Clase Práctica 9 y 10 : Programación Lógica

Tomás Felipe Melli

June 21, 2025

Índice

1 Introducción

Prolog es un lenguaje de programación lógica. En este lenguaje los programas se escribem en un **subconjunto de la LPO** (o sea, se basa en la LPO pero no implementa la totalidad de su expresividad). Es **declarativo**, y por tanto, tenemos que especificar los **hechos**, **reglas de inferencia** y **objetivos** sin indicar cómo se obtiene este último a partir de los primeros.
El modelo de cómputo (motor lógico) de prolog está basado en **cláusulas de Horn y resolución SLD (Selective Linear Definite clause resolution)**. Es decir :

- Una cláusula de Horn es una fórmula lógica que tiene la forma $A :- B_1, B_2, ..., B_n$ que en lógica es equivalente a $(B_1 \wedge B_2 \wedge ... \wedge B_n) \rightarrow A$. Una regla lógica de una única conclusión.
- La resolución SLD (Selective Linear Definite clause resolution) es el mecanismo de inferencia que utiliza prolog para buscar respuestas a las consultas :
  - Selective: selecciona una submeta a resolver (de izquierda a derecha).
  - Linear: trabaja con una única cadena de resoluciones (una a la vez).
  - Definite clause: las cláusulas de Horn son un subconjunto especial con una sola cabeza.

  El algoritmo sería, dada una **base de conocimiento** y una consulta

  1. Elige una submeta de la consulta.
  2. Busca una cláusula cuya cabeza unifique con esa submeta.
  3. Reemplaza la submeta por el cuerpo de esa cláusula.
  4. Repite hasta:
     - Resolver todas las submetas (éxito), o
     - No encontrar coincidencias (falla → backtracking).

Decimos que Prolog es un lenguaje de **mundo cerrado**(*Closed World Assumption(CWA)*), es decir, sólo se puede suponer lo que se declaró explícitamente, todo lo que no pueda deducirse a partir de la base de conocimiento, se supone falso. Si pensamos en LPO, en esta , *el hecho de que algo no esté demostrado como verdadero no implica que sea falso − podría ser desconocido*, es la *Open World Assumption*.
Sólo hay un tipo en Prolog, los **términos**.

Base de conocimiento

Podemos pensar los programas como **bases de conocimiento** que describen el dominio del problema. Están formados por hechos y reglas de inferencia. La idea es consultar sobre esa base.

```
% base de conocimiento
zombie(juan).
zombie(valeria).

tomo_mate_despues(juan,carlos).
tomo_mate_despues(clara,juan).

infectado(ernesto).
infectado(X) :- zombie(X).
infectado(X) :- zombie(Y), tomo_mate_despues(Y,X).
```

Cláusulas y Consultas

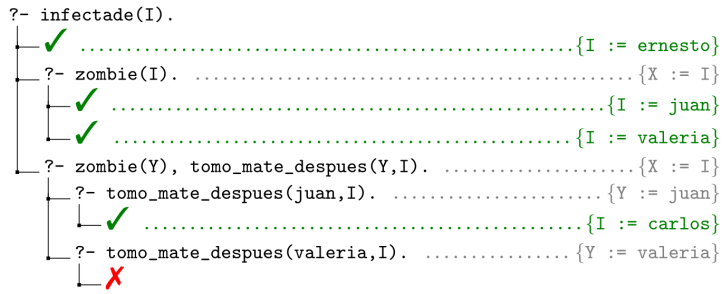
```
2 ?- zombie(juan).
true.

3 ?- tomo_mate_despues(juan,X).
X = carlos.

4 ?- infectado(I).
I = ernesto ;
I = juan ;
I = valeria ;
I = carlos ;
false.
```

Cuando escribimos la consulta por el infectado, si damos enter se termina, pero si presionamos ";" sigue consultado.

