Tema 1. PROGRAMACION en PROLOG

1.1. Sintaxis: Hechos, Preguntas y Reglas.

1.2. Sintaxis: Objetos estructurados. Listas.

1.3. Computación: Unificación y Regla de Resolución.

1.4. Estrategia de Resolución usada en PROLOG.

1.5. Recorrido en el Espacio de Búsqueda.

1.6. Aspectos Declarativos y Procedurales.

TEMA 1. PROGRAMACION en PROLOG

La programación lógica está basada en la lógica de primer orden LPO (o lógica de predicados). Un programa PROLOG consiste en un conjunto de sentencias (o fórmulas), de la forma: **A** :- $\mathbf{B_1}$ **, ... ,** $\mathbf{B_n}$ **.** con $n \geq 0$.

- Cuando n>0, la sentencia se escribe **A** :- $\mathbf{B_1}$ **, ... ,** $\mathbf{B_n}$ **.** y se denomina **regla.**
- Si n=0, la sentencia se escribe **A .** y se denomina **hecho.**

Informalmente, un programa PROLOG consiste en un conjunto de hechos (afirmaciones simples) y de reglas que afirman "El hecho A es cierto si son ciertos los hechos B1 y … y Bn". Esto es, las reglas servirán para deducir nuevos hechos a partir de otros.

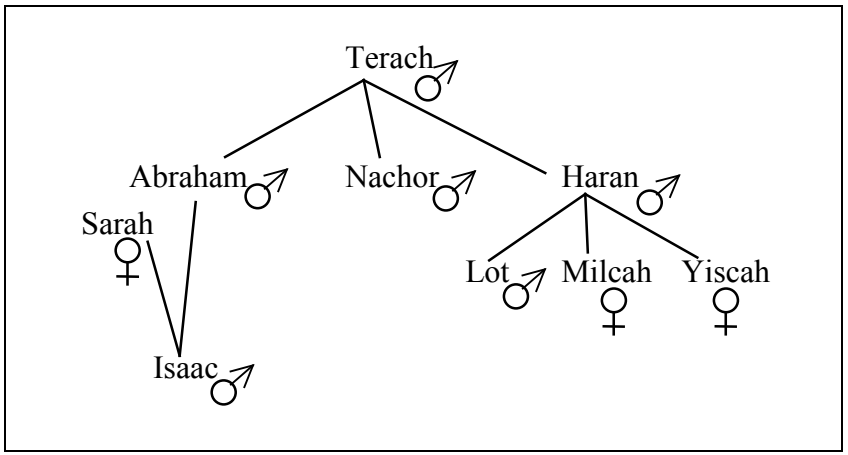
A un programa PROLOG se le hacen **preguntas**. Una pregunta se escribe como: **?A1, A2, …, Am.** siendo m>0. Informalmente, dicha pregunta se leerá: "¿Son ciertos los hechos A1 y A2 y … y Am?".

1.1. Sintaxis: Hechos, Preguntas y Reglas

Los **hechos** son las sentencias más sencillas. Un hecho es una fórmula atómica o átomo: $\mathbf{p(t_1, ..., t_n)}$ e indica que se verifica la relación (predicado) **p** sobre los objetos (términos) $\mathbf{t_1, ..., t_n}$.

Ejemplos: es_padre(abraham, isaac) es un hecho que indica "Abraham es padre de Isaac"
es_hombre(abraham) es un hecho que indica "Abraham es un hombre"
suma(3,2,5) es un hecho que indica "la suma de 3 y 2 es 5"

Un conjunto de hechos constituye un programa (la forma más simple de programa lógico) que puede ser visto como una base de datos que describe una situación. Por ejemplo, el **Programa 1** refleja la base de datos de las relaciones familiares que se muestran en el siguiente gráfico.



Programa 1:

1. es_padre(terach, abraham).
2. es_padre(terach, nachor).
3. es_padre(terach, haran).
4. es_padre(abraham, isaac).
5. es_padre(haran, lot).
6. es_padre(haran, milcah).
7. es_padre(haran, yiscah).
8. es_madre(sarah, isaac).
9. es_hombre(terach).
10. es_hombre(abraham).
11. es_hombre(nachor).
12. es_hombre(haran).
13. es_hombre(isaac).
14. es_hombre(lot).
15. es_mujer(sarah).
16. es_mujer(milcah).
17. es_mujer(yiscah).

Todos los hechos de este programa son *hechos de base* (sin variables), pero también se pueden introducir *hechos con variables* como axiomas, por ejemplo: **suma(0, X, X).** En ellos, las variables se consideran cuantificadas *universalmente*. Es decir, **∀x suma(0, x, x).**
Al igual que el hecho es_mujer(sarah) establece la verdad de la sentencia "Sarah es mujer", el hecho suma(0, X, X) establece la verdad para cualquier valor que pueda tomar la variable, es decir, nos dice que "para todo término x, la suma de 0 con x es x" . Equivale a un conjunto de hechos de base como serían: suma(0, 1, 1), suma(0, 2, 2), etc.

Una vez que se tiene el programa describiendo una situación, se pueden hacer **preguntas** para obtener información acerca de él. Por ejemplo, podemos hacer preguntas al **Programa 1** del tipo siguiente:

PREGUNTA	SIGNIFICADO	RESPUESTA
?es_padre(abraham,isaac).	¿es padre Abraham de Isaac?	**Sí**, pues encuentra un hecho que lo afirma.
? es_padre(abraham,lot).	¿es padre Abraham de Lot?	**No**, pues no hay ningún hecho que lo afirme.
? es_padre(abraham,X).	¿existe un X tal que es padre Abraham de X?	**Sí**. El programa contestará dando para X los valores que verifican dicha relación: **X=isaac**.
? es_padre(haran,X).	¿existe un X tal que es padre Haran de X?	**Sí**, dando todas las soluciones posibles: **X=lot ; X=milcah ; X=yiscah**.

