

Laboratorio de Conocimiento y Razonamiento Automatizado

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



Universidad
de Alcalá

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 la 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 8 de marzo, Práctica 2 el 12 de abril y Práctica 3 el 10 de 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”

`https://courses.cs.washington.edu/courses/cse341/03sp/brna.pdf`

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

`https://www.ida.liu.se/~ulfni53/lpp/bok/bok.pdf`

- 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**.

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.
```

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!**).

Estructuras

- 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`:

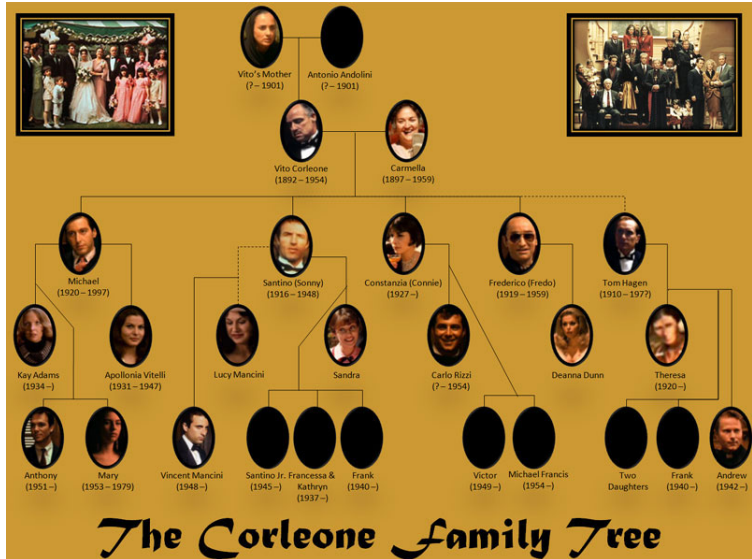
```
:-consult('c:/temp/test.pl').
```

Práctica 0: Introducción

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

I'm gonna make him an offer he cannot refuse



Práctica 0: Objetivos

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

- `ancestro(X, Y)` : X es ancestro de Y si es progenitor de Y o progenitor de un ancestro de Y.
- `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 X está conectado con Y mediante la correspondiente relación. Por ejemplo, `tio(X, Y)` es cierto si X es tío de Y.
- `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 permita 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), !.
```

Repeticiones

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

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

Ejemplo (correcto)

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

Práctica 1: Akinator

El objetivo de la práctica que se propone es realizar una versión propia del juego Akinator (<https://es.akinator.com/>) que consiste en un sistema inteligente que, mediante un sistema de preguntas, adivina un personaje, real o ficticio, que el jugador está pensando.

A lo largo del juego el genio va haciendo una serie de preguntas sencillas que tienen que ser respondidas con alguna de las siguientes cinco opciones: Sí, No, No lo sé, Probablemente o Probablemente no. Dependiendo de las respuestas, el programa irá reduciendo sus opciones.

Akinator puede hacer en cada ronda hasta 25 preguntas mostrando al final del proceso una respuesta que el usuario debe indicar si es correcta o no. De no ser la respuesta buscada, el programa permite añadir al personaje a su base de conocimiento.

Práctica 1: Preparativos del juego

El objetivo de la práctica es implementar una versión propia de Akinator de acuerdo a las siguientes restricciones:

- La solución propuesta debe basarse en el uso de listas para construir la Base de Conocimiento del programa. Éstas tienen que relacionar de algún modo cada uno de los personajes con su conjunto de características propias.
- Cada grupo debe elegir una temática diferente en la medida de lo posible. Algunas ideas son, por ejemplo personajes de juegos/series/libros, modelos de coches, novelas, películas, actores, deportistas, lugares del mundo, marcas comerciales, videojuegos, animales, etc.
- Para construir la Base de Conocimiento se exige un mínimo de 10 características por personaje aunque el valor queda a elección del grupo de alumnos.

Práctica 1: Desarrollo del juego

- Hay que definir un predicado de aridad cero (**jugar**) que incorpore la preparación del juego y que, además, llame al predicado encargado de gestionar el desarrollo del juego.
- El sistema preguntará cada vez por una característica de los personajes que almacena en su Base de Conocimiento. Como respuesta, admitirá solamente dos posibilidades: Si / No. En función del tipo de respuesta, el programa seleccionará un subconjunto de candidatos que cumplan con dicha característica de modo que, en el siguiente turno, buscará el candidato solución en dicho subconjunto.
- Si el programa no adivina el personaje que está pensando el jugador, deberá incluirlo en la Base de Conocimiento.

