Transparencias de Programación Lógica y Funcional

Ingeniería Superior en Informática

Prof. Jesús Almendros

Despacho 1.48.

CITE III

Programación Lógica y Funcional.

Temario:

1.-Introducción.

2.-Programación Lógica: Claúsulas de Horn, Hechos, Consultas, Reglas, Ejemplos.

3.-Prolog: Términos Prolog, Árbol de Búsqueda Prolog, Números Naturales, Listas, Árboles.

4.-Teoría: Resolución, Resolvente, Derivación, Refutación, Árboles SLD.

4.-Programación en Prolog: Terminación, Soluciones, Metapredicados, Corte, Negación, Lectura y Escritura.

5.-Introducción a la Programación Lógico-Funcional.

Programación Lógica y Funcional.

Bibliografía:

- Sterling, Leon, The Art of prolog : advanced programming techniques, The Mit Press, 1997.
- Apt, K. R. From logic programming to Prolog, Prentice Hall, 1997.
- Clocksin, W.F. Programación en Prolog / W.F. Clocksin, C.S. Mellish. Gustavo Gili, D.L. 1993.
- Berk, A.A, Prolog : programación y aplicaciones en Inteligencia Artificial, Anaya Multimedia, D.L. 1986.

Introducción. Historia.

- La **programación lógica** es un modelo de cómputo basado en la **lógica matemática**.
- Resolución. (1965)
- Resolución-SL. *Kowalski* (1971)
- PROLOG. (1972)

Introducción. Fifth Generation Computer Systems

La programación lógica recibió gran ímpetu en 1981 cuando el gobierno japonés inició un ambicioso proyecto dedicado a la **construcción de computadoras optimizadas** para ejecutar programas escritos en lenguajes de programación lógica

Introducción. Lógica y Control

- La idea central del paradigma de programación lógica puede sintetizarse en la siguiente **ecuación informal de Kowalski**:

algoritmo = lógica + control

- La programación lógica incluye el **control** como componente del módelo de cómputo. Esto permite **liberar** al programador de la especificación de **construcciones de control**.

Introducción. Programación declarativa

En el paradigma de programación lógica un programa es un **conjunto de enunciados** que declaran las propiedades del cómputo. Por esta razón, la programación lógica es conocida como **programación declarativa**.

Introducción. Aplicaciones

Adecuada para:

- Procesamiento de Lenguaje Natural
- Bases de datos
- Compiladores
- Sistemas expertos
- Inteligencia Artificial

Introducción. Líneas de Investigación

- Programación lógica pura
- Programación lógica paralela y distribuida
- Programación lógica inductiva
- Programación lógica con restricciones
- Integración con otros paradigmas: OO, programación funcional.

Programación lógica

La programación lógica es un paradigma de programación basado en la **representación de axiomas (cláusulas de Horn)** y en un **mecanismo de inferencia** conocido como **resolución-SLD**.

Programación Lógica. Claúsulas de Horn

- La Programación Lógica está basada en la **lógica de primer orden** o lógica de predicados.
- Un predicado o su negación es llamado una **literal**.
- Una **hecho** es un literal positivo
- Una **claúsula o regla** es una fórmula que especifica una **consecuencia lógica**: si se cumplen una serie de literales se puede afirmar un cierto literal positivo

Programación Lógica. Claúsulas de Horn.

- Una **claúsula** es una fórmula de la forma:

$$\forall\, x1\ x2 \ldots xn.\ (L1 \vee \ldots \vee Lm)$$

donde x1,...,xn son variables y L1...Lm son literales.

- Esta fórmula se puede transformar en una **forma clausal** como sigue:

$$a1 \leftarrow b1,\ldots,br$$

$$a2 \leftarrow b1,\ldots,br$$

$$\ldots\ldots$$

$$ak \leftarrow b1,\ldots,br$$

donde los ai (**conclusiones**) son los **Li positivos** y los bj (**premisas**) son los **Lj negativos**

Programación Lógica. Claúsula negativa u Objetivo

- Si no hay bi se llama **hecho** en vez de **claúsula o regla**:

$$ai \leftarrow$$

- Cuando no existe la conclusión (no existe ai) entonces se trata de una **claúsula negativa u objetivo** y sirve cómo **consulta** al programa:

$$\leftarrow \; b1,...,br$$

Programación Lógica. Cláusula vacía

Cuando tanto el conjunto de premisas como el conjunto de conclusiones son el vacío entonces hablamos de la **cláusula vacía** y se denota por □.

Programación Lógica. Ejemplos. Hechos.

ascendente(patricia,alberto).

ascendente(antonio, alberto).

ascendente(antonio, luisa).

ascendente(alberto, ana).

ascendente(alberto, maria).

ascendente(maria, juan).

- Especifica los ascendentes familiares de un grupo de personas.
- Son hechos compuestos de un **predicado** "ascendente" y pares de nombres que en programación lógica son **constantes** (**comienzan en minúscula**).

