

# INTART

## Práctica 3: PROLOG

## Introducción:

En esta práctica se nos manda realizar una serie de ejercicios de un lenguaje de programación nuevo, PROLOG. Por lo que hemos observado en estas prácticas, este lenguaje es algo diferente a los demás lenguajes que hemos cursado, aunque mantiene muchas similitudes con Lisp, otro lenguaje que hemos cursado en esta asignatura en particular. Este lenguaje pertenece al paradigma lógico, debido a su alto componente lógico, como su uso basado en lógica de predicados.

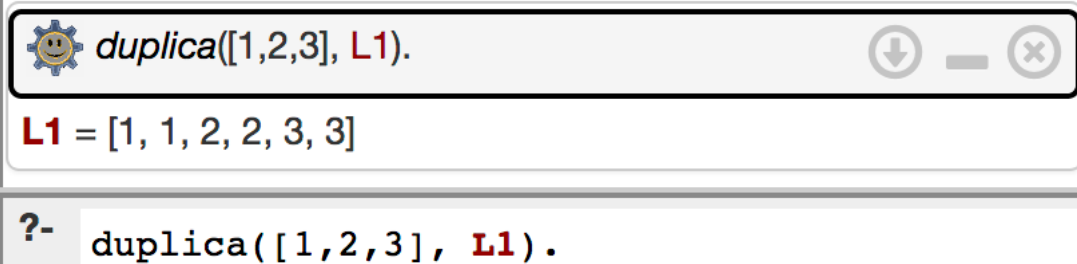
## Ejercicio 1

Para este primer ejercicio se nos pedía una función la cual dada una lista, duplicase su contenido.

Apoyándonos en la lógica de listas de PROLOG, lo que hicimos fue implementar una función la cual extrayendo el primer valor de la lista inicial, se añadiese a la segunda lista. Esta segunda lista, duplicaría los valores que se fuese encontrando, de tal manera que, si el primer valor que encontraba en una lista [1,2,3], sería 1, dejando la lista tras dicha iteración como: [1,1,2,3].

```
duplica([], []).  
duplica([X|R], [X, X|S]) :- duplica(R, S).
```

Cuando ejecutamos en el interprete de PROLOG obtenemos la siguiente salida:



The screenshot shows a Prolog interpreter window with a title bar containing a gear icon, a download icon, a minus icon, and a close icon. The main area displays the query `duplica([1,2,3], L1).` and the result `L1 = [1, 1, 2, 2, 3, 3]`. Below the result, there is a prompt `?- duplica([1,2,3], L1).`

## Ejercicio 2

Para este segundo ejercicio, lo que se realiza es una función la cual invierte una lista en otra, y que verifica que, dadas dos listas, si una es la inversa de la otra.

En este ejercicio se nos pide también que implementemos una función concatena, la cual realiza la misma función que la ya dada por PROLOG `'append'`.

Primero implementamos la función concatena. Para ello nos volvimos a apoyar en las propiedades de listas de PROLOG, usando el operador '|', como vemos en el código adjunto:

```
concatena([], L, L).  
concatena([X|L1], L2, [X|L3]) :-  
    concatena(L1, L2, L3).
```

La lógica de este predicado es sencillo, tenemos dos listas a concatenar, L1 y L2, las cuales una vez concatenadas, estarán en L3. Para ello, empezaremos obteniendo el primer elemento de la lista de L1, y añadiéndoselo a L3, de tal manera que tal y como nos dicta la lógica de PROLOG, "si está en la primera lista, deber estar en la final" que es la que obtendrá la concatenación de ambas listas. Una vez que todos los elementos del L1, estén en L3, nos apoyamos en el caso base del predicado para hacer el resto, el cual nos dice que, si los elementos de la segunda lista, están en dicha segunda lista, entonces deberán estar en la lista final también.

Una vez que esta función ha sido implementada, decidimos jugar con su lógica para implementar invierte, como se ve en el código:

```
invierte([], []).  
invierte([X|L], R) :-  
    invierte(L, S),  
    concatena(S, [X], R).
```

En esta función lo que hemos hecho ha sido coger el primer elemento de la lista L, y realizar llamadas recursivas hasta que cada elemento de cada lista estuviese en una ramificación distinta. Una vez que esto ocurre, es cuando empezamos a concatenar los elementos de la lista L, gracias al '*backtracking*' de PROLOG, el cual sabe que el último elemento que partió de la lista, es el primero en recogerse, y por lo tanto en concatenarse a la lista final. Esto se hace con todos los elementos de L hasta que se llega al primero.

