Lenguajes, Tecnologías y Paradigmas de la programación (LTP)

Práctica 8: Introducción a las listas en Prolog

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Sergio Pérez
serperu@dsic.upv.es

Introducción a las listas en Prolog

OBJETIVOS DE LA PRÁCTICA

- Comprender la representación de listas
- Entender, usar y definir operaciones sobre listas

Notación para listas en Prolog

Haskell **Prolog**

Notación para listas en Prolog

	Haskell	Prolog
Lista vacia		
Lista no vacia		

Notación para listas en Prolog

	Haskell	Prolog
Lista vacia →	`[]`	
Lista no vacia →	`(h:t)`	

Notación para listas en Prolog

	Haskell	Prolog
Lista vacia →	`[]`	`[]`
Lista no vacia →	`(h:t)`	

Notación para listas en Prolog

	Haskell	Prolog
Lista vacia →	`[]`	`[]`
Lista no vacia →	`(h:t)`	`[H|T]`

Ejemplo de lista en Prolog

¿Cómo represento la lista [1,2,3] en Prolog?

Ejemplo de lista en Prolog

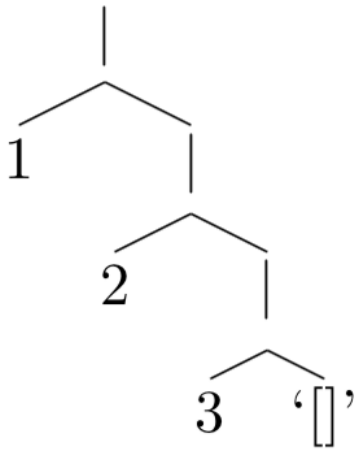
¿Cómo represento la lista [1,2,3] en Prolog?

[1|[2|[3|[]]]]

Ejemplo de lista en Prolog

¿Cómo represento la lista [1,2,3] en Prolog?

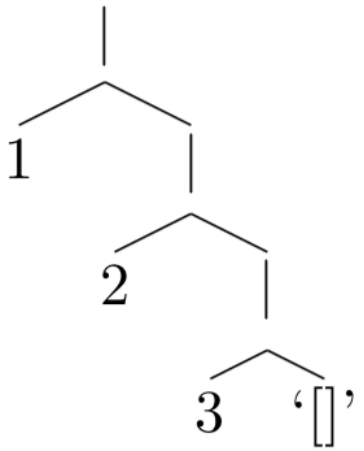
[1|[2|[3|[]]]]



Ejemplo de lista en Prolog

¿Cómo represento la lista [1,2,3] en Prolog?

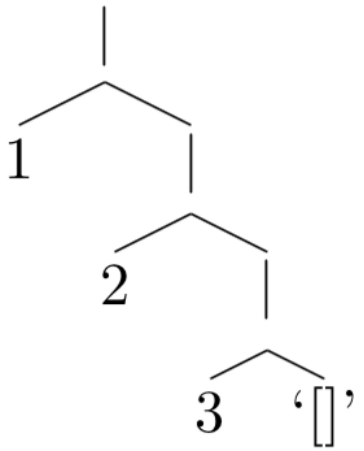
```
[1|[2|[3|[]]]]
[1,2,3|[]]
```



Ejemplo de lista en Prolog

¿Cómo represento la lista [1,2,3] en Prolog?

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



Operaciones sobre listas en Prolog

Operación `member` para saber si un elemento está en una lista

Operaciones sobre listas en Prolog

Operación `member` para saber si un elemento está en una lista

```
% member(E,L), E pertenece a L
```

