Javier Delgado del Cerro y Javier López Cano - Pareja 2

Práctica 3

En esta práctica, como el código de las funciones es muy simple y corto, lo incluimos directamente en cada apartado.

Ejercicio 1

Batería de ejemplos

En este caso, al ser la función tan simple, la batería de ejemplos es bastante reducida.

```
?- duplica([1], [1, 1]). %%% true
?- duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]). %%% true
?- duplica([1, 2, 3], [1, 1, 2, 2, 3, 3, 4, 4]). %%% false
?- duplica([1, 2, 3], X). %%% X = [1, 1, 2, 2, 3, 3]
?- duplica(L, [1, 1, 2, 2, 3, 3]). %%% L = [1, 2, 3]
?- duplica(L, [1, 1, 2, 2, 3]). %%% false
```

Comprobamos que todos los ejemplos funcionan correctamente.

Pseudo-código

```
duplica(L1, L2):
    vacia(L1) y vacia(L2):
        devolver true
    si L1[0] == L2[0] == L2[1]:
        devolver duplica(resto(L1), resto(resto(L2)))
    si no:
    devolver false

```

Código

El código de la función implementada sería por tanto el siguiente:

```
duplica([], []).
duplica([X|T], [X, X|L]):-duplica(T, L).
```

Ejercicio 2

Batería de ejemplos

Incluimos los ejemplos de prueba para las dos funciones que hemos desarrollado en este apartado:

```
?- concatena([], [], []). %%% true
?- concatena([1], [], [1]). %%% true
```

```
?- concatena([], [1], [1]). %%% true
?- concatena([1], [], []). %%% false
?- concatena([], [1], []). %%% false
?- concatena([1, 2], [3, 4], [1, 2, 3, 4]). %%% true
?- concatena([1, 2], [3, 4], [1, 2, 3]). %%% false
?- concatena([1, 2], [3, 4], X). %%% X = [1, 2, 3, 4]

?- invierte([], []). %%% true
?- invierte([1], [1]). %%% true
?- invierte([1, 2], [2, 1]). %%% true
?- invierte([1, 2], [1, 2]). %%% false
?- invierte([1, 2, 3], X). %%% X = [3, 2, 1]
```

De nuevo, tras ejecutar estos ejemplos comprobamos que la función es correcta.

Pseudo-código

Incluimos simplemente el pseudo-código de la función *invierte*, pues el código de *concatena* se nos da en el enunciado de la práctica.

```
invierte(L1, L2):
        si vacio(L1) y vacio(L2):
                devolver true
        si no:
                H = primer_elem(L1)
                T = resto(L1)
                devolver (R tal que satisface invierte(T, R)) y concatena(R, lista(H), L2)
```

Código

El código resultante de la función es el siguiente.

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

invierte([], []).
invierte([H|T],L):-invierte(T,R),concatena(R,[H],L).
```

Al igual que en el apartado anterior, no hay ningún comentario especial sobre la implementación, pues lo único difícil fue comprender el funcionamiento inicial de Prolog.

Ejercicio 3

Batería de ejemplos

```
?- palindromo([]). %%% true
?- palindromo([1]). %%% true
?- palindromo([1, 1]). %%% true
?- palindromo([1, 2]). %%% false
?- palindromo([1, 2, 3, 2, 1]). %%% true
?- palindromo([1, 2, 3, 3, 1]). %%% false
```

En el caso de llamar a la función con una variable no instanciada, genera la lista vacía, pero pulsando los botones *Next*, *10*, *100*... que aparecen, se muestran nuevos ejemplos con listas que incluyen posiciones de memoria, de forma que las listas son palíndromos. En nuestro ejemplo, generando 10 más:

