```
no_ataca(_Y,[],_).
no_ataca(Y1,[Y2|L],D) :-
   Y1-Y2 #\= D,
   Y2-Y1 #\= D,
   D1 is D+1,
   no_ataca(Y1,L,D1).
```


Restricciones sobre dominios finitos: CLP(FD)

- **Optimización:**
  - `minimize(P,V)` **busca una solución del problema** `P` **que minimiza el valor de** `V`**. Por ejemplo,**

```
[eclipse 9]: X :: 1..6, V #= X*(X-6), minimize(indomain(X),V).
Found a solution with cost -5
Found a solution with cost -8
Found a solution with cost -9

X = 3
V = -9
Yes (0.00s cpu)

[eclipse 10]: X :: 1..6, V #= X*(6-X), minimize(indomain(X),V).
Found a solution with cost 5
Found a solution with cost 0

X = 6
V = 0
Yes (0.00s cpu)
```

Restricciones sobre dominios finitos: CLP(FD)

- **Problemas de planificación óptima de tareas con CLP(FD):**
  - **Planificación**

```
[eclipse 11]: Tiempos_iniciales = [Ta,Tb,Tc,Td,Tf],
              Tiempos_iniciales :: 0..10,
              Ta+2 #=< Tb,
              Ta+2 #=< Tc,
              Tb+3 #=< Td,
              Tc+5 #=< Tf,
              Td+4 #=< Tf.

Tiempos_iniciales = [Ta{[0, 1]}, Tb{[2, 3]}, Tc{[2..5]}, Td{[5, 6]}, Tf{[9, 10]}]

Delayed goals:
 Tf{[9, 10]} - Tc{[2..5]}#>=5
 Tc{[2..5]} - Ta{[0, 1]}#>=2
 Tf{[9, 10]} - Td{[5, 6]}#>=4
 Td{[5, 6]} - Tb{[2, 3]}#>=3
 Tb{[2, 3]} - Ta{[0, 1]}#>=2
Yes (0.00s cpu)
```


Restricciones sobre dominios finitos: CLP(FD)

- **Traza:**

$Paso$		Ta	Tb	Tc	Td	Tf
		0..10	0..10	0..10	0..10	0..10
1	$Ta + 2 \leq Tb$	0..8	2..10			
2	$Tb + 3 \leq Td$		2..7		5..10	
3	$Td + 4 \leq Tf$				5..6	9..10
4	$Tb + 3 \leq Td$		2..3			
5	$Ta + 2 \leq Tb$	0..1				
6	$Ta + 2 \leq Tc$			2..10		
7	$Tc + 5 \leq Tf$			2..5		

Restricciones sobre dominios finitos: CLP(FD)

- **Optimización**

```
[eclipse 11]: Tiempos_iniciales = [Ta,Tb,Tc,Td,Tf],
              Tiempos_iniciales :: 0..20,
              Ta+2 #=< Tb,
              Ta+2 #=< Tc,
              Tb+3 #=< Td,
              Tc+5 #=< Tf,
              Td+4 #=< Tf,
              minimize(labeling(Tiempos_iniciales),Tf).
Found a solution with cost 9
Ta = 0
Tb = 2
Tc = 2
Td = 5
Tiempos_iniciales = [0, 2, 2, 5, 9]
Tf = 9
Yes (0.00s cpu)
```


Restricciones sobre dominios finitos: CLP(FD)

- **Reducción del espacio de búsqueda**
  - **Genera y prueba**

```
p(1).  p(3).  p(7).  p(16).  p(15).  p(14).
solución_1(X,Y,Z) :-
   p(X), p(Y), p(Z),
   q_1(X,Y,Z).
q_1(X,Y,Z) :-  Y =:= X+1, Z =:= Y+1.
```

  - **Estadísticas**

```
[eclipse 25]: lib(port_profiler).
[eclipse 26]: port_profile(solución_1(X,Y,Z),[]).
PREDICATE       call
p           /1    42
+           /3   218
=:=         /2   218
q_1         /3   208
solución_1  /3     1

X = 14   Y = 15   Z = 16
Yes (0.00s cpu)
```

Restricciones sobre dominios finitos: CLP(FD)

- **Restringe y genera**

```
:- lib(fd).

solución_2(X,Y,Z) :-
   q_2(X,Y,Z),
   p(X), p(Y), p(Z).

q_2(X,Y,Z) :- Y #= X+1, Z #= Y+1.
```

- **Estadísticas**

```
[eclipse 27]: port_profile(solución_2(X,Y,Z),[]).
PREDICATE       call
p           /1     9
q_2         /3     1
solución_2  /3     1

X = 14   Y = 15   Z = 16
Yes (0.00s cpu)
```


Bibliografía

- **I. Bratko *Prolog Programming for Artificial Intelligence (Third ed.)* (Prentice–Hall, 2001)**
  - **Cap 14: "Constraint logic programming"**
- **A.M. Cheadle, W. Harvey, A.J. Sadler, J. Schimpf, K. Shen y M.G. Wallace *ECLiPSe: An Introduction* (Imperial College London, Technical Report IC–Parc–03–1, 2003)**
- **K. Marriott y P.J. Stuckey *Programming with Constraints. An Introduction.* (The MIT Press, 1998).**

Capítulo 9

Formalización en Prolog de la lógica proposicional

Tema 9: Formalización en Prolog de la lógica proposicional

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
http://www.cs.us.es/~jalonso

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sintaxis de la lógica proposicional

- **Alfabeto proposicional:**
  - **símbolos proposicionales.**
  - **conectivas lógicas: $\neg$ (negación), $\wedge$ (conjunción), $\vee$ (disyunción), $\rightarrow$ (condicional), $\leftrightarrow$ (equivalencia).**
  - **símbolos auxiliares: "(" y ")".**
- **Fórmulas proposicionales:**
  - **símbolos proposicionales**
  - **$\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$**
- **Eliminación de paréntesis:**
  - **Eliminación de paréntesis externos.**
  - **Precedencia: $\neg, \wedge, \vee \rightarrow, \leftrightarrow$**
  - **Asociatividad: $\wedge$ y $\vee$ asocian por la derecha**

Sintaxis de la lógica proposicional

- **Sintaxis en Prolog**

Usual	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$
Prolog	-	&	v	=>	<=>

- **Declaración de operadores:**

```
:- op(610, fy,  -).     % negación
:- op(620, xfy, &).     % conjunción
:- op(630, xfy, v).     % disyunción
:- op(640, xfy, =>).    % condicional
:- op(650, xfy, <=>).   % equivalencia
```


Valores de verdad

- **Valores de verdad:**
  - **1: verdadero y 0: falso**
- **Def. de `valor_de_verdad`:**
  - `valor_de_verdad(?V)` **si** `V` **es un valor de verdad.**

```
valor_de_verdad(0).
valor_de_verdad(1).
```

Funciones de verdad

- **Funciones de verdad:**

i	$\neg i$
1	0
0	1

i	j	i ∧ j	i ∨ j	i → j	i ↔ j
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

  - `función_de_verdad(+Op, +V1, +V2, -V)` **si** `Op(V1,V2)=V`.
    `función_de_verdad(+Op, +V1, -V)` **si** `Op(V1)) = V`

```
función_de_verdad(v,   0, 0, 0) :- !.
función_de_verdad(v,   _, _, 1).
función_de_verdad(&,   1, 1, 1) :- !.
función_de_verdad(&,   _, _, 0).
función_de_verdad(=>,  1, 0, 0) :- !.
función_de_verdad(=>,  _, _, 1).
función_de_verdad(<=>, X, X, 1) :- !.
función_de_verdad(<=>, _, _, 0).

función_de_verdad(-,  1, 0).
función_de_verdad(-,  0, 1).
```


Valor de una fórmula

- **Representación de las interpretaciones**
  - **Listas de pares de variables y valores de verdad**
  - **Ejemplo:** `[(p,1),(r,0),(u,1)]`
- **Def. del valor de una fórmula en una interpretación**
  - `valor(+F, +I, -V)` **se verifica si el valor de la fórmula** `F` **en la interpretación** `I` **es** `V`
  - **Ejemplos:**

```
?- valor((p v q) & (-q v r),[(p,1),(q,0),(r,1)],V).
V = 1
?- valor((p v q) & (-q v r),[(p,0),(q,0),(r,1)],V).
V = 0
```

Valor de una fórmula

- **Def. de** `valor`**:**

```
valor(F, I, V) :-
   memberchk((F,V), I).
valor(-A, I, V) :-
   valor(A, I, VA),
   función_de_verdad(-, VA, V).
valor(F, I, V) :-
   functor(F,Op,2), arg(1,F,A), arg(2,F,B),   % F =..[Op,A,B],
   valor(A, I, VA),
   valor(B, I, VB),
   función_de_verdad(Op, VA, VB, V).
```


Interpretaciones de una fórmula

- ***I interpretación principal* de *F* syss *I* es una aplicación del conjunto de los símbolos proposicionales de *F* en el conjunto de los valores de verdad.**
- **Cálculo de las interpretaciones principales:**
  - `interpretaciones_fórmula(+F,-L)` **se verifica si** `L` **es el conjunto de las interpretaciones principales de la fórmula** `F`**.**
  - **Ejemplo**

```
?- interpretaciones_fórmula((p v q) & (-q v r),L).
L = [[ (p, 0),  (q, 0),  (r, 0)],
     [ (p, 0),  (q, 0),  (r, 1)],
     [ (p, 0),  (q, 1),  (r, 0)],
     [ (p, 0),  (q, 1),  (r, 1)],
     [ (p, 1),  (q, 0),  (r, 0)],
     [ (p, 1),  (q, 0),  (r, 1)],
     [ (p, 1),  (q, 1),  (r, 0)],
     [ (p, 1),  (q, 1),  (r, 1)]]
```

  - **Def. de** `interpretaciones_fórmula`**:**

```
interpretaciones_fórmula(F,U) :-
   findall(I,interpretación_fórmula(I,F),U).
```

Interpretaciones de una fórmula

- **Interpretación de una fórmula:**
  - `interpretación_fórmula(?I,+F)` **se verifica si** `I` **es una interpretación de la fórmula** `F`**.**
  - **Ejemplo:**

```
?- interpretación_fórmula(I,(p v q) & (-q v r)).
I = [ (p, 0),  (q, 0),  (r, 0)] ;
I = [ (p, 0),  (q, 0),  (r, 1)] ;
I = [ (p, 0),  (q, 1),  (r, 0)] ;
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 0)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

  - **Def. de** `interpretación_fórmula`**:**

```
interpretación_fórmula(I,F) :-
   símbolos_fórmula(F,U),
   interpretación_símbolos(U,I).
```


Interpretaciones de una fórmula

- **Símbolos de una fórmula**
  - `símbolos_fórmula(+F,?U)` **se verifica si** `U` **es el conjunto ordenado de los símbolos proposicionales de la fórmula** `F`**.**
  - **Ejemplo:**

```
?- símbolos_fórmula((p v q) & (-q v r), U).
U = [p, q, r]
```

  - **Def. de** `símbolo_fórmula`

```
símbolos_fórmula(F,U) :-
   símbolos_fórmula_aux(F,U1),
   sort(U1,U).
símbolos_fórmula_aux(F,[F]) :-
   atom(F).
símbolos_fórmula_aux(-F,U) :-
   símbolos_fórmula_aux(F,U).
símbolos_fórmula_aux(F,U) :-
   arg(1,F,A), arg(2,F,B),       % F =..[_Op,A,B],
   símbolos_fórmula_aux(A,UA),
   símbolos_fórmula_aux(B,UB),
   union(UA,UB,U).
```

Interpretaciones de una fórmula

- **Interpretación de una lista de símbolos:**
  - `interpretación_símbolos(+L,-I)` **se verifica si** `I` **es una interpretación de la lista de símbolos proposicionales** `L`**.**
  - **Ejemplo:**

```
?- interpretación_símbolos([p,q,r],I).
I = [ (p, 0),  (q, 0),  (r, 0)] ;
I = [ (p, 0),  (q, 0),  (r, 1)] ;
I = [ (p, 0),  (q, 1),  (r, 0)] ;
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 0)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

  - **Def. de** `interpretación_símbolos`

```
interpretación_símbolos([],[]).
interpretación_símbolos([A|L],[(A,V)|IL]) :-
   valor_de_verdad(V),
   interpretación_símbolos(L,IL).
```


Modelo de una fórmula

- **La interpretación *I* es un *modelo de la fórmula F* si el valor de *F* en *I* es verdadero.**
- **Comprobación de modelo de una fórmula:**
  - `es_modelo_fórmula(+I,+F)` **se verifica si la interpretación** `I` **es un modelo de la fórmula** `F`**.**
  - **Ejemplos:**

```
?- es_modelo_fórmula([(p,1),(q,0),(r,1)], (p v q) & (-q v r)).
Yes
?- es_modelo_fórmula([(p,0),(q,0),(r,1)], (p v q) & (-q v r)).
No
```

  - **Def. de** `es_modelo_fórmula`**:**

```
es_modelo_fórmula(I,F) :-
   valor(F,I,V),
   V = 1.
```

Cálculo de los modelos de una fórmula

- **Cálculo de los modelos principales de una fórmula:**
  - `modelo_fórmula(?I,+F)` **se verifica si** `I` **es un modelo principal de la fórmula** `F`**.**
  - **Ejemplo:**

```
?- modelo_fórmula(I,(p v q) & (-q v r)).
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

  - **Def. de** `modelo_fórmula`**:**

```
modelo_fórmula(I,F) :-
   interpretación_fórmula(I,F),
   es_modelo_fórmula(I,F).
```


Cálculo de los modelos de una fórmula

- `modelos_fórmula(+F,-L)` **se verifica si** `L` **es el conjunto de los modelos principales de la fórmula** `F`**.**
- **Ejemplo:**

