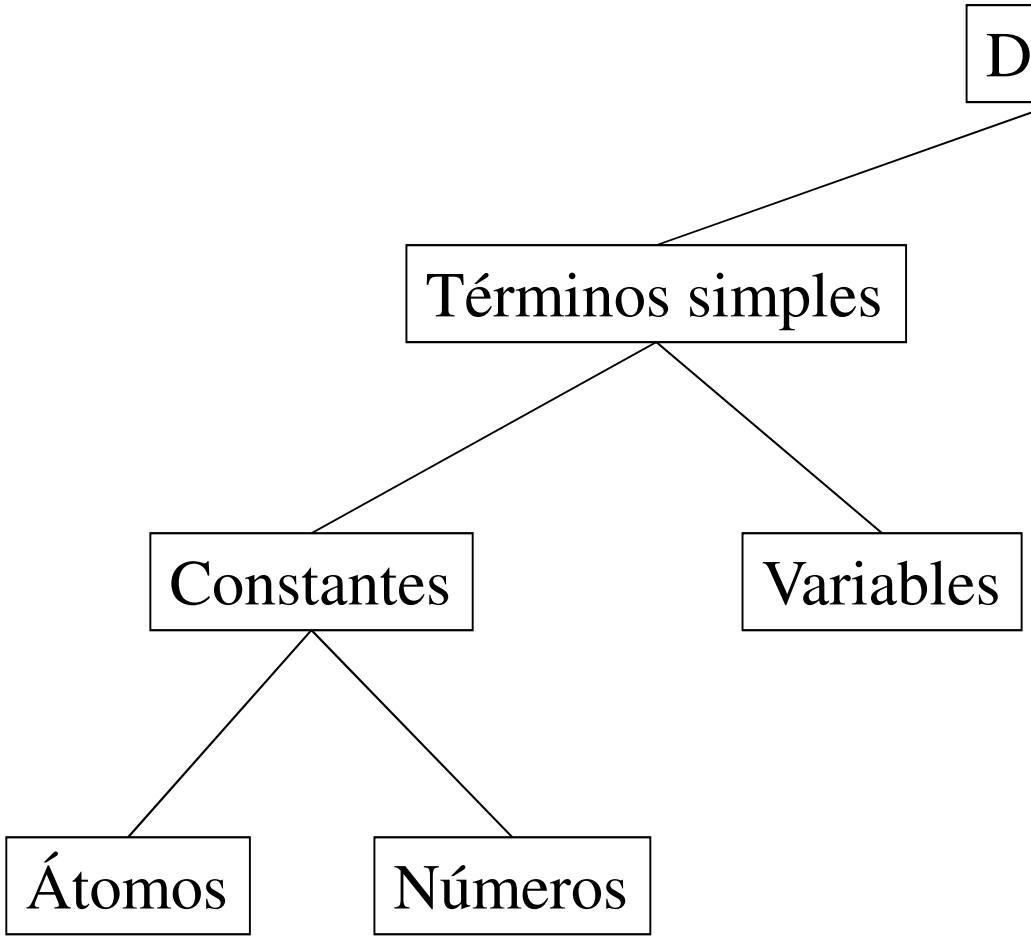
PROLOG - II

Estructuras de Datos

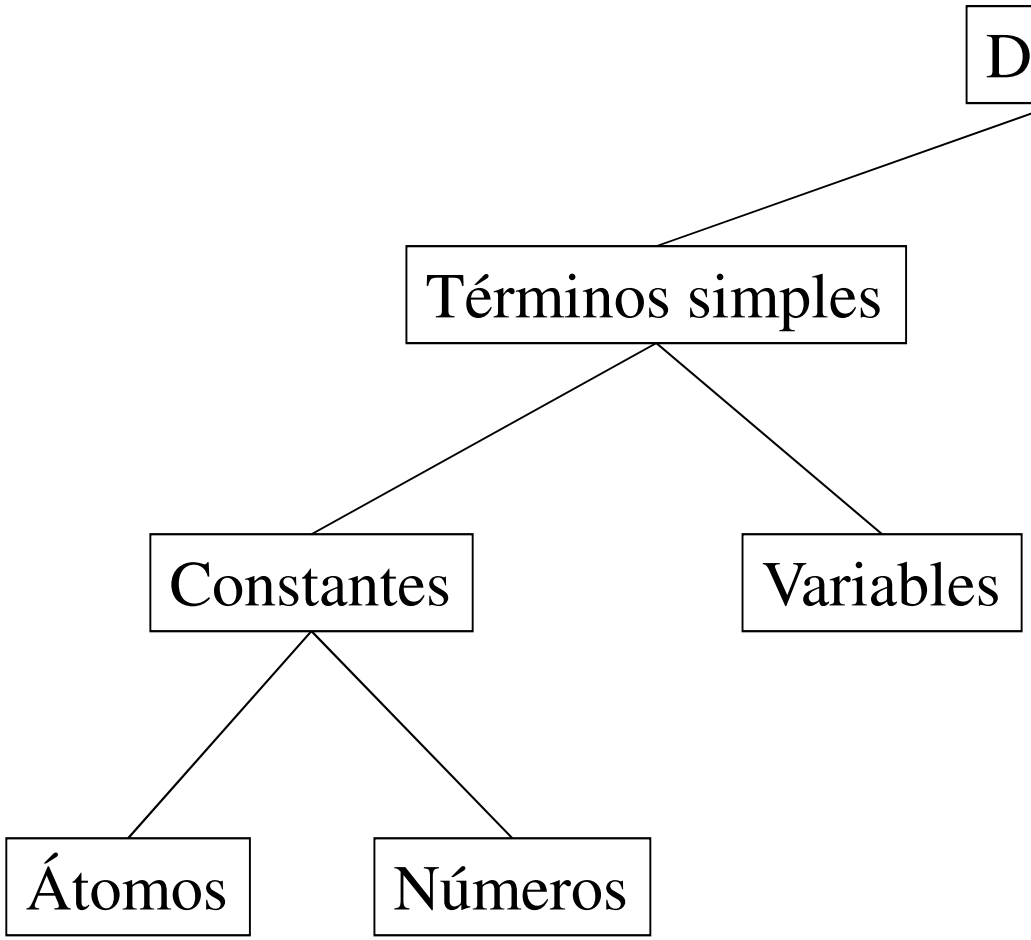


Una **estructura** es un único objeto que se compone de una colección de otros objetos, llamados componentes, lo que facilita su tratamiento. Una estructura se escribe en Prolog especificando su *nombre*, y sus *componentes* (*argumentos*). Las componentes están encerradas entre paréntesis y separadas por comas; el nombre se escribe justo antes de abrir el paréntesis:

nombre (comp1, comp2, ..., compn)

