

Laboratorio de Conocimiento y Razonamiento Automatizado

Francisco Javier Bueno Guillén
José Enrique Morais San Miguel
Adrián Domínguez Díaz



- 0.- Introducción al Prolog
- 1.- Prolog - Juego
- 2.- Prolog - Procesamiento en Lenguaje Natural
- 3.- Scheme (λ -Cálculo)

Desarrollo y Evaluación de las Prácticas

- La mayor parte debe realizarse en las propias sesiones.
- El día de las defensa traer los programas en un pendrive.
- Se tendrá en cuenta el trabajo realizado en el laboratorio.
- A la hora de evaluar las prácticas, se tendrán en cuenta aspectos como: calidad del código, estilo de programación, limpieza del código, comentarios,....
- La defensa de las prácticas se realizará, salvo causas de fuerza mayor, los siguientes días: Práctica 1 el 12 de marzo, Práctica 2 el 16 de abril y Práctica 3 en una fecha por determinar en mayo.

Clásicos de la programación en Prolog:

- W. Clocksin, C. Mellish, “Programming in PROLOG”, Springer–Verlag.
Libro que describe, de facto, el estándar de Prolog.
- W. Clocksin, “Clauses and Effects”, Springer–Verlag.
Otro buen libro de Prolog.
- L. Sterling, E. Shapiro, “The Art of Prolog”, 3th Ed, MIT Press, 1999.
Otro clásico.

Pueden ser útiles los siguientes enlaces:

- P. Brna, “Prolog programming”

[http://comp.mq.edu.au/units/comp248/resources
/brna-prolog-book.pdf](http://comp.mq.edu.au/units/comp248/resources/brna-prolog-book.pdf)

- U. Nilsson, J. Maluszynski, “Logic Programming and Prolog”,
2nd Ed, Wiley, 1995.

<http://www.ida.liu.se/~ulfni/lpp/>

- Algunas notas y transparencias sobre Prolog en

<http://staff.science.uva.nl/~ulle/teaching/prolog/>

SWI-Prolog

- Ejecutables, código fuente y manuales en

<http://www.swi-prolog.org/>

- Versiones para Windows, Unix y Mac OS.
- Entorno de desarrollo: Eclipse

<http://www.eclipse.org>

con el plugin:

<http://prodevtools.sourceforge.net/>

- El Prolog está basado en la lógica y es **interpretado** (típicamente).
- Los programas tienen **hechos** y **reglas**.
- Los **hechos** son informaciones y premisas sobre el problema a resolver.
- Las **reglas** indican cómo inferir información de los hechos.
- Los programas se ejecutan mediante **preguntas** al sistema.
- Las **preguntas** indican la información que queremos obtener.

Los **hechos** codifican información sobre el problema.

Ejemplo

```
padre(juan,pedro).% Juan es el padre de Pedro  
padre(fernando,pedro).% ...  
madre(alonso,maria).% ...
```

- Los hechos, las preguntas y las reglas terminan siempre en ''
- Los comentarios se indican con %

Preguntas

Las **preguntas** sirven para **ejecutar** el programa.

Ejemplo

```
?-padre(fernando,pedro).
```

True.

```
?-padre(fernando,paco).
```

False.

```
?-padre(fernando,X).
```

X=pedro.

True.

Las variables comienzan **siempre** con **mayúscula**.

Reglas

Las **reglas** indican cómo extraer información sobre el problema.

Ejemplo

```
hermanos(X,Y):-padre(X,Z),padre(Y,Z).
```

```
hermanos(X,Y):-madre(X,Z),madre(Y,Z).
```

% Son hermanos si tienen el mismo padre o

% la misma madre.

La ',' indica la conjunción lógica, el ';' la disyunción y '\+' la negación.

Ejemplo

```
?-hermanos(juan,fernando).
```

True.

```
?-hermanos(juan,X).
```

X=juan

X=fernando

¿Cómo contesta a las preguntas el Prolog (1)?

- Lee el programa de manera descendente (**¡el orden importa!**).
- Busca hechos o reglas que se correspondan con la pregunta.
- Intenta averiguar el valor de las variables para que la pregunta se correspondan con hechos o reglas.

Ejemplo

```
padre(juan, pedro).  
padre(fernando, pedro).  
madre(alonso, maria).  
?- padre(juan, X).
```

X=pedro

True.

Si la pregunta es compuesta, se contestan las subpreguntas de izquierda a derecha (**¡el orden sigue importando!**).

¿Cómo contesta a las preguntas el Prolog (2)?

Ejemplo

```
padre(juan,pedro).  
padre(fernando,pedro).  
madre(alonso,maria).
```

```
?-padre(fernando,X),padre(Y,X).
```

Primero considera la primera subpregunta

```
--> padre(fernando,X)
```

Obtiene como respuesta

```
--> X=pedro
```

Substituye X por pedro en lo que resta y considera la subpregunta

```
--> padre(Y,pedro)
```

*Da como **primera** respuesta a la pregunta global*

Y=juan

...

