

# Prácticas de laboratorio de LTP (Parte III : Programación Lógica)

## Práctica 8: Introducción a las listas en Prolog



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

Jose Luis Pérez  
[jlperez@dsic.upv.es](mailto:jlperez@dsic.upv.es)

# 1. Objetivo de la práctica

Los objetivos de la práctica son dos:

- 1) Introducir el uso de **listas** en **Prolog**.
- 2) Introducir el uso de la **recursividad** en dicho lenguaje.

**Nota:** En Poliformat se dispone del fichero **pr8.pl** que contiene código prolog que utilizaremos en los ejercicios a realizar durante la sesión.

**Ejercicio 19 (práctica 7) :** Escribir un predicado **factorial** para poder responder con éxito a la consulta:

```
?- factorial(6,Y).
```

¿Sirve este mismo predicado resolver la siguiente consulta?

```
?- factorial(X,24).
```

**Solución:**

**factorial.pl**

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

```
factorial(N,F) :- N > 1, N1 is N-1, factorial(N1,F1), F is N*F1.
```

```
?- consult(factorial).
```

```
...
```

## 2. Listas en Prolog

En Prolog, la representación interna de las listas puede verse como un caso especial de la representación de los términos. En este contexto, una lista puede definirse inductivamente como sigue:

**[]** → es una lista (vacía);

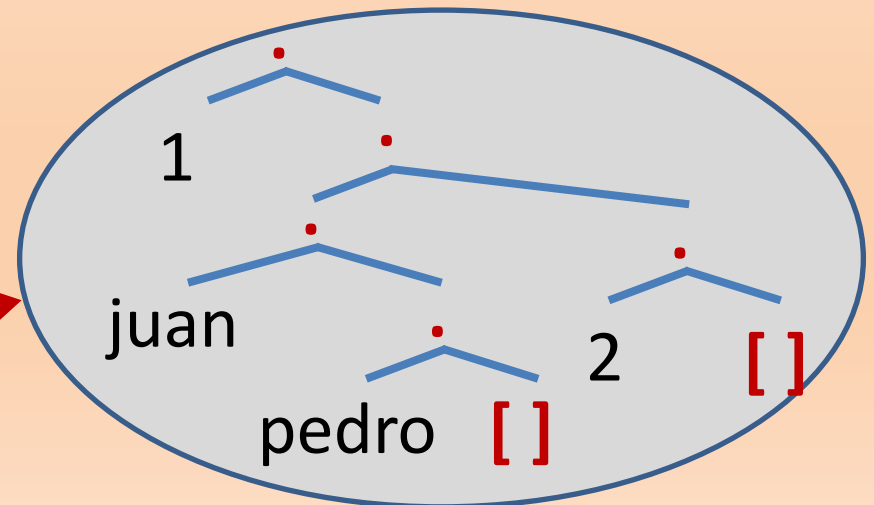
**.(E,L)** → es una lista (no vacía), si **E** es un elemento y **L** una lista.  
El elemento **E** se denomina la **cabeza** y **L** es la **cola** de la lista.

Es decir, Prolog define la listas como términos compuestos mediante los constructores **[]** y **.(\_,\_)**. Veamos ejemplos de listas válidas:

**.(1,.(2,.(3,[])))**

**.(a,.(b,.(c,.(d,[]))))**

**.(1,.(.(juan,.(pedro,[])),.(2,[])))**



## 2.1 Un representación simple para las listas

En la notación alternativa, el constructor prefijo binario “ $.(_,_)$ ” se sustituye por el operador infijo “ $[_ | _]$ ”.

