Tema 2. DEFINICIÓN RECURSIVA de PROGRAMAS

2.1. Ejemplos de programas

2.2. Comportamiento Recursivo e Iterativo.

2.3. Recursión a izquierdas.

2.4. Generación infinita.

TEMA 2. DEFINICION RECURSIVA de PROGRAMAS

En este tema se muestran cuestiones relacionadas con la definición recursiva de programas, utilizando fundamentalmente en los ejemplos la estructura de listas. Al ser dicha estructura recursiva (el "resto" de una lista es a su vez una lista), la mayoría de los programas sobre listas serán recursivos de forma natural. Mostraremos también "versiones iterativas" de algunos programas recursivos, programas generadores de infinitas respuestas, etc.

2.1. Ejemplos de Programas

El objetivo de esta sección es mostrar mediante ejemplos cómo diseñar programas recursivos.

Ejemplo 1: miembro de una lista

Supongamos que tenemos la **lista** de las personas que asistirán a una reunión y queremos averiguar si una determinada persona irá a ella.
La forma de programarlo en PROLOG será ver si dicha persona es la primera ó **cabeza** de la lista o (en otro caso) si está en el resto de la lista, lo que supondrá mirar de nuevo si es la **cabeza del resto** o está en el resto de dicho resto, etc. Si llegamos al final de la lista, es decir, a la lista vacía, será que la persona dada no está en la lista de asistentes. El programa es el siguiente:

1. **miembro(X, [X|Y]).**
2. **miembro(X, [Y|Z]) :- miembro(X,Z).**

Para especificarlo se ha usado un predicado "**miembro(X, L)**" que significa X es miembro de la lista L. La primera regla del programa indica "X es miembro de la lista que tiene X como cabeza" y la segunda expresa "X es miembro de una lista si X es miembro del resto de dicha lista". Las dos reglas juntas definen el predicado **miembro** y dicen a PROLOG cómo recorrer una lista de principio a fin, para buscar un elemento.
Una posible pregunta al programa es: **? miembro(pedro, [maria, pedro, juan, ana]).**

El programa puede verse como un procedimiento con llamada recursiva. Como todo procedimiento de este tipo deben buscarse las condiciones límite de la recursión (los casos triviales) y el caso recursivo.

En el ejemplo, hay dos **condiciones límite**:

- La primera condición viene dada de forma *explícita* por la primera regla del programa: ¡ya hemos encontrado a X en la lista!
- La segunda condición, que tendrá lugar cuando el segundo argumento sea la lista vacía, viene dada de forma *implícita* (no hay ninguna regla para este caso), por lo que una pregunta del tipo **?miembro(pedro,[ ]).** fracasará.

Para estar seguros de llegar a las condiciones límite (o de parada) debemos observar el caso recursivo, que viene dado por la segunda regla del programa. Al utilizar esta regla vemos que se produce una llamada al propio procedimiento con una lista más corta (el resto de la lista), por lo que es seguro que el procedimiento recursivo terminará, siempre que el segundo parámetro se use en "modo **in**", es decir, como parámetro de entrada.

Observemos cómo sería la secuencia de llamadas (los resolventes del proceso de refutación) para la siguiente pregunta:

? miembro(pedro, [maria, pedro, juan, ana]).

2. | {X1→pedro, Y1→maria, Z1→ [pedro,juan,ana]}

? miembro(pedro, [pedro, juan, ana]).

1. | {X2→pedro, Y2→ [juan,ana]}

□ EXITO (la contestación es **si**)

Como podemos observar, en cada paso de resolución del árbol de llamadas, se ha indicado cuál es la sentencia (hecho o regla) usada poniendo su número, así como cuáles son los valores ligados a las variables usadas. En general, cada vez que se use una sentencia del programa, optaremos por ***renombrar sus variables*** poniéndolas como subíndice el número del paso de resolución seguido.

Para esta pregunta concreta, se ha encontrado una refutación por lo que la contestación es afirmativa.

¿Cuál sería la secuencia de llamadas, en la búsqueda de una respuesta a la siguiente pregunta?:

```
? miembro(vicente, [maria, pedro, juan, ana]).
  | 2.
? miembro(vicente, [pedro, juan, ana]).
  | 2.
? miembro(vicente, [juan, ana]).
  | 2.
? miembro(vicente, [ana]).
  | 2.
? miembro(vicente, [ ]).
      ⊗ FRACASO
```

