Prolog

Predicados: (Campus)

sort(+Lista, -ListaOrdenada): Ordena una lista en orden creciente y elimina duplicados.
msort(+Lista, -ListaOrdenada): Ordena una lista en orden creciente sin eliminar duplicados.
length(?Lista, ?Longitud): Longitud de la lista.
nth1(+Índice, +Lista, ?Elem): Devuelve el elemento en la posición del índice (comenzando desde 1) de la lista.
nth0(+Indice, +Lista, ?Elem): Devuelve el elemento en la posición Índice (comenzando desde 0) de la lista.
member(?Elemento, +Lista): Pertenece
append(?Lista1, ?Lista2, ?ListaConcatenada): Concatena dos listas.
last(+Lista, ?Elem): Último elemento.
between(+Menor, +Mayor, ?Valor): enera o verifica si Valor está en el rango entre Menor y Mayor
is_list(+Término): Verifica si Termino es una lista.
list_to_set(+Lista, -Conjunto): Convierte una lista en un conjunto (elimina duplicados).
is_set(+Lista): Verifica si Lista es un conjunto (sin duplicados).
union(+Conjunto1, +Conjunto2, -Unión): Calcula la unión de dos conjuntos.
intersection(+Conjunto1, +Conjunto2, -Intersección): Calcula la intersección de dos conjuntos.
subset(+Subconjunto, +Conjunto): Verifica si Subconjunto es un subconjunto de Conjunto.
subtract(+Conjunto1, +Conjunto2, -Resultado): Calcula la diferencia entre dos conjuntos.
select(+Elemento, +Lista, -Resto): Selecciona y elimina un elemento de una lista.
delete(+Lista, +Elemento, -Resto): Elimina todas las ocurrencias de un elemento de una lista.
reverse(+Lista, -ListaReversa): Invierte el orden de los elementos de una lista.
atom(+Término): Si Termino es un átomo.
number(+Término): Si termino es un número.
numlist(+Inicio, +Fin, -Lista): Genera una lista de números consecutivos entre Inicio y Fin.
sum_list(+Lista, -Suma): Calcula la suma de los elementos de una lista.
flatten(+ListaAnidada, -ListaPlana): Aplana una lista anidada en una lista plana.

Operaciones extra-lógicas: is, \=, ==, =:=, =\=, >, <, =<, >=, abs, max, min, gcd, var, nonvar, ground, trace, notrace, make, halt

Ejercicios de la guia útiles para referencia:

```
%permutación(+L1, ?L2). usando insertar
insertar(X, [], [X]).
insertar(X, [Y|Ys], [X, Y|Ys]).
insertar(X, [Y|Ys], [Y|Zs]) :- insertar(X, Ys, Zs).
permutación([], []).
permutación([X|Xs], Ys) :- permutación(Xs, Ys2), insertar(X, Ys2, Ys).
```