```
X = []
X = [_1164]
X = [_1192, _1192]
X = [_1226, _1232, _1226]
X = [_1266, _1272, _1272, _1266]
X = [_1312, _1318, _1324, _1318, _1312]
X = [_1364, _1370, _1376, _1376, _1370, _1364]
X = [_1422, _1428, _1434, _1440, _1434, _1428, _1422]
X = [_1486, _1492, _1498, _1504, _1504, _1498, _1492, _1486]
X = [_832, _838, _844, _850, _856, _850, _844, _838, _832]
X = [_908, _914, _920, _926, _932, _932, _926, _920, _914, _908]
```

Código

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

Comentarios sobre la implementación

En el caso de esta función, su implementación se basa simplemente en llamar a la función *invierte* desarrollada en el apartado anterior, pues internamente realizan la misma función. Por esta razón, consideramos que no es necesario realizar el pseudocódigo de la función, pues no aporta nada y empeora la lectura del texto.

Ejercicio 4

Batería de ejemplos

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

Pseudo-código

El pseudo-código de la función a desarrollar sería por tanto el siguiente:

```
divide(L, N, L1, L2):
    X = primer_elem(L)
    T = resto(L)
    si (L1 == lista(X)) y (L2 == T) y (N == 1):
        devolver true
    si no:
        devolver (existe H tal que satisface divide(T, N-1, H, L2)) y
concatena(lista(X), H, L1)
```

Código

```
divide([X|T],1,[X],T).
divide([X|T],N,L1,L2):-
    N2 is N-1,
    divide(T, N2, H, L2),
    concatena([X], H, L1).
```

Una vez programado, vemos que efectivamente funciona de la manera esperada, comprobando si L1 contiene los N primeros elementos de L, y L2 el resto. Además, por la lógica de *backtraking* empleada por Prolog, nos permite inferir L1 y/o L2 dados L y N, sin embargo, por la forma en la que está programado, no permite inferir N. Como no es algo pedido en el enunciado, consideramos que no es un problema.

Ejercicio 5

Batería de ejemplos

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

En el caso de ponerlo al revés, como *aplasta(R, [a, b, c, d])*, la función devuelve *false*, pues emplea el primer elemento de R, que no está inicializado en este caso.

Pseudo-código

```
aplasta(L, Res):
        si vacio(L) y vacio(Res):
                devolver true
        si Res == lista(L):
                devolver true
        si no:
                L1 = lista aplastada de primer_elem(L)
                L2 = lista aplastada de resto(L)
                devolver concatenar(L1, L2) == Res
```

Código

El código de nuestra función sería entonces:

```
aplasta([], []) :- !.
aplasta([X|T], Res) :-
    !,
    aplasta(X, L1),
    aplasta(T, L2),
    append(L1, L2, Res).
aplasta(L, [L]).
```

Ejercicio 6

Batería de ejemplos

Incluimos ejemplos para probar, en primer lugar, la función auxiliar *next_factor*, y posteriormente, la función principal *primos*.

```
next_factor(5, 2, 3) %%% true
next_factor(5, 2, NF) %%% NF = 3
next_factor(2, 5, 3) %%% false porque 5>sqrt(2)
next_factor(2, 5, NF) %%% false porque 5>sqrt(2)
next_factor(25, 3, NF) %%% NF = 5
next_factor(25, 5, NF) %%% NF = 25. El último factor es él mismo, por si el número es primo.

primos(1, []) %%% true, el caso base
primos(4, [2, 2]) %%% true
primos(4, [2, 3]) %%% false
primos(63, [3, 3, 7]) %%% true
primos(63, [3, 3, 8]) %%% false
primos(63, L) %%% L = [3, 3, 7]
primos(4, L) %%% L = [2, 2]
primos(524287, L) %%% L = [524287], es un primo de Mersenne
```

Pseudo-código

Definimos, como se indica en el enunciado, la función *next_factor*, además de otra función auxiliar, *primos_aux* que se encarga de hacer toda la recusión para obtener los divisores, de forma que *primos* se basa en comprobar si el numero es 1, y en caso contrario, llamar a *primos_aux* con el factor 2.

