PRÁCTICA 3

INTELIGENCIA ARTIFICIAL

Javier Martinez Rubio javier.martinezrubio@estudiante.uam.es e357532
Jorge Santisteban Rivas jorge.santisteban@estudiante.uam.es e360104

EJERCICIO 1

Implemente un predicado duplica(L,L1), que es cierto si la lista L1 contiene los elementos de L duplicados

```
duplica([],[]).

duplica([A|L], [A,A|L1]) :- duplica(L,L1).
```

El caso base es cuando encontramos dos listas vacías. Para entrar en recursión el primer elemento de la primera lista debe ser igual al primer elemento y al segundo de la segunda lista.

Ejemplos

```
?-duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]).
true

?-duplica([1, 2, 3], [1, 1, 2, 3, 3]).
false

?-duplica([1, 2, 3], L1).
L1 = [1, 1, 2, 2, 3, 3]

?-duplica(L, [1, 2, 3]).
false
```

EJERCICIO 2

Implementa el predicado invierte(L, R) que se satisface cuando R contiene los elementos de L en orden inverso. Utiliza el predicado concatena/3 (/n: indica n argumentos):

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

que se satisface cuando su tercer argumento es el resultado de concatenar las dos listas que se dan como primer y segundo argumento.

```
%%%Concatena%%%

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

%%%Invierte%%%

invierte([],[]).
invierte([H|T],ListaInv):-
    invierte(T,InvT), concatena(InvT, [H], ListaInv).
```

Vamos a invertir la lista mediante recursión. Vamos a ir concatenando los elementos empezando por el final, dando lugar a la lista invertida.

Ejemplos

```
?-concatena([], [1, 2, 3], L).
L = [1, 2, 3]

?-concatena([1, 2, 3], [4, 5], L).
L = [1, 2, 3, 4, 5]

?-invierte([1, 2], L).
L = [2, 1]

?-invierte([], L).
L = []

?-invierte([1,2,3,4,5],L).
L = [5, 4, 3, 2, 1]
```

EJERCICIO 3

Implementar el predicado palindromo(L) que se satisface cuando L es una lista palíndroma, es decir, que se lee de la misma manera de izquierda a derecha y de derecha a izquierda.

```
%%%Concatena%%%

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

%%%Invierte%%%

invierte([],[]).
invierte([H|T],ListaInv):-
    invierte(T,InvT), concatena(InvT, [H], ListaInv).

%%%Palindromo%%%

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

Simplemente tenemos que,utilizando la funcion invierte, si L es la inversa de si misma. Si cumple eso será un palíndromo.

Ejemplos

```
?- palindromo([1, 2, 1]).
true

?- palindromo([1, 2, 1, 1]).
false

?- palindromo([S,O,B,O,R,N,O,S,S,O,N,R,O,B,O,S]).
true

?- palindromo(L).
L = []
L = [_1148]
```

```
L = [_1148, _1148]
L = [_1148, _1154, _1148]
L = [_1148, _1154, _1154, _1148]
L = [_1148, _1154, _1160, _1154, _1148]
```

Lo que hace es que cada vez que pulsamos el boton Next nos da la estructura de todas las posibles listas que son palindromos.

EJERCICIO 4

Implementar el predicado divide(L,N,L1,L2) que se satisface cuando la lista L1 contiene los primeros N elementos de L y L2 contiene el resto.

```
divide(L,N, L1, L2) :-
    length(L1, N),
    append(L1, L2, L).
```

En primer lugar utilizamos la función length/2 la cual define el tamaño de L1 a N. Una vez con una lista con un tamaño determinado utilizamos la función append/3. Esta devuelve True si L1 concatenado con L2 es L. En este caso como ya hemos definido el tamaño de L1 con la función length/2 para que se cumpla en L1 tendremos los N primeros elementos de L y en L2 el resto.

Ejemplos

```
?-divide([1],0,L1,L2).
L1 = [],
L2 = [1]

?-divide([1,2],1,L1,L2).
L1 = [1],
L2 = [2]

?-divide([1, 2, 3, 4, 5], 3, L1, L2).
L1 = [1, 2, 3],
L2 = [4, 5]

?-divide(L, 3, [1, 2, 3], [4, 5, 6]).
L = [1, 2, 3, 4, 5, 6]
```

EJERCICIO 5

Implementar el predicado aplasta(L, L1) que se satisface cuando la lista L1 es una versión “aplastada” de la lista L, es decir, si uno de los elementos de L es una lista, esta será remplazada por sus elemento, y así sucesivamente.