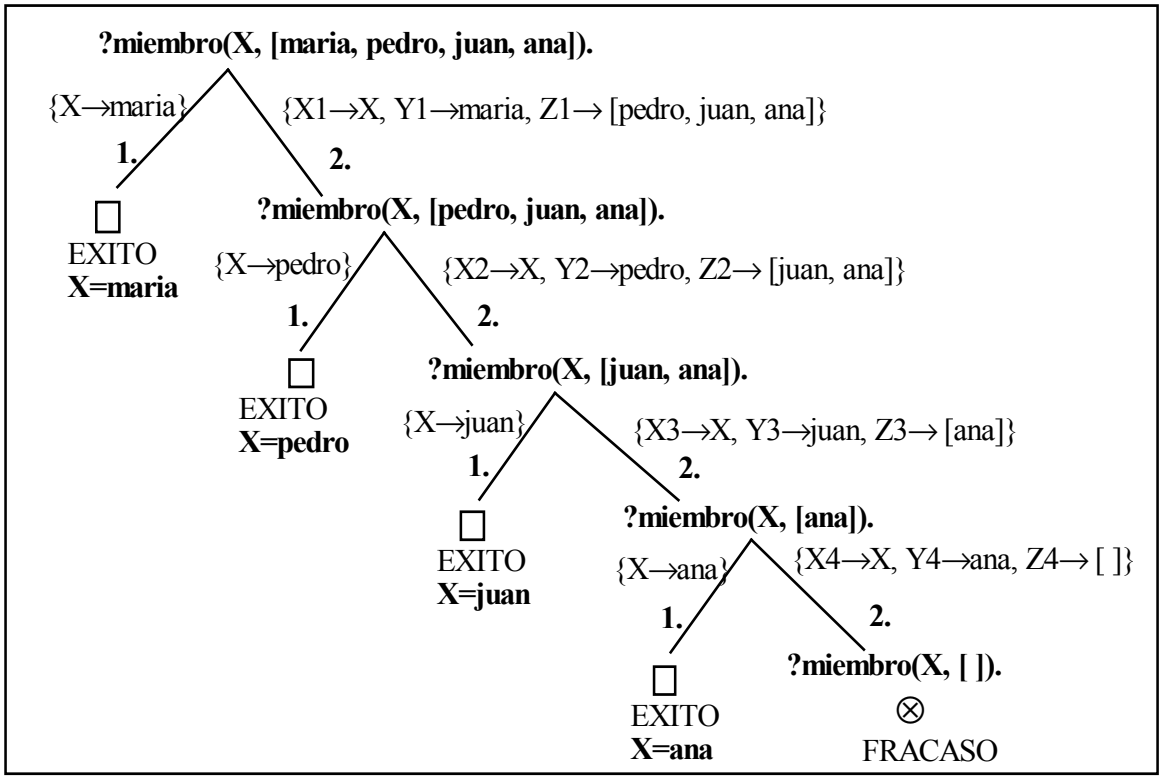
Aquí hemos indicado en cada paso la sentencia del programa con la que se resuelve el objetivo en curso (sin mostrar los unificadores).
Tras llegar al final de la rama y obtener un fracaso, PROLOG debe seguir intentando la búsqueda de soluciones por otras ramas. Sin embargo, la vuelta atrás le lleva a que no existe otra posible rama mediante la cual seguir la búsqueda.
Para esta pregunta, por tanto, no se ha encontrado ninguna refutación por lo que la contestación es negativa: **no**.

Como ya se dijo anteriormente, en la definición de un programa no existen parámetros de entrada y/o salida, el ***modo de uso se establece cuando se realiza la llamada*** a dicho programa. En las dos preguntas anteriores los dos parámetros han sido **de entrada**. Es decir, el uso dado ha sido el que nosotros de alguna manera habíamos supuesto al realizar el diseño del programa *miembro*. Sin embargo, para este programa existen más usos correctos. Podemos, a través de una llamada **miembro(Var, lista)**, encontrar todos los miembros de una lista dada:

```
? miembro(X, [maria,pedro,juan,ana]).
  X=maria ;
  X=pedro ;
  X=juan ;
  X=ana
```

Nota: El sistema da la primera contestación **X=maria** y se queda en espera de que el usuario le indique si quiere o no más soluciones. Cuando el usuario escribe un "**;**" significa que quiere otra solución. En nuestro caso, dará la segunda **X=pedro**. Si la respuesta del usuario es la de pulsar la tecla <RETURN>, significará que no precisa de más soluciones. Si tras la última solución **X=ana** se escribe de nuevo un "**;**" (se piden más soluciones), PROLOG contesta **no** ("no hay más"), en otro caso (si pulsa la tecla <RETURN>) contesta **si**.