$.(E,L)$    $[E | L]$

$.(1,.(2,.(3,[ ])))$	$[1   [2   [3   [ ]]]]$
$.(a,.(b,.(c,.(d,[ ]))))$	$[a   [b   [c   [d   [ ]]]]]$
$.(1,.(.(juan,.(pedro,[ ])),.(2,[ ])))$	$[1   [[juan   [pedro   [ ]]]   [2   [ ]]]]$

Esta notación todavía no es lo suficientemente legible, por lo que Prolog admite y comprende las siguientes abreviaturas:

$[e1 | [e2 | [e3 | ...[en | L]]]]$  se abrevia a  $[e1, e2, e3, ..., en | L]$   
 $[e1, e2, e3, ..., en | [ ]]$  se abrevia a  $[e1, e2, e3, ..., en]$

$[1   [2   [3   [ ]]]]$	$[1,2,3]$
$[a   [b   [c   [d   [ ]]]]]$	$[a,b,c,d]$
$[1   [[juan   [pedro   [ ]]]   [2   [ ]]]]$	$[1,[juan,pedro],2]$

# Ejercicios

**Ejercicio 1:** Edita un programa Prolog e introduce los siguientes hechos:

```
countTo(1,[1]).  
countTo(2,[1,2]).  
countTo(3,[1,2,3]).  
countTo(4,[1,2,3,4]).
```

Carga el código y escribe las consultas:

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

¿Es **Y** un elemento o una lista en cada caso?

**Ejercicio 2:** Prueba las siguientes consultas:


```
?- countTo(4,[H | T]).  
?- countTo(4,[H1,H2 | T]).  
?- countTo(4,[_,X | _]).  
?- countTo(2,[H1,H2 | T]).  
?- countTo(2,[H1,H2,H3 | T]).
```

¿Que ves? ¿Por que? ¿Que efecto tiene usar el símbolo de subrayado `_`?

# Ejercicios

Es fácil definir inductivamente un predicado **islist** que confirme si un dato es o no una lista, definiendo un **caso base** (hecho) para la lista vacía **[]** y un **caso recursivo** (regla) para listas no vacías, formadas usando el constructor **[ | ]**:

```
% islist(L) o is_list(L), L es una lista  
islist([]).  
islist([_ | T]) :- islist(T).
```



```
?- islist([a | a]).  
False
```

La mayoría de los sistemas Prolog proporcionan una buena colección de predicados predefinidos para la manipulación de listas.

Dos de los mas usuales son **member** y **append**:

- El predicado **member(E,L)** permite comprobar si un elemento **E** esta contenido en la lista **L**, si bien este predicado es invertible y puede emplearse también para extraer los elementos de una lista.
- El predicado **append(L1,L2,L)** permite concatenar las listas **L1** y **L2** para formar la lista **L**. Este predicado es predefinido en Prolog pero puedes definirlo tu mismo:

# Ejercicios

**Ejercicio 3:** Prueba las siguientes consultas, escribiendo “;” tras cada respuesta del interprete para explorar todo el espacio de soluciones:

```
?- member(2,[5,2,3,2]).  
?- member(X,[5,2,3]).
```

**Ejercicio 4:** Escribe tu propio predicado **mymember** a partir del siguiente código (con errores) que debes corregir para obtener la solución.

```
mymember(E,[E,_]).  
mymember(E,[H|L]) :- mymember(H,L).
```

El ejercicio es trivial sabiendo que la definición del predicado **member** es:

```
% member(E,L), E pertenece a L  
member(E,[E|_]).  
member(E,[_|L]) :- member(E,L).
```

**Ejercicio 5:** Define un predicado llamado **myappend** que haga exactamente lo mismo que **append** pero estando este definido por ti mismo. El código siguiente es un punto de comienzo pero hay un error en el que debes encontrar y corregir.

```
myappend([],L,L).  
myappend([E|L1],L2,[X|L3]) :- X = E, myappend(L1,L2,L1).
```



# Ejercicios

De nuevo, la solución del **ejercicio 5** es trivial sabiendo que la definición en Prolog del predicado **append** es como sigue:

```
% append(L1,L2,L), la concatenacion de L1 y L2 es L
append([],L,L).
append([E | L1],L2,[E | L3]) :- append(L1,L2,L3).
```

Ten en cuenta que a veces Prolog da advertencias sobre “**singleton variables**”. Este aviso significa que hay una variable que aparece en un solo lugar y que no se usa en ninguna otra parte de la regla, por lo que es mejor usar una variable anónima (representada con un subrayado).