Árboles

```
%13. %I. inorder(+AB, -Lista)
inorder(nil, []).
inorder(bin(I, V, D), L) :- inorder(I, L1), inorder(D, L2), append(L1, L2, L3), append(L3, [V], L).
%II. arbolConInorder(+Lista, -AB). OBTIENE TODOS LOS AB POSIBLES
arbolConInorder([], nil).
arbolConInorder(Ls, bin(I, V, D)) :- append(Ls1, [V], Ls), append(X, Y, Ls1), arbolConInorder(X, I),
arbolConInorder(Y, D).
%III. aBB(+T)
aBB(nil). aBB(bin(nil, _, nil)).
aBB(bin(I, V, D)) :- raiz(I, X), raiz(D, Y), V >= X, Y > V, aBB(I), aBB(D).
```

```
%IV. aBBInsertar(+X, +T1, -T2)
aBBInsertar(X, nil, bin(nil, X, nil)).
aBBInsertar(X, bin(I,V,D), bin(I, V, T2)) :- aBB(bin(I,V,D)), X > V, aBBInsertar(X, D, T2).
aBBInsertar(X, bin(I,V,D), bin(T2, V, D)) :- aBB(bin(I,V,D)), V >= X, aBBInsertar(X, D, T2).

%GENERATE AND TEST
%14. coprimos(-X,-Y)
coprimos(X, Y) :- desde(1, N), suman(Y, X, N), gcd(X,Y) =:= 1.
suman(X, Y, S) :- S1 is S-1, between(1, S1, X), Y is S-X.
%16.
%.I. esTriángulo(+T)
esTriángulo(tri(A,B,C)) :- A1 is B+C, A2 is abs(B-C),  A1 > A, A > A2,
                           B1 is A+C, B2 is abs(A-C),  B1 > B, B > B2,
                           C1 is B+A, C2 is abs(A-B),  C1 > C, C > C2.

%II. perímetro(?T,?P)
perimetro(tri(A,B,C), P) :- ground(tri(A,B,C)), esTriángulo(tri(A,B,C)), P is A+B+C.
perimetro(tri(A,B,C), P) :- not(ground(tri(A,B,C))), generarTri(tri(A,B,C), P), P is A+B+C.

generarTri(tri(A,B,C), P) :- desde2(1, P, A), between(1, A, B), between(1, B, C), esTriángulo(tri(A,B,C)).
%III. triángulo(-T)
triangulo(T) :- desde(1, P), perimetro(T, P).

%19.  corteMásParejo(+L,-L1,-L2) Uso del NOT para explorar las posibilidades!
corteMásParejo(L, I, D) :- append(I, D, L), sumlist(I, X), sum_list(D, Y), K is abs(X-Y), not(otroCorte(L, K)).

otroCorte(L, K) :- append(I, D, L), sumlist(I, X), sum_list(D, Y), Z is abs(X-Y), K > Z.
```

Ejercicios de Último Repaso:

```
sublistaMasLargaDePrimos(L,P) :- sublistaDePrimosDeLong(L,P,Long), not((sublistaDePrimosDeLong(L,_,Long2),
Long2 > Long)).

sublistaDePrimosDeLong(L,P,Long) :- sublista(L,P), soloPrimos(P), length(P,Long).

sublista(_,[]).
sublista(L,S) :- append(P,_,L), append(_,S,P), S \= [].

soloPrimos(L) :- not((member(X,L), not(esPrimo(X)))).

% esPrimo(+P)
esPrimo(P) :- P \= 1, P2 is P-1, not((between(2,P2,D), mod(P,D) =:= 0)).

%?- listaDeÁrboles(L).
%L = [];
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).

tamArbol(0,nil). tamArbol(N,bin(I,_,D)) :- tamArbol(NI,I), tamArbol(ND,D), N is 1+NI+ND.

Predicados hechos en clase

```
%entre(+X,+Y,-Z)
entre(X,Y,X) :- X =< Y.
entre(X,Y,Z) :- X < Y, X2 is X+1, entre(X2,Y,Z).

%sinConsecutivosRepetidos(+XS,-YS)
sinConsecutivosRepetidos([],[]).
sinConsecutivosRepetidos([X],[X]).
sinConsecutivosRepetidos([X,X|XS],L) :- scr([X|XS],L).
sinConsecutivosRepetidos([X,Y|XS],[X|L]) :- X \= Y, scr([Y|XS],L).

%partes(+XS,-YS)
partes([],[]).
partes([X|XS],[X|L]) :- partes(XS,L).
partes([_|XS],L) :- partes(XS,L).

%prefijo(+L,?P)
prefijo(L,P) :- append(P,_,L).

%sufijo(+L,?P)
sufijo(L,P) :- append(_,P,L).

%sublista(+L,?SL)
sublista(_,[]).
sublista(L,S) :- append(P,_,L), append(_,S,P), S \= [].

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

%permutacion(+L,?P)
permutacion([],[])
permutacion([X|XS],P) :- permutacion(XS,L), insertar(X,L,P).

