MEMORIA PRÁCTICA 3

Alejandro Santorum - alejandro.santorum@estudiante.uam.es
Sergio Galán Martín - sergio.galanm@estudiante.uam.es
Inteligencia Artificial
Práctica 3 Pareja 9

10 de abril de 2019

Contents

1 Introducción

Llegamos a la práctica 3, comezando de nuevo en el mundo de otro lenguaje de programación, en este caso Prolog. Ha sido complicado adaptarse al nuevo paradigma de programación, ya que muchas veces se pensaba en cómo devolver tal variable/valor o como iterar sobre tal lista, cosa que no es muy adecuada en Prolog. Se podría decir que las cosas empezaron a ir más fluídas cuando descubrimos que pensar en implementar un predicado que satisfaciera tal propiedad era más adecuado que pensar en el clásico estilo de función de C, le pasas parámetros y obtienes resultados.

2 Ejercicio 1: predicado duplica(L,L1)

El primer ejercicio trata de implementar un predicado que evalúa a verdadero si y solo si la lista L1 contiene los elementos de L duplicados, llamado duplica(L,L1).

Pseudocódigo duplica(L,L1)

```
for i in 0 to len(L):
    if L[i] == L1[i] and L[i] == L1[i+1]:
        continue
    else:
        false
true
```

Código duplica(L,L1)

```
duplica([],[]). % base case

duplica([X|L], [X, X|L1]) :- % first element of L is equal to the first
    duplica(L,L1).           % and second elements of L1
```

Se muestran a continuación la ejecución de los casos de prueba propuestos más los añadidos adicionalmente:

Casos de prueba

```
?-duplica([], L).
    L = []
?-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
```

3 Ejercicio 2: predicado invierte(L,R)

En este ejercicio se nos pide implementar el predicado invierte(L,R), que evalúa a verdadero cuando R contiene los elementos de L en orden inverso.

En un primer momento se implementó una función asimétrica: si R se dejaba libre el predicado funcionaba correctamente y devolvía la lista inversa a L, no obstante cuando L era la libre, el predicado no devolvía la lista inversa en R. Se ha intentado programar un predicado simétrico, y se ha conseguido... pero con ayuda (claro estaba). A continuación se muestra el código de ambos, el pseudocódigo de la implementación asimétrica (el realizado por nosotros) y la explicación del funcionamiento de la implementación simétrica. Adicionalmente se explicarán los motivos por los cuales la implementación asimétrica funcionaba verdaderamente asimétricamente.

Código concatena(L1, L2, L3) (ya proporcionado)

```
concatena([], L, L). % base case

concatena([X|L1], L2, [X|L3]) :- % elements of first lists goes first
    concatena(L1, L2, L3).       % in the concatenated list
```

Pseudocódigo invierte(L,R)

```
length := len(L)
for i in 0 to length:
    R[length-i] = L[i]
```

Código invierte(L,R) (simétrico) (*)

```
invierte([],Z,Z) :- !. % base case, reversed and auxiliary list must be equal

invierte([H|T],Z,Acc) :-    % Inserting first element of the first list
    invierte(T,Z,[H|Acc]).  % at the beginning of the auxiliary list

invierte(L1, L2) :-         % Interface for invierte/3
    invierte(L1, L2, []).
```

Casos de prueba de invierte simétrico

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

Código invierte(L,R) (asimétrico)

```
% Auxiliary function
last_and_rest([X],X, []).
% It is supposed to return the entire list but the last element
last_and_rest([A|Xs],E, [A|L]) :-
    last_and_rest(Xs, E, L).

invierte([],[]). % base case

invierte([X], [X]):-!. % base case

invierte([X|R], L) :-
    last_and_rest(L, X, Rest), % checking that X is the last element of
    invierte(R, Rest),!.       % the reversed list
```

Los casos de prueba en este caso son iguales a los del caso simétrico.

No se tenía pensado probar el caso invierte(L, [1,2,3]) debido a que era la implementación asimétrica, pero por casualidad decidimos probar después de buscar la implementación de (*). El giro de acontecimientos que sucedió no nos lo esperábamos, funcionaba correctamente, habíamos implementado el predicado invierte de forma simétrica ya desde el principio sin querer y ni siquiera lo habíamos probado.