**Ejercicio 6:** Prueba las siguientes consultas, escribiendo “;” tras cada respuesta del interprete para explorar todo el espacio de soluciones:

```
?- append([a,Y],[Z,d],[X,b,c,W]).
?- append(L1,L2,[a,b,c,d]).
```

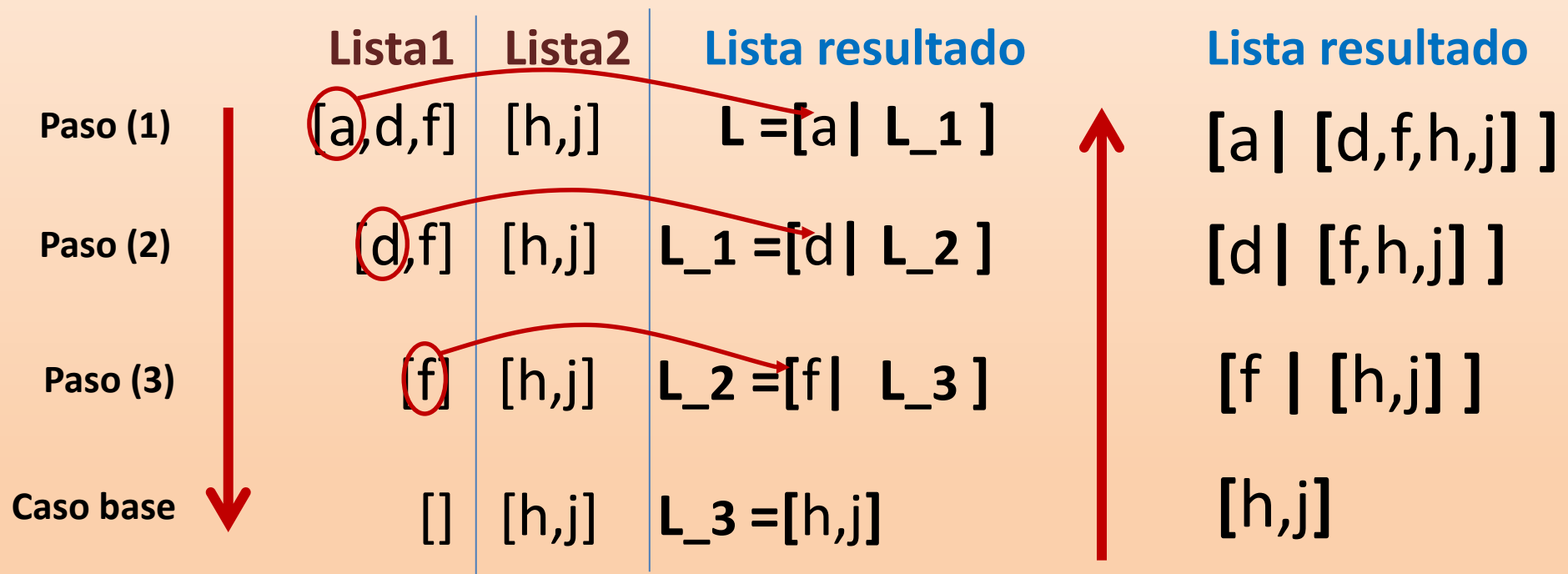
Con la ayuda de los predicados **member** y **append** se puede definir una gran variedad de predicados, como veremos en los siguientes subapartados.

# Traza de append

```
% append(L1,L2,L), la concatenacion de L1 y L2 es L
append([],L,L).
append([E | L1],L2,[E | L3]) :- append(L1,L2,L3).
```

? append([a,d,f],[h,j],L).

