



## **UNIDAD 1 - PROLOG**

### **1.1 INTRODUCCIÓN.**

**Prolog** representa el lenguaje principal en la categoría de Programación Lógica. A diferencia de otros lenguajes, Prolog no es un lenguaje de programación para usos generales, sino que está orientado a resolver problemas usando el cálculo de predicados. Las aplicaciones de Prolog provienen en general de dos dominios distintos:

**Preguntas a bases de datos:** Las bases de datos modernas indican típicamente relaciones entre los elementos que están guardados en la base de datos. Pero no todas estas relaciones se pueden indicar. Por ejemplo, en una línea aérea puede haber entradas que indiquen números de vuelo, ubicación, hora de salida, ubicación y hora de llegada. Sin embargo, si un individuo necesita hacer un viaje que requiera cambiar de avión en algún punto intermedio, es probable que la relación no esté especificada en forma explícita.

**Pruebas matemáticas.** También se pueden especificar las relaciones entre objetos matemáticos a través de una serie de reglas y sería deseable un mecanismo para generar pruebas de teoremas a partir de este modelo. Aunque la búsqueda ascendente inherente en LISP da cabida a la construcción de sistemas generadores de pruebas, sería eficaz un lenguaje orientado a mayor grado hacia la prueba de propiedades de relaciones.

Estas dos aplicaciones son similares y se pueden resolver usando Prolog. El objetivo para Prolog era proporcionar las especificaciones de una solución y permitir que la computadora dedujera la secuencia de ejecución para esa solución, en vez de especificar un algoritmo para la solución de un problema, como es el caso normal de casi todos los lenguajes.

### **1.2 HISTORIA.**

El desarrollo de Prolog se inició en 1970 con Alain Coulmeauer y Philippe Roussel, quienes estaban interesados en desarrollar un lenguaje para hacer deducciones a partir de texto. El nombre corresponde a "PROgramming in Logic" (Programación en lógica). Prolog fue desarrollado en Marsella, Francia, en 1972. El principio de resolución de Kowalski, de la Universidad de Edimburgo pareció un modelo apropiado para desarrollar sobre el mecanismo de inferencia. Con la limitación de la resolución de cláusulas de Horn, la unificación condujo a un sistema eficaz donde el no determinismo inherente de la resolución se manejó por medio de un proceso de exploración a la inversa, el cual se podía implementar con facilidad.

La primera implementación de Prolog se completó en 1972 usando el compilador de ALGOL W de Wirth y los aspectos básicos del lenguaje actual se concluyeron en 1973. El uso de Prolog se extendió gradualmente entre quienes



se dedicaban a la programación lógica principalmente por contacto personal y no a través de una comercialización del producto. Existen varias versiones diferentes, aunque bastante similares. Aunque no hay un estándar del Prolog, la versión desarrollada en la Universidad de Edimburgo ha llegado a ser utilizada ampliamente. El uso de este lenguaje no se extendió sino hasta los años ochenta. La falta de desarrollos de aplicaciones eficaces de Prolog inhibió su difusión.

### 1.3 BREVE PERSPECTIVA DEL LENGUAJE.

Un programa en Prolog se compone de una serie de hechos, relaciones concretas entre objetos de datos (hechos) y un conjunto de reglas, es decir, un patrón de relaciones entre los objetos de la base de datos. Estos hechos y reglas se introducen en la base de datos a través de una operación de consulta.

Un programa se ejecuta cuando el usuario introduce una pregunta un conjunto de términos que deben ser todos ciertos. Los hechos y las reglas de la base de datos se usan para determinar cuáles sustituciones de variables de la pregunta (llamadas unificación) son congruentes con la información de la base de datos.

Como intérprete, Prolog solicita entradas al usuario. El usuario digita una pregunta o un nombre de función. La verdad 'Yes' o falsedad 'No' de esa pregunta se imprime, así como una asignación a las variables de la pregunta que hacen cierta la pregunta, es decir que unifican la pregunta. Si se introduce un ';', entonces se imprime el próximo conjunto de valores que unifican la pregunta, hasta que no son posibles más sustituciones, momento en el que Prolog imprime 'No' y aguarda una nueva pregunta. Un cambio de renglón se interpreta como terminación de la búsqueda de soluciones adicionales.

La ejecución de Prolog, aunque se basa en la especificación de predicados, opera en forma muy parecida a un lenguaje aplicativo como LISP o ML. El desarrollo de las reglas en Prolog requiere el mismo "pensamiento recursivo" que se necesita para desarrollar programas en esos otros lenguajes aplicativos.

Prolog tiene una sintaxis y semántica simples. Puesto que busca relaciones entre una serie de objetos, la variable y la lista son las estructuras de datos básicas que se usan. Una regla se comporta en forma muy parecida a un procedimiento, excepto que el concepto de unificación es más complejo que el proceso relativamente sencillo de sustitución de parámetros por expresiones.

### 1.4 EVALUACIÓN DEL LENGUAJE.

Prolog ha alcanzado una medida razonable de éxito como lenguaje para resolver problemas de relaciones, como el procesamiento de consultas a bases de datos. Ha alcanzado un éxito limitado en otros dominios.



El objetivo original de poder especificar un programa sin proporcionar sus detalles algorítmicos no se ha alcanzado realmente. Los programas en Prolog se leen secuencialmente, aunque el desarrollo de reglas sigue un estilo aplicativo. Se usan cortes con frecuencia para limitar el espacio de búsqueda para una regla, con el efecto de hacer el programa tan lineal en cuanto a ejecución como un programa típico de C o Pascal. Aunque las reglas se suelen expresar en forma aplicativo, el lado derecho de cada regla se procesa de manera secuencial. Esto sigue un patrón muy similar al de los programas en ML.



## **UNIDAD 2 - CONCEPTOS BÁSICOS**

### **2.1 DATOS.**

El sistema Prolog reconoce el tipo de un objeto en el programa por medio de su forma sintáctica. Esto es posible porque la sintaxis de Prolog especifica diferentes formas para cada tipo de datos. La forma de distinguir entre átomos y variables es que las variables empiezan con letras mayúsculas, mientras que los átomos con minúsculas. No existe información adicional para comunicarle a Prolog el orden para reconocer el tipo de un objeto.

Los átomos pueden construirse de 3 formas:

Cadenas de letras, dígitos y el caracter '\_', empezando con una letra minúscula:

```
ana
pedro
x25
x_25
procedimiento_uno
categoria_x
```

Cadenas de caracteres especiales:

```
<---->
=====>
....
```

Cadenas de caracteres encerradas en apóstrofes. Esto se utiliza cuando se quiere, por ejemplo, tener un átomo que empiece con una letra mayúscula.

```
'Tom'
'Polo_Norte'
```

**Nota:** en algunos compiladores, en vez de utilizar apóstrofes se utilizan comillas " " como en el caso de Prolog.

- **NÚMEROS:**

Los números usados en Prolog incluyen números enteros y números reales. La sintaxis de los enteros es simple:

```
1
1313
0
-97
```

El tratamiento de números reales depende de la implementación de Prolog. Asumiendo una sintaxis simple:

```
3.14
-0.0035
```



100.2

Los números reales no son muy utilizados en programas de Prolog. La razón de esto es que Prolog es principalmente utilizado como un lenguaje simbólico, no de computación numérica. En la computación simbólica, los enteros son utilizados, por ejemplo, para contar el número de elementos en una lista; por lo que los números reales son poco utilizados.

## 2.2 VARIABLES.

Las variables son cadenas de letras, dígitos y el signo '\_'. Estas empiezan con una letra mayúscula o el símbolo '\_':

```
X
Objeto2
_23
Resultado
_x23
Lista_Participantes
```

Cuando una variable aparece en una cláusula sola, no se necesita inventar un nombre para ella. Se usa llamarla variable "anónima", cuando es escrito únicamente el signo '\_'. Por ejemplo, consideremos la siguiente regla:

```
hijo(X) :- padre (X,Y) .
```

La regla dice: para toda X, X tiene un hijo si X es el padre de alguna Y. Nosotros estamos definiendo la propiedad hijo el cual, no depende del nombre del hijo. Entonces, aquí hay un lugar en donde podemos usar una variable anónima. Si rescribimos la cláusula:

```
hijo(X) :- padre(X,_) .
```

## 2.3 OPERADORES.

Los operadores de Prolog están divididos en 2 clases: *aritméticos* y *relacionales*.

Los operadores aritméticos incluyen los símbolos para *suma*, *resta*, *multiplicación* y *división*. En Prolog, si dos enteros son sumados, restados o multiplicados, el resultado será un entero. Siempre que uno de los operandos, en cualquiera de estas operaciones sea un número real, el resultado siempre será un real. El resultado de una división, siempre que los operandos sean enteros o reales, será siempre real.

### - ARITMÉTICOS:

Todas las versiones de Prolog soportan los siguientes operadores aritméticos, listados en el orden de prioridad de ejecución.



+ → SUMA  
- → RESTA  
\* → MULTIPLICACIÓN  
/ → DIVISIÓN  
// → DIVISION ENTERA  
MOD → RESIDUO  
^ → POTENCIA

Los paréntesis se pueden utilizar para dar preferencia de ejecución en una expresión compuesta. Prolog utiliza la precedencia de operadores con la regla de la mano izquierda.

A continuación se muestran algunas operaciones más complejas que PROLOG provee:

OPERACIÓN	DESCRIPCIÓN
Sqrt(X)	Calcula la raíz cuadrada de la variable
log(X)	Calcula el logaritmo de X
ln(X)	Calcula el logaritmo natural de X
abs(B)	Regresa el valor absoluto de B
sin(T)	Seno de T
cos(A)	Coseno de A
tan(C)	Tangente de C

#### - RELACIONALES:

Prolog soporta los siguientes operadores relacionales:

= → IGUAL QUE  
> → MAYOR QUE  
>= → MAYOR O IGUAL QUE  
<= → MENOR O IGUAL QUE  
<> → DESIGUAL QUE  
/= → DIFERENTE QUE  
is → EVALUADOR DE EXPRESIÓN  
seed → GENERADOR DE NÚMEROS ALEATORIOS

Cuando dos objetos que son símbolos o cadenas de caracteres son comparados, los caracteres son convertidos a su equivalente ASCII. El valor de cada carácter es examinado a partir del operador relacional, de izquierda a derecha.



## 2.4 PREDICADOS.

Un predicado es la relación directa con una expresión. Cada predicado usado en una cláusula de Prolog debe ser declarado, basado en la declaración de los tipos de dominios para cada uno de los nombres de los objetos.

```
paciente(nombre, edad, peso, presion_sanguinea).  
mercado(encargado, vendedor).
```

## 2.5 ESTRUCTURAS.

Los objetos estructurados (o simplemente estructuras) son objetos que tienen varios componentes. Los componentes pueden ser a su vez estructuras. Por ejemplo, la fecha, puede ser vista como una estructura con 3 componentes: día, mes y año. Las estructuras son tratadas en el programa como objetos simples, aunque estén formadas por muchos componentes. El orden en que se combinan los componentes dentro de un objeto simple, es la forma en que escogemos una functor. Un functor conveniente para nuestro ejemplo es fecha. Entonces la fecha 1o. de Enero 1975 puede escribirse:

```
fecha(1, enero, 1975)
```

Todos los componentes en este ejemplo son constantes (2 enteros y un átomo). En la siguiente figura vemos como se representa en forma de árbol, y a su vez cómo está escrito en Prolog:

Ahora bien, cualquier día de enero de 1975 puede representarse mediante la estructura:

```
fecha(Día, enero, 1975)
```

Donde Día es una variable que puede ser instanciada por cualquier objeto en cualquier momento de la ejecución del programa. Sintácticamente, todos los objetos de datos en Prolog son términos. Por ejemplo, enero y *date(1,enero,1975)* son términos.

## 2.6 CONSULTAS.

Para plantear una consulta en Prolog, el usuario simplemente prueba ésta, para ver si ésta es verdadera. Si la prueba es positiva, Prolog contesta: YES, de lo contrario responde NO, o también se usa TRUE o FALSE, dependiendo del programa que se use para la programación del lenguaje.

```
paciente ("ana", femenino).  
yes
```



## 2.7 COMENTARIOS.

Cuando se quiere hacer un comentario que cuenta con más de una línea, se hace de la siguiente manera:

```
/*  
Comentario x  
Comentario y  
*/
```

Si el comentario es de una sola línea simplemente se antecede el signo de % al comentario. Es importante saber que los comentarios no tienen efecto en la ejecución del programa.

```
% Comentario
```

## 2.8 CLÁUSULAS Y RELACIONES.

La programación lógica está basada en la noción de *relación*. Debido a que en la relación es un concepto más general de una aplicación. La programación lógica es potencialmente de alto nivel.

Considerar 2 conjuntos de valor S y T, R es la Relación entre S y T, para toda X que pertenece a S y Y que pertenece a T y  $R(X,Y)$  es verdadero o falso.

Dado a, determinar el valor  $m(a)$ . En la programación Lógica se implementa las relaciones. Sea R una relación:

```
Dado a y b, determinar cuando  $R(a,b)$  es verdadero.  
Dado a, encontrar todos los Y/ $R(a,y)$  es verdadero.  
Dado b, encontrar todos los X/ $R(x,b)$  es verdadero.  
Encontrar X y Y/ $R(x,y)$  es verdadero.
```

### - TIPOS DE RELACIONES:

```
Si  $R(x)$  entonces relación unitaria.  
Si  $R(x,y)$  entonces relación binaria.  
Si  $R(x,y,z)$  entonces relación ternaria.
```

Un programa en PROLOG define una colección de relaciones. Cada relación es definida por una o más *cláusulas*.