- Se valorará positivamente cualquier mejora del juego sobre lo expuesto anteriormente.
- Algunas ideas son las siguientes:
 - Permitir otros tipos de respuesta como las del juego original: Sí, No, No lo sé, Probablemente o Probablemente no. En este caso se debe implementar una lógica que permita seleccionar de forma eficiente el subconjunto de candidatos en función de la respuesta.
 - Mejorar la interfaz de usuario con algún elemento de tipo 'ASCII art'.
 - Mostrar por pantalla el subconjunto de candidatos seleccionado en cada turno de juego.
 - Optimizar el mecanismo de búsqueda para que adivine el personaje con el mínimo número de preguntas posible.

Codificar enteros tomando como base una codificación de los naturales es relativamente sencillo. Para ello, dado un par (m, n) , con $m, n \in \mathbb{N}$, consideraremos que representa al entero $m - n^a$.

Al igual que ocurre con otras codificaciones de los enteros, los pares no representan a los enteros de manera única, p.e. (n, m) codifica el mismo entero que $(n + k, m + k)$.

^aEn la asignatura de Estructuras de Datos se empleó una codificación similar, en vez de escribir un entero como un par (n, m) se le representaba mediante una serie de llamadas a *suc* y *pred* (n a *suc* y m a *pred*)

En el fichero *enteros.rkt* del espacio virtual de la asignatura se puede ver una codificación de los enteros usando λ -Cálculo e implementada empleando *Scheme*. Dicha codificación incluye las siguientes operaciones:

- Suma y resta
- Multiplicación
- División euclídea
- Cálculo del máximo común divisor
- Relaciones de igualdad y de orden
- Reducción a representante canónico^a



^aTomaremos $(n, 0)$ ó $(0, n)$ como representantes canónicos.



- (1) El conjunto de los números módulo p , \mathbb{Z}_p , es el conjunto cociente de \mathbb{Z} mediante la relación de equivalencia

$$n R m \text{ si y sólo si } n - m \text{ es múltiplo de } p$$

Teniendo en cuenta lo anterior y la codificación, relaciones de orden y operaciones de enteros definidas en el fichero *enteros* mencionado, se pide codificar en λ -cálculo \mathbb{Z}_p . Dicha codificación debe incluir las siguientes operaciones:

- (a) Reducción a representante canónico. 
- (b) Aritmética: suma, producto.
- (c) Decisión sobre la inversibilidad y cálculo del inverso en el caso de que exista. 

Objetivos de la práctica (y 2)

(2) Codificada la aritmética modular, se pide codificar las matrices 2×2 en \mathbb{Z}_p con p primo. Esta codificación debe incluir las siguientes operaciones:

- (a) Suma y producto.
- (b) Determinante.
- (c) Decisión sobre inversibilidad y cálculo de inversa y del rango.
- (d) Cálculo de potencias (naturales) de matrices. Este cálculo se tiene que hacer usando el **algoritmo binario** para el cálculo de potencias, también conocido como exponenciación binaria.



Las necesidades de Scheme para esta práctica quedan cubiertas, salvo en el caso de la recursión, con la siguiente observación: el término del λ -Cálculo $\lambda x.M$ se codifica en Scheme mediante `(lambda (x) M)`. En el caso de que necesitáramos dar un nombre a un término para su posterior reutilización, la forma de hacerlo sería la siguiente `(define termino (lambda (x) M))`.

Por ejemplo, siguiendo lo visto en clase uno puede definir:

```
(define true (lambda (x y) x))  
(define false (lambda (x y) y))  
(define if (lambda (p x y) (p x y)))  
.  
.  
.
```

Dos observaciones sobre cómo simular la recursividad en el λ -cálculo con Scheme:

- El combinador de punto fijo **Y** ha de definirse aplicando una η -expansión (líneas 23 a 29 del fichero *enteros.rkt*).
- Aún definiendo así **Y**, la recursividad no funciona tal cual se ha visto en clase. Ejemplos de cómo se puede simular la recursión se encuentran en el fichero *enteros.rkt*. Por ejemplo, en la definición del resto de la división euclídea (líneas 145-168)^a.

^aEsta simulación de la recursividad se ha extraído de

http://www.shlomifish.org/lecture/Lambda-Calculus/slides/lc_church_div.scm.html