Temas de "Programación declarativa" (2003–04)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 30 de Junio de 2005

Esta obra está bajo una licencia Reconocimiento–NoComercial–CompartirIgual 2.5 Spain de Creative Commons.

Se permite:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor.

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor-

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite `http://creativecommons.org/licenses/by-nc-sa/2.5/es/` o envie una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Índice general

Capítulo 1

Introducción a Prolog

Programación declarativa (2004–05)

Tema 1: Introducción a Prolog

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

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
- Pensar declarativamente

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, DLV, Smodels.
- Ventajas;
  - Imperativo: Programas rápidos y especializados.
  - Declarativo: Programas generales, cortos y legibles.

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.

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 e animal es una cebra.

Deducción Prolog en lógica proposicional

- Programa:

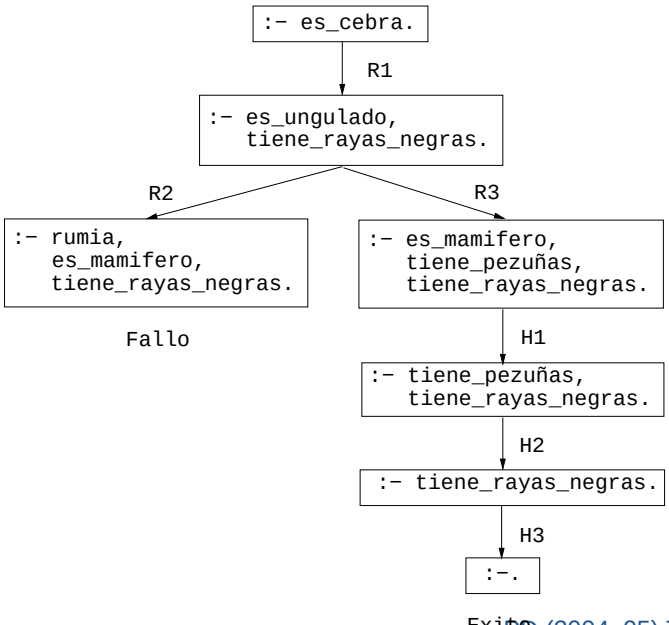
```
es_cebra    :- es_ungulado, tiene_rayas_negras. % R1
es_ungulado :- rumia, es_mamífero.              % R2
es_ungulado :- es_mamífero, tiene_pezuñas.      % R3
es_mamífero.                                    % H1
tiene_pezuñas.                                  % H2
tiene_rayas_negras.            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

- Sesión:

```
> pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.3.14)
Copyright (c) 1990-2003 University of Amsterdam.
?- [animales].
Yes
?- es_cebra.
Yes
```

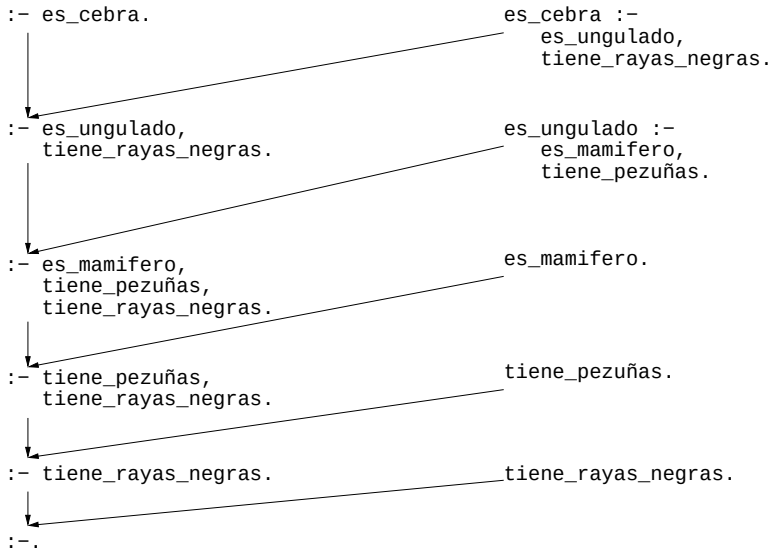
Deducción Prolog en lógica proposicional

- Árbol de deducción:



Deducción Prolog en lógica proposicional

- Demostración por resolución SLD:



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
```

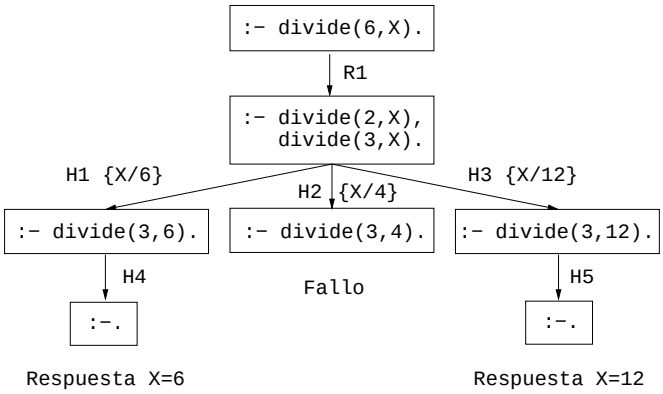

Deducción Prolog en lógica relacional

- 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.

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

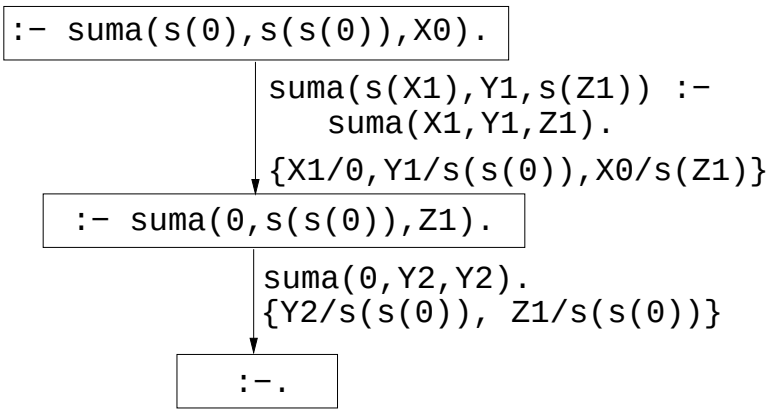
```
suma(0,Y,Y).                          % R1
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).     % R2
```

- 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)))
```

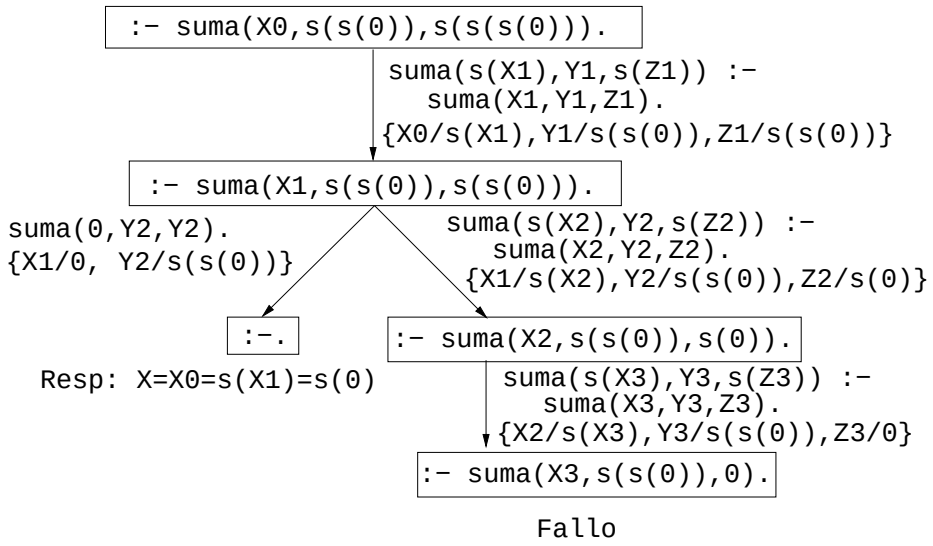

Deducción Prolog en lógica funcional

- 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:



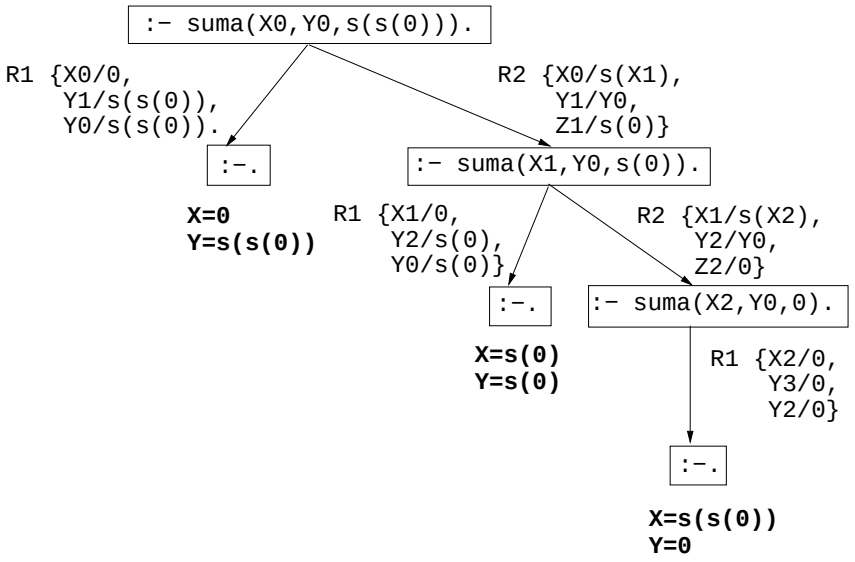
Deducción Prolog en lógica funcional

- 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:



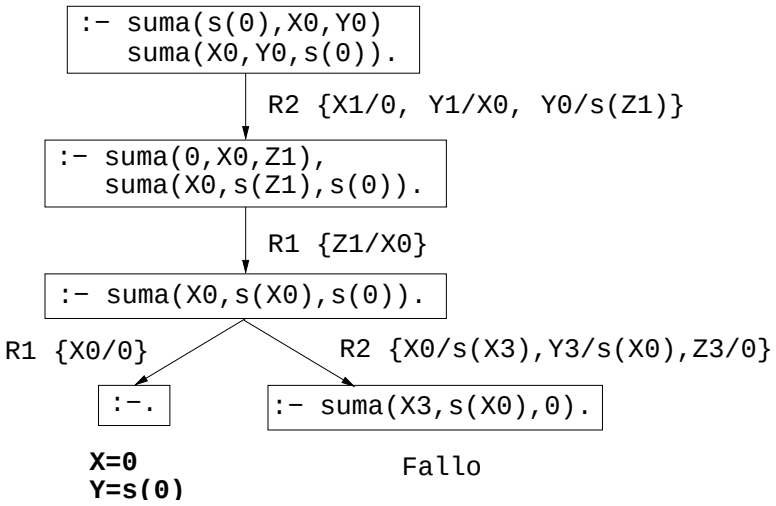
Deducción Prolog en lógica funcional

- 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

- J.A. Alonso y J. Borrego
Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)
(Ed. Kronos, 2002)
  - Cap. 2: Introducción a la programación lógica con Prolog.
- I. Bratko *Prolog Programming for Artificial Intelligence (2nd ed.)*
(Addison–Wesley, 1990)
  - Cap. 1: “An overview of Prolog”
  - Cap. 2: “Syntax and meaning of Prolog programs”
- W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)*
(Springer Verlag, 1994)
  - Cap. 1: “Tutorial introduction”
  - Cap. 2: “A closer look”

Capítulo 2

Listas, operadores y aritmética

Programación declarativa (2004–05)

Tema 2: Listas, operadores y aritmética

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Listas

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

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

Listas

- 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]
```