Seguimiento de consulta



Útil

- `swipl nombre-archivo.pl` para iniciar el intérprete con ese archivo
- `?- make.` para recargarlo
- `%` comentarios

Sintaxis

- **Variables :** tenemos 3 tipos:

  1. Normal : comienzan con mayúscula y se usan para obtener o comparar valores. Prolog las unifica y se pueden reutilizar. Ejemplo Hijo

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

  ?- padre(juan, Hijo).
  Hijo = maria ;
  Hijo = carlos.
  ```

  Que guarda el valor y podría reutilizarse.

  2. Anónima (_): Se usan cuando no importa el valor que puede tomar una variable. Sirven para ignorar argumentos. Ejemplo

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

  ?- padre(juan, _).
  true.
  ```

  No nos importa de quién, sólo si es padre.

  3. Variables que comienzan con _ : se utilizan para mostrar que no será reutilizada pero sí se comporta como una variable normal ya que guarda el valor.

- **Átomo** : un nombre constante, se usa para representar entidades, propiedades, entre otras cosas. Básicamente son valores indivisibles (sin estructura interna). Ejemplos

```
juan
'ejemplo con espacio'
```

- **Término Compuesto (Estructura)** : Consiste en un objeto con componentes. Tiene estructura interna. Consiste en

```
atomo(arg1, arg2, ..., argN)
```

Donde cada *arg* es un término y decimos que n es la **aridad** del término compuesto. Ejemplo:

```
tomo_mate_despues(clara,juan)
```

- **Término**: un término en prolog es la unidad básica de datos, es decir, cualquier cosa que puede aparecer como parte de una cláusula, un argumento de un predicado o una expresión lógica. (variable, número, átomo, término compuesto). Estos nos permitirán **construir conocimiento** y expresar **hechos y reglas**.

Reglas: hermano(X, Y) :- padre(Z, X), padre(Z, Y).

- **Números** : En Prolog, los números son **términos atómicos (átomo)** que pueden representar valores enteros, flotantes o expresiones aritméticas. Pero ojo! No podemos hacer **comparaciones numéricas** como estamos acostumbrados...

```
?- 2 + 2 = 4.
false.
```

  Ya que en prolog, se comparan estructuras, y en este caso $2+2$ es una estructura que no fue evaluada. Para realizar comparaciones contamos con **comparadores evaluativos**

  - $=:=$ **igualdad numérica**
  - $=\backslash=$ **distinto numéricamente**
  - **Comparaciones estandard** : $<, >, =<, >=$ (fijarse que en prolog no podemos hacer aparecer una flecha acá)

  Contamos con **operadores aritméticos** como

  - $+, -, *, /$
  - `div , mod`
  - `abs`
  - `max(X,Y), min(X,Y)`
  - `sqrt(X)`
  - `round(X), floor(X), ceiling(X), truncate(X)`

  Algo interesante de los números es el **predicado** `is`. Este se usa para asignar resultados numéricos. Este predicado existe ya que prolog no es simbólico por defecto, es decir, no hace cuentas automáticamente ya que trata a los términos como estructuras y es evaluativo si se le indica (con **is** o los comparadores evaluativos).

```
?- X = 3 + 4.
X = 3+4.  % No lo eval a

?- X is 3 + 4.
X = 7.    % Ahora s

?- X is Y + 1.
ERROR: Arguments are not sufficiently instantiated
```

  Falla si Y no tiene valor aún. **is** necesita que la derecha esté completamente instanciada.

- **Cláusula** : una cláusula es una unidad básica de conocimiento. **Hechos** (cláusulas sin cuerpo), **reglas**(cláusulas con cuerpo) y **negación**(cláusulas vacías) son unidades de conocimiento. Ejemplos

```
gato(felix).    % hecho

hermano(X, Y) :- padre(Z, X), padre(Z, Y).  % regla

:- false.   % negacion
```

- **Predicado** : una colección de cláusulas.
- **Objetivo (goal)** : es una expresión lógica que le pedimos a Prolog que pruebe.

```
?- objetivo.
```

  con cierta base de conocimiento, prolog intentará demostrarlo y si lo logra nos da

```
true.
```