L=[a,d,f,h,j]



Como puede verse en esta traza, **el coste es lineal con la longitud de la primera cadena**, ya que se van insertando los elementos de la primera lista uno a uno al comienzo de la segunda.

## 2.2 Aplanar una lista

Vamos a definir la relacion **flatten(L,A)**, donde **L** es una lista de listas, tan compleja en su anidamiento como queramos imaginar, y **A** es la lista que resulta de reorganizar los elementos contenidos en las listas anidadas en un único nivel. Por ejemplo:

```
?- flatten([[a,b],[c,[d,e]],f],L).
L = [a,b,c,d,e,f]
```

La versión que damos hace uso del predicado predefinido **atomic(X)**, que comprueba si el objeto que se enlaza o vincula a la variable **X** es o no un objeto simple (atómico), es decir, un entero o una cadena de caracteres:

```
% flatten(L, A), A es el resultado de aplanar L
flatten([], []).
flatten([X|L],[X|P]) :- atomic(X), flatten(L,P).
flatten([X|L],P) :- not(atomic(X)), flatten(X,P_X),
                    flatten(L,P_L), append(P_X,P_L,P).
```

## 2.3 Último elemento de una lista

El elemento **E** en cabeza de una lista  $[E | L]$  es directamente accesible por unificación. Sin embargo, para acceder al ultimo de una lista, es preciso definir un predicado respecto **last(L,U)**, donde **U** es el ultimo elemento de la lista **L**. Este predicado puede definirse como sigue:

```
% last(L,U), U es el ultimo elemento de la lista L  
last([U],U).  
last([_ | L],U) :- last(L,U).
```

En el caso base (hecho) **U** es el último elemento de la lista en caso de que la lista contenga un solo elemento, es decir, la lista sea de la forma  $[U]$ . Con el caso general (regla) se recorre toda la lista, desechando cada uno de los elementos, hasta alcanzar el último.

El predicado **last(L,U)**, también puede definirse de manera mucho mas concisa usando la relación **append**:

```
% last(L,U), U es el ultimo elemento de la lista L  
last(L,U) :- append(_,[U],L).
```

## 2.4 Operaciones con sublistas

El predicado **append** tambien puede utilizarse para definir operaciones con **sublistas**. Si consideramos que un *prefijo* de una lista es un segmento inicial de la misma y que un *sufijo* es un segmento final, podemos definir estas relaciones con gran facilidad haciendo uso del predicado **append**:

```
% prefix(P,L), P es un prefijo de la lista L  
prefix(P,L) :- append(P,_,L).
```

```
% suffix(P,L), P es un sufijo de la lista L  
suffix(P,L) :- append(,P,L).
```

```
% sublist(S,L), S es una sublista de la lista L  
sublist(S,L) :- suffix(L1,L), prefix(S,L1).
```

La ultima definici3n afirma que **S** es una sublista de **L** si **S** esta contenida como prefijo de un sufijo de **L**.

## Observaciones relativas a la eficiencia...

- En muchas ocasiones, **comprobar los dominios** de los argumentos de los predicados definidos **puede afectar el rendimiento** de un programa por lo que a menudo se omiten dichas comprobaciones.
- Por otro lado, **la llamada indirecta a otros predicados en el cuerpo de una regla también puede afectar al rendimiento**. Por este motivo, puede ser aconsejable **sustituir la llamada por el cuerpo del predicado** al que se llama (renombrando adecuadamente las variables de la clausula para que su cabeza coincida con la llamada).

A esta técnica se le llama **unfolding**. Veamos un ejemplo aplicado a **sublist**:

% sublist(S,L), S es una sublista de la lista L

sublist(**S**,L) :- **suffix(L1,L)**, **prefix(S,L1)**.

suffix(**P**,L) :- append(**\_**,**P**,L).

prefix(**P**,L) :- append(**P**,**\_**,L).

append(**\_**,**L1**,L)

append(**S**,**\_**,**L1**)

sublist(**S**,L) :- append(**\_**,**L1**,L), append(**S**,**\_**,**L1**).