La sintaxis para las estructuras es la misma que para los hechos.

Estructuras de Datos



Objetos que contienen datos como componentes.

- **Functores**
- **Listas**

Estructuras de Datos

Functores

Sintaxis: *functor(comp#1, comp#2,... ...,comp#n).*

- En general, *functor* es un nombre que designa:
  - un hecho,
  - una relación,
  - una función
- A su vez, *comp#i*, puede ser un functor

Ejemplo:

curso-27(titulacion("Grado en Informática"), materia(sistemas_inteligentes)).

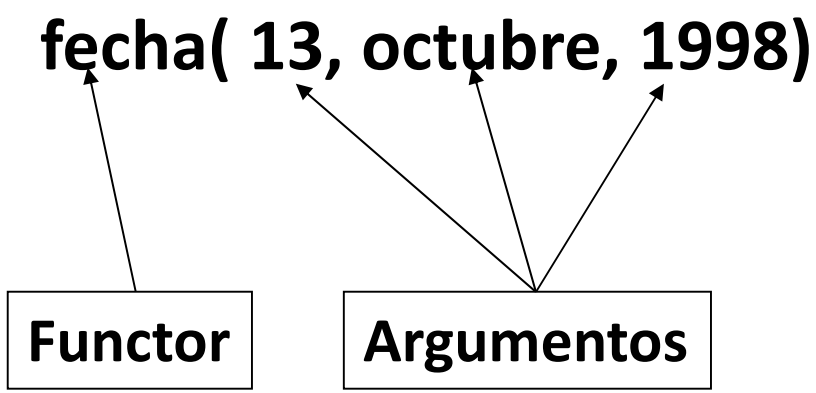
Incluso las cláusulas se representan como functores:

?- clause(X, Y). / Se satisface con clausulas de cabeza X y cuerpo Y */*

Estructuras de Datos

Functores

Ejemplo: una estructura para representar la fecha.



Un aspecto notable de Prolog es que las relaciones pueden establecerse no sólo entre átomos sino que también entre términos estructurados. Un *término* es ya sea un átomo (identificador, número, string, variable) o un nombre de término (o *functor*) asociado a una lista de argumentos $t(t_1,t_2,...,t_n)$.

Estructuras de Datos

Listas

- Una lista es una secuencia ordenada de elementos que puede tener cualquier longitud.
- Los elementos de una lista pueden ser cualquier término (constantes, variables, estructuras) u otras listas.
- Las listas pueden representarse como un tipo especial de árbol. Una lista puede definirse recursivamente como:
  - una lista vacía [], sin elementos, o
  - una estructura con dos componentes:
    - cabeza: primer argumento
    - cola: segundo argumento, es decir, el resto de la lista.
- El final de una lista se suele representar como una cola que contiene la lista vacía.
- La cabeza y la cola de una lista son componentes de una estructura cuyo nombre es ".".

Estructuras de Datos

Las listas se manipulan dividiéndolas en cabeza y cola

Lista	[X|_]	[_|X]
[a,b,c]	a	[b,c]
[[a,b],c]	[a,b]	[c]
[a,[b,c]]	a	[[b,c]]
[a,[b,c],d]	a	[[b,c],d]
[X+Y,Z]	X+Y	[Z]
[]	ninguna	Ninguna
[a]	a	[]

?- [X|Y] = [maulla, [el, gato]].

X = maulla
Y = [[el, gato]]

Yes

Estructuras de Datos

Operaciones con listas

Primer elemento de una lista

```
primero([X|_],X).
```

```
?- primero([a,b,c],X).
X = a
```

Estructuras de Datos

Operaciones con listas

Resto de una lista

```
resto([_|L],L).
```

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

Estructuras de Datos

Operaciones con listas

Construcción de una lista

```
cons(X,L,[X|L]).
```

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

Estructuras de Datos

Operaciones con listas

Pertenencia a una lista

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

```
?- pertenece(b,[a,b,c]).
Yes
?- pertenece(d,[a,b,c]).
No
```

Estructuras de Datos

Operaciones con listas

Concatenación de listas

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

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

Estructuras de Datos

Otras operaciones con listas de interés

- Lista inversa
- Último elemento
- Penúltimo elemento
- Selección de un elemento
- Inserción de un elemento
- Sublista
- Permutación
- Rotación
- Longitud de una lista

Estructuras de Datos

Recursividad

antepasado(X,Y):-progenitor(X,Y).
antepasado(X,Y):-progenitor(X,Z),
antepasado(Z,Y).

- En general, en una definición recursiva, es necesario considerar 2 casos:
- **Caso básico**: Momento en que se detiene la computación
- **Caso Recursivo**: Suponiendo que ya se ha solucionado un caso más simple, cómo descomponer el caso actual hasta llegar al caso simple.
- Tanto el caso básico como el caso recursivo no tienen porqué ser únicos (puede haber varios casos básicos y varios casos recursivos)

