Programación Lógica (2ª parte)

3.1. Elementos extra-lógicos de Prolog

En Prolog están predefinidos los operadores aritméticos elementales con la precedencia habitual entre ellos:

$X + Y$	suma
$X - Y$	resta
$-X$	opuesto
$X * Y$	producto
X/Y	división
$X//Y$	cociente de la división entera
X mod Y	resto de la división entera
$X\hat{}Y$	potencia, exponenciación

También incluye funciones elementales, como por ejemplo:

abs(X)	valor absoluto
max(X,Y)	máximo
min(X,Y)	mínimo
sqrt(X)	raíz cuadrada
floor(X)	parte entera

la función exponencial e^x denotada por `exp(X)`, la función logaritmo neperiano denotada por `log(X)`, y las funciones trigonométricas,

Prolog no es una calculadora. Por ejemplo, si escribimos

```
?- 1+1.
```

se obtiene el error

```
ERROR: toplevel: Undefined procedure: (+)/2 (DWIM could not correct goal)
```

La filosofía de Prolog es la consecución de objetivos, es decir, tratar de satisfacer ó "hacer verdad" determinadas fórmulas.

El predicado aritmético is se utiliza para evaluar expresiones aritméticas ó asociar un valor a una determinada variable, lo cual en la jerga correspondiente se denomina instanciar dicha variable. Introduzca lo siguiente.

```
?- X is 1+1.
```

Resulta

```
X = 2.
```

En la instrucción anterior se ha evaluado la expresión `1+1`, y el valor resultante 2 se ha asignado al símbolo de variable `X`.

También podemos escribirlo como

```
?- is(X,1+1).
```

con resultado idéntico.

En la instrucción

```
?- Y is 5, X is Y+1.
```

se hacen dos cosas: Primero se asigna el valor 5 a la variable `Y`, y a continuación el valor de `Y+1`, es decir, 6, se asigna a `X`. Por tanto ambas instrucciones se satisfacen cuando `Y=5, X=6`.

En

```
?- N is 4, N is 4+1.
```

se obtiene como salida `false`, pues ambas no pueden satisfacerse simultáneamente.

Sin embargo, en

```
?- N is 0, N is N+N.
```

se obtiene `N=0` como respuesta, ya que para ese valor ambas se satisfacen.

Si escribe

```
?- X is Y.
```

resulta

```
ERROR: is/2: Arguments are not sufficiently instantiated
```

pues no hay ningún valor para asignarlo. El 2 que aparece después de `is` indica la aridad de dicho predicado.

Introduzca los siguientes ejemplos:

```
?- X is 1/2.

?- X is 1//2.

?- X is abs(-7).

?- X is min(9,7).

?- X is sqrt(100).

?- is(X,floor(3.01)).

?- is(X,floor(2.9)).

?- is(X,floor(-3.01)).

?- is(X,floor(-2.9)).
```

Los operadores relacionales aritméticos se escriben de manera infija, relacionan dos cantidades numéricas, y devuelven `true` ó `false`. Éstos son::

X =:= Y	igual que
X < Y	menor que
X =< Y	menor o igual que
X =\= Y	distintos
X > Y	mayor que
X >= Y	mayor o igual que

Introduzca los siguientes comandos:

```
?- 2 =:= 1+1.

?- 1+1 =:= 3.

?- 2^2 =:= 4.

?- 0 =< 1.

?- 3 =< 2.

?- 3 =\= 4.

?- 3 =\= 3.

?- X=:=1.
```

En este último ejemplo se obtiene

```
ERROR: =:=/2: Arguments are not sufficiently instantiated
```

ya que la variable `X` no tiene ningún valor asignado, es decir, no ha sido instanciada.

```
?- X is 3-2, X =:= 1.
```

resulta

```
X = 1.
```

Ahora la variable `X` tiene un valor asignado antes de compararla con la constante 1.

```
?- 2 =< 2.
```

devuelve `true`, pero la instrucción

```
?- 2 <= 2.
```

devuelve

```
ERROR: Syntax error: Operator expected
ERROR: 2
ERROR: ** here **
ERROR: <= 2 .
```

Utilizando los elementos anteriores, podemos definir nuevos predicados, por ejemplo:

- minimo(X,Y,Z), que se satisface si y sólo si Z es el mínimo de X e Y.
- suma(X,Y,Z), que se satisface si y sólo si Z es igual a X+Y.
- es_par(X), que se satisface si y sólo si X es par.
- factorial(N,F), que se satisface si y sólo si F es el factorial de N.