Listas

- Concatenación de listas (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`; es decir,
    1. Si `A` es la lista vacía, entonces la concatenación de `A` y `B` es `B`.
    2. Si `A` es una lista cuyo primer elemento es `X` y cuyo resto es `D`, entonces la concatenación de `A` y `B` es una lista cuyo primer elemento es `X` y cuyo resto es la concatenación de `D` y `B`.

    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).
```

Listas

- Concatenación de listas (`append`):
  - *Nota:* Analogía entre la definición de `conc` y la de `suma`,
  - *Consulta:* ¿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]
```

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

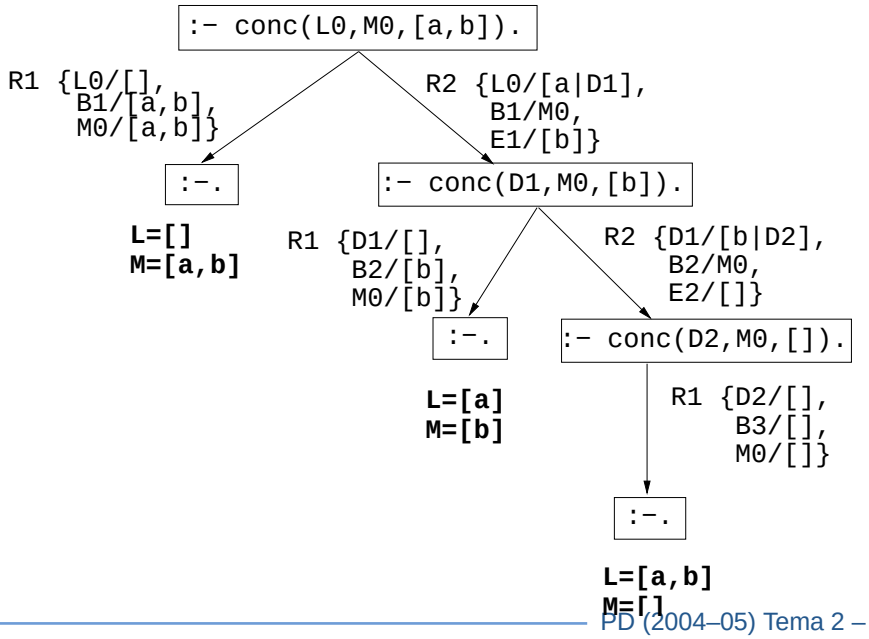
```
?- conc([a,b],L,[a,b,c,d]).
L = [c, d]
```

  - *Consulta:* ¿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
```


Listas

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



Listas

- 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).
```

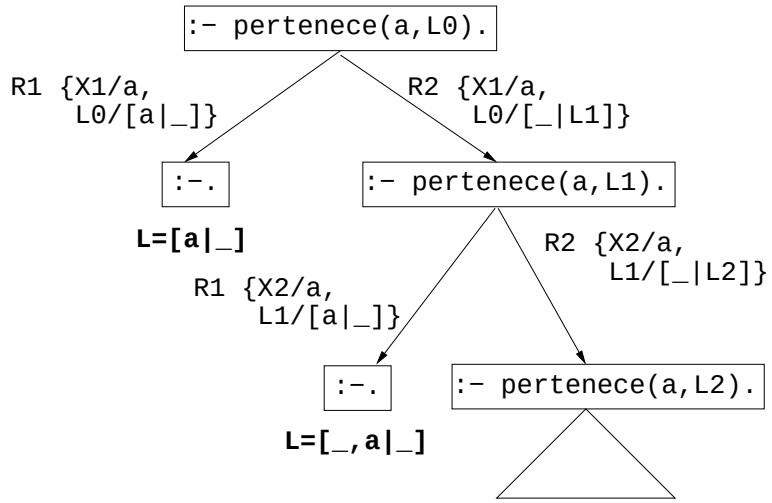

Listas

- La relación de pertenencia (`member`):
  - *Consultas:*

```
?- 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
```

Listas

- 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).
```

Operadores

- 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
```


Operadores

- 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

- Ejemplos de asociatividad:

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

Operadores

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


Operadores

- Definición de operadores:
  - Definición (`ejemplo_operadores.pl`)

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

juan y ana estudian lógica.
```

  - Consultas

```
?- [ejemplo_operadores].
Yes

?- Quienes estudian lógica.
Quienes = juan y ana

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

Aritmética

- 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
```


Aritmética

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

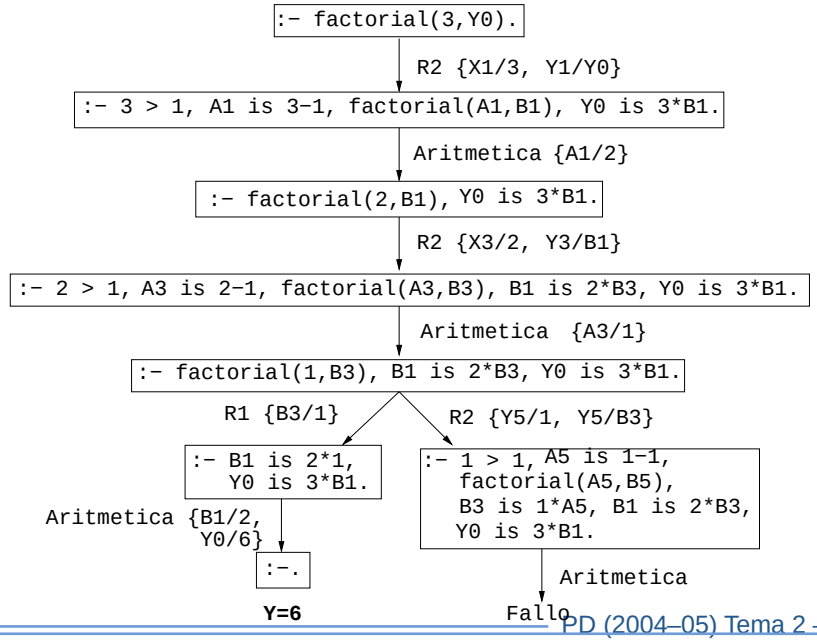
```
?- factorial(3,Y).
Y = 6 ;
No
```

  - Definición:

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

Aritmética

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



Bibliografía

- Alonso, J.A. y Borrego, J. *Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)* (Ed. Kronos, 2002)
- Bratko, I. *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
- Clocksin, W.F. y Mellish, C.S. *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
- Covington, M.A.; Nute, D. y Vellino, A. *Prolog Programming in Depth* (Prentice Hall, 1997)
- Sterling, L. y Shapiro, E. *L'art de Prolog* (Masson, 1990)
- Van Le, T. *Techniques of Prolog Programming* (John Wiley, 1993)

Capítulo 3

Estructuras

Programación declarativa (2004–05)

Tema 3: Estructuras

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Objetos estructurados

- Segmentos verticales y horizontales:
  - Representación:
    ```
    punto(X,Y)
    seg(P1,P2)
    ```
  - `horizontal(S)` se verifica si el segmento `S` es horizontal.
  - `vertical(S)` se verifica si el segmento `S` es vertical. Por ejemplo,
    ```
    vertical(seg(punto(1,2),punto(1,3)))   =>  Sí
    vertical(seg(punto(1,2),punto(4,2)))   =>  No
    horizontal(seg(punto(1,2),punto(1,3))) =>  No
    horizontal(seg(punto(1,2),punto(4,2))) =>  Sí
    ```
  - Programa: `ver_hor.pl`
    ```
    horizontal(seg(punto(X,Y),punto(X1,Y))).
    vertical(seg(punto(X,Y),punto(X,Y1))).
    ```

Objetos estructurados

- Segmentos verticales y horizontales (cont.):
  - ¿Es vertical el segmento que une los puntos `(1,1)` y `(1,2)`?
    ```
    ?- vertical(seg(punto(1,1),punto(1,2))).
    Yes
    ```
  - ¿Es vertical el segmento que une los puntos `(1,1)` y `(2,2)`?
    ```
    ?- vertical(seg(punto(1,1),punto(2,2))).
    No
    ```
  - ¿Hay algún `Y` tal que el segmento que une los puntos `(1,1)` y `(2,Y)` sea vertical?
    ```
    ?- vertical(seg(punto(1,1),punto(2,Y))).
    No
    ```


Objetos estructurados

- Segmentos verticales y horizontales (cont.):
  - ¿Hay algún `X` tal que el segmento que une los puntos `(1,2)` y `(X,3)` sea vertical?
    ```
    ?- vertical(seg(punto(1,2),punto(X,3))).
    X = 1 ;
    No
    ```
  - ¿Hay algún `Y` tal que el segmento que une los puntos `(1,1)` y `(2,Y)` sea horizontal?
    ```
    ?- horizontal(seg(punto(1,1),punto(2,Y))).
    Y = 1 ;
    No
    ```

Objetos estructurados

- Segmentos verticales y horizontales (cont.):
  - ¿Para qué puntos el segmento que comienza en `(2,3)` es vertical?

```
?- vertical(seg(punto(2,3),P)).
P = punto(2, _G459) ;
No
```

  - ¿Hay algún segmento que sea horizontal y vertical?