Programación Lógica. Ejemplos. Consultas Prolog.

?- ascendente(alberto, maria).

Yes

?- ascendente(luisa, ana).

Yes

?- ascendente(juan, pedro).

No.

?- ascendente(X, luisa).

X=antonio

Programación Lógica. Ejemplos. Consultas Prolog.

?- ascendente(alberto, X).

X=ana;

X=maria;

?- ascendente(X, Y).

X=patricia, Y=alberto;

X=antonio, Y=alberto;

?- ascendente(Y, ana), ascendente(X, Y).

?.- ascendente(antonio, X), ascendente(X, Y).

Programación Lógica. Ejemplos.

```
mujer(patricia).
hombre(antonio).
hombre(alberto)
mujer(luisa).
mujer(ana).
mujer(maria).
hombre(juan).
```

Programación Lógica. Ejemplos. Reglas Prolog.

```
hijo(X,Y):- ascendente(Y,X), hombre(X).
madre(X,Y):- ascendente(X,Y), mujer(X).

hermana(X, Y) :- ascendente(Z, X),
                 ascendente(Z, Y),
                 mujer(X).
abuela(X, Z):- ascendente(X,Y),
               ascendente(Y,Z),
               mujer(X).
```

Programación Lógica. Ejercicios.

- Definir la relación *nieto*
- Definir la relación *tía*
- Definir la relación *primo hermano*
- Definir la relación *abuela paterna*
- Definir la relación *bisabuelo*

Programación Lógica. Mas Ejemplos. Recursión.

```
antepasado(X, Z):- ascendente(X, Z).

antepasado(X, Z) :- ascendente(X, Y),
                    antepasado(Y, Z).
```

Prolog. Términos Prolog.

ascendente(pedro,juan).

suma(1,1,2).

append(X,[Y|L]).

horario(plf,fecha(lunes,5)).

Prolog. Términos Prolog. Unificación

X=? juan

[X|L]=?[1,2]

[1|L]=?[X,2,2]

fecha(lunes,X)=?fecha(Y,Z)

Prolog. Términos Prolog. Substitución.

Sustitución: Vínculo de variables lógicas (Variables prolog) a términos prolog.

X/juan

X/1, L/[2]

X/1, L/[2,2]

Y/lunes, X/Z

Prolog. Términos Prolog.

```
profesor(Profesor,Curso) :-
    curso(Curso,Horario,Profesor,Lugar).


enseña(Profesor,Dia) :-
    curso(Curso,horario(Dia,Inicio,Fin),
       Profesor,Lugar).
```

Prolog. Términos Prolog.

```
duracion(Curso,Horas) :-
      curso(Curso,horario(Dia,Inicio,Fin), Profesor,Lugar)
         Horas is Fin-Inicio.


ocupado(Salon,Dia,Hora) :-
      curso(Curso,horario(Dia,Inicio,Fin), Profesor,Salon),
         Inicio<=Hora, Hora<=Fin.
```

Prolog. Términos Prolog. Ejercicios.

Definir las reglas:

- lugar(Curso,Edificio).
- ocupado(Profesor,Hora).
- no_pueden_reunirse(Prof1,Prof2).
- conflicto_programación(Hora, Lugar,Curso1,Curso2).

Prolog. Listas.

- Una **lista** es una estructura de datos binaria en la cual el primer argumento es un **elemento**, y el segundo argumento es una **lista**.
- Para terminar cómputos recursivos sobre una lista es necesario un símbolo de constante. La **lista vacía**, llamada *nil*, es denotada por el símbolo [].
- Históricamente, la constante de utilizada para denotar listas es “.”. El término .(X,Y) es usualmente denotado por **[X|Y]**. Sus componentes tienen nombres especiales: X es llamado la **cabeza** y Y el **resto**.

Prolog. Listas. Operaciones.

- **Pertenencia a una lista**

```
member(X,[X|Tail]).
member(X,[Head|Tail]) :-
        member(X,Tail).
```