```
next_factor(N, F, NF):
        si F==2 y NF==3:
                return true
        si (F < sqrt(N)) y (NF == F+2):
                devolver true
        si F < N:
                devolver true
        si no:
                devolver false

primos_aux(N, L, F):
        si N == 1:
                return true
```

```
        si primer_elem(L) == F:
                T = resto(L)
                devolver ((N%F == 0) y primos_aux(N/F, T, F))
        si no:
                Fn = numero que satisface next_factor(N, F, Fn)
                devolver (N%F != 0) y primos_aux(N, L, Fn)

primos(N, L):
        si (N == 1) y vacia(L):
                devolver true
        si no:
            devolver primos_aux(N, L, 2)
```

Código

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

primos_aux(1, [], _).
primos_aux(N, [F|T], F) :-
    0 is mod(N, F),
    Nn is N/F,
    primos_aux(Nn, T, F),
    !.

primos_aux(N, L, F) :-
    0\= mod(N, F),
    next_factor(N, F, Fn),
    primos_aux(N, L, Fn).

primos(1, []) :- !.
primos(N, L) :-
    primos_aux(N, L, 2).
```

Ejercicio 7

Apartado 7.1

Batería de ejemplos

```
?- cod_primero(1, [1, 2], [2], [1, 1]) %%% true
?- cod_primero(1, [1, 1], [], [1, 1, 1]) %%% true
?- cod_primero(1, [1, 1], Lrem, Lfront) %%% Lrem = [], Lfront = [1, 1, 1]
?- cod_primero(1, [1, 2], [2], [1]) %%% false
?- cod_primero(1, [1, 2], [1, 2], [1]) %%% false
?- cod_primero(1, [1, 1, 2, 2], [2, 2], [1, 1, 1]) %%% true
?- cod_primero(1, [1, 1, 2, 3], Lrem, Lfront) %%% Lrem = [2, 3], Lfront = [1, 1, 1]
```

Con estos ejemplos comprobamos que el código funciona como esperábamos.

Pseudo-código

Con estos ejemplos, podemos desarrollar el pseudo-código de la función, que sería el siguiente:

```
cod_primero(X, L, Lrem, Lfront):
    si vacio(L) y vacio(Lrem) y (Lfront == lista(X)):
        devolver true
    si (L == Lrem) y (Lfront == lista(X)) y (primer_elem(L) != X):
        devolver true
    si no:
        devolver cod_primero(X, resto(L), Lrem, resto(Lfront))
```

Código

Y, a partir del pseudo-código del apartado anterior, obtenemos el código:

```
cod_primero(X, [], [], [X]).
cod_primero(X, [Y|T], [Y|T], [X]) :-
    Y\=X.

cod_primero(X, [X|T], Lrem, [X|Lfront]) :-
    !,
    cod_primero(X, T, Lrem, Lfront).
```

Comentarios sobre la implementación

En este caso no se han tenido que tomar decisiones importantes para la implementación del ejercicio. Fue bastante sencilla, exceptuando el conseguir que diera una única solución como válida, para lo que tuvimos que añadir la condición `cod_primero(X, [Y|T], [Y|T], [X]) :- Y\=X.`

Apartado 7.2

Batería de ejemplos

Una vez desarrollada y probada la función anterior, esta es bastante simple, con lo que bastan unos pocos ejemplos para comprobar su funcionamiento:

```
?- cod_all([1, 1, 1], [[1, 1, 1]]) %%% true
?- cod_all([1, 1, 1], L) %%% L = [[1, 1, 1]]
?- cod_all([1, 1, 2, 2], [[1, 1], [2, 2]]) %%% true
?- cod_all([1, 1, 2, 2], L) %%% L = [[1, 1], [2, 2]]
?- cod_all([1, 1, 2, 3, 3, 3, 3], L) %%% L = [[1, 1], [2], [3, 3, 3, 3]]
```

De nuevo, vemos que el código es correcto.