```
?- vertical(S),horizontal(S).
S = seg(punto(_G444, _G445),
        punto(_G444, _G445)) ;
No
?- vertical(_),horizontal(_).
Yes
```


Recuperación de la información

- Descripción de la familia 1 :
  - el padre es Tomás García Pérez, nacido el 7 de Mayo de 1950, trabaja de profesor y gana 75 euros diarios
  - la madre es Ana López Ruiz, nacida el 10 de marzo de 1952, trabaja de médica y gana 100 euros diarios
  - el hijo es Juan García López, nacido el 5 de Enero de 1970, estudiante
  - la hija es María García López, nacida el 12 de Abril de 1972, estudiante

Recuperación de la información

- Representación de la familia 1 :

```
familia(persona([tomas,garcia,perez],
                fecha(7,mayo,1950),
                trabajo(profesor,75)),
        persona([ana,lopez,ruiz],
                fecha(10,marzo,1952),
                trabajo(medica,100)),
        [ persona([juan,garcia,lopez],
                  fecha(5,enero,1970),
                  estudiante),
          persona([maria,garcia,lopez],
                  fecha(12,abril,1972),
                  estudiante) ]).
```


Recuperación de la información

- Descripción de la familia 2:
  - el padre es José Pérez Ruiz, nacido el 6 de Marzo de 1953, trabaja de pintor y gana 150 euros/día
  - la madre es Luisa Gálvez Pérez, nacida el 12 de Mayo de 1954, es médica y gana 100 euros/día
  - un hijo es Juan Luis Pérez Pérez, nacido el 5 de Febrero de 1980, estudiante
  - una hija es María José Pérez Pérez, nacida el 12 de Junio de 1982, estudiante
  - otro hijo es José María Pérez Pérez, nacido el 12 de Julio de 1984, estudiante

Recuperación de la información

- Representación de la familia 2:

```
familia(persona([jose,perez,ruiz],
                fecha(6,marzo,1953),
                trabajo(pintor,150)),
        persona([luisa,galvez,perez],
                fecha(12,mayo,1954),
                trabajo(medica,100)),
        [ persona([juan_luis,perez,perez],
                  fecha(5,febrero,1980),
                  estudiante),
          persona([maria_jose,perez,perez],
                  fecha(12,junio,1982),
                  estudiante),
          persona([jose_maria,perez,perez],
                  fecha(12,julio,1984),
                  estudiante) ]).
```


Recuperación de la información

- Consultas:
  - ¿Existe alguna familia sin hijos?

```
?- familia(_,_,[]).
No
```

  - ¿Existe alguna familia con tres hijos?

```
?- familia(_,_,[_,_,_]).
Yes
```

  - ¿Existe alguna familia con cuatro hijos?

```
?- familia(_,_,[_,_,_,_]).
No
```

  - Buscar los nombres de los padres de familia con tres hijos

```
?- familia(persona(NP,_,_),_,[_,_,_]).
NP = [jose, perez, ruiz] ;
No
```

Recuperación de la información

- Hombres casados:
  - `casado(X)` se verifica si `X` es un hombre casado

```
casado(X) :- familia(X,_,_).
```

  - Consulta:

```
?- casado(X).
X = persona([tomas, garcia, perez],
            fecha(7, mayo, 1950),
            trabajo(profesor, 75)) ;
X = persona([jose, perez, ruiz],
            fecha(6, marzo, 1953),
            trabajo(pintor, 150)) ;
No
```


Recuperación de la información

- Mujeres casadas
  - `casada(X)` se verifica si `X` es una mujer casada

```
casada(X) :- familia(_,X,_).
```

  - Consulta:

```
?- casada(X).
X = persona([ana, lopez, ruiz],
            fecha(10, marzo, 1952),
            trabajo(medica, 100)) ;
X = persona([luisa, galvez, perez],
            fecha(12, mayo, 1954),
            trabajo(medica, 100)) ;
No
```

Recuperación de la información

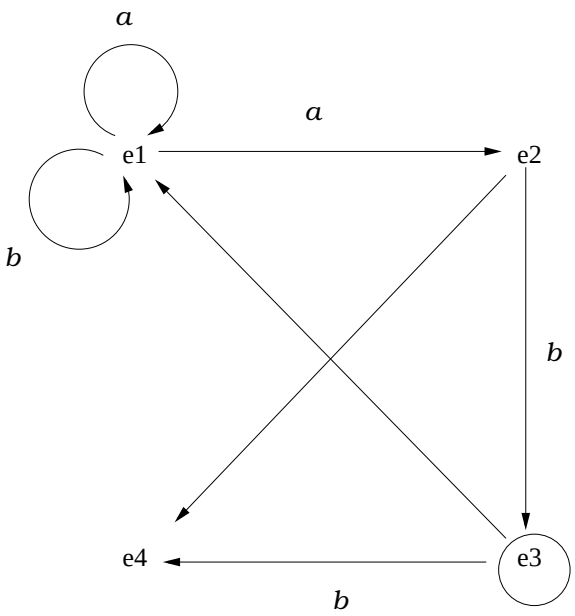
- Recuperación de la información:
  - Nombres de las mujeres casadas que trabajan

```
?- casada(persona([N,_,_],_,trabajo(_,_))).
N = ana ;
N = luisa ;
No
```

- Conceptos:
  - Uso de objetos estructurados.
  - Prolog como lenguaje de consulta de bases de datos.
  - Bases de datos como conjuntos de hechos.
  - Selectores.
  - Abstracción de datos.

Simulación de un autómata

- Autómata no determinista (con estado final e3):



Simulación de un autómata

- Representación del 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).
    ```


Simulación de un autómata

- Representación del autómata (`automata.pl`) (cont.):
  - `acepta(E,L)` se verifica si el autómata, a partir del estado `E` acepta la lista `L`.
    - Ejemplo:
      ```
      acepta(e1,[a,a,a,b]) => Sí
      acepta(e2,[a,a,a,b]) => No
      ```
    - Definición:
      ```
      acepta(E,[]) :-
         final(E).
      acepta(E,[X|L]) :-
         trans(E,X,E1),
         acepta(E1,L).
      acepta(E,L) :-
         nulo(E,E1),
         acepta(E1,L).
      ```

Simulación de un autómata

- Consultas:
  - 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
```


Problema del mono

- Descripción: Un mono se encuentra en la puerta de una habitación. En el centro de la habitación hay un plátano colgado del techo. El mono está hambriento y desea coger el plátano, pero no lo alcanza desde el suelo. En la ventana de la habitación hay una silla que el mono puede usar. El mono puede realizar las siguientes acciones: pasear de un lugar a otro de la habitación, empujar la silla de un lugar a otro de la habitación (si está en el mismo lugar que la silla), subirse en la silla (si está en el mismo lugar que la silla) y coger el plátano (si está encima de la silla en el centro de la habitación).
- Representación: `estado(PM,AM,PS,MM)`
  - `PM` posición del mono (puerta, centro o ventana)
  - `AM` apoyo del mono (suelo o silla)
  - `PS` posición de la silla (puerta, centro o ventana)
  - `MM` mano del mono (con o sin) plátano

Problema del mono

- Acciones:
  - `movimiento(E1,A,E2)` se verifica si `E2` es el estado que resulta de realizar la acción `A` en el estado `E1`

```
movimiento(estado(centro,silla,centro,sin),
           coger,
           estado(centro,silla,centro,con)).
movimiento(estado(X,suelo,X,U),
           subir,
           estado(X,silla,X,U)).
movimiento(estado(X1,suelo,X1,U),
           empujar(X1,X2),
           estado(X2,suelo,X2,U)).
movimiento(estado(X,suelo,Z,U),
           pasear(X,Z),
           estado(Z,suelo,Z,U)).
```


Problema del mono

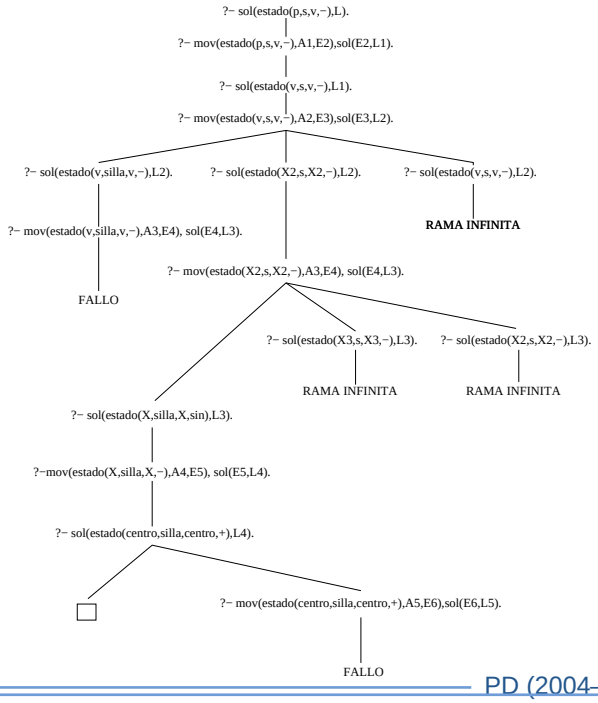
- Solución:
  - `solución(E,S)` se verifica si `S` es una sucesión de acciones que aplicadas al estado `E` permiten al mono coger el plátano.
  - Ejemplo:

```
?- solucion(estado(puerta,suelo,ventana,sin),L).
L = [pasear(puerta, ventana),
     empujar(ventana, centro),
     subir,
     coger]
```

  - Definición :

```
solucion(estado(_,_,_,con),[]).
solucion(E1,[A|L]) :-
   movimiento(E1,A,E2),
   solucion(E2,L).
```

Problema del mono



Bibliografía

- J.A. Alonso y J. Borrego
  Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)
  (Ed. Kronos, 2002)
  - Cap. 2: Introducción a la programación lógica con Prolog.
- I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)*
  (Addison–Wesley, 2001)
  - Cap. 2: “Syntax and meaning of Prolog programs”
  - Cap. 4: “Using Structures: Example Programs”
- T. Van Le *Techniques of Prolog Programming* (John Wiley, 1993)
  - Cap. 2: “Declarative Prolog programming”.
- W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)*
  (Springer Verlag, 1994)
  - Cap. 3: “Using Data Structures”

Capítulo 4

Retroceso, corte y negación

Programación declarativa (2004–05)

Tema 4: Retroceso, corte y negación

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Control mediante corte

- 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`.
  - Definición:

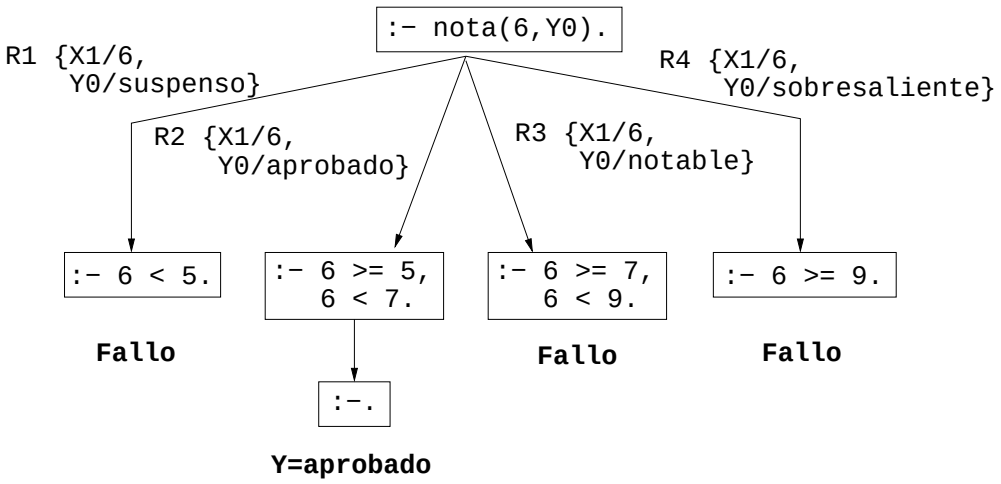
```
nota(X,suspenso)      :- X < 5.
nota(X,aprobado)      :- X >= 5, X < 7.
nota(X,notable)       :- X >= 7, X < 9.
nota(X,sobresaliente) :- X >= 9.
```

  - Ejemplo: ¿cuál es la calificación correspondiente a un 6?:

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

Control mediante corte

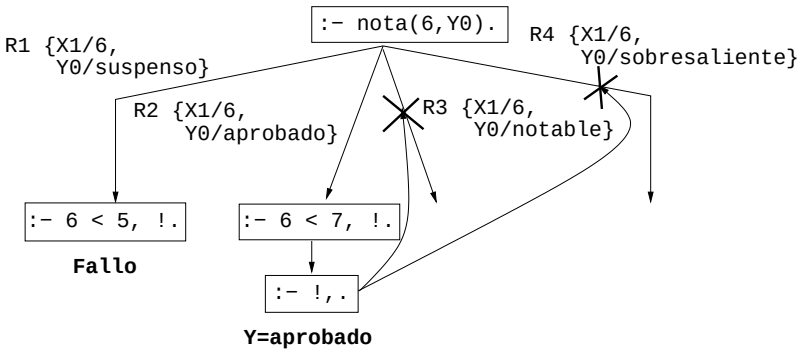
- Árbol de deducción de `?- nota(6,Y).`



Control mediante corte

- Ejemplo de `nota` con cortes

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




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

Control mediante corte

- 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).
```


Negación como fallo

- Negación como fallo:
  - Definición de la negación como fallo (`not`):

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

  - Programa con negación:

```
aprobado(X) :- no(suspenso(X)), matriculado(X).    %
matriculado(juan).                                 %
matriculado(luis).                                 %
suspenso(juan).                                    %
```

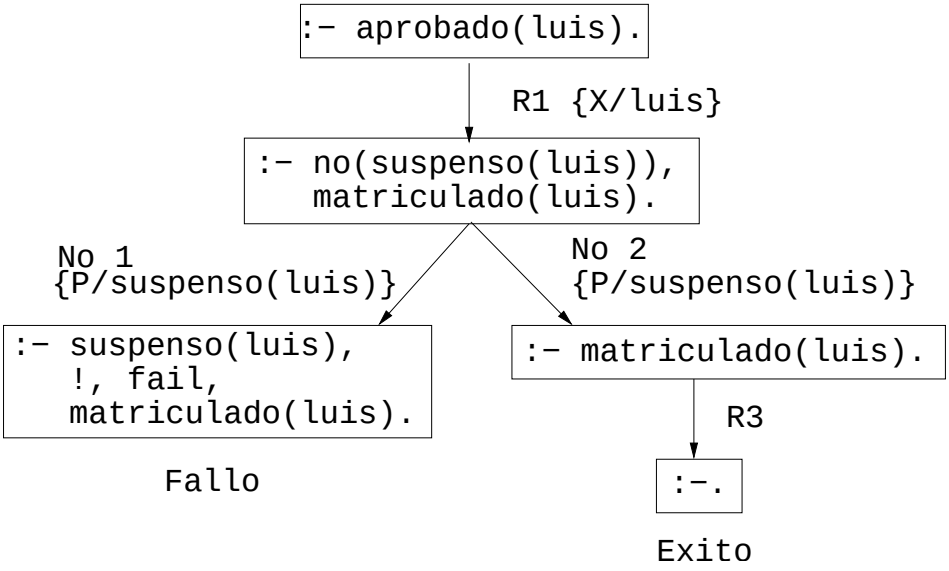
  - Consultas:

```
?- aprobado(luis).
Yes

?- aprobado(X).
No
```

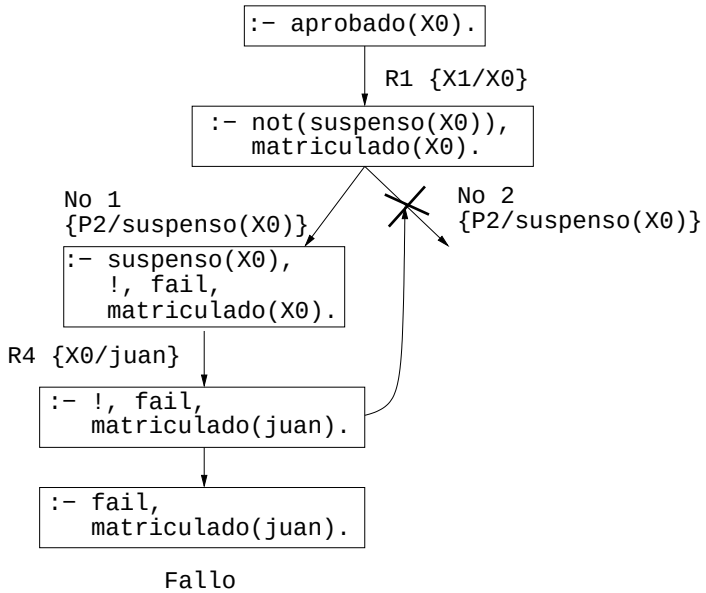
Negación como fallo

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



Negación como fallo

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



Negación como fallo

- Modificación del orden de los literales
  - Programa:

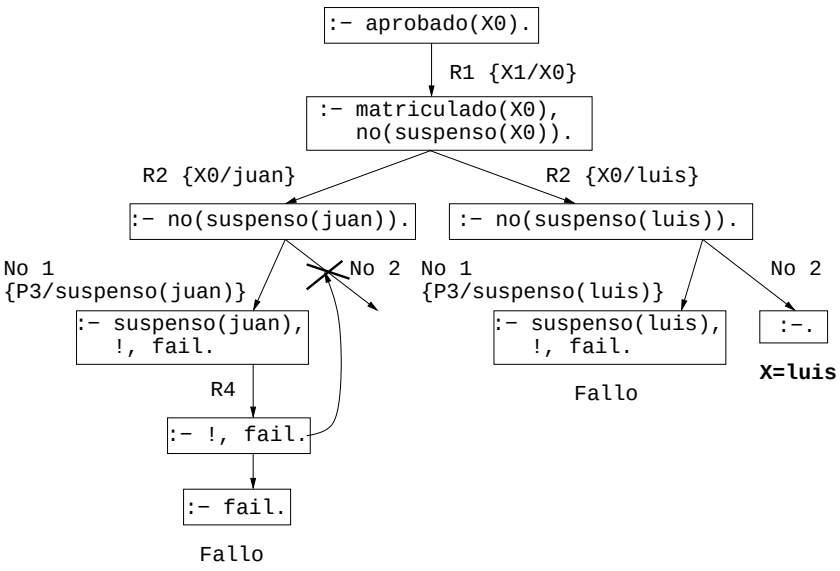
```
aprobado(X) :- matriculado(X), no(suspenso(X)).   %
matriculado(juan).                                %
matriculado(luis).                                %
suspenso(juan).                                   %
```

  - Consulta:

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


Negación como fallo

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



Negación como fallo

- 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
```


Negación como fallo

- Ejemplo de definición con `not` y con corte (cont.):
  - 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).
```

  - 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).
```

Negación como fallo

- Ejemplo de definición con `not` y con corte (cont.):
  - 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).
```


El condicional

- Definición de `nota` con el condicional:

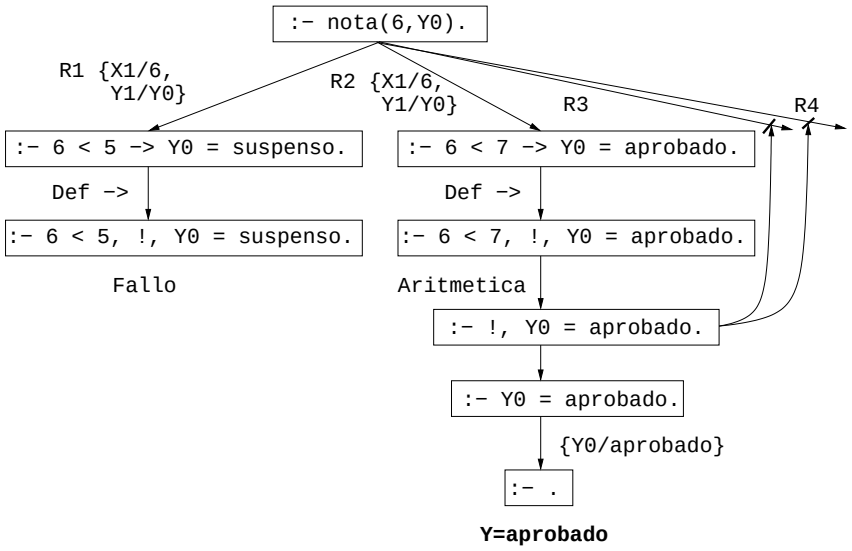
```
nota(X,Y) :-
   X < 5 -> Y = suspenso ;          % R1
   X < 7 -> Y = aprobado ;          % R2
   X < 9 -> Y = notable ;           % R3
   true  -> Y = sobresaliente.      % R4
```

- Definición del condicional y verdad:

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

El condicional

- Árbol de deducción correspondiente a la pregunta `?- nota(6,Y).`



Bibliografía

- J.A. Alonso y J. Borrego
  Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)
  (Ed. Kronos, 2002)
  - Cap. 2: Introducción a la programación lógica con Prolog, pp. 21–29
- I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)*
  (Addison–Wesley, 2001)
  - Cap. 5: “Controlling backtracking”
- W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)*
  (Springer Verlag, 1994)
  - Cap. 4: “Backtracking and the cut”
- L. Sterling y E. Shapiro *The Art of Prolog (2nd Edition)* (The MIT Press, 1994)
  - Cap. 11: “Cuts and negation”

Capítulo 5

Programación lógica de segundo orden

Programación declarativa (2004–05)

Tema 5: Programación lógica de segundo orden

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Modificación de la base de conocimiento

- Predicados `assert` y `retract`:
  - `assert(+Term)` inserta un hecho o una cláusula en la base de conocimientos. `Term` es insertado como última cláusula del predicado correspondiente.
  - `retract(+Term)` elimina la primera cláusula de la base de conocimientos que unifica con `Term`.