```
aplasta([], []):- !.
aplasta([L|Ls], L1) :-
    !,
    aplasta(L, First),
    aplasta(Ls, Rest),
    append(First, Rest, L1).
aplasta(L, [L]).
```

El funcionamiento de la funcion aplasta/2 es el siguiente: iremos elemento a elemento de la lista eliminando las sublistas para al final llegar a una lista única. Utilizaremos un método recursivo: mediante la función append concatenaremos dos listas para llegar a la lista final. Estas dos listas son el resultado de “aplastar” el first y el rest de la lista original. Iremos extrayendo elemento a elemento de las sublistas para al final insertarlos en la misma. Para ello tendremos dos casos base dependiendo de lo que reciba la funcion aplasta/2: si es solo un elemento devolvemos una lista con ese elemento, mientras que si es la lista vacía devolvemos la lista vacía y cortamos el backtracking. Por recursión iremos insertando uno a uno y en orden los elementos en la lista final.

Ejemplos

```
?-aplasta([[1,2]],L).
L = [1, 2]

?-aplasta([1,[2]],L).
L = [1, 2]

?-aplasta([[[[[[[[[[1]]]]]]]]]],2], L).
L = [1, 2]

?-aplasta([[[[[[[[[[1]]]]]]]]]],2], [[1],2]).
false

?-aplasta([[[[[[[[[[1]]]]]]]]]],2], [1,2]).
true

?-aplasta([1, [2, [3, 4], 5], [6, 7]], L).
L = [1, 2, 3, 4, 5, 6, 7]

?-aplasta(L,[1, 2, 3]).
false
```

La última query devuelve false ya que cuando la lista no esta definida va al caso base y devuelve la lista vacía (tanto para el first como para el rest). Por ello cuando aplicamos append devuelve falso porque concatenar dos listas vacías no es nuestra lista “aplastada”. Además es imposible devolver todas las posibles listas primigenias que dieron lugar a nuestra lista aplastada, por lo que devolviendo false es una manera rápida y eficiente de indicarlo.

EJERCICIO 6

Implementar el predicado primos(N, L) que se satisface cuando la lista L contiene los factores primos del número N. Para esta función puede ser útil crear el predicado next_factor(N,F,NF), que genera los factores que vamos a probar. Dado el número n y el último factor generado, el predicado se satisface si:
1. F=2 y NF=3, o
2. F<sqrt(N) y NF=F+2
(no hace falta probar los número pares, excepto 2).

```
%%%next_factor%%%

next_factor(_,2,3):- !.
next_factor(N,F,NF):-
    F*F < N,!,NF is F + 2.
next_factor(N,_,N).

%%%Primos/2%%%
```

```
primos(N,L):-
    N > 0,
    primos(N,L,2).

%%%Primos/3%%%

primos(1,[],_):- !.
primos(N, [Factor|Resto], Factor):-
    Div is N // Factor,
    N =:= Div*Factor,
    !,
    primos(Div, Resto, Factor).

primos(N,Resto, Factor):-
    N > Factor,
    next_factor(N,Factor, NFactor),
    primos(N,Resto, NFactor).
```

En este ejercicio definiremos tres funciones: next_factor/3 , primos/2 y primos/3. Las dos primeras son auxiliares para la principal, primos/2.

- Next_factor/3 es una función que a partir de un número y su último posible factor nos da el siguiente posible factor. Esta función es útil para ir probando posibles factores de una manera más eficiente ya que, por ejemplo, salvo el 2 ningún numero par puede ser factor primo de un número. Para poder obtener el siguiente factor se tiene que satisfacer dos predicados: Factor=2 y NextFactor=3, o Factor<sqrt(Numero) (equivalente a Factor * Factor < Numero) y NextFactor=Factor+2.
- Primos/2 es la función principal que simplemente comprueba que N sea positivo y hace una llamada a primos/3 introduciendo como primer posible factor el 2.
- Primos/3 es una función auxiliar pero es la que hace todo el trabajo. En esta función vamos añadiendo a la lista todos los factores primos ordenados. Para comprobar que es factor utilizamos // (si tenemos A // B nos devolverá el suelo del cociente). De esta manera si el número es igual al factor por el resultado de // (Div) tendremos un factor, añadiendolo a la lista y aplicando la funcion primos/3 a Div para obtener el resto de factores. Si resulta que un número no es factor, hallaremos su siguiente factor (utilizando next_factor) y volveremos a aplicar primos/3 hasta que lleguemos al caso base donde N=1,devolviendo la lista vacía y finalizando la búsqueda de factores primos.

Ejemplos