Vamos a explicar ambas implementaciones y comentar por qué funcionan. En primer lugar, el modelo

del predicado marcado como (*) fue conseguido gracias a un amigo de la Internet en la página *StackOverflow*. Su diseño es muy inteligente, se basa en iterar sobre la primera lista e introducir los elementos que vamos encontrando al principio de una lista auxiliar (así, los primeros introducidos en esta aparecerán al final una vez acabe la recursión). Una vez la primera lista está vacía, comprueba en el caso base que la lista auxiliar coincide con la segunda lista y supuesta inversa.

Por otro lado, nosotros habíamos mejorado el predicado `last` que se había implementado en clase para que devolviese además la lista pero sin el elemento final. La idea de hacer esto era para iterar sobre la primera lista y, entonces, cada elemento que extraíamos tenía que estar al final de la segunda lista e iniciábamos de nuevo la recursión pero ahora la segunda lista era la lista sin el último elemento ya comprobado. La clave de la simetría de este predicado está en que la mejora proporcionada al predicado `last` es simétrica, por lo que el predicado `invierte` es simétrico también al estar basado en `last_and_rest`.

4 Ejercicio 3: palindromo(L)

En el tercer ejercicio se pide implementar el predicado palindromo(L), que evalúa a verdadero si y solo si L es una lista palíndroma.

Este ejercicio es bastante fácil si recordamos la definición de palídromo: número o lista de dígitos que se lee igual en ambos sentidos.

Sabiendo esto está claro que hay que usar el predicado `invierte`, programado anteriormente.

Pseudocódigo palindromo(L)

```
L == reversed(L)
```

Código palindromo(L)

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

Casos de prueba

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

Lo único de posible importancia a comentar es que palídromo(L) devuelve únicamente la lista vacía. Esto es porque la implementación de invertir contiene un *cut* para evitar posibles bucles infinitos. Esto tampoco es ningún problema, el objetivo principal de un predicado es decidir si la expresión es verdadera o falsa, no un generador de expresiones verdaderas.

5 Ejercicio 4: divide(L,N,L1,L2)

Se nos pide implementar el predicado divide(L,N,L1,L2) que evalúa a verdadero si y solo si la lista L1 contiene los primeros N elementos de L y L2 contiene el resto.

Pseudocódigo divide(L,N,L1,L2)

```
for i in 0 to N-1:
    L1[i] = L[i]
for i in N to len(L):
    L2[i-N] = L[i]
```

Código divide(L,N,L1,L2)

```
divide(L, 0, [], L). % base case, L1 is empty, so the rest of the
                     % elements belogs to L2

divide([F|R1],N,[F|R2],L) :- % iterating over first N elements of L
    divide(R1, M, R2, L),    % inserting those N elems into L1
    N is M+1.
```

Casos de prueba

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

```
    L2 = []
?-divide([1,2,3,4,5], 2, L1,L2).
    L1 = [1, 2],
    L2 = [3, 4, 5]
    false
?-divide(L, 3, [1, 2, 3], [4, 5, 6]).
    L = [1,2,3,4,5,6]
```

Visto esto podemos concluir que el predicado funciona según lo especificado y continuamos en la lista de ejercicios.

6 Ejercicio 5: aplasta(L,L1)

Se pide 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.

Pseudocódigo aplasta(L,L1)

```
aplasta(L, flatten_list):
    while(!is_empty(L)):
        f = first(L)
        r = rest(L)
        if is_list(f):
            aplasta(f, aux)  % aux := a new flatten_list got from recursion
            cat = concatenar(flatten_list, aux)
        else:
            cat = concatenar(flatten_list, list(f))
        aplasta(r, cat)
```

Código aplasta(L,L1)

```
aplasta(L, Ret) :-          % interface for aplasta/3
    aplasta(L, [], Ret),!.

aplasta([], Ret, Ret).      % base case

aplasta([F|R], Aux, Ret) :-
    is_list(F),             % case in which first(L) is a list
    !,
    aplasta(F, Ret2),
    concatena(Aux, Ret2, Concat), % concatenating flatten list with the list got before
    aplasta(R, Concat, Ret).

aplasta([F|R], Aux, Ret) :-   % case in which first(L) is not a list
    concatena(Aux, [F], Concat),
    aplasta(R, Concat, Ret).
```

Nota: Posiblemente sobresalte a los ojos qué hace un *cut* en medio del predicado `aplasta([F|R], Aux, Ret)`. Está colocado ahí para que afecte a los predicados llamados después de él, pero no al predicado `is_list(F)` que le precede, para evitar posibles confusiones entre este predicado y el predicado que supone que el primer elemento de la lista a aplastar es realmente un elemento y no otra lista.