Pseudo-código

El pseudo-código para esta función sería por tanto:

```
cod_all(L, L1):
    si vacio(L) y vacio(L1):
        devolver true
    si no:
        X = primer_elem(L)
        Y = primer_elem(L1)
        Si existe Lrem que satisface cod_primero(X, resto(L), Lrem, Y):
            devolver cod_all(Lrem, resto(L1))
        si no:
            devolver false
```

Código

Entonces, obtenemos el siguiente código.

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

Comentarios sobre la implementación

Como hemos mencionado en el apartado de *Batería de ejemplos*, una vez desarrollada la función *cod_primero*, esta es muy sencilla, con lo que no hemos econtrado nada destacable en su implementación.

Apartado 7.3

Batería de ejemplos

De nuevo, bastan unos pocos ejemplos para comprobar si el código funciona o no correctamente, pues el único caso que se podría considerar "especial" es cuando solo hay un elemento en la lista.

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

Pseudo-código

Incluimos el pseudo-código de la función principal pedida, junto con el de otras dos funciones auxiliares explicadas en el apartado *Comentarios sobre la implementación*:

```
run_length(L, L1):
    Si existe Ls que satisface cod_all(L, Ls):
        devolver run_length_aux(Ls, L1)
    si no:
        devolver false

run_length_aux(Ls, L1):
    si vacio(Ls) y vacio(L1):
```

```
        devolver true
    si no:
        X = primer_elem(Ls)
        [N, P] = primer_elem(L1)
        devolver comprobar_tupla(X, [N, P]) y run_length_aux(resto(Ls), resto(L1))

comprobar_tupla(L, [N, X]):
    devolver (longitud(L) == N) y (L[0] == X)
```

Código

```
comprobar_tupla(L, [N, X]) :-
    length(L, N),
    nth0(0, L, X).

run_length(L, L1):-
    cod_all(L, Ls),
    run_length_aux(Ls, L1).

run_length_aux([], []).
run_length_aux([X|T], [[N,P]|S]):-
    comprobar_tupla(X, [N, P]),
    run_length_aux(T, S).
```

Comentarios sobre la implementación

Cabe destacar en este ejemplo el uso de dos funciones auxiliares, de forma que *run_length* se encarga simplemente de llamar a *cod_all*, para obtener una lista por cada número distinto de la lista inicial, mientras que *run_length_aux* se encarga de comparar estas listas con las tuplas de L1, para lo que empleamos recursividad, y la segunda función auxiliar desarrollada: *comprobar_tupla*, encargada de comprobar que la longitud de la lista es la indicada en la tupla, y que el primer elemento de la lista es el indicado en la tupla, pues como dicha lista ha sido generada por *cod_all*, todos los elementos son el mismo.

Ejercicio 8

Apartado 8.0

Batería de ejemplos

En este caso, no incluimos en la batería de ejemplos los proporcionados en el enunciado, pues ya se ha comprobado que su funcionamiento es correcto, y al ocupar tanto espacio empeoran la lectura del texto.

Los ejemplos básicos que determinan el correcto funcionamiento del código serían entonces:

```
?- build_tree([a-1, b-2, g-7, f-6, n-14], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% true

?- build_tree([a-1, b-2, g-7, f-6, n-14], tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(g, nil, nil), tree(n, nil, nil))))))
%% false

?- build_tree([a-1, b-2, g-7, f-6, n-14], X)
%% X = tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil)))))
```

Pseudo-código

```
concatena2(X, Y, Z):
    si Z = "X-Y":
        devolver true
    si no:
        devolver false

build_tree(X, Y):
    si X es una lista de un elemento de la forma "A-B" e Y es Tree(A, nil, nil):
        devolver true.
    si el primer elemento de X es de la forma "A-B", Y es de la forma tree(1, L, R) donde
    L es tree(A, nil, nil) y build_tree(rest(X), R) es true, entonces:
        devolver true.
    si no:
        devolver false
```

Código

El código de la función sería por tanto:

```
concatena2(X, Y, X-Y).

build_tree([X], Y):-
    Y = tree(Z, nil, nil),
    concatena2(Z, _, X).

build_tree([X|Rs], Y):-
        concatena2(Z, _, X),
    L = tree(Z, nil, nil),
    build_tree(Rs, R),
    Y = tree(1, L, R).