El árbol completo generado por PROLOG para obtener todas las respuestas anteriores es:



Nota: Al resolverse la pregunta de partida con la regla **2**, se ha obtenido como **umg** : {X1→X, Y1→maria, Z1→[pedro, juan, ana]}. Pero hubiera sido también válido el siguiente: {X→X1, Y1→maria, Z1→[pedro, juan, ana]}, que ligaría **X→X1** y daría lugar al resolvente:
?miembro(X1, [pedro, juan, ana]).
En ambos casos las respuestas son **las mismas**.

Optaremos por elegir el unificador que da lugar al resolvente que mantiene las variables de la pregunta. Esto es, en el ejemplo: elegimos el primer caso (X1→X) en lugar de sustituir la variable X de la pregunta por la variable X1.

Finalmente, un detalle más respecto al programa *miembro*. La primera sentencia indica que el elemento buscado es la cabeza de la lista que estamos tratando. Si esto es así en realidad el valor del resto de la lista ya no nos interesa tratarlo. De igual modo, en la segunda sentencia sólo nos interesa conocer el valor asociado al resto de la lista y no el de la cabeza. Por tanto, cuando no interese el valor de una determinada variable puede colocarse una ***variable muda***, que se expresará mediante el símbolo "_". El programa *miembro* quedaría de esta forma así:

1. **miembro(X, [X|_]).**
2. **miembro(X, [_|Z]) :- miembro(X, Z).**

Ejemplo 2: concatenación de listas

Este programa es muy usado como parte de otros programas y corresponde a la concatenación de dos listas. Es decir, dadas dos listas L1 y L2, obtener la lista L3 formada por la concatenación (o yuxtaposición) de los elementos de L1 con los de L2.
Por ejemplo, dadas **L1** = [a, b, c] y **L2** = [f, e], la concatenación de **L1** con **L2** es **L3**, donde **L3** = [a, b, c, f, e].
Necesitaremos un predicado **concatenar(L1, L2, L3)** que signifique "L3 es la concatenación de la lista L1 con la lista L2". El programa será de nuevo recursivo, siguiendo la definición recursiva de lista para el primer argumento:

1. **concatenar([], L, L).**
2. **concatenar([X|Y], Z, [X|U]) :- concatenar(Y, Z, U).**

En este programa el caso recursivo viene dado por la segunda regla. La recursividad terminará para preguntas en las que el primer parámetro es de entrada, debido a que éste es cada vez más corto en cada llamada recursiva (el resto de la lista).
La condición de límite se dará cuando el primer argumento es la lista vacía, caso que viene dado por la primera regla del programa.
Un uso en **modo in** del tercer parámetro, también asegura la terminación del programa de forma similar.

PREGUNTA	RESPUESTA
? concatenar([a, b, c], [d, e, f], X).	X= [a, b, c, d, e, f]
? concatenar([a, b, c], X, [a, b, c, d, e]).	X= [d, e]
? concatenar([a, b, c], X, [a, c, d, e]).	no
? concatenar(X, Y, [a, b, c, d, e]).	X=[ ] Y=[a, b, c, d, e] ;
	X=[a] Y=[b, c, d, e] ;
	X=[a, b] Y=[c, d, e] ;
	X=[a, b, c] Y=[d, e] ;
	X=[a, b, c, d] Y=[e] ;
	X=[a, b, c, d, e] Y=[ ] ;
	no

El árbol de llamadas para la primera de las preguntas hechas en el cuadro anterior sería:

?concatenar([a, b, c], [d, e, f], X).

2. **g1**={X1→a, Y1→ [b, c], Z1→ [d, e, f], X→ [a | U1]}

?concatenar([b, c], [d, e, f], U1).

2. **g2** ={X2→b, Y2→ [c], Z2→ [d, e, f], U1→ [b | U2]}

?concatenar([c], [d, e, f], U2).

2. **g3** ={X3→c, Y3→ [ ], Z3→ [d, e, f], U2→ [c | U3]}

?concatenar([ ], [d, e, f], U3).

1. **g4** ={L4→ [d, e, f], U3→ [d, e, f]}

□

EXITO

Una vez finalizada la construcción del árbol con los **umg** en cada paso, es necesario componer la solución o respuesta que daría el sistema. Como en la pregunta inicial aparece una variable, **X**, el sistema debe contestar con el valor asociado a esa variable, si es que finaliza con éxito (como en este caso).
Por tanto, sabemos que:
X = [a | U1], por el primer paso (con umg **g1**),
U1 = [b | U2], por el segundo paso (con **g2**),
U2 = [c | U3], por el tercer paso (con **g3**),
U3 = [d, e, f], por el cuarto paso (con **g4**).