Representación de listas

- Una lista es una colección de objetos entre corchetes.
- Se usa el operador | para acceder al primer elemento y a la lista de los restantes.
- El operador | también sirve para construir listas.

Ejemplo

```
?-[X|Y]=[0,1,2,3,4,a,b,5,6].
```

```
X=0
```

```
Y=[1,2,3,4,a,b,5,6]
```

```
True.
```

```
?-X=[0|[1,2]].
```

```
X=[0,1,2]
```

```
True.
```

Más Ejemplos

Ejemplo

```
?- [X,Y|Z]=[0,1,2,3,4,a,b,5,6].
```

X=0

Y=1

Y=[2,3,4,a,b,5,6]

True.

```
?- [X,0|Z]=[0,Y,2,3,4,a,b,5,6].
```

X=0

Y=0

Y=[2,3,4,a,b,5,6]

True.

Operaciones con listas

- **append/3**: concatena dos listas.
- **union/3**: une dos listas (elimina duplicados).
- **member/2**: verifica si un término está en la lista.
- **length/2**: da la longitud de la lista.
- **reverse/2**: invierte la lista.

Ejemplo

```
?-append([1,2],[3,4],X).  
    X=[1,2,3,4] ...  
  
?-append([1,2],X,[1,2,3,4]).  
    X=[3,4] ...  
  
?-union([1,2],X,[1,2,5,6]).  
    X=[1,2,5,6] ...  
  
?-member(X,[1,2,3]).  
    X=1  
    X=2 ...
```

Recorrido de Listas

- Para recorrer listas hay que construir reglas recursivas.
- Estas reglas indican qué es lo que hay que hacer con los primeros elementos y cuándo hay que parar.

Ejemplo

```
list_longitud([_ |Y],N) :- list_longitud(Y,N1),  
                      N is N1 + 1.  
  
list_longitud([],N) :- N is 0.
```

La variable `_` se usa cuando no nos interesa el resultado.

Ejemplo

```
write_list(X) :- member(Y,X), write(Y), fail.
```

`fail` es siempre falso (**¡no encuentra respuesta!**).

- Los términos compuestos o estructuras reúnen varios átomos de forma ordenada. Tienen **funtor** y **argumentos**.
- Se puede acceder a sus elementos empleando el operador '`=..`' que *devuelve* una lista.
- **¡Un átomo es una estructura sin argumentos!**
- **¡Las listas son estructuras!**

Ejemplo

```
?-padre(juan,pedro) =.. X.  
    X=[padre,juan,pedro]  
?-X =.. [hombre,juan].  
    X=hombre(juan) ...  
?-paco=.. X.  
    X=[paco] ...  
?- [1,2,3]=... .(1,.(2,.(3,[]))).  
    True
```

I/O

- Para escribir en pantalla se usa el predicado `write`. Se evalúa siempre a True pero tiene el efecto secundario de producir salida por pantalla.
- Para leer de teclado se utiliza el predicado `read`
- Se puede acceder a sus elementos empleando el operador '`=..`' que *devuelve* una lista.

```
:read(X),write('Has dicho...'),write(X).
```

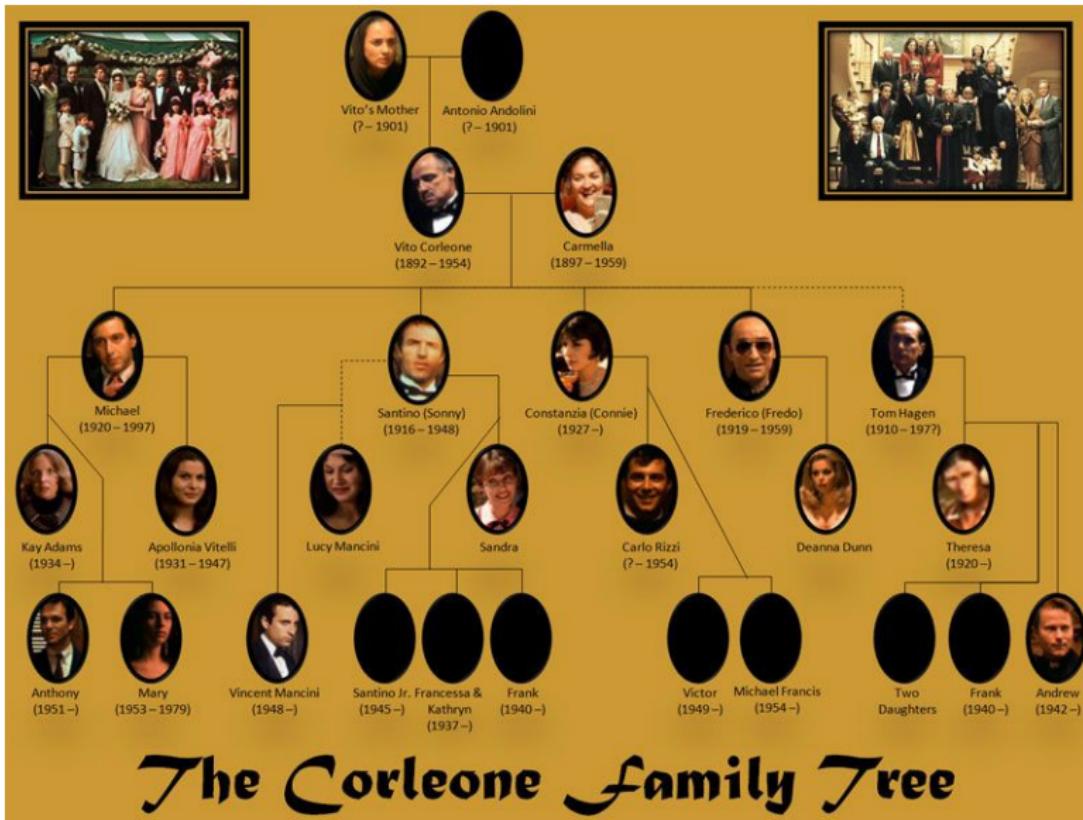
Para consultar (cargar en memoria) programas se utiliza `consult`:

```
:consult('c:/temp/test.pl').
```