- **Concatenación de listas**

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

member(X,L) :- append(L1,[X|L2],L).
```

Prolog. Listas. Operaciones.

- **Añadir un elemento a una lista**

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

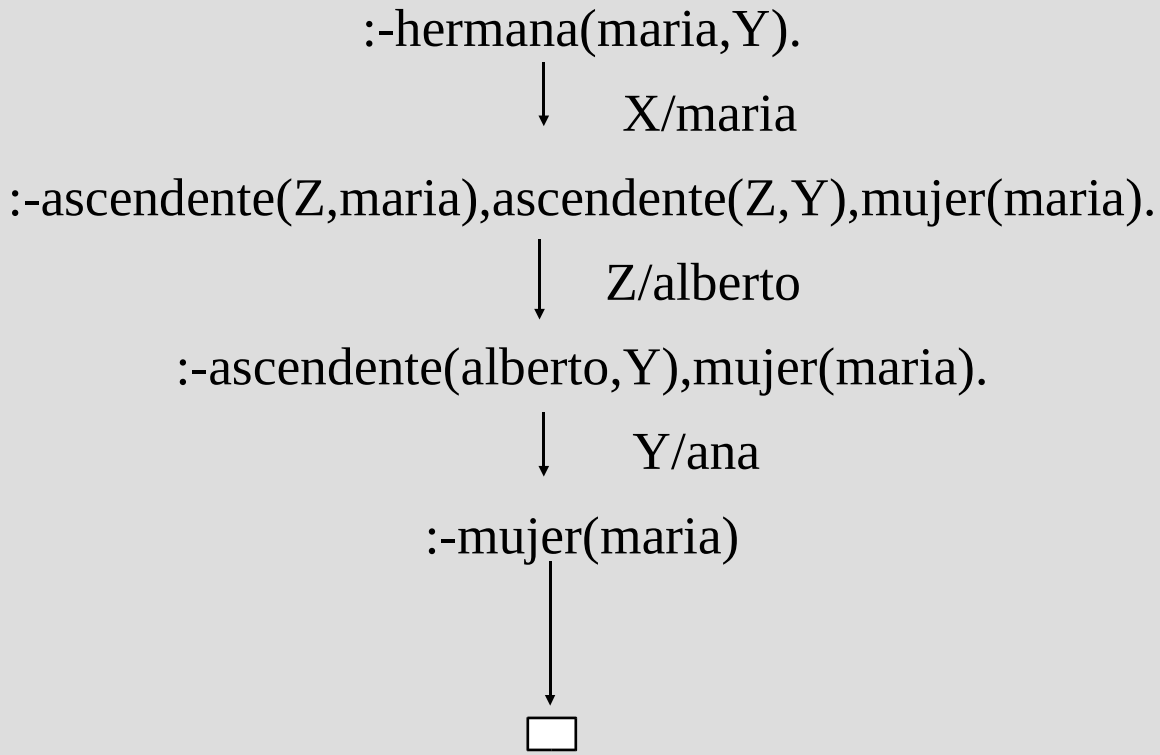
- **Borrar un elemento de una lista**

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail1]) :-
        del(X,Tail,Tail1).
```

- **Sublista de una lista**

```
sublist(S,L) :-append(L1,L2,L), append(S,L3,L2).
```

Prolog. Árbol de Búsqueda Prolog.

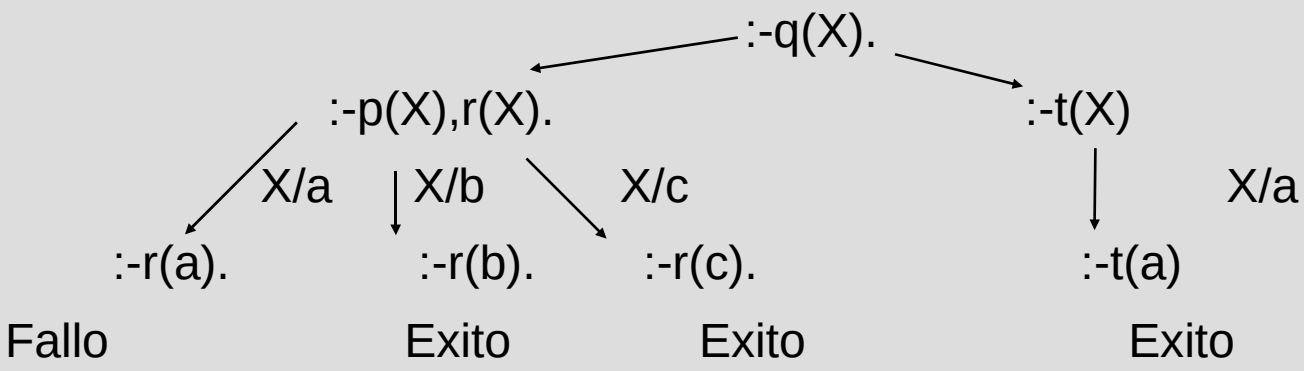


Prolog. Árbol de Búsqueda Prolog.

Programa prolog:

```
p(a).
p(b).
p(c).
r(b).
r(c).
s(c).
t(a).
q(X):-p(X),r(X).
q(X):-t(X).
```

Prolog. Árbol de Búsqueda Prolog.