Escriba en el editor las siguientes reglas y hechos, guárdelas en un archivo y cárguelas en Prolog.

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

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

es_par(X) :- X mod 2 =:= 0.
```

```
factorial(0,1).
factorial(N,F) :- N > 0, M is N-1, factorial(M,T), F is N*T.
```

Si escribimos ahora en el intérprete

```
?- minimo(1,2,Z).
```

se obtiene la respuesta `Z = 1` y queda en espera. Tanto si pulsamos punto y coma como ENTER devuelve `false` y termina. Vea el siguiente ejemplo:

```
?- minimo(2,1,Z).
```

Ahora imprime directamente `Z = 1` y termina. ¿Por qué ocurre ésto, si el cálculo se supone es el mismo?

Las dos reglas con las que se calcula la función `minimo` son:

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

La primera regla (R1) se aplica cuando el primer argumento `X` es menor o igual que el segundo `Y`, devolviendo como salida en el tercer argumento el valor que tiene el primero `X`. La segunda regla (R2) se aplica cuando el segundo argumento `Y` es menor que el primero `X`, devolviendo en el tercer argumento el valor de `Y`.

Cuando preguntamos `minimo(1,2,Z)`, Prolog encuentra la respuesta `Z=1` usando (R1). Como todavía queda la regla (R2) sin usar, por eso Prolog queda a la espera, por si queremos que intente obtener más soluciones usando (R2) en vez de (R1). Nosotros sabemos a priori que no resultarán más soluciones pues ambas reglas son excluyentes, pero Prolog no lo sabe. Existen mecanismos de programación para indicarle a Prolog estas circunstancias.

Cuando preguntamos `minimo(2,1,Z)`, Prolog aplica (R1) sin éxito y entonces retrocede y aplica (R2), la cual sí tiene éxito. Como ya no hay más reglas disponibles, acaba directamente sin quedar a la espera.

También tenemos el predicado `suma(X,Y,Z)` que se satisface cuando a su vez se ha satisfecho el subobjetivo `Z is X+Y`, es decir, cuando `Z` contiene la suma de `X` e `Y`.

```
?- suma(2,5,Z).
?- suma(2,-2,Z).
?- suma(3,5,8).
```

Observe el predicado `es_par(X)` en el que se hace uso de la función aritmética mod que calcula el resto de la división entera.

```
?- es_par(2).
?- es_par(-8).
?- es_par(0).
?- es_par(7).
?- es_par(2*100+1).
```

Ejercicio. Escriba un predicado `es_impar(X)` que se satisfaga cuando `X` sea impar. No hay que comprobar que `X` es un número entero.

Ejercicio. Escriba un predicado `div(X,Y)` que se satisfaga cuando el número entero `X` divide al número entero `Y`.

También hemos definido un predicado `factorial(N,F)` que se satisface cuando `F` es el factorial de `N`. Pruebe los siguientes ejemplos.

```
?- factorial(0,X).
```

```
?- factorial(3,X).
```

```
?- factorial(10,X).
```

Observe que el predicado `factorial` ha sido definido recursivamente. Primero decimos cuál es el factorial de `N=0`, y en otro caso, es decir, si `N>0`, se trata de satisfacer los objetivos `N>0, M is N-1, factorial(M,T), F is N*T`, entre los cuales se invoca de nuevo a `factorial`.

Cuando preguntamos, por ejemplo `factorial(2,X)`, se aplica la segunda regla y se llama a `factorial(1,X)`, aplicándose de nuevo la segunda regla, lo cual llama a `factorial(0,X)`. Ante este predicado, Prolog aplica la primera regla (realmente es un hecho) y va devolviendo hacia atrás los valores que se van calculando con `F is N*T` hasta que finalmente calcula `X=2` y queda a la espera. Ésto es así porque Prolog sabe que al usar el hecho `factorial(0,1)` existía una segunda regla que aún no ha intentado. Realmente con ésta no se obtiene ningún resultado, pues en este momento el primer argumento vale 0 y la segunda regla requiere que su primer argumento sea mayor que 0, pero Prolog desconoce ésto.

No obstante podemos corregir este comportamiento usando lo que se llama el corte en Prolog que poda el árbol de búsqueda. Reemplace en el programa el hecho