  Podemos usar varios objetivos separados por coma también

```
?- padre(juan, maria), padre(juan, pedro).
```

  Podemos consultar con variables también

```
?- padre(juan, X).
```

2 Ejercicio : Representar los N_0

Queremos representar los naturales con el `cero` y `suc(X)`

```
% Ejercicio - Naturales con el 0
natural(cero).
natural(suc(M)) :- natural(M).


mayorA2(suc(suc(suc(_)))). % podemos hacerlo expl citamente
% mayorA2(suc(suc(suc(X)))) :- natural(X). de manera de generar todos

esPar(cero).
esPar(suc(suc(X))) :- esPar(X). % si X es par, entonces el sucesor del sucesor tambi n lo es

menor(cero, suc(_)). % cero es m s chico que todos
menor(suc(X),suc(Y)) :- menor(X,Y).
```

```
3 ?- menor(cero,uno).
false.

4 ?- menor(cero,X).
X = suc(_).

5 ?- menor(suc(cero),cero).
false.

6 ?- menor(X,suc(suc(suc(cero)))).
X = cero .

7 ?- menor(X,Y).
X = cero,
Y = suc(_) .
```

3 Sustitución y Unificación

Una sustitución es una función $S : Var \to Term$ donde $Term$ es el conjunto de todos los términos. Usamos estas sustituciones con el **mgu** para igualar literales y aplicar la regla de resolución. En particular, Prolog lo que hace es: recibe un programa **P** y un **goal** $G_1, \ldots, G_n$ (por ejemplo : ?- padre(juan, X), hermano(X, Y).); queremos saber si el goal es consecuencia lógica de P. La regla que se utiliza es :

$$\frac{G_1, \ldots, G_n \qquad H :- A_1, \ldots, A_k \qquad S \texttt{ es el MGY de } G_1 \texttt{ y } H}{S(A_1, \ldots, A_K, G_2, \ldots, G_n)}$$

Es decir, **si G_1 unifica con la cabeza H de una cláusula del programa, entonces sustituye G_1 por el cuerpo $A_1, \ldots, A_k$ de esa cláusula y aplica la sustitución S a todo (incluido $G_2, \ldots, G_n$)**. Hagamos un ejemplo para visualizar cómo Prolog aplica esta regla :
Sea la base de conocimiento :

```
padre(juan, maria).
padre(juan, pedro).
abuelo(X, Y) :- padre(X, Z), padre(Z, Y).
```

Queremos consultar :

```
?- abuelo(juan, Y).
```

Queremos saber de quién es abuelo juan, o sea, $G_1 = abuelo(juan, Y)$. Entonces, buscamos la cláusula para aplicar, que en este caso es $abuelo(X, Y)$ que llamaremos H y por tanto, $A_1 = padre(X, Z)$ y $A_2 = padre(Z, Y)$. Unificamos G_1 con H y nos queda

$G_1 \stackrel{?}{=} H$

$abuelo(juan, Y) \stackrel{?}{=} abuelo(X, Y)$

$abuelo(juan, Y) \stackrel{?}{=} abuelo(juan, Y)$ `con S = { X := juan}`

Con esto...

$$\frac{abuelo(juan, Y) \qquad abuelo(X, Y) : padre(X, Z), padre(Z, Y)}{padre(juan, Z), padre(Z, Y)}$$

Ahora tenemos nuevos objetivos : $G'_1 = padre(juan, Z)$ *y* $G'_2 = padre(Z, Y)$
Tratamos de unificar por un lado con... `S2 :={ Z := maria }`

$$\frac{\dfrac{abuelo(juan, Y) \qquad abuelo(X, Y) : padre(X, Z), padre(Z, Y)}{padre(juan, Z), padre(Z, Y)} \qquad padre(juan, maria)}{padre(maria, Y)}$$