### - INTERPRETACIÓN DE UNA CLÁUSULA EN PROLOG.

```
A:- A1,...,An.  
:- → Es equivalente "Si" o "si".  
, → Es equivalente "AND".  
; → Es equivalente a "OR"
```





## 2.9 EJECUCIÓN DE UN PROGRAMA.

- *EN LINUX*

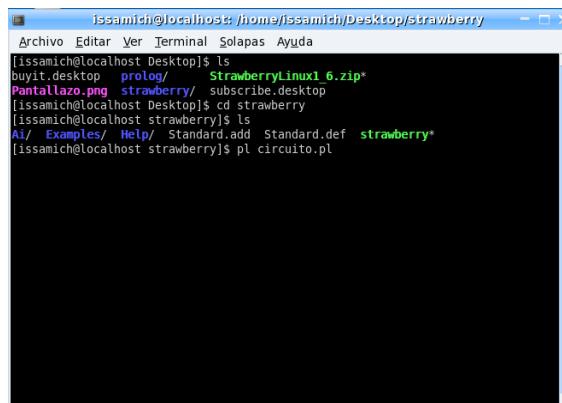
Instalar SWI-Prolog en linux

*pasos para la instalación de la ultima versión.*

Descarga la ultima versión desde la pagina del sitio oficial

Desde la terminal de Linux:

```
wget http://gollem.science.uva.nl/cgi-bin/nph-download/SWI-Prolog/binaries/pl-5.6.25-268.i586.rpm
```



```
Issamich@localhost: /home/issamich/Desktop/strawberry
Archivo  Editar  Ver  Terminal  Solapas  Ayuda
[issamich@localhost Desktop]$ ls
buyit.desktop  prolog/  StrawberryLinux1.6.zip*
Pantallazo.png strawberry/ subscribe.desktop
[issamich@localhost Desktop]$ cd strawberry
[issamich@localhost strawberry]$ ls
Ai/  Examples/  Help/  Standard.add  Standard.def  strawberry*
[issamich@localhost strawberry]$ pl circuito.pl
```

Ejemplo informativo de la terminal.

Convierte de rpm a deb usando alien. Si no lo tienes instalado usa

```
apt-get install alien
```

```
alien -d pl-5.6.25-268.i586.rpm
```

Instala:

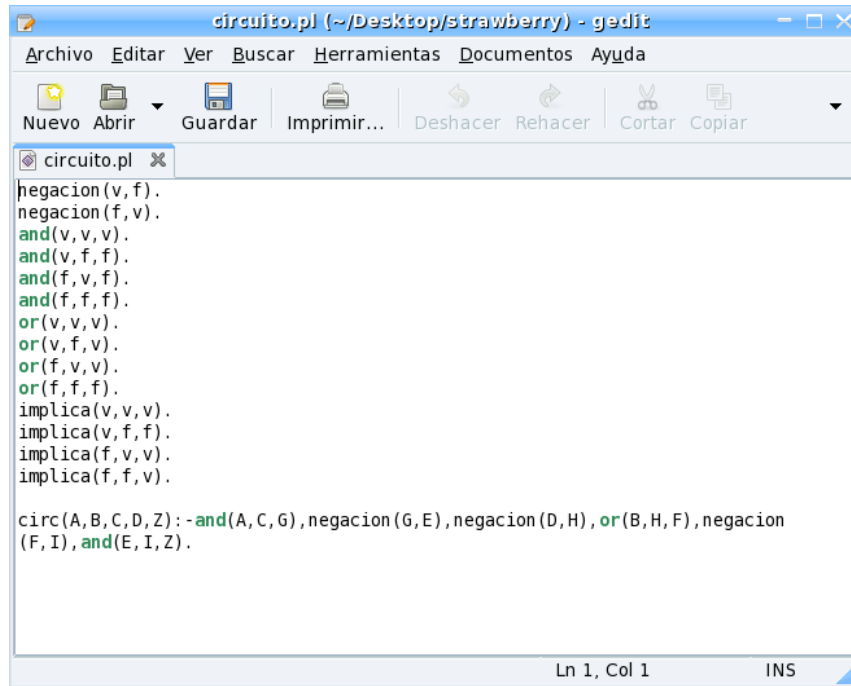
```
dpkg -i pl-5.6.25-268.i586.deb
```

Ejecuta:

```
pl
```

Una vez ejecutado esto desde la terminal puedes compilar los programas de prolog.

Para escribir los programas, solo abre cualquier editor de texto que tengas a la mano, escribe el código y guarda el archivo con terminación .pl.

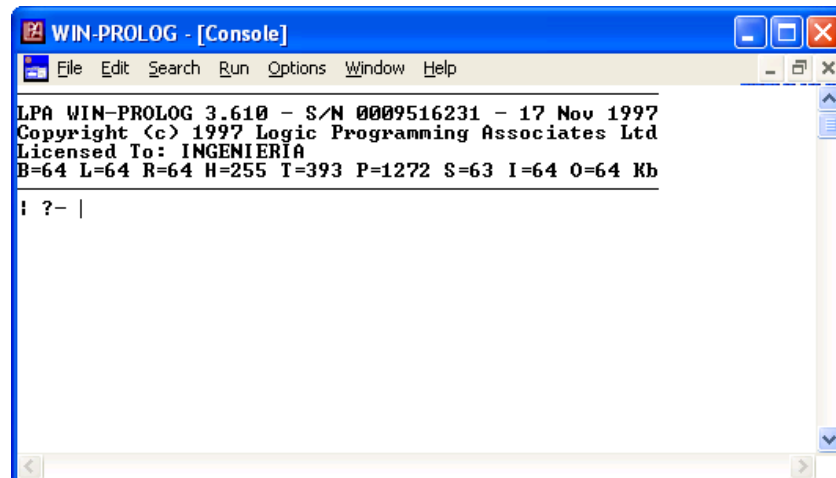


```
negacion(v,f).
negacion(f,v).
and(v,v,v).
and(v,f,f).
and(f,v,f).
and(f,f,f).
or(v,v,v).
or(v,f,v).
or(f,v,v).
or(f,f,f).
implica(v,v,v).
implica(v,f,f).
implica(f,v,v).
implica(f,f,v).

circ(A,B,C,D,Z):-and(A,C,G),negacion(G,E),negacion(D,H),or(B,H,F),negacion
(F,I),and(E,I,Z).
```

- *EN WINDOWS*

Al iniciar Prolog, aparece la consola en la cual se hacen las consultas



```
LPA WIN-PROLOG 3.610 - S/N 0009516231 - 17 Nov 1997
Copyright (c) 1997 Logic Programming Associates Ltd
Licensed To: INGENIERIA
B=64 L=64 R=64 H=255 I=393 P=1272 S=63 I=64 O=64 Kb

:-
```

En el menú de File/New se abre una nueva ventana para escribir código fuente.



```

LPA WIN-PROLOG 3.610 - S/N 0009516231 - 17 Nov 1997
Copyright (c) 1997 Logic Programming Associates Ltd
Licensed To: INGENIERIA
B-64 L-64 R-64 H-255 I-393 P-1272 S-63 I-64 O-64 Kb
: ?-

lista.pl
conc([],B,B).
conc([X:IR],B,[X:IZ]):-conc(R,B,Z).

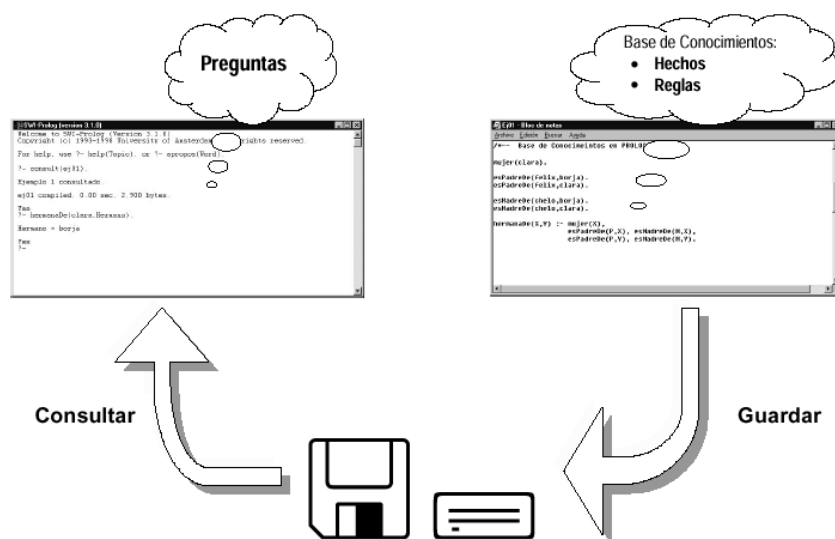
inv([],[]).
inv([X],[]).
inv([X,V],[]).
inv([X:IR],F):-inv(R,Z),conc(Z,[X],F).

repite(X,[],0).
repite(X,[X:IR],N):-repite(X,R,Z),N is Z+1.
repite(X,[Y:IR],N):-repite(X,R,N).

conjunto([],[],[]).
conjunto([X:IR],C,[X:ID]):-conjunto(R,C,D),repite(X,C,N),N>0,!.
conjunto([X:IR],[]):-conjunto(R,C,D),repite(X,C,0),!.

elemento(X,[X:IR]).
elemento(X,[Y:IR]):-elemento(X,R).
    
```

### - EJECUTAR UN PROGRAMA EN PROLOG.



1. Tener el programa de Prolog abierto.
2. En el código fuente se compila, haciendo clic en el menú compile.
3. En la consola se hace la consulta.

```

: ?- easter(2000,Y,D) .
Y=4,
D=23.
    
```

*Easter* es un programa que se encuentra en la carpeta de *Examples* que vienen en prolog.

*C:/Pro386w/examples/Easter.pl*



Si las premisas son contradictorias el argumento no es valido

$$P1 \wedge P2 \wedge \dots \wedge Pn \Rightarrow Q$$

$Q: \neg p1 \wedge \neg p2 \wedge \dots \wedge \neg Pn$  Cláusula = Clause

$\uparrow \uparrow$

Implicación      Premisa

$\therefore Q$  es una conclusión  $Q: \neg P1, P2, \dots Pn$ .



## **UNIDAD 3 – EJEMPLO INTRODUCTORIO**

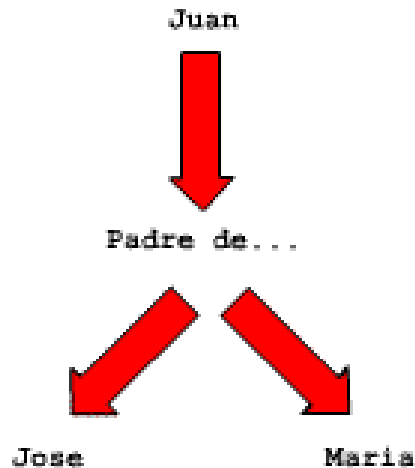
### **3.1 ÁRBOL GENEALÓGICO.**

*Buscar 01\_arbolgenealogico.pl en la carpeta de ejemplos*

El siguiente ejemplo es un sencillo programa en prolog de cómo usar la base de conocimientos y las consultas. Así también se explicará la forma en que prolog trabaja.

Primero tenemos un tipo de posibles consultas del programa una vez compilado.

```
padre(Z,K).           %Es verdadero si Z es padre de K.  
ancestro(X,Y).       %Es verdadero si X es ancestro de Y.
```



Base de conocimiento

```
juan.  
jose.  
padre(juan,jose).  
padre(juan,maria).
```

The screenshot shows a window titled 'Untitled' with a text area containing the following Prolog code: `juan.`, `jose.`, `padre(juan,jose).`, and `padre(juan,maria).`



Después de compilar, se hace la consulta en la ventana de consola.

```
Console
! ?- juan.
yes

! ?- jose.
yes

! ?- maria.
?
-----
! Error 20 : Predicate Not Defined
! Goal      : maria

Aborted
! ?- |
```

Para agregar elementos a un árbol genealógico, simplemente se agregan a la base de conocimientos y se hacen las consultas en la consola, como se muestra a continuación:

#### En la base de conocimientos:

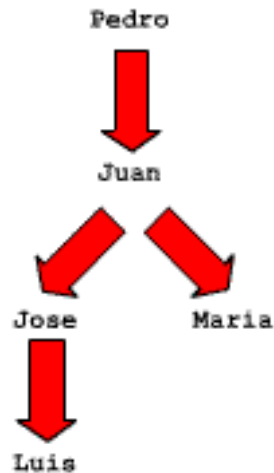
```
juan. %Esto es un comentario
jose.
padre(juan,jose).
padre(juan,maria).
padre(pedro,juan).
padre(jose,luís).
padre(X,Y).
ancestro(X,Y):-padre(X,Y).
ancestro(X,Y):-padre(X,Z),padre(Z,Y).
```

#### En la consola:

```
?-juan.
yes
?-padre(juan,jose).
yes
?-padre(X,jose).
X=juan
yes
?-ancestro(pedro,jose).
yes
```



**- ANÁLISIS DEL EJEMPLO ANTERIOR.**



**Consulta:**

```
:?-padre(pedro,jose) .  
yes
```

*Nivel 1.- X=pedro, Y=jose.*

Al recorrer la base de conocimientos pasara lo siguiente:

juan. → padre con juan, no coincide la sintaxis.  
jose. → padre con jose, no coincide la sintaxis.  
padre(pedro,juan). → padre con padre, si. pedro con pedro, si. jose con juan, no.  
padre(juan,jose). → padre con padre, si. pedro con juan, no.  
padre(juan,maria). → padre con padre, si. pedro con juan, no.  
padre(jose,luis). → padre con padre, si. pedro con jose, no.  
padre(X,Y). → padre con padre, si. pedro con X si. jose con Y, si.