```
factorial(0,1).
```

por la regla

```
factorial(0,1) :- !.
```

y ejecute de nuevo

```
factorial(3,X).
```

Como puede ver, Prolog ya no queda en espera.

Ejercicio. Dada la recurrencia $f(0) = 1,\ f(n) = 3f(n-1) + 2$, escriba un predicado `rec(N1,N2)` en Prolog que se satisfaga cuando `N1` y `N2` sean números naturales tales que $f($`N1`$) =$`N2`.

Ejercicio. Repita el ejercicio anterior para la recurrencia

$$f(0) = 1,\ f(1) = 1,\ f(n) = f(n-1) + n \cdot f(n-2).$$

También son bastantes útiles los siguientes predicados predefinidos:

- `succ(X,Y)` se satisface si y sólo si `X` e `Y` son números naturales y `X+1=Y`, es decir, cuando `Y` es el siguiente ó sucesor de `X`.
- `between(X,Y,Z)` se satisface si y sólo si `Z` es un número tal que $\texttt{X} \leq \texttt{Z} \leq \texttt{Y}$.

Introduzca los siguientes comandos:

```
?- succ(99,A).
```

```
?- succ(A,1).
?- succ(A,1000).
?- between(4,7,Z).
?- between(9,7,Z).
```

Nos proponemos ahora resolver en Prolog el problema siguiente: Dado un número natural N, descomponer N de todas las formas posibles como suma de dos números naturales pares.

Para resolverlo planteamos lo siguiente:

Si existen dos números naturales A y B tales que
$A \ mod \ 2 = 0 \wedge B \ mod \ 2 = 0 \wedge N = A + B$,
entonces (A, B) es una solución correcta.

El programa resultante es una transcripción de estas especificaciones y lo damos a continuación.

```
descomp(N,A,B) :-
    between(0,N,A), A mod 2 =:= 0,
    between(0,N,B), B mod 2 =:= 0,
    N =:= A+B.
```

Escriba en el editor dicho programa, guárdelo en un archivo y cárguelo en Prolog. Introduzca a continuación los siguientes ejemplos:

```
?- descomp(6,A,B).
?- descomp(7,A,B).
?- descomp(0,A,B).
```

Ejercicio. Dado un número natural `N`, escriba un predicado `test12(N)` que se satisface cuando `N` se descompone como suma de un número natural impar y otro par.

Ejercicio. Dado un número natural `N`, escriba un predicado `descomp12(N,X,Y)` que permita obtener las descomposiciones de `N` como suma de un número natural impar `X` y un número natural par `Y`.

Ejercicio. Dados números naturales `A,B,C`, queremos encontrar todos los pares ordenados de números naturales `(X,Y)` tales que `AX+BY=C`. Escriba un predicado `soluc(A,B,C,X,Y)` que resuelva este problema.

Ejercicio. Un número compuesto es un número natural $n \geq 2$ cuyos únicos dividores positivos son 1 y n. Escriba un predicado `comp(N)` que se satisfaga cuando `N` sea un número compuesto. A continuación escriba un predicado `primo(N)` que se satisfaga cuando `N` sea un número primo.

Ejercicio. Escriba un predicado `sumapot(N,K,S)` que se satisfaga cuando `S` sea igual a la sumatoria $1^K + 2^K + \cdots + N^K$. Se supone que `N` y `K` son números naturales, no teniendo que comprobar este hecho.

3.2. Comparación de términos y unificación en Prolog

Para la comparación de términos, Prolog predefine los siguientes predicados:

- Term1 == Term2. Vale true si y sólo si el término Term1 es idéntico al término Term2.

- Term1 \== Term2. Equivale a \+ Term1 == Term2.

Introduzca el comando

```
?- 2 + 1 == 1 + 2.
```

Devuelve `false` ya que como expresiones formales son diferentes. El primer argumento de la expresión 1+2 es 1 y el segundo es 2, que no coinciden con los respectivos argumentos en la expresión 2+1.