```
?- modelos_fórmula((p v q) & (-q v r),L).
L = [[ (p, 0),  (q, 1),  (r, 1)],
     [ (p, 1),  (q, 0),  (r, 0)],
     [ (p, 1),  (q, 0),  (r, 1)],
     [ (p, 1),  (q, 1),  (r, 1)]]
```

- **Def. de** `modelos_fórmula`

```
modelos_fórmula(F,L) :-
   findall(I,modelo_fórmula(I,F),L).
```

Satisfacibilidad

- **Una fórmula es *satisfacible* si tiene modelo e *insatisfacible* si no lo tiene.**
- **Comprobación de satisfacibilidad:**
  - `es_satisfacible(+F)` **se verifica si la fórmula** `F` **es satisfacible.**
  - **Ejemplos:**

```
?- es_satisfacible((p v q) & (-q v r)).
Yes
?- es_satisfacible((p & q) & (p => r) & (q => -r)).
No
```

  - **Def. de** `es_satisfacible`**:**

```
es_satisfacible(F) :-
   interpretación_fórmula(I,F),
   es_modelo_fórmula(I,F).
```


Contramodelo de una fórmula

- **Un *contramodelo principal* de F es una interpretación principal de F que no es modelo de F.**
- **Cálculo de contramodelos:**
  - `contramodelo_fórmula(?I,+F)` **se verifica si** `I` **es un contramodelo principal de la fórmula** `F`**.**
  - **Ejemplos:**

```
?- contramodelo_fórmula(I, p <=> q).
I = [ (p, 0),  (q, 1)] ;
I = [ (p, 1),  (q, 0)] ;
No
?- contramodelo_fórmula(I, p => p).
No
```

  - **Def. de** `contramodelo_fórmula`**:**

```
contramodelo_fórmula(I,F) :-
   interpretación_fórmula(I,F),
   \+ es_modelo_fórmula(I,F).
```

Validez. Tautologías

- ***F* es *válida* (o *tautología*) si todas las interpretaciones son modelos de *F*.**
- **Comprobación de tautologías:**
  - `es_tautología(+F)` **se verifica si la fórmula** `F` **es una tautología.**
  - **Ejemplos:**

```
?- es_tautología((p => q) v (q => p)).
Yes
?- es_tautología(p => q).
No
```

  - **Def. de** `es_tautología`**:**

```
es_tautología(F) :-
   \+ contramodelo_fórmula(_I,F).
```

  - **Definición alternativa:**

```
es_tautología_alt(F) :-
   \+ es_satisfacible(-F).
```


Interpretaciones de conjuntos

- **Una *interpretación principal* de un conjunto de fórmulas es una aplicación del conjunto de sus símbolos proposicionales en el conjunto de los valores de verdad.**
- **Cálculo de las interpretaciones principales de un conjunto de fórmulas:**
  - `interpretaciones_conjunto(+S,-L)` **se verifica si** `L` **es el conjunto de las interpretaciones principales del conjunto** `S`**.**
  - **Ejemplo:**

```
?- interpretaciones_conjunto([p => q, q=> r],U).
U = [[ (p, 0),  (q, 0),  (r, 0)], [ (p, 0),  (q, 0),  (r, 1)],
     [ (p, 0),  (q, 1),  (r, 0)], [ (p, 0),  (q, 1),  (r, 1)],
     [ (p, 1),  (q, 0),  (r, 0)], [ (p, 1),  (q, 0),  (r, 1)],
     [ (p, 1),  (q, 1),  (r, 0)], [ (p, 1),  (q, 1),  (r, 1)]]
```

  - **Def. de** `interpretaciones_conjunto`**:**

```
interpretaciones_conjunto(S,U) :-
   findall(I,interpretación_conjunto(I,S),U).
```

Interpretaciones de conjuntos

- `interpretación_conjunto(?I,+S)` **se verifica si** `I` **es una interpretación del conjunto de fórmulas** `S`**.**
- **Ejemplo:**

```
?- interpretación_conjunto(I,[p => q, q=> r]).
I = [ (p, 0),  (q, 0),  (r, 0)] ;
I = [ (p, 0),  (q, 0),  (r, 1)] ;
I = [ (p, 0),  (q, 1),  (r, 0)] ;
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 0)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 0)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

- **Def. de** `interpretación_conjunto`**:**

```
interpretación_conjunto(I,S) :-
   símbolos_conjunto(S,U),
   interpretación_símbolos(U,I).
```


Interpretaciones de conjuntos

- **Cálculo de los símbolos de un conjunto de fórmulas:**
  - `símbolos_conjunto(+S,?U)` **se verifica si** `U` **es el conjunto ordenado de los símbolos proposicionales del conjunto de fórmulas** `S`**.**
  - **Ejemplo:**

```
?- símbolos_conjunto([p => q, q=> r],U).
U = [p, q, r]
```

  - **Def. de** `símbolos_conjunto`**:**

```
símbolos_conjunto(S,U) :-
   símbolos_conjunto_aux(S,U1),
   sort(U1,U).

símbolos_conjunto_aux([],[]).
símbolos_conjunto_aux([F|S],U) :-
   símbolos_fórmula(F,U1),
   símbolos_conjunto_aux(S,U2),
   union(U1,U2,U).
```

Modelo de un conjunto de fórmulas

- **La interpretación *I* es un *modelo del conjunto de fórmulas S* si *I* es modelo de todas las fórmulas de *S*.**
- **Comprobación de modelo de un conjunto de fórmulas**
  - `es_modelo_conjunto(+I,+S)` **se verifica si la interpretación** `I` **es un modelo del conjunto de fórmulas** `S`**.**
  - **Ejemplos:**

```
?- es_modelo_conjunto([(p,1),(q,0),(r,1)], [(p v q) & (-q v r),q => r]).
Yes
?- es_modelo_conjunto([(p,0),(q,1),(r,0)], [(p v q) & (-q v r),q => r]).
No
```

  - **Def. de** `es_modelo_conjunto`**:**

```
es_modelo_conjunto(_I,[]).
es_modelo_conjunto(I,[F|S]) :-
   es_modelo_fórmula(I,F),
   es_modelo_conjunto(I,S).
```


Cálculo de modelos de conjuntos

- **Cálculo de los modelos principales de conjuntos de fórmulas**
  - `modelo_conjunto(?I,+S)` **se verifica si** `I` **es un modelo principal del conjunto de fórmulas** `S`**.**
  - **Ejemplo:**

```
?- modelo_conjunto(I,[(p v q) & (-q v r),p => r]).
I = [ (p, 0),  (q, 1),  (r, 1)] ;
I = [ (p, 1),  (q, 0),  (r, 1)] ;
I = [ (p, 1),  (q, 1),  (r, 1)] ;
No
```

  - **Def. de** `modelo_conjunto`**:**

```
modelo_conjunto(I,S) :-
   interpretación_conjunto(I,S),
   es_modelo_conjunto(I,S).
```

Cálculo de modelos de conjuntos

- `modelos_conjunto(+S,-L)` **se verifica si** `L` **es el conjunto de los modelos principales del conjunto de fórmulas** `S`**.**
- **Ejemplo:**

```
?- modelos_conjunto([(p v q) & (-q v r),p => r],L).
L = [[ (p, 0),  (q, 1),  (r, 1)],
     [ (p, 1),  (q, 0),  (r, 1)],
     [ (p, 1),  (q, 1),  (r, 1)]]
```

- **Def. de** `modelos_conjunto`**:**

```
modelos_conjunto(S,L) :-
   findall(I,modelo_conjunto(I,S),L).
```


Consistencia de un conjunto

- **Un conjunto de fórmulas es *consistente* si tiene modelo e *inconsistente* en caso contrario.**
- **Comprobación de consistencia:**
  - `consistente(+S)` **se verifica si el conjunto de fórmulas** `S` **es consistente e** `inconsistente(+S)`**, si es inconsistente.**
  - **Ejemplos:**

```
?- consistente([(p v q) & (-q v r),p => r]).
Yes
?- consistente([(p v q) & (-q v r),p => r, -r]).
No
```

  - **Def. de** `consistente` **e** `inconsistente`**:**

```
consistente(S) :-
   modelo_conjunto(_I,S), !.

inconsistente(S) :-
   \+ modelo_conjunto(_I,S).
```

Consecuencia lógica

- **La fórmula F es *consecuencia lógica* del conjunto de fórmulas S ($S \models F$) si todos los modelos de S son modelos de F.**
- **($S \models F$) syss todas las interpretaciones principales de $S \cup \{F\}$ que son modelos de S también son modelos de F.**
- **Comprobación de consecuencia lógica:**
  - `es_consecuencia(+S,+F)` **se verifica si la fórmula** `F` **es consecuencia del conjunto de fórmulas** `S`**.**
  - **Ejemplos:**

```
?- es_consecuencia([p => q, q => r], p => r).
Yes
?- es_consecuencia([p], p & q).
No
```

  - **Def. de** `es_consecuencia`**:**

```
es_consecuencia(S,F) :-
   \+ contramodelo_consecuencia(S,F,_I).
```


Consecuencia lógica

- `contramodelo_consecuencia(+S,+F,?I)` **se verifica si** `I` **es una interpretación principal de** `S` $\cup$ {`F`} **que es modelo del conjunto de fórmulas** `S` **pero no es modelo de la fórmula** `F`**.**
- **Ejemplos:**

```
?- contramodelo_consecuencia([p], p & q, I).
I = [ (p, 1),  (q, 0)] ;
No
?- contramodelo_consecuencia([p => q, q=> r], p => r, I).
No
```

- **Def. de** `contramodelo_consecuencia`**:**

```
contramodelo_consecuencia(S,F,I) :-
   interpretación_conjunto(I,[F|S]),
   es_modelo_conjunto(I,S),
   \+ es_modelo_fórmula(I,F).
```

- **Definición alternativa de** `es_consecuencia`**:**

```
es_consecuencia_alt(S,F) :-
   inconsistente([-F|S]).
```

Ejemplo: veraces y mentirosos

- **El problema de los veraces y los mentirosos:**
  - **Enunciado: En una isla hay dos tribus, la de los veraces (que siempre dicen la verdad) y la de los mentirosos (que siempre mienten). Un viajero se encuentra con tres isleños A, B y C y cada uno le dice una frase**
    - **A dice "B y C son veraces syss C es veraz"**
    - **B dice "Si A y B son veraces, entonces B y C son veraces y A es mentiroso"**
    - **C dice "B es mentiroso syss A o B es veraz"**

    Determinar a qué tribu pertenecen A, B y C.
  - **Representación:**
    `a`, `b` **y** `c` **representan que A, B y C son veraces**
    `-a`, `-b` **y** `-c` **representan que A, B y C son mentirosos**

Ejemplo: veraces y mentirosos

- **Idea: las tribus se determinan a partir de los modelos del conjunto de fórmulas correspondientes a las tres frases.**

```
?- modelos_conjunto([a <=> (b & c <=> c),
                     b <=> (a & c => b & c & -a),
                     c <=> (-b <=> a v b)],
                    L).
L = [[ (a, 1), (b, 1), (c, 0)]]
```

- **Solución: A y B son veraces y C es mentiroso.**

Ejemplo: El problema de los animales

- **El problema de los animales**
  - **Enunciado: Disponemos de una base de conocimiento compuesta de reglas sobre clasificación de animales y hechos sobre características de un animal.**
    - **Regla 1: Si un animal es ungulado y tiene rayas negras, entonces es una cebra.**
    - **Regla 2: Si un animal rumia y es mamífero, entonces es ungulado.**
    - **Regla 3: Si un animal es mamífero y tiene pezuñas, entonces es ungulado.**
    - **Hecho 1: El animal tiene es mamífero.**
    - **Hecho 2: El animal tiene pezuñas.**
    - **Hecho 3: El animal tiene rayas negras.**

  Demostrar a partir de la base de conocimientos que el animal es una cebra.