Componiendo estos unificadores (g = g1•g2•g3•g4), tendremos (en lo referente a las variables que afectan a X) que

U2 = [c | U3]= [c | [d, e, f]]= [c, d, e, f]
U1 = [b | U2]= [b | [c, d, e, f]]= [b, c, d, e, f]
X = [a | U1]= [a | [b, c, d, e, f]= [a, b, c, d, e, f]

Por tanto la respuesta de PROLOG es **X = [a, b, c, d, e, f]**.

Si en cada paso del proceso de desarrollo del árbol anterior, hubiéramos ido realizando la composición de los umg **gi** (obtenidos hasta dicho paso) se tendría un árbol como el que se muestra a continuación:

? concatenar([a, b, c], [d, e, f], X).

2. **g1**={X1→a, Y1→ [b, c], Z1→ [d, e, f], X→ [a | U1]}

? concatenar([b, c], [d, e, f], U1).

2. **g2**= {X2→b, Y2→ [c], Z2→ [d, e, f], U1→ [b | U2]}
g12= **g1 • g2** = {X1→a, Y1→ [b, c], Z1→ [d, e, f], X→ [a, b |U2],
X2→b, Y2→ [c], Z2→ [d, e, f], U1→ [b | U2]}

? concatenar([c], [d, e, f], U2).

2. **g3**={X3→c, Y3→ [ ], Z3→ [d, e, f], U2→ [c | U3]}
g123 = **g12•g3** ={X1→a, Y1→ [b, c], Z1→ [d, e, f], X→ [a, b, c |U3],
X2→b, Y2→ [c], Z2→ [d, e, f], U1→ [b, c | U3],
X3→c, Y3→ [ ], Z3→ [d, e, f], U2→ [c | U3]}

? concatenar([ ], [d, e, f], U3).

1. **g4**={L4→ [d, e, f], U3→ [d, e, f]}
g1234 = **g123•g4** ={X1→a, Y1→[b, c], Z1→[d, e, f], **X→ [a,b,c,d,e,f]**}
X2→b, Y2→ [c], Z2→ [d, e, f], U1→ [b, c, d, e, f],
X3→c, Y3→ [ ], Z3→ [d, e, f], U2→ [c, d, e, f],
L4→ [d, e, f], U3→ [d, e, f]}

□
EXITO

X= [a, b, c, d, e, f]

De esta forma observamos directamente cuál es el estado real de las ligaduras de las variables en cada paso. Es decir, vemos cómo se va componiendo la solución del problema y al llegar a la cláusula vacía ya tenemos la respuesta, X=[a, b, c, d, e, f].
Sin embargo, para hacer el proceso de desarrollo menos engorroso, optaremos por no describir en el árbol estas composiciones parciales, realizando la composición de unificadores al final.

Se deja como ejercicio el resolver, de forma similar, las demás preguntas de la tabla anterior:

? concatenar([a, b, c], X, [a, b, c, d, e]),
? concatenar([a, b, c], X, [a, c, d, e]),
? concatenar(X, Y, [a, b, c, d, e]),

desarrollando el árbol de búsqueda de soluciones y obteniendo, mediante la composición de unificadores de cada rama, las respuestas que da el sistema a dichas preguntas.

Ejemplo 3: Piezas de una bicicleta

Supongamos que trabajamos en una fábrica de bicicletas, donde es preciso llevar un inventario de las piezas de una bicicleta. Si queremos construir una bicicleta tenemos que saber qué piezas coger del almacén. Cada pieza de una bicicleta puede tener sub-piezas, por ejemplo, cada rueda puede tener radios, una llanta y un plato. A su vez el plato, por ejemplo, puede constar de un eje y de piñones. Consideraremos ahora una base de datos que nos permita hacer preguntas sobre las piezas que se requieren para construir (una parte de) una bicicleta.
Hay dos clases de componentes para construir nuestra bicicleta. En concreto, las piezas de *ensamblaje* y las *piezas básicas* . Cada ensamblaje consiste en un conjunto de *componentes* básicos y/o ensamblajes, como la rueda que consta de radios, llanta y plato. Las piezas básicas no se componen de otras menores, simplemente se combinan entre ellas para conformar ensamblajes.
Podemos representar las piezas básicas simplemente como **hechos**. Naturalmente la lista de piezas básicas necesarias para una bicicleta dada aquí no es real, pero nos vale como ejemplo. Por otra parte, los ensamblajes pueden representarse también como **hechos,** mediante un predicado de dos argumentos. Por ejemplo, **ensamblaje (bici, unir([rueda, rueda, cuadro]))** indica que una **bici** es un ensamblaje formado por la unión de dos ruedas y un cuadro.