## 2.5 Invertir una lista:

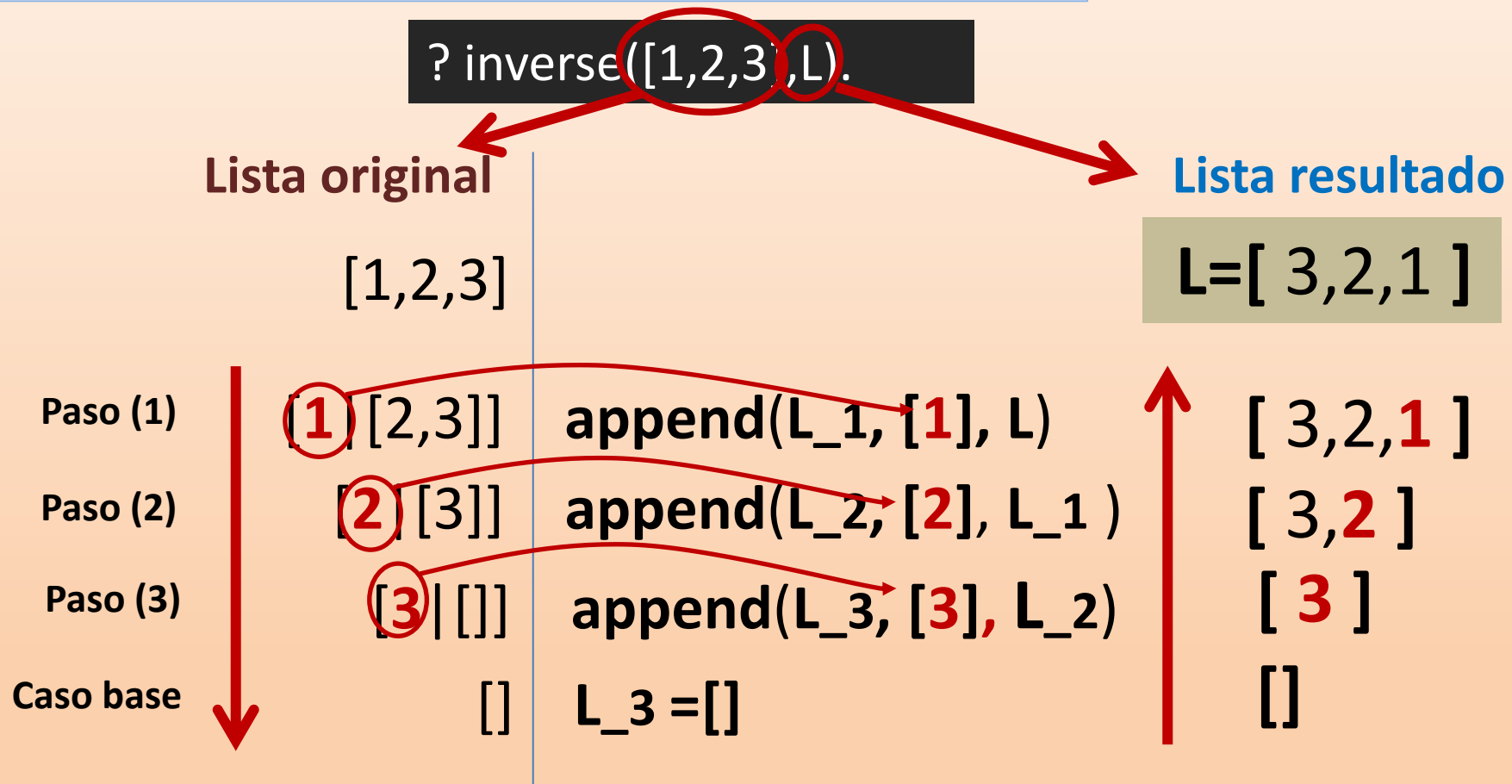
La inversión de listas puede definirse de forma directa en términos del predicado **append**:

```
% inverse(L,I), I es la lista que resulta de invertir L  
inverse([],[]).  
inverse([H|T],L) :- inverse(T,Z), append(Z,[H],L).
```