Estas preguntas sencillas se llaman fines. Como puede verse en la lectura de las preguntas, en éstas se consideran las variables cuantificadas *existencialmente*. También se pueden hacer preguntas más complejas, como conjunción de fines, de la forma:

PREGUNTA	SIGNIFICADO	RESPUESTA
? es_padre(haran,lot), es_hombre(lot).	¿es padre Haran de Lot y es hombre Lot?	**Sí**, pues ambos fines son ciertos.
? es_padre(abraham,lot), es_hombre(lot).	¿es padre Abraham de Lot y es hombre Lot?	La respuesta es **no** pues algún fin no es cierto.
? es_padre(abraham,X), es_hombre(X).	¿existe un X tal que es padre Abraham de X y es hombre X?	La respuesta es **sí** pues ambos fines son ciertos para algún valor de X. El programa contestará **X=isaac**.
? es_padre(haran,X), es_mujer(X).	¿existe un X tal que es padre Haran de X y es mujer X?	El programa contestará que **sí**, dando todas las soluciones posibles: **X=milcah ; X=yiscah**.

Las **reglas** son sentencias de la forma: **A** :- $\mathbf{B_1}$ **, ... ,** $\mathbf{B_n}$ **.** donde **A** y cada $\mathbf{B_i}$ son átomos. **A** es la cabeza de la regla y los $\mathbf{B_i}$'s componen el cuerpo de la regla.

Ejemplo: Una regla que define la relación "ser hijo" a partir de las relaciones dadas podría ser: **es_hijo(X,Y) :- es_padre(Y,X), es_hombre(X).**
que se leería de la forma: "para cualesquiera X e Y, X es hijo de Y si Y es padre de X y X es hombre" ya que se corresponde con la fórmula lógica: $\forall x \, \forall y \, ((es\_padre(y,x) \wedge es\_hombre(x)) \rightarrow es\_hijo(x,y))$.
De igual forma se definirían otras relaciones mediante reglas:

```
es_hija(X,Y) :- es_padre(Y,X), es_mujer(X).
es_abuelo(X,Z) :- es_padre(X,U), es_padre(U,Z).
```

La última regla podría leerse "para cualesquiera X, Z y U, X es abuelo de Z si X es padre de U y U es padre de Z", que se corresponde con la fórmula $\forall x \, \forall z \, \forall u \, (es\_padre(x,u) \wedge es\_padre(u,z) \rightarrow es\_abuelo(x,z))$, o bien como "para cualesquiera X y Z, X es abuelo de Z si existe un U tal que X es padre de U y U es padre de Z" que se corresponde con la fórmula $\forall x \, \forall z \, (\exists u \, (es\_padre(x,u) \wedge es\_padre(u,z)) \rightarrow es\_abuelo(x,z))$.

Esto es debido a que ambas fórmulas son lógicamente equivalentes.
Con estas tres nuevas relaciones entre objetos y los hechos de base anteriores podemos crear el siguiente **Programa 2**.

Programa 2 = **Programa 1** +

```
18. es_hijo(X,Y):- es_padre(Y,X), es_hombre(X).
19. es_hija(X,Y):- es_padre(Y,X), es_mujer(X).
20. es_abuelo(X,Z):- es_padre(X,Y), es_padre(Y,Z).
```

Ahora podemos hacer preguntas al **Programa 2** sobre las nuevas relaciones introducidas:

PREGUNTA	SIGNIFICADO	RESPUESTA
? es_hijo(lot,haran).	¿es hijo lot de Haran?	**Sí**, pues este hecho puede deducirse, ya que según la regla 18 es equivalente a preguntar: ?es_padre(haran,lot), es_hombre(lot).
? es_hija(X,haran).	¿existe un X tal que es hija X de Haran?	Según la regla 19 es equivalente a preguntar ? es_padre(haran,X), es_mujer(X). por lo que el programa contestará que **sí**, dando las soluciones: **X=milcah ; X=yiscah**.

Otras preguntas y respuestas a este programa serían:

PREGUNTA	RESPUESTA
? es_abuelo(terach,isaac).	**sí**
? es_abuelo(terach,X), es_mujer(X).	**X=milcah ; X=yiscah**
? es_madre(X,Y).	**X=sarah Y=isaac**
? es_abuelo(X,Y), es_hombre(Y).	**X=terach Y=isaac ; X=terach Y=lot**

1.2. Sintaxis: Objetos estructurados. Listas.

Los objetos estructurados (o estructuras) en PROLOG son términos de la forma **$f(t_1, ..., t_n)$** donde **f** es un símbolo de función n-ario, o funtor, y **$t_1, ..., t_n$** son a su vez términos (que pueden ser constantes, variables o a su vez estructuras).

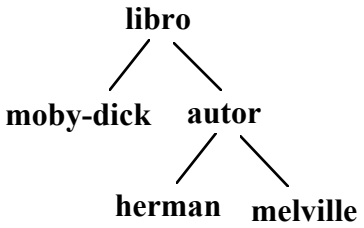
Ejemplos de estructuras son las siguientes:

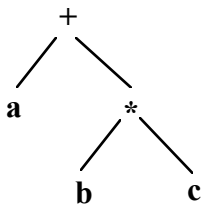
libro(moby-dick, autor(herman, melville))

a+(b*c) ó +(a, *(b, c))

Las estructuras se suelen representar por árboles donde el funtor es un nodo y los componentes son los subárboles que cuelgan de dicho nodo.

Los ejemplos anteriores se representarían de la forma:





Introduciendo variables en una estructura se puede obtener información. Por ejemplo, supongamos que se tiene el siguiente programa **BIBLIOTECA**

```
esta-en-biblio( libro(moby_dick, autor(herman, melville)) ).
esta-en-biblio( libro(sueños_de_robot, autor(isaac, asimov)) ).
esta-en-biblio( libro(the_art_of_PROLOG, autor(leon, sterling)) ).
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
esta-en-biblio( revista(eroski, mensual) ).
esta-en-biblio( revista(autopista, semanal) ).
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

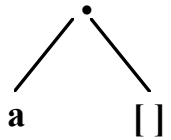
A este programa **BIBLIOTECA** se le pueden hacer preguntas como las siguientes:

PREGUNTA	RESPUESTA
? esta-en-biblio(libro(X, autor(isaac,asimov))).	Dará los libros de Isaac Asimov que están en la biblioteca. **X=sueños_de_robot ;** ...
? esta-en-biblio(libro(moby_dick, X)).	Dará el autor del libro moby-dick. **X=autor(herman, melville)**
? esta-en-biblio(revista(X, mensual)).	Las respuestas serán las revistas mensuales de la biblioteca. **X=eroski ;** ...

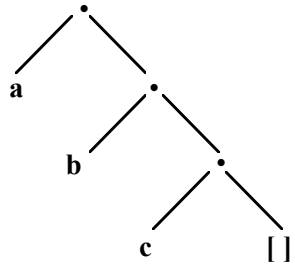
Una estructura de datos muy común en la programación no numérica son las listas. Estas serán unas estructuras PROLOG especiales.
Una **lista** es una secuencia ordenada de elementos que puede tener cualquier longitud. Los elementos de una lista (como los de cualquier otra estructura) son términos, los cuales pueden ser en particular otras listas.
Las listas pueden representarse como un tipo especial de árbol. Una lista es o bien la lista vacía, sin elementos, denotada **[ ]**, o bien una estructura cuyo funtor se denota "**.**" (punto) y con dos componentes llamados "cabeza" y "resto" de la lista.

Ejemplos:

• La lista con un solo elemento **a** es **.(a, [ ])** y su árbol es:



• La lista que consta de los elementos **a**, **b** y **c** sería: **.(a, .(b, .(c, [ ])))** y en forma de árbol es:



Pero la forma sintáctica más habitual de escribir las listas es separando los elementos entre comas y toda la lista encerrada entre corchetes. Las listas anteriores se escribirían de la forma:

[a] y **[a,b,c]**

La forma de manejar las listas es dividiéndolas en *cabeza* (el primer argumento del funtor punto o el primer elemento de la lista) y *resto* (el segundo argumento del funtor punto o la lista formada por el resto de elementos de la lista).

LISTAS	CABEZA	RESTO
[a,b,c]	a	[b,c]
[a]	a	[ ]
[[el,gato], maulla]	[el,gato]	[maulla]
[el, [gato, maulla]]	el	[[gato, maulla]]
[X+Y, x+y]	X+Y	[x+y]
[ ]	____	____

La forma sintáctica en PROLOG para describir una lista con cabeza X y resto Y es la notación **[X|Y].**

Ejemplo:

1. **p([1, 2, 3]).**
2. **p([el, gato, estaba, [en, el, tejado]]).**

? p([X|Y]).
X=1 Y= [2,3];
X= el Y=[gato, estaba, [en, el, tejado]]

Se puede especificar más de un elemento en la cabeza, indicando primero, segundo,...y resto de la lista:

? p([X, Y | V]).
X=1 Y= 2 V=[3]
X= el Y=gato V=[estaba, [en, el, tejado]]

También se puede consultar la lista combinando las dos formas sintácticas anteriores ([a1, a2, …, an] y [A|B]):

? p([X, Y, Z, [U|V]]).
X= el Y= gato Z= estaba U= en V= [el, tejado]

NOTA:
La naturaleza de las listas, como estructuras con el funtor punto (.), puede verse mediante el predicado predefinido "display":

?- display([a, b, c]).
.(a, .(b, .(c, [])))

Comentarios sobre la notación usada:

La notación que se ha seguido aquí para presentar los programas y preguntas PROLOG se conoce como "la notación Prolog de Edimburgo".
La notación de los símbolos del alfabeto es la siguiente:

- **Los símbolos de predicado y de función** (excepto algunos predefinidos) se denotarán mediante una cadena de letras minúsculas (se puede incluir los caracteres numéricos y el '_') donde ***el primer carácter*** deberá ser necesariamente una ***letra minúscula***.
- **Los símbolos de constante** de igual forma que los anteriores, excepto las constantes numéricas que se representarán directamente.
- **Las variables** se denotarán como cadenas de caracteres (letras, números y algunos símbolos como '_' o '?'), siendo ***el primer carácter*** una ***letra mayúscula.***

Comentarios sobre la base lógica de Prolog: Cláusulas de Horn

Una pregunta en PROLOG (con variables X1, ..., Xp) tiene la forma **?B1, B2, ... , Bn** y se lee ¿existen valores para las variables x1, ..., xp tales que B1 y B2 y ...y Bn son ciertos?. Es decir, dicha ***pregunta*** se corresponde con la fórmula lógica: $\exists x1\ \exists x2 \ldots\ \exists xp\ (B1 \wedge B2 \wedge \ldots \wedge Bn)$
Conviene notar que la ***negación de esta pregunta*** se corresponde con la fórmula anterior negada:
$\neg\ (\exists x1\ \exists x2 \ldots\ \exists xp\ (B1 \wedge B2 \wedge \ldots \wedge Bn))$ que es lógicamente equivalente a la fórmula $\forall x1\ \forall x2 \ldots\ \forall xp\ (\neg B1 \vee \neg B2 \vee \ldots \vee \neg Bn)$.

Desde un punto de vista formal, cuando se hace una pregunta a un programa lógico, se está trabajando en realidad con el conjunto de fórmulas lógicas (***cláusulas***), cuantificadas universalmente, correspondientes a ***los hechos y reglas del programa junto con la negación de la pregunta***.
Cada cláusula es por tanto una ***disyunción de literales*** donde ***a lo sumo uno*** de ellos es ***positivo***. Se denominan ***cláusulas de Horn.***

- Una regla **A** :- $\mathbf{B_1 , \ldots , B_n}$ es una cláusula de Horn con un literal positivo y **n** negativos: $A \vee \neg B1 \vee \neg B2 \vee \ldots \vee \neg Bn$ con $n>0$.
- Un hecho **A** es una cláusula de Horn con un literal positivo y **0** negativos.