Ejemplo: El problema de los animales

- **Solución:**

```
?- es_consecuencia(
     [es_ungulado & tiene_rayas_negras => es_cebra,
      rumia & es_mamifero => es_ungulado,
      es_mamifero & tiene_pezugnas => es_ungulado,
      es_mamifero,
      tiene_pezugnas,
      tiene_rayas_negras],
     es_cebra).
Yes
```

Ejemplo: Problema de los trabajos

- **El problema de los trabajos**
  - **Enunciado: Juan, Sergio y Carlos trabajan de programador, ingeniero y administrador (aunque no necesariamente en este orden). Juan le debe 1000 euros al programador. La esposa del administrador le ha prohibido a su marido pedir dinero prestado (y éste le obedece). Sergio está soltero. Determinar el trabajo de cada uno.**
  - **Representación:**
    `cp` **(Carlos es programador)**
    `ci` **(Carlos es ingeniero)**
    `ca` **(Carlos es administrador)**
    `jp` **(Juan es programador)**
    `ji` **(Juan es ingeniero)**
    `ja` **(Juan es administrador)**
    `sp` **(Sergio es programador)**
    `si` **(Sergio es ingeniero)**
    `sa` **(Sergio es administrador).**

Ejemplo: Problema de los trabajos

- **Solución:**

```
modelos_conjunto(
 [jp v ji v ja,  % Juan es programador, ingeniero o administrador
  sp v si v sa,  % Sergio es programador, ingeniero o administrador
  cp v ci v ca,  % Carlos es programador, ingeniero o administrador
  % No hay más de un programador:
  (jp & -sp & -cp) v (-jp & sp & -cp) v (-jp & -sp & cp),
  % No hay más de un ingeniero:
  (ji & -si & -ci) v (-ji & si & -ci) v (-ji & -si & ci),
  % No hay más de un administrador:
  (ja & -sa & -ca) v (-ja & sa & -ca) v (-ja & -sa & ca),
  % Juan le debe 1000 pesetas al programador [Luego, Juan no es el programador]:
  -jp,
  % La esposa del administrador le ha prohibido a su marido pedir dinero
  % prestado (y éste le obedece) [Luego, Juan no es el administrador]:
  -ja,
  % Sergio está soltero [Luego, no es el adminitrador]:
  -sa],
 L).
L = [[(ca,1),(ci,0),(cp,0), (ja,0),(ji,1),(jp,0), (sa,0),(si,0),(sp,1)]].
```

- **Conclusión: Carlos es administrador, Juan es ingeniero y Sergio es programador.**

Ejemplo: El problema de los cuadrados

- **El problema de los cuadrados**
  - **Enunciado: Existe nueve símbolos proposicionales que se pueden ordenar en un cuadrado. Se sabe que existe alguna letra tal que para todos los números las fórmulas son verdaderas (es decir, existe una fila de fórmulas verdaderas). El objetivo de este ejercicio demostrar que para cada número existe una letra cuya fórmula es verdadera (es decir, en cada columna existe una fórmula verdadera).**

```
a1  a2  a3
b1  b2  b3
c1  c2  c3
```

  - **Solución:**

```
?- es_tautología((a1 & a2 & a3) v
                 (b1 & b2 & b3) v
                 (c1 & c2 & c3)
                 =>
                 (a1 v b1 v c1) &
                 (a2 v b2 v c2) &
                 (a3 v b3 v c3)).
Yes
```


Ejemplo: Problema del coloreado

- **El problema del coloreado del pentágono (con dos colores)**
  - **Enunciado: Demostrar que es imposible colorear los vértices de un pentágono de rojo o azul de forma que los vértices adyacentes tengan colores distintos.**
  - **Representación:**
    - **1, 2, 3, 4, 5 representan los vértices consecutivos del pentágono**
    - **`ri` ($1 \leq$ `i` ≤ 5) representa que el vértice `i` es rojo**
    - **`ai` ($1 \leq$ `i` ≤ 5) representa que el vértice `i` es azul**

Ejemplo: Problema del coloreado

- **Solución:**

```
?- inconsistente(
    [% El vértice i (1 <= i <= 5) es azul o rojo:
     a1 v r1, a2 v r2, a3 v r3, a4 v r4, a5 v r5,

     % Un vértice no puede tener dos colores:
     a1 => -r1, r1 => -a1, a2 => -r2, r2 => -a2, a3 => -r3,
     r3 => -a3, a4 => -r4, r4 => -a4, a5 => -r5, r5 => -a5,

     % Dos vértices adyacentes no pueden ser azules:
     -(a1 & a2), -(a2 & a3), -(a3 & a4), -(a4 & a5), -(a5 & a1),

     % Dos vértices adyacentes no pueden ser rojos:
     -(r1 & r2), -(r2 & r3), -(r3 & r4), -(r4 & r5), -(r5 & r1)]).
Yes
```


Ejemplo: Problema del coloreado

- **El problema del coloreado del pentágono (con tres colores)**
  - **Enunciado: Demostrar que es posible colorear los vértices de un pentágono de rojo, azul o negro de forma que los vértices adyacentes tengan colores distintos.**
  - **Solución:**

```
?- modelo_conjunto(I,
    [% El vértice i (1 <= i <= 5) azul, rojo o negro:
     a1 v r1 v n1, a2 v r2 v n2, a3 v r3 v n3, a4 v r4 v n4, a5 v r5 v n5,

     % Un vértice no puede tener dos colores:
     a1 => -r1 & -n1, r1 => -a1 & -n1, n1 => -a1 & -r1,
     a2 => -r2 & -n2, r2 => -a2 & -n2, n2 => -a2 & -r2,
     a3 => -r3 & -n3, r3 => -a3 & -n3, n3 => -a3 & -r3,
     a4 => -r4 & -n4, r4 => -a4 & -n4, n4 => -a4 & -r4,
     a5 => -r5 & -n5, r5 => -a5 & -n5, n5 => -a5 & -r5,
```

Ejemplo: Problema del coloreado

```
        % Dos vértices adyacentes no pueden ser azules:
        -(a1 & a2), -(a2 & a3), -(a3 & a4), -(a4 & a5), -(a5 & a1),

        % Dos vértices adyacentes no pueden ser rojos:
        -(r1 & r2), -(r2 & r3), -(r3 & r4), -(r4 & r5), -(r5 & r1),

        % Dos vértices adyacentes no pueden ser negros:
        -(n1 & n2), -(n2 & n3), -(n3 & n4), -(n4 & n5), -(n5 & n1)]).
```

```
I = [ (a1,0),(a2,0),(a3,0),(a4,0),(a5,1),
      (n1,0),(n2,1),(n3,0),(n4,1),(n5,0),
      (r1,1),(r2,0),(r3,1),(r4,0),(r5,0)].
```

- **Conclusión: colorear el vértice 1 de rojo, el 2 de negro, el 3 de rojo, el 4 de negro y el 5 de azul.**

Ejemplo: Problema del palomar

- **El problema del palomar**
  - **Enunciado: Cuatro palomas comparten tres huecos. Demostrar que dos palomas tienen que estar en la misma hueco.**
  - **Representación:** `pihj` (`i`$= 1, 2, 3, 4$ y `j`$= 1, 2, 3$) **representa que la paloma** `i` **está en la hueco** `j`**.**

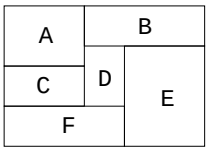
Ejemplo: Problema del palomar

- **Solución:**

```
?- inconsistente([
    % La paloma i está en alguna hueco:
    p1h1 v p1h2 v p1h3, p2h1 v p2h2 v p2h3,
    p3h1 v p3h2 v p3h3, p4h1 v p4h2 v p4h3,

    % No hay dos palomas en la hueco 1:
    -p1h1 v -p2h1, -p1h1 v -p3h1, -p1h1 v -p4h1,
    -p2h1 v -p3h1, -p2h1 v -p4h1, -p3h1 v -p4h1,

    % No hay dos palomas en la hueco 2:
    -p1h2 v -p2h2, -p1h2 v -p3h2, -p1h2 v -p4h2,
    -p2h2 v -p3h2, -p2h2 v -p4h2, -p3h2 v -p4h2,

    % No hay dos palomas en la hueco 3:
    -p1h3 v -p2h3, -p1h3 v -p3h3, -p1h3 v -p4h3,
    -p2h3 v -p3h3, -p2h3 v -p4h3, -p3h3 v -p4h3]).
Yes
```


Ejemplo: Problema de los rectángulos

- **El problema de los rectángulos**
  - **Enunciado: Un rectángulo se divide en seis rectángulos menores como se indica en la figura. Demostrar que si cada una de los rectángulos menores tiene un lado cuya medida es un número entero, entonces la medida de alguno de los lados del rectángulo mayor es un número entero.**



  - **Representación**
    - `base`: **la base del rectángulo mayor es un número entero**
    - `altura`: **la altura del rectángulo mayor es un número entero**
    - `base_x`: **la base del rectángulo X es un número entero**
    - `altura_x`: **la altura del rectángulo X es un número entero**

- **Solución:**

```
?- es_consecuencia(
     [base_a v altura_a, base_b v altura_b,
      base_c v altura_c, base_d v altura_d,
      base_e v altura_e, base_f v altura_f,
      base_a <=> base_c,
      base_a & base_d => base_f,
      base_f & base_a => base_d,
      base_f & base_d => base_a,
      base_d & base_e => base_b,
      base_b & base_d => base_e,
      base_b & base_e => base_d,
      base_a & base_b => base,
      base   & base_a => base_b,
      base   & base_b => base_a,
      base_a & base_d & base_e => base,
      base   & base_a & base_d => base_e,
      base   & base_a & base_e => base_d,
      base   & base_d & base_e => base_a,
      base_f & base_e => base,
      base   & base_f => base_e,
      base   & base_e => base_f,
```


Ejemplo: Problema de los rectángulos

```
      altura_d & altura_f => altura_e,
      altura_e & altura_d => altura_f,
      altura_e & altura_f => altura_d,
      altura_a & altura_c & altura_f => altura,
      altura   & altura_a & altura_c => altura_f,
      altura   & altura_a & altura_f => altura_c,
      altura   & altura_c & altura_f => altura_a,
      altura_b & altura_d & altura_f => altura,
      altura   & altura_b & altura_d => altura_f,
      altura   & altura_b & altura_f => altura_d,
      altura   & altura_d & altura_f => altura_b,
      altura_b & altura_e => altura,
      altura   & altura_b => altura_e,
      altura   & altura_e => altura_b],
     base v altura).
Yes
```

Ejemplo: Problema de las 4 reinas

- **El problema de las 4 reinas**
  - **Enunciado: Calcular las formas de colocar 4 reinas en un tablero de 4x4 de forma que no haya más de una reina en cada fila, columna o diagonal.**
  - **Representación:** `cij` **(**$1 \leq$ `i,j` ≤ 4**) indica que hay una reina en la fila** `i` **columna** `j`**.**

Ejemplo: Problema de las 4 reinas

- **Solución:**

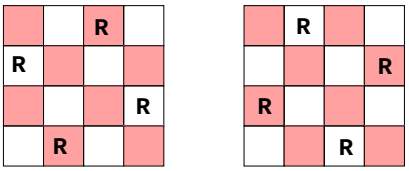
```
?- modelos_conjunto([
   % En cada fila hay una reina:
   c11 v c12 v c13 v c14, c21 v c22 v c23 v c24,
   c31 v c32 v c33 v c34, c41 v c42 v c43 v c44,
   % Si en una casilla hay reina, entonces no hay más reinas en sus líneas:
   c11 => (-c12 & -c13 & -c14) & (-c21 & -c31 & -c41) & (-c22 & -c33 & -c44),
   c12 => (-c11 & -c13 & -c14) & (-c22 & -c32 & -c42) & (-c21 & -c23 & -c34),
   c13 => (-c11 & -c12 & -c14) & (-c23 & -c33 & -c43) & (-c31 & -c22 & -c24),
   c14 => (-c11 & -c12 & -c13) & (-c24 & -c34 & -c44) & (-c23 & -c32 & -c41),
   c21 => (-c22 & -c23 & -c24) & (-c11 & -c31 & -c41) & (-c32 & -c43 & -c12),
   c22 => (-c21 & -c23 & -c24) & (-c12 & -c32 & -c42) & (-c11 & -c33 & -c44)
          & (-c13 & -c31),
   c23 => (-c21 & -c22 & -c24) & (-c13 & -c33 & -c43) & (-c12 & -c34)
          & (-c14  & -c32 & -c41),
   c24 => (-c21 & -c22 & -c23) & (-c14 & -c34 & -c44) & -c13 & (-c33 & -c42),
   c31 => (-c32 & -c33 & -c34) & (-c11 & -c21 & -c41) & -c42 & (-c13 & -c22),
   c32 => (-c31 & -c33 & -c34) & (-c12 & -c22 & -c42) & (-c21 & -c43)
          & (-c14  & -c23 & -c41),
```

Ejemplo: Problema de las 4 reinas

```
    c33 => (-c31 & -c32 & -c34) & (-c13 & -c23 & -c43) & (-c11 & -c22 & -c44)
           & (-c24 & -c42),
    c34 => (-c31 & -c32 & -c33) & (-c14 & -c24 & -c44) & (-c12 & -c23 & -c43),
    c41 => (-c42 & -c43 & -c44) & (-c11 & -c21 & -c31) & (-c14 & -c23 & -c32),
    c42 => (-c41 & -c43 & -c44) & (-c12 & -c22 & -c32) & (-c31 & -c24 & -c33),
    c43 => (-c41 & -c42 & -c44) & (-c13 & -c23 & -c33) & (-c21 & -c32 & -c34),
    c44 => (-c41 & -c42 & -c43) & (-c14 & -c24 & -c34) & (-c11 & -c22 & -c33)],
   L),
```

```
L = [[(c11,0),(c12,0),(c13,1),(c14,0),(c21,1),(c22,0),(c23,0),(c24,0),
      (c31,0),(c32,0),(c33,0),(c34,1),(c41,0),(c42,1),(c43,0),(c44,0)],
     [(c11,0),(c12,1),(c13,0),(c14,0),(c21,0),(c22,0),(c23,0),(c24,1),
      (c31,1),(c32,0),(c33,0),(c34,0),(c41,0),(c42,0),(c43,1),(c44,0)]]
```

- **Conclusión: Gráficamente los modelos son**



Ejemplo: Problema de Ramsey

- **El problema de Ramsey**
  - **Enunciado: Probar el caso más simple del teorema de Ramsey: entre seis personas siempre hay (al menos) tres tales que cada una conoce a las otras dos o cada una no conoce a ninguna de las otras dos.**
  - **Representación:**
    - **1,2,3,4,5,6 representan a las personas**
    - **`pij` ($1 \leq i < j \leq 6$) indica que las personas `i` y `j` se conocen.**
  - **Solución:**

```
?- es_tautología(
% Hay 3 personas que se conocen entre ellas:
(p12 & p13 & p23) v (p12 & p14 & p24) v (p12 & p15 & p25) v (p12 & p16 & p26) v
(p13 & p14 & p34) v (p13 & p15 & p35) v (p13 & p16 & p36) v (p14 & p15 & p45) v
(p14 & p16 & p46) v (p15 & p16 & p56) v (p23 & p24 & p34) v (p23 & p25 & p35) v
(p23 & p26 & p36) v (p24 & p25 & p45) v (p24 & p26 & p46) v (p25 & p26 & p56) v
(p34 & p35 & p45) v (p34 & p36 & p46) v (p35 & p36 & p56) v (p45 & p46 & p56) v
```