```
?- 1 + 2 == 1 + 2.
```

Ahora las expresiones son idénticas, de ahí la respuesta obtenida. Recordemos de nuevo el predicado para comparar los valores de expresiones numéricas:

```
?- 1 + 2 =:= 2 + 1.
```

Devuelve `true` ya que los valores numéricos son el mismo.

```
?- f(X,a) == f(x,a).
```

Devuelve `false` debido a que la primera `X` es un símbolo de variable, mientras la segunda es un símbolo de constante.

Por consiguiente el comando siguiente devolverá `true`.

```
?- f(X,a) \== f(x,a).
```

En Lógica de predicados, unificar los términos $t_1, \ldots, t_n$, intuitivamente significa reemplazar en cada término t_i cada una de sus variables constitutivas x_j por algún término s_j, de modo que todos los términos resultantes tras estos reemplazamientos, sean el mismo.

Por ejemplo, los términos

$$t_1 = f(y, f(a, x)),\ t_2 = f(g(b), z)$$

se unifican si reemplazamos la variable y por el término $g(b)$ y la variable z por el término $f(a, x)$. La variable x se queda como está. La sustitución que hemos aplicado se representa por

$$\sigma = (x|x,\ y|g(b),\ z|f(a, x)),$$

resultando

$$\sigma(t_1) = \sigma(t_2) = f(g(b), f(a, x)).$$

Recalcamos que unificar términos en un lenguaje de predicados es un proceso sintáctico que consiste en reemplazar variables por términos.

Los términos $f(x),\ f(f(x))$ no son unificables, pues para serlo, los términos $x,\ f(x)$ a su vez tendrían que serlo, cosa que no sucede. Cualquier remplazamiento del símbolo de variable x por un término no produce la igualdad con el término resultante de aplicar dicho reemplazamiento en $f(x)$.

En teoría veremos con más detalle cómo se resuelve el problema de unificación de términos.

En Prolog, si escribimos `t1 = t2`, le estamos preguntando si los términos `t1` y `t2` son unificables.

Si introducimos

```
?- f(X)=f(a).
```

obtenemos

```
X = a.
```

Es decir, los términos de Prolog `f(X)` y `f(a)` se igualan sustituyendo la variable `X` por `a`.

```
?- f(g(a,X))=f(g(Y,b)).
```

Como era de esperar, resulta que ambos términos se igualan haciendo

```
X = b,
Y = a.
```

Pruebe con los siguientes ejemplos.

```
?- f(g(Y),h(c,d)) = f(X,h(W,d)).
```

La respuesta es

```
X = g(Y),
W = c.
```

```
?- f(g(a,X))=f(g(Y,Y)).
```

Dependiendo de la versión de Prolog, la respuesta puede variar. En la versión que utilizamos, la respuesta es:

```
X = Y, Y = a.
```

Aunque la respuesta que da Prolog no está muy bien expresada. Nosotros la expresaremos mejor como `X=a, Y=a`.

```
?- a=b.
```

Da `false` ya que dos símbolos de constante distintos no son unificables.

```
?- x=f(x).
```

Da `false` pues ambos términos son distintos y no contienen variables para ser sustituídas.

```
?- X=f(X).
```

En nuestra versión de Prolog, la respuesta es `X = f(X)`.

Aunque parace que se unifican para Prolog, realmente estos términos no son unificables en Lógica de predicados pues estamos ante una situación de recursividad. En otras versiones de Prolog, la respuesta es `X = f(**)`.

Un comando más específico de unificación es `unify_with_occurs_check(t1, t2)` que permite realizar unificación de términos, aunque de forma más cuidadosa, pues sólo llevará a cabo la unificación de una variable con un término si éste no contiene a dicha variable. Ésto evita situaciones de recursividad en ecuaciones de términos, como ocurría en el ejemplo anterior.

```
?- unify_with_occurs_check(X, f(X)).
```

Ahora sí se detecta la recursividad y da `false`, es decir, los dos términos proporcionados no son unificables.

Observe también los ejemplos siguientes.

```
?- unify_with_occurs_check(X,X).
```

Devuelve `true`. En otras versiones de Prolog se puede obtener algo como `X = _G368`, donde `_G368` es un nombre de variable usado internamente por Prolog.