- La negación de una pregunta es una cláusula de Horn con todos sus literales negativos: $\neg B1 \vee \neg B2 \vee \ldots \vee \neg Bn$ con $n \geq 0$.

1.3. Computación: Unificación y Regla de Resolución

El cómputo que PROLOG realiza para contestar a una pregunta (y un programa dado) se basa en los conceptos de ***unificación*** y ***resolución***.

Unificación

Para definir formalmente la *unificación* son necesarias algunas definiciones previas.

Una **sustitución** es un conjunto finito de pares de la forma **{v1 → t1, v2 → t2, …, vn → tn}**, donde cada **vi** es una variable, cada **ti** es un término (distinto de **vi**) y las variables **vi** son todas distintas entre sí.
Cuando se aplica una sustitución a una expresión (regla, hecho o pregunta) se obtiene una nueva expresión, reemplazado en la expresión original cada aparición de la variable **vi** por el término **ti** (1<= i <= n).

Por ejemplo, dada la pregunta **c** = es_padre(Y,X), es_hombre(X), y la sustitución **s** ={Y → haran, X → lot}, la pregunta obtenida al aplicar **s** a **c** será **c s** = es_padre(haran, lot), es_hombre(lot).

Dadas dos sustituciones **s** y **s'**, se define la **operación de composición** **s•s'** de esta forma:
Si **s** ={v1 → t1, v2 → t2, …, vn → tn} y **s'**= {w1 → t'1, w2 → t'2, …, wn → t'n}, entonces
s•s' = {v1 → t1 s', v2 → t2 s', …, vn →tn s', w1→ t'1, w2 → t'2, …, wn → t'n} - {vi → ti s' | vi = ti s'} - {wj → t'j | wj ∈ {v1, …, vn}}

La operación de composición es asociativa y tiene como elemento neutro la sustitución vacía que denotamos por **ε**. Además, dada una expresión **c** y dadas dos sustituciones **s** y **s'** se verifica: **(c s) s' = c (s•s').** Esto es, aplicar a una expresión una sustitución tras otra es equivalente a haber aplicado directamente la composición de dichas sustituciones.

Por ejemplo, dada la pregunta **c** = es_padre(Y,X), es_hombre(X) y las sustituciones **s** ={Y → Z , X → lot} y **s'**= {Z → haran, X → juan}, la pregunta obtenida al aplicar primero **s** a **c** y después **s'** al resultado será: **(c s) s' = (** es_padre(Z, lot), es_hombre(lot) **) s' =** es_padre(haran, lot), es_hombre(lot). que puede ser obtenida como **c (s•s')** ya que **s•s'** = {Y → haran , X → lot**,** Z → haran}.

Un **unificador** de dos átomos **c1** y **c2**, es una sustitución **s** tal que **c1 s = c2 s**, es decir, *"s unifica o iguala c1 y c2"*.

Por ejemplo, dados c1= p(a,Y) y c2= p(X, f(Z)). Los siguientes son unificadores de c1 y c2:

s1 = {X → a, Y → f(a), Z → a} ya que c1 s1 = c2 s1 = p(a, f(a)).

s2 = {X → a, Y → f(f(a)), Z → f(a)} ya que c1 s2 = c2 s2 = p(a, f(f(a))).

Un **unificador más general** (**umg**) de dos átomos **c1** y **c2** es el unificador **g** de ambos tal que, cualquier unificador **s** de c1 y c2 se puede obtener como la composición de **g** con alguna otra sustitución **u**, es decir **s = g•u**. El unificador más general de dos átomos es único (salvo renombramiento de variables).

En el ejemplo anterior el umg de c1 y c2 sería: **g** = {X → a, Y → f(Z)} con c1 g = c2 g = p(a, f(Z)).
El resultado p(a, f(Z)) obtenido tras aplicar **g**, es *más general* que p(a, f(a)), obtenido aplicando **s1**, y que p(a, f(f(a))), obtenido aplicando **s2**. De hecho, estos dos átomos son casos particulares de p(a, f(Z)), para Z→ a y para Z→f(a) respectivamente. Es decir, **s1= g•u1** siendo **u1** ={Z → a} y **s2= g•u2** siendo **u2** ={Z → f(a)}.

Existe un algoritmo para decidir si dos átomos son unificables (esto es, si existe para ellos algún unificador). Este algoritmo siempre termina.
El resultado que obtiene es el siguiente: Si los átomos son unificables el algoritmo devuelve el umg de ellos, en otro caso, devuelve "fracaso".
A continuación veremos dicho algoritmo, en su versión original, debida a Robinson.

En el algoritmo de unificación se utiliza la noción de "conjunto de disparidad" de dos átomos, definido como el conjunto formado por los dos términos que se extraen de dichos átomos a partir de la primera posición en que difieren en un símbolo. Por ejemplo, el conjunto de disparidad de p(X, f(Y,Z)) y p(X, a) es **D** ={ f(Y,Z), a}, mientras que el conjunto de disparidad de p(X, f(Y,Z)) y p(X, f(a,b)) es **D** = {Y, a}.
Si **D = {V, t},** con **V** variable y **t** término, la comprobación de que la variable **V** no aparezca en **t** se denomina **"occur-check".**

Algoritmo de Unificación (de Robinson)

Entrada: dos átomos **c1** y **c2**

Salida: el umg **g** de **c1** y **c2** o fracaso

Algoritmo:

g := ε
fracaso := falso
mientras c1 g ≠ c2 g **y no** (fracaso)
    calcular conjunto de disparidad D de c1 g y c2 g
    si existe en D una variable V y un término t tal que V no aparece en t
        entonces g := g • {V → t}.
        sino fracaso := cierto
    fin si
fin mientras
si fracaso
    entonces devolver fracaso
    sino devolver g
fin si

Ejercicio: Aplica el algoritmo de unificación ¿qué devuelve en cada caso?

	ÁTOMO c1	ÁTOMO c2
1.	p([X, Y, Z, [U|V]])	p([el, gato, estaba, [en, el, tejado]])
2.	p([X, Y, Z])	p([juan, come, pan])
3.	p([gato])	p([X|Y])
4.	p([X, Y | Z])	p([maria, bebe, vino])
5.	p([[la, Y] | Z])	p([[X,liebre], [esta, aqui]])
6.	p([plaza | C])	p([plaza, mayor])
7.	p([buen, hombre])	p([hombre, X])
8.	p(X, f(X))	p(a, b)
9.	p(X, f(X))	p(a, f(b))
10.	p(X, f(X))	p(Y,Y).

Nota: Algunos sistemas no hacen el "occur-check", por lo que no hacen una unificación correcta. Por ejemplo, en SWI-Prolog se puede comprobar esto así:

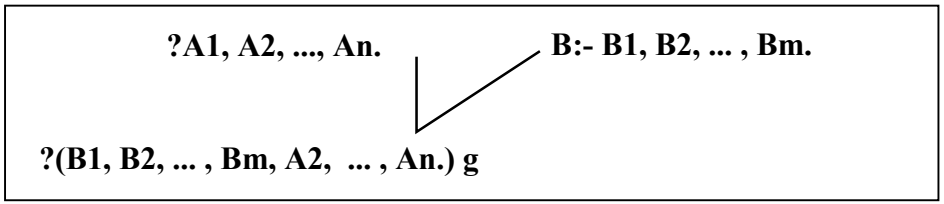
```
?- A = f(A).
   A = f(**)
```