```

Comentarios sobre la implementación

En la implementación de este ejercicio hemos creado una función auxiliar `concatena2` que concatena 2 elementos X e Y de modo que queden de la forma "X-Y". Esta función se emplea para que la función `build_tree` reciba como argumento una lista cuyos elementos tienen el formato "A-B" donde B es el dato respecto al que se ordenará el árbol como en el ejemplo propuesto en el enunciado.

A la hora de implementar `build_tree` se van generando los nodos hoja, y para generar los nodos intermedios se llama de nuevo a `build_tree` con el resto de la lista que se pasa como argumento.

Apartado 8.1

Batería de ejemplos

De nuevo evitamos repetir los mismos ejemplos que se nos proporcionan en el enunciado para evitar redundancia con este. De este modo los ejemplos necesarios para probar el correcto funcionamiento de este apartado son:

```
encode_elem(g, [1, 0], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% true

encode_elem(g, [0], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% false

encode_elem(n, [0], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% true

encode_elem(n, X, tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% X = [0]

encode_elem(g, X, tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% X = [1, 0]

```

Pseudo-código

Con estos ejemplos, podemos desarrollar el pseudo-código de este apartado, que sería el siguiente:

```
encode_elem(X, Y, T):
    si Y es lista vacía y T es de la forma tree(X, nil, nil):
        devolver true
    si T es tree(1, _, R) donde encode_elem(X, rest(Y), T) es true:
        devolver true
    si Y es una lista con solo el elemento 0 y T es tree(1, L, _) donde encode_elem(X, _, L) es true:
        devolver true
    si no:
        devolver false
```

Código

Por último, una vez tenemos el pseudo-código del apartado anterior, basta con programarlo en prolog:

```
encode_elem(X, Y, T):-
    Y = [],
    T = tree(X, nil, nil).

encode_elem(X, [1|Y], T):-
    T = tree(1, _, R),
    encode_elem(X, Y, R).

encode_elem(X, [0], T):-
    T = tree(1, L, _),
    encode_elem(X, _, L).
```

Comentarios sobre la implementación

El método que sigue esta función para codificar el elemento es recorrer el árbol de acuerdo a los elementos de la lista Y (el elemento codificado), de este modo, si el primer elemento de Y es un 1 se baja por el nodo derecho, y si es un 0 por el izquierdo, tras lo cual se vuelve a llamar a la función con el resto de la lista y con el sub-árbol de la rama resultante. Si al terminar la lista Y nos encontramos en el nodo con el elemento que buscamos, devolvemos true, si no, false.

Apartado 8.2

Batería de ejemplos

Para probar el correcto funcionamiento de la función `encode_list` empleamos los siguientes ejemplos además de los proporcionados en el enunciado:

```
encode_list([a, b, c, d, g], [1, 0], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% false

encode_list([a, b, f, g, n], [[1, 1, 1, 1], [1, 1, 1, 0], [1, 1, 0], [1, 0], [0]], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% true

encode_list([a, b, f, g, n], [[0], [1, 1, 0], [1, 1, 0], [1, 0], [1, 1, 1, 0]], tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% false

encode_list([a, b, c, d, g], X, tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% false

encode_list([a, b, f, g, n], X, tree(1, tree(n, nil, nil), tree(1, tree(g, nil, nil), tree(1, tree(f, nil, nil), tree(1, tree(b, nil, nil), tree(a, nil, nil))))))
%% X = [[1, 1, 1, 1], [1, 1, 1, 0], [1, 1, 0], [1, 0], [0]]