Esta versión es muy ineficiente debido a que consume y reconstruye la lista original usando el predicado **append**, siendo su coste cuadrático con respecto al número de elementos de la lista que se está invirtiendo.

# Traza de **inverse**

```
% inverse(L,I), I es la lista que resulta de invertir L  
inverse([],[]).  
inverse([H|T],L) :- inverse(T,Z), append(Z,[H],L).
```



**El número de pasos es lineal con la longitud de la lista, pero en cada paso hay que hacer una inserción, cuyo coste también es lineal con la longitud de la lista, por lo que el coste total es cuadrático con la longitud de la lista.**



## 2.5 Invertir una lista:

parámetros de acumulación y recursión de cola

```
% inverse(L,I), I es la lista que resulta de invertir L  
inverse([],[]).  
inverse([H|T],L) :- inverse(T,Z), append(Z,[H],L).
```

Para eliminar las llamadas a **append** y lograr una mayor eficiencia se hace necesario el uso de un parámetro de acumulación que, como indica su nombre, se utiliza para almacenar resultados intermedios. En el caso que nos ocupa, se almacena la lista que acabará por ser la lista invertida, en sus diferentes fases de construcción.

```
% inverse(L,I), I es la lista que resulta de invertir L  
inverse(L,I) :- inv(L,[],I).  
% inv(Lista,Acumulador,Invertida)  
inv([],I,I).  
inv([X|L],A,I) :- inv(L,[X|A],I).
```

# Traza de **inverse** (con parámetro de acumulación)

% inverse(L,I), I es la lista que resulta de invertir L

**inverse(L,I) :- inv(L,[],I).**

% inv(Lista,Acumulador,Invertida)