```
?- hace_frio.
No
?- assert(hace_frio).
Yes
?- hace_frio.
Yes
?- retract(hace_frio).
Yes
?- hace_frio.
No
```

Modificación de la base de conocimiento

- El predicado `listing`:
  - `listing(+Pred)` lista las cláusulas en cuya cabeza aparece el predicado `Pred`, Por ejemplo,

```
?- assert((gana(X,Y) :- rápido(X), lento(Y))).
?- listing(gana).
gana(A, B) :- rápido(A), lento(B).
?- assert(rápido(juan)),assert(lento(jose)),
   assert(lento(luis)).
?- gana(X,Y).
X = juan  Y = jose ; X = juan  Y = luis ;
No
?- retract(lento(X)).
X = jose ;
X = luis ;
No
?- gana(X,Y).
No
```


Modificación de la base de conocimiento

- Los predicados `asserta` y `assertz`:
  - `asserta(+Term)` equivale a `assert/1`, pero `Term` es insertado como primera cláusula del predicado correspondiente.
  - `assertz(+Term)` equivale a `assert/1`.

```
?- assert(p(a)), assertz(p(b)), asserta(p(c)).
Yes
?- p(X).
X = c ;
X = a ;
X = b ;
No
?- listing(p).
p(c).
p(a).
p(b).
Yes
```

Modificación de la base de conocimiento

- Los predicados `retractall` y `abolish`:
  - `retractall(+C)` elimina de la base de conocimientos todas las cláusulas cuya cabeza unifica con `C`.
  - `abolish(+SimbPred/+Aridad)` elimina de la base de conocimientos todas las cláusulas que en su cabeza aparece el símbolo de predicado `SimbPred/Aridad`.

```
?- assert(p(a)), assert(p(b)).
?- retractall(p(_)).
?- p(a).
No
?- assert(p(a)), assert(p(b)).
?- abolish(p/1).
?- p(a).
[WARNING: Undefined predicate: 'p/1']
No
```


Modificación de la base de conocimiento

- Multiplicaciones:
  - `crea_tabla` añade los hechos `producto(X,Y,Z)` donde `X` e `Y` son números de 0 a 9 y `Z` es el producto de `X` e `Y`. Por ejemplo,

```
?- crea_tabla.
Yes
?- listing(producto).
producto(0,0,0).
producto(0,1,0).
...
producto(9,8,72).
producto(9,9,81).
Yes
```

Modificación de la base de conocimiento

- Multiplicaciones(cont.):
  - Definición:

```
crea_tabla :-
    L = [0,1,2,3,4,5,6,7,8,9],
    member(X,L),
    member(Y,L),
    Z is X*Y,
    assert(producto(X,Y,Z)),
    fail.
crea_tabla.
```

  - Determinar las descomposiciones de 6 en producto de dos números.

```
?- producto(A,B,6).
A=1 B=6 ; A=2 B=3 ; A=3 B=2 ; A=6 B=1 ; No
```


Todas las soluciones

- `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,voc),L).
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 = []
```

Todas las soluciones

- `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,_Y^clase(_X,_Y),L).
L = [a, b, c, e]
?- setof(letra(_X),_Y^clase(_X,_Y),L).
L = [letra(a), letra(b), letra(c), letra(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
```


Todas las soluciones

- `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,_Y^clase(_X,_Y),L).
L = [a, b, e, c]
?- bagof(letra(_X),_Y^clase(_X,_Y),L).
L = [letra(a), letra(b), letra(e), letra(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
```

Todas las soluciones

- Operaciones conjuntistas:
  - `setof0(T,0,L)` es como `setof` salvo en el caso en que ninguna instancia de `T` verifique `O`, en cuyo caso `L` es la lista vacía. Por ejemplo,

```
?- setof0(X,
          (member(X,[c,a,b]),member(X,[c,b,d])),
          L).
L = [b, c]
?- setof0(X,
          (member(X,[c,a,b]),member(X,[e,f])),
          L).
L = []
```

Definición:

```
setof0(X,O,L) :- setof(X,O,L), !.
setof0(_,_,[]).
```


Todas las soluciones

- Operaciones conjuntistas (cont.):
  - `intersección(S,T,U)` se verifica si `U` es la intersección de `S` y `T`. Por ejemplo,

```
?- intersección([1,4,2],[2,3,4],U).
U = [2,4]
```

Definición:

```
intersección(S,T,U) :-
   setof0(X, (member(X,S), member(X,T)), U).
```

  - `unión(S,T,U)` se verifica si `U` es la unión de `S` y `T`. Por ejemplo,

```
?- unión([1,2,4],[2,3,4],U).
U = [1,2,3,4]
```

Definición:

```
unión(S,T,U) :-
   setof(X, (member(X,S); member(X,T)), U).
```

Todas las soluciones

- Operaciones conjuntistas (cont.):
  - `diferencia(S,T,U)` se verifica si `U` es la diferencia de los conjuntos de `S` y `T`. Por ejemplo,
    ```
    ?- diferencia([5,1,2],[2,3,4],U).
    U = [1,5]
    ```
    Definición:
    ```
    diferencia(S,T,U) :-
       setof0(X, (member(X,S), not(member(X,T))), U).
    ```
  - `partes(X,L)` se verifica si `L` es el conjunto de las partes de `X`. Por ejemplo,
    ```
    ?- partes([a,b,c],L).
    L = [[],[a],[a,b],[a,b,c],[a,c],[b],[b,c],[c]]
    ```
    Definición:
    ```
    partes(X,L) :-
       setof(Y,subconjunto(Y,X),L).
    ```


Todas las soluciones

- Operaciones conjuntistas (cont.):
  - `subconjunto(-L1,+L2)` se verifica si `L1` es un subconjunto de `L2`. Por ejemplo,
    ```
    ?- subconjunto(L,[a,b]).
    L = [a, b] ;
    L = [a] ;
    L = [b] ;
    L = [] ;
    No
    ```
    Definición:
    ```
    subconjunto([],[]).
    subconjunto([X|L1],[X|L2]) :-
       subconjunto(L1,L2).
    subconjunto(L1,[_|L2]) :-
       subconjunto(L1,L2).
    ```

Procesamiento de términos

- 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)
```


Procesamiento de términos

- Transformación entre términos y listas (cont.):
  - `alarga(+F1,+N,-F2)` se verifica si `F1` y `F2` son figuras geométricas del mismo tipo y el tamaño de la `F1` es el de la `F2` multiplicado por `N`, donde las figuras geométricas se representan como términos en los que el functor indica el tipo de figura y los argumentos su tamaño; por ejemplo,

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

?- alarga(cuadrado(3),2,F).
F = cuadrado(6)
```

Procesamiento de términos

- Transformación entre términos y listas (cont.):
  - Definición:

```
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).
```


Procesamiento de términos

- Los procedimientos `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)
```

Transformaciones entre átomos y listas

- La relación `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
```


Transformaciones entre átomos y listas

- La relación `name` (cont.):
  - `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
```

Definición:

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

Procedimientos aplicativos

- `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,[1,2,3,4,5]]).=>  X=[3,4,5]
```

- Definición de `apply`:

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


Procedimientos aplicativos

- `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]
```

- Definición de `maplist`:

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

Predicados sobre tipos de término

- 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
```


Predicados sobre tipos de término

- Definir `suma_segura(X,Y,Z)` que se verifique 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]
```

Definición:

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

Comparación y ordenación de términos

- 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
```


Comparación y ordenación de términos

- Definir el predicado `cuenta(A,L,N)` que 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
```

Definición:

```
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).
```

Comparación y ordenación de términos

- 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
```

- Ordenación con `sort`:
  - `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]
```


Bibliografía

- J.A. Alonso y J. Borrego
  Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)
  (Ed. Kronos, 2002)
  - Cap. 2: “Introducción a la programación lógica con Prolog”
- I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)*
  (Addison–Wesley, 2001)
  - Cap. 7: “More Built–in Procedures”
- T. Van Le *Techniques of Prolog Programming* (John Wiley, 1993)
  - Cap. 6: “Advanced programming techniques and data structures”
- W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)*
  (Springer Verlag, 1994)
  - Cap. 6: “Built–in Predicates”

Capítulo 6

Estilo y eficiencia en programación lógica

Programación declarativa (2004–05)

Tema 6: Estilo y eficiencia en programación lógica

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Principios generales

- Un programa debe ser:
  - Correcto.
  - Eficiente.
  - Fácil de leer.
  - Modificable: Modular y transparente.
  - Robusto.
  - Documentado.

Orden de los literales y corrección

- Pensar procedimental y declarativamente:
  - Programas:

```
siguiente(a,1).
siguiente(1,b).

sucesor_1(X,Y) :-
   siguiente(X,Y).
sucesor_1(X,Y) :-
   siguiente(X,Z),
   sucesor_1(Z,Y).
```

  - Sesiones:

```
?- sucesor_1(X,Y).
X = a  Y = 1 ; X = 1  Y = b ; X = a  Y = b ; No
?- findall(X-Y,sucesor_1(X,Y),L).
L = [a-1, 1-b, a-b]
```


Orden de los literales y corrección

- Pensar procedimental y declarativamente:
  - Programas:

```
siguiente(a,1).
siguiente(1,b).

sucesor_2(X,Y) :-
   siguiente(X,Y).
sucesor_2(X,Y) :-
   sucesor_2(Z,Y),
   siguiente(X,Z).
```

  - Sesiones:

```
?- sucesor_2(X,Y).
X = a  Y = 1 ; X = 1  Y = b ; X = a  Y = b ;
ERROR: Out of local stack
?- findall(X-Y,sucesor_2(X,Y),L).
ERROR: Out of local stack
```

Orden de los literales y efi ciencia

- Orden de los literales y eficiencia

```
?- listing(número).
número(1). número(2). ... número(999). número(1000).
Yes

?- listing(múltiplo_de_100).
múltiplo_de_100(100). ... múltiplo_de_100(1000).
Yes

?- time((número(_N),múltiplo_de_100(_N))).
101 inferences in 0.00 seconds (Infinite Lips)
Yes

?- time((múltiplo_de_100(_N),número(_N))).
2 inferences in 0.00 seconds (Infinite Lips)
Yes
```


Combinatoria

- `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
```

- Definiciones de `combinación`:

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

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

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

Combinatoria

- `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]]
```

- Definiciones de `combinaciones`:

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

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).
```


Combinatoria

- Comparación de eficiencia:

```
?- 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 (Infinite Lips)
 92 inferences in 0.00 seconds (Infinite Lips)
?- 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 (2855100 Lips)
   457 inferences in 0.00 seconds (Infinite Lips)
?- 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 (2039242 Lips)
      2,915 inferences in  0.00 seconds (Infinite Lips)
```

Combinatoria

- `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
```


Combinatoria

- `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
```

Definición de `permutación`:

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

Combinatoria

- `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,
```

Definiciones de `variación`

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


Combinatoria

- `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, c]]
```

Definiciones de `variaciones`:

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

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).
```

Combinatoria

- Comparación de eficiencia

```
?- findall(_N,between(1,100,_N),_L1),
   time(variaciones_1(_L1,2,_L2)), time(variaciones_2(_L1,2,
221,320 inferences in 0.27 seconds (819704 Lips)
 40,119 inferences in 0.11 seconds (364718 Lips)
?- findall(_N,between(1,200,_N),_L1),
   time(variaciones_1(_L1,2,_L2)), time(variaciones_2(_L1,2,
1,552,620 inferences in 2.62 seconds (592603 Lips)
  160,219 inferences in 0.67 seconds (239133 Lips)
?- findall(_N,between(1,400,_N),_L1),
   time(variaciones_1(_L1,2,_L2)), time(variaciones_2(_L1,2,
11,545,220 inferences in 19.02 seconds (607004 Lips)
   640,419 inferences in  2.51 seconds (255147 Lips)
```


Ordenación

- `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]
```

- Definición 1 (generación y prueba)

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

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

Ordenación

- Definición 2 (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

- Definición 3 (ordenación rápida (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 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

Genera y prueba

- Cuadrado mágico:
  - 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

  - Programa 1 (por generación y prueba):

```
cuadrado_1([A,B,C,D,E,F,G,H,I]) :-
   permutacion([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.
```

Genera y prueba

- Cuadrado mágico:
  - Sesión 1:

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


Genera y prueba

- Cuadrado mágico:
  - Programa 2 (por comprobaciones parciales):

```
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.
```

Genera y prueba

- Cuadrado mágico:
  - Sesión 2:

```
?- cuadrado_2(L).
L = [2, 7, 6, 9, 5, 1, 4, 3, 8] ;
L = [2, 9, 4, 7, 5, 3, 6, 1, 8]
Yes
?- setof(_X,cuadrado_1(_X),_L),
   setof(_X,cuadrado_2(_X),_L).
Yes
```


Genera y prueba

- Cuadrado mágico:
  - Comparación de eficiencia:

```
?- time(cuadrado_1(_X)).
161,691 inferences in 0.58 seconds (278778 Lips)
?- time(cuadrado_2(_X)).
  1,097 inferences in 0.01 seconds (109700 Lips)
?- time(setof(_X,cuadrado_1(_X),_L)).
812,417 inferences in 2.90 seconds (280144 Lips)
?- time(setof(_X,cuadrado_2(_X),_L)).
  7,169 inferences in 0.02 seconds (358450 Lips)
```

Uso de listas

- En general sólo cuando el número de argumentos no es fijo o es desconocido.
- Ejemplos:

```
?- setof(N,between(1,1000,N),L1),
   asserta(con_lista(L1)),
   Term =.. [f|L1],
   asserta(con_term(Term)).
?- listing(con_lista).
con_lista([1, 2, ..., 999, 1000]).
?- listing(con_term).
con_term(f(1, 2,...,999, 1000)).
?- time((con_lista(_L),member(1000,_L))).
1,001 inferences in 0.00 seconds (Infinite Lips)
?- time((con_term(_T),arg(_,_T,1000))).
2 inferences in 0.00 seconds (Infinite Lips)
```


Uso de la unifi cac̈n

- `intercambia(+T1,-T2)` se verifica si `T1` es un término con dos argumentos y `T2` es un término con el mismo símbolo de función que `T1` pero sus argumentos intercambiados. Por ejemplo,

```
?- intercambia(opuesto(3,-3),T).
T = opuesto(-3, 3)
```

Definiciones:

```
intercambia_1(T1, T2) :-
   functor(T1,F,2),    functor(T2,F,2),
   arg(1,T1,X1),       arg(2,T1,Y1),
   arg(1,T2,X2),       arg(2,T2,Y2),
   X1 = Y2,            X2 = Y1.

intercambia_2(T1,T2) :-
   T1 =.. [F,X,Y],
   T2 =.. [F,Y,X].
```

Uso de la unificación

- `lista_de_tres(L)` se verifica si `L` es una lista de 3 elementos.
- Definiciones:
  - Definición 1:
    ```
    lista_de_tres(L) :- length(L, N), N = 3.
    ```
  - Definición 2:
    ```
    lista_de_tres(L) :- length(L,3).
    ```
  - Definición 3:
    ```
    lista_de_tres([_,_,_]).
    ```


Acumuladores

- `inversa(+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).
  ```
- 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).
  ```

Acumuladores

- 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 (1253753 Lips)
  1,002 inferences in 0.00 seconds (Infinite Lips)

?- findall(_N,between(1,2000,_N),_L1),
   time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).
2,003,001 inferences in 1.59 seconds (1259749 Lips)
    2,002 inferences in 0.00 seconds (Infinite Lips)

?- findall(_N,between(1,4000,_N),_L1),
   time(inversa_1(_L1,_)), time(inversa_2(_L1,_)).
8,006,001 inferences in 8.07 seconds (992070 Lips)
    4,002 inferences in 0.02 seconds (200100 Lips)
```


Uso de lemas

- La sucesión de Fibonacci es `1, 1, 2, 3, 5, 8, ...` y está definida por
  `f(1) = 1`
  `f(2) = 1`
  `f(n) = f(n-1)+f(n-2)`, si `n > 2`
- `fibonacci(N,X)` se verifica si `X` es el `N`–ésimo término de la sucesión de Fibonacci.

```
fibonacci_1(1,1).
fibonacci_1(2,1).
fibonacci_1(N,F) :-
   N > 2,
   N1 is N-1,
   fibonacci_1(N1,F1),
   N2 is N-2,
   fibonacci_1(N2,F2),
   F is F1 + F2.
```

Uso de lemas

- Definición de Fibonacci con lemas

```
:- dynamic fibonacci_2/2.

fibonacci_2(1,1).
fibonacci_2(2,1).
fibonacci_2(N,F) :-
   N > 2,
   N1 is N-1,
   fibonacci_2(N1,F1),
   N2 is N-2,
   fibonacci_2(N2,F2),
   F is F1 + F2,
   asserta(fibonacci_2(N,F)).
```


Uso de lemas

- Comparación

```
?- time(fibonacci_1(20,N)).
40,585 inferences in 1.68 seconds (24158 Lips)
N = 6765
?- time(fibonacci_2(20,N)).
127 inferences in 0.00 seconds (Infinite Lips)
N = 6765
?- listing(fibonacci_2).
fibonacci_2(20, 6765). fibonacci_2(19, 4181). ... fibonacci_
?- time(fibonacci_2(20,N)).
3 inferences in 0.00 seconds (Infinite Lips)
N = 6765
```

Uso de lemas

- Definición de Fibonacci con un acumulador

```
fibonacci_3(N,F) :-  fibonacci_3_aux(N,_,F).
fibonacci_3_aux(0,_,0).
fibonacci_3_aux(1,0,1).
fibonacci_3_aux(N,F1,F) :-
   N > 1,
   N1 is N-1,
   fibonacci_3_aux(N1,F2,F1),
   F is F1 + F2.
```

- Comparación;

```
?- time(fibonacci_1(20,N)).
40,585 inferences in 1.68 seconds (24158 Lips)
?- time(fibonacci_2(20,N)).
127 inferences in 0.00 seconds (Infinite Lips)
?- time(fibonacci_3(20,N)).
21 inferences in 0.00 seconds (Infinite Lips)
```


Determinismo

- `descompone(+E,-N1,-N2)` se verifica si `N1` y `N2` son dos enteros no negativos tales que `N1+N2=E`.