Operaciones sobre listas en Prolog

Operación `member` para saber si un elemento está en una lista

- Un hecho para el caso base

```
% member(E,L), E pertenece a L
```

Operaciones sobre listas en Prolog

Operación `member` para saber si un elemento está en una lista

- Un hecho para el caso base

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

Operaciones sobre listas en Prolog

Operación `member` para saber si un elemento está en una lista

- Un hecho para el caso base
- Una o más reglas para el caso general

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

Operaciones sobre listas en Prolog

Operación `member` para saber si un elemento está en una lista

- Un hecho para el caso base
- Una o más reglas para el caso general

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

Operaciones sobre listas en Prolog

Operación append para concatenar dos listas

- Un hecho para el caso base
- Una o más reglas para el caso general

Operaciones sobre listas en Prolog

Operación append para concatenar dos listas

- Un hecho para el caso base
- Una o más reglas para el caso general

```
% append(L1,L2,L), la concatenación de L1 y L2 es L
```

Operaciones sobre listas en Prolog

Operación append para concatenar dos listas

- Un hecho para el caso base
- Una o más reglas para el caso general

```
% append(L1,L2,L), la concatenación de L1 y L2 es L
append([],L,L).
```

Operaciones sobre listas en Prolog

Operación append para concatenar dos listas

- Un hecho para el caso base
- Una o más reglas para el caso general

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

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L

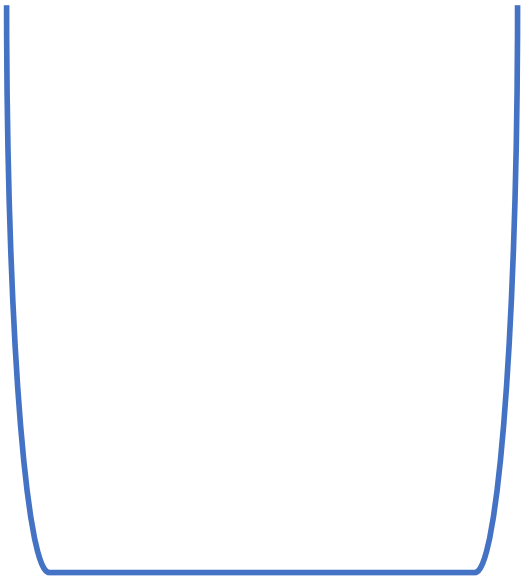
```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L



```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

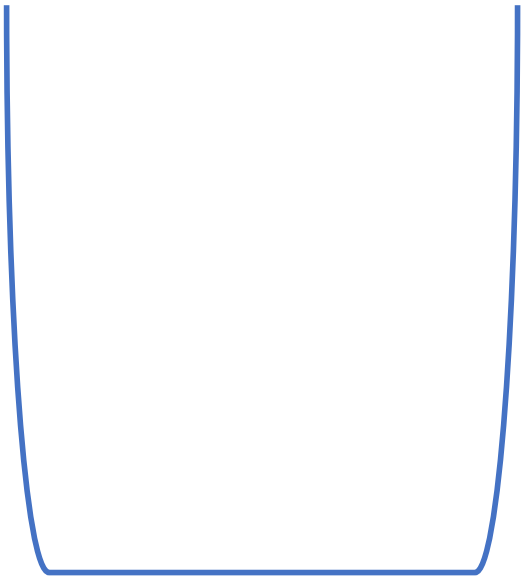
¿inverse [1,2,3]?

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L



```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