Ejemplo: Problema de Ramsey

```
% Hay 3 personas tales que cada una desconoce a las otras dos:
(-p12 & -p13 & -p23) v (-p12 & -p14 & -p24) v
(-p12 & -p15 & -p25) v (-p12 & -p16 & -p26) v
(-p13 & -p14 & -p34) v (-p13 & -p15 & -p35) v
(-p13 & -p16 & -p36) v (-p14 & -p15 & -p45) v
(-p14 & -p16 & -p46) v (-p15 & -p16 & -p56) v
(-p23 & -p24 & -p34) v (-p23 & -p25 & -p35) v
(-p23 & -p26 & -p36) v (-p24 & -p25 & -p45) v
(-p24 & -p26 & -p46) v (-p25 & -p26 & -p56) v
(-p34 & -p35 & -p45) v (-p34 & -p36 & -p46) v
(-p35 & -p36 & -p56) v (-p45 & -p46 & -p56)).
Yes
```


Ejemplos: Comparación

- **Comparación de la resolución de los problemas:**

Problema	Símbolos	Inferencias	Tiempo
mentirosos	3	646	0.00
animales	6	4,160	0.00
trabajos	9	71,044	0.07
cuadrados	9	56,074	0.06
pentágono_3	15	117,716	0.13
palomar	12	484,223	0.50
rectángulos	14	1,026,502	1.08
4 reinas	16	15,901,695	19.90
Ramsey	15	29,525,686	44.27

Bibliografía

- **Alonso, J.A. y Borrego, J.** ***Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)*** **(Ed. Kronos, 2002)**
  `www.cs.us.es/~jalonso/libros/da1-02.pdf`
  - **Cap. 3: Elementos de lógica proposicional**
- **Ben–Ari, M.** ***Mathematical Logic for Computer Science (2nd ed.)*** **(Springer, 2001)**
  - **Cap. 2: Propositional Calculus: Formulas, Models, Tableaux**
- **Fitting, M.** ***First-Order Logic and Automated Theorem Proving (2nd ed.)*** **(Springer, 1995)**
- **Nerode,A. y Shore, R.A.** ***Logic for Applications*** **(Springer, 1997)**

Capítulo 10

Programación lógica y aprendizaje automático

Tema 10: Programación lógica y aprendizaje automático

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
http://www.cs.us.es/~jalonso

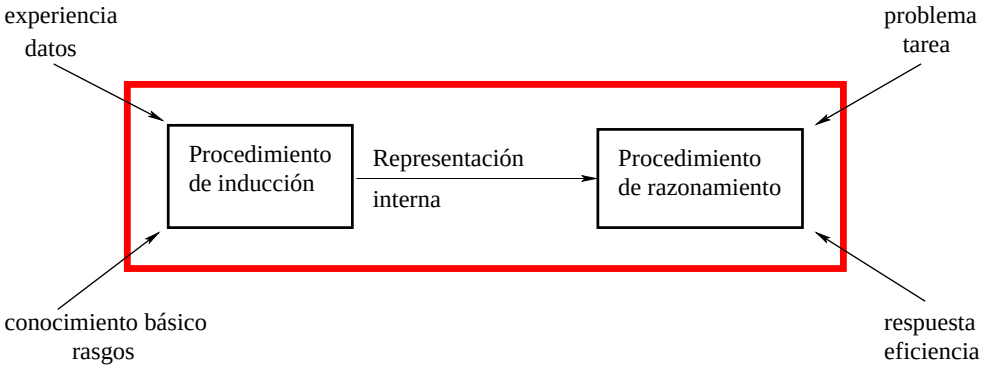
Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Aprendizaje automático

- **El aprendizaje automático estudia cómo contruir programas que mejoren automáticamente con la experiencia.**
- **Aspectos a mejorar:**
  - **La amplitud: hacer más tareas.**
  - **La calidad: hacer mejor.**
  - **La eficiencia: hacer más rápido.**

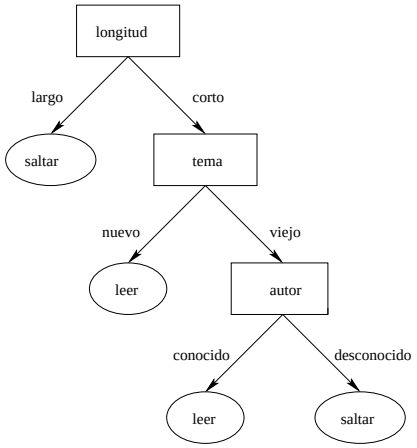
Arquitectura de aprendizaje automático



Ejemplo de datos

Ejemplo	Acción	Autor	Tema	Longitud	Sitio
e1	saltar	conocido	nuevo	largo	casa
e2	leer	desconocido	nuevo	corto	trabajo
e3	saltar	desconocido	viejo	largo	trabajo
e4	saltar	conocido	viejo	largo	casa
e5	leer	conocido	nuevo	corto	casa
e6	saltar	conocido	viejo	largo	trabajo
e7	saltar	desconocido	viejo	corto	trabajo
e8	leer	desconocido	nuevo	corto	trabajo
e9	saltar	conocido	viejo	largo	casa
e10	saltar	conocido	nuevo	largo	trabajo
e11	saltar	desconocido	viejo	corto	casa
e12	saltar	conocido	nuevo	largo	trabajo
e13	leer	conocido	viejo	corto	casa
e14	leer	conocido	nuevo	corto	trabajo
e15	leer	conocido	nuevo	corto	casa
e16	leer	conocido	viejo	corto	trabajo
e17	leer	conocido	nuevo	corto	casa
e18	leer	desconocido	nuevo	corto	trabajo

Árbol de decisión



Descripción de árboles de decisión

- **Elementos del árbol de decisión:**
  - **Nodos distintos de las hojas: atributos.**
  - **Arcos: posibles valores del atributo.**
  - **Hojas: clasificaciones.**
- **Reglas correspondientes a un árbol de decisión:**

```
longitud(E)=largo -> acción(E)=saltar
longitud(E)=corto y trama(E)=nuevo -> acción(E)=leer
longitud(E)=corto y trama(E)=viejo y autor(E)=conocido -> acción(E)=leer
longitud(E)=corto y trama(E)=viejo y autor(E)=desconocido -> acción(E)=saltar
```

- **Fórmula correspondiente a un árbol de decisión.**

Cuestiones sobre árboles de decisión

- **¿Cuál es el árbol de decisión correcto?:**
  - **Navaja de Occam.**
  - **El mundo es inherentemente simple.**
  - **El árbol de decisión más pequeño consistente con la muestra es el que tiene más probabilidades de identificar objetos desconocidos de manera correcta.**
  - **Menor profundidad.**
  - **Menor número de nodos.**
- **¿Cómo se puede construir el árbol de decisión más pequeño?**
  - **Búsqueda en escalada**

Búsqueda del árbol de decisión

- **¿Están todos los ejemplos en la misma clase?**

```
Ejemplos positivos = {E: acción(E)=leer} =
                       = {e2, e5, e8, e13, e14, e15, e16, e17, e18}
Ejemplos negativos = {E: acción(E)=saltar} =
                       = {e1, e3, e4, e6,  e7,  e9,  e10, e11, e12}
P = número de ejemplos positivos = 9
N = número de ejemplos negativos = 9
T = número total de ejemplos = P + N = 18
```

- **Información**
  - **Fórmula:** $\boldsymbol{I(P,N)} = \begin{cases} \mathbf{0}, & \textbf{si } \boldsymbol{N*P=0}; \\ -\frac{P}{T}\log_2\frac{P}{T} - \frac{N}{T}\log_2\frac{N}{T}, & \textbf{si } \boldsymbol{N*P\neq 0}. \end{cases}$
  - **Ejemplo:** $\boldsymbol{I(9,9)} = -\frac{9}{18}\log_2\frac{9}{18} - \frac{9}{18}\log_2\frac{9}{18} = 1$

Búsqueda del árbol de decisión

- **Información tras la división por un `Atributo`:**

  $$I = \frac{N1*I1+N2*I2}{N1+N2},$$

  donde

  - **N1 = número de ejemplos en la clase 1.**
  - **N2 = número de ejemplos en la clase 2.**
  - **I1 = cantidad de información en los ejemplos de la clase 1.**
  - **I2 = cantidad de información en los ejemplos de la clase 2.**

Búsqueda del árbol de decisión

- **Ganancia de información al dividir por autor.**
  - **Distribución:**

```
            | leer              | saltar
------------+-------------------+---------------------
conocido    | 05,13,14,15,16,17 | 01,04,06,09,10 12
desconocido | 02,08,18          | 03,07,11
```

  - **Información de `autor(E)=conocido`:** $I(6,6) = -\frac{6}{12}\log_2\frac{6}{12} - \frac{6}{12}\log_2\frac{6}{12} = 1$
  - **Información de `autor(E)=desconocido`:** $I(3,3) = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1$
  - **Información de la división por `autor`:** $\frac{12*1+6*1}{12+6} = 1$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir por tema.**
  - **Distribución:**

```
      | leer                 | saltar
------+----------------------+----------------------
nuevo | 02,05,08,14,15,17,18 | 01,10,12
viejo | 13,16                | 03,04,06,07,09,11
```

  - **Información de** `tema(E)=nuevo`: $I(7,3) = -\frac{7}{10}\log_2\frac{7}{10} - \frac{3}{10}\log_2\frac{3}{10} = 0.881$
  - **Información de** `tema(E)=viejo`: $I(2,6) = -\frac{2}{8}\log_2\frac{2}{8} - \frac{6}{8}\log_2\frac{6}{8} = 0.811$
  - **Información de la división por** `tema`: $\frac{10*0.881+8*0.811}{18} = 0.850$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir por longitud.**
  - **Distribución:**

```
         | leer                       | saltar
---------+----------------------------+----------------------
largo    |                            | 01,03,04,06,09,10,12
corto    | 02,05,08,13,14,15,16,17,18 | 07,11
```

  - **Información de** `longitud(E)=largo`: $I(0,7) = 0$
  - **Información de** `longitud(E)=corto`: $I(9,2) = -\frac{9}{11}\log_2\frac{9}{11} - \frac{2}{11}\log_2\frac{2}{11} = 0.684$
  - **Información de la división por** `longitud`: $\frac{7*0+11*0.684}{18} = 0.418$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir por sitio de lectura.**
  - **Distribución:**

	leer	saltar
casa	05,13,15,17	01,04,09,11
trabajo	02,08,14,16,18	03,06,07,10,12

  - **Información de** `sitio(E)=casa`: $I(4,4) = -\frac{4}{8}\log_2\frac{4}{8} - \frac{4}{8}\log_2\frac{4}{8} = 1$
  - **Información de** `sitio(E)=trabajo`: $I(5,5) = -\frac{5}{11}\log_2\frac{5}{11} - \frac{5}{11}\log_2\frac{1}{11} = 1$
  - **Información de la división por** `sitio`: $\frac{8*1+10*1}{18} = 1$

Búsqueda del árbol de decisión

- **Conclusiones:**
  - **Mejor atributo para dividir: longitud**
  - `Clase_1 = {E: longitud(E)=largo} = {} U {1,3,4,6,9,10,12}`
  - `Clase_2 = {E: longitud(E)=corto} = {2,5,8,13,14,15,16,17,18} U {7,11}`
- **Clasificación de** `Clase_1`**:**
  - **Están todos los ejemplos en la misma clase:** `acción(E)=saltar`**.**
- **Clasificación de** `Clase_2`**:**
  - **No están todos los ejemplos en la misma clase.**
  - **Repetir sobre** `Clase_2` **con los restantes atributos (autor, tema y sitio).**
  - **Información de** `longitud(E)=corto`: $I(9,2) = -\frac{9}{11}\log_2\frac{9}{11} - \frac{2}{11}\log_2\frac{2}{11} = 0.684$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir `Clase_2` por autor.**
  - **Distribución:**

```
            | leer              | saltar
------------+-------------------+----------------------
conocido    | 05,13,14,15,16,17 |
desconocido | 02,08,18          | 07,11
```

  - **Información de `autor(E)=conocido`:** $I(6,0) = 0$
  - **Información de `autor(E)=desconocido`:** $I(3,2) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.971$
  - **Información de la división por `autor`:** $\frac{6*0+5*0.971}{11} = 0.441$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir `Clase_2` por tema.**
  - **Distribución:**

```
       | leer                 | saltar
-------+----------------------+----------------------
nuevo  | 02,05,08,14,15,17,18 |
viejo  | 13,16                | 07,11
```

  - **Información de `tema(E)=nuevo`:** $I(7,0) = 0$
  - **Información de `tema(E)=viejo`:** $I(2,2) = -\frac{2}{4}\log_2\frac{2}{4} - \frac{2}{4}\log_2\frac{2}{4} = 1$
  - **Información de la división por `tema`:** $\frac{7*0+4*1}{11} = 0.364$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir `Clase_2` por sitio de lectura.**
  - **Distribución:**

	leer	saltar
casa	05,13,15,17	11
trabajo	02,08,14,16,18	07

  - **Información de `sitio(E)=casa`:** $I(4,1) = -\frac{4}{5}\log_2\frac{4}{5} - \frac{1}{5}\log_2\frac{1}{5} = 0.722$
  - **Información de `sitio(E)=trabajo`:** $I(5,1) = -\frac{5}{6}\log_2\frac{5}{6} - \frac{1}{6}\log_2\frac{1}{6} = 0.650$
  - **Información de la división por `sitio`:** $\frac{5*0.722+6*0.650}{11} = 0.683$

Búsqueda del árbol de decisión

- **Conclusiones:**
  - **Mejor atributo para dividir `Clase_2`: tema.**
  - `Clase_3 = {E: longitud(E)=corto, tema(E)=nuevo} = {2,5,8,14,15,17,18} U {}`
  - `Clase_4 = {E: longitud(E)=corto, tema(E)=viejo} = {13,16} U {7,11}`
- **Clasificación de `Clase_3`:**
  - **Están todos los ejemplos en la misma clase: `acción(E)=leer`.**
- **Clasificación de `Clase_4`:**
  - **No están todos los ejemplos en la misma clase.**
  - **Repetir sobre `Clase_4` con los restantes atributos (autor y sitio).**
  - **Información de `longitud(E)=corto, tema(E)=viejo`:** $I(2,2) = -\frac{2}{4}\log_2\frac{2}{4} - \frac{2}{4}\log_2\frac{2}{4} = 1$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir `Clase_4` por autor.**
  - **Distribución:**

```
            | leer  | saltar
------------+-------+-------
conocido    | 13,16 |
desconocido |       | 07,11
```

