Prolog

% var(X): Tiene éxito si X es una variable no instanciada en el momento de la llamada.
% nonvar(X): Tiene éxito si X no es una variable no instanciada.
% ground(X): Tiene éxito si X es un término completamente instanciado, lo que significa
%   que X y todos sus sub-términos no contienen variables no instanciadas.

%desde(+X,-Y)
```
desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).
```

% desde2(+X, -Y)
```
desde2(X, X).
desde2(X, Y) :- var(Y), N is X+1, desde2(N, Y).
desde2(X, Y) :- nonvar(Y), X < Y.
```

% desde2 no se cuelga cuando le pasas valores instanciados
% no poner desde2 en cualquier caso para evitar que se cuelgue (es para casos muy especificos)

% between(0,10,5). devuelve true, funciona tamb sin instanciar.

% esListaFinitaPositiva(-L)
```
esListaFinitaPositiva([]).
esListaFinitaPositiva(X) :- desde2(0, Y), sumlist(X,Y).
```

% generarLista(+suma, -lista)
```
generarLista(0,[]).
generarLista(S,[X|XS]) :-
   S > 0,
   desde2(1,X), % instancio x >= 1
   S1 is S-X,   % S1 = S-X
   generarLista(S1,XS). % sumlist(XS)=S-X=S1
```

% partes(+A,-P(A))
```
partes([],[]).
partes([X|XS],[X,L]) :- partes(XS,L)
partes([X|XS], L) :- partes(XS,L)
```

% sinRepetidos
```
scr([],[]).
scr([X],[X]).
scr([X,X|XS],L) :- scr([X|XS], L).
scr([X,Y|XS],[X|L]) :- X \= Y, scr([Y|XS],L).
```

% append

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

% insertar

```
insertar(X,L,LX) :- append(I,D,L), append(I,[X|D], LX).
```

% permutacion(+L,?P) ~ usando append.

```
permutacion([],[]).
permutacion([X|XS], P) :- permutacion(XS,L), insertar(X,L,P).
```

% iesimo

```
iesimo(0,[X|_],X).
iesimo(I,[_|XS],X) :- iesimo(I2,XS,X), I is I2+1.
```

% paresMenoresQueX(+X,-Y)

```
pmq(X,Y) :- between(0,X,Y), Y mod 2 =:= 0.
```

% generarPares(X,Y)

```
paresSuman(S,X,Y) :- S1 is S-1, between(1,S1,X), Y is S-X.
generarPares(X,Y) :- desde2(2,S), paresSuman(S,X,Y).
```

% coprimos(-X,-Y)

```
coprimos(X,Y) :- generarPares(X,Y), gcd(X,Y)==1.
%                       generate       test
```

% EJ triangulos (tipo parcial) - programar perimetro usando esTriangulo(.)

```
esTriangulo(tri(A,B,C)) :- A < B+C, B < A+C, C < B+C.

% Vemos cada caso: T instanciada (o no) y P instanciada (o no)
% caso triangulo instanciado, P instanciado (o no)

perimetro(tri(A,B,C),P) :- ground(tri(A,B,C)),
                esTriangulo(tri(A,B,C)), P is A+B+C.

% caso triangulo no instanciado o parcialmente instanciado, P instanciado (o no)
perimetro(tri(A,B,C),P) :- not(ground(tri(A,B,C))), nonvar(P),
                armarTriplas(P,A,B,C), esTriangulo(tri(A,B,C)).

% desde2 no se cuelga cuando le pasas valores instanciados
% no poner desde2 en cualquier caso para evitar que se cuelgue (es para casos muy
especificos)
```

armarTriplas(P,A,B,C) :- desde2(3,P), between(0,P,A), S is P-A, between(0,S,B), C is S-B.