```
?- unify_with_occurs_check(A, f(A)).
   No
```

Regla de Resolución

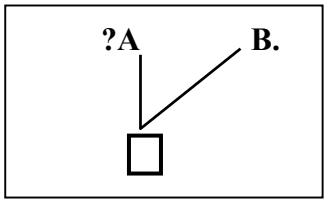
Una vez visto qué es el umg de dos átomos, se puede establecer la regla de computación que utiliza PROLOG. Dicha regla se denomina "regla de resolución" y obtiene, a partir de una pregunta y una sentencia del programa, una nueva pregunta. En concreto:
Dada la pregunta **?A1, A2, ,..., An**. donde A1 es un átomo de la forma **p(t1, …, tk)** y dada una sentencia del programa **B:- B1, B2, ... , Bm.** con cabeza **B** de la forma **p(s1, …, sk)**, si **A1** y **B** son unificables se aplica la regla de resolución para obtener una nueva pregunta (llamada *"resolvente"*) **? (B1, B2, ..., Bm**, **A2,..., An) g** siendo **g** el umg de los átomos $\mathbf{A_1}$ (primer fin de la pregunta) y **B** (cabeza de la regla).

Gráficamente:



En particular, cuando tenemos una pregunta con un único fin **?A** de la forma **p(t1, …, tk)** que se resuelve con un hecho del programa **B** de la forma **p(s1, …, sk)**, el resultado de aplicar la regla de resolución es la pregunta vacía, que lo expresaremos con .

Es decir, si existe un umg **g** tal que **A g = B g**, entonces la nueva pregunta es vacía lo que significa que la pregunta inicial es cierta para ese programa.



1.4. Estrategia de Resolución usada en PROLOG

Dado un programa PROLOG como conjunto de *hechos* y *reglas*, y dada una pregunta a dicho programa, el cómputo que se realiza es aplicar paso a paso la regla de resolución con los hechos y/o reglas del programa hasta conseguir que en alguno de esos pasos queden eliminados todos los fines de la pregunta (es decir, hasta llegar a la pregunta vacía).

En concreto, el procedimiento a seguir es el siguiente:
Dada una pregunta de la forma **?A_1, A_2, .., A_n**. se buscará la primera sentencia del programa **B:-B_1, ..., B_m**. tal que **A_1** y **B** sean unificables por un umg **g**. PROLOG elige (de entre las posibles sentencias cuya cabeza **B** se unifica con **A_1**) la primera introducida en el programa.

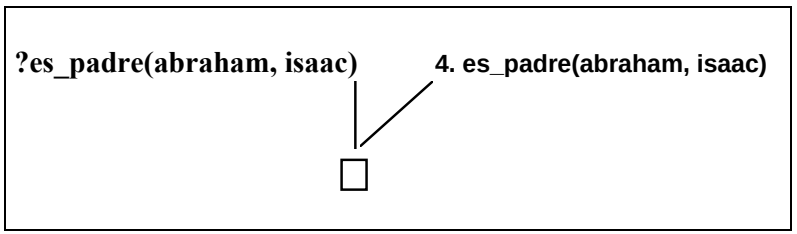
Por tanto, el orden en que se introducen las sentencias de un programa es significativo, entre otras cosas determinará el orden de las respuestas.
Entonces la nueva pregunta (el resolvente) será **?(B_1, ..., B_m, A_2, ..., A_n) g**.

El proceso continuará de igual forma hasta obtener la pregunta vacía (lo que corresponde a un éxito), o una pregunta para la cual no exista *resolvente* con ninguna sentencia del programa (lo que corresponderá a un fracaso). Esta estrategia de resolución que sigue PROLOG se denomina *estrategia de resolución SLD*. La secuencia de pasos desde la pregunta original hasta la pregunta vacía se llama *refutación SLD*.

Ejemplos

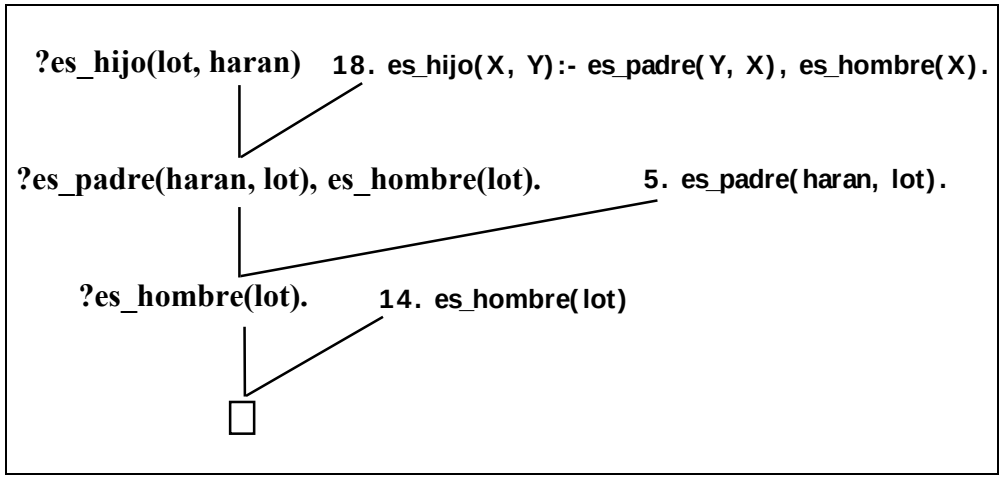
El proceso que se ha seguido para contestar las preguntas formuladas en el apartado anterior para el **Programa 2**, es el siguiente:

- **Pregunta 1: ?es_padre(abraham, isaac).**



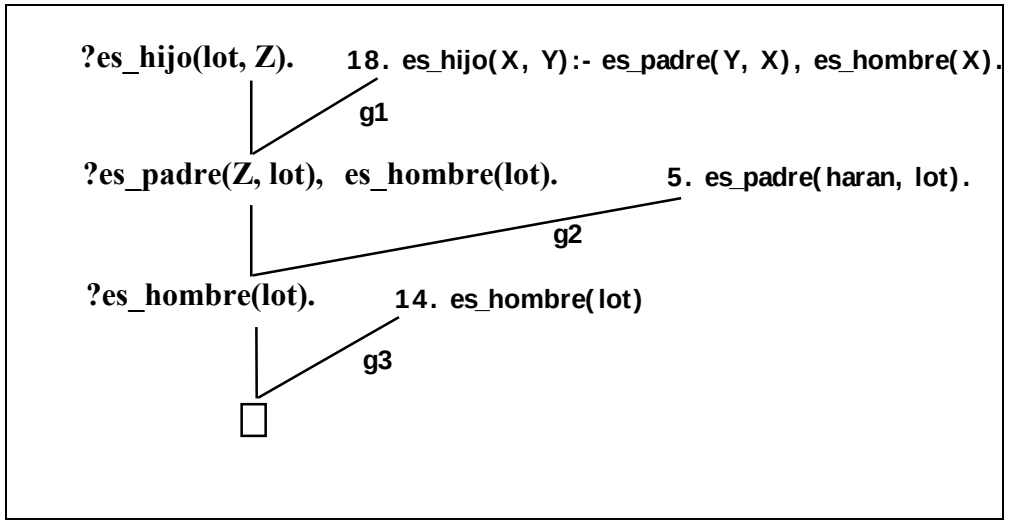
En este caso se obtiene en un solo paso un éxito: la pregunta vacía es el resolvente de la pregunta dada y de la sentencia 4. El hecho **es_padre(abraham, isaac)** es cierto para el programa, por tanto el sistema contesta **sí**.

• **Pregunta 2:** **?es_hijo(lot,haran).**



En este ejemplo se necesitan más pasos de resolución hasta llegar a un éxito. Como se ha llegado a la pregunta vacía (se ha encontrado una refutación), el hecho **es_hijo(lot,haran)** se deduce del programa, por lo que PROLOG contesta **sí**.

• **Pregunta 3:** **?es_hijo(lot,Z)**. En los casos en que la pregunta contiene variables, el funcionamiento es similar:



Como se ha encontrado una refutación, el sistema PROLOG debe contestar de forma afirmativa.
La fórmula **∃Z es-hijo(lot,Z)** se deduce del programa y la contestación sería **sí**. Sin embargo, interesa respuestas concretas, es decir, valores de **Z** que cumplan la pregunta.
Para ello, el sistema guardará las *"ligaduras"* de las variables con los valores tomados a lo largo del proceso.