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

%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.

%paresMenorQue(+X, -Y)
paresMenorQue(X,Y) :- between(0,X,Y), Y mod 2 =:= 0.

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.

%corteMasParejo(+L,-L1,-L2)
corteMasParejo(L,L1,L2) :- unCorte(L,L1,L2,D), not((unCorte(L,_,_,D2), D2 < D)).

unCorte(L,L1,L2,D) :- append(L1,L2,L), sumlist(L1,S1), sumlist(L2,S2), D is abs(S1-S2).

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

perimetro(tri(A,B,C),P) :- ground(tri(A,B,C)), esTriangulo(tri(A,B,C)), P is A+B+C.
perimetro(tri(A,B,C),P) :- not(ground(tri(A,B,C))), armarTriplas(P,A,B,C), esTriangulo(tri(A,B,C)).

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

triangulos(T) :- perimetro(T,_).
```

Justificaciones Reversibilidad:

```
%prefijoHasta(?X, +L, ?Prefijo)
prefijoHasta(X, L, Prefijo) :- append(Prefijo, [X | _], L).
```

Si L no está instanciada, no hay manera de instanciarla, ya que el segundo argumento del append tiene un sufijo sin instanciar, y el tercero es la misma L que no está instanciada.
Si L está instanciada, tanto X como Prefijo pueden estar o no instanciados, porque append se encarga de instanciar los si no lo están, o de verificar que coincidan con un prefijo de L y el elemento siguiente si ya están instanciados.

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

Si X no está instanciada, desde(X,Y) va a arrojar el resultado Y = X, y luego al entrar en la segunda cláusula va a arrojar un error al intentar realizar una operación aritmética sobre Z sin instanciar.
Si Y está instanciada, va a tener éxito si Y >= X, pero luego (o siempre, si Y < X) se va a colgar porque va a seguir generando infinitos valores para Z y comparando sus sucesores con Y, lo cual nunca va a tener éxito.

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

X debe estar instanciada por el mismo motivo que en desde/2.
Si Y no está instanciada, instancia Y en X para el primer resultado, y luego entra por la tercera cláusula y va generando infinitos valores para Y.
Si Y está instanciada, entra por la primera cláusula si es igual a X, y por la segunda en caso contrario, haciendo una comparación entre dos variables ya instanciadas, lo cual funciona correctamente.