Prolog. Árbol de Búsqueda Prolog

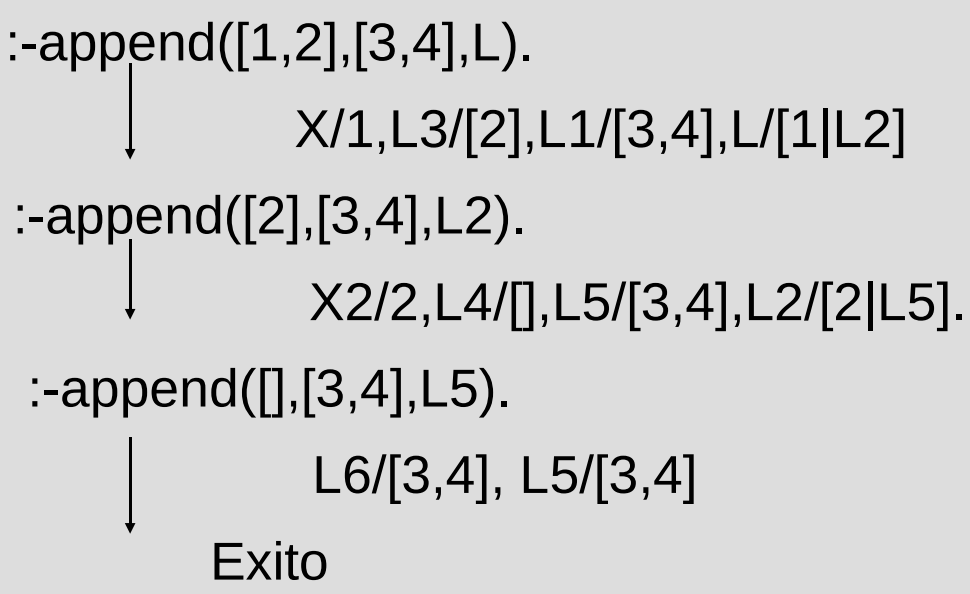
Recorrido:

- Reglas de arriba hacia abajo
- Cada regla de izquierda a derecha
- Soluciones: Composición de las sustituciones calculadas.

Prolog. Árbol de Búsqueda Prolog.

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

Prolog. Árbol de Búsqueda Prolog.



Solución o Respuesta: L=[1,2,3,4]

Teoría. Programa lógico.

- Un **programa lógico** (o programa) es un conjunto finito no vacío de cláusulas de programa. Un **cómputo** es una **deducción de conclusiones** del programa.

- Esta idea puede sintetizarse en las siguientes ecuaciones informales:

programa = conjunto de axiomas

cómputo = demostración constructiva de un objetivo a partir del programa

Teoría. Interpretaciones.

Existen dos maneras de interpretar la cláusula

A <- B1... Bn

- *A* es verdad si B1 y ... y Bn son verdad. Esta interpretación es llamada **interpretación declarativa.**
- para resolver A resolver hay que resolver B1 y... Bn.

Esta interpretación es llamada **interpretación procedural** y es la que distingue a la programación lógica de los lenguajes de primer orden.

Teoría. Resolución-SLD.

El cómputo en un programa lógico es realizado por medio de la combinación de dos mecanismos:

reemplazo y unificación

Teoría. Derivación.

Mediante la iteración de este proceso de reemplazo se obtiene una secuencia de **resolventes** la cual es llamada una **derivación**.

Teoría. Refutación.

Una derivación puede ser **finita** o **infinita**. Si su última cláusula es la cláusula vacía entonces es llamada una **refutación** del objetivo.

Teoría. Refutación.

Es decir, a partir del **programa *P*** y un objetivo **O**, una refutación deriva una contradicción:
:-O es “no se cumple O”
y se deriva **:-true** que es “no cierto”

Teoría. Cómputo.

Un programa lógico se utiliza para realizar un cómputo. Esto se consigue mediante el **uso repetido del algoritmo de unificación**, que produce **asignaciones de términos a las variables**.

Teoría. Resolución.

- Sea **P** un programa y **O** un objetivo **:-A1,...,An**
- Sea R una regla de P de la forma **B:-C1,...,Cm**
- Supongamos que **Ai y B** unifican con un unificador más general θ

Teoría. Resolvente

Entonces a

:-(A1,...,Ai-1,C1,...,Cm,Ai+1,...,An)θ

se le llama **resolvente**

del objetivo O y la regla R bajo θ

Teoría. Resolvente.

Así pues, un **resolvente es obtenido** mediante la aplicación de los siguientes pasos:

a) Seleccionar un predicado A_i

b) Intentar unificar a *B* y A_i

Teoría. Resolvente

- **Si (b) tiene éxito** entonces Ai se sustituye por C1,...,Cm aplicando θ a todo.
- **Si (b) no tiene éxito** entonces se prueba con otra regla R
- **Si no hay más reglas** la derivación es **fallida**

Teoría. Derivación-SLD.

- Una **derivación-SLD** de P y O es una secuencia de objetivos **O1,O2,…,Ok,** donde O1 es O, una secuencia de **variantes** de reglas **R1,…Rk** y una secuencia de sustituciones θ1,….θk tal que **Oi+1** es resolvente de **Oi** y **Ri** bajo θi
- Una **variante de una regla** siempre tiene variables distintas a las del objetivo

Teoría. Refutación-SLD.

Cuando uno de los **subojectivos** Oi es **vacío** entonces es la última cláusula de la derivación. Una tal derivación es llamada **refutación-SLD**. Se dice que una **derivación-SLD** ha **fallado** si ésta es **finita y no es una refutación**.