EJS PREPARCIAL

% Ej. PESOS

```
peso([],0).
peso([X],Y) :- peso(X,Y).
peso([X|XS], P) :- peso(X,P1), peso(XS,P2), P is P1+P2.
peso(X,X).

pesoMaximo([],0).
pesoMaximo([X|XS], P) :- peso(X,P2), pesoMaximo(XS,P1), P is max(P2,P1).

elementoMasPesado(L,X) :- member(X,L), pesoMaximo(L,P), peso(X,P).
```

% EJ. Arboles

```
% camino(+A, -C)
% C es un camino de la raiz a alguna hoja
camino(bin(nil,V,nil), [V]).
camino(bin(I,V,D),[V|XS]) :- camino(D,XS), D \= nil.
camino(bin(I,V,D),[V|XS]) :- camino(I,XS), I \= nil.

% caminoMasLargo(+A, -C)
caminoMasLargo(A,C) :- camino(A,C),
                   length(C,L1),
                   not(
                       (camino(A,C2),
                        length(C2,L2),
                        L2 > L1
                       )
                    ).

% caminoUnicoDeLong(+A, +N, -C)
% si C es un camino de A de longitud N y no hay otro camino de longitud N
caminoUnicoDeLong(A,N,C) :- camino(A,C),
                       length(C,N),
                       not(
                           (camino(A,C2),
                            length(C2,N),
                            C \= C2
                           )
                       ).
```

% EJ Palabras

```
% palabra(+A,+N, ?P), al final P nos quedo reversible
% generar palabras de longitud N de un alfabeto A
palabra(A,0,[]).
palabra(A,N,[X|XS]) :- N >= 0, member(X,A), N1 is N-1, palabra(A,N1,XS).
```

```
% frase(+A,-F)
% una frase es un a lista finita de palabras no vacias en A generar todas las frases posibles
frase(_,[]).
frase(A,F) :- desde2(1,N), fraseSumaX(A,N,F).

% fraseSumaX(+A,+N,-F)
fraseSumaX(A,N,[F]) :- palabra(A,N,F).
fraseSumaX(A,N,[F|FS]) :- N>0, between(1,N,N1), palabra(A,N1,F), N2 is N-N1, N2>0,
fraseSumaX(A,N2,FS).
```

% EJ ochoReinas(+XS)

```
% la posicion en la lista es la columna y el numero es la fila
ochoReinas([]).
ochoReinas([X|XS]) :- ochoReinasAux([X|XS], 1).

% C representa el indice de R
ochoReinasAux([R|XS],C) :- between(1,8,R),
                         not(
                          (member(R1,XS),
                          encontrarIndice(R1,XS,C1), % C1 es el indice de R1 en XS.
                          C2 is C+C1,
                          colision(R,C,R1,C1))
                          ),
                          ochoReinas(XS).

encontrarIndice(X,[X|_], 1).
encontrarIndice(X,[Y|XS],P) :- encontrarIndice(X,XS,P2), P is P2+1.

colision(R,_,R,_).
colision(_,C,_,C).
colision(R1,C1,R2,C2) :- between(1,8,Z), R2 is R1+Z, C2 is C1+Z.
```

% EJ listaDeArbolesNoVacios

```
listaDeArboles(L) :- desde(0,S), listaAcotadaDeArboles(S,L).

listaAcotadaDeArboles(0,[]).
listaAcotadaDeArboles(S,[X|XS]) :- between(1,S,Na),
                    arbolDeN(Na,X), S2 is S-Na,
                    listaAcotadaDeArboles(S2,XS).


arbolDeN(0,nil).
arbolDeN(N,bin(I,_,D)) :- N > 0, N2 is N-1, paresQueSuman(N2,NI,ND), arbolDeN(NI,I),
arbolDeN(ND,D).

paresQueSuman(S,X,Y) :- between(0,S,X), Y is S-X.
```

Smalltalk

###3

```
Object subclass: #Robot
   instanceVariableNames: `x y`
   ...

   Robot >> InitWith: aBlock
      b := aBlock.
      x := 0.
      y := 0.
      ^self.

   Robot class >> newWith: aBlock
      |r|
      r := self new.
      ^r InitWith: aBlock

   Robot >> avanzar
      |res|
      res := b value: x value: y.
      x := res at:1.
      y := res at:2.
      ^ self.

Robot subclass: #Drone
   instanceVariableNames: `z`
   ...

   Drone class >> newWith: aBlock
      |r|
      r := super newWith: aBlock.
      ^r init.

   Drone >> init
      z := 0.
      ^self.

   Drone >> avanzar
      z<10 ifTrue: [z := z+1].
      ^super avanzar.
```

```
###################################################################3
SmallInteger >> fact
		|res, m|
		m := self
		m == 0 ifTrue: [ res := 1. ] ifFalse: [ res := (m - 1) fact * m]
		^ res

// vale que mcm(a,b) = a*b/gcd(a,b)

SmallInteger >> mcm: aNumber
   |res|
   res := (self * aNumber) / (self gcd aNumber)
   ^res

#########################################################################
minimo: aBlock
   | minElement minValue |
   self do: [:each |
      | val |
      minValue ifNotNil: [
         (val := aBlock value: each) < minValue ifTrue: [
            minElement := each.
            minValue := val]
         ]
      ifNil: ["first element"
         minElement := each.
         minValue := aBlock value: each].
   ].
   ^minElement
```