Como vemos en esta captura, se consigue invertir la lista, en su inversa:



### Ejercicio 3

Para este ejercicio, lo que se nos pedía era la resolución de un nuevo predicado, 'palíndromo', el cual se satisfacía si la lista que le enviamos es un palíndromo, es decir, que es leído igual de izquierda a derecha que de derecha a izquierda.

Para ello, nos apoyamos en una función previamente implementada, *invierte*, la cual invertía una lista y la copiaba en otra, y es que es ahí donde se desarrolla toda la lógica que nos llevó a resolver esta función.

Como vemos en el código es *invierte* la que hace todo nuestro trabajo:

```
palindromo([L]) :- invierte(L, L).
```

El predicado *invierte*, como todos los predicados en PROLOG puede actuar para obtener resultados o para chequear los mismos, en este caso nuestro predicado comprueba, y usamos *invierte* como un predicado que comprueba, y no como uno que obtiene resultados.

Como explicamos previamente, el *backtracking* de PROLOG es una herramienta de la que se puede sacar mucho provecho si se usa en determinadas situaciones. En este caso *invierte* vuelve a utilizarlo, pero para chequear que si *invierte* la lista, obtiene la inicial en ambos casos, es decir, que los elementos de ambas listas tienen los mismo elementos pero en las posiciones contrarias de la lista con respecto a sí mismos.

### Ejercicio 4


En este ejercicio se hará el predicado *divide*, éste predicado se usará para, dada una lista, obtener los N primeros elementos en una y el resto en otra.

```
divide(L, 0, [], L).  
divide([X|L], N, [X|L1], L2) :-  
    N > 0,  
    M is N-1,  
    divide(L, M, L1, L2).
```


Para resolverlo, lo que hacemos es establecer un contador el cual no irá diciendo cuantos elementos faltan por obtener de la lista original, para insertarla en la lista con los N-primeros elementos. Posteriormente con el caso base establecemos que si el contador llega a 0, la lista con los elementos que quedan de la lista original será la segunda lista.

La salida de este predicado es la siguiente:

Para la obtención de resultados:

```
 divide([1,2,3,4], 2, L1, L2).  
L1 = [1, 2],  
L2 = [3, 4]
```

Y para el chequeo de estos:

```
 divide([1,2,3,4], 2, [1,2], [3,4]).  
true 1
```

Donde vemos que la salida es satisfactoria.


## Ejercicio 5

Para este ejercicio, implementamos el predicado `aplasta`. Este predicado se encarga de coger listas de listas, y 'aplastarlas' en una única lista. Para ello lo que hemos hecho ha sido establecer diferentes casos base, tal y como vemos en el código:

```
aplasta([], []).  
aplasta([X|L], LA) :-  
    aplasta(X, X1),  
    aplasta(L, L1),  
    concatena(X1, L1, LA).  
aplasta(X, [X]).
```

Tenemos por una parte el caso base, el cual nos dice que, dada una lista vacía se devuelve la misma, luego otro que dice que si obtenemos un elemento, se devuelve dicho elemento como lista, y por último, el caso general.

El caso general se encarga de obtener el primer elemento de una lista, hacer de dicho elemento una lista y concatenarlo a la lista donde estaba, obteniendo una nueva lista con todos los elementos concatenados.

```
 aplasta([1,2,[1,2,4],5,6], L1).  
L1 = [1, 2, 1, 2, 4, 5, 6]
```

## Ejercicio 6

Es este ejercicio se nos pide la factorización de números primos de un número  $N$  dado. Para ello seguimos el algoritmo enseñado en primaria para la factorización de números. Este consiste en ir desde 2 hacia arriba chequeando si dicho número es divisor de  $N$ , en caso de sí serlo, dividimos  $N$  por dicho número, y repetimos la operación, así hasta que se  $N$  sea 1.

Por tanto tenemos que evaluar dos casos, el caso en el que sea factor, y por lo tanto dicho número divida a  $N$ , y el caso en el que no, que entonces tendremos que buscar el siguiente factor.

Por otra parte, la manera en la que buscamos el siguiente factor es implementando la función 'next\_factor', tal que así:

```
next_factor(_, 2, 3).
next_factor(N, F, NF):-
    F<sqrt(N),
    NF is F+2.
next_factor(N, _, N):-
    N > 2.
```

Esta función tiene varios casos:

El primero nos dice que si el primer factor es 2, el siguiente factor será 3.