Pero como **falla**, hace **backtracking** y tratamos de unificar por el otro con... `S3 :={ Z := pedro }`

$$\frac{\dfrac{abuelo(juan, Y) \qquad abuelo(X, Y) : padre(X, Z), padre(Z, Y)}{padre(juan, Z), padre(Z, Y)} \qquad padre(juan, pedro)}{padre(pedro, Y)}$$

Pero en ambos casos, **falla**. Como exploró todas las ramas en ninguna pudo derivar exitosamente, **concluye false**.("No hay ninguna manera de satisfacer el goal con las reglas y hechos dados"). Esta forma de explorar las ramas sigue el esquema del **DFS**, es por ello que el orden de las cláusulas y los literales dentro de cada regla afecta a la eficiencia. Prolog intenta resolver **los goals de izquierda a derecha**.

4 Reversibilidad

La reversibilidad en prolog es la capacidad que tiene para ejecutar un predicado en múltiples direcciones, es decir,

- Usar un predicado para generar soluciones a partir de variables no instanciadas.
- Usar el mismo predicado para verificar si una solución cumple ciertas condiciones.

O sea, se puede usar un predicado para consultar, generar o probar, dependiendo de qué variables estén instanciadas o libres. Veamos un ejemplo claro

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

Podemos consultar, es decir, le damos los argumentos y queremos saber el resultado....

```
3 ?- append([1,2], [3], R).
R = [1, 2, 3].
```

O, podemos pasarle el resultado y queremos saber qué argumentos cumplen...

```
4 ?- append(A, B, [1,2,3]).
A = [],
B = [1, 2, 3] ;
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
A = [1, 2, 3],
B = [] ;
```

Recién hablamos de variables instanciadas o libres, esto se convierte en un problema para la reversibilidad por ejemplo en el siguiente predicado

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

Ya que, podemos consultar por

```
6 ?- suma(2,3,Z).
Z = 5.
```

Pero si queremos usar **is** con variables no instaciadas, falla. Esto es un claro ejemplo de un predicado no reversible (ya que evalúa, no unifica)

```
7 ?- suma(2,Z,3).
ERROR: Arguments are not sufficiently instantiated
```

Tenemos una notación especial a la hora de leer una especificación de un predicado (modo de instanciación):

- **+X** : debe estar instanciado
- **-X** : no debe estar instanciado
- **?X** : puede o no debe estar instanciado

Con esta notación nos piden definir el predicado **entre(+X,+Y,-Z)** que sea verdadero cuando el número $\in$ Z esté comprendido entre los entreros X e Y (inclusive). O sea nos piden algo como

```
?- entre(1,3,Z).
Z = 1;
Z = 2;
Z = 3;
false.
```

Entonces, definimos el predicado

```
entre(X,Y,X) :- X =< Y.
```

Que verifica si $X \leq Y$ con lo cual, $Z = X$, si no lo es, tenemos que definir una regla

```
entre(X,Y,Z) :- X < Y, X1 is X + 1, entre(X1, Y, Z).
```

que nos permite incrementar X para explorar el siguiente número en el rango. En particular, poder generar lo pedido :

```
17 ?- entre(1,3,Z).
Z = 1 ;
Z = 2 ;
Z = 3 ;
false.

18 ?- entre(1,5,3).
true .

19 ?- entre(1,2,3).
false.
```

5 Listas

Las listas en Prolog son una colección ordenada de elementos, se definen usando corchetes [ ]. La estructura interna es $[Head \mid Tail]$. Un ejemplo

```
[1, 2, 3]  =  [1 | [2 | [3 | [ ]]]]
```

Podemos por ejemplo definir la función *longitud*

```
longitud([], 0).
longitud([_|T], N) :- longitud(T, N1), N is N1 + 1.
```

Nos piden definir:

1. `long(+L,-N)` que relaciona una lista con su longitud.
2. `sacar(+X, +XS, -YS)` que elimina todas las ocurrencias de X.
3. `sinConsecRep(+XS, -YS)` que elimina repeticiones consecutivas

```
long([], 0).
long([_|T], N) :- long(T, N1), N is N1 + 1.

sacarTodas(_, [], []).
sacarTodas(X, [X|T], R) :- sacarTodas(X, T, R).
sacarTodas(X, [H|T], [H|R]) :- X \= H, sacarTodas(X, T, R).

sinConsecRep([], []).
sinConsecRep([X], [X]).
sinConsecRep([X,X|T], R) :- sinConsecRep([X|T], R).
sinConsecRep([X,Y|T], [X|R]) :- X \= Y, sinConsecRep([Y|T], R).
```

Obtenemos entonces ...

```
39 ?- sacarTodas(2,[1,2,2,3,2],L).
L = [1, 3] ;

40 ?- sinConsecRep([1,2,2,3,2],L).
L = [1, 2, 3, 2] ;
```

A continuación :

1. `prefijo(+L, ?P)`
2. `sufijo(+L, ?S)`
3. `sublista(+L, ?SL)`
4. `insertar(?X, +L, ?LX)`
5. `permutacion(+L, ?P)`

```
prefijo(L, P) :- append(P, _, L).

sufijo(L, S) :- append(_, S, L).

sublista(L, SL) :- append(_, L2, L), append(SL, _, L2).

insertar(X, L, LX) :- append(P, S, L), append(P, [X|S], LX).

sacarUna(X, [X|T], T).
sacarUna(X, [H|T], [H|R]) :- sacarUna(X, T, R).

permutacion([], []).
permutacion(L, [X|P]) :-sacarUna(X, L, R), permutacion(R, P).
```

Con esto,

```
25 ?-  prefijo([1,2,3], P).
P = [] ;
P = [1] ;
P = [1, 2] ;
P = [1, 2, 3] ;
false.

25 ?- sufijo([1,2,3], S).
S = [1, 2, 3] ;
S = [2, 3] ;
S = [3] ;
S = [] ;
false.

26 ?- sublista([1,2,3], SL).
SL = [] ;
SL = [1] ;
SL = [1, 2] ;
SL = [1, 2, 3] ;
SL = [] ;
SL = [2] ;
SL = [2, 3] ;
SL = [] ;
SL = [3] ;
SL = [] ;
false.

27 ?- insertar(9, [1,2,3], LX).
LX = [9, 1, 2, 3] ;
LX = [1, 9, 2, 3] ;
LX = [1, 2, 9, 3] ;
LX = [1, 2, 3, 9] ;
false.

33 ?- permutacion([1,2,3], P).
P = [1, 2, 3] ;
P = [1, 3, 2] ;
P = [2, 1, 3] ;
P = [2, 3, 1] ;
P = [3, 1, 2] ;
P = [3, 2, 1] ;
false.
```

6 Estructuras parcialmente instanciadas

```
capicua([]).
capicua([_]).
capicua([H|T]) :- append(M, [H], T), capicua(M).

capicuaB ( L ) : - reverse (L , L ).
```

Qué pasa si L no está instanciada ?

```
42 ?- capicua(L).
L = [] ;
L = [_] ;
L = [_A, _A] ;
L = [_A, _, _A] ;
L = [_A, _B, _B, _A] ;
L = [_A, _B, _, _B, _A] ;
L = [_A, _B, _C, _C, _B, _A] ;
L = [_A, _B, _C, _, _C, _B, _A] .
```

Consideremos ...

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

Miremos las siguientes consultas

```
44 ?- member(2,[1,2]).
true .

45 ?- member(X,[1,2]).
X = 1 ;
X = 2 ;
false.

46 ?- member(5,[X,3,X]).
X = 5 ;
X = 5 ;
false.

1 ?- member(2,[1,2,2]).
true .

2 ?- length(L,2), member(5,L), member(2,L).
L = [5, 2] ;
L = [2, 5] ;
false.
```

7 Nomenclatura para patrones de instanciación

Por convención:

- `p(+A)` indica que `A` debe proveerse instaciado
- `p(-A)` indica que `A` no debe proveerse instanciado
- `p(?A)` indica que `A` puede o no proveerse instanciado

*Existe un caso en el cuál un argumento puede aparecer **semi-instanciado**(es decir, que contiene variables libres) como $[p, r, o, X, o, \_]$ que unifica con $[p, r, o, l, o, g]$.
Tenemos una serie de predicados útiles :

- `var(A)` que tiene éxito si A **es variable libre**
- `nonvar(A)` que tiene éxito si A **no es variable libre**
- `ground(A)` que tiene éxito si A **no contiene variables libres**

8 Ejercicio : iésimo

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

Tenemos que evitar:

```
iesimo(I, [X|_], X) :- I = 0.
```

Ya que en Prolog el = es un predicado de unificación, no una comparación. Si I ya está instanciado, funciona. Pero si no, no busca valores para que I = 0 se cumpla; solo intenta unificar I con 0. Por otro lado, sin la condición $K > 0$, Prolog intentaría hacer backtracking innecesario: si preguntamos por$K = 0$, entra al caso base y puede intentar la recursiva (aunque falle después). Al poner explícitamente $K > 0$ en la recursiva, evitamos este solapamiento y las reglas quedan disjuntas.
Nos preguntan si es reversible esta implementación **en I**, o sea, I puede tener la instanciación contraria (en este caso es no instanciada)?. Recordemos que predicado es reversible si puede usarse en más de una dirección, es decir, si no solo funciona para producir un resultado, sino también para "buscar" entradas que lo generan. Entonces, si *I no estuviese instanciada, que esperaríamos que devuelva?* Devolver los valores de la lista con su correspondiente índice. Luego, necesitamos **justificar por qué es reversible o no**. Para lograrlo habría que ver que no se cuelgue y que el predicado cumpla con lo pedido. Como el motor aritmético no es capaz de resolver $I > 0$ cuando I no está instanciada, no es reversible en I. La idea ahora es poder hacerla reversible en I de forma que si viene instanciada, también funcione.

```
% iesimo(?I, +L, -X)
iesimo2(I, L, X) :- nonvar(I), I >= 0, length(L1, I), append(L1, [X|_], L).
iesimo2(I, L, X) :- var(I), append(L1, [X|_], L), length(L1, I).
```

Esta llamada al revés asegura que el predicado instancie al N. O sea, primero hacemos la recursión para que se instancie y luego el resto. En casos de predicados infinitos conviene separar los casos en **var** y **nonvar**

```
ejemplo(0, _, _) :- !.
ejemplo(N, A, B) :- nonvar(N), N > 0, ... .
```

Sólo lo computo si N tiene un valor.

9 Ejercicio : desde

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

Este predicado nos dice si podemos ir de X a Y, el tema es que ambos tienen que estar instanciados. X ya que sino el motor aritmético no tiene idea cómo dar un valor para N. Y ya que en la recursión, queda libre, y por tanto, no sabe cuando parar.

```
% desdeReversible(+X, ?Y)
desdeReversible(X,Y) :- var(Y), desde(X,Y).
desdeReversible(X,Y) :- nonvar(Y), X =< Y.
```

10 Ejercicio : pmq(pares menores que)

Pares menores que es el predicado definido como `pmq(+X, -Y)`.
Pensemos en algo como:

```
pmq(X,Y) :- desde(0,Y), Y =<X, par(Y)
%par(+Y)
par(Y) :- 0 =:= Y mod 2.
```

Fijarse que para que `par` funcione adecuadamente tenemos que tener instanciado Y. El problema es usamos un predicado infinito para uno finito, la solución es usar **between**

```
%pmq(+X, -Y)
pmq(X, Y) :- between(0, X, Y),  par(Y).
% generamos los y, testeamos que cumplan
```

Esto es lo que se llama **Generate & Test**.