Estructuras de Datos

Unificación

- La unificación («matching») aplicada, junto con la regla de resolución, es lo que nos permite obtener respuestas a las preguntas formuladas a un programa lógico.
- La unificación constituye uno de los mecanismos esenciales de Prolog, y consiste en buscar instancias comunes a dos átomos, uno de los cuales está en la cabeza de una cláusula y el otro en el cuerpo de otra cláusula.
- Prolog intentará hacerlos coincidir (unificarlos) mediante las siguientes reglas:
  - Una variable puede instanciarse con cualquier tipo de término, y naturalmente con otra variable.
  - Los números y los átomos sólo serán iguales a sí mismos. Evidentemente, también se pueden instanciar con una variable.
  - Dos estructuras son iguales si tienen el mismo nombre y el mismo número de argumentos, y todos y cada uno de estos argumentos son unificables.
- El predicado de igualdad (=) es un operador infijo que intentará unificar ambas expresiones. Un objetivo con el predicado no igual (\=) se satisface si el = fracasa, y fracasa si el = se satisface.

Estructuras de Datos

Unificación

```
?- X=juan, X=Y.
X=juan, Y=juan

?- X=Y, X=juan.
X=juan, Y=juan

?- juan=juan.
yes

?- juan=pepe.
no

?- 1024=1024.
yes

?-
amigo(pepe,juan)=amigo(pepe,X).
X=juan
```

```
?- amigo(pepe,juan)=Y.
Y=amigo(pepe,juan)

?- 9 \= 8.
yes

?- letra(C)=palabra(C).
no

?- 'juan'=juan.
yes

?- "juan"=juan.
no

?- f(X,Y)=f(A).
no
```

Estructuras de Datos

Reevaluación o backtracking

- Consiste en volver a mirar lo que se ha hecho e intentar resatisfacer los objetivos buscando una forma alternativa de hacerlo.
- Si se quieren obtener más de una respuesta a una pregunta dada, puede iniciarse la reevaluación pulsando la tecla «y» cuando Prolog acaba de dar una solución y pregunta «More (y/n) ?», con lo que se pone en marcha el proceso de generación de soluciones múltiples.

Estructuras de Datos

Corte

- El **corte** permite decirle a Prolog cuales son las opciones previas que no hace falta que vuelva a considerar en un posible proceso de reevaluación.
- Es un mecanismo muy delicado, y que puede marcar la diferencia entre un programa que funcione y uno que no funcione. Su utilidad vienen dada por:
  - *Optimización del tiempo de ejecución*: no malgastando tiempo intentando satisfacer objetivos que de antemano sabemos que nunca contribuirán a una solución.
  - *Optimización de memoria*: al no tener que registrar puntos de reevaluación para un examen posterior.
- El corte se representa por el objetivo "!" que se satisface inmediatamente y no puede resatisfacerse de nuevo.

Estructuras de Datos

Predicados de control

Existen una serie de predicados predefinidos que ayudan cuando queremos utilizar estructuras de control.

- fail — Siempre falla.
- true — Siempre tiene éxito.
- repeat — Permite simular bucles junto con la combinación corte-fail.
- +Condicion -> +Accion — Permite simular la estructura condicional: *Si-Entonces* y *Si-Entonces-Sino*.

Estructuras de Datos

Otros predicados

`var(X)`	Devuelve éxito si X no está instanciada.
`nonvar(X)`	Al revés.
`atom(X)`	Devuelve éxito si X es un "átomo" en el sentido de Prolog (constantes y cadenas de caracteres son "átomos"; variables, funciones y números no).
`integer(X)`	Devuelve éxito si X es un entero.
`atomic(X)`	Devuelve éxito si X es un "átomo" ó un entero.
`write(X)`	Escribe en el periférico por defecto el contenido de la variable X.
`read(X)`	Lee un valor del periférico por defecto y lo almacena en la variable X.
`nl`	Escribe un retorno de carro.
`==`	Compara si dos "átomos" son iguales.
`\==`	Compara si dos "átomos" son distintos.