```
?-primos(2,L).
L = [2]

?-primos(6,L).
L = [2, 3]

?-primos(6,[2,3]).
true

?-primos(6,[3,2]).
false

?-primos(100,L).
L = [2, 2, 5, 5]

?-primos(450,L).
```

```
L = [2, 3, 3, 5, 5]
```

EJERCICIO 7

Codificación run-length de una lista: si la lista contiene N términos consecutivos iguales a X, esto son codificados como un par [N, X].

Ejemplo:

[1, 1, 2, 3, 3, 4, 4, 4, 5, 5, 5, 5, 6, 7, 7]
se codifica como
[[2, 1], [1,2], [2, 3], [3, 4], [4, 5], [1, 6], [2, 7]]

EJERCICIO 7.1

Empezamos con el predicado cod_primero(X, L, Lrem, Lfront). Lfront contiene todas las copias de X que se encuentran al comienzo de L, incluso X; Lrem es la lista de elementos que quedan:

```
%%%7.1%%%

cod_primero(X,[],[],[X]).
cod_primero(X,[X|Ys], Lrem, [X|Xs]):-
    cod_primero(X,Ys,Lrem,Xs).
cod_primero(X, [Y|Ys], [Y|Ys], [X]):-
    dif(X,Y).
```

Esta función va elemento a elemento de L comprobando que el *first* sea igual a X. Si ocurre esto llamamos de manera recursiva a cod_primero pero con el *rest* de la lista. Para que el programa acabe solo hay dos posibles caso: que encontremos un elemento distinto a X en la lista o que los hayamos explorado todos. Para el primer caso comprobamos con la función dif/2 que sean distintos, devolviendo como Lrem el resto de la lista y como Lfront X en forma de lista. De esta manera iremos insertando de manera recursiva X en Lfront n+1 veces, siendo n el número de veces que aparece X en L. Para el segundo caso Lrem simplemente será una lista vacía y con Lfront actuaremos de la misma manera que en el primer caso.

Ejemplos

```
?-cod_primero(1, [1, 1], Lrem, Lfront).
Lfront = [1, 1, 1],
Lrem = []
false

?-cod_primero(1, [1, 1, 2, 3], Lrem, Lfront).
Lfront = [1, 1, 1],
Lrem = [2, 3]
false

?-cod_primero(1, [2, 3, 4], Lrem, Lfront).
Lfront = [1],
Lrem = [2, 3, 4]
```

EJERCICIO 7.2

El predicado cod_all(L, L1) aplica el predicado cod_primero/4 a toda la lista L.

```
%%%7.1%%%

cod_primero(X,[],[],[X]).
cod_primero(X,[X|Ys], Lrem, [X|Xs]):-
    cod_primero(X,Ys,Lrem,Xs).
cod_primero(X, [Y|Ys], [Y|Ys], [X]):-
    dif(X,Y).

%%%7.2%%%

cod_all([],[]).
cod_all([X|Resto], [Y|Lfront]):-
        cod_primero(X,Resto,Lrem,Y),
        cod_all(Lrem, Lfront).
```

Esta función agrupa en sublistas todos los elementos del mismo tipo teniendo en cuenta el orden. Para ello iremos obteniendo estas listas para cada uno de los elementos diferentes (y sin orden). Aplicamos cod_primero/4 para obtenerla y de manera recursiva aplicamos cod_all/2 a la lista de elementos restantes. De esta manera obtendremos cada una de las sublistas hasta llegar al caso base donde la lista sea vacía, devolviendo a su vez esta misma. De forma recursiva añadiremos las sublistas a la lista final.

Ejemplos