```

Pseudo-código

```
encode_list(X, Y, Z):
    si X e Y son listas de un solo elemento y encode_elem(elemento(X), elemento(Y), Z)
es true:
        devolver true
    si encode_elem(first(X), first(Y), Z) es true y encode_list(rest(X), rest(Y), z)
tambien es true:
        return true
    si no:
        return false
```

Código

```
encode_list([X], [Y], T):-
    encode_elem(X, Y, T).
encode_list([X|R1], [Y|R2], T):-
    encode_elem(X, Y, T),
    encode_list(R1, R2, T).
```

Comentarios sobre la implementación

Para implementar esta función, simplemente vamos recorriendo la lista de elementos que se quieren codificar y vamos realizando un `encode_elem` de ellos y comprobamos que la lista resultado está compuesta por los elementos codificados en orden, y si es así devolvemos true.

Para recorrer las listas simplemente realizamos el `encode_elem` sobre el primer elemento de la lista que se quiere codificar y el primer elemento de la lista resultado, y tras esto llamamos a la función `encode_list` con el resto de ambas listas para comprobar que esto se cumple para e resto de elementos de estas.

Apartado 8.3

Batería de ejemplos

Por ultimo para probar que nuestra implementación de este ejercicio codifica correctamente empleamos los siguientes ejemplos de la función `encode`:

```
encode([e, n, u, n, l, u, g, a, r, d, e, l, a, m, a, n, c, h, a], [[1, 1, 1, 1, 0], [1,
0], [1, 1, 0], [1, 0], [1, 1, 1, 0], [1, 1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 0], [0], [1,
1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 0], [1, 1, 1, 0], [0], [1,
1, 1, 1, 1, 1, 0], [0], [1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1,
0], [0]])
%% true

encode([e, n, u, n, l, u, g, a, r, d, e, l, a, m, a, n, c, h, a], [[1, 1, 1, 1, 0], [1,
0], [1, 1, 0]])
%% false

encode([a, b, a, c, c, a, /], [[0], [1, 1], [0], [1, 0], [1, 0], [0]])
%% false

```

```
encode([e, n, u, n, l, u, g, a, r, d, e, l, a, m, a, n, c, h, a], X)
%% X = [[1, 1, 1, 1, 0], [1, 0], [1, 1, 0], [1, 0], [1, 1, 1, 0], [1, 1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 0], [0], [1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 0], [1, 1, 1, 0], [0], [1, 1, 1, 1, 1, 1, 0], [0], [1, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 0], [0]]

encode([a, b, a, c, c, a, /], X)
%% false
```

Pseudo-código

En este caso, al ser una función más compleja, incluimos el pseudo-código de la función principal pedida, junto con el de otras cuatro funciones auxiliares explicadas en el apartado *Comentarios sobre la implementación*:

```
number_times(X, Y, Z):
    si Y es lista vacía y Z es la tupla [X, 0]:
        return true
    Si first(Y) = X y Z es la tupla [X, N] donde N es Naux + 1 donde Naux cumple que number_times(X, rest(Y), [X, Naux]) es true:
        return true
    si Z es la tupla [X, N] donde N cumple que number_times(X, rest(Y), [X, N]) es true:
        return true
    si no:
        return false


times_list(X, Y, Z):
    si Y y Z son listas vacías:
        return true
    si number_times(first(Y), X, [E, N]) es true, donde E y N cumplen que first(Z) = E-N y times_list(X, rest(Y), rest(Z)) es true:
        return true
    si no:
        return false


ordena(X, Y):
    si Y es X sin repeticiones y ordenada respecto al número de apariciones, y sus elementos tienen la forma "A, B", donde A es el elemento y B el número de apariciones:
        return true
    si no:
        retun false


admisible(X):
    si X es lista vacía:
        return true
    si firts(X) pertenece a diccionario y rest(X) es admisible:
        return true
    si no:
        return false
```

```
35
36
37 encode(X, Y):
38     si X es admisible, y encode_list(X, Y, T) es true donde T es el arbol resultado de
   build_tree(L, T) con L resultado de ordena(X, L):
39         return true
40     si no:
41         return false
```