11 Generate & Test

Una técnica que vamos a usar para :

1. Generar todas las posibles soluciones de un problema (*candidatos a solución según cierto* ***criterio general***)
2. Testear cada uno de los candidatos generados (hacerlos fallar con el **criterio particular**)

El esquema general de un predicado que usa `G&T` se define mediante otros dos:

`pred(X1,...,Xn)` : − `generate(X1,...Xn)`, `test(X1,...,Xm)`

En este sentido, las tareas se dividen en :

- `generate(...)` que deberá **instanciar** ciertas variables.
- `test(...)` que deberá **verificar** si los valores instanciados pertenecen a la solución, pudiendo asumir que ya está instanciada.

12 Ejercicio : coprimos

El predicado `coprimos(-X,-Y)` debe instanciar todos los pares de números coprimos en X e Y. Nos sugieren usar `gcd(greatest common divisor)` del motor aritmético.

```
%coprimos(-X, -Y)
coprimos(X, Y) :- generarPares(X,Y), X > 0, Y > 0, 1 =:= gcd(X,Y).

%generarPares(-X, -Y)
generarPares(X,Y) :- desde(0, N), paresQueSuman(N, X, Y).

%paresQueSuman(+N, -X, -Y)
paresQueSuman(N, X, Y) :- between(0, N, X), Y is N-X.
```

coprimos(X, Y) es reversible en X e Y, porque: *generarPares*(X, Y) genera todos los posibles pares enteros no negativos. El filtro X ¿ 0, Y ¿ 0, 1 =:= gcd(X,Y) simplemente descarta los pares que no cumplen las condiciones, pero no depende de que X ni Y estén instanciados.

13 Predicado `not`

```
not(P) :- call(P), !, fail.
not(P).
```

`not(p(X1,...,Xn))` tiene éxito si **no existe** instanciación posible para las variables no instanciadas en $\{X1...Xn\}$ que haga que P tenga éxito. El `not` **no deja instanciadas** las variables luego de su ejecución.

14 Negación por Falla

14.1 `corteMásParejo`

```
%corteM sParejo(+L,-L1,-L2)
corteMasParejo(L, I, D) :- append(I, D, L), not(hayUnCorteMasParejo(I,D,L)).

hayUnCorteMasParejo(I,D,L) :- append(I2, D2, L), esMasParejo(I2, D2, I, D).

esMasParejo(I2, D2, I, D) :- sum_list(I2, SI2), sum_list(D2, SD2),
                             sum_list(I, SI), sum_list(D, SD),
                             abs(SI - SD) >  abs(SI2 - SD2).
```

14.2 `próximoPrimo`

```
%proximoPrimo(+N,-P) --> instancia P en el menor primo >= N
proximoPrimo(N, N2) :- N2 is N+1, esPrimo(N2).
proximoPrimo(N, P)  :- N2 is N+1, not(esPrimo(N2)), proximoPrimo(N2, P).


% esPrimo(+N)
esPrimo(N) :- N > 1, not(tieneUnDivisorNoTrivial(N)).
tieneUnDivisorNoTrivial(N) :- N1 is N-1, between(2, N1, D), 0 =:= N mod D.
```

15 Metapredicados

Los **metapredicados** son predicados que reciben otros predicados como argumentos o manipulan llamadas a predicados.

15.1 `setof(-Var, +Goal, -Set)`

Se usa para recolectar todas las soluciones de una consulta, agruparlas sin repeticiones y ordenarlas. Para lograrlo unifica `Set` con la lista ordenada y sin repetidos de `Var` que satisfacen Goal. Detalle importante, cuando decimos *agrupar por variable libre*, lo que hace prolog es dar todas las soluciones separadas según el valor que puede tomar aquella. Miremos este ejemplo :

```
padre(juan, ana).
padre(juan, lucas).
padre(mario, ines).

?- setof(Hijo, padre(Padre, Hijo), Lista).
Padre = juan,
Lista = [ana, lucas] ;
Padre = mario,
Lista = [ines].
```