Por lo tanto, como encontró una igualdad, la respuesta es si.

**Consulta:**

```
:?-ancestro(pedro,luis) .  
yes
```

*Nivel 1.- X=pedro, Y=luis.*

Al recorrer la base de conocimientos pasara lo siguiente:

juan. → ancestro con juan, no coincide la sintaxis.  
jose. → ancestro con jose, no coincide la sintaxis.  
padre(pedro,juan). → ancestro con padre, no.  
padre(juan,jose). → ancestro con padre, no.  
padre(juan,maria). → ancestro con padre, no.  
padre(jose,luis). → ancestro con padre, no.  
padre(X,Y). → ancestro con padre, no.  
ancestro(X,Y):-padre(X,Y). → ancestro con ancestro, si.



*Nivel 2.- Entra a la cláusula padre(X,Y), donde:*

*X=pedro, Y=luis.*

juan. → padre con juan, no coincide la sintaxis.  
jose. → padre con jose, no coincide la sintaxis.  
padre(pedro,juan). → padre con padre, si. pedro con pedro, si. luis con juan, no.  
padre(juan,jose). → padre con padre, si. pedro con juan, no.  
padre(juan,maria). → padre con padre, si. pedro con juan, no.  
padre(jose,luis). → padre con padre, si. pedro con jose, no.  
padre(X,Y). → padre con padre, si. pedro con X si. luis con Y, si.

Por lo tanto, como encontré una igualdad, la respuesta es si.

Ahora veremos el uso de las variables en las consultas.

### **Consulta:**

```
:-ancestro(X,luis).  
X=jose
```

Aquí estamos preguntando quien es el padre de luis.

*Nivel 1.- X=X, Y=luis.*

juan. → ancestro con juan, no.  
jose. → ancestro con jose, no.  
padre(pedro,juan). → ancestro con padre, no.  
padre(juan,jose). → ancestro con padre, no.  
padre(juan,maria). → ancestro con padre, no.  
padre(jose,luis). → ancestro con padre, no.  
padre(X,Y). → ancestro con padre, no.  
ancestro(X,Y):-padre(X,Y). → ancestro con ancestro, si.

*Nivel 2.- Entra a la cláusula padre(X,Y), donde:*

*X=X, Y=luis*

juan. → padre con juan, no.  
jose. → padre con jose, no.  
padre(pedro,juan). → padre con padre, si. X con pedro, si. luis con juan, no.  
padre(juan,jose). → padre con padre, si. X con juan si. luis con jose, no.  
padre(juan,maria). → padre con padre, si. X con juan, si. luis con maria, no.  
padre(jose,luis). → padre con padre, si. X con jose si. luis con luis, si.

Por lo tanto, la coincidencia fue total, así que se deduce que X=jose. El padre de Luis es José.





## UNIDAD 4 – PROGRAMACIÓN CON LISTAS

### 4.1 LISTAS.

Una *lista* en PROLOG es un conjunto de nombres de objetos, o átomos, separados por comas y encerrados en paréntesis cuadrados.

[member1,member2,...,memberN]

Los miembros de una lista deben ser nombres válidos de objetos, pero todos los miembros deben ser declaraciones de un mismo dominio.

Ejemplo:

```
["Maria", "Ana", "Juan"]  
[33, 25.51, 20, 10, 7]  
[a, b, c, d, e, f, g]
```

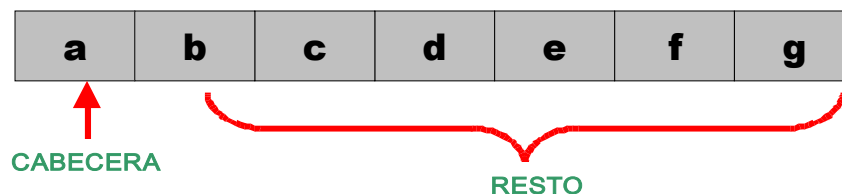
La lista puede ser vista, como un objeto don dos partes: [Cabeza|Cola]

La cabeza de la lista, conformada por el primer elemento.

La cola, es la parte restante de la lista.

En la última lista del ejemplo, la cabeza es a, mientras que la cola de la lista es la lista es [b,c,d,e,f,g,].

#### - PARTES DE UNA LISTA



### 4.2 EL OPERADOR DE CORTE.

El operador corte, representado por el símbolo "!" nos da un cierto control sobre el mecanismo de deducción del PROLOG. Su función es la de controlar el proceso de reevaluación, limitándolo a los hechos que nos interesen. Supongamos la siguiente regla:

```
regla:- hecho1,hecho2,!,hecho3,hecho4,hecho5.
```

PROLOG efectuará reevaluaciones entre los hechos 1, 2 sin ningún problema, hasta que se satisface el hecho2. En ese momento se alcanza el hecho3, pudiendo haber a continuación reevaluaciones de los hechos 3, 4 y 5. Sin



embargo, si el hecho3 fracasa, no se intentara de ninguna forma reevaluar el hecho2.

Una interpretación práctica del significado del corte en una regla puede ser que "si has llegado hasta aquí es que has encontrado la única solución a este problema y no hay razón para seguir buscando alternativas".

Aunque se suele emplear como una herramienta para la optimización de programas, en muchos casos marca la diferencia entre un programa que funciona y otro que no lo hace.

### 4.3 ELEMENTOS DE UNA LISTA.

*Buscar 02\_elementosdeunalista.pl en la carpeta de ejemplos.*

El siguiente código es para saber si un elemento se encuentra dentro de una lista y/o para sacar a la cabecera de la lista. Elemento (X,R). es verdadero si X es una lista y R es elemento de esa lista.

Así también veremos el uso de la recursividad y el caso base en prolog.

#### Código:

```
elemento([X|R],X). %A esto se le llama el caso base.  
elemento([X|R],Y):-elemento(R,Y).
```

#### Consulta:

```
:-elemento([a,b,c,d],c).  
yes  
:-elemento([a,b,c,d],e).  
no
```

#### - ANÁLISIS DEL EJEMPLO ANTERIOR.

*Nivel 1.- X=a, R=[b,c,d], Y=c*

elemento([X|R],X). → Aquí diría que si, si Y fuera igual a 'a' o sea el primer elemento.  
elemento([X|R],Y):-elemento(R,Y). → Todo coincide así que entra al siguiente nivel.

*Nivel 2.- X=b, R=[c,d], Y=c*

elemento([X|R],X). → no.  
elemento([X|R],Y):-elemento(R,Y). → Todo coincide así que entra al siguiente nivel.

*Nivel 3.- X=c, R=[d], Y=c*

elemento([X|R],X).

→ X con X, si. R con R, si. X con X, si, puesto que X (el primer elemento de la lista coincide con el elemento que se busca). Por lo tanto la consola dirá que si.



Si observamos lo que se hace en este ejemplo no es más que recursividad. Primero se separa la lista en dos partes, el primer elemento X y el resto R, y se al elemento que se busca, o sea, la letra c también se le asigna una variable X, si las dos X son iguales, entonces la respuesta es si, y si no se sigue con la siguiente línea.

En la segunda línea el primer elemento vuelve a ser X, R el resto y Y el elemento que se busca. En este caso la sintaxis si coincide, por lo que entra a la primera acción de esta línea que es otra consulta *elemento(R,Y)*.

Entonces volvemos a empezar, X es igual al resto del nivel anterior, si recordamos R incluye todos los elementos de la lista a excepción del primero, por lo que vemos que hace una recursividad eliminando uno a uno los elementos de la lista hasta que el que esta primero, llamado X, coincida con el que se busca en el caso base. De no coincidir en ninguno de los casos, prolog responde que no.

Pero hay un punto que se debe de notar para no confundirnos, en prolog, aunque aya varias variables con el mismo nombre, no quiere decir que tienen que tener el mismo valor, esto depende del nivel y de los estados.

#### 4.4 ARIDAD.

Buscar 03\_aridad.pl en la carpeta de ejemplos.

Aridad es el numero de elementos de una lista.

##### Código:

```
aridad([ ],0). %Caso base
aridad([X|R],N):-aridad(R,Z),N is Z+1.
```

##### Consulta:

```
:?-aridad([a,b,c,d,e],N).
N=5.
```

##### - ANÁLISIS DEL EJEMPLO ANTERIOR.

*Nivel 1.- X=a, R=[b,c,d,e], N=N*

aridad([ ],0). → Aquí tenemos una lista vacía, así que continua a la siguiente línea.  
aridad([X|R],N):-aridad(R,Z),N is Z+1.

→ Aquí todo coincide, pues son variables, así que pasa a la primera acción, que es volver a empezar pero sin el primer elemento de la lista y así hasta llegar a la e.

*Nivel 2.- X=b, R=[c,d,e], N=N*

*Nivel 3.- X=c, R=[d,e], N=N*

*Nivel 4.- X=d, R=[e], N=N*

*Nivel 5.- X=e, R=[ ], N=N*



En este nivel vemos que la lista vacía y la variable N coincidirán con el caso base, así que se dará por terminada la acción *aridad(R,Z)* por lo que se continuara con la segunda acción del Nivel 5, *N is Z+1*

*Nivel 5.- N=1*

Al terminar con esto, terminara con el nivel 4 y continuara con la segunda acción del nivel anterior.

*Nivel 4.- N=2*

*Nivel 3.- N=3*

*Nivel 2.- N=4*

*Nivel 1.- N=5*

Y así es como conseguimos la respuesta de la consola.

## 4.5 CONCATENAR.

Buscar *04\_concatenar.pl* en la carpeta de ejemplos.

Concatenar no es más que unir dos elementos o listas. Lo cual se podría decir que es unir 2 listas, pues una lista puede contener solo un elemento. De esto concluimos que concatenar es unir la lista L1 y L2 en una lista L3.

### Código:

```
conc([ ], L2, L2) .
conc([X|R], L2, [X|Z]) :- conc(R, L2, Z) .
```

### Consulta:

```
:?-conc([a,b,c,d,e], [1,2,3], R) .
R=[a,b,c,d,e,1,2,3]
```

### - ANÁLISIS DEL EJEMPLO ANTERIOR.

$\text{conc}([ ], L2, L2)$ . → No entra por que no se tiene un arreglo vacío.  
 $\text{conc}([X|R], L2, [X|Z])$ :- $\text{conc}(R, L2, Z)$ . → Entra al primer nivel.

Aquí lo que se hace es estar eliminando el primer elemento de la lista hasta lograr una lista vacía para así poder entrar al caso base.

*Nivel 1.- X=a, R=[b,c,d,e], L2=[1,2,3]*

*Nivel 2.- X=b, R=[c,d,e], L2=[1,2,3]*

*Nivel 3.- X=c, R=[d,e], L2=[1,2,3]*

*Nivel 4.- X=d, R=[e], L2=[1,2,3]*

*Nivel 5.- X=e, R=[ ], L2=[1,2,3]*

En este nivel entrara al caso base.



Nivel 6.-  $L2=[1,2,3]$

Aquí comenzara a regresar cerrando los niveles y así ir concatenando todos los elementos el la lista R, terminando así con la conclusión.

$R=[a,b,c,d,e,1,2,3]$

#### 4.6 INVERSO DE UNA LISTA.

Buscar *05\_inversodeunalista.pl* en la carpeta de ejemplos.

El inverso de (X, L) es verdadero si X es una lista y Y es el inverso de la lista X. Si tenemos  $L=[a,b,c]$ , su inverso es  $Li=[c,b,a]$ .

##### Código:

```
inverso([ ],[ ]).
inverso([X],[X]).
inverso([X,Y],[Y,X]).
inverso([X|R],E):-inverso(R,Z),conc(Z,[X],E).
```

##### Consulta:

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

Para este ejemplo es importante tener en cuenta que se esta usando el programa concatenar que ya vimos con anterioridad, así que para que este ejemplo funcione tenemos que abrir también el programa concatenar y compilarlos todos o bien agregar el código del programa anterior a este.

##### - ANÁLISIS DEL EJEMPLO ANTERIOR.

Nivel 1.-  $X=a, R=[b,c], \rightarrow inverso([b,c],Z)$

Nivel 2.-  $X=b, R=[c], \rightarrow inverso([c],Z)$

Nivel 3.-  $X=c, R=[ ], \rightarrow inverso([ ],Z)$

Aquí entrara al caso base y regresándose para entrar a la segunda acción que es *conc(Z,[X],E)* que estará concatenando primero c, luego b y al final a.

#### 4.7 DUPLICAR LOS ELEMENTOS DE UNA LISTA.

Buscar *06\_duplicar.pl* en la carpeta de ejemplos.

El siguiente código duplicara a cada uno de los elementos pertenecientes a la lista X. *duplicar(X,Y)*. es verdadero si X es una lista y Y es las lista X pero con los elementos duplicados.



$Lx=[1,2,3] \rightarrow Ly=[1,1,2,2,3,3]$

### Código:

```
duplicar([ ],[ ]).
duplicar([X],[X,X]).
duplicar([C|L1],[C,C|L2]):-duplicar(L1,L2).
```

### Consulta:

```
:?-duplicar ([a,b,c],R).
R=[a,a,b,b,c,c]
```

#### - ANÁLISIS DEL EJEMPLO ANTERIOR.

Nivel 1.-  $C=a$ ,  $L1=[b,c]$ ,  $duplicar([b,c],[a,a,b,c])$

Nivel 2.-  $C=b$ ,  $L1=[c]$ ,  $duplicar([c],[a,a,b,b,c])$

Nivel 3.-  $C=c$ ,  $L1=[ ]$ ,  $duplicar([ ],[a,a,b,b,c,c])$

$R=[a,a,b,b,c,c]$

## 4.8 QUITAR EL ÚLTIMO ELEMENTO DE LA LISTA.

Buscar 07\_quitar.pl en la carpeta de ejemplos.