Casos de prueba

```
?-aplasta([], L).
    L = []
?-aplasta([[1]], L).
    L = [1]
?-aplasta([[[[[1,2,3]],4]]], L).
    L = [1,2,3,4]
?-aplasta([1, [2, [3, 4], 5], [6, 7]], L).
    L = [1, 2, 3, 4, 5, 6, 7]
?-aplasta([1, [2, [3, 4], 5], [6, 7]], [1, 2, 3, 4, 5, 6, 7]).
    true
aplasta([1, [2, [3, 4], 5], [6, 7]], [1, 2, 3, 4, 5, 6, 7, 8]).
    false
?-aplasta([[[[[1,[2]],3],5],6,7],8,[9,[10],[11],12]], L).
    L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Adicionalmente, se pide en el enunciado que expliquemos la situación **?-aplasta(L,[1, 2, 3])**. En nuestro caso el resultado que obtenemos es **L = [1,2,3]**, que es la lista más sencilla que satisface que [1,2,3] sea su lista aplasta. No continúa buscando más debido a un *cut* usado en el caso base.

7 Ejercicio 6: primos(N,L)

En este ejercicio se nos pedía implementar el predicado primos(N, L) que se satisface cuando la lista L contiene los factores primos del número N.

Pseudocódigo primos(N,L)

```
primos(N, List, Factor, Ninit):
    If N%Factor == 0:
        NewN = N/Factor
        F = first(L)
        R = rest(L)
        NewN == F % condicion para que la lista sea correcta
```

```
        primos(NewN, R, Factor, Ninit) % volvemos a llamarla con el mismo
                                       %factor por si tiene multiplicidad mayor
    Else:   % No es divisible por el factor actual
        NextF = next_factor(Factor, Ninit) % Conseguimos otro posible factor
        primos(N, L, NextF, Ninit)
```

Código primos(N,L)

```
next_factor(N, F, NF) :- % getting possible next factor
    F is 2,
    NF is 3;
    F =< N//2, % we had to fix it from documentation
    NF is F+2,
    1 is mod(F,2).

primos(1,[],_,_). % base case
primos(N, [N],_,_). % base case if N is prime

primos(N, [NF|R], F, Init) :-
    next_factor(Init, F, NF), % Initializing factor list if not instanciated
    primos(N, [NF|R], NF, Init).

primos(N, L, F, Init) :-
    next_factor(Init, F, NF), % F does not divide N
    primos(N, L, NF, Init).

primos(N, [F|R], F, Init) :-
    0 is mod(N,F),  % case in which F divides N
    X is div(N,F),
    primos(X, R, F, Init).

primos(N, L) :- primos(N, L, 2, N), !. % interface for primos/4
```

Casos de prueba

```
?-primos(1,[]).
    true
?-primos(1, L).
    L = []
?-primos(2,[2]).
    true
?-primos(2, L).
    L = [2]
?-primos(4, L).
    L = [2,2]
?-primos(12, L).
    L = [2,2,3]
?-primos(30, [2,3,5]).
    true
?-primos(100, [2,2,5,5,7]).
    false
?-primos(100, [2,2,5,5]).
    true
```

Con esto podemos estar bastante seguros de que la implementación es decente.

8 Ejercicio 7: codificación *run-length* de una lista

En este ejercicio se nos pide implementar la codificación run-length de una lista. Para ello, se nos subdivide esa tarea en tres partes, las dos primeras destinadas a crear funciones que serán esenciales para la implementación final de la función run-length en la tercera.

8.1 Ej. 7.1: cod_primero(X, L, Lrem, Lfront).

Para empezar, haremos una función que, dado un elemento y una lista, devuelva ese elemento y todos los que sean igual a él de la otra lista (siempre y cuando estén al principio de dicha lista), y por otro lado el resto de la lista.

Pseudocódigo cod_primero(X,L,Lrem,Lfront)