Como `Padre` no está instanciada en `Var`, pero aparece en la consulta, prolog va a **agrupar**, es decir, unir los resultados de las distintas soluciones que surgen al unificar `Padre` con alguien. El tema es que en `setof`, los duplicados los saca. Veamos el ejemplo de la práctica :

```
primeraComponente([(X, )| ],X).
primeraComponente([ |XS],X) :- primeraComponente(XS,X).

?- setof(X,primeraComponente([(2,2),(1,3),(1,4)],X),L).
L = [1,2].
```

15.2 `bagof(-Var, +Goal, -Set)`

Es igual que `setof` pero se conservan duplicados y el orden de generación. La agrupación es análoga. Se podría utilizar el cuantificador existencial para avisarle que no agrupe y que devuelva todo junto:

```
padre(juan, ana).
padre(juan, lucas).
padre(mario, ines).

5 ?- bagof(Hijo, Padre^padre(Padre, Hijo), Lista).
Lista = [ana, lucas, ines].
```

El ejemplo de la práctica es,

```
primeraComponente([(X, )| ],X).
primeraComponente([ |XS],X) :- primeraComponente(XS,X).

?- bagof(X,primeraComponente([(2,2),(1,3),(1,4)],X),L).
L = [2,1,1].
```

15.3 `findall(-Var, +Goal, -Set)`

Con lo visto anteriormente con el uso de ^ en el `bagof`, el `findall` hace eso directo, o se, *ignora las variables libres dentro del* `Goal` que no sea `Var`. Ejemplo :

```
?- findall(X, member((X,Y),[(1,2),(3,4)]),L).
L = [1,3].
```

15.4 `maplist`

Básicamente aplica un predicado a cada elemento (o a pares de elementos) de una lista o listas. Tenemos :

15.4.1 `maplist(+Goal, ?List)`

En este caso el `Goal` (predicado) a aplicar sobre cada elemento de la lista deberá tener **aridad 1**. La forma de pasar el predicado sin los paréntesis ni argumentos explícitos se llama "referencia a predicado" o también a veces "predicado parcialmente aplicado". Si miramos cómo está hecho `maplist` :

```
maplist(_, []).
maplist(Pred, [X|Xs]) :-
    call(Pred, X),
    maplist(Pred, Xs).
```

Vemos que hace el `call` al predicado, que básicamente es un *metapredicado* también que lo que hace es invocar un predicado que lo tenemos como dato y no como un nombre fijo. Miremos el ejemplo de la práctica :

```
esPar(X) :- mod(X,2) =:= 0.
uno(1).
?- maplist(esPar,[2,6,8]).
true.
?- length(L,4), maplist(uno,L).
L = [1, 1, 1, 1].
```

15.4.2 `maplist(+Goal, ?List1, ?List2)`

Este es análogo, nada más que el predicado en `Goal` debe tener aridad 2, y por tanto admite otro argumento. El ejemplo es :

```
sumarK(K,X,Y) :- Y is X+K.
?- maplist(sumarK(4),[1,2,3],L).
L = [5,6,7].
```

Como ya mencionamos esta manera de pasar los argumentos, se puede llamar también **currificación parcial**.

15.5 `limit(+Count, +Goal)`

Básicamente limita el número de soluciones del `Goal`. Ejemplo :

```
?- limit(5,desde(1,X)).
X = 1;
X = 2;
X = 3;
X = 4;
X = 5.
```

15.6 `forall(+Gen, +Cond)`

Es un metapredicado que verifica que todas las soluciones de `Gen` cumplan `Cond`. Es como un Generate and Test.

16 Generación infinita : triángulos

```
%perimetro(?T, ?P)
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))), triplasQueSuman(P, A, B, C), esTriangulo(tri(A,B,C)).

%triplasQueSuman(?P, -A, -B, -C)
triplasQueSuman(P, A, B, C) :-
    desdeReversible(0,P),
    between(1,P,A), between(1,P,B),
    C is P - A - B,  C > 0.


%triangulo(-T)
triangulo(T) :- perimetro(T, _).
```