Teoría. Regla de Selección.

En cada paso de una **derivación-SLD** son requeridas dos selecciones para la construcción de un nuevo resolvente:

- **Selección del predicado elegido**
- **Selección de la regla**

Teoría. Árboles-SLD.

Un **árbol-SLD** es un árbol tal que

- Sus ramas son derivaciones-SLD
- Cada nodo **O** tiene un descendiente para cada regla ***R*** de ***P*** tal que el predicado elegido de ***Ai*** se unifica con la conclusión de **R**.

Teoría. Árboles SLD.

Así que el problema se reduce a una **búsqueda sistemática en el árbol-SLD.** Algunas implementaciones explotan algún ordenamiento de las reglas de programa, por ejemplo, el ordenamiento textual en el programa.

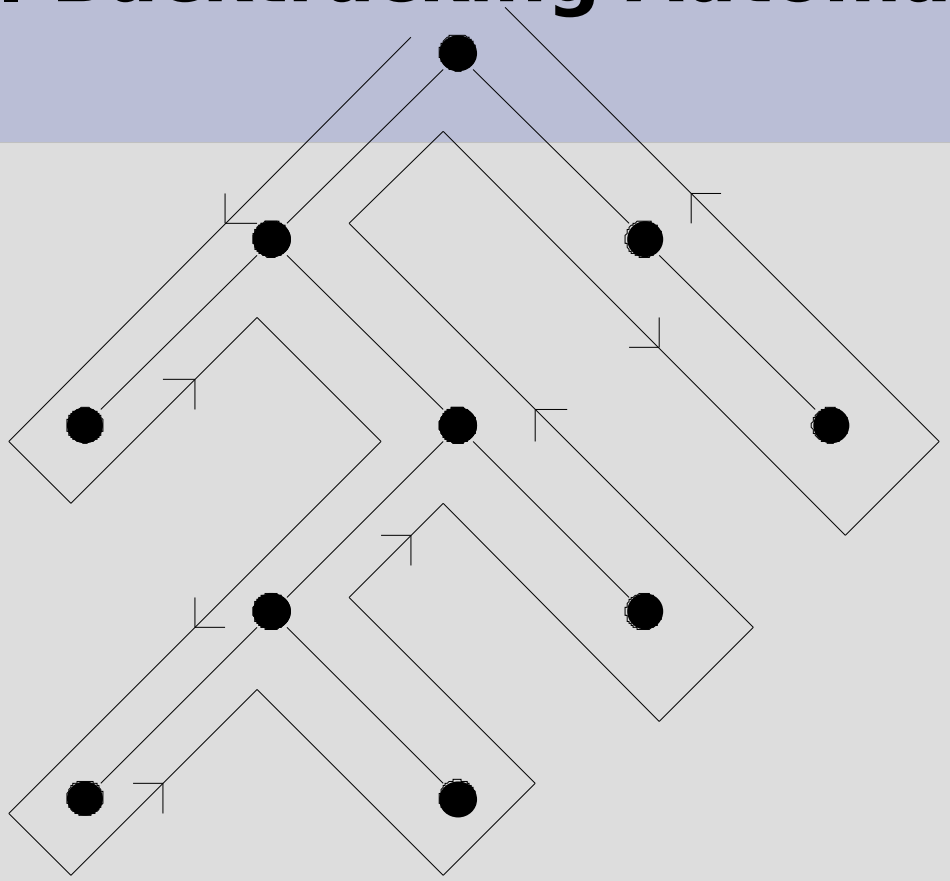
Teoría. Árboles SLD.

Esto impone el ordenamiento a los arcos que descienden de un nodo del árbol-SLD. El árbol es recorrido entonces mediante una **estrategia primero en profundidad** siguiendo este ordenamiento. Para un árbol-SLD finito esta estrategia es **completa**.

Teoría. Backtracking automático

Cuando un nodo hoja del árbol-SLD es alcanzado, el **recorrido continua por backtracking** al último nodo precedente del camino con ramas no exploradas.

Teoría. Backtracking Automático.



Búsqueda primero en profundidad

Prolog. Recursividad.

```
arco(a,b).
arco(b,c).
arco(c,d).
arco(a,c).
arco(c,e).
camino(X,Y):-arco(X,Y).
camino(X,Y):-arco(X,Z),camino(Z,Y).

:-camino(a,X).
X=b;
X=c;
...
```

Prolog. Números Naturales.

```
nat(0).
nat(s(X)):-nat(X).

nat(0).
nat(N):-nat(M),N is M+1.
```

Prolog. Operadores aritméticos.

- + suma
- - resta
- * multiplicación
- / división
- mod módulo
- is igual a (o asignación)

Prolog. Operadores aritméticos.

Tamaño de una lista:

```
length([],0).
length([X|Tail],N) :-
    length(Tail,N1),
    N is 1 + N1.
```

Prolog. Operadores relacionales.