```
inverse [1,2,3] = inverse [2,3] ++ [1]
```

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

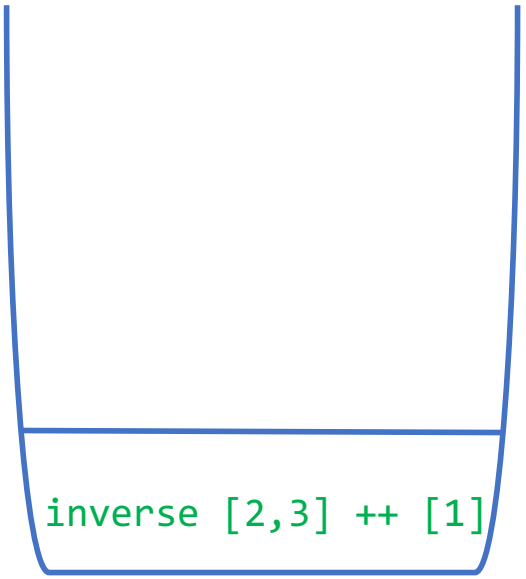
¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

inverse [1,2,3] = inverse [2,3] ++ [1]

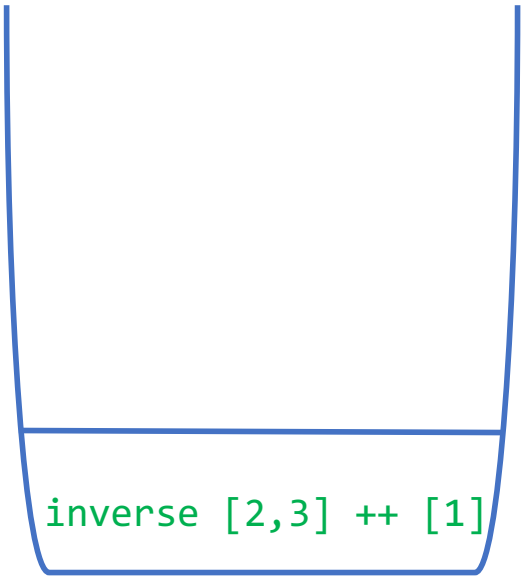


Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L




```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

```
inverse [1,2,3] = inverse [2,3] ++ [1]
inverse [2,3] = inverse [3] ++ [2]
```

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

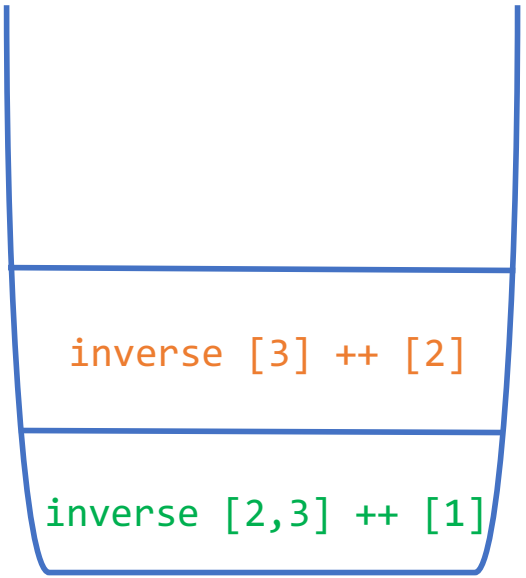
¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

```
inverse [1,2,3] = inverse [2,3] ++ [1]
inverse [2,3] = inverse [3] ++ [2]
```

