```
mortal(P):- hombre(P). % regla

hombre(adan). % hecho


% implementar los numeros naturales

    %nat(N). N es cierto si es un numero naturales
    %s(x)    el numero sucesor al numero natural X

    nat(0). % cero es un numero natural
    nat(s(X)) :- nat(X).    %paso recursivo.

% ejercicio 2: implementar par de un numero
%par(N) es cierto si N es un numero

    par(0).  % paso base
    par(s(s(X))):-par(X). %regla recursiva

%ejercicio3: implementar impar de un numero
%impar(N) es cierto si N es numero impar
%s(X) el numero sucesor al numero natural X

impar(s(0)). %Hecho. paso base
impar(s(X)):- impar(X). %regla recursiva



%ejercicio4: implementar suma
%suma(X,Y,Z). es cierto si X+Y=Z
%sintaxis semántica

suma(0,X,X).  % paso base -> elemento neutro de la suma
suma(s(X),Y,s(Z)):- suma(X,Y,Z). %regla recursiva

%ejercicio5: implementar la multiplicacion
%mult(X,Y,Z). es cierto si X*Y=Z
%pista:  emplear propiedad distributiva de la suma para implementar
% la multiplicacion, es decir, expresar la multiplicacion como sumas sucesivas

mult(0,X,0).  % paso base -> elemento neutro de la suma
mult(s(0),Y,W):- mult(X,Y,Z), suma(Z,Y,W). %regla recursiva

% .       AND logico
% ..      OR logico
% not(...)NOT logico

%ejercicio6: implementar la exponencial
```

%exp(X,Y,Z). es cierto si X^Y=Z

exp(0,X,0). % paso base
exp():- exp().%regla recursiva

PRÁCTICA 1

% El paradigma de programación declarativo, se fundamenta en la lógica de predicados o de primer orden. En este paradigma se especifica QUE hacer para resolver un problema y no COMO resolverlo. Que es responsabilidad del programador establecer la SINTAXIS(notación) y la SEMÁNTICA(significado) de los programas. Escribir un programa en Prolog consiste en representar el conocimiento acerca de objetos y sus relaciones y sus reglas.
Los programasen Prolog se componen de cláusulas
Las cláusulas son la BASE del CONOCIMIENTO del programa y estan conformadas por Hechos y Reglas.
Hechos: Son cláusulas sin cuerpo que muestran una relación explícita entre objetos y declaran los valores que son verdaderos para un predicado
Reglas: Cuando la verdad de un hecho depende de la verdad de otro hecho o de un grupo de hechos se usa una regla
Las reglas son cláusulas con una cabeza y un cuerpo que declaran las condiciones para que un predicado sea cierto con una implicación que puede relacionar hechos para dar valores de verdad a un predicado

Prolog no tiene ciclos, emplea la recursividad y la unificación (matching) como mecanismo de inferencia para obtener una solución.

caso base: define cuando termina la inferencia.
caso recursivo: que define como descomponer el caso general para llegar al caso base.
Si existe mas de una solución el programa hace un backtracking, que es un mecanismo de búsqueda en profundidad que permite al intérprete explorar diferentes caminos para encontrar una solución al problema.

Las listas en prolog, se emplean para representar secuencias de elementos, ofreciendo una forma flexible de organizar lainformación.

Una lista es una estructura de elementos entre corchetes separados por comas, donde no es necesario especificar el tipo de sus elementos.

cabeza, se denota por Car: Contents of the Address part of Register Number
colca, se denota por Cdr: Contents of the Decrements part of Register Number

Cadr, que hace referencia al 2º elemento de una lista
Caddr, que hace referencia al 3º elemento de la lista y así sucesivamente

Los criterios de terminación con listas de recursividad son:

- lista vacía: []