En esta práctica se deben modelizar las relaciones familiares que se dan entre varios individuos. Para ello se debe considerar el árbol genealógico de la familia Corleone de la siguiente transparencia y que puede encontrarse en

[http://www.destinationhollywood.com/movies/
godfather/feature_familytree.shtml](http://www.destinationhollywood.com/movies/godfather/feature_familytree.shtml)

I'm gonna make him an offer he cannot refuse



Escribir los siguientes predicados (teniendo en cuenta el árbol anterior):

- `ancestro(X,Y)`: Y es ancestro de X si es progenitor de X o progenitor de un ancestro de X.
- `descendientes(X,Y)` devuelve la lista de los descendientes de X de los más lejanos a los más cercanos.
- Para las relaciones hermano, abuelo, nieto, tío, sobrino, suegro, cuñado, yerno y nuera, escribir un predicado `relacion(X,Y)` que se verifique como cierto si Y está conectado con X mediante la correspondiente relación. Por ejemplo, `tio(X,Y)` es cierto si Y es tío de X.
- `relacion(X,Y,Relacion)` cierta si `Relacion(X,Y)` es cierta y Relacion es una de las relaciones anteriores. Para esto, se necesita el predicado unario `call` que “lanza” otro predicado. Así, `call(p)` ejecutaría el predicado p. Ejemplo:
`call(write(``Hello World!'',nl)).`

Podemos identificar el tipo de término vía:

- `var(X)`: cierto si X es una variable
- `nonvar(X)`: cierto si X no es una variable
- `atom(X)`: cierto si X es un átomo
- `integer(X)`: cierto si X es un entero
- `atomic(X)`: cierto si X es un átomo o un entero

Además de las operaciones vistas hasta el momento, también están disponibles:

- `select(Elem,List1,List2)`: es cierto cuando List2 es el resultado de eliminar `Elem` del `List1`.
- `nth1(N,List,Elem)`: es cierto cuando el elemento en la posición `N` de `List` es `Elem`.
- `sort(L,L1)`: es cierto si `L1` es la lista `L` ordenada según el orden usual.
- `random(N1,N2,X)` que elige “aleatoriamente” un número entero `X`, si `N1` y `N2` son enteros, tal que $N1 \leq X < N2$.

Podemos obtener información de la estructuras mediante:

- `functor(T,F,N)`: cierto si T es un predicado con functor F y número de argumentos N
- `arg(N,T,A)`: cierto si A es el argumento número N de T
- `X =.. L`: cierto si L es la lista formada por el functor de tt X y sus argumentos (manteniendo el orden)
- `name(A,L)`: es cierto si L es la lista de códigos ASCII del átomo A

Ejemplo

```
padre(juan,pedro).  
padre(paco,pedro).  
padre(maria,pedro).  
primogenito(X,Y):-padre(Y,X).
```

Tal y como está definido `primogenito`, a la pregunta
`?-primogenito(pedro,X)` no sólo devuelve `X=juan`, si no que
también puede devolver `X=paco`,...Se necesita cortar la búsqueda
de soluciones de Prolog.

La cortadura (y2)

La cortadura permite controlar las búsquedas de Prolog, cortando las mismas. Las variables unificadas hasta el momento no vuelven a reunificarse en caso de fallo posterior.

Ejemplo (correcto)

```
padre(juan,pedro).  
padre(paco,pedro).  
padre(maria,pedro).  
primogenito(X,Y):-padre(Y,X),!.
```

En Prolog, el predicado `repeat` permite obtener múltiples soluciones vía *backtracking*. Aún siendo un predicado *built-in*, se puede definir como

```
repeat.  
repeat:-repeat.
```

Ejemplo (correcto)

```
?- repeat,padre(X,Y).
```