```
?-cod_all([1],L).
L = [[1]]
false

?-cod_all([1,2],L).
L = [[1], [2]]
false

?-cod_all([1,2,2,3], L).
L = [[1], [2, 2], [3]]
false

?-cod_all([1,2,3,3,1],L).
L = [[1], [2], [3, 3], [1]]
false

?-cod_all([1, 1, 2, 3, 3, 3, 3], L).
L = [[1, 1], [2], [3, 3, 3, 3]]
false
```

EJERCICIO 7.3

El predicado run_length(L, L1) aplica el predicado pack y luego transforma cada una de las listas en la codificación run-length:

```
%%%7.1%%%

cod_primero(X,[],[],[X]).
cod_primero(X,[X|Ys], Lrem, [X|Xs]):-
```

```
    cod_primero(X,Ys,Lrem,Xs).
cod_primero(X, [Y|Ys], [Y|Ys], [X]):-
    dif(X,Y).

%%%7.2%%%

cod_all([],[]).
cod_all([X|Resto], [Y|Lfront]):-
        cod_primero(X,Resto,Lrem,Y),
        cod_all(Lrem, Lfront).

%%%7.3%%%

run_length(L,L1):-
    cod_all(L,LAux),
    transformar(LAux,L1).

transformar([],[]).
transformar([[X|Xs]|Ys], [[Len,X] | Resto]):-
    length([X|Xs],Len),
    transformar(Ys,Resto).
```

Para este ejercicio necesitamos dos funciones: run_length/2 y una función auxiliar a la que hemos llamado transformar/2.

En run_length/2 en primer lugar hemos transformado la lista principal L en una lista de sublistas con elementos del mismo tipo. Esa es la lista que tendremos que transformar. Para ello iremos sublista a sublista pasandola a la codificación run-length de manera recursiva. Para obtener el número de elementos de la sublista utilizamos la función length/2. Una vez obtenida la longitud insertaremos en la lista final una sublista con dos elementos: el primero la longitud y segundo el elemento que se repite. Iremos repitiendo el mismo procedimiento de manera recursiva hasta llegar al caso base cuando la lista a transformar sea la lista vacía, finalizando la codificación run-length.

Ejemplos

```
?-run_length([a],L).
L = [[1, 1]]
false

?-run_length([a,b],L).
L = [[1, a], [1, b]]
false

?-run_length([a,b], [[1, a], [2, b]]).
false

?-run_length([a,b], [[1, a], [1, b]]).
true
false

?-run_length([a, a, a, c, c, e, e, f, g, i, i, i, i, i, i, l, l, n, n, r, t, t],L).
L = [[3, a], [2, c], [2, e], [1, f], [1, g], [6, i], [2, l], [2, n], [1, r], [2, t]]
false
```

EJERCICIO 8

Un árbol de Huffman es una estructura usada para codificar y comprimir información. El objetivo es codificar un texto formado por símbolos utilizando códigos binarios. El esquema de codificación utiliza la frecuencia que aparece cada uno de los símbolos en un texto para minimizar el número de bits necesario para almacenar la información. En concreto,los símbolos más frecuentes en el texto están codificados con bloques de bits más cortos,. A modo de ejemplo, la cadena en ASCII:

AAAADAAACCCAAAAAAAABAAAAABAAA

usa para almacenar cada caracter 1 Byte. Sin embargo, podemos reducir el espacio ocupado por la cadena. De forma simple podemos expresarlo como:

```
A   0
B  111
C  10
D  110
```

De este modo, utilizando los códigos ASCII por los códigos de bloques de bits especificados en esta tabla, es posible reducir la longitud de la cadena sin perder información. Es importante observar que ningún código es prefijo de otro código (por ejemplo, solo el código del carácter 'A', el más frecuente en la cadena,empieza por '0'. De forma análoga, solo el código del carácter 'C' comienza por '10', etc.), por lo que no hay ambigüedad al descodificar.

En esta práctica implementaremos una versión simplificada de este tipo de estructura. Para ello insertaremos el elemento con mayor probabilidad a la izquierda del nodo raíz, y seguiremos con este mismo proceso expandiendo el árbol por la derecha. Cuando solo queden 2 elementos dejaremos de expandir el árbol y terminaremos.

Implementa el predicado build_tree(List, Tree) que transforma una lista de pares de elementos ordenados en una versión simplificada de un árbol de Huffman. Para representar árboles usaremos las funciones tree(Info, Left, Right) y nil. También usaremos el predicado concatena/3. Los nodos hoja del árbol se corresponden con los elementos de la lista ordenada y almacenan el elemento en el campo Info, mientras que los nodos intermedios siempre almacenan un 1.