- X > Y X es mayor que Y
- X < Y X es menor que Y
- X >= Y X es mayor o igual a Y
- X =< Y X es menor o igual a Y
- X =:= Y X es igual a Y
- X =\=Y X es diferente a Y

Prolog. Números Naturales.

```
leq(0,X).
leq(s(X),s(Y)):- leq(s(X),s(Y)).

:-leq(0,X).
Yes.

suma(0,X,X):-nat(X).
suma(s(X),Y,s(Z)):-suma(X,Y,Z).
:-suma(s(0),X,s(s(0))).
X=s(0).
:-suma(X,Y,s(s(0))).
X=0,Y=s(s(0)).
X=s(0),Y=s(0).
.....
```

Prolog. Números Naturales.

```
factorial(0,1).
factorial(N,F):-M is N-1,factorial(M,F1),F is F1*N.

minimo(M,N,M):-M<=N.
minimo(M,N,N):-N<M.

mcd(M,M,M).
mcd(M,N,K):-M>N, R is M-N, mcd(R,N,K).
mcd(M,N,K):-M<N,R is N-M, mcd(M,R,K).
```

Prolog. Listas.

```
list([]).
list([X|Xs]):-list(Xs).

member(X,[X|Xs]).
member(X,[Y|Xs]):-member(X,Xs).

:-member(X,[1,2,3])
X=1;
X=2;
X=3;
```

Prolog. Listas.

```
prefijo([],Ys).
prefijo([X|Xs],[X|Ys]):-prefijo(Xs,Ys).

:-prefijo(X,[1,2,3]).
X=[];
X=[1];
X=[1,2];
X=[1,2,3].

sufijo(Xs,Ys).
sufijo(Xs,[Y|Ys]):-sufijo(Xs,Ys).
```

Prolog. Listas.

```
member(X,Xs):-sublista([X],Xs).

sublista(Xs,Ys):-prefijo(Ps,Ys),sufijo(Xs,Ps).

sublista(Xs,Ys):-prefijo(Xs,Ss),sufijo(Ss,Ys).

sublista(Xs,Ys):-prefijo(Xs,Ys).
sublista(Xs,[Y|Ys]):-sublista(Xs,Ys).

sublista(Xs,AsXsBs):-append(As,XsBs,AsXsBs),
                     append(Xs,Bs,XsBs).
```

Prolog. Listas.

```
reverse([],[]).
reverse([X|Xs]):-reverse(Xs,Ys),append(Ys,[X],Zs).

reverse(Xs,Ys):-reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys):-reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

Parámetro Acumulador para la recursión final.

Prolog. Listas.

```
delete([X|Xs],X,Ys):-delete(Xs,X,Ys).
delete(([X|Xs],Z,[X|Ys]):-X\==Z,delete(Xs,Z,Ys).
delete([],X,[]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

Prolog. Listas.

```
ordenar(Xs,Ys):-permutacion(Xs,Ys),
                ordenada(Ys).


ordenada([X]).
ordenada([X,Y|Ys]):-X<=Y,ordenada([Y|Ys]).


permutacion(Xs,[Z|Zs]):-
                select(Z,Xs,Ys),
                permutacion(Ys,Zs).
permutacion([],[]).
```

Prolog. Listas.

```
ordenar([X|Xs],Ys):-
                ordenar(Xs,Ys),
                insertar(X,Zs,Ys).

ordenar([],[]).


insertar(X,[],[X]).
insertar(X,[Y|Ys],[Y|Zs]):-X>Y,insertar(X,Ys,Zs).
```

Prolog. Listas.

```
quicksort([X|Xs],Ys):-
    particion(Xs,X,Menores,Mayores),
    quicksort(Menores,Mns),
    quicksort(Mayores,Mys),
    append(Mns,[X|Mys],Ys).

particion([X|Xs],Y,[X|Ls],Bs):
    X<=Y, particion(Xs,Y,Ls,Bs).
particion([X|Xs],Y,Ls,[X|Bs]):-
    X>Y, particion(Xs,Y,Ls,Bs).
particion([],Y,[],[]).
```

Prolog. Árboles.

```
arbol_binario(vacio).
arbol_binario(nodo(Elem,HI,HD)):-
        arbol_binario(HI),arbol_binario(HD).


miembro(X,nodo(X,HI,HD)).
miembro(X,nodo(X,HI,HD)):-
                miembro(X,HI).
miembro(X,nodo(X,HI,HD)):-
                miembro(X,HD).
```

Prolog. Árboles.

```
isomorfo(vacio,vacio).

isomorfo(nodo(X,HI1,HD1),nodo(X,HI2,HD2)):-
        isomorfo(HI1,HI2),
        isomorfo(HD1,HD2).

isomorfo(nodo(X,HI1,HD1),nodo(X,HI2,HD2)):-
    isomorfo(HI1,HD2),
    isomorfo(HI2,HD1).
```

Prolog. Árboles.

```
preorden(nodo(X,L,R),Xs):-
    preorden(L,Ls),
    preorden(R,Rs),
    append([X|Ls],Rs,Xs).
preorden(vacio,[]).

postorden(nodo(X,L,R),Xs):-
   postorden(L,Ls),
   postorden(R,Rs),
   append(Rs,[X],Rs1),
   append(Ls,Rs1,Xs).
postorden(vacio,[]).
```

Programación en Prolog. Terminación.

casado(juan,marta).

casado(pedro,enrique).

casado(X,Y):-casado(Y,X).