La base de datos de piezas básicas y ensamblajes necesaria para nuestra sencilla bici es la siguiente:

1. piezabasica(llanta).
2. piezabasica(cuadro-trasero).
3. piezabasica(piñones).
4. piezabasica(tuerca).
5. piezabasica(radios).
6. piezabasica(manillar).
7. piezabasica(tornillo).
8. piezabasica(horquilla).

9. ensamblaje(bici, unir([rueda, rueda, cuadro])).
10. ensamblaje(rueda, unir([radios, llanta, plato])).
11. ensamblaje(cuadro, unir([cuadro_trasero, cuadro_delantero])).
12. ensamblaje(cuadro_delantero, unir([horquilla, manillar])).
13. ensamblaje(plato, unir([piñones, eje])).
14. ensamblaje(eje, unir([tornillo, tuerca])).

Ahora ya tenemos la información suficiente para escribir el programa que, dado un componente, devuelve la lista de las piezas básicas que se necesitan para construirlo:
- Si el componente que queremos montar es una pieza básica, entonces la lista formada con dicha pieza es la respuesta.
- Sin embargo, si queremos montar una pieza de ensamblaje tenemos que aplicar el proceso de búsqueda de las piezas básicas para cada componente del ensamblaje.

Definimos un predicado **piezasde(X, Y)**, donde **X** es el nombre de un componente e **Y** es la lista de piezas básicas que se necesitan para construir **X**, mediante las sentencias:

```
15. piezasde(X, [X]) :- piezabasica(X).
16. piezasde(X, P) :- ensamblaje(X, unir(Subpiezas)), listapiezasde(Subpiezas, P).
```

La condición de límite tiene lugar cuando **X** es una pieza básica (sentencia 15). En este caso simplemente se devolverá **X** en una lista.
La siguiente condición se da si **X** es un ensamblaje (sentencia 16). En dicho caso tenemos que encontrar si existe un **ensamblaje** de nombre **X** en la base de datos y aplicar la búsqueda de piezas básicas a la lista de **Subpiezas** que conforman el ensamblaje **X,** utilizando un nuevo predicado llamado **listapiezasde**.

El proceso que seguirá **listapiezasde** (descrito en las sentencias 17 y 18) consiste en: obtener en **Piezascabeza** las piezas básicas de la cabeza de la lista (mediante **piezasde)**, obtener en **Piezasresto** las piezas básicas del resto de la lista (mediante **listapiezasde)** y, finalmente, concatenar ambos resultados en la lista **Total** de piezas básicas buscadas.

```
17. listapiezasde([], []).
18. listapiezasde([P|Resto], Total):-
                    piezasde(P, Piezascabeza),
                    listapiezasde(Resto, Piezasresto),
                    concatenar(Piezascabeza, Piezasresto, Total).
```

La lista construida por **piezasde** no contiene información de la cantidad de piezas necesarias, de forma que pueden aparecer piezas duplicadas en la lista. Se propone como ejercicio hacer una versión mejorada de este programa para solventar estas deficiencias. Por ejemplo, podríamos diseñar **ensamblaje** de tal forma que indique los distintos componentes necesarios y sus cantidades, a través de una estructura como **componente(nombre_componente, cantidad_componente)**. Así podemos expresar la información de los ensamblajes de la siguiente manera: **ensamblaje(bici, unir([componente(rueda, 2), componente(cuadro, 1)]))**.

2.2. Comportamiento Recursivo e Iterativo

Presentamos dos versiones para el problema de invertir los elementos de una lista. Usaremos el predicado **inversa (X,Y)** para indicar que Y es la inversa de la lista X. Es decir, dada la lista **X=[e1, e2 , ..., e_{n-1}, e_n]** el programa obtendrá como inversa **Y =[e_n, e_{n-1}, ..., e_2, e_1]**.

Programa 1 de "inversa":

1. **inversa([ ], [ ]).**
2. **inversa([X|Y], L):- inversa(Y,R), concatenar(R, [X], L)**

Para hallar la inversa de una lista de **n** elementos, tendríamos un árbol de llamadas como el que se muestra en el siguiente recuadro. En él se aprecia que tras producirse **n+1** llamadas recursivas a "inversa" (en **n+1** pasos consecutivos), se tiene un objetivo con **n** llamadas a "concatenar". Estas llamadas se resolverán: la primera en un paso, la segunda en dos pasos, y así sucesivamente (haciendo un total de llamadas de orden $\mathbf{n^2}$), lo que supone un costo en tiempo (y quizás en memoria).
La sucesión de llamadas a "concatenar" obtenida tras el paso **n+1** se debe a que, en la regla 2 del programa, la llamada recursiva a **inversa** se produce antes que la llamada a **concatenar**, quedando pendiente la resolución de todas las llamadas a "concatenar" tras la resolución de todas las llamadas a "inversa".