```
%preorder(+A, ?L)
preorder(Nil, []).
preorder(Bin(I, R, D),[R | L]) :- preorder(I, LI), preorder(D, LD), append(LI, LD, L).
```

Si A no está instanciado, funciona para el caso L = [], porque solo unifica con la primera cláusula. Pero para L no vacía o no instanciada va a entrar (eventualmente) en la segunda cláusula, y va a llamar a preorder con dos variables sin instanciar, lo cual solo le va a permitir generar árboles Nil y listas vacías para luego volver a llamarse infinitamente con argumentos sin instanciar.
Si A está instanciado y L no, instancia L con [] si A es Nil, y en caso contrario calcula los respectivos preorders con I y D ya instanciados y los junta con append.
Si ambos argumentos están instanciados, utiliza unificación y/o append para verificar que L coincida con el preorder de A.

Resolución Lógica Primer Orden

Pasaje de fórmula a forma clausal, todos los pasos son equivalentes lógicos menos Skolem (solo preserva satisfacibilidad y no validez):

1. Reescribir ⇒ usando ¬ y ∨.
2. Pasar a formula normal negada, empujando ¬ hacia adentro.
3. Pasar a formula normal prenexa, extrayendo ∀, ∃ hacia afuera..
4. Pasar a f.n. de Skolem, Skolemizando los existenciales.
   Ejemplos:
   - ∀X. ∃Y. P(X, Y) es sat. sii ∀X. P(X, f(X)) es sat.
   - ∀X. ∀V. ∃Y. ∀W. P(X, Y) es sat. sii ∀X. P(X, f(X, V)) es sat.

   (Para cada Existe que eliminamos, creamos una función nueva que recibirá tantos parámetros como Para Todos haya a la izquierda del Existe)

5. Pasar a fórmula normal conjuntiva, distribuyendo ∨ sobre ∧.
6. Empujar los cuantificadores hacia adentro de las conjunciones. (Cambiar nombres es recomendable)

Luego, para determinar si una fórmula de primer orden σ es válida:

1. Pasar su negación ¬σ a forma clausal. Se obtiene un conjunto C de cl´ausulas tal que ¬σ es satisfacible sii C es satisfactible. (Nuestra base de conocimientos no se debe negar, solo lo que queremos demostrar)
2. Aplicar repetidamente la Regla de Resolución.
3. Si se alcanza la cláusula vacía, ¬σ es insatisfacible y σ válida.
4. El método puede no terminar.

Regla de Resolución de Primer Orden:

$$\frac{\{\sigma_1, \ldots, \sigma_p, \ell_1, \ldots, \ell_n\} \quad \{\neg\tau_1, \ldots, \neg\tau_q, \ell'_1, \ldots, \ell'_m\} \quad \mathbf{S} = \mathrm{mgu}(\{\sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{?}{=} \ldots \stackrel{?}{=} \sigma_p \stackrel{?}{=} \tau_1 \stackrel{?}{=} \tau_2 \stackrel{?}{=} \ldots \stackrel{?}{=} \tau_q\})}{\mathbf{S}(\{\ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_m\})}$$

con $p > 0$ y $q > 0$.

Regla Binaria (No es completa):

$$\frac{\{\sigma, \ell_1, \ldots, \ell_n\} \quad \{\neg\tau, \ell'_1, \ldots, \ell'_m\} \quad \mathbf{S} = \mathrm{mgu}(\{\sigma \stackrel{?}{=} \tau\})}{\mathbf{S}(\{\ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_m\})}$$

Resolución SLD

Menor generalidad a cambio de mayor eficiencia, solo se puede aplicar cuando tenemos cláusulas de Horn.
La regla de resolución SLD involucra siempre a una cláusula de definición y una cláusula objetivo.
Cláusulas Objetivo: Sin Literales Positivos.
Cláusulas de Definición: A lo sumo un literal positivo.

El metodo de resolucion es SLD sii:

- Se utilizan solo cláusulas de Horn.
- Se empieza por una cláusula objetivo.
- Se realiza de manera lineal (Usando siempre la obtenida).
- Se utiliza la regla de resolución binaria en vez de la general.

SmallTalk

Colecciones: Bag (Multiconjunto), Set (Conjunto), Array (Arreglo), OrderedCollection (Lista), SortedCollection, (Lista ordenada), Dictionary (Hash).
Se pueden crear llamando a la clase y con el mensaje with:, o usando el #
#(1 2 4) = (Array with: 1 with: 2 with: 4). Bag withAll: #(1 2 4).

Mensajes Comunes:
add: agrega un elemento.
at: devuelve el elemento en una posición.
at:put: agrega un elemento a una posición.
includes: responde si un elemento pertenece o no.

includesKey: responde si una clave pertenece o no.
do: evalúa un bloque con cada elemento de la colección.
keysAndValuesDo: evalúa un bloque con cada par clave-valor.
keysDo: evalúa un bloque con cada clave.
select: devuelve los elementos de una colección que cumplen un predicado (filter de funcional).
reject: la negación del select:
collect: devuelve una colección que es resultado de aplicarle un bloque a cada elemento de la colección original (map de funcional).
detect: devuelve el primer elemento que cumple un predicado.
detect:ifNone : como detect:, pero permite ejecutar un bloque si no se encuentra ningún elemento.
reduce: toma un bloque de dos o más parámetros de entrada y hace fold de los elementos de izquierda a derecha (foldl de funcional).

Ejemplo de iteración:

```
mínimo: 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
```

¿¿ ??