Código

El código de estas funciones sería por tanto:

```
1  number_times(X, [], [X, 0]).
2  number_times(X, [X|R], [X, N]):-
3      number_times(X, R, [X, N2]),
4      N is N2+1.
5  number_times(X, [_|R], [X, N]):-
6      number_times(X, R, [X, N]).
7
8  times_list(_, [], []).
9  times_list(L1, [X|Ls], [Y|L2]):-
10     number_times(X, L1, [E, N]),
11     Y = E-N,
12     times_list(L1, Ls, L2).
13
14
15 diccionario([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]).
16
17
18 ordena(L1, L2):-
19     sort(L1, Aux),  % Eliminamos repetidos
20     invierte(Aux, Aux2),% Para que en caso de igual número de repeticiones se ordene
   igual que en el ejemplo propuesto.
21     times_list(L1, Aux2, X),
22     sort(2, @>=, X, L2).
23
24
25 admisible([]).
26 admisible([X|R]):-
27     diccionario(L),
28     member(X, L),
29     admisible(R).
30
31 encode(L1, L2):-
32     admisible(L1),
33     ordena(L1, L),
34     build_tree(L, T),
35     encode_list(L1, L2, T).
```

Comentarios sobre la implementación

Para implementar el funcionamiento pedido, hemos decidido implementar varias funciones auxiliares para que el código fuese mas simple, limpio y claro.

La primera de estas funciones auxiliares es `number_times(X, Y, Z)` que comprueba que la tupla Z de la forma [X, N] cumple que N es el número de repeticiones de X en la lista Y, de este modo que introducimos una variable en el campo Z, la función nos devolverá una tupla con X y el número de repeticiones de este X en Y.

Basándonos en esta implementamos la función `times_list(X, Y, Z)` que comprueba que en la lista Z están las tuplas que genera `number_times` con los elementos de la lista Y comprobando las repeticiones de estos en la lista X. De este modo si se introduce una variable en el campo Z, la función devolverá una lista con las tuplas correspondientes a los elementos de la lista Y con las repeticiones en la lista X. Además esta función ya no devuelve las tuplas en el formato [A, N], sino en el formato A-N que es el que queremos.

Otra función auxiliar es `ordena(X, Y)` que Comprueba que Y es la lista X sin repeticiones, con sus elementos en forma de tuplas de `times_list` ordenados de acuerdo a los N de estas tuplas y sin repeticiones. De este modo al introducir una variable en el campo Y la función devuelve la lista X ordenada de acuerdo al número de repeticiones de los elementos y sin elementos duplicados, todo ello en el formato adecuado "A-N". Comentar que en esta función empleamos la función *invierte* implementada en un ejercicio anterior para darle la vuelta a la lista *Aux* resultado de *sort*, de modo que al hacer el segundo *sort*, los elementos que tengan el mismo número de apariciones en la lista queden en el mismo orden que en el ejemplo de ejecución que se nos proporciona en el enunciado, sin embargo, si eliminásemos el *invierte* la implementación sería igualmente válida.

La última función auxiliar implementada es `admisible(X)` que emplea el predicado `diccionario` proporcionado en el enunciado, y comprueba que todos los elementos de la lista X pertenecen a la lista que devuelve este predicado, que es el alfabeto.

Por último al implementar la función `encode(X,Y)` comprobamos que los elementos de X sean admisibles, los ordenamos en una lista auxiliar con la función `ordena`, y generamos el árbol correspondiente a esta lista auxiliar con `build_tree`, tras esto comprobamos que la lista Y es la lista X codificada según `encode_list` con el árbol que hemos generado. De este modo si en el campo Y se introduce una variable, la función devuelve la lista X codificada como se nos pide en este apartado.