  - **Información de `autor(E)=conocido`:** $I(2,0) = 0$
  - **Información de `autor(E)=desconocido`:** $I(0,2) = 0$
  - **Información de la división por `autor`:** $\frac{2*0+2*0}{4} = 0$

Búsqueda del árbol de decisión

- **Ganancia de información al dividir `Clase_4` por sitio de lectura.**
  - **Distribución:**

```
        | leer | saltar
--------+------+-------
casa    | 13   | 11
trabajo | 16   | 07
```

  - **Información de `sitio(E)=casa`:** $I(1,1) = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$
  - **Información de `sitio(E)=trabajo`:** $I(1,1) = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$
  - **Información de la división por `sitio`:** $\frac{2*1+2*1}{4} = 1$

Búsqueda del árbol de decisión

- **Conclusiones:**
  - **Mejor atributo para dividir** `Clase_4`**: autor.**
  - `Clase_5 = {E: longitud(E)=corto, tema(E)=nuevo, autor(E)= conocido}`
    `= {13,16} U {}.`
  - `Clase_6 = {E: longitud(E)=corto, tema(E)=viejo, autor(E)= desconocido}`
    `= {} U {7,11}.`
- **Clasificación de** `Clase_5`**:**
  - **Están todos los ejemplos en la misma clase:** `acción(E)=leer`**.**
- **Clasificación de** `Clase_6`**:**
  - **Están todos los ejemplos en la misma clase:** `acción(E)=saltar`**.**

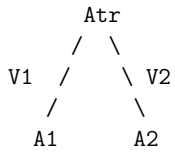
Algoritmo ID3

- **Sesión.**

```
?- ['aprende_ad.pl','aprende_ad_e1.pl'].
Yes

?- aprende_ad(acción,
                  [e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,e15,e16,e17,e18],
                  [autor,tema,longitud,sitio],
                  AD).
AD = si(longitud=largo, saltar,
            si(tema=nuevo, leer,
               si(autor=desconocido, saltar,
                  leer)))
Yes
```

Algoritmo ID3

1. **Si todos los ejemplos coinciden (es decir, si todos tienen el mismo valor en el atributo objetivo, entonces el árbol de decisión se reduce a una hoja con dicho valor.**
2. **En caso contrario,**
   (a) **elegir el atributo (`Atr`) con mayor ganancia de información,**
   (b) **sea `C1` el conjunto de ejemplos donde el atributo seleccionado toma un valor (`V1`) y `C2` el de los ejemplos donde toma el otro valor (`V2`),**
   (c) **sea `A1` el árbol construido aplicando el procedimiento a los ejemplos de `C1` y con el mismo atributo objetivo y `A2` el árbol construido aplicando el procedimiento a los ejemplos de `C2`,**
   (d) **el árbol de decisión es**



Algoritmo ID3

- **Representación del problema `aprende_ad_e1.pl`.**
  - `val(Objeto,Atributo,Valor)` **se verifica si el valor del `Atributo` del `Objeto` es `Valor`.**

```
val(e1,acción,saltar).
val(e1,autor,conocido).
val(e1,tema,nuevo).
val(e1,longitud,largo).
val(e1,sitio,casa ).

.........................

val(e18,acción,leer).
val(e18,autor,desconocido).
val(e18,tema,nuevo).
val(e18,longitud,corto).
val(e18,sitio,trabajo).
```

Algoritmo ID3

- **Algoritmo de aprendizaje de árboles de decisión.**
  - `aprende_ad(+Objetivo,+Ejemplos,+Atributos,-AD)` **se verifica si** `AD` **es el árbol de decisión inducido para el** `Objetivo` **a partir de la lista de** `Ejemplos` **y** `Atributos`**.**

```
aprende_ad(Objetivo, Ejemplos, _ , Val) :-
   coinciden_todos_ejemplos(Objetivo, Ejemplos, Val).
aprende_ad(Objetivo, Ejemplos, Atributos, si(Atr=ValPos,APos,ANeg)) :-
   not(coinciden_todos_ejemplos(Objetivo, Ejemplos, _)),
   selecciona_division(Objetivo, Ejemplos, Atributos, Atr, Resto_Atr),
   divide(Ejemplos, Atr, ValPos, Positivos, Negativos),
   aprende_ad(Objetivo, Positivos, Resto_Atr, APos),
   aprende_ad(Objetivo, Negativos, Resto_Atr, ANeg).
```


Algoritmo ID3

- **¿Están todos los ejemplos en la misma clase?**
  - `coindiden_todos_ejemplos(+Objetivo,+Ejemplos,-Valor)` **se verifica si el valor del atributo** `Objetivo` **de todos los** `Ejemplos` **es** `Valor`**.**

```
coinciden_todos_ejemplos(_,[],_).
coinciden_todos_ejemplos(Atributo,[Obj|Resto],Val) :-
   val(Obj,Atributo,Val),
   coinciden_todos_ejemplos(Atributo,Resto,Val).
```

Algoritmo ID3

- **Selección del mejor atributo para dividir.**
  - `selecciona_division(+Objetivo,+Ejemplos,+Atributos, -Atributo,-Restantes_atributos)` **se verifica si** `Atributo` **es el mejor elemento de la lista de** `Atributos` **para determinar el** `Objetivo` **a partir de los** `Ejemplos` **(es decir, la información resultante del** `Objetivo` **en los** `Ejemplos` **usando como división el** `Atributo` **es mínima), y** `Restantes_atributos` **es la lista de los restantes** `Atributos`**. Falla si para ningún atributo se gana en información.**

```
selecciona_division(Objetivo, Ejemplos, [A|R], Atributo, Resto_Atr) :-
    informacion_division(Objetivo,Ejemplos,A,I),
    selecciona_max_ganancia_informacion(Objetivo,Ejemplos,R,A,I,
                                        Atributo,[],Resto_Atr).
```


Algoritmo ID3

- `informacion_division(+Objetivo,+Ejemplos,+Atributo,-I)` **se verifica si** `I` **es la información resultante del** `Objetivo` **en los** `Ejemplos` **usando como división el** `Atributo`**; es decir,** `I = (N1*I1 + N2*I2) / (N1+N2)`.

```
informacion_division(Objetivo,Ejemplos,Atributo,Inf) :-
   divide(Ejemplos,Atributo,_,Clase_1,Clase_2),
   informacion(Objetivo,Clase_1,I1),
   informacion(Objetivo,Clase_2,I2),
   length(Clase_1,N1),
   length(Clase_2,N2),
   Inf is (N1*I1 + N2*I2)/(N1+N2).
```

Algoritmo ID3

- `informacion(+Objetivo,+Ejemplos,-I)` **se verifica si** `I` **es la cantidad de información en los** `Ejemplos` **respecto del** `Objetivo`**; es decir,**

$$\boldsymbol{I} = \begin{cases} \mathbf{0}, & \textbf{si } \boldsymbol{N} * \boldsymbol{P} = \mathbf{0}; \\ -\frac{\boldsymbol{P}}{\boldsymbol{T}} \log_2 \frac{\boldsymbol{P}}{\boldsymbol{T}} - \frac{\boldsymbol{N}}{\boldsymbol{T}} \log_2 \frac{\boldsymbol{N}}{\boldsymbol{T}}, & \textbf{si } \boldsymbol{N} * \boldsymbol{P} \neq \mathbf{0}. \end{cases}$$

```
informacion(Objetivo,Ejemplos,I) :-
   cuenta(Objetivo,_,Ejemplos,NP,NN),
   ( (NP=0 ; NN=0) ->
       I=0
   ; true ->
       NT is NP + NN,
       I is - NP/NT * log2(NP/NT) - NN/NT * log2(NN/NT)).
```


Algoritmo ID3

- `cuenta(+Atributo,?VP,+Ejemplos,-NP,-NN)` **se verifica si** `NP` **es el número de ejemplos positivos (es decir, elementos de** `Ejemplos` **tales que el valor de su** `Atributo` **es** `VP` **(valor positivo)) y** `NN` **es el número de ejemplos negativos.**

```
cuenta(_,_,[],0,0).
cuenta(Atributo,VP,[E|R],NP,NN) :-
   val(E,Atributo,VP),
   cuenta(Atributo,VP,R,NP1,NN),
   NP is NP1+1.
cuenta(Atributo,VP,[E|R],NP,NN) :-
   val(E,Atributo,VNeg),
   not(VP=VNeg),
   cuenta(Atributo,VP,R,NP,NN1),
   NN is NN1+1.
```

Algoritmo ID3

- `selecciona_max_ganancia_informacion(+Objetivo, +Ejemplos, +Atributos, +Mejor_atributo_actual, +Mejor_info_actual, -Atributo, +Atributos_analizados, -Resto_atributos)` **se verifica si** `Atributo` **es el elemento de** `Atributos` **tal la información resultante del** `Objetivo` **en los** `Ejemplos` **usando como división el** `Atributo` **es mínima.**

```
selecciona_max_ganancia_informacion(_,_,[],MejorA,_,
                                    MejorA, A_analizados, A_analizados).
selecciona_max_ganancia_informacion(Objetivo, Ejs, [A|R], MejorA, MejorI,
                                    Atributo, A_analizados, Resto_Atr) :-
   informacion_division(Objetivo,Ejs,A,Informacion),
   ( Informacion > MejorI ->
       selecciona_max_ganancia_informacion(
         Objetivo,Ejs,R,MejorA,MejorI,Atributo,[A|A_analizados],Resto_Atr)
   ;  true ->
       selecciona_max_ganancia_informacion(
         Objetivo,Ejs,R,A,Informacion,Atributo,[MejorA|A_analizados],Resto_Atr)
   ).
```


Algoritmo ID3

- **División de los ejemplos**
  - `divide(+Ejemplos,+Atributo,?Valor,-Positivos,-Negativos)` **se verifica si** `Positivos` **es la lista de elementos de** `Ejemplos` **tales que el valor de** `Atributo` **es** `Valor` **y** `Negativos` **es la lista de los restantes** `Ejemplos`**.**

```
divide([],_,_,[],[]).
divide([Ej|Rest],Atributo,ValPos,[Ej|Positivos],Negativos) :-
   val(Ej,Atributo,ValPos),
   divide(Rest,Atributo,ValPos,Positivos,Negativos).
divide([Ej|Rest],Atributo,ValPos,Positivos,[Ej|Negativos]) :-
   val(Ej,Atributo,ValNeg),
   not(ValNeg = ValPos),
   divide(Rest,Atributo,ValPos,Positivos,Negativos).
```

Sistemas TDIDP

- **Los sistemas basados en árboles de decisión forman una familia llamada TDIDT (*Top–Down Induction of Decision Tree*).**
- **Representantes de TDIDT:**
  - **ID3 (Interactive Dichotomizer) [Quinlan, 1986].**
  - **C4.5 [Quinlan, 93] es una variante de ID3 que permite clasificar ejemplos con atributos que toman valores continuos.**
- **Quinlan, J. R. *C4.5: Programs for Machine Learning* (Morgan Kaufmann, 1993).**

Limitaciones de árboles de decisión

- **Una representación formal limitada (lenguaje de pares atributo–valor equivalente al de la lógica proposicional).**
- **Tienden a ser demasiado grandes en aplicaciones reales.**
- **Dificultad del manejo del conocimiento base.**

Programación Lógica Inductiva

- **Datos:**
  - Ejemplos positivos: $\boldsymbol{E}^{\oplus}$.
  - Ejemplos negativos: $\boldsymbol{E}^{\ominus}$.
  - Conocimiento base: $\boldsymbol{T}$.
  - Lenguaje de hipótesis: $\boldsymbol{L}$.
- **Condiciones:**
  - ***Necesidad a priori:*** $(\exists e^{\oplus} \in \boldsymbol{E}^{\oplus})[\boldsymbol{T} \nvdash e^{\oplus}]$.
  - ***Consistencia a priori:*** $(\forall e^{\ominus} \in \boldsymbol{E}^{\ominus})[\boldsymbol{T} \nvdash e^{\ominus}]$.
- **Objetivo:**
  - Encontrar un conjunto finito $\boldsymbol{H} \subset \boldsymbol{L}$ tal que se cumplan
    - ***Suficiencia a posteriori:*** $(\forall e^{\oplus} \in \boldsymbol{E}^{\oplus})[\boldsymbol{T} \bigcup \boldsymbol{H} \vdash e^{\oplus}]$.
    - ***Consistencia a posteriori:*** $(\forall e^{\ominus} \in \boldsymbol{E}^{\ominus})[\boldsymbol{T} \bigcup \boldsymbol{H} \nvdash e^{\ominus}]$.