Estructuras de Datos

Predicados Extra-lógicos: E/S

Predicado	*Se cumple cuando:*	*Efecto lateral*	Ejemplos Sencillos

write(X)	Siempre	Escribe el valor del término *X*	`| ?- write(f(x,3+4)).`
			`f(x,3+4).`
			`yes`

read(X)	Si es posible unificar X con el valor leído
	
	✓ *No se hace backtracking. Analiza los caracteres de entrada hasta encontrar un término y, si unifica, se cumple, sino, falla.*
		Lee un término *Prolog*
		
		✓ *Los términos Prolog deben acabar en punto*
			`| ?- read(X).`
			`|:<usuario> f(x,3+4).`
			`X = f(x,3 + 4)`
			
			`| ?- read(X), X = 2.`
			`|:<usuario> 3.`
			`no`

display(X)	Siempre	Escribe el valor del término *X* en notación functor	`|?- display(f(x,3+4)).`
			`f(x,(+)(3,4))`
			`yes`

nl	Siempre	Escribe un salto de línea	

Estructuras de Datos

Predicados Extra-lógicos: E/S

`saludo:-write('Tu nombre?'),`
`  read(N),`
`  write('Hola '), write(N).`
 	
`| ?- saludo.`
`Tu nombre?`
`|:<usuario> juan.`
`Hola juan`
`yes`
	saludo: Pide al usuario un nombre (término *Prolog*), lo lee y lo muestra por pantalla.

Estructuras de Datos

Predicados Extra-lógicos: E/S

Predicado	*Se cumple cuando:*	*Efecto lateral*	*Standard*
get0(X)	Si es posible unificar el caracter leido con *X*	*Lee un caracter del teclado*	`get_char`
get(X)	Si es posible unificar el caracter leido con *X*	*Lee el siguiente caracter distinto de blanco.*	
put(X)	Siempre	Escribe el caracter *X*	`put_char`

Estructuras de Datos

Predicados Extra-lógicos: E/S

Predicado	*Se cumple cuando:*	*Efecto lateral*	*Standard*
tell(F)	Si no hay errores de apertura	Abre F como *stream de* salida actual	`open`
see(F)	Si no hay errores de apertura	Abre *F* como *stream actual de* entrada.	`open`
told	Siempre	Cierra *stream* de salida actual	`close`
seen	Siempre	Cierra *stream* de entrada actual	`close`
telling(F)	Unifica *F* con el nombre del *stream* de salida actual	Ninguno	
seeing(F)	Unifica *F* con el nombre del *stream* de salida actual	Ninguno	

Estructuras de Datos

Predicados Extra-lógicos: E/S

```
verFich:-write('Nombre fichero?'),
            read(N),
            seeing(Antes),
            see(N),
            bucle,
            seen,
            see(Antes).

bucle:-repeat,
        get0(C),
        minMay(C,Cm),
        put(Cm),
        at_end_of_file,
        !.


minMay(C,Cm):-minuscula(C),!,
              Cm is C - 0'a + 0'A.
minMay(C,C).

minuscula(C):- C >= 0'a, C =< 0'z.
```

verFich:-Pregunta un nombre de fichero al usuario, lo abre y visualiza su contenido en mayúsculas.

Estructuras de Datos

Predicados Extra-lógicos: acceso a BD

Los sistemas Prolog ofrecen la posibilidad de modificar en tiempo de ejecución el contenido de la base de conocimiento

Predicado	*Se cumple cuando:*	*Efecto lateral*
asserta(T)	Siempre	Añade al principio de la base de conocimiento el término T
assertz(T)	Siempre	Añade al final de la base de conocimiento el término *T*.
retract(T)	Si T unifica con el término eliminado.	Elimina de la base de conocimiento el término *T*

Estructuras de Datos

Predicados Extra-lógicos: Directivas

Indican al sistema tareas a realizar u opciones de compilación.