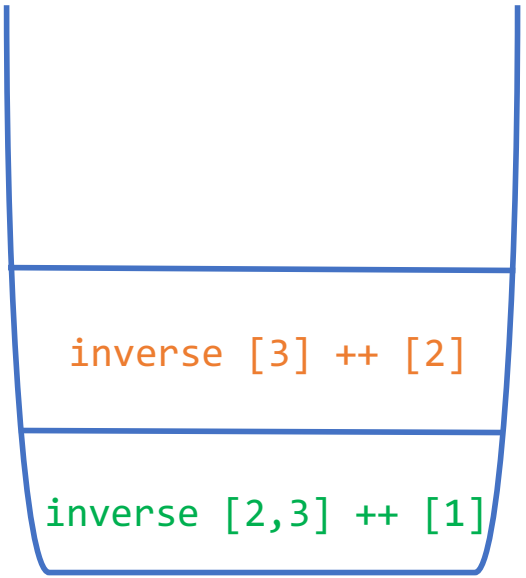


Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L




```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

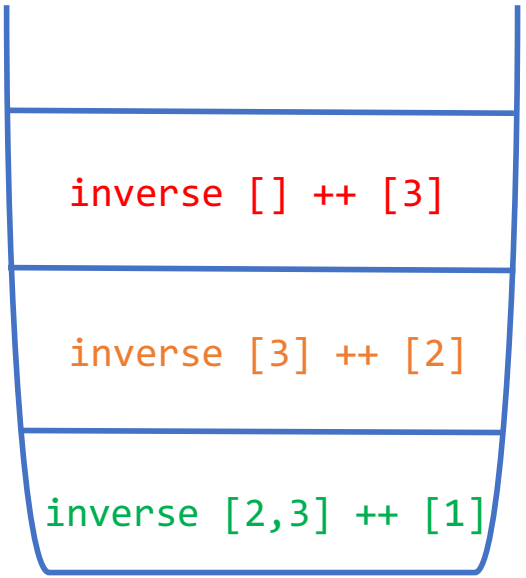
```
inverse [1,2,3] = inverse [2,3] ++ [1]
inverse [2,3] = inverse [3] ++ [2]
inverse [3] = inverse [] ++ [3]
```

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L




```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

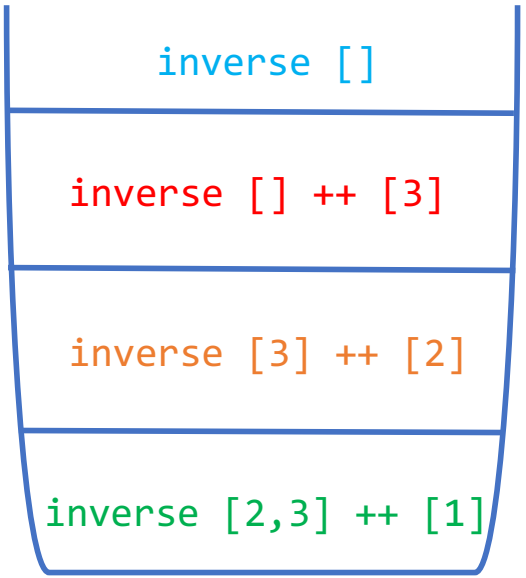
```
inverse [1,2,3] = inverse [2,3] ++ [1]
inverse [2,3] = inverse [3] ++ [2]
inverse [3] = inverse [] ++ [3]
```

Bonus Track: Recursión de Cola

¿En qué consiste la recursión de cola?

¿Cómo funciona la recursión?

Problema: Calcula la lista que resulta de invertir una lista L




```
inverse :: [a] -> [a]
inverse [] = []
inverse (x:xs) = (inverse xs) ++ [x]
```

¿inverse [1,2,3]?

```
inverse [1,2,3] = inverse [2,3] ++ [1]
inverse [2,3] = inverse [3] ++ [2]
inverse [3] = inverse [] ++ [3]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

`Problema: Calcula la lista que resulta de invertir una lista L`

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse L =
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse L =
inverse' :: [a] -> [a] -> [a]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]

inverse' (x:xs) res = inverse' xs ([x] ++ res)
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

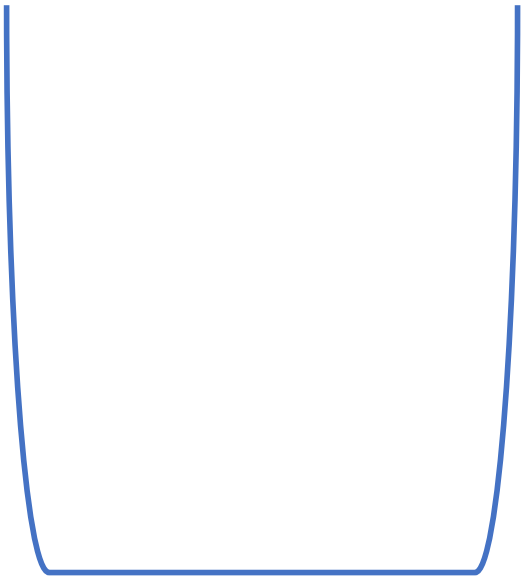
```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L



```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
```