La cláusula  $quitar(A,B)$  es verdadero si A es una lista y B es la misma lista sin el último elemento de la lista A.

$L1=[a,b,c,d] \rightarrow Lq=[a,b,c]$

### Código:

```
quitar([A],[ ]).
quitar([C|P1],[C|P2]):-quitar(P1,P2).
```

### Consulta :

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

#### -4 ANÁLISIS DEL EJEMPLO ANTERIOR.

Nivel 1.-  $C=a$ ,  $P1=[b,c,d]$ ,  $quitar([b,c,d],X)$

Nivel 2.-  $C=b$ ,  $P1=[c,d]$ ,  $quitar([c,d],X)$

Nivel 3.-  $C=c$ ,  $P1=[d]$ ,  $quitar([d],X)$

De regreso:

Nivel 3.-  $X=[a]$

Nivel 2.-  $X=[a,b]$

Nivel 1.-  $X=[a,b,c]$



## 4.9 QUITAR LOS DOS ÚLTIMOS ELEMENTOS DE UNA LISTA.

Buscar 08\_quitardos.pl en la carpeta de ejemplos.

quitardos(X,Y) . Es verdadero si X es una lista y Y es la misma lista, sin los dos últimos elementos.

$$L1=[a,b,c,d] \rightarrow Lq=[a,b]$$

### Código:

```
quitardos([A,B],[ ]).
quitardos([C|P1],[C|P2]):-quitardos(P1,P2).
```

### Consulta :

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

### - ANÁLISIS DEL EJEMPLO ANTERIOR.

Nivel 1.-  $C=a$ ,  $P1=[b,c,d]$ ,  $quitardos([b,c,d],X)$

Nivel 2.-  $C=b$ ,  $P1=[c,d]$ ,  $quitardos([c,d],X)$

De regreso:

Nivel 3.-  $X=[a]$

Nivel 2.-  $X=[a,b]$

## 4.10 BORRAR UN ELEMENTO CUALQUIERA DE UNA LISTA.

Buscar 09\_borrar.pl en la carpeta de ejemplos.

borrar(X,Y,Z). es verdadero si X es un elemento de la lista Y, y Z es la lista Y sin el elemento X.

$$L1=[a,b,c,d] \rightarrow Lq=[a,b,d]$$

### Código:

```
borrar(X,[ ],[ ]).
borrar(X,[X|R],L):-borrar(X,R,L).
borrar(X,[Y|R],[Y|L]):-X\==Y,borrar(X,R,L).
```

### Consulta:

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



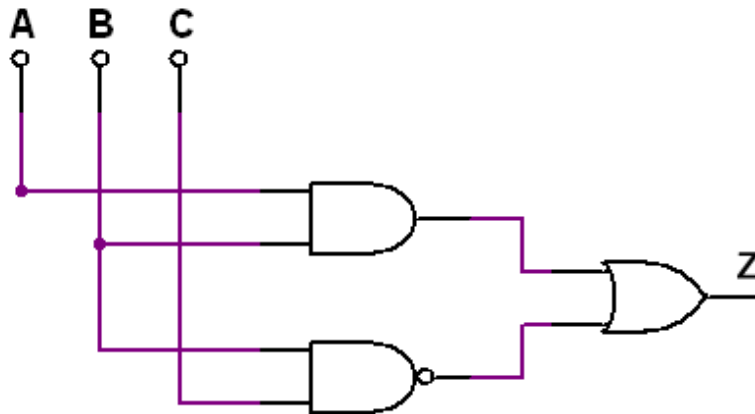
## 4.11 COMPUERTAS LÓGICAS.

Buscar 10\_compuertaslogicas.pl en la carpeta de ejemplos.

Las compuertas lógicas básicas funcionan bajo las siguientes tablas de verdad:

AND			OR			NOT	
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

En base a estas compuertas podemos tener otras como las NAND, NOR, entre otras. En el siguiente ejemplo, haremos un programa para resolver el siguiente diagrama.



### Código:

```
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
or(0,0,0).
or(0,1,1).
or(1,0,1).
or(1,1,1).
not(0,1).
not(1,0).
circ(A,B,C,Z):-and(A,B,X),and(B,C,Y),not(Y,R),or(X,R,Z).
```

### Consulta:

```
?-circ(0,1,0,Z).
Z=1
```

**Nota:** Para un mejorar el código se puede implementar lo que conocemos como corte,"!" y el guión bajo, el cual nos ayuda a decirle a prolog que no importa lo que haya en ese espacio, se usa como comodín.





### Código:

```
and(0,0,0) .  
and(1,1,1):-!.  
and(_,_ ,0) .  
or(0,0,0):-!.  
or(_,_ ,1) .  
not(0,1) .  
not(1,0) .  
circ(A,B,C,Z):-and(A,B,X),and(B,C,Y),not(Y,R),or(X,R,Z) .
```

Y así reducimos en parte el código haciendo más eficiente el programa, así como la forma de programarlo.

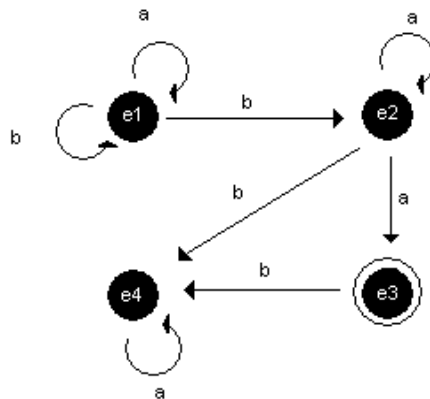
Se necesitan dos 1's para que se cumpla la premisa de una compuerta AND, cualquier otra cosa, será falso.

En la compuerta OR, solo cuando se tienen dos 0's de entrada que dan falso, en los demás casos dará verdadero. (\_,\_,0) significa que no importa lo que se ponga, siempre dará como resultado 0.

### 4.12 AUTÓMATA NO DETERMINISTICO.

Buscar 11\_automata.pl en la carpeta de ejemplos.

Dos caminos posibles para un mismo estado. Observar si acepta una cadena.



e1 es el estado inicial y e4 es el estado final.

### Código:

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



```
aceptar(e3, []).
aceptar(E, [X|R]) :- trans(E, X, F), aceptar(F, R).
```

### Consulta:

```
?-aceptar(e1, [a,a,a,b,a,a]).
yes
```

Responderá que si, mientras la combinación sea valida a la programación del autómatas, de lo contrario dirá no.

## 4.13 DIVIDIR UNA LISTA EN PARES Y NONES.

Buscar 12\_dividir.pl en la carpeta de ejemplos.

El programa será valido si tenemos una lista con números pares y nones y obtenemos dos listas, una con los pares y otra con los nones.

$L1=[3,4,6,7] \rightarrow Lp=[4,6] \rightarrow Ln[3,7]$

### Código:

```
dividir([], [], []).
dividir([L|R], [L|N], P) :- T is L/2, C is fp(T), C>0, dividir(R, N, P).
dividir([L|R], N, [L|P]) :- W is L/2, E is fp(W), E=0, dividir(R, N, P).
```

### Consulta:

```
?-dividir([3,4,6,7], N, P).
N=[3,7],
P=[4,6]
```

### - ANÁLISIS DEL EJEMPLO ANTERIOR.

Nivel 1.-  $L=3$ ,  $R=[4,6,7]$ ,  $N=[]$ ,  $p=[]$ ,  $dividir([4,6,7], N, P)$

Nivel 2.-  $L=4$ ,  $R=[6,7]$ ,  $N=[]$ ,  $p=[]$ ,  $dividir([6,7], N, P)$

Nivel 3.-  $L=6$ ,  $R=[7]$ ,  $N=[]$ ,  $p=[]$ ,  $dividir([7], N, P)$

Nivel 4.-  $L=7$ ,  $R=[]$ ,  $N=[]$ ,  $p=[]$ ,  $dividir([], N, P)$

De regreso gracias al caso base ira a pegando una a una cada lista.



## **UNIDAD 5 – ALGORITMOS DE ORDENACIÓN DE LISTAS**

### **5.1 BUBLE SORT.**

*Buscar 13\_bubblesort.pl en la carpeta de ejemplos.*

El algoritmo de la burbuja (BubbleSort) consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el arreglo anterior, {40,21,4,9,10,35}:

#### **Primera pasada:**

{21,40,4,9,10,35} → Se cambia el 21 por el 40.  
{21,4,40,9,10,35} → Se cambia el 40 por el 4.  
{21,4,9,40,10,35} → Se cambia el 9 por el 40.  
{21,4,9,10,40,35} → Se cambia el 40 por el 10.  
{21,4,9,10,35,40} → Se cambia el 35 por el 40.

#### **Segunda pasada:**

{4,21,9,10,35,40} → Se cambia el 21 por el 4.  
{4,9,21,10,35,40} → Se cambia el 9 por el 21.  
{4,9,10,21,35,40} → Se cambia el 21 por el 10.

#### **Código:**

```
burbuja([], []).  
burbuja(Xs, Ys) :- comparar(Ws, [A, B | Zs], Xs), B < A,  
    comparar(Ws, [B, A | Zs], Vs),  
    burbuja(Vs, Ys), !.  
burbuja(Xs, Xs).  
comparar([], Ys, Ys).  
comparar([X | Xs], Ys, [X | Zs]) :- comparar(Xs, Ys, Zs).
```

#### **Consulta:**

```
?-burbuja([10, 1, 0, 4, 21, 9], B).  
B=[0, 1, 4, 9, 10, 21]
```

### **5.2 QUICK SORT.**

*Buscar 14\_quicksort.pl en la carpeta de ejemplos.*

Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el arreglo en arreglos más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del arreglo como pivote, y se mueven todos los elementos menores de este pivote a su izquierda, y los mayores a su derecha.

La técnica de diseño de algoritmos llamada "divide y vencerás" (divide and conquer) consiste en descomponer el problema original en varios sub-



problemas más sencillos, para luego resolver éstos mediante un cálculo sencillo.

Por último, se combinan los resultados de cada sub-problema para obtener la solución del problema original.

### Código:

```
suma([ ],L,L). % caso base para la unión de listas.
suma([C|L1],L2,[C|L3]):-suma(L1,L2,L3). % saca el resto de la
lista uno y la lista dos para unir las tres listas.
dividir(_,[ ],[ ],[ ]). % caso base para la división de listas.
dividir(E,[C|R],[C|Men],May):-E >= C, % divide y compara el
elemento con la otra lista para insertarlos con los mayores.
dividir(E,R,Men,May). % hace la división de las listas menores y
mayores.
dividir(E,[C|R],Men,[C|May]):-E < C, % divide y compara el
elemento con la otra lista para insertarlos con los menores.
dividir(E,R,Men,May).
qsort([ ],[ ]). % caso baso cuando la lista esta vacía.
qsort([C|R],L):-dividir(C,R,Men,May),qsort(Men,MayM), % saca la
cabecera y el resto de la lista para empezar a dividir la lista
con los mayores y menores.
qsort(May,MayM),suma(MenM,[C|MayM],L). % tiene los elementos
menores y mayores en sus respectivas listas, hace la concatenación
o unión.
```

### Consulta:

```
?-qsort([1,20,6,5,3,0],Q).
Q = [0,1,3,5,6,20]
```

## 5.3 MERGE SORT.

Buscar *15\_mergesort.pl* en la carpeta de ejemplos.

Puede verse claramente que el algoritmo mergesort requiere de dos pasadas a los vectores a “mergear”, una para realizar el merge y otra para copiar del vector auxiliar al vector original. El tiempo del algoritmo de merge es  $T(n) = 2n$

Merge:  $T(n) = Q(n)$

La recurrencia del *algoritmo de MergeSort Completo* es:

$T(n)=1$  Si  $n=1$   
 $T(n) = T(n/2)+(n)$

Esta es una recurrencia conocida. Aplicando el teorema podemos obtener que:

MergeSort =  $Q(n \log n)$ .

Por lo tanto el algoritmo de MergeSort



$$T(n) = Q ( n \cdot \log(n) )$$

Es estable. Requiere de un vector auxiliar para realizar el ordenamiento.