? inversa([a_1, a_2,..., a_n], X).

(1) | **2.** {X1→a_1, Y1→ [a_2, a_3,..., a_n], L1→X}

? inversa([a_2,..., a_n], R1), concatenar(R1, [a_1], X).

(2) | **2.** {X2→a_2, Y2→ [a_3, a_4,..., a_n], L2→R1}

? inversa([a_3,..., a_n], R2), concatenar(R2, [a_2], R1), concatenar(R1, [a_1], X).

•
•
•

(n) | **2.** {Xn→a_n, Yn→ [ ], Ln→Rn-1}

? inversa([ ], R_n), concatenar(R_n, [a_n], R_{n-1}),... , concatenar(R2, [a_2], R1), concatenar(R1, [a_1], X).

(n+1) | **1.** {Rn→ [ ]}

? concatenar([ ], [a_n], R_{n-1}),..., concatenar(R_1, [a_1], X).

(n+2) | **1'.** {Ln+2→ [a_n], Rn-1→ [a_n]}

? concatenar([a_n], [a_{n-1}], R_{n-2}),..., concatenar(R_1, [a_1], X).

(n+3) | **2'.** {Xn+3→ a_n, Yn+3 → [ ], Zn+3 → [a_{n-1}], R_{n-2} → [a_n | Un+3]}

? concatenar([ ], [a_{n-1}], U_{n+3}),..., concatenar(R_1, [a_1], X).

•
•

Para ganar en eficiencia, evitando el uso de "concatenar", veremos otra versión que simula el siguiente algoritmo iterativo:

```
Inversa (dato:lista): lista
   T := [ ]                                  inicialización
   mientras  dato <> [ ]  hacer
     T := [cabeza(dato)|T]                   bucle
     dato := resto(dato)
   finmientras
   devolver  T                               salida
```

Programa 2 de "inversa":

```
1. inversa(X, Y):- inversa(X, Y, [ ]).        % inicialización
2. inversa([X|Z], Y, T):- inversa(Z, Y, [X|T]).   % bucle
3. inversa([ ], T, T).                        % salida
```

Se ha utilizado un tercer argumento que funciona como la variable auxiliar T del algoritmo iterativo. La primera sentencia corresponde a la inicialización, la segunda al bucle y la tercera a la salida. Con el nuevo programa, conseguimos obtener la solución del problema en un número de pasos de orden **n**:

? inversa([a_1,...,a_n], X).

(1) **1.** | {X1→ [a_1, ..., a_n], Y1→ X}
? inversa([a_1,...,a_n], X, [ ]).

(2) **2.** | {X2→ a_1, Z2→ [a_2, ..., a_n], Y2→ X, T2→ [ ]}
? inversa([a_2,...,a_n], X, [a_1]).

(3) **2.** | {X3→ a_2, Z3→ [a_3, ..., a_n], Y3→ X, T3→ [a_1]}
? inversa([a_3,...,a_n], X, [a_2, a_1]).

•
•

(n+1) **2.** |
? inversa([ ], X, [a_n,...,a_1]).

(n+2) **3.** | {Tn+2 → [a_n,...,a_1], X → [a_n,...,a_1]}
□
ÉXITO **X=[a_n,...,a_1]**

Esta segunda propuesta gana en eficiencia a la primera, al construir paso a paso la solución del problema sin dejar pendiente tareas posteriores a la llamada recursiva. Es decir, tras la llamada recursiva no hay ningún otro objetivo a resolver, todas las tareas necesarias se realizan antes de la nueva llamada a inversa (ó directamente en la misma llamada). Cuando la llamada recursiva se produce como último objetivo de cualquiera de las reglas del procedimiento que se está definiendo, se denomina ***recursión de cola***.
Aunque tanto el programa 1 como el 2 tienen una definición recursiva y resuelven el mismo problema, sus comportamientos son diferentes. El programa 2 simula el algoritmo iterativo indicado y por tanto muestra un ***comportamiento iterativo***. Es decir, ***la recursión de cola*** equivale a un ***comportamiento iterativo***. Por su parte el primer programa no tiene recursión de cola y por tanto su comportamiento no es iterativo. Diremos simplemente que su ***comportamiento es recursivo***.

Conocida la diferencia entre los dos tipos de comportamiento, ¿cuál es el comportamiento de los tres ejemplos de la sección anterior?
Para los dos primeros ejemplos, **miembro** y **concatenar**, la contestación es directa. Sin embargo, el tercer ejemplo es algo más extenso y tendríamos que hacer un análisis procedimiento por procedimiento.
Se deja como ejercicio a resolver cuál sería la respuesta para los procedimientos *piezabasica, ensamblaje, piezasde* y *listapiezasde*.