```
%%%Build_tree%%%

build_tree([],nil).
build_tree([X-_], tree(X,nil,nil)):- !.
build_tree([X-_|Resto], T):-
    build_tree(Resto,S),
    T = tree(1, tree(X,nil,nil), S).
```

Esta función construye el árbol de Huffman a partir de una lista con elementos cuyo formato es Identificador-NºdeVecesQueAparece. A partir de la lista tendremos que crear el árbol. Utilizaremos un método recursivo donde iremos insertando elemento a elemento en el árbol. La primera vez que entra define el nodo raíz como 1, la parte de la izquierda como un nodo hoja con el primer elemento de la lista y la parte de la derecha como el árbol resultante de una llamada recursiva al resto de la lista. De esta manera iremos creando el árbol de Huffman hasta llegar al caso base donde solo queda un elemento en la lista,creando el nodo hoja y finalizando el árbol.

Ejemplos

```
?-build_tree([p-0, a-6, g-7, p-9, t-2, 9-99], X).
X = tree(1, tree(p, nil, nil), tree(1, tree(a, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree
```

```
(p, nil, nil), tree(1, tree(t, nil, nil), tree(9, nil, nil))))))

?-build_tree([p-55, a-6, g-7, p-9, t-2, 9-99], X).
X = tree(1, tree(p, nil, nil), tree(1, tree(a, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree
(p, nil, nil), tree(1, tree(t, nil, nil), tree(9, nil, nil))))))

?-build_tree([p-55, a-6, g-2, p-1], X).
X = tree(1, tree(p, nil, nil), tree(1, tree(a, nil, nil), tree(1, tree(g, nil, nil), tree(p, nil,
 nil))))

?-build_tree([a-11, b-6, c-2, d-1], X).
X = tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, nil), tree(d, nil,
 nil))))
```

EJERCICIO 8.1

Implementar el predicado encode_elem(X1, X2, Tree) que codifica el elemento (X1) en (X2) basándose en la estructura del árbol (Tree).

```
%%%Concatena%%%

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

%%%Build_tree%%%

build_tree([],nil).
build_tree([X-_], tree(X,nil,nil)):- !.
build_tree([X-_|Resto], T):-
    build_tree(Resto,S),
    T = tree(1, tree(X,nil,nil), S).