### Código:

```
mergesort([], []). % caso base cuando las listas estén vacías.
mergesort([A], [A]). % se introducirá una lista, y arrojará un
    resultado con esa misma lista, pero ordenado.
mergesort([A,B|R], S):-partir([A,B|R], L1, L2), %separa cada elemento
    de la lista y lo mete en S.
    mergesort(L1, S1), %coloca la lista 1 en S1.
    mergesort(L2, S2), %coloca la lista 2 en S2.
    merge(S1, S2, S). % manda llamar a merge donde hace las
    particiones.
partir([], [], []). % hace las tres particiones.
partir([A], [A], []). % tiene las dos listas y una vacía donde
    pondrá la lista ordenada.
partir([A,B|R], [A|Ra], [B|Rb]):-partir(R, Ra, Rb).
merge(A, [], A). % si la segunda lista es vacía , el resultado sería
    los elementos de la lista 1.
merge([], B, B). % si la primera lista es vacía , el resultado sería
    los elementos de la lista 2.
merge([A|Ra], [B|Rb], [A|M]):-A <= B, merge(Ra, [B|Rb], M). % compara
    un elemento de la lista A si es menor o igual al elemento de la
    lista B.
merge([A|Ra], [B|Rb], [B|M]):-A > B, merge([A|Ra], Rb, M). % compara un
    elemento de la lista A si es mayor al elemento de la lista B.
```

### Consulta:

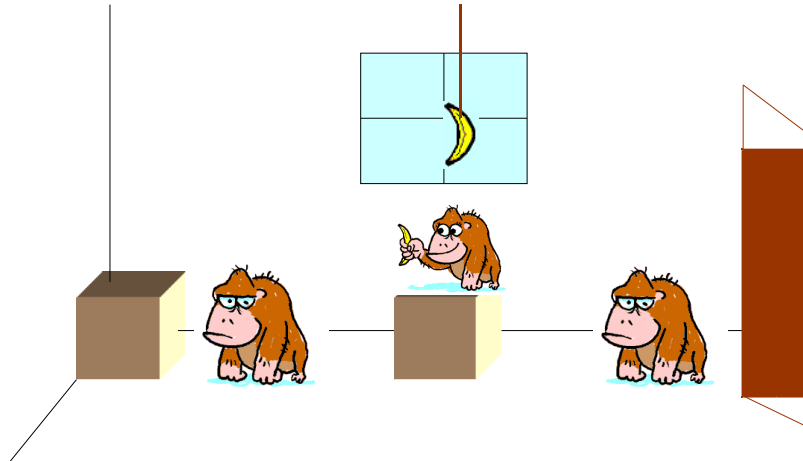
```
?-mergesort([2,5,3,7], R).
R=[2,3,5,7]
```



## UNIDAD 6 – EJEMPLOS ADICIONALES.

### 6.1 EL MONO Y LA BANANA.

Buscar 16\_monobanana1.pl y 17\_monobanana2.pl en la carpeta de ejemplos.



#### Movimientos (movs):

- tomar el plátano
- subir a la caja
- empujar la caja
- caminar

**Descripción:** edo(PosHorizMono, PosVertMono, PosCaja, TienePlátano).

- PosHorizMono: { puerta, ventana, centro }
- PosVertMono: { piso, sobre\_caja }
- PosCaja: { puerta, ventana, centro }
- TienePlátano: { si, no }

Representar cambios de estado con 3 argumentos:

#### Código:

```
mov(estado1, accion, estado2).  
mov(edo(centro, sobre_caja, centro, no), toma,  
    edo(centro, sobre_caja, centro, si)).  
mov(edo(X, piso, X, Y), sube, edo(X, sobre_caja, X, Y)).  
mov(edo(X, piso, X, Y), empuja(X, Z), edo(Z, piso, Z, Y)).  
mov(edo(X, piso, Y, Z), camina(X, W), edo(W, piso, Y, Z)).  
come(edo(_, _, _, si)).  
come(Edo1) :- mov(Edo1, Acc, Edo2),  
    come(Edo2).  
P2.
```

#### Consulta:

```
?- come(edo(puerta, piso, ventana, no)).  
Yes
```



```
?-come(edo(ventana,sobre_caja,centro,si)).
yes
```

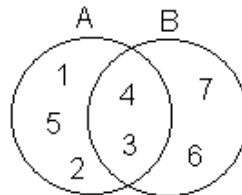
Otro ejemplo seria:

**Código:**

```
sol(estado(_,_,_,_,si)).
sol(E1):-trans(E1,Accion,E2),sol(E2).
trans(estado(_,_,_,si,no),comer,estado(_,_,_,si,si)).
trans(estado(y,y,si,no,no),tomar,estado(y,y,si,si,no)).
trans(estado(P1,P1,no,_,_),tregar,estado(P1,P1,si,_,_)).
trans(estado(P1,P1,no,no,_),mover,estado(P2,P2,no,no,_-P1=\=P2.
trans(estado(_,P1,no,_,_),caminar,estado(_,P2,no,_,_-P1=\=P2.
```

## 6.2 UNIÓN – INTERSECCIÓN.

Buscar 18\_union-interseccion.pl en la carpeta de ejemplos.



UNION = 1,2,3,4,5,6,7  
INTERSECCION = 3,4

**Código:**

```
%elemento(X,L).
elemento(X,[]):-fail,!.
%la linea anterior se puede borrar para optimizar
elemento(X,[X|R]).
elemento(X,[Y|R]):-elemento(X,R).
%union(P,Q,Z).
union([],Q,Q).
union([X|R],Q,[X|Z]):- \+ elemento(X,Q),!,union(R,Q,Z).
union([X|R],Q,Z):-union(R,Q,Z).
%inters(P,Q,Z).
inters([],Q,[]).
inetr([X|R],Q,[X|Z]):-elemento(X,Q),!,inters(R,Q,Z).
inters([X|R],Q,Z):-inters(R,Q,Z).
```

**Consulta:**

```
?-union([1,2,3,5],[2,1,6,7],Z).
Z = [3,5,2,1,6,7] ;
```



### 6.3 8 REINAS.

Buscar 19\_8reinas.pl en la carpeta de ejemplos.

Programa que genera posiciones el las que en una tabla de ajedrez se acomodan 8 reinas de tal manera que no se puedan comer unas a otras.

					↖		♚
			♚	↖			
			↖			♚	
---	---	+	---	---	---	---	---
	↖				♚		
↖	♚						
				♚			
♚							

#### Código:

```

elemento(X,[X|R]). % Busca el elemento en una lista
elemento(X,[Y|R]):-elemento(X,R).
plantilla([1/S1,2/S2,3/S3,4/S4,5/S5,6/S6,7/S7,8/S8]). % plantilla
de columnas y renglones
solucion([]).% caso base la lista esta vacía
solucion([A/B|R]):-solucion(R),elemento(B,[1,2,3,4,5,6,7,8]),
no_ataca(A/B,R).
no_ataca(_,[]).
no_ataca(X/Y,[A/B|R]):-X=\=A,Y=\=B,
A-X=\=B-Y,
X-A=\=B-Y, % que los renglones y columnas sean diferentes para
que no se coman
no_ataca(X/Y,R). % Busca que no se encuentren en la misma lista o
renglón

```

#### Consulta:

?- plantilla(S),solucion(S).

S = [1 / 4,2 / 2,3 / 7,4 / 3,5 / 6,6 / 8,7 / 5,8 / 1] ;

Si presionamos ';' tendremos mas posibles resultados.

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] ;



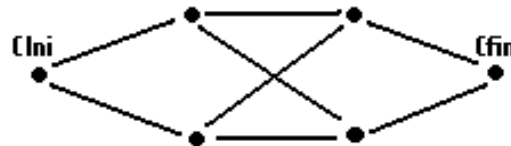


```
S = [1 / 3,2 / 6,3 / 4,4 / 2,5 / 8,6 / 5,7 / 7,8 / 1] ;
S = [1 / 5,2 / 7,3 / 1,4 / 3,5 / 8,6 / 6,7 / 4,8 / 2] ;
S = [1 / 4,2 / 6,3 / 8,4 / 3,5 / 1,6 / 7,7 / 5,8 / 2] ;
S = [1 / 3,2 / 6,3 / 8,4 / 1,5 / 4,6 / 7,7 / 5,8 / 2] ;
S = [1 / 5,2 / 3,3 / 8,4 / 4,5 / 7,6 / 1,7 / 6,8 / 2] ;
S = [1 / 5,2 / 7,3 / 4,4 / 1,5 / 3,6 / 8,7 / 6,8 / 2] ;
S = [1 / 4,2 / 1,3 / 5,4 / 8,5 / 6,6 / 3,7 / 7,8 / 2] ; ...
```

Aproximadamente son 64 posiciones posibles para que estas reinas no se coman entre si.

## 6.4 RUTA DE VUELOS.

Buscar 20\_rutas.pl en la carpeta de ejemplos.



Ejemplos de posibles rutas.

### Código:

```
% plan(Ciudad1, Ciudad2, hora, dia, variable).
plan(CiudadIni, CiudadFin, Dia, TSal,TLleg,NVuelo):-
    vuelo(Dia, CiudadIni, CiudadFin).
vuelo(Dia,CiudadIni,CiudadFin):-tabla(CiudadIni,CiudadFin,Lista),
    elemento(LDias,Lista).
elemento(X,[X|R]).
elemento(Y,[X|R]):-elemento(Y,R).
% tabla de vuelos
tabla(tij,cul,[8,10,ticu1,[lu,mi,vi,do]]).
tabla(tij,cul,[14,17,ticu2,[ma,ju,sa]]).
tabla(tij,cul,[19,22,ticu3,[lu,sa,do]]).
tabla(cul,chi,[9,14,tichi1,[lu,ju,sa]]).
tabla(cul,chi,[12,17,tichi2,[lu,mi,sa,do]]).
tabla(chi,df,[18,20,chidf1,[lu,sa]]).
plan(CInic,CFinal,Dia,Hora,[CInic-CFinal/V/TS]):-
    tabla(CInic,CFinal,[TS,TLL,V,LDias]),
    Hora <= TS, % que la hora deseada sea antes de la hora de
    vuelo
    elemento(Dia,LDias). %Busca el dia en la lista de días del
    vuelo deseado
plan(CInic,CFinal,Dia,Hora,[CInic-E/V/TS|R]):-
    tabla(CInic,E,[TS,TLL,V,LDias]),
    Hora <= TS,
    elemento(Dia,LDias),
    plan(E,CFinal,Dia,TLL,R).
```

### Consulta:

```
?-plan(tij,cul,lu,8,S).
S=[tij - cul / ticu1 / 8] ;
S=[tij - cul / ticu3 / 19] ;
```



no



## **UNIDAD 7 – LECTURA Y ESCRITURA.**

### **7.1 LECTURA Y ESCRITURA DE TÉRMINOS.**

PROLOG, al igual que la mayoría de lenguajes de programación modernos incorpora predicados predefinidos para la entrada y salida de datos. Estos son tratados como reglas que siempre se satisfacen.

- *write*.

Su sintaxis es:

```
write('Hello world.').
```

Las comillas simples encierran constantes, mientras que todo lo que se encuentra entre comillas dobles es tratado como una lista. También podemos mostrar el valor de una variable, siempre que esté instanciada:

```
write(X).
```

- *nl*.

El predicado *nl* fuerza un retorno de carro en la salida. Por ejemplo:

```
write('linea 1'), nl, write('linea 2').
```

tiene como resultado:

```
linea                               1
linea 2
```

- *write\_In(+Termino)*

Equivalente a:

```
write(+Termino),nl
```

- *writeln(+Formato,+Argumentos)*

Escritura con formato

- *tab(+X)*

Desplaza el cursor a la derecha *X* espacios. *X* debe estar instanciada a un entero (una expresión evaluada como un entero positivo).

- *display(+Termino)*

Se comporta como “write”, excepto que pasa por alto cualquier declaración de operadores que se haya hecho.



- *read.*

Lee un valor del teclado. La lectura del comando read no finaliza hasta que se introduce un punto ".". Su sintaxis es:

```
read(X).
```

Instancia la variable X con el valor leído del teclado.

```
read(ejemplo).
```

Se evalúa como cierta siempre que lo tecleado coincida con la constante entre paréntesis (en este caso 'ejemplo').

### - *LOS COMANDOS WRITE Y READ.*

*Buscar 21\_write-read.pl en la carpeta de ejemplos.*

Programa que te pide la longitud de una lista e imprime el numero de los elementos con '\*' y busca cualquier elemento

#### **Código:**

```
lista :- write('cuantos elementos?'), read(N), % lee el numero de
elementos de la lista
    leer(N,L),impri_lista(L),nl,asteriscos(L),nl,      % lee los
elementos y los imprime con *.
    write('Elemento a buscar en la lista :'), % escribe enunciado
    read(X), busca(X,L). % lee en elemento a buscar y lo busca en la
lista
impri_lista([]).
impri_lista([A|R]):- write([A]),impri_lista(R). % imprime la lista
leer(0,L).
leer(N,[A|L]):-read(A), Z is N-1,leer(Z,L). %lee los N elementos y
los mete ala lista L
busca(X,[]):-write('No esta !'),nl.
busca(X,[X|R]):-write('Si esta !'),nl. % busca el elemento X en la
lista y te dice si esta
busca(X,[Y|R]):-busca(X,R). % si no esta empieza a buscar el
siguiente
asteriscos([]):-!.
asteriscos([A|R]):- pon(A),nl,asteriscos(R). % saca la lista y la
imprime en asteriscos
pon(0). % no hace nada si detecta un cero
pon(X):-write('*'),Z is X-1,pon(Z). % escribe asteriscos un X un
número de veces
```

#### **Consulta:**

```
?- lista.
cuantos elementos?: 6.
|: 1.
|: 2.
|: 3.
|: 4.
```



```
|: 5.  
|: 6.  
[1][2][3][4][5][6]  
*  
**  
***  
****  
*****  
*****
```

```
Elemento a buscar en la lista |: 10.  
No esta !  
yes
```

### - PROGRAMA CUBO.

Buscar 22\_cubo-a.pl y 23\_cubo-b.pl en la carpeta de ejemplos.

Programa que imprime el cubo de un número

#### - Parte a)

#### Código:

```
cubo(N,C):-C is N*N*N. % N es un numero
```

#### Consulta:

```
?- cubo(3,C).  
C = 27
```

#### - Parte b)

#### Código:

```
cubo:-write('dime un numero '),  
      read(X), % lee en numero  
      proceso(X).  
proceso(alto):-!. % se detiene hasta que encuentra la palabra  
                'alto'  
proceso(X):-C is X*X*X, %multiplica el numero 3 veces  
write('El cubo es '), % escribe oración  
write(C), nl, cubo. % imprime el resultado y te pide otro número
```

#### Consulta:

```
?- cubo.  
dime un numero |: 3.  
El cubo es 27  
dime un numero |: 4.  
El cubo es 64  
dime un numero |: 5.  
El cubo es 125  
dime un numero |: 6.  
El cubo es 216  
dime un numero |: 7.  
El cubo es 343
```



```
dime un numero |: 8.  
El cubo es 512  
dime un numero |: 9.  
El cubo es 729  
dime un numero |: 10.  
El cubo es 1000
```

## 7.2 LECTURA Y ESCRITURA DE CARACTERES.

El caracter es la unidad más pequeña que se puede leer y escribir. Prolog trata a los caracteres en forma de enteros correspondientes a su código ASCII.