- un elemento encontrado
- una posición encontrada

```
%primero/2
%primero(L,E) es cierto sii E es el primer elemento de la lista L.

primero([Car|_],Car). // primero([Car|Cdr],Car).

%segundo/2
%segundo(L,E) es cierto sii E es el segundo elemento de la lista L.

segundo([Car,Cadr|Cdr],Cadr).  // segundo([,Cadr|_],Cadr).

%L=[1,2,3]= segundo([1,2|[3]],2)

%ultimo/2
%ultimo(L,E) es cierto sii E es el ultimo elemento de la lista L

ultimo([Car],Car).
ultimo([_|Cdr],R):-ultimo(Cdr,R).

%penultimo/2
%penultimo(L,E) es cierto sii E es el penultimo elemento de la lista L.

penultimo([Car,_],Car).
penultimo([_|Cdr],R):-penultimo(Cdr,R).

%negacion por fallo: mecanismo para determinar si algo no es cierto. se emplea cuando no
se puede encontrar una prueba de que algo sea verdadero

%izqda_der/3
%izqda_der(I,D,L) es cierto sii I está a la inmediata izquierda de D en la lista L
%izqda_der(I,D,[1,2,3]) -> I=1, D=2, D=3

izqda_der(I,D,[I,D|_]).
izqda_der(I,D,[_|L]):-izqda_der(I,D,L).


EXPONENCIAL
exp(X,0,s(0)).
exp(X,s(0),x)
%exp(X,s(Y),):-(X,Y,Z), mult(X,Z,W).

%contiguo/3
%contiguo(X,Y,L). es cierto sii X e Y es contiguo

contiguo(X,Y,L):-izqda_der(X,Y,L). % X está a la izquierda de Y
contiguo(X,Y,L):-izqda_der(Y,X,L). % Y está a la izquierda de X
```

%intercalar/3
%intercalar(L1,L2,R). es cierto sii R es la lista resultante de intercalar los elementos de las listas L1 y L2

%intercalar([1,2][a],R)
?- R=[1,a,2]

%intercalar([1][a,b],R)
?- R=[1,a]

%intercalar([][a],R)
?- R=[]

%intercalar([1][],R)
?- R=[1]

PRACTICA 2

L=[a,b,c]=[a|[b|c]]= [a|[b|[c]|[]]]

%primero(L,E) cierto sii E es el primer elemento de la lista L

primero ([Car|_],Car).

%intercalar/3
%intercalar(L1,L2,R). es cierto sii R es a lista resultante de intercalar los elementos de las listas L1 y L2. Ejemplos:
%intercalar([1,2],[a],R) ?- R=[1,a,2]
%intercalar([1],[a,b],R) ?- R=[1,a]
%intercalar([],[a],R) ?- R=[]
%intercalar([1],[],R) ?- R=[1]

intercalar([],_,[]).
intercalar([Car|_], [] ,[Car]).
intercalar([Car1|Cdr1], [Car2|Cdr2] , [Car1,Car2|R]) :- intercalar(Cdr1,Cdr2,R).

%select/3
%select (E,L,R). es cierto sii E es un elemento de la lista L y R es el resultado de eliminar E
select(E,[Car|Cdr],[Car|R]):- select(E,Cdr,R).de la lista L

```
select(Car,[Car|Cdr],Cdr).
select(E,[Car|Cdr],[Car|R]):- select(E,Cdr,R).


%miembro/2
%miembro(E,L). es cierto sii E es un elemento de la lista L

miembro(Car,[Car|_]).
miembro(X,[_|Cdr]):- miembro(X,Cdr).

%concat(L1,L2,R). es cierto sii R es el resultado de concatenar las listas L1 y L2

concat([],X,X).
concat([Car|Cdr],L,[Car|R]):- concat(Cdr,L,R).

% [1,2,3]  [a,b]
%        [1|[2,3]], [a,b], :- [2,3],[a,b],[2,3,a,b]


%inv/2
%inv(L,I). es cierto sii I es la lista inversa de la lista L

inv([],[]).
inv([Car|Cdr],R):- inv(Cdr,I),concat(I,[Car],R).

%[1,2,3]
%[1|[2,3]],       [3,2,1]            [2,3],[3,2]        [3,2],[1] R=[3,2,1]


%aplanar/2
%aplanar(L,R). es cierto sii R es la lista resultante de aplanar la lista L
%aplanar([1,[a,b,[c]],2,[3]],R)  R=[1,a,b,c,2,3]

aplanar([],[]).
aplanar(Atomo,[Atomo]):- atomic(Atomo), Atomo\==[].

aplanar([Car|Cdr],R):- aplanar(Car,Car_aplanado),
                       aplanar(Cdr,Cdr_aplanado),
                       concat(Car_aplanado,Cdr_aplanado,R).


%subconjunto/2
%subconjunto(L1,L2). es cierto sii la loisa L1 es una sublista de los elementos de la lista L2.

subconjunto([],_).
subconjunto([Car|Cdr],L):-select(Car,L,NuevaLista),subconjunto(Cdr,NuevaLista).
```