%%%Encode_elem%%%

encode_elem(X, [0], tree(1, tree(X, nil, nil), _)).
encode_elem(X,[0], tree(X, nil, nil)).
encode_elem(X, Cod, tree(1,_,Resto)):-
    encode_elem(X, S, Resto),
    concatena([1], S, Cod).
```

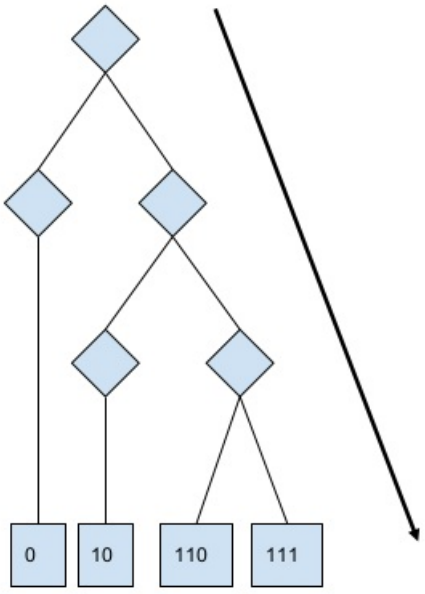
Esta función codifica los elementos del árbol según su posición en el mismo. Iremos elemento a elemento del árbol comprobando si se trata del elemento a codificar. Si no se trata del elemento quiere decir que debemos bajar un nivel en el árbol, por lo que añadiremos un 1 a la codificación y concateramos ese resultado con la llamada recursiva de encode_elem/3 a la parte derecha del árbol. Se pueden dar dos situaciones a la hora de identificar un elemento en el árbol. Si no es el “último” elemento (elemento más a la derecha) devolveremos una lista con el 0, que se concatenara con el número de 1’s que haya previamente. Si es el último elemento simplemente devolvemos la lista vacía, obteniendo tantos 1’s como el número de niveles que hayamos descendido. Además, si el elemento a codificar no se encuentra en el árbol la función devuelve *false*.

Ejemplos

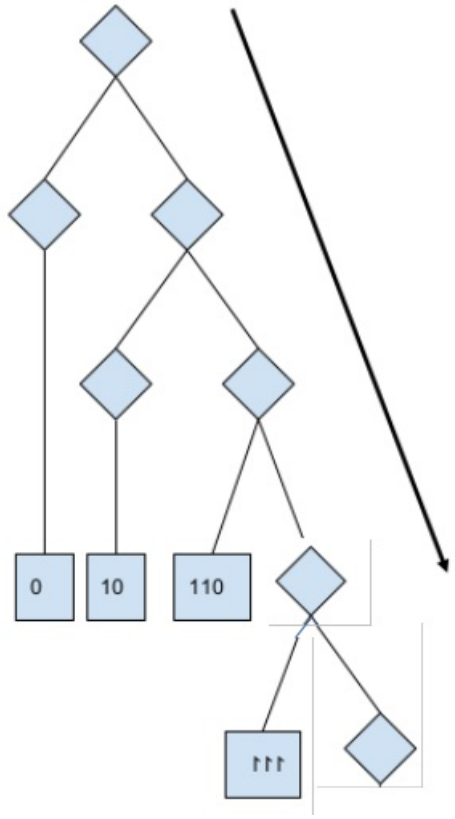
```
?-encode_elem(a, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, ni
l), tree(d, nil, nil))))).
X = [0]
false
```

```
?-encode_elem(b, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, ni
l), tree(d, nil, nil))))).
X = [1, 0]
false

?-encode_elem(c, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, ni
l), tree(d, nil, nil))))).
X = [1, 1, 0]
false

?-encode_elem(d, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, ni
l), tree(d, nil, nil))))).
X = [1, 1, 1, 0]
false

?-
encode_elem(e, X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, nil)
, tree(d, nil, nil))))).
false
```

Nos gustaría puntualizar una modificación respecto a los ejemplos dados en la memoria a la hora de codificar. En el ejemplo, al elemento más a la derecha del árbol lo codifica con todo 1's. Sin embargo hemos implementado una codificación que siempre termina en 0 y que el número de 1's indica la profundidad del elemento en cuestión. Este problema surge a la hora de crear el árbol cuando este tiene un único elemento. En ese caso el nodo raíz debe ser ese mismo elemento, sino la estructura del árbol cambiaría de esta:



A esta:



En un esfuerzo de mantener la estructura del árbol lo más intacta posible (salvo en el caso del ya explicado árbol con único elemento) hemos decidido que la codificación sea así y no como la indicada en los ejemplos. Además en el último apartado los ejemplos muestran la codificación que acabamos de definir y no la presentada en este apartado.

EJERCICIO 8.2

Implementar el predicado encode_list(L1, L2, Tree) que codifica la lista (L1) en (L2) siguiendo la estructura del árbol (Tree).

```
%%%Concatena%%%

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

%%%Build_tree%%%

build_tree([],nil).
build_tree([X-_], tree(X,nil,nil)):- !.
build_tree([X-_|Resto], T):-
    build_tree(Resto,S),
    T = tree(1, tree(X,nil,nil), S).

%%%Encode_elem%%%

encode_elem(X, [0], tree(1, tree(X, nil, nil), _)).
encode_elem(X,[0], tree(X, nil, nil)).
encode_elem(X, Cod, tree(1,_,Resto)):-
    encode_elem(X, S, Resto),
```

```
    concatena([1], S, Cod).