```
while(L not empty and first(L) == X):
    Lfront.append(X)
    L.delete(0)  %We delete the first element of L
Lfront.append(X)
Lrem = L
```

Código cod_primero(X,L,Lrem,Lfront)

```
cod_primero(X,[],[],[X]). %base case

cod_primero(X,[Y|R],[Y|R],[X]) :- %base case
    not(X is Y).

cod_primero(X,[X|R],LRem,[X|R2]) :- %recursion
    cod_primero(X,R,LRem,R2).
```

Casos de prueba

```
?-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]
    false
?-cod_primero(Elem, L, [2,3,4], [1]).
    Elem = 1,
    L = [2, 3, 4]
    false
?-cod_primero(1, [1, 1, 2, 3], [1,1,1,1], [2,3]).
    false
```

8.2 Ej. 7.2: cod_all(L, L1)

Una vez hecha la función base, la empezaremos generalizando para separar los bloques de números seguidos de una lista.

Pseudocódigo cod_all(L,L1)

```
while(L not empty):
    cod_primero(L.first(),L,Lrem,Lfront)
    L1.append(Lfront)
    L = Lrem
```

Código cod_all(L,L1)

```
cod_all([F|R],[E]) :-   %base case
    cod_primero(F,R,[],E).

cod_all([F1|R1],[F2|R2]) :- %recursion
cod_primero(F1,R1,Aux,F2), cod_all(Aux,R2).
```

Casos de prueba

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

8.3 Ej. 7.3: run_length(L, L1)

Por último vamos a juntar las funciones implementadas anteriormente (y algunas auxiliares nuevas) para poder implementar la función objetivo del ejercicio, run_length.

Pseudocódigo run_length(L,L1)

```
%% is_coded(L1,[count,elem]):
    count = 0
    while(L1 not empty):
        if(L1.first() == elem):
            count++
        else:
            false
    true
%% format_list(L,L1):
    if(L is empty and L1 is empty):
        true
    if(L is empty or L1 is empty):
        false
    is_coded(L.first(), L1.first()) and format_list(L.rest(),L1.rest)
run_length(L,L1):
    if(L is empty and L1 is empty):
        true
    if(L is empty or L1 is empty):
        false
    cod_all(L, Aux)
    format_list(Aux, L1)
```

Código run_length(L,L1)

```
is_coded([],[0,_]). %base case

is_coded([F1|R1],[F2,F1]) :-
    is_coded(R1,[F3,F1]), F2 is F3+1.

format_list([],[]).

format_list([F1|R1],[F2|R2]) :-
```

```
    is_coded(F1,F2),format_list(R1,R2).

run_length(L,L1) :-
    cod_all(L,Aux),format_list(Aux,L1).
```

Casos de prueba

```
?-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]]
    false
?-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
    false
```

Vamos a aclarar un poco el funcionamiento de las dos funciones auxiliares que hemos implementado para este apartado. La primera, is_coded(L,L1), es un predicado que evaluara a true cuando L1 es la codificación del bloque L. Es decir, L1 será una tupla en la que el primer elemento es el número de apariciones del elemento en la lista y el segundo el elemento de la lista. La lista que recibirá siempre será una lista de repeticiones del mismo elemento. La segunda, format_list(L,L1), básicamente aplicará is_coded/2 a cada bloque de números de una lista dada.

9 Ejercicio 8: Árbol de Huffman & build_tree(List, Tree)

Llegamos al último ejercicio de la práctica, donde vamos a implementar y usar el **Árbol de Huffman**, una estructura de datos usada para odificar y comprimir información.

En este primer apartado se diseñará el predicado `build_tree(List, Tree)` que tiene como objetivo construir un arbil binario a partir de una lista dada.

Pseudocódigo build_tree(L,T)

```
    if len(L) == 1:
        tree(L[0],nil,nil)
    else:
        tree(1, tree(L[0],nil,nil), build_tree(L[1:]))
```

Código build_tree(L,T))