%linea/3
%linea(E,L,R). es cierto sii R es la lista resultado de emparejar el elemento E con cada elemento de la lista L
linea(E,[],[]).
linea(E,[Car|Cdr],[[E|Car]|R]):-linea(E,Cdr,R).

%cartesiano/3
%cartesiano(L1,L2,R). es cierto sii R es la lista que contiene los pares de producto cartesiano de las listas L1 y L2
%cartesiano ([1,2],[a,b,c],R) ?- R = [[[1,a],[1,b],[1,c]],[[2,a],[2,b],[2,c]]]

cartesiano([],L,[]).
cartesiano([Car|Cdr],L,):- linea(Car,L,Linea), cartesiano(Cdr,L,Resto),concat(Linea,Resto,R).

PRACTICA EVALUABLE: 5/11 : LISTAS Y CORTE

PRACTICA 3

/*cuando se interroga al interprete con una consulta,el motor de inferencia de
 * prolog busca las clausulas que coinciden(unifican) con la consulta.
 *
 * si la unificación tiene exito, se explora esa rama del arbol, en caso
 * contrario, se retrocede(backtracking) y se prueba la siguiente clausula.
 *
 * Es decir,el orden en que se declaran las clausulas, influye en el orden
 * en que se explora el espacio de busqueda.
 *
 * Por lo tanto, colocar los hechos(caso base) al principio puede ayudar a que
 * el motor de inferencia encuentre estos casos base antes y evite entrar en
 * bucles.
 *
 * Respecto a las reglas recursivas,situar las clausulas mas especificas
 * (con mas condiciones) antes que las mas generales, permite encontrar soluciónes
 * mas precisas antes.
 * /

% el corte,cut(!) es un operador de control de flujo, un predicado predefinido, que le indica al motor de inferencia, que si alcanza un punto en la ejecucion del programa, no debe considerar ninguna otra alternativa

%es decir, el corte (!), permite podar ramas del arbol de busqueda aun no exploradas en la busqueda de una solucion

%en otras palabras,”poda” el arbol de busqueda en ese punto, evitando que prolog siga explorando ramas que,segun el criterio del programador no conducen a soluciones validas.

%Tipos de corte:

%**Corte verde**: su objetivo es evitar soluciones redundantes o innecesarias. Al ejecutarse garantiza que se ha encontrado una solucion y que no se buscará ninguna otra para la consulta actual.

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

%**Corte rojo**: su objetivo es evitar explorar ramas inutiles y mejorar la eficiencia del programa. Permite implementar construcciones condicionales if-then-else

```
%if_then_else(P,Q,R):- P,!,Q.
%if_then_else(P,Q,R):- R.
```

%“Si P es cierto, entonces probar Q, si no probar R”

```
%interseccion/3
%interseccion(L1,L2,I) es cierto sii I es la intersección de las listas L1 y L2.

interseccion([],_,[]).
interseccion([Car|Cdr],C,[Car|I]):-miembro(Car,C),!,interseccion(Cdr,C,I).
interseccion([Car|Cdr],C,I):- interseccion(Cdr,C,I).

%union/3
%union(L1,L2,U) es cierto sii U es la union de las listas L1 y L2

union([],C,C).
union([Car|Cdr],C,U):- miembro(Car,C),!,union(Cdr,C,U).
union([Car|Cdr],C,[Car|U]):- union(Cdr,C,U).

%respuesta de superiores(5,[4,8,2,3,6],X). tiene que ser  X=[8,6]

%superiores/3
%superiores(Elementalidad,Lista,Resultado)

superiores(_,[],[]).
superiores(E,[Car|Cdr],[Car|L]):-E<Car, !, superiores(E,Cdr,L).
superiores(E,[_|Cdr],L):-superiores(E,Cdr,L).


%listar_hasta([1,2,3,4],3,X). tiene que ser X=[1,2]

%listar_hasta/3
%listar_hasta(Lista,Elem,Respuesta)
listar_hasta([],_,[]).
listar_hasta([Car|_],Car,[]):-! .
listar_hasta([Car|Cdr],E,[Car|L]):- listar_hasta(Cdr,E,L).
```