```
descompone_1(E, N1, N2) :-
   between(0, E, N1),
   between(0, E, N2),
   E =:= N1 + N2.
descompone_2(E, N1, N2) :-
   between(0, E, N1),
   N2 is E - N1.
```

- Comparación:

```
?- time(setof(_N1+_N2,descompone_1(1000,_N1,_N2),_L)).
1,004,019 inferences in 1.29 seconds (778309 Lips)
?- time(setof(_N1+_N2,descompone_2(1000,_N1,_N2),_L)).
2,018 inferences in 0.01 seconds (201800 Lips)
```

Añadir al principio

- `lista_de_cuadrados(+N,?L)` se verifica si `L` es la lista de los cuadrados de los números de `1` a `N`. Por ejemplo,

```
?- lista_de_cuadrados(5,L).
L = [1, 4, 9, 16, 25]
```

- Programa 1 (añadiendo por detrás):

```
lista_de_cuadrados_1(1,[1]).
lista_de_cuadrados_1(N,L) :-
   N > 1,
   N1 is N-1,
   lista_de_cuadrados_1(N1,L1),
   M is N*N,
   append(L1,[M],L).
```


Añadir al principio

- Programa 2 (añadiendo por delante):

```
lista_de_cuadrados_2(N,L) :-
   lista_de_cuadrados_2_aux(N,L1),
   reverse(L1,L).

lista_de_cuadrados_2_aux(1,[1]).
lista_de_cuadrados_2_aux(N,[M|L]) :-
   N > 1,
   M is N*N,
   N1 is N-1,
   lista_de_cuadrados_2_aux(N1,L).
```

- Programa 3 (con `findall`):

```
lista_de_cuadrados_3(N,L) :-
   findall(M,(between(1,N,X), M is X*X),L).
```

Añadir al principio

- Comparación:

```
?- tiempo(lista_de_cuadrados_1(10000,_L)).
50,015,003 inferencias en 68.79 segundos y 549,416 bytes.
Yes

?- tiempo(lista_de_cuadrados_2(10000,_L)).
20,007 inferencias en 0.03 segundos y 309,464 bytes.
Yes

?- tiempo(lista_de_cuadrados_3(10000,_L)).
20,016 inferencias en 0.03 segundos y 189,588 bytes.
Yes
```


Listas de diferencias

- Representaciones de `[a,b,c]` como listas de diferencias:

```
[a,b,c,d] - [d]
[a,b,c,1,2,3] - [1,2,3]
[a,b,c|X] - [X]
[a,b,c] - []
```

- Concatenación de listas de diferencias:
  - Programa:
    `conc_ld(A-B,B-C,A-C).`
  - Sesión:

```
?- conc_ld([a,b|RX]-RX,[c,d|RY]-RY,Z-[]).
RX = [c, d]   RY = []   Z = [a, b, c, d]
?- conc_ld([a,b|_RX]-_RX,[c,d|_RY]-_RY,Z-[]).
Z = [a, b, c, d]
Yes
```

Bibliografía

- I. Bratko *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
  - Cap. 8: “Programming Style and Technique”
- W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
  - Cap. 6: “Using Data Structures”
- M.A. Covington *Efficient Prolog: A Practical Guide*

Capítulo 7

Aplicaciones de programación declarativa

Programación declarativa (2004–05)

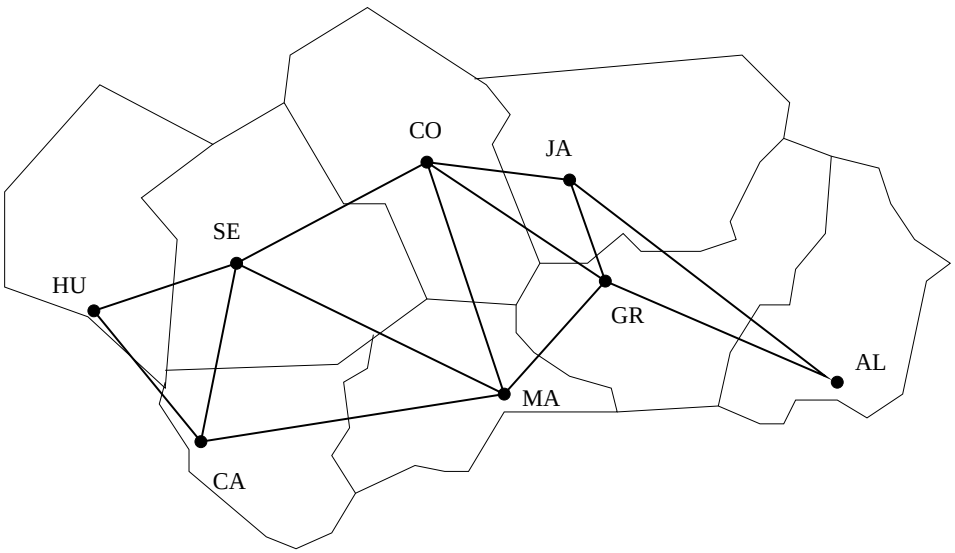
Tema 7: Aplicaciones de PD: problemas de grafos y el problema de las reinas

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Grafos

- Grafo de Andalucía:



Grafos

- 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]).
```

  - `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).
```


Grafos: 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
```

  Definición de `camino`

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

Grafos: 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 losde `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).
```


Grafos: 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, huel
?- findall(_C,hamiltoniano(_C),_L), length(_L,N).
N = 16
```

Definición de `hamiltoniano`

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

Grafos: Caminos hamiltonianos

- 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)
```


Grafos: Generacion de grafos completos

- `completo(+N,-G)` se verifica si `G` es el grafo completo de orden `N`.
  Por ejemplo,

```
completo(1,G) => G = []
completo(2,G) => G = [1-2]
completo(3,G) => G = [1-2,1-3,2-3]
completo(4,G) => G = [1-2,1-3,1-4,2-3,2-4,3-4]
```

  Definición:

```
completo(N,G) :-
   findall(X-Y,arco_completo(N,X,Y),G).

arco_completo(N,X,Y) :-
   N1 is N-1,
   between(1,N1,X),
   X1 is X+1,
   between(X1,N,Y).
```

Grafos: Generación de grafos aleatorios

- `aleatorio(+P,+N,-G)` se verifica si `G` es un subgrafo de `{1..N}x{1..N}`, donde cada arco se ha elegido con la probabilidad `P` (`0` $\leq$ `P` $\leq$ `1`). Por ejemplo,

```
?- aleatorio(0.3,5,G).
G = [1-2, 3-4, 4-5]
?- aleatorio(0.3,5,G).
G = [1-2, 2-4, 3-4, 4-5]
```

  Definición:

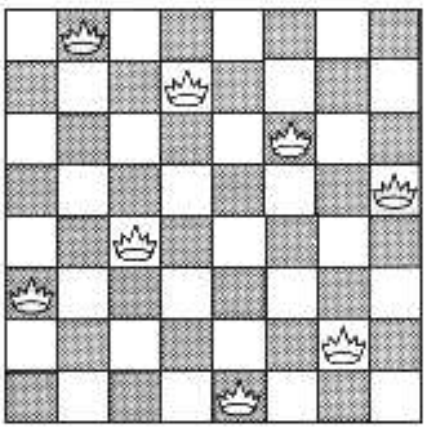
```
aleatorio(P,N,G) :-
   findall(X-Y,
           (arco_completo(N,X,Y),random(Z),Z=<P),
           G).
```

- `random(X)` se verifica si `X` es un número aleatorio en el intervalo `[0,1]`.

```
random(X) :- Y is random(1000), X is Y/1000.
```


El problema de las 8 reinas

- El problema de las ocho reinas consiste en colocar 8 reinas en un tablero rectangular de dimensiones 8 por 8 de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.



El problema de las 8 reinas: Representación 1

- Sesión:

```
?- tablero(S), solución(S).
S = [1-4, 2-2, 3-7, 4-3, 5-6, 6-8, 7-5, 8-1] ;
S = [1-5, 2-2, 3-4, 4-7, 5-3, 6-8, 7-6, 8-1] ;
S = [1-3, 2-5, 3-2, 4-8, 5-6, 6-4, 7-7, 8-1]
Yes
```

- `tablero(L)` se verifica si `L` es una lista de posiciones que representan las coordenadas de 8 reinas en el tablero.

```
tablero(L) :-
   findall(X-_Y,between(1,8,X),L).
```


El problema de las 8 reinas: Representación 1

- `solución_1(?L)` se verifica si `L` es una lista de pares de números que representan las coordenadas de una solución del problema de las 8 reinas.

```
solución_1([]).
solución_1([X-Y|L]) :-
   solución_1(L),
   member(Y,[1,2,3,4,5,6,7,8]),
   no_ataca(X-Y,L).
```

- `no_ataca([X,Y],L)` se verifica si la reina en la posición `(X,Y)` no ataca a las reinas colocadas en las posiciones correspondientes a los elementos de la lista `L`.

```
no_ataca(_,[]).
no_ataca(X-Y,[X1-Y1|L]) :-
   X =\= X1,        Y =\= Y1,
   X-X1 =\= Y-Y1,   X-X1 =\= Y1-Y,
   no_ataca(X-Y,L).
```

El problema de las 8 reinas: Representación 2

- `solución_2(L)` se verifica si `L` es una lista de 8 números, $[n_1, \ldots, n_8]$, de forma que si las reinas se colocan en las casillas $(1, n_1), \ldots, (8, n_8)$, entonces no se atacan entre sí.

```
solución_2(L) :-
   permutación([1,2,3,4,5,6,7,8],L),
   segura(L).
```

- `segura(L)` se verifica si `L` es una lista de `m` números $[n_1, \ldots, n_m]$ tal que las reinas colocadas en las posiciones $(x, n_1), \ldots, (x + m, n_m)$ no se atacan entre sí.

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


El problema de las 8 reinas

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

El problema de las 8 reinas: Representación 3

- `solución_3(?L)` se verifica si `L` es una lista de 8 números, $[n_1, \ldots, n_8]$, de forma que si las reinas se colocan en las casillas $(1, n_1), \ldots, (8, n_8)$, entonces no se atacan entre sí.

```
solución_3(L) :-
   solución_3_aux(
      L,
      [1,2,3,4,5,6,7,8],
      [1,2,3,4,5,6,7,8],
      [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
      [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).
```


El problema de las 8 reinas

- `solución_3_aux(?L,+Dx,+Dy,+Du,+Dv)` se verifica si `L` es una permutación de los elementos de `Dy` de forma que si `L` es $[y_1, \ldots, y_n]$ y `Dx` es $[1, \ldots, n]$, entonces $y_j - j \ (1 \leq j \leq n)$ son elementos distintos de `Du` e $y_j + j \ (1 \leq j \leq n)$ son elementos distintos de `Dv`.