NO TERMINA

estan_casados(X,Y):-casado(X,Y).

estan_casados(X,Y):-casado(Y,X).

SI TERMINA

padre(carlos,paquito).

hijo(enrique,eufrasio).

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

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

Programación en Prolog. Soluciones.

madre(X,Y):-ascendente(X,Y),mujer(X).

:-madre(pepe,Z).

SOLUCIONES REDUNDANTES:

minimo(X,Y,X):-X<=Y.

minimo(X,Y,Y):-Y<=X.

MEJOR:

minimo(X,Y,Y):-Y<X.

Programación en Prolog. Soluciones.

```
member(X,[X|L]).
member(X,[Y|L]):-member(X,L).
```

MEJOR:

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

Programación en Prolog. Metapredicados.

Term=..L.

:- f(a,b,c)=..Lista.

Lista=[f,a,b,c]

:- Termino=..[ascendente,ana,juan].

Termino=ascendente(ana,juan).

Programación en Prolog. Metapredicados.

functor(Term,F,N).

:- functor(f(a,b,c),Fun,Arity).

Fun=f

Artity=3

arg(N,Term,A).

:- arg(2,f(a,g(b),c),A).

A= g(b)

Programación en Prolog. Metapredicados.

```
:- functor(F,fecha,3),arg(1,F,11),arg(2,F,abril),arg
   (3,F,1996).

F=fecha(11,abril,1996).
```

Programación en Prolog. Metapredicados.

- **Var:** comprueba si es variable
- **Nonvar:** comprueba si no es variable
- **Integer:** si es un entero
- **Float:** si es un real
- **Number:** si es un numero entero o real
- **String:** si es un string
- **Atomic:** si es una constante, entero, real o string
- **Compound:** si es compuesto

Programación en Prolog. Metapredicados.

```
plus(X,Y,Z):-nonvar(X),nonvar(Y), Z is X+Y.
plus(X,Y,Z):-nonvar(X),nonvar(Z), Y is  Z-X.
plus(X,Y,Z):-nonvar(Y),nonvar(Z), X is Z-Y.

:-plus(X,2,5).

:-plus(2,X,5).
```

OTRO CASO:

```
suma(0,X,X):-number(X).
```

Programación en Prolog. Metapredicados.

```
member(X,X).
member(X,T):-compound(T),
                    functor(T,Op,Arity),
                    member_Args(X,T,1,Arity).

member_Args(X,T,I,Arity):-I<=Arity,
                                arg(T,I,ArgI),
                                member(X,ArgI).

member_Args(X,T,I,Arity):-I<Arity, J is I+1,
                                member_Args(X,T,J,Arity).
```

Programación en Prolog. Metapredicados.

```
:-member(1,[1,2,3]).

:-member(1,f(1,g(2),h(3))).

:-member(1,nodo(2,nodo(1,vacio,vacio),vacio))).
```

Programación en Prolog. Metapredicados.

- assert(C).

  Añade la regla C al programa.

- retract(C).

  Elimina la regla C del programa.

Programación en Prolog. Metapredicados.

?- assert(p(a)), assertz(p(b)), asserta(p(c)).

?- p(X).

X=c;

X=a;

X=b;

no

asserta(C).

Añade la regla C al inicio del programa.

assertz(C).

Añade la regla C al final del programa.

Programación en Prolog. Metapredicados.

Findall(X,p(X,Y,Z),L).

Bagof(Z,X^Y^p(X,Y,Z),L).

P(a,b,c).

P(a,e,d).

Solo una respuesta, como el findall.

Bagof(Z,X^p(X,Y,Z),L).

>Y=b, Z=[c].

>Y=e, Z=[d].

Setof(X,p(X),L).

No repite soluciones en L.

Programación en Prolog. Corte.

En general, un **árbol-SLD** puede tener ramas fallidas y muy pocas o sólo una rama exitosa (refutación). Es posible incluir **información de control** en el programa para prevenir la construcción de ramas fallidas.

Programación en Prolog. Corte.

Para controlar la búsqueda se introduce el **corte** Prolog. Sintácticamente el **corte** es denotado por "**!**" y es colocado en las condiciones de una regla como uno de sus predicados.

Programación en Prolog. Corte. Ejemplos.