%%%Encode_list%%%

encode_list([],[],_).
encode_list([X], CodList, Tree):-
    encode_elem(X, Cod1, Tree),
    concatena([Cod1],[],CodList),!.
encode_list([X|Resto], CodList, Tree):-
    encode_elem(X, Cod1, Tree),
    encode_list(Resto, CodResto, Tree),
    concatena([Cod1], CodResto, CodList).
```

Esta función codifica todos los elementos de una lista uno a uno de manera recursiva. Codifica el primer elemento de la lista utilizando encode_elem/3 y lo concatena al resultado de aplicar encode_list/3 al resto de la lista. El caso base llega cuando solo hay un elemento en la lista, donde simplemente devolvemos la codificación de ese elemento. También definimos el caso en el que recibamos una lista vacía, devolviendo simplemente la lista vacía como resultado. Si el elemento no se encuentra en el árbol encode_elem/3 devuelve false, por lo que encode_list/3 devolverá automaticamente false.

Ejemplos

```
?-encode_list([], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, n
il), tree(d, nil, nil))))).
X = []

?-encode_list([a,a], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1,
tree(c, nil, nil), tree(d, nil, nil))))).
X = [[0], [0]]
false

?-encode_list([a,d,a], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil),
tree(1, tree(c, nil, nil), tree(d, nil, nil))))).
X = [[0], [1, 1, 1, 0], [0]]
false

?-encode_list([a,d,a,q], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil),
tree(1, tree(c, nil, nil), tree(d, nil, nil))))).
false
```

EJERCICIO 8.3

Implementar el predicado encode(L1, L2) que codifica la lista (L1) en (L2). Para ello haced uso del predicado dictionary:

dictionary([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).

```
%%%Dictionary%%%

dictionary([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).

%%%Funciones auxiliares Ejercicio 7%%%

cod_primero(X,[],[],[X]).
cod_primero(X,[X|Ys], Lrem, [X|Xs]):-
    cod_primero(X,Ys,Lrem,Xs).
```

```
cod_primero(X, [Y|Ys], [Y|Ys], [X]):-
    dif(X,Y).

%%%7.2%%%

cod_all([],[]).
cod_all([X|Resto], [Y|Lfront]):-
        cod_primero(X,Resto,Lrem,Y),
        cod_all(Lrem, Lfront).

%%%7.3%%%

run_length(L,L1):-
    cod_all(L,LAux),
    transformar(LAux,L1).
transformar([],[]).
transformar([[X|Xs]|Ys], [[Len,X] | Resto]):-
    length([X|Xs],Len),
    transformar(Ys,Resto).

%%%Validar_txt%%%

validarTxt([]):- !.
validarTxt([X|Resto]):-
    dictionary(D),
    member(X,D),
    validarTxt(Resto).

%%%Formatear%%%

formatear([],[]):- !.
formatear([[A,B] | Resto], [[B-A]|FormatL]):-
    formatear(Resto, FormatL).

%%%InsertarElem%%%

insertarElem([A-B],[],[A-B]).
insertarElem([A-B],[C-D | Resto],[A-B, C-D | Resto]):-
    B >= D.
insertarElem([A-B],[C-D | Resto],[C-D | E]):-
    insertarElem([A-B],Resto,E),
    B < D.

%%%OrdenarLista%%%

ordenarLista([],[]).
ordenarLista([[A-B] | Resto], OrdLista):-
    ordenarLista(Resto, RecLista),
    insertarElem([A-B], RecLista, OrdLista).


%%%Concatena%%%

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

%%%Build_tree%%%

build_tree([],nil).
```

```
build_tree([X-_], tree(X,nil,nil)):- !.
build_tree([X-_|Resto], T):-
    build_tree(Resto,S),
    T = tree(1, tree(X,nil,nil), S).

%%%Encode_elem%%%

encode_elem(X, [0], tree(1, tree(X, nil, nil), _)).
encode_elem(X,[0], tree(X, nil, nil)).
encode_elem(X, Cod, tree(1,_,Resto)):-
    encode_elem(X, S, Resto),
    concatena([1], S, Cod).

%%%Encode list%%%

encode_list([],[],_).
encode_list([X], CodList, Tree):-
    encode_elem(X, Cod1, Tree),
    concatena([Cod1],[],CodList),!.
encode_list([X|Resto], CodList, Tree):-
    encode_elem(X, Cod1, Tree),
    encode_list(Resto, CodResto, Tree),
    concatena([Cod1], CodResto, CodList).