%creciente ([2,5,1,8,9]). es false

%creciente(L)

creciente([_]):-!.
creciente([Car,Cadr|Cdr]):- Car<Cadr, creciente([Cadr|Cdr]).

PRÁCTICA 4

%partir/4
%partir(Pivote,Lista,Menores_que_pivote).
%es cierto sii Menores_que_pivote contiene los elementos < que pivote y Mayores_o_iguales_que_pivote contiene los elementos de Lista >= que Pivote.

partir(_,[],[],[]).
partir(P,[Car|Cdr],[Car|Izq],Der):- Car<P, !, partir(P,Cdr,Izq,Der).
partir(P,[Car|Cdr],Izq,[Car|Der]):- partir(P,Cdr,Izq,Der).

%quicksort/2
%quicksort(L,0). es cierto sii 0 es la lista resultante de ordenar los elementos de la lista L segun el algoritmo Quicksort

quicksort([],[]).
quicksort([Car|Cdr],R):-
partir(Car,Cdr,Izq,Der),quicksort(Izq,Izq_ordenada),quicksort(Der,Der_ordenada),concat(Izq_ordenada,[Car|Der_ordenada],R).

concat([],X,X).
concat([Car|Cdr],L,[Car|R]):- concat(Cdr,L,R).

%El predicado es un predicado predefinido en Prolog que siempre falla.
%Es decir,Al ejecutarlo el motor de inferencia retrocede(backtracking) y busca otras alternativas para resolver el objetivo actual.

%fail se verifica siempre y al verificarse corta la rama actual provocando el retroceso.

%Negación por fallo, permite expresar la usencia de una solucion a una consulta. En lugar de una negación explicita,Prolog infiere la falsedad de una proposicion si no encuentra una manera de demostrarla, es decir, algo es falso si no se puede demostrar que es verdadero

%se define así:

not(P):- P,!,fail.
not(_).

%”Si P se verifica, el corte(!) le indica al motor de inferencia que no necesita seguir buscando mas (cortamos todas las alternativas) y luego el fail, corta la rama actual.”
%Se puede interpretar este fallo como negación, en otro caso sería cierto.

```
coche(dodge,caliber).
coche(opel,corsa).
coche(toyota,prius).
europea(opel).
comprar(Marca,Modelo):- coche(Marca,Modelo),not(europea(Marca)).

puedo_comprar(Marca,Modelo):- freeze(Marca,not(europea(Marca))),coche(Marca,Modelo).
puedo_comprar(Marca,Modelo):-when(nonvar(Marca),not(europea(Marca))),coche(Marca,
Modelo).

%nonvar(X). es un predicado predefinido que verifica si un termino es una variable
instanciada o no
%nonvar(X) comprueba si X tiene un valor asignado o continua siendo variable libre.
%si X tiene un valor asignado, enton nonvar(X) unifica a true.
%Si X es una variable sin valor asignado, enton nonvar(X) falla y unifica a false.

%El predicado when (Condicion,Objetivo): retrasa la ejecucion del Objetivo hasta que la
condicion de evalue a true.

% El predicado freeze(Variable,Objetivo):  retrasa la ejecucion del Objetivo hasta que la
variable sea instanciada, es decir, tenga valor asignado.

soltero(Persona):-varon(Persona), not(casado(Persona)).
es_soltero(Persona):- freeze(Persona, not(casado(Persona))), varon(Persona).
casado(juan).
varon(juan).
varon(pepe).
```