- *put(+Character)*

Si *Character* está instanciada a un entero entre 0..255, saca por pantalla el caracter correspondiente a ese código ASCII. Si *Carácter* no está instanciada o no es entero, error.

- *get(-Character)*

Lee caracteres desde el teclado, instanciando *Character* al primer carácter imprimible que se teclee. Si *Character* ya está instanciada, los comparará satisfaciéndose o fracasando.

- *get0(-Character)*

Igual que el anterior, sin importarle el tipo de carácter tecleado.

- *skip(+Character)*

Lee del teclado hasta el carácter *Character* o el final del fichero.

## 7.3 LECTURA Y ESCRITURA EN ARCHIVOS.

Existe un archivo predefinido llamado *user*. Al leer de este archivo se hace que la información de entrada venga desde el teclado, y al escribir, se hace que los caracteres aparezcan en la pantalla. Este el modo normal de funcionamiento. Pero pueden escribirse términos y caracteres sobre archivos utilizando los mismos predicados que se acaban de ver. La única diferencia es que cuando queramos escribir o leer en un archivo debemos cambiar el *canal de salida activo* o el *canal de entrada activo*, respectivamente. Posiblemente queramos leer y escribir sobre archivos almacenados en discos magnéticos. Cada uno de estos tendrá un *nombre del archivo* que utilizamos para identificarlo.

En Prolog los nombres de archivos se representan como átomos. Los archivos tienen una longitud determinada, es decir, contienen un cierto número de caracteres. Al final del archivo, hay una marca especial de *fin del archivo*. Cuando la entrada viene del teclado, puede generarse un fin del archivo tecleando el carácter de control ASCII 26 o control-Z. Se pueden tener abiertos



varios archivos, si conmutamos el canal de salida activo sin cerrarlos (*told*), permitiéndonos escribir sobre ellos en varios momentos diferentes, sin destruir lo escrito previamente.

- *tell(+NomArchivo)*

Si *NomArchivo* está instanciada al nombre de un fichero, cambia el canal de salida activo. Crea un nuevo fichero con ese nombre.

Si *NomArchivo* no está instanciada o no es un nombre de fichero, producirá un error.

- *telling(?NomArchhivo)*

Si *NomArchivo* no está instanciada, la instanciará al nombre del fichero que es el canal de salida activo.

Si *NomArchivo* está instanciada, se satisface si es el nombre del archivo actual de salida.

- *told*

Cierra el fichero para escritura, y dirige la salida hacia la pantalla.

- *see(+NomFichero)*

Si *NomArchivo* está instanciada al nombre de un fichero, cambia el canal de entrada activo al fichero especificado. La primera vez que se satisface, el fichero pasa a estar abierto y empezamos al comienzo del fichero.

Si *NomArchivo* no está instanciada o no es un nombre de fichero, producirá un error.

- *seeing(?NomFichero)*

Si *NomArchivo* no está instanciada, la instanciará al nombre del fichero que es el canal de entrada activo.

Si *NomArchivo* está instanciada, se satisface si es el nombre del fichero actual de entrada.

- *seen*

Cierra el fichero para lectura, y dirige la entrada hacia el teclado.



**- LEER DATOS DE UN ARCHIVO.**

Buscar 24\_leer.pl en la carpeta de ejemplos.

**Código:**

```
browse(File):-  
  seeing(User),  
  see(File),  
  repeat,  
  read(Datos),  
  proceso(Datos),  
  seen,  
  see(User),  
  !.  
proceso(end_of_file):- !.  
proceso(Datos):-write(Datos),nl,fail.
```

**- OTRAS FUNCIONES:**

```
Put(C). % regresa el carácter, donde C es un ASCII  
Get(C). % regresa el ASCII de C  
Get0(M). % lee un carácter del teclado  
Var(X). % comprueba si X es solo una variable, no se ha unificado  
        con nada.  
Nonvar(X). % comprueba si X es una variable asociada  
Atom(X). % comprueba si x es átomo  
Integer(X). % comprueba si X es un entero  
Real(X). % comprueba si X es un número real  
Atomic(X). % comprueba si X es atómico, entero o real  
Assert(X). % agrega algo a la base de conocimientos  
Retract(X). % Quita algo de la base de conocimientos
```

**- En la consola:**

**Consulta:**

```
?-assert(rojo).  
Yes.
```

**Consulta:**

```
?-rojo.  
Yes.
```

**Consulta:**

```
?- retract(rojo).  
Yes.
```

**Consulta:**

```
?-rojo.  
No.
```





## 7.4 COMPONENTES DE ESTRUCTURAS.

Como ya hemos comentado anteriormente Prolog considera los hechos y las reglas, e incluso los mismos programas, como estructuras. Veamos ahora una de sus ventajas : el considerar a las propias cláusulas como estructuras nos permite manipularlas (construir, modificar, eliminar, añadir, ...) con los predicados de construcción y acceso a los componentes de las estructuras *functor*, *arg*, *=..* y *name*.

```
name(?Atomo, ?Lista)
```

**Átomo:** es un átomo formado por los caracteres de la lista Lista(códigos ASCII). Puede tanto crear un átomo con los caracteres de la lista de códigos ASCII, como crear la lista de caracteres correspondientes al átomo.

- **OBTENER EL FORMATO ASCII DE UNA RELACIÓN.**

abc= Nombre de la relación.

L = Lista de ASCII de A.

### Consulta

```
?- name(abc, L) .  
L = [97, 98, 99]
```

## 7.5 MANIPULACIÓN DE LA BASE DE CONOCIMIENTOS.

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

```
listing  
listing(+Predicado)
```

Todas las cláusulas que tienen como predicado el átomo al que está instanciada la variable *Predicado* son mostradas por el fichero de salida activo (por defecto la pantalla). El predicado de aridad 0 *listing* (sin parámetro) muestra todas las cláusulas de la Base de Conocimientos.

```
clause(?Cabeza, ?Cuerpo)
```

Se hace coincidir *Cabeza* con la cabeza y *Cuerpo* con el cuerpo de una cláusula existente en la Base de Conocimientos. Por lo menos *Cabeza* debe estar instanciada. Un *hecho* es considerado como una cláusula cuyo cuerpo es

```
true.  
assert(+Clausula)  
asserta(+Clausula)  
assertz(+Clausula)
```



Estos predicados permiten añadir nuevas cláusulas a la base de conocimientos. El predicado *asserta* la añade al principio (letra «a») y *assertz* la añade al final (letra «z») de cualquier otra cláusula del mismo tipo que hubiese en la base de conocimientos. En todos los casos, *Cláusula* debe estar previamente instanciada a una cláusula. Dicha cláusula queda incorporada a la base de conocimientos y no se pierde aunque se haga reevaluación.

```
retract(+Cláusula)
retractall(+Cláusula)
```

Permite eliminar una cláusula de nuestra base de conocimientos. Para ello, *Cláusula* debe estar instanciada y se quitará la primera cláusula de la base de conocimientos que empareje con ella. Si se resatisface el objetivo, se irán eliminando, sucesivamente, las cláusulas que coincidan. Con *retractall* se eliminarán todas.

#### - PROGRAMA QUE QUITA LOS ESPACIOS DE UN ENUNCIADO.

Buscar 25\_quitaespacios.pl en la carpeta de ejemplos.

#### Código:

```
squeeze:-get0(C), % lee carácter del teclado
put(C), % regresa el carácter,
dorest(C).
dorest(46):-!. % supprime espacios
dorest(32):-!, % supprime espacios
get(C), % regresa el ASCII de C
put(C),
dorest(C).
dorest(L):-squeeze.
```

#### Consulta:

```
?- squeeze.
|: Programación lógica practicas de laboratorio 2007.
Programación lógica practicas de laboratorio 2007.
yes
```

### 7.6 PREDICADOS DE CONTROL.

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

Siempre falla.

- *fail*

Siempre tiene éxito



- *true*

Permite simular bucles junto con la combinación corte-fail.

- *repeat*

- *PROGRAMA QUE ABRE UN ARCHIVO YA EXISTENTE.*

*Buscar 26\_abrearchivo.pl en la carpeta de ejemplos.*

**Código:**

```
browse(File):- % abre el archivo existente
    seeing(User),
    see(File), % busca archivo
    repeat,
    read(Datos), %lee datos del archivo
    proceso(Datos),
    seen,
    see(User),
    !.
proceso(end_of_file):- !.
proceso(Datos):-write(Datos),nl,fail.
```

**Consulta:**

```
?-browse('archi2.txt').
padre(juan,maria)
padre(maria,jose)
yes
```

- *PROGRAMA QUE CREA Y ABRE UN ARCHIVO.*

*Buscar 27\_creaabrearchivo.pl en la carpeta de ejemplos.*

**Código:**

```
escribir:-get0(X),X=\=13,name(T,[X]),write(T),escribir. % permite
capturar el nombre del archivo
escribir:-X=13,put(32),nl,escribir,nl.
grabar(File):-telling(user), % procedimiento que graba el archivo
    tell(File), % abrir archivo
    escribir,
    told, % cerrar archivo
    tell(user). % salida del archivo
abrir(File):-seeing(user), % procedimiento que abre un archivo
    see(File), % abrir archivo
    repeat,
    read(Dato), %leer datos del archivo
    proceso(Dato),
    seen,
    see(user), % le la fuente de archivo
    !. % detener
proceso(end_of_file):-!. % indica fin del archivo
proceso(Dato):-write(Dato),
    put(46),
    assert(Dato),
```



```
nl, fail.
```

### Consulta:

```
?-grabar('color.txt').  
|: rojo.  
|: azul.  
|: Esc
```

Cierras y vuelves abrir Prolog

### Consulta:

```
?-abrir ('color.txt').  
yes
```

### Consulta:

```
?-azul.  
yes
```

- PROGRAMA QUE CAPTURA UN PROGRAMA EN UN ARCHIVO DESDE LA CONSOLA Y LO AGREGA A LA BASE DE CONOCIMIENTOS.

*Buscar 28\_creaprograma.pl en la carpeta de ejemplos.*

### Código:

```
escribir(File):- write('Escribe:'), % imprime este enunciado  
    nl,  
    telling(User), % le indica el mensaje  
    tell(File), % le indica que archivo  
    escribe, % manda llamar el procedimiento de escribe  
    told, %instrucciones son mandadas  
    tell(User),  
    leer(File), % manda llamar el procedimiento de leer el  
archivo  
    !.  
escribe:-read(N), % lee instrucción  
    proc(N). % manda llamar a este procedimiento en caso de no  
haya nada  
proc(alto):-!.  
proc(N):-write(N),write('.'), % permite volver a capturar el  
archivo  
    nl, escribe.  
leer(File):-seeing(User), % procedimiento que lee el archivo  
    see(File), % busca el archivo  
    repeat,  
    read(Datos), % lee datos  
    proceso(Datos), %manda llamar el proceso de datos  
    seen,  
    see(User), !.  
proceso(end_of_file):-!.  
proceso(Datos):-assertz(Datos),nl, fail. % inserta los datos a la  
base de conocimientos escribir('programa.txt').
```



### Consulta:

```
?- escribir('programa.txt').
Escribe:
|: juan.
|: pepe.
|: elisa.
|: pedro.
|: padre(juan,pepe).
|: padre(elisa,pedro).
|: ALTO.
yes
```

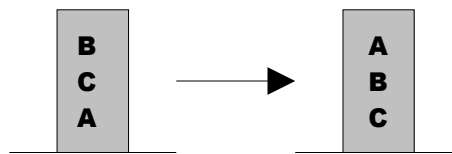
### Consulta:

```
?- leer('programa.txt').
yes
```

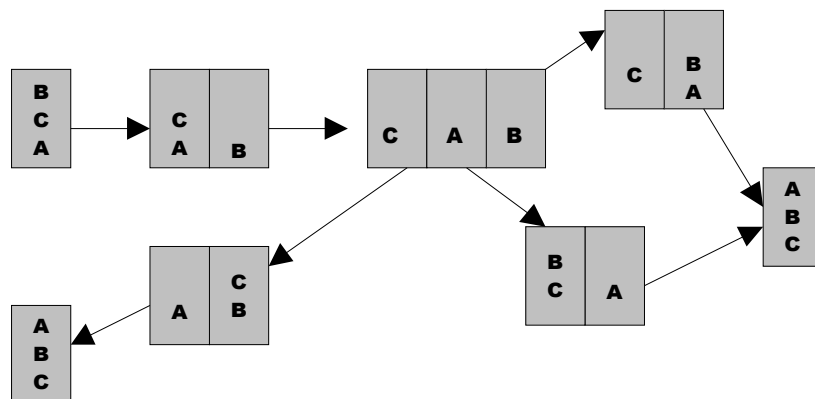
### - BLOQUES

Buscar 29\_bloques.pl en la carpeta de ejemplos.

Programa que ordena una lista de la forma que se le asigne. Lista inicial es X, lista final es Y, listas auxiliares son M y N.