NATURALES en Prolog:

```
natural(cero).
natural(suc(X)) :- natural(X).

menor(cero,suc(X)) :- natural(X).
menor(suc(X),suc(Y)) :- menor(X,Y).
```

Deduccion Natural

Deducción natural

Ejercicio 9 ★

Demostrar en deducción natural que vale $\vdash \sigma$ para cada una de las siguientes fórmulas, **sin usar principios de razonamiento clásicos**, salvo que se indique lo contrario:

I. Intercambio ($\forall$): $\forall X.\forall Y.P(X,Y) \iff \forall Y.\forall X.P(X,Y)$.

II. Intercambio ($\exists$): $\exists X.\exists Y.P(X,Y) \iff \exists Y.\exists X.P(X,Y)$.

III. Intercambio ($\exists/\forall$): $\exists X.\forall Y.P(X,Y) \implies \forall Y.\exists X.P(X,Y)$.

IV. Universal implica existencial: $\forall X.P(X) \implies \exists X.P(X)$.

V. Diagonal ($\forall$): $\forall X.\forall Y.P(X,Y) \implies \forall X.P(X,X)$.

VI. Diagonal ($\exists$): $\exists X.P(X,X) \implies \exists X.\exists Y.P(X,Y)$.

VII. de Morgan (I): $\neg\exists X.P(X) \iff \forall X.\neg P(X)$.

VIII. de Morgan (II): $\neg\forall X.P(X) \iff \exists X.\neg P(X)$.
Para la dirección $\Rightarrow$ es necesario usar principios de razonamiento clásicos.

IX. Universal/conjunción: $\forall X.(P(X) \wedge Q(X)) \iff (\forall X.P(X) \wedge \forall X.Q(X))$.

X. Universal/disyunción: $\forall X.(P(X) \vee \sigma) \iff (\forall X.P(X)) \vee \sigma$, asumiendo que $X \notin \mathsf{fv}(\sigma)$.
Para la dirección $\Rightarrow$ es necesario usar principios de razonamiento clásicos.

XI. Existencial/disyunción: $\exists X.(P(X) \vee Q(X)) \iff (\exists X.P(X) \vee \exists X.Q(X))$.

XII. Existencial/conjunción: $\exists X.(P(X) \wedge \sigma) \iff (\exists X.P(X) \wedge \sigma)$, asumiendo que $X \notin \mathsf{fv}(\sigma)$.

XIII. Principio del bebedor: $\exists X.(P(X) \implies \forall X.P(X))$.
En este ítem es necesario usar principios de razonamiento clásicos.

Posible plan:
Como c es un camino que no conduce a Roma, el mismo no debe conducir a otro punto X tal que comunique con Roma por algún camino, ni conduce a este punto X. En particular c no comunica X con Roma, lo cual implica que c no es camino, absurdo! (Llegamos a la refutación).

1. $\{\neg \mathtt{camino}(C_1), \mathtt{comunica}(f(C_1), g(C_1), C_1)\}$
2. $\{\neg \mathtt{comunica}(A_2, B_2, C_2), \mathtt{camino}(C_2)\}$
3. $\{\mathtt{comunica}(X_3, Y_3, h(X_3, Y_3))\}$
4. $\{\neg \mathtt{comunica}(X_4, Y_4, C_4), \mathtt{conduceA}(Y_4, C_4)\}$
5. $\{\neg \mathtt{conduceA}(X_5, C_5), \neg \mathtt{comunica}(X_5, Y_5, D_5), \mathtt{conduceA}(Y_5, C_5)\}$
6. $\{\mathtt{camino}(c)\}$
7. $\{\neg \mathtt{conduceA}(Roma, c)\}$

SLD:
■ Se utilizan solo cl´ausulas de Horn. ■ Se empieza por una cl´ausula objetivo. ■ Se realiza de manera lineal. ■ Se utiliza la regla de resoluci´on binaria en vez de la general.