Otro ejemplo similar consiste en las versiones recursiva e iterativa del problema de "*hallar el factorial de un número*", donde se hace uso del predicado predefinido **is** para operaciones aritméticas (que veremos más adelante).

Programa 1- factorial

1. **fact(0, 1).**
2. **fact(N, Y):- M is N-1, fact(M, Z), Y is N*Z.**

Programa 2-factorial

1. **fact(X, Y):- fact-aux(X, 1, Y) .**
2. **fact-aux(0, Y, Y) .**
3. **fact-aux(N, Y, Z):- U is N*Y, M is N-1, fact-aux(M, U, Z).**

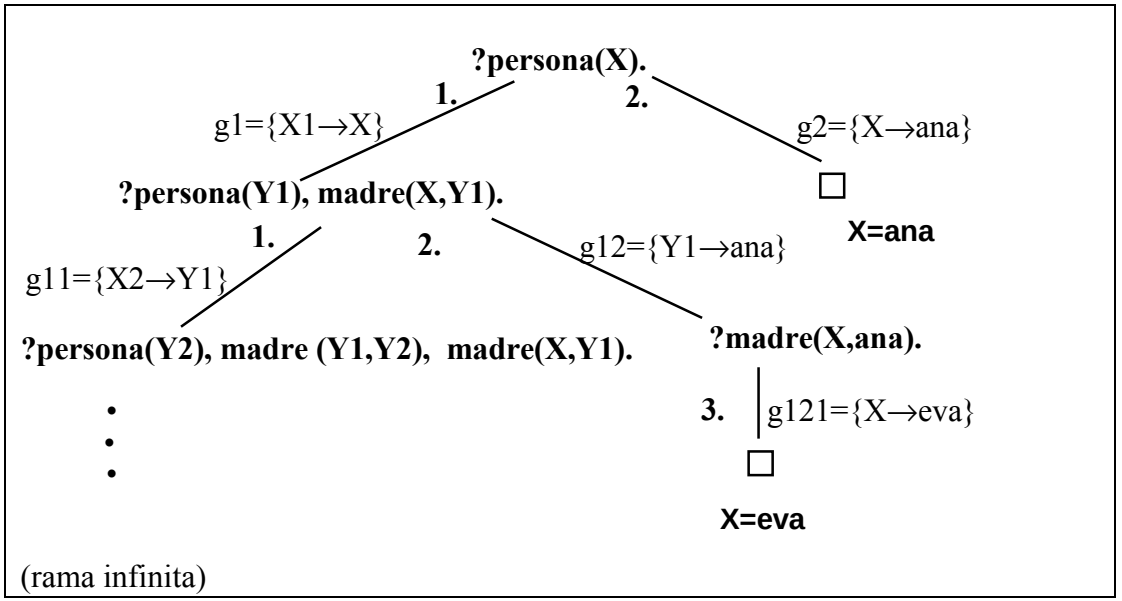
Se deja como ejercicio ver cuál es el árbol de llamadas para la pregunta **? fact(3, F)** con cada uno de los programas. ¿Cuál de los programas tiene comportamiento iterativo y cuál es el algoritmo iterativo asociado a él?

2.3. Recursión a izquierdas

La recursión a izquierdas se produce cuando en un paso de resolución, una regla PROLOG con definición recursiva, genera una nueva llamada, cuyo primer fin es equivalente al fin original (que causó el uso de esa regla). El árbol de búsqueda de soluciones obtenido con un programa de estas características tendrá una rama infinita. Por ejemplo:

Programa 1-persona

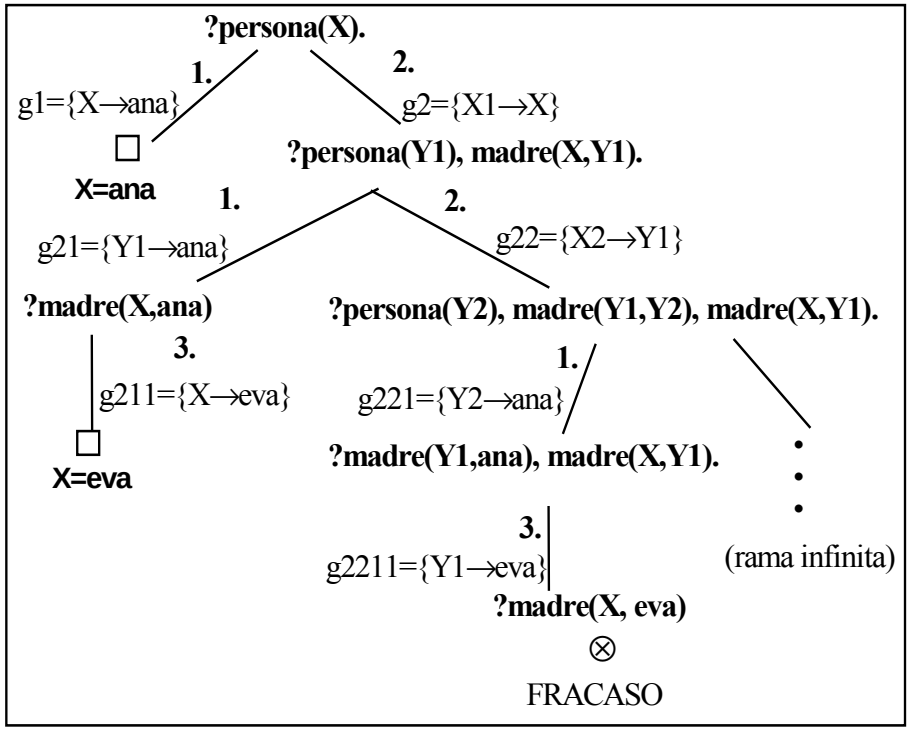
1. **persona(X) :- persona(Y), madre(X, Y).**
2. **persona(ana).**
3. **madre(eva, ana).**



En algunos casos, alterar el **orden de las sentencias** dentro del programa puede evitar que la rama infinita quede a la izquierda y, por tanto, permite a PROLOG dar soluciones aun cuando el árbol siga siendo infinito.
En el ejemplo, intercambiando el orden de 1 y 2 se obtienen dos respuestas, pero si pedimos una tercera respuesta el programa no parará.

Programa 2-persona

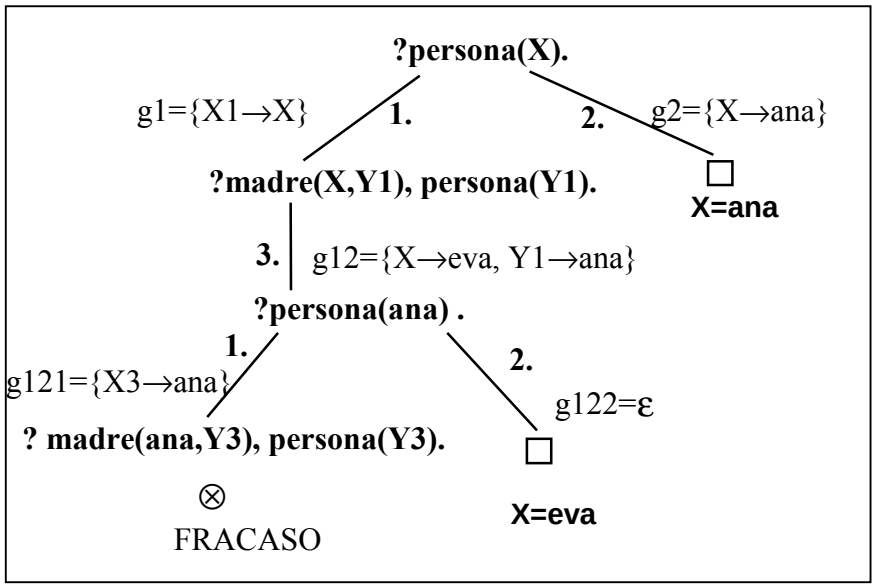
1. **persona(ana).**
2. **persona(X) :- persona(Y), madre(X,Y).**
3. **madre(eva, ana).**



En otros casos, alterando el **orden de los literales** dentro de una sentencia puede evitarse dicha recursión a izquierdas y encontrar un árbol de búsqueda de soluciones finito. En el ejemplo, intercambiando el orden de los literales de la regla de **persona**, se obtiene:

Programa 3-persona

1. **persona(X) :- madre(X,Y), persona(Y).**
2. **persona(ana).**
3. **madre(eva, ana).**



Desde el punto de vista metodológico de la programación, si el problema a tratar tiene un número finito de respuestas, se deben escribir los literales de cada sentencia en el orden adecuado para conseguir (si existe) un **árbol de búsqueda finito.** Esto asegura la terminación de dicho programa.

2.4. Generación infinita

En algunos casos puede interesar que el árbol de búsqueda de soluciones tenga una rama infinita, como es el caso de los predicados generadores de infinitas soluciones. Por ejemplo, el siguiente programa sirve como generador de números naturales:

1. **generanat(0).**
2. **generanat(N) :- generanat(M), N is M+1.**

Para representar las infinitas soluciones del problema, es preciso que el árbol para **?generanat(X)** sea infinito. El orden de las sentencias 1 y 2 es esencial para obtener las respuestas.