Máximo de dos números

```
max(X,Y,X) :- X >= Y.

max(X,Y,Y) :- X < Y.
```

Este programa se escribe en Prolog **utilizando corte** como:

```
max(X,Y,X) :- X >= Y, !.

max(X,Y,Y).
```

Programación en Prolog. Corte. Ejemplos.

Pertenencia con solución única

```
member(X, [X|L]) :- !.

member(X, [Y|L]):-member(X,L).
```

Programación en Prolog. Corte.
Efectos del corte.

- Divide la condición de la regla en dos partes donde el **backtracking** es realizado por separado.
- Una vez que el corte llega a ser el predicado elegido para resolución, ya **no es posible realizar backtracking** hacia las literales que aparecen en la **parte izquierda del corte**. Sin embargo, **la parte derecha se ejecuta normalmente**.

Programación en Prolog. Corte.Tipos de Cortes.

- **Corte Verde:** Si el corte **elimina únicamente las ramas fallidas** del árbol-SLD éste no tiene influencia en el significado del programa.
- **Corte Rojo:** Sin embargo, es posible que el corte elimine algunas refutaciones-SLD, eliminando soluciones. **Razón: se ha usado para repetición de soluciones**.

Programación en Prolog. Corte. Ejemplos.

```
padre(X,Y) :- ascendente(X,Y),
                    hombre (X), !.
ascendente(antonio, alberto).
ascendente(patricia, alberto).
hombre(antonio).
hombre(alberto).
```

Programación en Prolog. Corte. Ejemplos.

```
orden(Xs,Ys) :-
   append(As,[X,Y|Bs],Xs),
   X>Y,!,
   append(As,[Y,X|Bs],Xs1),
   orden(Xs1,Ys).
orden(Xs,Xs):-ordenada(Xs).
```

Programación en Prolog. Negación.

Se dice que un árbol-SLD es **de fallo finito** si es finito y no contiene **ninguna refutación**. Lo que es lo mismo, todas las ramas del árbol son derivaciones que han fallado.

Programación en Prolog. Negación. Implementación.

El operador *not* es implementado en PROLOG con las reglas. **En SWI-Prolog es +**.

```
not(X) :- X, !, fail.
not(X).
```

Programación en Prolog. Negación. Ejemplos.

```
estudiante_soltero(X):-
        not casado(X),estudiante(X).
estudiante(roberto).
estudiante(marta).
casado(marta).
```

Programación en Prolog. Negación. Ejemplos.

```
disjuntas(L,L2):-
    not(member(X,L),member(X,L2)).
```

Programación en Prolog. Lectura y Escritura.

read(X).

write(X).

Sin backtracking, ojo!

```
p(a).
p(b).
q(X):-p(X),write(X).
:-q(X).
a
```

Programación en Prolog. Lectura y Escritura.

Solución:

```
q(X):-p(X),write(X),fail.
```

Más:

Readln,Writeln,nl.

Introducción a la Programación Lógico-Funcional

- Uso de **Funciones** en lugar de Predicados
- **Reglas y hechos como Funciones**.
- Manejo de **Ecuaciones** en lugar de Predicados.

Introducción a la Programación Lógico-Funcional

```
factorial(0):=1.
factorial(N):=N*factorial(N-1).

append([],L):=L.
append([X|L],L1):=[X|append(L,L1)].
```

Introducción a la Programación Lógico-Funcional

Lenguajes Lógico-Funcionales:

- CURRY
- TOY
- BABEL
- MERCURY
- OZ

Introducción a la Programación Lógico-Funcional

Características Nuevas:

- Manejo de Tipos
- Datos Infinitos.
- Funciones No Estrictas, Parciales.
- Funciones Perezosas.

Introducción a la Programación Lógico-Funcional

from(N):=[N|from(s(N))].
from(0) vale [0,s(0),s(s(0)),....]

member(X,[X|L]):=true.
member(X,[Y|L]):=member(X,L).

:-member(s(s(0)),from(0))==true.

Introducción a la Programación Lógico-Funcional

```
first(X,Y):=X.
g(0):=1.
g(1):=2.

:-first(0,g(2))==X.
X=0.
```

Introducción a la Programación Lógico-Funcional

```
take(0,L):=[].
take(N,[X|L]):=[X|take(N-1,L)]
                        :- N>0 == true.
```

Introducción a la Programación Lógico-Funcional

```
member(X,L):=true :-
        append(L1,[X|L2])==L.
```

```
:-member(2,[3,2,5])==true.
:-member(2,[2,5])==true.
Yes.
```

Mecanismo de Ejecución: Narrowing (Estrechamiento)

Introducción a la Programación Lógico-Funcional

:-member(2,[3,2,5])==true

:-append(L1,[2|L2])==[3,2,5]

L1=[3], L2=[5]

:- true==true

Yes.

Introducción a la Programación Lógico-Funcional

- Narrowing.
- Árbol de Busqueda como en Prolog.
- Resolución de Ecuaciones
- Respuestas Finitas