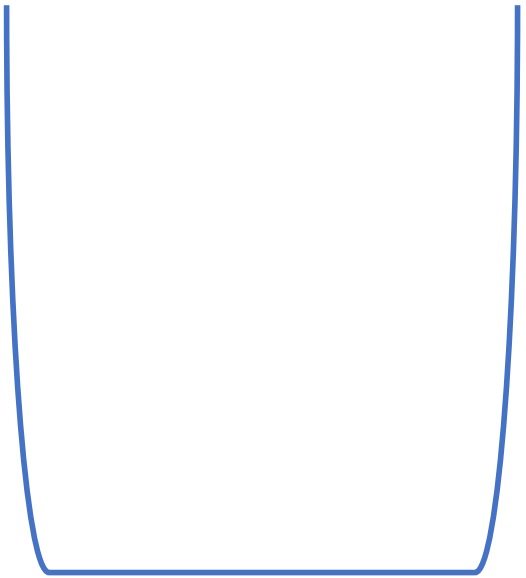
¿inverse [1,2,3]?

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L



```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse' [1,2,3] []
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

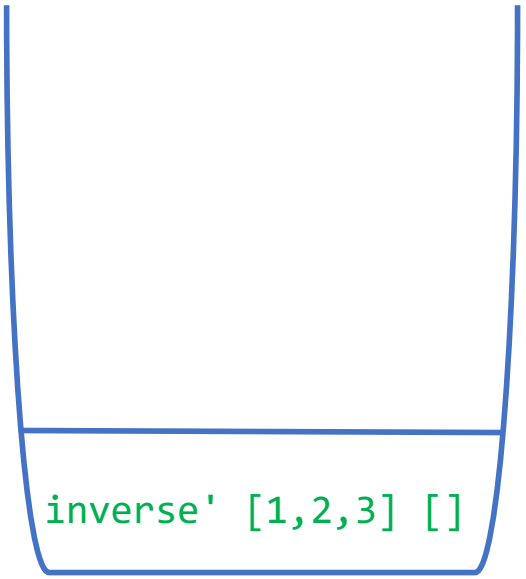
Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
```

¿inverse [1,2,3]?

```
inverse [1,2,3] = inverse' [1,2,3] []
```

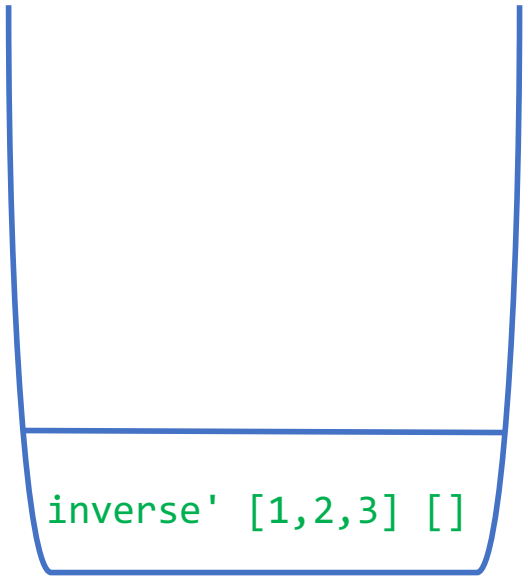


Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L




```
inverse :: [a] -> [a]
inverse L = inverse’ L []
inverse’ :: [a] -> [a] -> [a]
inverse’ [] res = res
inverse’ (x:xs) res = inverse’ xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse’ [1,2,3] []
inverse’ [1,2,3] [] = inverse’ [2,3] [1]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

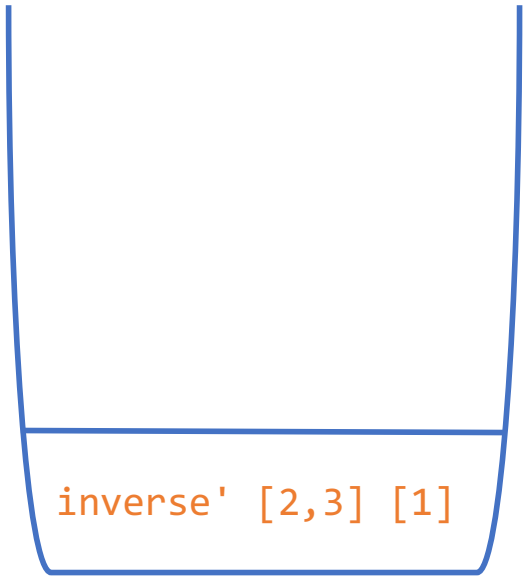
Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
```