### Caminos posibles para la solución



### Código:

```
concatenar([ ],L,L). % cuando concatenas una lista con una vacía,
regresa la lista
concatenar([X|C1],L2,[X|C3]):-concatenar(C1,L2,C3). % el primer
elemento de la lista se pasa a la lista resultante
```



```

invertir([], []).
invertir([X|C], Z) :-invertir(C, C1),
    concatenar(C1, [X], Z). %invierte una lista
elimina(X, [X|Cola], Cola) :-!. % elimina un elemento X de la lista
elimina(X, [_|C1], [_|C2]) :-elimina(X, C1, C2).
add(X, L, [_|L]). %agrega un elemento a la lista
primero([X|R], X). % regresa el primer elemento de la lista
elemento(X, [X|R]) :-!. % busca si el elemento esta en la lista
elemento(Y, [_|R]) :-elemento(Y, R).
elementat(1, [X|R], X) :-!.
elementat(N, [X|R], S) :-M is N-1, elementat(M, R, S).
elemento2(Y, [X|R], 1) :-elemento(Y, X), !. %busca un elemento en una
lista de listas
elemento2(Y, [X|R], N) :-not(elemento(Y, X)),
    elemento2(Y, R, N1),
    N is N1+1.
insertar(Y, [X|L], 1, Z) :-concatenar([Y], X, T), add(T, L, Z), !.
insertar(Y, [X|L], N, [X|T]) :-M is N-1, insertar(Y, L, M, T).
insertar2(Y, [X|L], 1, Z) :-concatenar([Y], [], T), add(T, L, Z), !.
insertar2(Y, [X|L], N, [X|T]) :-M is N-1, insertar(Y, L, M, T).
vacio([_|_] | Ef, 0) :-!.
vacio([X|Ef], N) :-vacio(Ef, N1), N is N1+1.
donde([X|Ef], 0, X) :-length(X, A), A>0, !.
donde([X|Ef], N, F) :-donde(Ef, N1, F), N is N1+1.
mover([[_|T] | R], N, M) :-insertar(X, R, N, Y), concatenar([T], Y, M).
numacomp(Ef, I, S) :-donde(Ef, S, G), invertir(G, I).
intercambiar(L, N, Q) :-elementat(N, L, X), elimina(X, L, H), add(X, H, Q).
solu(Ei, Ef) :-write(Ei), nl, nl, numacomp(Ef, Y, N), sol(Ei, Ef, Y, N).
sol(X, X, Y, N) :-!. %cuando pones la lista igual de como la quieres
sol([Q|Ei], Ef, [X|Y], N) :-not(elemento(X, Q)),
    elemento2(X, [Q|Ei], B),
    intercambiar([Q|Ei], B, E),
    sol(E, Ef, [X|Y], N),
    !. % cuando no esta en la primer lista el numero que sigue,
    entonces busca el elemento en las demás listas y acomoda al
    principio la lista donde encontrara el elemento
sol([_|_] | Ei, Ef, [X|Y], N) :-elemento2(X, [_|_] | Ei, B),
    intercambiar([_|_] | Ei, B, E),
    sol(E, Ef, [X|Y], N),
    !. % cuando la primer lista esta vacía, busca el las demás
    listas el elemento y acomoda la lista en el principio
sol([Q|Ei], Ef, [Z|Y], N) :-primero(Q, X), Z==X,
    mover([Q|Ei], N, E),
    write(E),
    nl, nl,
    sol(E, Ef, Y, N), !.
sol([Q|Ei], Ef, Y, N) :-vacio([Q|Ei], P),
    P=\=N, mover([Q|Ei], P, E),
    write(E),
    nl, nl,
    sol(E, Ef, Y, N),
    !.
sol([Q|Ei], Ef, Y, N) :-M is N-1,
    mover([Q|Ei], M, E),
    write(E),
    nl, nl,
    sol(E, Ef, Y, N). % cuando el elemento esta al final de la
    primer lista y ya no hay listas vacías donde poner los elementos
    anteriores a el, estos se acomodan en la numero de listas menos 1
    penúltima

```



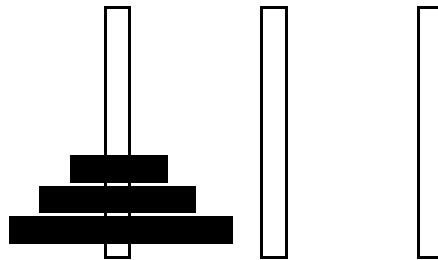
### Consulta:

```
?-solu([[f,d,a,b,c],[],[],[],[],[[],[[],[[],[[,[a,b,c,d,f]]]).
[[f,d,a,b,c],[],[],[[,[,[[
[[d,a,b,c],[[,[,[,[,[f]]
[[a,b,c],[[,[,[,[,[d,f]]
[[b,c],[a],[[,[,[,[d,f]]
[[c],[a],[b],[[,[d,f]]
[[[,[a],[b],[[,[c,d,f]]
[[[,[,[a],[[,[b,c,d,f]]
[[[,[[,[[,[[,[a,b,c,d,f]]

yes
```

### - TORRES DE HANOI

Buscar 30\_torres.pl en la carpeta de ejemplos.



Programa en el que se tiene que mover todos los aros hacia el tercer poste, usando el segundo como un poste auxiliar. Para calcular los movimientos es  $mov=(2^n)-1$ , donde n es el # de aros

### Código:

```
mover(1,X,Y,_):-write('Mover desde arriba '),
                write(X), % imprime en donde esta el aro a mover
                write(' hacia '),
                write(Y), % imprime hacia donde se moverá el aro
                nl.
mover(N,X,Y,Z):- % manda llamar el método
                N>1, % pregunta si n > 1, n es el numero de aros en el poste
                M is N-1, % almacena n-1 en m
                mover(M,X,Z,Y), % manda llamar el método con el Nuevo
                contador como parámetro
                mover(1,X,Y,_),
                mover(M,Z,Y,X). % manda llamar al método con los parámetros
                cambiados
```

### Consulta:



```
?-mover(3,izquierda,derecha,centro) .  
Mover desde arriba izquierda hacia derecha  
Mover desde arriba izquierda hacia centro  
Mover desde arriba derecha hacia centro  
Mover desde arriba izquierda hacia derecha  
Mover desde arriba centro hacia izquierda  
Mover desde arriba centro hacia derecha  
Mover desde arriba izquierda hacia derecha  
yes
```

### - EL PROBLEMA DEL GRANJERO, EL LOBO, EL PATO Y EL TRIGO.

Buscar 31\_glpt.pl en la carpeta de ejemplos.

Un granjero tiene que cruzar un río, el granjero se encuentra al extremo izquierdo del río y debe pasar con sus animales y el trigo hacia el lado derecho, pero solo puede pasar con uno a la vez, por lo cual debe de dejar dos, pero si deja al pato y al trigo, el pato se come al trigo, si deja al lobo y al pato, el lobo se come al pato.

#### Código:

```
glpt([d,d,d,d],_):- nl. % caso base cuando ya están todos del lado  
derecho  
glpt([M,M,P1,T1],R):-cambia(M,M2),  
    cambia(M,L2),  
    P2 = P1,  
    T2 = T1, % cambia de posición al lobo y al granjero, los demás  
permanecen igual  
    not(peligro(M2,L2,P2,T2)), % chequea si no hay peligro de que se  
coman entre si  
    not(elemento([M2,L2,P2,T2],R)), % saca elemento por elemento de  
la lista  
    glpt([M2,L2,P2,T2],[[M,M,P1,T1]|R]), %saca elemento por elemento  
de la lista modificada  
    write('Cambia el granjero y el lobo... '), % imprime este texto  
    write(M2),  
    write(L2),  
    write(P2),  
    write(T2),  
    nl.% imprime las posiciones de cada elemento  
% se repiten los procedimientos, con los elementos que se quieren  
cambiar  
glpt([M,L1,M,T1],R):-cambia(M,M2),  
    cambia(M,P2),  
    L2 = L1,  
    T2 = T1, % cambia de posición al pato y al granjero, los demás  
permanecen igual  
    not(peligro(M2,L2,P2,T2)),  
    not(elemento([M2,L2,P2,T2],R)),  
    glpt([M2,L2,P2,T2],[[M,L1,M,T1]|R]),  
    write('Cambia el granjero y el pato... '),  
    write(M2),  
    write(L2),  
    write(P2),  
    write(T2),  
    nl.
```





```

glpt([M,L1,P1,M],R):-cambia(M,M2),
    cambia(M,T2),
    P2 = P1,
    L2 = L1, % cambia de posición al trigo y al granjero , los demás
    permanecen igual
    not(peligro(M2,L2,P2,T2)),
    not(elemento([M2,L2,P2,T2],R)),
    glpt([M2,L2,P2,T2],[[M,L1,P1,M]|R]),
    write('Cambia el granjero y el trigo... '),
    write(M2),
    write(L2),
    write(P2),
    write(T2),
    nl.
glpt([M,L1,P1,T1],R):-cambia(M,M2),
    T2 = T1,
    P2 = P1,
    L2 = L1, % cambia de posición al granjero cada vez cuando vuelve
    por los demás
    not(peligro(M2,L2,P2,T2)),
    not(elemento([M2,L2,P2,T2],R)),
    glpt([M2,L2,P2,T2],[[M,L1,P1,T1]|R]),
    write('Cambia solo al granjero... '),
    write(M2),
    write(L2),
    write(P2),
    write(T2),
    nl.
peligro(d,i,i,_). % hay peligro cuando el lobo y el pato están
    juntos del lado izquierdo
peligro(i,d,d,_). % hay peligro cuando el lobo y el pato están
    juntos del lado derecha
peligro(d,_,i,i). % hay peligro cuando el pato y el trigo están
    juntos del lado izquierdo.
peligro(i,_,d,d). % hay peligro cuando el pato y el trigo están
    juntos del lado derecho
cambia(d,i). % cambia de posición a los elementos de derecha a
    izquierda
cambia(i,d). % cambia de posición a los elementos de izquierda a
    derecha.
elemento(X, [X|R]).
elemento(X, [Y|R]):- elemento(X,R). % saca el un elemento de la
    lista

```

### Consulta:

```

?- glpt([i,i,i,i],[ ]).
Cambia el granjero y el pato... dddd
Cambia solo al granjero... idid
Cambia el granjero y el trigo... ddid
Cambia el granjero y el pato... idii
Cambia el granjero y el lobo... dddi
Cambia solo al granjero... iidi
Cambia el granjero y el pato... didi
yes

```

**NOTA:** los resultados salen de abajo hacia arriba, el primer renglón, indica el estado final de todos, es decir cuando todos llegan al lado derecho a salvo, se tendría que hacer un proceso para que los resultados los invierta.





## - PROBLEMA DE LOS MISIONEROS Y CANÍBALES.

Buscar 32\_miscan.pl en la carpeta de ejemplos.

Tres misioneros y tres caníbales tienen que cruzar un río en una balsa y poder llegar todos al lado extremo derecho del río, pero si hay mas caníbales que misioneros en la balsa o en cualquiera de los extremos (izquierdo o derecho) los caníbales se comerán a los misioneros.

### Código:

```
miscan:-write('Numero de canibales: '),
        read(Cnum),
        write('Numero de misioneros:'),
        read(Mnum),nl,nl,
        write('Solucion:'),nl,nl,
        cambiap(Mnum,Mnum1),
        cambiap(Cnum,Cnum1),

solucion([ [Mnum1,Cnum1,1], [0,0,0], [[0,0,0], [Mnum1,Cnum1,1]] ).
solucion(Iniciopos,Finalpos):-
movimientos(Iniciopos,Finalpos,[Iniciopos],L),
    imprimesol(L).
    imprimesol([]).
imprimesol([L|T]):-imprimesol(T),
    imprimepaso(L),nl.
imprimepaso([ [A,B,C], [D,E,F] ]):-cambian(A,A1),
    cambian(B,B1),
    cambian(D,D1),
    cambian(E,E1),
    write(' Izquierda: '),
    write([A1,B1,C]),
    write(' Derecha: '),
    write([D1,E1,F]).
movimientos(Finalpos,Finalpos,L,L).
movimientos(Desde,Finalpos,Ltemp,L):-mueveuno(Desde,Hasta),
    noelemento(Hasta,Ltemp),
    movimientos(Hasta,Finalpos,[Hasta|Ltemp],L).
mueveuno([ [Ml1,C11,1], [Mr1,Cr1,0], [Ml2,C12,0], [Mr2,Cr2,1] ]):-
menos(Arriba,s(s(0))),
    menos(s(0),Arriba),
    menos(Marriba,s(s(0))),
    add(Marriba,Carriba,Arriba),
    menos(s(0),Arriba),
    add(Ml2,Marriba,Ml1),
    add(Mr1,Marriba,Mr2),
    add(C12,Carriba,C11),
    add(Cr1,Carriba,Cr2),
    permitido([Ml2,C12]),
    permitido([Mr2,Cr2]).
mueveuno([ [Ml1,C11,0], [Mr1,Cr1,1], [L,R] ]):-
mueveuno([ [Mr1,Cr1,1], [Ml1,C11,0], [R,L] ]).
permitido([0,_]).
permitido([s(M),C]):-menos(C,s(M)).
add(0,X,X).
add(s(X),Y,s(Z)):-add(X,Y,Z).
menos(0,_).
menos(s(X),s(Y)):-menos(X,Y).
```



```
cambian(0,0).
cambian(s(X),N):- cambian(X,N1), N is N1+1.
cambiap(0,0).
cambiap(N,s(X)):-N=\0, N1 is N-1, cambiap(N1,X).
noelemento(_,[]).
noelemento(A,[H|T]):- A \== H,
    noelemento(A,T).
```