`:-ensure_loaded(F).`	Indica al sistema que cargue el fichero *F*
`:-multifile(P/N).`	Indica que el predicado P de aridad N puede definirse en varios ficheros. Por defecto, las definiciones deben estar en un solo fichero.
`:-dynamic(P/N).`	Indica al sistema que la definición del predicado P de aridad N puede modificarse de forma dinámica
`:-initialization(C).`	Declara que el objetivo C debe ser ejecutado tras cargar el fichero.
`:-op(Prioridad,Asociatividad,Atomo).`	Define *Atomo* como un operador con la *Prioridad* y *Asociatividad* dadas

Estructuras de Datos

Modificación de la Base de Conocimientos: adición

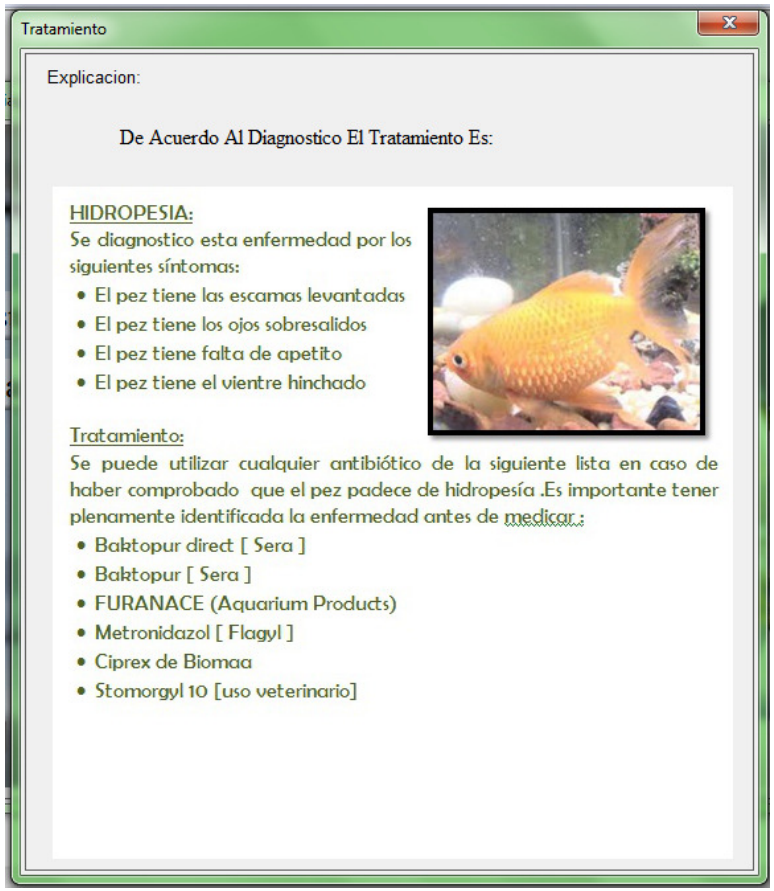
Si queremos que Prolog lea nuevas cláusulas de un fichero que hemos preparado previamente, podemos utilizar los predicados **consult** y **reconsult**. Esto será conveniente cuando tengamos que trabajar con programas de tamaño considerable y no queramos teclear cada vez todas las cláusulas.

Modificación de la Base de Conocimientos: manipulación

Existen predicados predefinidos que podemos utilizar para ver (**listing**), obtener (**clause**), añadir (**assert**) o quitar (**retract** y **abolish**) cláusulas de nuestra base de conocimientos.

Ejercicio

Analizar el código y hacer funcionar el programa Prolog que permite diagnosticar la enfermedad que se refleja en la ficha y proponer un tratamiento, y cuyo código fuente está en el fichero **sistema-experto-diagnosticador-master.zip**



https://github.com/jofese/sistema-experto-diagnosticador

Fuentes y bibliografía

- www.swi-prolog.org/
- ftp://www.cc.uah.es/pub/Alumnos/G_Ing_Informatica/Conocimiento_y_Razonamiento_Automatizado/
- Ejercicios de programación declarativa con Prolog. José A. Alonso Jiménez. Grupo de Lógica Computacional. Dpto. de Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla.
- Prácticas de Lógica. Prolog. Faraón Llorens Largo, Mª Jesús Castel de Haro. Universidad de Alicante.
- Programación Práctica en Prolog. Jose E. Labra G.. Área de Lenguajes y Sistemas Informáticos. Departamento de Informática. Universidad de Oviedo.