¿inverse [1,2,3]?

```
inverse [1,2,3] = inverse' [1,2,3] []
inverse' [1,2,3] [] = inverse' [2,3] [1]
```

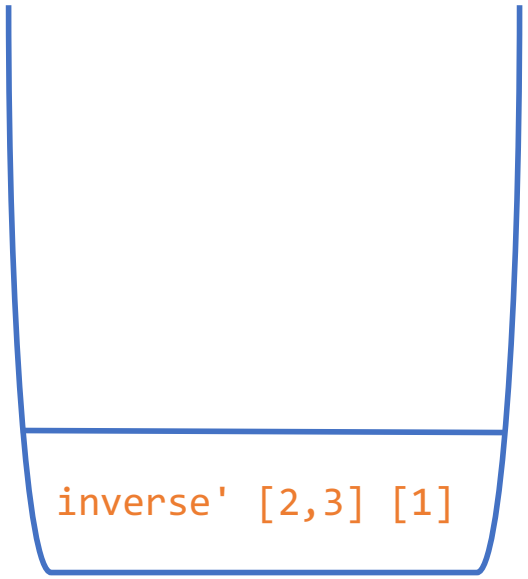


Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L



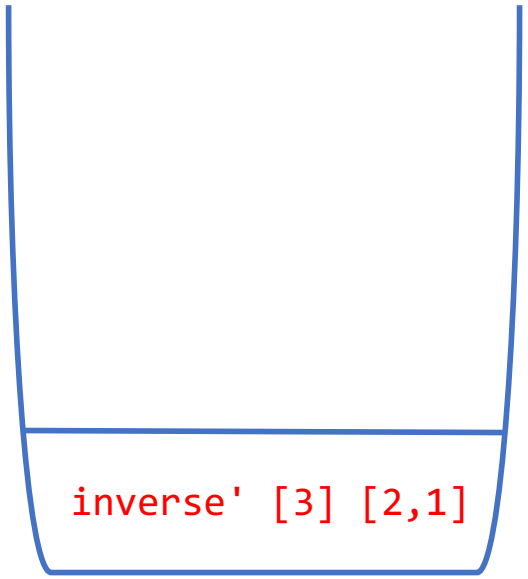

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse' [1,2,3] []
inverse' [1,2,3] [] = inverse' [2,3] [1]
inverse' [2,3] [1] = inverse' [3] [2,1]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L



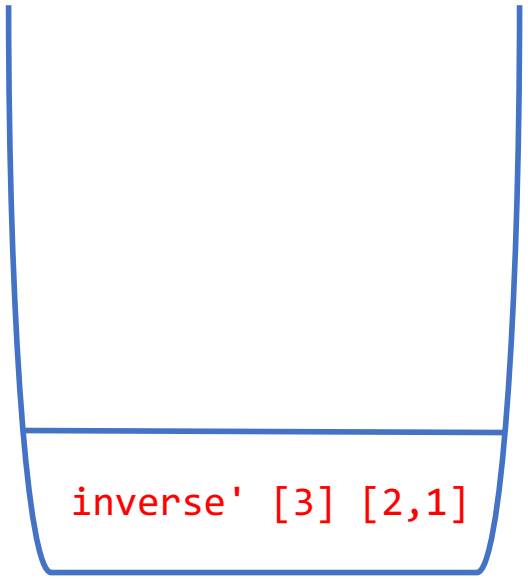

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse' [1,2,3] []
inverse' [1,2,3] [] = inverse' [2,3] [1]
inverse' [2,3] [1] = inverse' [3] [2,1]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L