Ejemplo con FOIL

- **Descripción del problema `familia.pl`.**
  - Ejemplos positivos:

```
padre(carlos,juan).        padre(carlos,eva).
```

  - Ejemplos negativos:

```
no(padre(carlos,aurora)).  no(padre(aurora,juan)).
```

  - Conocimiento básico:

```
hombre(carlos).            hombre(juan).
mujer(eva).                mujer(aurora).
progenitor(carlos,juan).   progenitor(carlos,eva).
progenitor(aurora,juan).   progenitor(aurora,eva).
```

  - Parámetros:

```
foil_predicates([padre/2,hombre/1,mujer/1,progenitor/2]).
foil_cwa(false).           % No usa la hipótesis del mundo cerrado
foil_use_negations(false). % No usa información negativa en el cuerpo
foil_det_lit_bound(0).     % No añade literales determinados
```

Ejemplo con FOIL

- **Sesión:**

```
?- [foil, familia].
Yes

?- foil(padre/2).

Uncovered positives: [padre(carlos, juan), padre(carlos, eva)]

Adding a clause ...
Specializing current clause: padre(A, B).

Covered negatives: [padre(carlos, aurora), padre(aurora, juan)]

Covered positives: [padre(carlos, juan), padre(carlos, eva)]
```


Ejemplo con FOIL

```
Ganancia:  0.830 Cláusula: padre(A, B):-hombre(A)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(B)
Ganancia:  0.000 Cláusula: padre(A, B):-mujer(A)
Ganancia:  0.000 Cláusula: padre(A, B):-mujer(B)
Ganancia:  0.000 Cláusula: padre(A, B):-progenitor(C, A)
Ganancia:  0.000 Cláusula: padre(A, B):-progenitor(A, C)
Ganancia:  0.830 Cláusula: padre(A, B):-progenitor(C, B)
Ganancia:  0.000 Cláusula: padre(A, B):-progenitor(B, C)
Ganancia:  0.000 Cláusula: padre(A, B):-progenitor(A, A)
Ganancia:  0.000 Cláusula: padre(A, B):-progenitor(B, A)
Ganancia:  0.830 Cláusula: padre(A, B):-progenitor(A, B)
Ganancia:  0.000 Cláusula: padre(A, B):-progenitor(B, B)

Specializing current clause: padre(A, B) :- hombre(A).

Covered negatives: [padre(carlos, aurora)]

Covered positives: [padre(carlos, juan), padre(carlos, eva)]
```

Ejemplo con FOIL

```
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), hombre(A)
Ganancia:  0.585 Cláusula: padre(A, B):-hombre(A), hombre(B)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), mujer(A)
Ganancia: -0.415 Cláusula: padre(A, B):-hombre(A), mujer(B)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), progenitor(C, A)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), progenitor(A, C)
Ganancia:  1.170 Cláusula: padre(A, B):-hombre(A), progenitor(C, B)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), progenitor(B, C)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), progenitor(A, A)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), progenitor(B, A)
Ganancia:  1.170 Cláusula: padre(A, B):-hombre(A), progenitor(A, B)
Ganancia:  0.000 Cláusula: padre(A, B):-hombre(A), progenitor(B, B)

Clause found: padre(A, B) :- hombre(A), progenitor(A, B).

Found definition: padre(A, B) :- hombre(A), progenitor(A, B).
```


Ganancia de información en FOIL

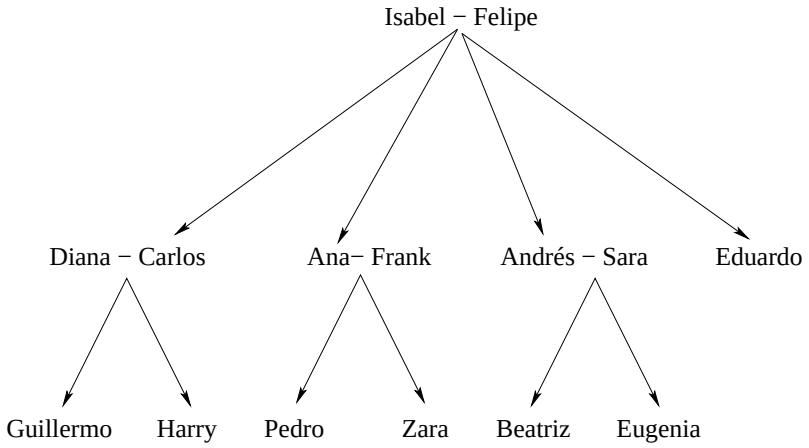
- **Información correspondiente a una cláusula:**
  Sean P el número de ejemplos positivos cubiertos por la cláusula y N el número de ejemplos negativos cubiertos por la cláusula.

$$I(C) = \begin{cases} 0, & \text{si } P = 0; \\ -\log_2 \frac{P}{P+N}, & \text{si } P \neq 0. \end{cases}$$

- **Ganancia de información de la cláusula C_1 a la cláusula C_2:**
  Sean P el número de ejemplos positivos cubiertos por C_2 y N el número de ejemplos negativos cubiertos C_2.

$$G(C_1, C_2) = \begin{cases} 0, & \text{si } P = 0; \\ P * (I(C_1) - I(C_2)), & \text{si } P \neq 0. \end{cases}$$

Ejemplo de FOIL con CWA



Ejemplo de FOIL con CWA

- **Representación** `familia_1.pl`**:**
  - **Ejemplos positivos**

```
padre(felipe,carlos).      padre(felipe,ana).     padre(felipe,andres).  ...
madre(isabel,carlos).      madre(isabel,ana).     madre(isabel,andres).  ...
abuelo(felipe,guillermo).  abuelo(felipe,harry).  abuelo(felipe,pedro).  ...
```

  - **Parámetros**

```
foil_predicates([padre/2, madre/2, abuelo/2]).
foil_cwa(true).              % Usa la hipótesis del mundo cerrado
foil_use_negations(false).   % No usa información negativa en el cuerpo
foil_det_lit_bound(0).       % No añade literales determinados
```

Ejemplo de FOIL con CWA

- **Sesión**

```
?- [foil, familia_1].
Yes
?- foil(abuelo/2).

Uncovered positives: [(felipe,guillermo),(felipe,harry),(felipe,pedro),
                      (felipe,zara),(felipe,beatriz),(felipe,eugenia)]

Adding a clause ...
Specializing current clause: abuelo(A,B).

Covered negatives: [(ana,ana),(ana,andres),...]
Covered positives: [(felipe,guillermo),(felipe,harry),(felipe,pedro),
                    (felipe,zara),(felipe,beatriz),(felipe,eugenia)]
```


Ejemplo de FOIL con CWA

```
Ganancia:  0.000 Cláusula: abuelo(A,B):-padre(C,A)
Ganancia: 15.510 Cláusula: abuelo(A,B):-padre(A,C)
Ganancia:  3.510 Cláusula: abuelo(A,B):-padre(C,B)
Ganancia:  0.000 Cláusula: abuelo(A,B):-padre(B,C)
Ganancia:  0.000 Cláusula: abuelo(A,B):-padre(A,A)
Ganancia:  0.000 Cláusula: abuelo(A,B):-padre(B,A)
Ganancia:  0.000 Cláusula: abuelo(A,B):-padre(A,B)
Ganancia:  0.000 Cláusula: abuelo(A,B):-padre(B,B)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(C,A)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(A,C)
Ganancia:  3.510 Cláusula: abuelo(A,B):-madre(C,B)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(B,C)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(A,A)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(B,A)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(A,B)
Ganancia:  0.000 Cláusula: abuelo(A,B):-madre(B,B)

Specializing current clause: abuelo(A,B) :- padre(A,C).
```

Ejemplo de FOIL con CWA

```
Covered negatives: [(andres,ana),(andres,andres),...]
Covered positives: [(felipe,guillermo),(felipe,harry),(felipe,pedro),
                    (felipe,zara),(felipe,beatriz),(felipe,eugenia)]

Ganancia:  3.087 Cláusula: abuelo(A,B):-padre(A,C),padre(A,D)
Ganancia:  3.510 Cláusula: abuelo(A,B):-padre(A,C),padre(D,B)
Ganancia:  7.932 Cláusula: abuelo(A,B):-padre(A,C),padre(C,D)
Ganancia: 10.575 Cláusula: abuelo(A,B):-padre(A,C),padre(C,B)
Ganancia:  3.510 Cláusula: abuelo(A,B):-padre(A,C),madre(D,B)
Ganancia:  7.932 Cláusula: abuelo(A,B):-padre(A,C),madre(C,D)
Ganancia:  5.288 Cláusula: abuelo(A,B):-padre(A,C),madre(C,B)
....

Clause found: abuelo(A,B) :- padre(A,C),padre(C,B).
```


Ejemplo de FOIL con CWA

```
Uncovered positives: [(felipe,pedro),(felipe,zara)]

Adding a clause ...
Specializing current clause: abuelo(A,B).

Covered negatives: [(ana,ana),(ana,andres),...]
Covered positives: [(felipe,pedro),(felipe,zara)]

Ganancia:  5.444 Cláusula: abuelo(A,B):-padre(A,C)
Ganancia:  1.196 Cláusula: abuelo(A,B):-padre(C,B)
Ganancia:  1.196 Cláusula: abuelo(A,B):-madre(C,B)
...

Specializing current clause: abuelo(A,B) :- padre(A,C).

Covered negatives: [(andres,ana),(andres,andres),...]
Covered positives: [(felipe,pedro),(felipe,zara)]
```

Ejemplo de FOIL con CWA

```
Ganancia:  1.181 Cláusula: abuelo(A,B):-padre(A,C),padre(A,D)
Ganancia:  1.348 Cláusula: abuelo(A,B):-padre(A,C),padre(D,B)
Ganancia:  3.213 Cláusula: abuelo(A,B):-padre(A,C),padre(C,D)
Ganancia:  1.348 Cláusula: abuelo(A,B):-padre(A,C),madre(D,B)
Ganancia:  3.213 Cláusula: abuelo(A,B):-padre(A,C),madre(C,D)
Ganancia:  8.132 Cláusula: abuelo(A,B):-padre(A,C),madre(C,B)
...

Clause found: abuelo(A,B) :- padre(A,C),madre(C,B).

Found definition:
abuelo(A,B) :- padre(A,C),madre(C,B).
abuelo(A,B) :- padre(A,C),padre(C,B).
```


Ejemplo de FOIL con CWA y conocimiento básico

- **Representación** `familia_2.pl`**:**
  - **Ejemplos positivos:**

```
padre(felipe,carlos).      padre(felipe,ana).     padre(felipe,andres).  ...
madre(isabel,carlos).      madre(isabel,ana).     madre(isabel,andres).  ...
abuelo(felipe,guillermo).  abuelo(felipe,harry).  abuelo(felipe,pedro).  ...
```

  - **Conocimiento básico:**

```
progenitor(X,Y) :- padre(X,Y).
progenitor(X,Y) :- madre(X,Y).
```

  - **Parámetros:**

```
foil_predicates([padre/2, madre/2, abuelo/2, progenitor/2]).
foil_cwa(true).              % Usa la hipótesis del mundo cerrado
foil_use_negations(false).   % No usa información negativa en el cuerpo
foil_det_lit_bound(0).       % No añade literales determinados
```

Ejemplo de FOIL con CWA y conocimiento básico

- **Sesión:**

```
?- [foil, familia_2].
Yes

?- foil(abuelo/2).

Uncovered positives: [(felipe, guillermo), (felipe, harry), (felipe, pedro),
                      (felipe, zara), (felipe, beatriz), (felipe, eugenia)]

Adding a clause ...
Specializing current clause: abuelo(A, B).

Covered negatives: [(ana, ana), (ana, andres), ...]
Covered positives: [(felipe, guillermo), (felipe, harry), (felipe, pedro),
                    (felipe, zara), (felipe, beatriz), (felipe, eugenia)]
```


Ejemplo de FOIL con CWA y conocimiento básico

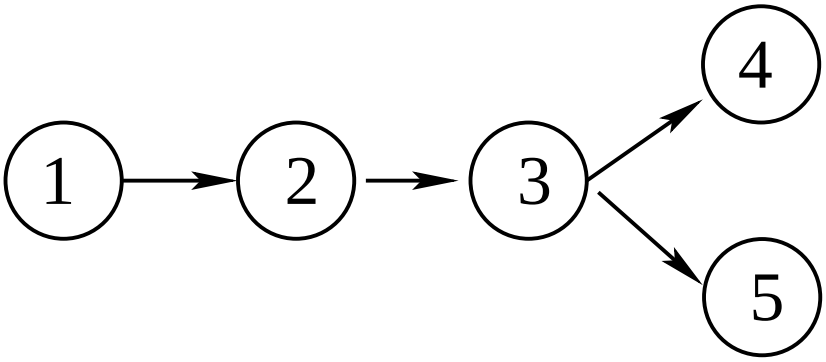
```
Ganancia: 15.510 Cláusula: abuelo(A, B):-padre(A, C)
Ganancia:  3.510 Cláusula: abuelo(A, B):-padre(C, B)
Ganancia:  3.510 Cláusula: abuelo(A, B):-madre(C, B)
Ganancia:  9.510 Cláusula: abuelo(A, B):-progenitor(A, C)
Ganancia:  3.510 Cláusula: abuelo(A, B):-progenitor(C, B)

Specializing current clause: abuelo(A, B) :- padre(A, C).

Covered negatives: [(andres, ana), (andres, andres), ...]
Covered positives: [(felipe, guillermo), (felipe, harry), (felipe, pedro),
                    (felipe, zara), (felipe, beatriz), (felipe, eugenia)]
```

Ejemplo de FOIL con CWA y conocimiento básico

```
Ganancia:  3.087 Cláusula: abuelo(A, B):-padre(A, C), padre(A, D)
Ganancia:  3.510 Cláusula: abuelo(A, B):-padre(A, C), padre(D, B)
Ganancia:  7.932 Cláusula: abuelo(A, B):-padre(A, C), padre(C, D)
Ganancia: 10.575 Cláusula: abuelo(A, B):-padre(A, C), padre(C, B)
Ganancia:  3.510 Cláusula: abuelo(A, B):-padre(A, C), madre(D, B)
Ganancia:  7.932 Cláusula: abuelo(A, B):-padre(A, C), madre(C, D)
Ganancia:  5.288 Cláusula: abuelo(A, B):-padre(A, C), madre(C, B)
Ganancia:  3.087 Cláusula: abuelo(A, B):-padre(A, C), progenitor(A, D)
Ganancia:  3.510 Cláusula: abuelo(A, B):-padre(A, C), progenitor(D, B)
Ganancia:  7.932 Cláusula: abuelo(A, B):-padre(A, C), progenitor(C, D)
Ganancia: 15.863 Cláusula: abuelo(A, B):-padre(A, C), progenitor(C, B)

Clause found: abuelo(A, B) :- padre(A, C), progenitor(C, B).

Found definition: abuelo(A, B) :- padre(A, C), progenitor(C, B).
```