```
solución_3_aux([],[],Dy,Du,Dv).
solución_3_aux([Y|Ys],[X|Dx1],Dy,Du,Dv) :-
   select(Dy,Y,Dy1),
   U is X-Y,
   select(Du,U,Du1),
   V is X+Y,
   select(Dv,V,Dv1),
   solución_3_aux(Ys,Dx1,Dy1,Du1,Dv1).
```

El problema de las 8 reinas

- Comparaciones

```
?- time((findall(_S,(tablero_1(_S), solucion_1(_S)),_L),leng
211,330 inferences in 0.12 seconds (1761083 Lips)
N = 92

?- time((findall(_S,solución_2(_S),_L),length(_L,N))).
1,422,301 inferences in 0.72 seconds (1975418 Lips)
N = 92

?- time((findall(_S,solucion_3(_S),_L),length(_L,N))).
120,542 inferences in 0.07 seconds (1722029 Lips)
N = 92
```


Búsqueda de todas las soluciones para N reinas:

	solución 1		solución 2		solución 3	
N	inferencias	seg.	inferencias	seg.	inferencias	seg.
4	401	0.00	543	0.00	546	0.00
6	8,342	0.00	20,844	0.01	6,660	0.01
8	195,628	0.15	1,422,318	0.91	120,614	0.09
10	5,303,845	4.05	150,300,540	96.82	2,774,095	2.01
12	182,574,715	147.22			83,067,721	64.93

Bibliografía

- I. Bratko *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
  - Cap. 4: “Using Structures: Example Programs”
  - Cap. 9: “Operations on Data Structures”
- L. Sterling y E. Shapiro *The Art of Prolog (2nd edition)* (The MIT Press, 1994)
  - Cap. 2 “Database programming”

Capítulo 8

Procesamiento de lenguaje natural

Programación declarativa (2004–05)

Tema 8: Procesamiento de lenguaje natural

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

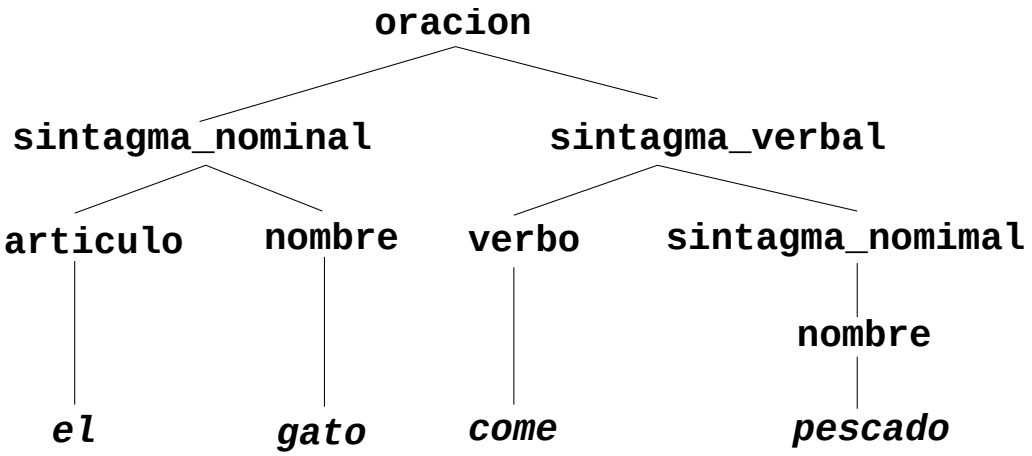
Gramáticas libres de contexto: Ejemplo

- 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]
```

Gramáticas libres de contexto: Árbol de análisis

- Árbol de análisis



Gramáticas libres de contexto: Definiciones

- 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 $w \in V$

Reconocedor de GLC mediante append

- Representación de oraciones en Prolog

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

- Reconocedor de GLC en Prolog mediante append:
  - Sesión (con coste)

```
?- 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

```
oración(O)            :- sintagma_nominal(SN),
                         sintagma_verbal(SV),
                         append(SN,SV,O).
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]
Yes
?- findall(_O,oración(_O),_L),length(_L,N).
N = 64
```

  - 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]]
```


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

  - 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]]
```

Gramáticas de cláusulas definidas: Ejemplo

- Ejemplo de GCD
  - 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].
```


Gramáticas de cláusulas definidas: Usos

- 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
```

Gramáticas de cláusulas definidas: Usos

- 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]]
```

- Determinacion de elementos

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


Gramáticas de cláusulas definidas: Usos

- La relación `phrase`

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

Gramáticas de cláusulas definidas: Compilación

- Compilación

```
?- 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).
```


Reglas recursivas en GCD: Primera propuesta

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

```
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: Primera propuesta

- Sesión

```
?- 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

- 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: Segunda propuesta

- Sesión

```
?- 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

- 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: Tercera propuesta

- Sesión

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


GCD para un lenguaje formal

- 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]
Yes
```

  - GCD

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

Árbol de análisis con GCD

- Sesión

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

- Definición

```
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,nombre]).
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).

nombre(n(gato),[gato|A],A).
nombre(n(perro),[perro|A],A).
nombre(n(pescado),[pescado|A],A).
nombre(n(carne),[carne|A],A).
```

Concordancia de género en GCD

- Sesión

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


Concordancia de género en GCD

- Definición

```
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

- Sesión

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


Concordancia de número en GCD

- Definición

```
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].
```

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

Separación de reglas y lexicón

- Sesión

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

- 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)}.
```

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”

Capítulo 9

Metaprogramación

Programación declarativa (2004–05)

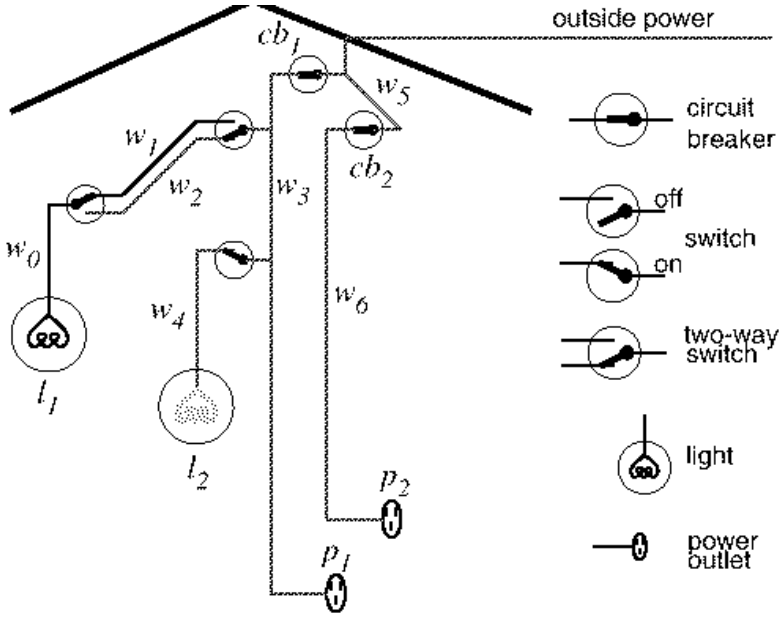
Tema 9: Metaprogramación

José A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

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.
```

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

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

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

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


Ejemplo de BC objeto

- `conectado(?D1,?D2)` se verifica si los dispositivos `D1` y `D2` está conectados (de forma que puede fluir la corriente eléctrica 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) <- está_bien(cc1).
conectado(e2,c6) <- verdad.
conectado(c6,c5) <- está_bien(cc2).
conectado(c5,entrada) <- verdad.
```

Ejemplo de BC objeto

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

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

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

```
está_encendida(L) <-
   luz(L) &
   está_bien(L) &
   tiene_corriente(L).
```

- Sesión con la BC y el metaintérprete simple:

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


Metaintérprete simple

- 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).
```

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)).
```

Metaintérprete con profundidad acotada

- Ejemplo:
  - Programa objeto:
    ```
    hermano(X,Y) <- hermano(Y,X).
    hermano(b,a) <- verdad.
    ```
  - Sesión:
    ```
    ?- prueba(hermano(X,Y)).
    ERROR: Out of local stack

    ?- prueba_pa(hermano(X,Y),1).
    X = a  Y = b ;
    X = b  Y = a ;
    No
    ```


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 preguntas

- Ejemplo: Modificación de `i_electrica.pl`

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

- Sesión

```
?- prueba_p(está_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
```


Metaintéprete con preguntas

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

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

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).
```


Metaintéprete con preguntas

- `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,_)).
```

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

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

Metaintéprete con preguntas

- `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).
```


Metaintérprete con árbol de prueba

```
?- [meta_con_explicacion_arbol, i_electrica].
?- prueba_con_demostracion(está_encendida(L),T).
L = l2
T = si(está_encendida(l2),
        (si(luz(l2), verdad) &
         si(está_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(está_bien(cc1), verdad)) &
                        si(tiene_corriente(c5),
                            (si(conectado(c5, entrada), verdad)
                            si(tiene_corriente(entrada), verdad
```

Metaintérprete con árbol de prueba

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

```
prueba_con_demostración((verdad,verdad).
prueba_con_demostración(((A & B),(PA & PB)) :-
   prueba_con_demostración((A,PA),
   prueba_con_demostración((B,PB).
prueba_con_demostración((O,si(O,PB)) :-
   (O <- B),
   prueba_con_demostración((B,PB).
```


Metaintérprete con explicación

- Sesión

```
?- [meta_con_explicacion_como, i_electrica].
Yes

?- prueba_con_explicación(está_encendida(L)).
está_encendida(l2) :-
   1: luz(l2)
   2: está_bien(l2)
   3: tiene_corriente(l2)
|: 3.
tiene_corriente(l2) :-
   1: conectado(l2, c4)
   2: tiene_corriente(c4)
|: 2.
```

Metaintérprete con explicación

- Sesión

```
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 explicación

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

```
prueba_con_explicación(O) :-
   prueba_con_demostración((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 explicación

- `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 explicación

- `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).
```

Bibliografía

- Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)
  - Cap. 5: “Knowledge engineering”

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] P. Blackburn, J. Bos y K. Striegnitz *Learn Prolog Now!.*
`http://www.coli.uni-sb.de/~kris/learn-prolog-now`

[3] I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.).* (Addison–Wesley, 2001).

[4] W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition).* (Springer Verlag, 1994).

[5] M.A. Covington *Efficient Prolog: A Practical Guide.*
`http://www.ai.uga.edu/ftplib/ai-reports/ai198908.pdf`

[6] M.A. Covington, D. Nute y A. Vellino *Prolog Programming in Depth.* (Prentice Hall, 1997).

[7] P. Flach *Simply Logical (Intelligent Reasoning by Example).* (John Wiley, 1994).

[8] U. Nilsson y J. Maluszynski *Logic, Programming and Prolog (2nd ed.).*
`http://www.ida.liu.se/~ulfni/lpp`

[9] R.A. O'Keefe *The Cratf of Prolog.* (The MIT Press, 1990).

[10] L. Sterling y E. Shapiro *The Art of Prolog.* (MIT Press, 1994).

[11] T. Van Le *Techniques of Prolog Programming.* (John Wiley, 1993).