```
?- X=X.
```

La respuesta es similar al ejemplo anterior. En nuestra versión devuelve `true`.

```
?- f(g(a,X))=f(g(b,b)).
```

Devuelve `false`.

En este ejemplo, para ser ambos términos unificables, `a` tendría que ser igual a `b` y `X` igual a `b`. Como lo primero no pasa, por eso devuelve `false`.

```
?- p(f(a),g(X))=p(Y,Y).
```

Devuelve `false` pues este ejemplo nos lleva unificar `f(a)` con `g(X)`, cosa que no es posible.

En el ejemplo siguiente la unificación sí es posible.

```
?- p(a,X,h(g(Z)))=p(Z,h(Y),h(Y)).
```

Por tanto en la notación de la Lógica de predicados, un unificador para los dos términos dados es

$$\sigma = (x|h(g(a)),\ z|a,\ y|g(a)).$$

```
?- r(f(X),g(a,Y),h(Z))=r(f(Y),g(a,Z),h(c)).
```

La respuesta es `X = Y, Y = Z, Z = c` Como hemos observado antes en otro ejemplo, habría que decir que la unificación se lleva a cabo haciendo `X=c, Y=c, Z=c`. Esta última respuesta sí se obtiene en otras versiones de Prolog.

Para unificar tres o más términos, digamos t_1, t_2, t_3, t_4, escribimos

$$t_1 = t_2,\ t_2 = t_3,\ t_3 = t_4.$$

También sería equivalente plantearlo como

$$t_1 = t_2,\ t_1 = t_3,\ t_1 = t_4.$$

```
?- r(g(X),f(Z))=r(g(f(Y)),f(Y)), r(g(X),f(Z))=r(g(f(f(a))),f(g(a))).
```

cuya respuesta es `false`.

```
?- s(f(X),f(h(Z)))=s(Y,Y), s(f(X),f(h(Z)))=s(f(h(T)),f(h(g(a,b)))).
```

Ahora se obtiene

```
X = h(g(a, b)),
Z = g(a, b),
Y = f(h(g(a, b))),
T = g(a, b).
```

Por tanto, recapitulando podemos decir que dos términos son unificables si:

1. son idénticos, o bien
2. pueden reemplazarse algunas de sus variables por términos, de manera que los términos resultantes sean idénticos.

De manera un poco más precisa en Prolog:

1. una variable unifica con cualquier término,
2. una constante sólo unifica consigo misma,

3. un estructura unifica con otra estructura si:

   a) tienen el mismo functor (nombre y aridad),

   b) sus respectivos argumentos son unificables,

   c) no se producen asignaciones recursivas (como por ejemplo X=f(a,g(X,Y))).

La unificación de términos es el mecanismo empleado por el motor de inferencia de Prolog para el paso de parámetros a procedimientos. Al invocar un procedimiento, hay que unificar los parámetros actuales y los formales. La unificación es la única forma en que se puede instanciar una variable (es decir, ésta puede recibir un valor) en Prolog. El sistema siempre intenta encontrar lo que se denomina un unificador de máxima generalidad, concepto que estudiaremos en teoría.

Si no ha tenido bastante con los ejemplos anteriores, introduzca (algunos de) los siguientes comandos:

```
?- amigo(X,juan)=amigo(pepe,Y).
?- letra(C)=palabra(C).
?- f(X,Y)=f(A).
?- f(g(a,X))=f(g(a,b)).
?- f(g(a,X))=f(g(X,b)).
?- f(g(a,X))=f(g(X,a)).
?- r(f(X),g(a,Y),h(Z))=r(f(Z),g(a,Z),h(c)).
?- p(g(a),X,h(g(g(Y))))=p(Z,h(Y),h(Y)).
?- p(f(T),g(X))=p(Y,T), q(X,a)=q(b,a).
?- r(g(X),f(Z))=r(g(f(Y)),f(Z)), r(g(X),f(a))=r(g(f(f(a))),f(g(a))).
?- r(g(X),f(Z))=r(g(f(Y)),f(Y)), r(g(X),Z)=r(g(f(a)),a).
?- f(g(a,Y),Z)=f(g(a,h(T)),b),g(T,f(b))=g(X,f(Z)),X=b.
```

3.2.1. El orden de los elementos en los programas Prolog.

Cuando cargamos un programa Prolog y le hacemos una pregunta al sistema, las respuestas que éste va encontrando, así como el orden de las mismas, depende en gran medida del orden en el que aparecen los hechos y reglas en nuestro programa.

Cree un programa Prolog con los siguientes hechos:

```
p(a).
p(b).
p(c).
```

Cárguelo en el sistema y a continuación ejecute el comando siguiente:

```
?- p(X).
```

Prolog busca el primer hecho ó la primera regla en la base de datos cuya parte izquierda es unificable con `p(X)`. En nuestro programa ello ocurre con el primer hecho `p(a)`, siendo la unificación la dada por `X=a`. De esta forma se ha completado el único objetivo que teníamos, por lo cual Prolog muestra en pantalla la respuesta `X=a` (¡qué casualidad!) y queda en espera, ya que sabe que hay otras dos alternativas (hechos en este caso) que todavía no ha intentado. Si pulsamos punto y coma, le estamos indicando que pruebe con la segunda opción. De nuevo plantea `p(X)=p(b)` y ahora obtiene el unificador `X=b`, muestra en pantalla la respuesta `X=b` y queda en espera. Si de nuevo pulsamos punto y coma, le estamos diciendo que intente la tercera alternativa, imprimiendo en pantalla `X=c` y termina, ya que sabe que no hay más posibilidades.

Seguidamente cambie el orden de los elementos del programa de la forma siguiente:

```
p(c).
p(b).
p(a).
```

Cárguelo de nuevo en el sistema y ejecute el mandato siguiente:

```
?- p(X).
```

Como puede ver, obtiene las mismas respuestas pero en otro orden.

Cuando planteamos una consulta a Prolog, éste lleva a cabo un proceso de búsqueda primero en profundidad con retroceso. Ya comentamos algo al respecto cuando vimos el ejemplo `factorial`.

Cree el siguiente programa Prolog:

```
p(X) :- q(X), r(X).
p(X) :- t(X).
q(a).
q(b).
r(a).
t(c).
```

Si le preguntamos

```
?- p(Y).
```

obtenemos la respuesta

```
Y = a
```

y queda a la espera. Su pulsamos punto y coma, encuentra la respuesta siguiente y termina:

```
Y = c.
```

Intentamos explicar un poco lo ocurrido. Encuentra la primera regla `p(X) :- q(X), r(X)` cuya parte izquierda es unificable con el objetivo propuesto `p(Y)`. El unificador obtenido es `X=Y`. Además ve que la segunda regla también es aplicable y la anota, por si posteriormente le pedimos más respuestas. Entonces se plantea la lista de objetivos `q(Y), r(Y)`, tratando de satisfacer el primero de ellos, es decir, `q(Y)` para lo cual de nuevo recorre la base de datos de forma ordenada intentando unificar una parte izquierda con `q(Y)`. Ello sucede cuando considera la tecera línea donde se encuentra el hecho `q(a)`. El unificador que se obtiene es `Y=a`. Por tanto `q(Y)` se satisface cuando `Y=a`. Además anota que en la cuarta línea hay otra alternativa `q(b)` que se unifica con `q(Y)`, por si en un futuro le pedimos más soluciones. Pero ahora tiene que intentar satisfacer el objetivo `r(Y)` cuando `Y=a`, es decir, `r(a)`, lo cual se consigue al recorrer de nuevo la base de datos desde el principio y llegar a la quinta línea. Así la lista de objetivos ha quedado vacía, es decir, se han satisfecho todos ellos. Ésto significa que ha encontrado una respuesta, concretamente `Y=a`. La imprime en pantalla y queda a la espera. Si pulsamos punto y coma, intenta la última alternativa que anotó, ésta es, unificar `q(Y)` con `q(b)`. Entonces la lista de objetivos queda como `r(b)`. Este objetivo no tiene éxito, pues el hecho `r(b)` no forma parte de la base de datos. Entonces retrocede para intentar la siguiente posibilidad que anotó, es decir, satisfacer el objetivo inicial `p(X)` usando la segunda regla en la base de datos. De nuevo resulta el unificador `X=Y` y la lista de objetivos queda como `t(Y)`. Recorre otra vez la base de datos y encuentra como única regla aplicable (realmente es un hecho), la número seis. Unifica

mediante `Y=c`, quedando vacía la lista de objetivos, imprime en pantalla la solución encontrada `Y=c` y termina, pues sabe que ya no hay más alternativas.

También puede influir en la ejecución del programa el orden en el que se escriben los elementos que aparecen en una regla. Por ejemplo la regla `q(X,Z) :- r(X,Y), q(Y,Z)` es desde un punto de vista lógico equivalente a `q(X,Z) :- q(Y,Z), r(X,Y)`. Sin embargo este pequeño detalle puede afectar al proceso de búsqueda que lleva a cabo Prolog.

Cree un archivo de programa Prolog con los elementos siguientes:

```
q(X,X).
q(X,Z):-r(X,Y),q(Y,Z).
r(b,c).
```

Cárguelo y pregúntele lo siguiente:

```
?- q(X,c).
```

Primero encuentra la respuesta

```
X = c
```

Si pulsamos punto y coma, encuentra otra

```
X = b
```

Si pulsamos de nuevo punto y coma, ya no encuentra más respuestas por lo cual devuelve `false` y acaba.

A continuación hacemos lo mismo con el programa siguiente:

```
q(X,X)
q(X,Z):-q(Y,Z),r(X,Y)
r(b,c)
```

Preguntamos

```
?- q(X,c).
```

y encuentra la respuesta

```
X = c
```

quedando a la espera. Pulsamos punto y coma, y obtiene otra respuesta

```
X = b
```

Volvemos a pulsar punto y coma y ahora resulta

```
ERROR: Out of local stack
?-
```

Lo que ocurre es que el sistema ha entrado en una búsqueda infinita que ha provocado que la memoria asignada para hacer los cálculos, llamada pila, se agote y muestre finalmente un mensaje de error.

La situación se puede agravar más todavía si cambiamos el orden de las reglas, hasta el punto de que el sistema sea incapaz de encontrar respuestas. Pruebe el programa siguiente y haga la misma pregunta de antes.

```
q(X,Z):-q(Y,Z),r(X,Y)
q(X,X)
r(b,c)
```

Obtenemos un mensaje del tipo

```
?- q(X,c).
ERROR: Out of local stack
   Exception: (440,976) q(_G346, c) ?
```

Cuando se queda bloqueado, tenemos que darle al botón [Iniciar]>[Recomenzar], ó bien pulsar al mismo tiempo las teclas Control y F9.

3.2.2. Listas en Prolog.

Una lista puede definirse recursivamente como:

- una lista vacía o sin elementos, denotada por `[ ]`, o bien
- una estructura con dos componentes:
  - **cabeza** o primer argumento, y
  - **cola** o segundo argumento, es decir, el resto de la lista.

En un programa Prolog cuando se escribe `[ X | Y ]` se está denotando una lista cuya cabeza es `X` y cuya cola es `Y`.

No obstante también se utiliza en Prolog la notación más sencilla de las listas que dispone a los elementos de la misma separados por comas, y encerrados entre corchetes. Por ejemplo:

Lista	**Cabeza**	**Cola**
`[a,b,c]`	a	`[b,c]`
`[a]`	a	`[ ]`
`[ ]`	(no hay)	(no hay)
`[[x,y],a,[b,c],[ ]]`	`[x,y]`	`[a,[b,c],[ ]]`

En una lista el orden de los elementos es importante y pueden haber elementos repetidos.

Al introducir

```
?- [X|Y]=[a,b,c,d].
```

Se obtiene

```
X = a,
Y = [b, c, d].
```

Análogamente para los siguientes ejemplos:

?- [X|Y]=[a].

Resulta

```
X = a,
Y = [].
```

Para

```
?- [X|Y]=[a,[1,2,3],c,d].
```

Se obtiene

```
X = a,
```

```
Y = [[1, 2, 3], c, d].
```

También es interesante

```
?- [X|Y]=[].
```

que produce

```
false.
```

Pruebe también

```
?- [X,Y|Z]=[a,b,c,d,e].
```

En Prolog existe un módulo que se carga por defecto para el manejo de listas y que ilustramos brevemente a continuación.

El comando `is_list` se aplica a un argumento y nos dice si éste es o no una lista.