Si no se cumple la condición de mas arriba, entonces se irán generando factores a partir de 3 sumando de 2 en 2 (obteniendo así solo factores impares).

Y el 3<sup>er</sup> caso contempla el caso en el que el primer factor sea mayor que 2, ya que si fuese menor, es decir que el primer factor fuese 1, rompería toda la cadena de comprobaciones.

Por último, la función principal se divide en las dos partes que hemos mencionado previamente:

Que el factor que estamos comprobando sea, en efecto, factor de  $N$ .

Y que si por descarte, este no lo es, tendremos entonces que seguir buscando, llamando a la función previamente explicada, next\_factor.

Como vemos en la captura del código que adjuntamos.

Dado este código:

```

primos(1, [_]).
primos(N, [N]):-
    N < 3.

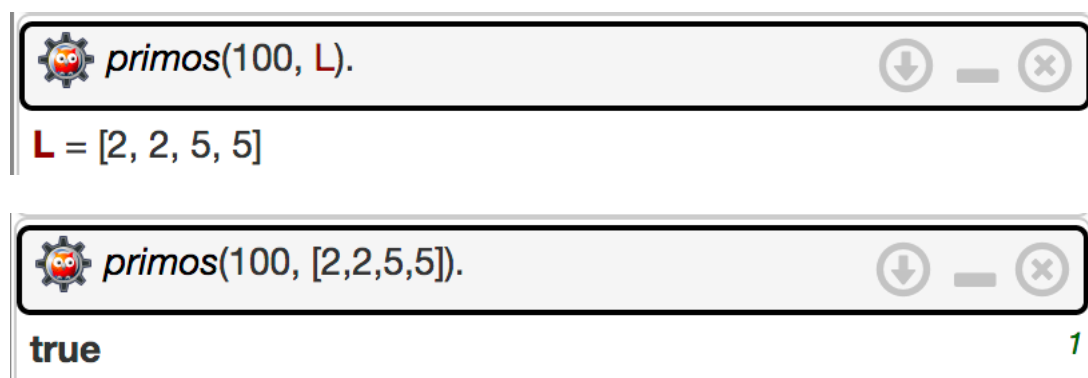
primos(N, L):-
    N > 1,
    primos_rec(N, 2, L).

primos_rec(1, _, []).
primos_rec(N, F, [F|L]):-
    0 is N mod F,
    A is N / F,
    primos_rec(A, F, L).

primos_rec(N, F, L):-
    next_factor(N, F, NF),
    primos_rec(N, NF, L).

```

Obtenemos la siguiente salida:



## Ejercicio 7

Para este ejercicio tenemos que agrupar las repeticiones en una lista. Se nos piden realizar 3 funciones, para la resolución de este ejercicio, `cod_primer`, `cod_all` y `run_length`.

Para la implementación de la primera función, `cod_primer` se pide que dado un valor y una lista, se agrupe en una lista los valores de la lista original que coincidan con el valor y en otra diferente, los que no.

Nuestra implementación de dicho predicado es la siguiente:

```

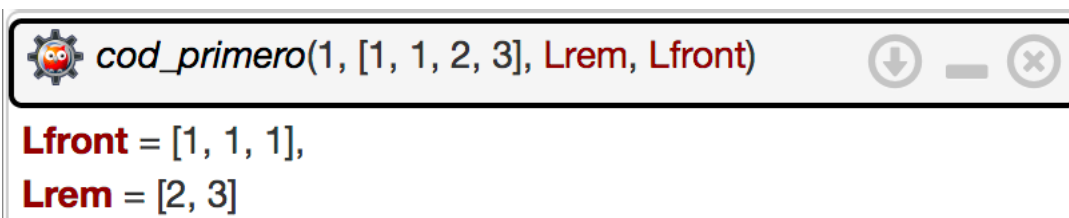
% caso base
cod_primer(X, [], [], [X|[]]).
% caso en el que los valores del array coincidan.
cod_primer(X, [X|L], Lrem, [X|Lfront]) :-
    cod_primer(X, L, Lrem, Lfront).

% caso en el que los valores del array NO coincidan.
cod_primer(X, [Y|L], [Y|Lrem], Lfront) :-
    (cod_primer(X, L, Lrem, Lfront)).

```

Donde contemplamos 3 casos, el caso de que la lista esté vacía, el caso en el que, en efecto el valor que se buscar coincida con el valor de la lista, y el caso en el que el valor no coincida.