En cada paso del proceso de resolución se obtiene un unificador más general (umg). La composición de todos estos unificadores da lugar a las ligaduras que han tomado las variables finalmente para llegar a la respuesta.

En el ejemplo:

g1 = umg (es_hijo(lot,Z), es_hijo(X,Y)) = {X→ lot, Y→ Z}

g2= umg (es_padre(Z,lot), es_padre(haran,lot)) = {Z→ haran}

g3 = umg (es_hombre(lot), es_hombre(lot)) = ε

Las ligaduras de las variables finalizado el proceso, vienen dadas por la composición de g1, g2 y g3:

g = g1•g2•g3 = {X→ lot, Y→ haran, Z→ haran}.

Si de este conjunto de ligaduras nos quedamos sólo con las correspondientes a las variables de la pregunta original, obtenemos la respuesta dada por PROLOG: **Z=haran**.

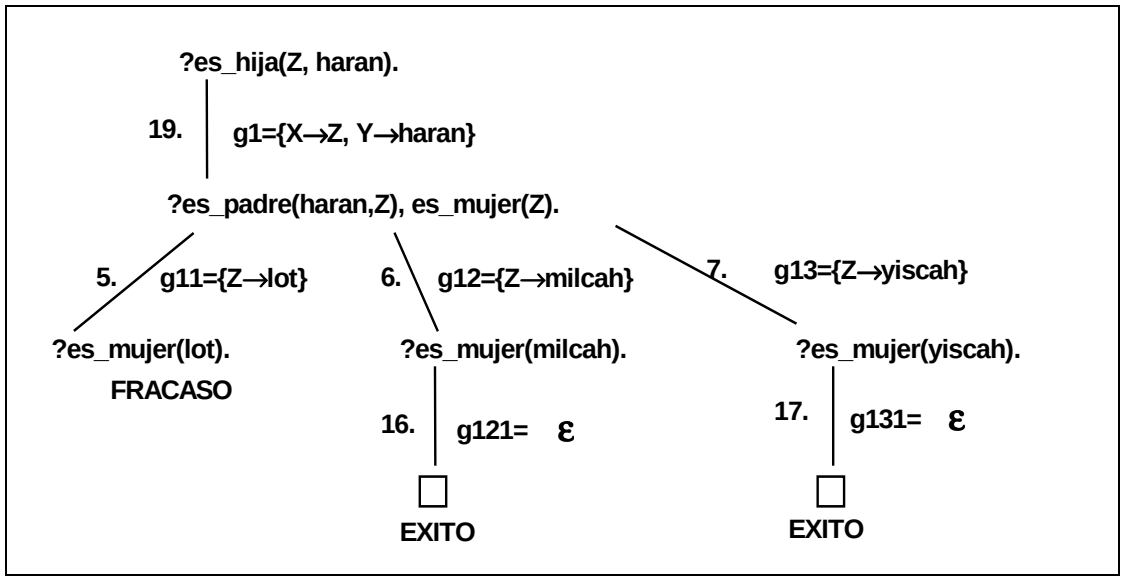
Distintas refutaciones para una misma pregunta original (y un mismo programa) pueden dar lugar a distintas respuestas, según sea el conjunto de ligaduras en cada refutación. PROLOG dará, si se le pide, todas las posibles respuestas a una pregunta. Esto significa que el sistema debe tener la capacidad de buscar **todas** las posibles refutaciones que existan.

Comentarios sobre la base lógica de Prolog: Refutación

Demostrar un teorema, a partir de unos axiomas, *por refutación,* significa negar el teorema y encontrar una contradicción con los axiomas. Cuando se hace una ***pregunta*** a un programa lógico se corresponde con probar la pregunta (el teorema): **∃x1 ∃x2... ∃xp (B1 ∧ B2 ∧...∧ Bn)** a partir de los hechos y reglas del programa (axiomas). Encontrar la cláusula vacía ( ) como resolvente significa encontrar una contradicción en el conjunto de fórmulas (cláusulas de Horn) formado por los hechos y reglas del programa más la ***negación de esta pregunta*.**

1.5. Recorrido en el Espacio de Búsqueda

PROLOG es capaz de dar todas las posibles respuestas a una pregunta, es decir, buscará todas las posibles refutaciones SLD. El espacio de búsqueda de soluciones se puede representar mediante un árbol, donde cada rama representa una posible refutación SLD. (En lugar de expresar en cada paso de resolución la pregunta a tratar y la sentencia con la que se resuelve, tal y como vimos en el apartado anterior, indicaremos únicamente el número de la sentencia con la que se resuelve dicha pregunta, tal y como se muestra en la figura siguiente.)



Las respuestas a las dos refutaciones vienen dadas por las ligaduras de las variables en cada rama:

1) g =g1•g12•g121={X→milcah, Y→haran, Z→milcah} que restringido a las variables de la pregunta da como respuesta **Z=milcah**

2) g =g1•g13•g131 = {X→yiscah, Y→haran, Z→yiscah} que restringido a las variables de la pregunta da como respuesta **Z=yiscah**

Por tanto, el sistema contesta:

Z=milcah ;

Z=yiscah

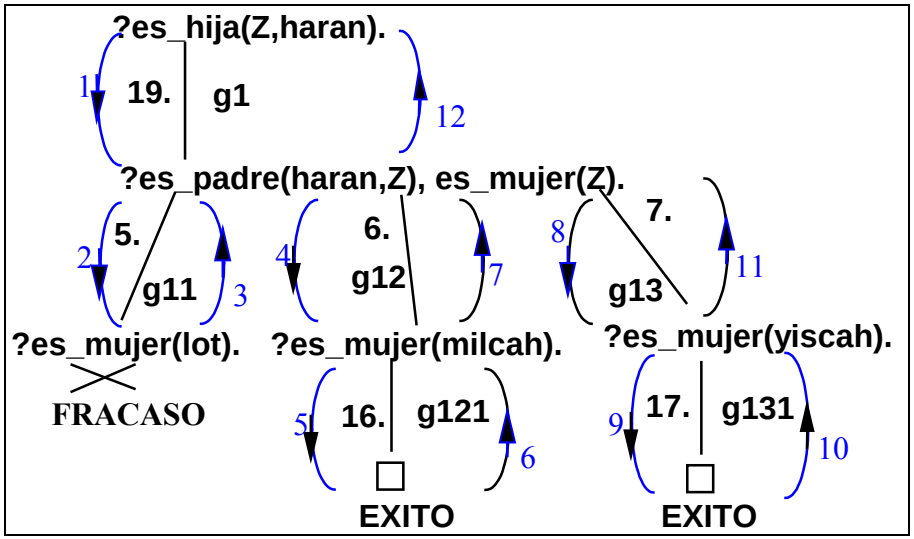
La exploración del árbol de soluciones PROLOG, es en *profundidad*, es decir, explora una rama del árbol hasta llegar al final de ella (a un éxito o fracaso) y sólo después recorrerá las siguientes ramas de la misma forma. Si dibujamos el árbol de búsqueda de manera que las distintas posibilidades de resolución de cada objetivo se dibujen ordenadamente de izquierda a derecha (tal y como se muestra en el árbol del ejemplo), entonces el orden de exploración de las ramas es de izquierda a derecha.

El árbol de búsqueda de soluciones para una pregunta y un programa dados, representa todos los caminos que el sistema debe recorrer para encontrar las respuestas a la pregunta. Pero, **¿Qué ocurre cuando se llega al final de la rama?**

- Si ha sido un éxito (se obtiene) corresponde a una respuesta del sistema.
- Si ha sido un fracaso se debe seguir buscando por otra rama hasta encontrar una refutación. El proceso a seguir es hacer una *vuelta atrás* por los nodos recorridos hasta llegar a alguno del que salga otra rama. Entonces se recorre esta rama en profundidad y así sucesivamente.

Por tanto, el recorrido en el árbol es **en PROFUNDIDAD y con VUELTA ATRAS a la última elección hecha**. Este proceso de vuelta atrás para recorrer más ramas del árbol, no sólo se hace para encontrar la primera refutación (que corresponde a la primera respuesta del sistema), sino también si se quieren encontrar otras posibles refutaciones (otras posibles respuestas).