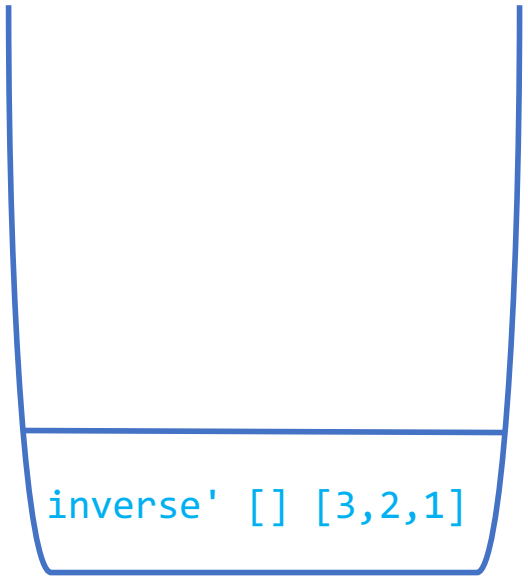

```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse' [1,2,3] []
inverse' [1,2,3] [] = inverse' [2,3] [1]
inverse' [2,3] [1] = inverse' [3] [2,1]
inverse' [3] [2,1] = inverse' [] [3,2,1]
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista L




```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse' [1,2,3] []
inverse' [1,2,3] [] = inverse' [2,3] [1]
inverse' [2,3] [1] = inverse' [3] [2,1]
inverse' [3] [2,1] = inverse' [] [3,2,1]
```

Bonus Track: Recursión de Cola

```
% inverse(L,I), I es la lista resultado de invertir L
```

Bonus Track: Recursión de Cola

```
% inverse(L,I), I es la lista resultado de invertir L


% Usando un parámetro de acumulación.
inverse(L,I) :- inv(L,[],I).
```

Bonus Track: Recursión de Cola

```
% inverse(L,I), I es la lista resultado de invertir L

% Usando un parámetro de acumulación.
inverse(L,I) :- inv(L,[],I).

% inv(Lista, Acumulador, Invertida)
inv([],I,I).
```

Bonus Track: Recursión de Cola

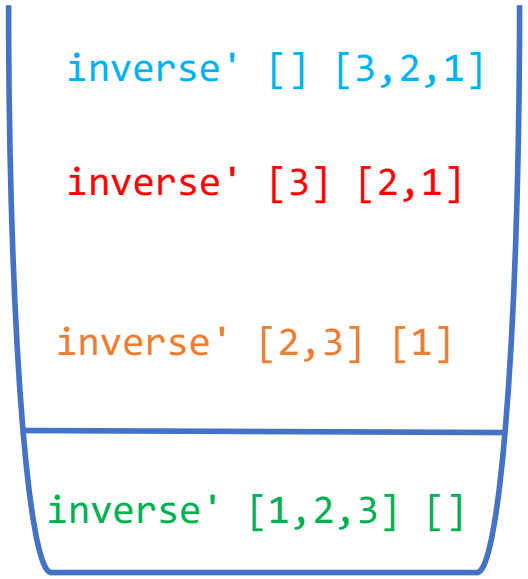
```
% inverse(L,I), I es la lista resultado de invertir L

% Usando un parámetro de acumulación.
inverse(L,I) :- inv(L,[],I).

% inv(Lista, Acumulador, Invertida)
inv([],I,I).
inv([X|L],A,I) :- inv(L,[X|A],I).
```

Bonus Track: Recursión de Cola

¿Podemos hacer algo para no apilar tantas llamadas?

Sí, almacenar el resultado en un parámetro de la llamada.

Problema: Calcula la lista que resulta de invertir una lista list




```
inverse :: [a] -> [a]
inverse L = inverse' L []
inverse' :: [a] -> [a] -> [a]
inverse' [] res = res
inverse' (x:xs) res = inverse' xs ([x] ++ res)
                ¿inverse [1,2,3]?
inverse [1,2,3] = inverse' [1,2,3] []
inverse' [1,2,3] [] = inverse' [2,3] [1]
inverse' [2,3] [1] = inverse' [3] [2,1]
inverse' [3] [2,1] = inverse' [] [3,2,1]
```