Para comprobarlo:



The screenshot shows a Prolog interpreter window with the title bar "cod\_primer(1, [1, 1, 2, 3], Lrem, Lfront)". Below the title bar, the following text is displayed:

```

Lfront = [1, 1, 1],
Lrem = [2, 3]

```

Para la implementación de la segunda función, *cod\_all*, lo que se pide es usar el predicado anterior, a modo de agrupador, y esta función será la que recoja y ordene las agrupaciones de cada uno de los elementos que la componen.

```

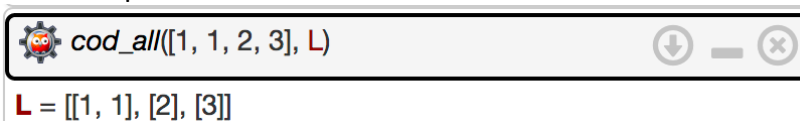
cod_all([], []).

cod_all([X|L], [Lfront|L1]) :-
    cod_primer(X, L, Lrem, Lfront),
    cod_all(Lrem, L1).

```

Como vemos en el código, volvemos a tener dos casos. El base se encarga de que si la lista es vacía se devuelva una lista vacía, y el segundo caso, el cual llama a la función que agrupa, sacando dos listas, la lista con las agrupaciones bajo ese valor, y los valores que quedan por agruparse, los cuales se pasarán a la llamada recursiva sobre dicha función.

Para comprobarlo:



The screenshot shows a Prolog interpreter window with the title bar "cod\_all([1, 1, 2, 3], L)". Below the title bar, the following text is displayed:

```

L = [[1, 1], [2], [3]]

```



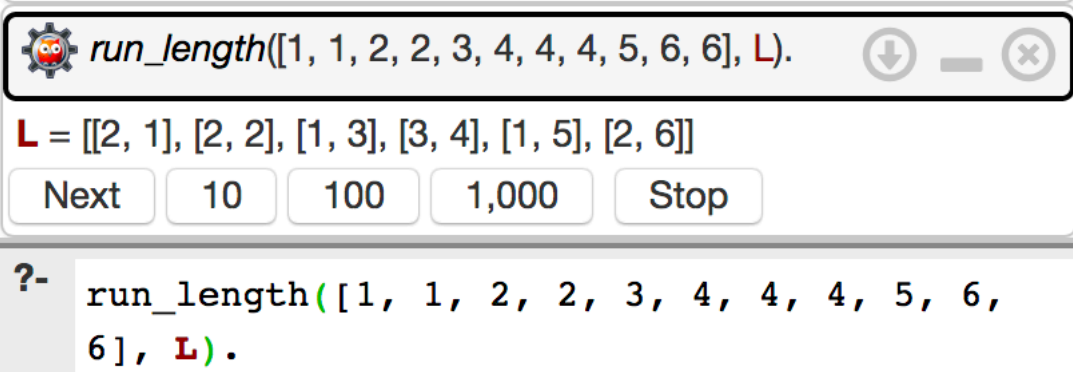
Y por último se implementa *run\_length*. Esta función se encarga de agrupar todas las listas que *cod\_all* genera y crear en la nueva lista que tiene los pares de veces repetidas/ valor que hay en la lista.

```
run_length([], []).  
run_length(L, L1):-  
    cod_all(L, Lres),  
    run_length_analyze(Lres, L1).
```

Para ello creamos una función que itera sobre las listas de listas que *cod\_all* genera, llamada *run\_length\_analyze*, la cual, saca una lista de la lista que nos da *cod\_all*, analiza su longitud, y crea una nueva lista en la que cada elemento es una lista que tiene un par veces repetida / valor.

```
run_length_analyze([], []).  
  
run_length_analyze([Lfront|Lres], [R|L1]):-  
    length(Lfront, N),  
    firstL(Lfront, Y),  
    concatena([N], [Y], R),  
    run_length_analyze(Lres, L1).
```

Obteniendo los resultados finales:



The screenshot shows a Prolog IDE interface. At the top, a command box contains the query `run_length([1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 6], L).` with a gear icon on the left and download, minus, and close icons on the right. Below the command box, the variable `L` is assigned the value `[[2, 1], [2, 2], [1, 3], [3, 4], [1, 5], [2, 6]]`. Underneath this, there are five buttons: "Next", "10", "100", "1,000", and "Stop". At the bottom, a console window shows the query `?- run_length([1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 6], L).` with a question mark icon on the left.