Ejemplo con FOIL de relación recursiva

- **Grafo**



Ejemplo con FOIL de relación recursiva

- **Representación** `camino.pl`**:**
  - **Ejemplos:**

```
enlace(1,2).  enlace(2,3).  enlace(3,4).  enlace(3,5).
camino(1,2).  camino(1,3).  camino(1,4).  camino(1,5).
camino(2,3).  camino(2,4).  camino(2,5).
camino(3,4).  camino(3,5).
```

  - **Parámetros:**

```
foil_predicates([camino/2, enlace/2]).
foil_cwa(true).              % Usa la hipótesis del mundo cerrado
foil_use_negations(false).   % No usa información negativa
foil_det_lit_bound(0).       % No añade literales determinados
```


Ejemplo con FOIL de relación recursiva

- **Sesión**

```
?- [foil,camino].
?- foil(camino/2).
Uncovered positives: [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5)]

Adding a clause ...
Specializing current clause: camino(A,B).

Covered negatives: [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),
                    (4,4),(4,5),(5,1),(5,2),(5,3),(5,4),(5,5)]
Covered positives: [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5)]

Ganancia: -2.630 Cláusula: camino(A,B):-enlace(C,A)
Ganancia:  5.503 Cláusula: camino(A,B):-enlace(A,C)
Ganancia:  2.897 Cláusula: camino(A,B):-enlace(C,B)
Ganancia: -1.578 Cláusula: camino(A,B):-enlace(B,C)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(A,A)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(B,A)
Ganancia:  5.896 Cláusula: camino(A,B):-enlace(A,B)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(B,B)
```

Ejemplo con FOIL de relación recursiva

```
Clause found: camino(A,B) :- enlace(A,B).

Uncovered positives: [(1,3),(1,4),(1,5),(2,4),(2,5)]

Adding a clause ...
Specializing current clause: camino(A,B).

Covered negatives: [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),
                    (4,4),(4,5),(5,1),(5,2),(5,3),(5,4),(5,5)]
Covered positives: [(1,3),(1,4),(1,5),(2,4),(2,5)]

Ganancia: -2.034 Cláusula: camino(A,B):-enlace(C,A)
Ganancia:  2.925 Cláusula: camino(A,B):-enlace(A,C)
Ganancia:  1.962 Cláusula: camino(A,B):-enlace(C,B)
Ganancia: -1.017 Cláusula: camino(A,B):-enlace(B,C)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(A,A)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(B,A)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(A,B)
Ganancia:  0.000 Cláusula: camino(A,B):-enlace(B,B)
```


Ejemplo con FOIL de relación recursiva

```
Specializing current clause: camino(A,B) :- enlace(A,C).

Covered negatives: [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
Covered positives: [(1,3),(1,4),(1,5),(2,4),(2,5)]

Ganancia:  7.427 Cláusula: camino(A,B):-enlace(A,C),camino(C,B)
Ganancia: -1.673 Cláusula: camino(A,B):-enlace(A,C),enlace(D,A)
Ganancia: -2.573 Cláusula: camino(A,B):-enlace(A,C),enlace(A,D)
Ganancia:  2.427 Cláusula: camino(A,B):-enlace(A,C),enlace(D,B)
Ganancia: -1.215 Cláusula: camino(A,B):-enlace(A,C),enlace(B,D)
Ganancia:  3.539 Cláusula: camino(A,B):-enlace(A,C),enlace(C,D)
Ganancia:  4.456 Cláusula: camino(A,B):-enlace(A,C),enlace(C,B)

Clause found: camino(A,B) :- enlace(A,C), camino(C,B).

Found definition:
camino(A,B) :- enlace(A,C), camino(C,B).
camino(A,B) :- enlace(A,B).
```

Algoritmo de FOIL

```
% foil(+Positivos, +Objetivo, +Negativos, -Clausulas)
foil(Positivos, Objetivo, Negativos, Clausulas) :-
   foil(Positivos, Objetivo, Negativos, [], Clausulas).

foil(Positivos, Objetivo, Negativos, Acumulador, Clausulas) :-
   ( Positivos = [] -> Clausulas = Acumulador
   ; extiende_clausula(Negativos, Positivos, (Objetivo :- true), Clausula),
     ejemplos_no_cubiertos(Clausula, Positivos, Positivos1),
     foil(Positivos1, Objetivo, Negativos, [Clausula|Acumulador], Clausulas)).

% extiende_clausula(+Negativos, +Positivos, +Actual, -Clausula)
extiende_clausula(Neg0, Pos0, Clausula0, Clausula) :-
   ( Neg0 = [] -> Clausula = Clausula0
   ; genera_posibles_extensiones(Clausula0, L),
     informacion(Clausula0, Pos0, Neg0, Info),
     mejor_extension(L, Neg0, Pos0, Clausula0, Info, 0, Clausula0, Clausula1),
     Clausula0 \== Clausula1,
     ejemplos_cubiertos(Clausula1, Pos0, Pos1),
     ejemplos_cubiertos(Clausula1, Neg0, Neg1),
     extiende_clausula(Neg1, Pos1, Clausula1, Clausula))).
```


Ejemplo con Progol en espacios infinitos

- **Ejemplos positivos:**

```
p([]).        p([0]).       p([0,0]).     p([1,1]).     p([0,0,0]).
p([0,1,1]).   p([1,0,1]).   p([1,1,0]).   p([0,0,0,0]). p([0,0,1,1]).
p([0,1,0,1]). p([1,0,0,1]). p([0,1,1,0]). p([1,0,1,0]). p([1,1,0,0]).
p([1,1,1,1]).
```

- **Ejemplos negativos:**

```
:- p([1]).       :- p([0,1]).     :- p([1,0]).     :- p([0,0,1]).
:- p([0,1,0]).   :- p([1,0,0]).   :- p([1,1,1]).   :- p([0,0,0,1]).
:- p([0,0,1,0]). :- p([0,1,0,0]). :- p([1,0,0,0]). :- p([0,1,1,1]).
:- p([1,0,1,1]). :- p([1,1,0,1]). :- p([1,1,1,0]).
```

Ejemplo con Progol en espacios infinitos

- **Modos:**

```
:- modeh(1,p(+lista_binaria))?
:- modeb(1,+constant = #constant)?
:- modeb(1,+lista_binaria = [-binario|-lista_binaria])?
:- modeb(1,p(+lista_binaria))?
:- modeb(1,not(p(+lista_binaria)))?
```

- **Tipos:**

```
lista_binaria([]).
lista_binaria([X|Y]) :- binario(X), lista_binaria(Y).

binario(0).
binario(1).
```


Ejemplo con Progol en espacios infinitos

- **Sesión:**

```
> progol par
CProgol Version 4.4
...
[Testing for contradictions]
[No contradictions found]
[Generalising p([]).]
[Most specific clause is]

p(A) :- A=[].

[C:0,16,15,0 p(A).]
[1 explored search nodes]
f=0,p=16,n=15,h=0
[No compression]
```

Ejemplo con Progol en espacios infinitos

```
[Generalising p([0]).]
[Most specific clause is]

p(A) :- A=[B|C], B=0, C=[], p(C).

[C:0,16,15,0 p(A).]
[C:-2,15,15,0 p(A) :- A=[B|C].]
[C:-4,8,7,0 p(A) :- A=[B|C], B=0.]
[C:-6,8,8,0 p(A) :- A=[B|C], p(C).]
[C:8,8,0,0 p(A) :- A=[B|C], B=0, p(C).]
[5 explored search nodes]
f=8,p=8,n=0,h=0
[Result of search is]

p([0|A]) :- p(A).

[8 redundant clauses retracted]
```


Ejemplo con Progol en espacios infinitos

```
[Generalising p([1,1]).]
[Most specific clause is]

p(A) :- A=[B|C], B=1, C=[B|D], not(p(C)), D=[], p(D).

[C:-9,10,15,0 p(A).]
[C:-14,9,15,0 p(A) :- A=[B|C].]
[C:-9,7,8,0 p(A) :- A=[B|C], B=1.]
[C:-12,4,3,0 p(A) :- A=[B|C], B=1, C=[B|D].]
[C:-9,7,7,0 p(A) :- A=[B|C], B=1, C=[D|E].]
[C:5,7,0,0 p(A) :- A=[B|C], B=1, C=[D|E], not(p(C)).]
[C:7,7,0,0 p(A) :- A=[B|C], B=1, not(p(C)).]
[C:-20,4,6,0 p(A) :- A=[B|C], C=[B|D].]
[C:-23,7,14,0 p(A) :- A=[B|C], C=[D|E].]
[C:-7,7,7,0 p(A) :- A=[B|C], not(p(C)).]
```

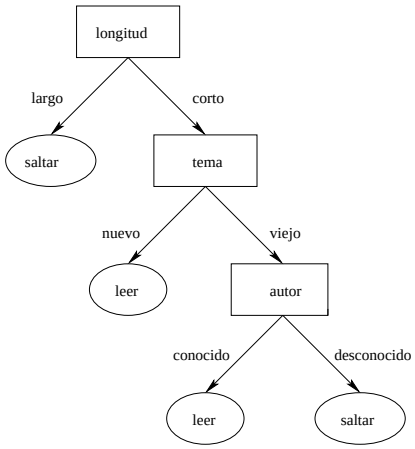
Ejemplo con Progol en espacios infinitos

```
[C:-12,4,3,0 p(A) :- A=[B|C], C=[B|D], not(p(C)).]
[C:0,4,0,0 p(A) :- A=[B|C], C=[B|D], p(D).]
[C:-9,7,7,0 p(A) :- A=[B|C], C=[D|E], not(p(C)).]
[C:-32,4,8,0 p(A) :- A=[B|C], C=[D|E], p(E).]
[14 explored search nodes]
f=7,p=7,n=0,h=0
[Result of search is]

p([1|A]) :- not(p(A)).

[7 redundant clauses retracted]
p([]).
p([0|A]) :- p(A).
p([1|A]) :- not(p(A)).
[Total number of clauses = 3]

[Time taken 0.030s]
```


Árboles de decisión en Progol

Ejemplo	Acción	Autor	Tema	Longitud	Sitio
e1	saltar	conocido	nuevo	largo	casa
e2	leer	desconocido	nuevo	corto	trabajo
e3	saltar	desconocido	viejo	largo	trabajo
e4	saltar	conocido	viejo	largo	casa
e5	leer	conocido	nuevo	corto	casa
e6	saltar	conocido	viejo	largo	trabajo
e7	saltar	desconocido	viejo	corto	trabajo
e8	leer	desconocido	nuevo	corto	trabajo
e9	saltar	conocido	viejo	largo	casa
e10	saltar	conocido	nuevo	largo	trabajo
e11	saltar	desconocido	viejo	corto	casa
e12	saltar	conocido	nuevo	largo	trabajo
e13	leer	conocido	viejo	corto	casa
e14	leer	conocido	nuevo	corto	trabajo
e15	leer	conocido	nuevo	corto	casa
e16	leer	conocido	viejo	corto	trabajo
e17	leer	conocido	nuevo	corto	casa
e18	leer	desconocido	nuevo	corto	trabajo

Árboles de decisión en Progol



Árboles de decisión en Progol

- **Representación** `softbot.pl`**:**
  - **Parámetros:**

```
:- set(r,1000)?
:- set(posonly)?
```

  - **Modos:**

```
:- modeh(1,accion(+ejemplo,#t_accion))?
:- modeb(1,autor(+ejemplo,#t_autor))?
:- modeb(1,tema(+ejemplo,#t_tema))?
:- modeb(1,longitud(+ejemplo,#t_longitud))?
:- modeb(1,sitio(+ejemplo,#t_sitio))?
```

Árboles de decisión en Progol

- **Tipos:**

```
ejemplo(e1).       ejemplo(e2).          ...   ejemplo(e18).
t_accion(saltar).  t_accion(leer).
t_autor(conocido). t_autor(desconocido).
t_tema(nuevo).     t_tema(viejo).
t_longitud(largo). t_longitud(corto).
t_sitio(casa).     t_sitio(trabajo).
```

- **Determinación:**

```
:- determination(accion/2, autor/2)?
:- determination(accion/2, tema/2)?
:- determination(accion/2, longitud/2)?
:- determination(accion/2, sitio/2)?
```


Árboles de decisión en Progol

- **Conocimiento base:**

```
autor(e1,conocido).   autor(e2,desconocido).  ... autor(e18,desconocido).
tema(e1,nuevo).       tema(e2,nuevo).         ... tema(e18, nuevo).
longitud(e1,largo).   longitud(e2,corto).     ... longitud(e18,corto).
sitio(e1,casa).       sitio(e2,trabajo).      ... sitio(e18, trabajo).
```

- **Ejemplos positivos:**

```
accion(e1,saltar).    accion(e2,leer).        ... accion(e18,leer).
```

- **Restricciones:**

```
:- hypothesis(Cabeza,Cuerpo,_),
   accion(A,C),
   Cuerpo,
   Cabeza = accion(A,B),
   B \= C.
```

Árboles de decisión en Progol

- **Sesión:**

```
> progol softbot
CProgol Version 4.4
...
[Testing for contradictions]
[No contradictions found]
[Generalising accion(e1,saltar).]
[Most specific clause is]

accion(A,saltar) :-
  autor(A,conocido), tema(A,nuevo), longitud(A,largo), sitio(A,casa).
```