### Consulta:

```
?- miscan.
Numero de canibales: |: 3.
Numero de misioneros: |: 3.
```

Solucion:

```
Izquierda: [3,3,1] Derecha: [0,0,0]
Izquierda: [3,1,0] Derecha: [0,2,1]
Izquierda: [3,2,1] Derecha: [0,1,0]
Izquierda: [3,0,0] Derecha: [0,3,1]
Izquierda: [3,1,1] Derecha: [0,2,0]
Izquierda: [1,1,0] Derecha: [2,2,1]
Izquierda: [2,2,1] Derecha: [1,1,0]
Izquierda: [0,2,0] Derecha: [3,1,1]
Izquierda: [0,3,1] Derecha: [3,0,0]
Izquierda: [0,1,0] Derecha: [3,2,1]
Izquierda: [0,2,1] Derecha: [3,1,0]
Izquierda: [0,0,0] Derecha: [3,3,1]
yes
```



## **UNIDAD 8 – PROGRAMACIÓN GRAFICA EN PROLOG (STRAWBERRY PROLOG).**

### **8.1 COMANDOS.**

Strawberry Prolog contiene una gran cantidad de funciones que nos permite hacer uso del modo grafico. A continuación se listan las principales funciones:

#### **- CREACION Y CONTROL DE UNA VENTANA**

`window` → Crea una ventana.  
`edit` → Crea una caja de texto.  
`button` → Crea y controla un botón.  
`check_box` → Crea y controla un check box.  
`radio_button` → Crea y controla un radio botón.  
`static` → Crea una etiqueta.  
`group_box` → Creaa un grupo de control.  
`bitmap` → Crea un control de bitmaps.  
`icon` → Crea y controla un icono.  
`animate` → Permite crear y controlar una animación.  
`list_box` → Crea y controla una lista

#### **- USO DE VENTANAS Y CONTROLES.**

`close_window` → Cierra la ventana.  
`update_window` → Actualiza la ventana.  
`get_text` → Obtiene el texto de un control.  
`set_text` → Pone un Nuevo texto a un control.  
`size` → Obtiene y pone valores a una ventana.  
`position` → Obtiene y pone la posicion de un ventana.  
`change_style` → Cambia el estilo de una ventana.  
`enable_window` → Activa y desactiva una ventana

#### **- MENU.**

`menú` → Crea un elemento de menu.  
`modify_menu` → Modifica un elemento de menu.

#### **- CHECKBOX.**

`get_check_box_value` → Obtiene el valor de un checkbox.  
`set_check_box_value` → Asigna un valor a un checkbox.

#### **- EDIT.**

`add_text` → Agrega texto a un Edits.  
`get_selected_text` → Obtiene el texto seleccionado del edit.  
`get_selection` → Obtiene el texto seleccionado.



*set\_selection* → Pone el texto seleccionado dentro de un edit box.  
*copy\_in\_edit* → Copia el texto seleccionado dentro del clipboard.  
*paste\_in\_edit* → Copia el texto seleccionado dentro del clipboard.  
*cut\_in\_edit* → Corta el texto seleccionado

#### **- DIBUJO.**

*line* → Dibuja una línea.  
*fill\_polygon* → Dibuja un polígono.  
*bezier\_line* → Dibuja una curva Bézier.  
*ellipse* → Dibuja una elipse.  
*draw\_pie* → Dibuja un pie.  
*draw\_arc* → Dibuja un arco.  
*rect* → Dibuja un rectángulo.  
*round\_rect* → Dibuja un rectángulo con esquinas redondeadas.  
*text\_out* → Dibuja un rectángulo con texto.  
*bitmap\_image* → Crea un objeto tipo bitmap.  
*draw\_bitmap* → Dibuja un bitmap en la ventana.

#### **- FUENTES, COLORES, LAPIZ Y BACKGROUNDS.**

*pen* → Especifica el tipo de pen a usar.  
*brush* → Especifica el color de relleno.  
*color\_text* → Especifica el color de texto.  
*color\_text\_back* → Especifica el background de texto.  
*window\_brush* → Especifica el background de la ventana.  
*system\_color* → Obtiene el color del sistema.  
*rgb* → Crea un color por valor RGB.  
*select\_color* → Invoca una caja de dialogo la cual permite seleccionar un color.  
*select\_font* → Invoca una caja de dialogo la cual permite seleccionar el tipo de fuente.

## **8.2 USO DE LAS FUNCIONES PRINCIPALES.**

### ***FUNCION WINDOW( ).***

Sintaxis:

`window(Handle, Parent, Win_func(_), Title, X, Y, Width, Height)`

Ejemplo:

```
?-window(_,_,win_func(_),"Ejemplo",100,100,200,200).  
win_func(paint):-line(0,0,200,200),line(0,200,200,0).
```

### ***FUNCION EDIT( ).***

Sintaxis:

`edit(Handle, Parent, Edit_func(_),Text, X, Y, Width, Height)`



**EJEMPLO:**

```
?-window(_,_,win_func(_),"Edit control demo",100,100, 00, 200).  
win_func(init):-window_brush(_,rgb(0,0,255)),%blue  
    edit(E1,_,edit_func(_),"Start Value",10,10,100,50).  
edit_func(init):-color_text(_, rgb(0, 255, 0)), % green  
    color_text_back(_, rgb(255, 0, 0)). % red
```

***FUNCION BUTTON().***

**Sintaxis:**

**button(Handle, Parent, Button\_func(\_), Title, X, Y, Width, Height)**

**Ejemplo:**

```
?-window(_,_, win_func(_),"Button demo",100,100,200,200).  
win_func(init):-button(_,_,bfunc(_),"button",10,10,50,50).  
bfunc(press):-message("Message", "Button pressed.", i).
```

***FUNCION CHECK BOX***

**Sintaxis:**

**check\_box(Handle, Parent, Check\_func,Text, X, Y, Width, Height)**

**Ejemplo:**

```
R is get_check_box_value(Check_Box)  
set_check_box_value(Check_Box,Value)
```

***FUNCION RADIO\_BUTTON().***

**Sintaxis:**

**radio\_button(Handle, Parent, Radio\_func, Text, X, Y, Width, Height)**

**Ejemplo:**

```
R is get_check_box_value(Radio_Button)  
set_check_box_value(Radio_Button,Value)
```

***FUNCION STATIC().***

**Sintaxis:**

**static(Handle, Parent, Static\_func(\_), Title, X, Y, Width, Height)**

**Ejemplo:**

```
?-window(_,_,win_func(_),"Static Label demo",100,100,200, 200).  
win_func(init):-static(St,1,fail(_),"Static Label",X,Y,100,20).
```



### ***FUNCION BITMAP().***

Sintaxis:

`bitmap(Handle, Parent, Bitmap_func(_),Bitmap, X, Y)`

Ejemplo:

```
?-window(_,_,win_func(_),"Bitmap control demo",100,100,200,200).  
win_func(init):-bitmap(_,_,fail(_),default,50,50).
```

### ***FUNCION ICON().***

Sintaxis:

`icon(Handle, Parent, Icon_func(_), Icon, X, Y)`

Ejemplo:

```
?-window(_,_,win_func(_),"Icon control demo",100,100,200,200).  
win_func(init):-  
icon(_,_,fail(_),default,50,50).
```

### ***FUNCION ANIMATE().***

Sintaxis:

`animate(Handle,Parent,Icon_func(_),AVIfile,X,Y)`

Ejemplo:

```
?-window(_,_,win_func(_),"Animation controldemo",100,100,200,200).  
win_func(init):-animate(_,_,fail(_),"res/Dillo.avi",50,50).
```

### ***FUNCION LIST\_BOX().***

Sintaxis:

`list_box(Handle,Parent,List_func,Lable,X,Y,Width,Height)`

Ejemplo:

```
?-window(_,_,win_func(_),"List Demo",100,50,370,420).  
win_func(init):-  
list_box(G_List_Box,_,list_func,"Name",10,10,150,350),  
change_style(G_List_Box,0,0,0x200),  
insert_list_item(G_List_Box,end,"First",_).  
list_func(edit(Item,Text)):-  
yes_no("List Demo","Are you sure that you want to change this
```





```
text.",?),  
set_list_label(G_List_Box,Item,0,Text).
```

### ***FUNCION TREE\_BOX().***

Sintaxis:

**tree\_box(Handle,Parent,Tree\_func,\_,X,Y,Width,Height)**

Ejemplo:

```
?-window(_,_,win_func(_),"Tree Demo",100,50,370,420).  
win_func(init):-  
tree_box(G_Tree_Box,_,tree_func,_,10,10,150,350),  
change_style(G_Tree_Box,0,0,8),  
insert_tree_item(G_Tree_Box,_,root,first,"First",_,_).  
tree_func(edit(Item,Text)):-  
yes_no("Tree Demo","Are you sure that you want to change this  
text.",?),  
set_tree_item(G_Tree_Box,Item,Text,_,_).
```

### ***FUNCION GET\_TEXT().***

Sintaxis:

**get\_text(Window)**

Ejemplo:

```
?-window(_,_,win_func(_),"set_text/get_textdemo",100,100,300,200).  
win_func(init):-menu(normal,_,_,mfunc(_),"Change text").  
mfunc(press):-set_text("New Text",_).
```

### ***FUNCION ENABLE\_WINDOW().***

Sintaxis:

**enable\_window(Window,Mode)**

Ejemplo:

```
?-window(Win,_,win_func(_),"Enable Demo",100,100,300,200),  
enable_window(Win,0).
```

### ***FUNCION MENU().***

Sintaxis:

**menu(Type,Handle,Parent,Menu\_func(\_),Text)**

Ejemplo:

```
?-window(_,_,win_func(_),"menu demo",100,100,200,200).
```





rect(X1,Y1,X2,Y2)

Ejemplo:

```
?-window(_,_,win_func(_),"Ellipse demo",100,100,200,200).  
win_func(paint):-ellipse(0,0,200,200).
```

### ***FUNCION RECT().***

Sintaxis:

rect(X1,Y1,X2,Y2)

Ejemplo:

```
?-window(_,_,win_func(_),"Rectangle demo",100,100,200,200).  
win_func(paint):-  
rect(50,50,150,150),round_rect(70,70,130,130,10,10).
```

### ***FUNCION BITMAP\_IMAGE().***

Sintaxis:

bitmap\_image(Source,\_)

Ejemplo:

```
?-...,  
Bitmapisbitmap_image("image.bmp",_),  
draw_bitmap(10,10,Bitmap,_,_).
```

### ***FUNCION PEN().***

Sintaxis:

pen(PenWidth,Color)

Ejemplo:

```
?-window(_,_,win_func(_),"Pen Demo",100,100,200,200).  
win_func(paint):-  
pen(1,rgb(255,0,0)),line(0,0,200,200),  
pen(5,rgb(0,0,255)),line(0,200,200,0).
```

### ***FUNCION BRUSH().***

Sintaxis:

brush(Color,Type)

Ejemplo:

```
?-window(_,_,win_func(_),"Brush Demo",100,100,200,200).
```



```
win_func(paint):-  
brush(rgb(255,0,0)),rect(0,50,200,150),  
brush(rgb(0,0,255)),rect(50,0,150,200).
```

### ***FUNCION COLOR\_TEXT().***

Sintaxis:

**color\_text(Window,Color)**

Ejemplo:

```
?-window(_,_,win_func(_),"Edit control demo",100,100,200,200).  
win_func(init):-  
window_brush(_,rgb(0,0,255)),%blue  
edit(E1,_,edit_func(_),"StartValue",10,10,100,50).  
edit_func(init):-  
color_text(_,rgb(0,255,0)),%green  
color_text_back(_,rgb(255,0,0)).%red
```



## VENTAJAS DE PROLOG

Prolog, y en general los lenguajes de programación lógica, tienen las siguientes ventajas frente a los lenguajes clásicos (procedimentales):

- **Expresividad:** un programa (base de conocimiento) escrito en prolog puede ser leído e interpretado intuitivamente. Son, por tanto, más entendibles, manejables y fáciles de mantener.
- **Ejecución y búsqueda incorporada en el lenguaje:** dada una descripción prolog válida de un problema, automáticamente se obtiene cualquier conclusión válida.
- **Modularidad:** cada predicado (procedimiento) puede ser ejecutado, validado y examinado independiente e individualmente. Prolog no tiene variables globales, ni asignación. Cada relación está auto contenida, lo que permite una mayor *modularidad*, *portabilidad* y *reusabilidad* de relaciones entre programas.
- **Polimorfismo:** se trata de un lenguaje de programación sin tipos, lo que permite un alto nivel de abstracción e independencia de los datos (objetos).
- **Manejo dinámico y automático de memoria.**



## REFERENCIAS.

**Tutorial:**

[http://www.csupomona.edu/~jrfisher/www/prolog\\_tutorial/contents.html](http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html)

**Colin Baker 'home page':**

<http://perso.wanadoo.fr/colin.barker/index.htm>

**Swin-Prolog:**

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

**Strawberry Prolog:**

<http://www.dobrev.com/>

**El club de los caminantes:**

<http://caminantes.metropoliglobal.com/web/informatica/prolog.htm>

**Programación lógica:**

[http://www.geocities.com/v.iniestra/apuntes/pro\\_log/](http://www.geocities.com/v.iniestra/apuntes/pro_log/)

**Programing in Prolog:**

<http://cwis.kub.nl/~fdl/general/people/rmuskens/courses/prolog/>

**Tecnológico de Monterrey Prolog:**

<http://w3.mor.itesm.mx/~esucar/IA/prolog.html>

**Ingenieros en infomatica.org:**

[http://ingenieroseninformatica.org/recursos/tutoriales/sist\\_exp/cap5.php](http://ingenieroseninformatica.org/recursos/tutoriales/sist_exp/cap5.php)