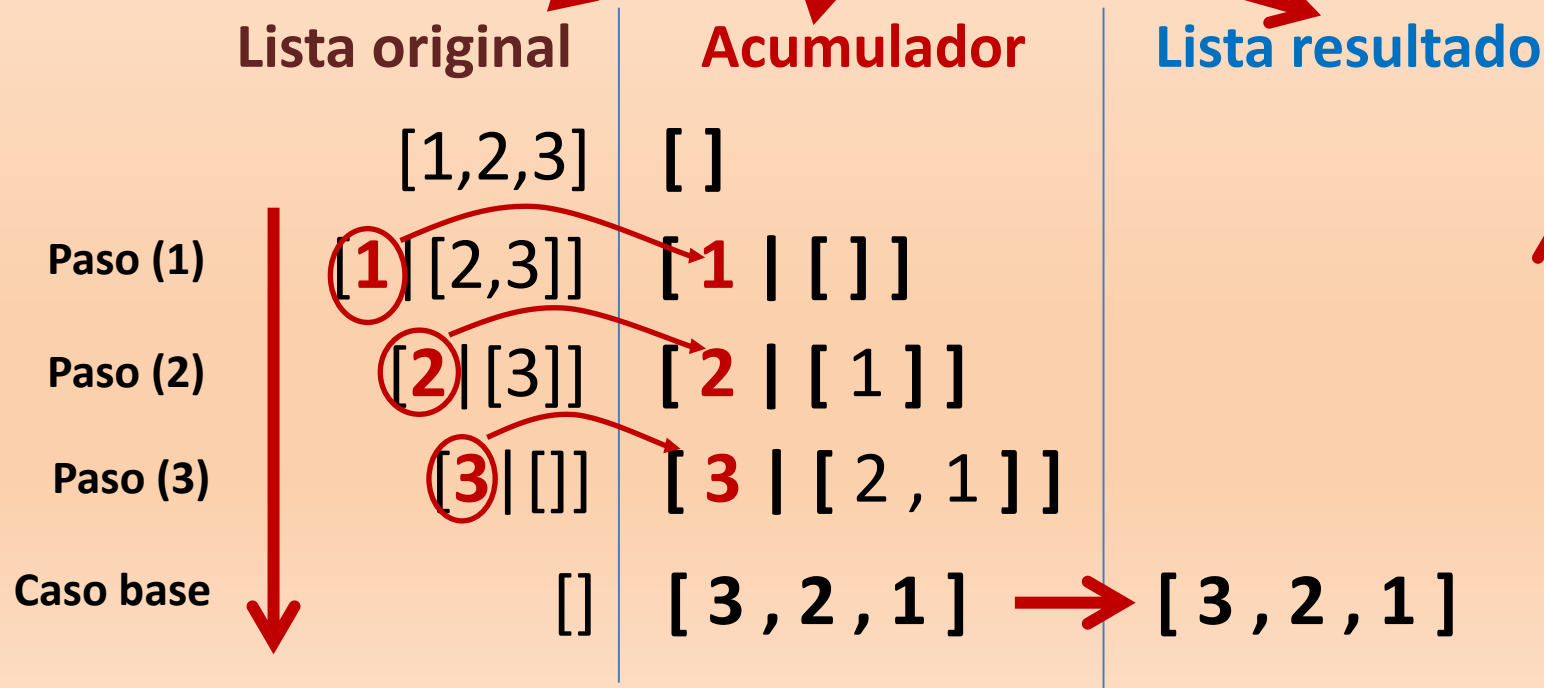
**inv([],I,I).**

**inv([X|L],A,I) :- inv(L,[X|A],I).**

? inverse([1,2,3],L).

I=[ 3,2,1 ]

? inv([1,2,3],[ ]L).



## 2.6 Ejercicios finales con listas

**Ejercicio 7:** Escribe un predicado binario **swap** que acepta una lista y genera una lista similar con los dos primeros elementos intercambiados.

**Ejercicio 8:** ¿Que hace el siguiente predicado misterioso y por que?

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

**Ejercicio 9:** Completa el siguiente programa Prolog para que implemente la operación de comprobar si una colección de elementos es subconjunto de otra:

```
subset([],_).  
subset([A | X],Y) :- member(A,Y),                      .
```

- A) subset(X,Y)
- B) append(X,[A],Y)
- C) subset(Y,X)
- D) member(X,Y)

## 2.6 Ejercicios finales con listas

**Ejercicio 10:** Completa el siguiente programa Prolog que comprueba si una lista esta ordenada:

```
sorted([]).  
sorted([_]).  
_____ :- X =< Y, sorted([Y | Ys]).
```

- A) sorted([X | Y, Ys])
- B) sorted([X, [Y | Ys]])
- C) sorted([X, Y | Ys])
- D) sorted(X, Y, Ys)

**Ejercicio 11:** Completa el siguiente programa lógico para que, dado un entero y una lista de enteros, elimine las ocurrencias de dicho entero de la lista. Los predicados predefinidos == y \== representan la igualdad y la desigualdad, respectivamente. Por ejemplo, la llamada

? remove(3,[1,2,3,1,2,3],L) computa la respuesta L = [1,2,1,2].

```
remove(_, [], []).  
remove(C, [X | R], L) :- X == C, remove(C, R, L).  
remove(C, [X | R], W) :- X \== C, _____.
```

- A) remove(C, R, L), W = [X | L]
- B) remove(C, R, W), L = [X | W]
- C) remove(C, R, L), W = L
- D) remove(C, [X | R], L), W = L