Árboles de decisión en Progol

```
[Learning accion/2 from positive examples]
[C:-39932,18,10000,0 accion(A,saltar).]
[C:-39936,18,10000,0 accion(A,saltar) :- autor(A,conocido).]
[C:-39936,18,10000,0 accion(A,saltar) :- tema(A,nuevo).]
[C:34,28,13,0 accion(A,saltar) :- longitud(A,largo).]
[C:13,12,7,0 accion(A,saltar) :- longitud(A,largo), sitio(A,casa).]
[C:-39936,18,10000,0 accion(A,saltar) :- sitio(A,casa).]
[C:-39940,18,10000,0 accion(A,saltar) :- autor(A,conocido), tema(A,nuevo).]
[C:31,24,11,0 accion(A,saltar) :- autor(A,conocido), longitud(A,largo).]
[C:7,12,7,0 accion(A,saltar) :- autor(A,conocido), longitud(A,largo), sitio(A,casa).]
[C:-39940,18,10000,0 accion(A,saltar) :- autor(A,conocido), sitio(A,casa).]
[C:25,12,5,0 accion(A,saltar) :- tema(A,nuevo), longitud(A,largo).]
[C:-39940,18,10000,0 accion(A,saltar) :- tema(A,nuevo), sitio(A,casa).]
[C:19,12,5,0 accion(A,saltar) :- autor(A,conocido), tema(A,nuevo), longitud(A,largo).]
[C:-39944,18,10000,0 accion(A,saltar) :- autor(A,conocido), tema(A,nuevo), sitio(A,casa)
[14 explored search nodes]
f=34,p=28,n=13,h=0
[Result of search is]

accion(A,saltar) :- longitud(A,largo).
```

Árboles de decisión en Progol

```
[7 redundant clauses retracted]
[Generalising accion(e2,leer).]
[Most specific clause is]

accion(A,leer) :-
  autor(A,desconocido), tema(A,nuevo), longitud(A,corto), sitio(A,trabajo).
```


Árboles de decisión en Progol

```
[Learning accion/2 from positive examples]
[C:-39932,18,10000,0 accion(A,leer).]
[C:-39936,18,10000,0 accion(A,leer) :- autor(A,desconocido).]
[C:-39936,18,10000,0 accion(A,leer) :- tema(A,nuevo).]
[C:-39936,18,10000,0 accion(A,leer) :- longitud(A,corto).]
[C:-39936,18,10000,0 accion(A,leer) :- sitio(A,trabajo).]
[C:-39940,18,10000,0 accion(A,leer) :- longitud(A,corto), sitio(A,trabajo).]
[C:34,28,12,0 accion(A,leer) :- tema(A,nuevo), longitud(A,corto).]
[C:19,16,8,0 accion(A,leer) :- tema(A,nuevo), longitud(A,corto), sitio(A,trabajo).]
[C:-39940,18,10000,0 accion(A,leer) :- tema(A,nuevo), sitio(A,trabajo).]
[C:13,12,7,0 accion(A,leer) :- autor(A,desconocido), tema(A,nuevo).]
[C:7,12,7,0 accion(A,leer) :- autor(A,desconocido), tema(A,nuevo), longitud(A,corto).]
[C:7,12,7,0 accion(A,leer) :- autor(A,desconocido), tema(A,nuevo), sitio(A,trabajo).]
[C:1,12,7,0 accion(A,leer) :- autor(A,desconocido), tema(A,nuevo), longitud(A,corto), si
[C:-39940,18,10000,0 accion(A,leer) :- autor(A,desconocido), longitud(A,corto).]
[C:-39940,18,10000,0 accion(A,leer) :- autor(A,desconocido), sitio(A,trabajo).]
[C:-39944,18,10000,0 accion(A,leer) :- autor(A,desconocido), longitud(A,corto), sitio(A,
[16 explored search nodes]
f=34,p=28,n=12,h=0
[Result of search is]
accion(A,leer) :- tema(A,nuevo), longitud(A,corto).
```

Árboles de decisión en Progol

```
[7 redundant clauses retracted]
[Generalising accion(e7,saltar).]
[Most specific clause is]

accion(A,saltar) :-
   autor(A,desconocido), tema(A,viejo), longitud(A,corto), sitio(A,trabajo).
```


Árboles de decisión en Progol

```
[Learning accion/2 from positive examples]
[C:-39932,18,10000,0 accion(A,saltar).]
[C:-39936,18,10000,0 accion(A,saltar) :- autor(A,desconocido).]
[C:-39936,18,10000,0 accion(A,saltar) :- tema(A,viejo).]
[C:-39936,18,10000,0 accion(A,saltar) :- longitud(A,corto).]
[C:-39936,18,10000,0 accion(A,saltar) :- sitio(A,trabajo).]
[C:-39940,18,10000,0 accion(A,saltar) :- longitud(A,corto), sitio(A,trabajo).]
[C:10,8,4,0 accion(A,saltar) :- autor(A,desconocido), tema(A,viejo).]
[C:1,8,4,0 accion(A,saltar) :- autor(A,desconocido), tema(A,viejo), longitud(A,corto).]
[C:-39940,18,10000,0 accion(A,saltar) :- autor(A,desconocido), longitud(A,corto).]
[C:-39940,18,10000,0 accion(A,saltar) :- autor(A,desconocido), sitio(A,trabajo).]
[C:-39940,18,10000,0 accion(A,saltar) :- tema(A,viejo), longitud(A,corto).]
[C:-39940,18,10000,0 accion(A,saltar) :- tema(A,viejo), sitio(A,trabajo).]
[C:-39944,18,10000,0 accion(A,saltar) :- tema(A,viejo), longitud(A,corto), sitio(A,traba
[C:-39944,18,10000,0 accion(A,saltar) :- autor(A,desconocido), longitud(A,corto), sitio(
[14 explored search nodes]
f=10,p=8,n=4,h=0
[Result of search is]
accion(A,saltar) :- autor(A,desconocido), tema(A,viejo).
```

Árboles de decisión en Progol

```
[2 redundant clauses retracted]
[Generalising accion(e13,leer).]
[Most specific clause is]

accion(A,leer) :-
   autor(A,conocido), tema(A,viejo), longitud(A,corto), sitio(A,casa).
```


Árboles de decisión en Progol

```
[Learning accion/2 from positive examples]
[C:-39932,18,10000,0 accion(A,leer).]
[C:-39936,18,10000,0 accion(A,leer) :- autor(A,conocido).]
[C:-39936,18,10000,0 accion(A,leer) :- tema(A,viejo).]
[C:-39936,18,10000,0 accion(A,leer) :- longitud(A,corto).]
[C:-39936,18,10000,0 accion(A,leer) :- sitio(A,casa).]
[C:-39940,18,10000,0 accion(A,leer) :- tema(A,viejo), longitud(A,corto).]
[C:-39940,18,10000,0 accion(A,leer) :- tema(A,viejo), sitio(A,casa).]
[C:-39940,18,10000,0 accion(A,leer) :- longitud(A,corto), sitio(A,casa).]
[C:-39940,18,10000,0 accion(A,leer) :- autor(A,conocido), tema(A,viejo).]
[C:10,8,4,0 accion(A,leer) :- autor(A,conocido), longitud(A,corto).]
[C:-39940,18,10000,0 accion(A,leer) :- autor(A,conocido), sitio(A,casa).]
[C:1,8,4,0 accion(A,leer) :- autor(A,conocido), tema(A,viejo), longitud(A,corto).]
[C:-39944,18,10000,0 accion(A,leer) :- autor(A,conocido), tema(A,viejo), sitio(A,casa).]
[C:-39944,18,10000,0 accion(A,leer) :- tema(A,viejo), longitud(A,corto), sitio(A,casa).]
[14 explored search nodes]
f=10,p=8,n=4,h=0
[Result of search is]
accion(A,leer) :- autor(A,conocido), longitud(A,corto).
```

Árboles de decisión en Progol

```
[2 redundant clauses retracted]
accion(A,saltar) :- longitud(A,largo).
accion(A,leer) :- tema(A,nuevo), longitud(A,corto).
accion(A,saltar) :- autor(A,desconocido), tema(A,viejo).
accion(A,leer) :- autor(A,conocido), longitud(A,corto).
[Total number of clauses = 4]

[Time taken 0.090s]
```


Algoritmo de Progol

- **Algoritmo de Progol:**
  1. **Empezar con la teoría vacía: $T = \emptyset$.**
  2. **Seleccionar un ejemplo para generalizarlo: E**
  3. **Construir la cláusula más específica que implica el ejemplo seleccionado y cumple las restricciones impuestas: $T \cup \{C_1\} \models E$.**
  4. **Buscar la mejor clásula que generaliza la anterior y añadirla a la teoría: $T := T \cup \{C_2\}$**
  5. **Borrar los ejemplos positivos cubiertos por la teoría.**
  6. **Si quedan ejemplos positivos, volver a 1; en caso contrario, devolver la teoría construida.**

Aplicaciones con PLI

- **Procesamiento de lenguaje natural:**
  - **aprender reglas gramaticales.**
- **Diseño asistido por ordenador:**
  - **aprender relaciones entre objetos.**
- **Diseño de medicamentos:**
  - **predecir actividad a apartir de propiedades moleculares.**
- **Musicología:**
  - **aprender el estilo de un músico.**

Aplicaciones con PLI

- **Predicción de la estructura secundaria de las proteinas.**
  - **Dada la estructura primaria de una proteina (secuencia de aminoácidos),**
    - **Encontrar la estructura secundaria.**
    - **Predecir si los residuos individuales forman una hélice levógira.**
  - **Ejemplos: 12 proteinas no homólogas (1612 residuos).**
  - **Conocimiento base: Propiedades físicas y químicas de los residuos individuales y su posición relativa dentro de la proteina.**
  - **Sistema: GOLEM.**
  - **Resultados:**
    - **21 cláusula producidas, cada una de unos 15 literales.**
    - **Su precisión sobre un test independiente fue del 82%, mientras que la precisión del mejor método convencional fue del 73%.**

Bibliografía

- **Flach, P. *Simply Logical (Intelligent Reasoning by Example)* (John Wiley, 1994).**
  - **Cap. 9: "Inductive reasoning".**
- **Fürnkranz, J. *Inductive Logic Programming* (`http://www.ai.univie.ac.at/~juffi/Ilp-Vu/ilp-vu-program.html`)**
- **Markov,Z. *Machine larning course* (Faculty of Mathematics and Informatics, Univ. of Sofia, 1998)**
- **Gómez, A.J. *Inducción de conocimiento con incertidumbre en bases de datos relacionales borrosas* (Tesis doctoral, Univ. Politécnica de Madrid, 1998).**
- **Mitchell, T.M. *Machine learning* (McGraw–Hill, 1997).**
  - **Cap. 3: "Decision tree learning".**
  - **Cap. 11: "Learning sets of rules".**

Bibliografía

- **Muggleton, S. y Firth, J. *CProgol4.4: a tutorial introduction* (En "Inductive Logic Programming and Knowledge Discovery in Databases". Springer–Verlag, 2000).**
- **Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998).**
  - **Cap. 11: "Learning".**
- **Russell, S. y Norvig, P. *Inteligencia artificial (Un enfoque moderno)* (Prentice–Hall Hispanoamericana, 1996).**
  - **Cap. 18: "Aprendizaje a partir de la observación".**
  - **Cap. 21: "El conocimiento en el aprendizaje".**
- **The Online School on Inductive Logic Programing and Knowledge Discovery in Databases (`http://www-ai.ijs.si/SasoDzeroski/ILP2/ilpkdd`).**
- **MLnet Online Information Service (`http://www.mlnet.org`).**

Bibliografía

[1] J.A. Alonso y J. Borrego *Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos).* (Ed. Kronos, 2002).
`http://www.cs.us.es/~jalonso/libros/da1-02.pdf`

[2] M. Ben-Ari *Mathematical Logic for Computer Science (2nd ed.)* (Springer, 2001)

[3] P. Blackburn, J. Bos y K. Striegnitz *Learn Prolog Now!.*
`http://www.coli.uni-sb.de/~kris/learn-prolog-now`

[4] I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.).* (Addison–Wesley, 2001).

[5] A.M. Cheadle et als. *ECLiPSe: An Introduction* (Imperial College London, Technical Report IC-Parc-03-1, 2003).

[6] W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition).* (Springer Verlag, 1994).

[7] M.A. Covington *Efficient Prolog: A Practical Guide.*
`http://www.ai.uga.edu/ftplib/ai-reports/ai198908.pdf`

[8] M.A. Covington, D. Nute y A. Vellino *Prolog Programming in Depth.* (Prentice Hall, 1997).

[9] P. Flach *Simply Logical (Intelligent Reasoning by Example).* (John Wiley, 1994).

[10] P. Lucas y L.v.d. Gaag *Principles of Expert Systems* (Addison-Wesley, 1991).

[11] K. Marriott y P.J. Stuckey *Programming with Constraints. An Introduction* (The MIT Press, 1998).

[12] D. Merritt *Building Expert Systems in Prolog* (Springer Verlag, 1989).

[13] T.M. Mitchell *Machine learning* (McGraw-Hill, 1997).

[14] A. Nerode y R.A. Shore *Logic for Applications* (Springer, 1997)

[15] U. Nilsson y J. Maluszynski *Logic, Programming and Prolog (2nd ed.).* `http://www.ida.liu.se/~ulfni/lpp`

[16] R.A. O'Keefe *The Cratf of Prolog.* (The MIT Press, 1990).

[17] D. Poole, A. Mackworth, A. y R. Goebel *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)

[18] S. Russell y P. Norvig *Inteligencia artificial (Un enfoque moderno)* (Prentice–Hall Hispanoamericana, 1996).

[19] Y. Shoham *Artificial Intelligence Techniques in Prolog* (Morgan Kaufmann, 1994)

[20] L. Sterling y E. Shapiro *The Art of Prolog.* (MIT Press, 1994).

[21] T. Van Le *Techniques of Prolog Programming.* (John Wiley, 1993).