```
?- is_list([x,y]).
?- is_list([[x,y],a,[b,c],[ ]]).
?- is_list([ ]).
?- is_list(x).
?- is_list(f(X,a)).
```

Mediante el uso de la recursividad, nosotros podemos definir un predicado que hace los mismo que el anterior. Cree un programa Prolog con los siguientes hechos y reglas:

```
?- es_lista([]).

?- es_lista([_|Xs]) :- es_lista(Xs).
```

Pruebe el predicado definido con los ejemplos anteriores.

El comando `member(Elem, List)` nos dice si `Elem` es un elemento o no de la lista `List`. Pruebe los siguientes ejemplos.

```
?- member(1,[3,2,1]).
?- member(0,[3,2,1]).
?- member([x,y],[[a],x,y]).
?- member([x,y],[[a],[x,y],1,2]).
?- member([x,y],[[a],[y,x],1,2]).
```

Análogamente, podemos definir un predicado que hace los mismo que el anterior. Cree un programa Prolog con los siguientes hechos y reglas:

```
?- pertenece(X,[X|_]).

?- pertenece(X,[_|Ys]) :-  pertenece(X,Ys).
```

Note que el orden de las reglas es decisivo. La segunda se intenta sólo si falla la primera. Pruebe el predicado definido con los ejemplos anteriores.

El predicado `length(List, Int)` se satisface si el valor `Int` es el número de elementos de la lista `List`.

```
?- length([],0).
?- length([1,2,3],3).
?- length([1,2,3],4).
?- length([1,2,3],X).
?- length([[1,2,3]],X).
```

Ejercicio. Escriba reglas y/o hechos que definan un predicado `longitud(List,Int)` el cual se satisface cuando `List` es una lista cuyo número de elementos es `Int`. No vale usar el predicado `length` ya predefinido en Prolog. A continuación pruebe el predicado definido con los ejemplos anteriores.

Otra operación interesante es la concatenación de dos listas `L1` y `L2`, es decir, la formación de una nueva lista cuyos elementos se obtienen al escribir primero los elementos de `L1` y a continuación los de `L2`.

En Prolog tenemos el predicado `append(L1, L2, L3)` que se satisface si `L3` es la concatenación de `L1` y `L2`.

Pruebe los siguientes ejemplos:

```
?- append([a,b],[c,d,a],Z).
?- append([a,b],[],Z).
?- append([],[c,d,a],Z).
?- append([],[],Z).
?- append(X,Y,[a,b,c]).
```

Ejercicio. Escriba reglas y/o hechos que definan un predicado `concatena(L1,L2,L3)` el cual se satisface cuando `L3` es la lista resultante de concatenar la lista `L1` con `L2`. No vale usar el predicado `append` ya predefinido en Prolog. A continuación pruebe el predicado definido con los ejemplos anteriores.

Ejercicio. Escriba reglas y/o hechos que definan un predicado `inversa(L1,L2)` el cual se satisface cuando `L2` es la lista formada por los elementos de la lista `L1` escritos en orden inverso. Se recomienda utilizar el predicado `concatena` propuesto en el ejercicio anterior. No vale basarse en algún predicado ya predefinido en Prolog. Por ejemplo, `inversa([a,b,c],L)` ha de devolver `L=[c,b,a]`, e `inversa([a,[x,y,z],b],L)` ha de devolver `L=[b,[x,y,z],a]`.

Ejercicio. Escriba reglas y/o hechos que definan un predicado `palindromo(L)` que se satisface cuando la lista `L` es igual a su inversa.

Ejercicio. Escriba reglas y/o hechos que definan un predicado `borra(X,L1,L2)` cuyo efecto es borrar una ocurrencia del elemento `X` en la lista `L1` resultando así la lista `L2`. Por ejemplo, `borra(a,[a,b,a,c,c,a],L)` ha de devolver `L=[b,a,c,c,a]`, y quedar a la espera de modo que si pulsamos punto y coma devuelva `L=[a,b,c,c,a]`, y por último, si pulsamos otra vez punto y coma, devuelva `L=[a,b,a,c,c]`.