Ejemplo:



Veamos el proceso para la pregunta **?es-hija(Z, haran).**

- Esta pregunta se unifica con la cabeza de la regla 19 (paso 1), se hace una marca para esta pregunta en el programa y se intentan satisfacer los fines de la pregunta **?es-padre(haran, Z), es-mujer(Z)**.

- Se comienza siempre con el fin más a la izquierda, en este caso con **?es-padre(haran, Z)**. Este fin se satisface al resolverse con el hecho 5. (paso 2), donde la variable Z se ha instanciado a **lot** por g11, se hace una marca para este fin en dicho punto del programa y se pasa a satisfacer la nueva pregunta obtenida que es **?es-mujer(lot).**

- Como el nuevo fin **?es-mujer(lot)** fracasa se hace una vuelta atrás (paso 3), intentando resatisfacer el fin **?es-padre(haran, Z)** (la ligadura de la variable Z con **lot** se ha deshecho), buscando desde el punto del programa donde se encontraba la marca de este fin en adelante (es decir, desde el hecho 6 en adelante).

- Ahora se satisface **?es-padre(haran,Z)** al resolverse con el hecho 6 (paso 4) por lo que se hace una marca para este fin en dicho punto del programa y se pasa a satisfacer el siguiente fin, que con el nuevo unificador g12 es **?es-mujer(milcah)**.

- Este fin se satisface al resolverse con el hecho 16 (paso 5) y se hace la marca correspondiente. Como los dos subfines se han satisfecho, el objetivo principal **?es-hija(Z,haran)** también se ha resuelto. Ya se ha encontrado una refutación y por tanto una respuesta. El sistema contesta con: **Z=milcah**.

- Si se quieren más respuestas (el usuario escribe **;**) el proceso es análogo a si el último fin **?es-mujer(milcah)** hubiera fracasado, es decir, se fuerza de nuevo a hacer vuelta atrás (paso 6 y paso 7), intentando resatisfacer el fin ?es-padre(haran,Z) (la ligadura de la variable Z con **milcah** también se deshace).

- Ahora se busca desde el punto del programa donde se encontraba la marca de este fin, es decir, desde el hecho 7 en adelante. El proceso es análogo hasta encontrar la siguiente refutación (pasos 8 y 9) que da lugar a la segunda respuesta **Z=yiscah**. Si de nuevo se piden más respuestas, el sistema hace vuelta atrás hasta el primer objetivo (pasos 10, 11 y 12) y termina (no hay más resoluciones posibles).

EN GENERAL, dado un objetivo a resolver (o pregunta) **?A_1, A_2,..., A_n** se busca en el programa PROLOG (ordenadamente, desde su primera sentencia) hasta encontrar **B**, un hecho o cabeza de una regla, que coincida con el primer fin de la pregunta: **A_1** (es decir, que pueda unificarse con él mediante algún umg **g**). Pueden darse tres casos:

• Si no se encuentra tal **B** el fin **A_1** FRACASA, y por tanto la pregunta u objetivo fracasa.

• Si se ha encontrado un hecho **B**, se dice que el fin $\mathbf{A_1}$ se ha "satisfecho" y se pasa a satisfacer el resolvente obtenido **?(A_2,..., A_n) g** (que es el resolvente de la pregunta inicial con **B**).

• Si se ha encontrado una regla **B :- B_1, B_2,..., B_m** cuya cabeza **B** se ha unificado con $\mathbf{A_1}$ (mediante **g**) entonces se pasa a satisfacer el resolvente obtenido **?(B_1, B_2,..., B_m, A_2,..., A_n) g**. En el proceso de resolución se intentarán satisfacer primero los subfines obtenidos de la resolución con la regla anterior (**B_1, B_2,..., B_m**) y tras ellos se seguirá con el resto de fines de la pregunta principal (**A_2,..., A_n**). Es decir, una vez satisfechos **B_1, B_2,..., B_m** el fin $\mathbf{A_1}$ se habrá satisfecho y quedarán el resto de fines del objetivo principal.

En cualquiera de los dos últimos casos, cuando $\mathbf{A_1}$ se satisface se marca la regla o hecho que lo ha satisfecho y se pasa a satisfacer el siguiente fin $\mathbf{A_2}$. Con $\mathbf{A_2}$ se realiza el mismo proceso, pero si $\mathbf{A_2}$ FRACASA entonces se intenta RESATISFACER el objetivo $\mathbf{A_1}$. El sistema PROLOG buscará de nuevo en el programa otra forma de satisfacer $\mathbf{A_1}$. Esto significa "deshacer" trabajo anterior. La búsqueda para satisfacer de nuevo $\mathbf{A_1}$ se hará, sin embargo, desde la marca anteriormente hecha para ese fin. Este proceso de reevaluación se denomina **backtracking** o **vuelta atrás**.