%%%Encode%%%

encode(L1,L2):-
    validarTxt(L1),
    sort(0,@=<,L1,SortL1),
    run_length(SortL1, RunL1),
    formatear(RunL1,FormatL1),
    ordenarLista(FormatL1, OrdenadaL1),
    build_tree(OrdenadaL1, TreeL1),
    encode_list(L1,L2,TreeL1).
```

Esta función codifica una lista de caracteres siguiendo el siguiente orden:

1º Comprobamos si los elementos de la lista a codificar están en el diccionario. Para ello utilizamos la función auxiliar validarTxt/1, la cual comprueba de manera recursiva si cada uno de los elementos de la lista están en el diccionario.

2º Ordenamos esa lista por orden alfabético utilizando la función ya definida en prolog sort/4, la cual nos devuelve los elementos ya ordenados.

3º En el paso 2 ordenamos la lista para poder aplicarle la codificación run-length, obteniendo la frecuencia de cada uno de los elementos aplicando run_length/2 definida en el ejercicio 7.

4º En este caso tenemos que definir una función que cambie el formato de los elementos de la lista resultado de run_length. Esto lo hacemos para poder utilizar las funciones que hemos creado en el ejercicio 8. Tendremos que pasar, por ejemplo, de [3,a] a a-3. Para ello hemos definido la función formatear/2 la cual va elemento a elemento de la lista formateandolos. Formatea el primer elemento de la lista y concatena ese resultado con el resultante de formatear el resto de la lista. Así ira recursivamente formateando todos los elementos hasta llegar al caso base, la lista vacía, devolviendo esta a su vez.

5º Una vez formateada la lista tendremos que ordenarla según la frecuencia de los elementos de esta. Para ello hemos definido ordenarLista/2. Esta se utiliza una función auxiliar denominada insertarElem/2. Su funcionamiento es el siguiente: llegaremos hasta el último elemento de la lista y desde el final iremos

insertando uno a uno en orden descendente y manteniendo el orden alfabético. Con ordenarLista/2 llegamos al final de la lista. Valiéndonos de insertarElem/2 iremos insertando los elementos anteriores. Si el segundo paramétro del elemento a insertar es mayor que el primero de la lista lo insertamos en el primer lugar. En otro caso, llamamos a la función de manera recursiva con el resto de la lista. De esta manera conseguiremos una lista con los elementos ordenados de manera descendente y en orden alfabético.

6º Tras ordenar la lista simplemente tendremos que aplicar las funciones build_tree/2 y encode_list/3 definidas previamente para obtener la codificación de la lista que queríamos.

Ejemplos

```
?-encode([i],X).
X = [[0]]
false

?-encode([i,a],X).
X = [[1, 0], [0]]
false

?-encode([i,n,t,e,l,i,g,e,n,c,i,a,a,r,t,i,f,i,c,i,a,l],X).
X = [[0], [1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1, 0], [0], [1, 1, 1
, 1, 1, 1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1, 1, 0], [1, 1, 0], [0], [1, 0], [1, 0], [1, 1, 1,
1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 0], [0], [1, 1, 1, 1, 1, 1, 1, 0], [0], [1, 1, 0], [0],
[1, 0], [1, 1, 1, 1, 0]]
false

?-encode([2,3],X).
false
```

Nos gustaría puntualizar un hecho que nos han sorprendido a la hora de realizar la funcion encode/2 en comparación con los ejemplos del enunciado de la práctica. A la hora de ordenar la lista con ordenarLista/2 nosotros los elementos con la misma frecuencia los ordenados en orden alfabético. Por ejemplo, en el caso de:

encode([i,n,t,e,l,i,g,e,n,c,i,a,a,r,t,i,f,i,c,i,a,l],X).

la lista ordenada que obtenemos es:

X = [i-6, a-3, c-2, e-2, l-2, n-2, t-2, f-1, g-1, r-1]

por lo que las codificaciones serían las resultantes:

```
i 0
a 10
c 110
e 1110
l 11110
n 111110
t 1111110
f 11111110
g 111111110
r 1111111110
```

y no:

X = [i-6, a-3, t-2, n-2, l-2, e-2, c-2, r-1, g-1, f-1]

```
i 0
a 10
t 110
n 1110
l 11110
e 111110
c 1111110
r 11111110
g 111111110
f 1111111110
```

que es la codificación que muestra el ejemplo en el enunciado de la práctica. Creemos que un orden descendente y, en caso de igualdad, alfabético, es más lógico a la hora de codificar, estando ahí la diferencia con el ejemplo original.