1.6. Aspectos Declarativos y Procedurales

Una regla **A:- B_1, B_2, ..., B_n.** tiene una lectura declarativa (o lógica) de la forma: "**A es cierto si B_1 y B_2 y ...y B_n son ciertos**", pero también tiene una lectura "procedural" de la siguiente forma: "**para resolver (ejecutar) A, se debe resolver (ejecutar) B_1 y B_2 y ...B_n**". Desde el punto de vista procedural, una sentencia (o conjunto de sentencias con el mismo predicado de cabeza) puede verse como la definición de un procedimiento.
Por ejemplo, **A :- B_1, B_2, ...B_n** siendo **A** de la forma **p(t1, …, tk)** es un procedimiento de nombre **p** y cuerpo **B_1, B_2, ...B_n**. Una pregunta del tipo **?p(s1, …, sk).** puede verse como la llamada a un procedimiento. Esta llamada da lugar al resolvente **?B'$_1$, B'$_2$, ...B'$_n$** (visto ahora como el cuerpo del procedimiento) que consta a su vez de una secuencia de llamadas a los correspondientes procedimientos **B'i** , obtenidos de **Bi** como resultado de aplicar el umg **g** de **p(t1, …, tk)** y **p(s1, …, sk)** (**g** visto como "paso de parámetros").

Para ilustrar esta vista procedural, supongamos el programa siguiente:

1. **es-padre(terach, abraham).**
2. **es-padre(abraham, isaac).**
3. **es-abuelo(terach, lot).**
4. **es-abuelo(X,Z) :- es-padre(X,Y), es-padre(Y,Z).**

Las reglas **3** y **4** forman un procedimiento de nombre "**es-abuelo**" con dos parámetros formales. La regla **3** indicaría un caso trivial (no hay cuerpo) y la **4** el caso general del procedimiento. La pregunta **?es-abuelo(terach, isaac)** es una llamada al procedimiento "es-abuelo" con los argumentos "terach" e "isaac". El proceso que se realiza, visto proceduralmente, es el siguiente:

• En primer lugar la unificación se encarga de hacer una *selección por casos*: elige el caso general pues "es-abuelo(terach, isaac)" no se unifica con "es-abuelo(terach, lot)", porque no hay resolvente posible entre la pregunta y la sentencia **3**.

• En segundo lugar, la propia unificación ha dado lugar al *paso de parámetros*. Esto es, hace que se unifiquen (o liguen) los parámetros formales **X** y **Z** de la regla **4** con los argumentos, **terach** e **isaac** (mediante el umg).

• Entonces la regla de resolución actúa como *intérprete del lenguaje,* dando lugar al cuerpo del procedimiento con el paso de parámetros realizado (al resolvente)**: ?es-padre(terach, Y), es-padre(Y, isaac).**

• Al hacer ahora la llamada **?es-padre(terach,Y), es-padre(Y, isaac)** el proceso de resolución hace primero que se unifique **Y** con **abraham** (al resolverse con **1**) para dar lugar a la llamada: **?es-padre(abraham, isaac).** En este paso la unificación realiza una construcción de datos (mediante Y = abraham) que es usada en el siguiente resolvente.

• Finalmente, **?es-padre(abraham, isaac)** se entenderá como una llamada correcta a un procedimiento sin cuerpo (al resolverse con **2**).

En el ejemplo se ha podido observar que en la llamada **?es-abuelo(terach, isaac)** al procedimiento "es_abuelo" se han usado los parámetros formales como "parámetros de entrada", mientras que más adelante, en la llamada **?es-padre(terach, Y)** al procedimiento "es_padre" el segundo parámetro formal se ha usado como "parámetro de salida", dando lugar a una construcción de datos.

¡IMPORTANTE!: En la definición de un procedimiento **no** hay parámetros predefinidos de "entrada" y/o "salida". El modo de uso de cada parámetro depende de la llamada o pregunta que se haga en cada momento al programa.

Por ejemplo, para el procedimiento es_abuelo anterior podemos tener estos cuatro **modos de uso** diferentes:

```
?es_abuelo(cte1, cte2).      % entrada, entrada
?es_abuelo(Var1, cte2).      % salida, entrada
?es_abuelo(cte1, Var2).      % entrada, salida
?es_abuelo(Var1, Var2)       % salida, salida
```