```
parse(X-Y, X, Y). % parsing element with the form of [a-b] as a and b separately

build_tree([], X) :- % base case: empty list of elements
    X = tree(nil,nil,nil),!.

build_tree([A], X) :- % base case: list with a single element
    parse(A, A1, _),
    X = tree(A1,nil,nil),!.

build_tree([F|R], X) :-
    parse(F, F1, _), % parsing element
    build_tree(R,Z), % calling recursively to keep building the tree
    X = tree(1, tree(F1,nil,nil), Z). % building the tree
```

Casos de prueba

```
?-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))))
```

9.1 Ej. 8.1: encode_elem(X1, X2, Tree)

Tenemos que implementar el predicado `encode_elem(X1,X2,T)`, que codifica el elemento X1 en X2 basándose en la estructura del árbol T.

Pseudocódigo encode_elem(X1,X2,T)

```
    if T.left() == X1:
        X2.append(0)
        return X2
    else:
        X2.append(1)
        return encode_elem(X1,X2,T.right())
```

Código encode_elem(X1,X2,T))

```
encode_elem(X1,[F],T) :- % base case
    T = tree(_, L, _),
    L = tree(X1,_,_),    % element is found at the left tree
    F is 0,!.            % the list must contain a 0

encode_elem(X1,[F],T) :- % base case
    T = tree(_, _, R),
    R = tree(X1,_,_),    % element is found at the last right tree
    F is 1,!.

encode_elem(X1,[1|Rest],T) :-
    T = tree(_, _, R),
    encode_elem(X1,Rest,R). % the element is not at the left tree, nor the right
                            % keep recursively
```

Casos de prueba

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

9.2 Ej. 8.2: encode_list(L1, L2, Tree)

En penúltimo apartado hay que programar un predicado, `encode_list(L1,L2,T)`, que codifica la lista L1 en L2 siguiendo la estructura del árbol T.

Pseudocódigo encode_list(L1,L2,T)

```
    for elem in L1:
        L2.append(encode_elem(elem,T))
```

Código encode_list(L1,L2,T))

```
encode_list([],[],_). % base case

encode_list([Fc|Rc],[Fk|Rk],T) :-
    encode_elem(Fc,Fk,T), % using last predicate as auxiliary
    encode_list(Rc,Rk,T). % carry on recursively
```

Casos de prueba

```
?-encode_list([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]]
?-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]]
?-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, 0], [0]]
?-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
```

9.3 Ej. 8.3: encode(L1, L2)

Finalmente llegamos al último apartado. En este caso se requiere implementar el predicado `encode(L1,L2)`, que codifica la lista L1 en L2. Para ello usaremos el predicado `dictionary(L)` para instanciar los símbolos permitidos.

Para este apartado adicionalmente se han implementado varias funciones auxiliares, con el objetivo de clarificar y modularizar el código. Serán explicadas al final.

Pseudocódigo encode(L1,L2)

```
    D = dictionary_init(allowed_symbols) % Creando el diccionario
    belong(L1,D) % Comprobando que todos los elementos de L1 estan en D
    list_freq = frequencies(L1) % Calculando las frecuencias de cada elem. en L1
    sorted_list = freq_sort(list_freq) % Ordenando la lista segun su frecuencia
    t = build_tree(sorted_list) % Construyendo el arbol de Huffman
    L2 = encode_list(L1, t). % Obteniendo resultado final
```

Código encode(L1,L2) y funciones auxiliares)

```
parse(X-Y, X, Y). % parsing element with the form of [a-b] as a and b separately

build_tree([], X) :- % base case: empty list of elements
```

```
    X = tree(nil,nil,nil),!.
build_tree([A], X) :- % base case: list with a single element
    parse(A, A1, _),
    X = tree(A1,nil,nil),!.
build_tree([F|R], X) :-
    parse(F, F1, _), % parsing element
    build_tree(R,Z), % calling recursively to keep building the tree
    X = tree(1, tree(F1,nil,nil), Z). % building the tree


% Encodes an element
encode_elem(X1,[F],T) :- % base case
    T = tree(_, L, _),
    L = tree(X1,_,_),    % element is found at the left tree
    F is 0,!.            % the list must contain a 0
encode_elem(X1,[F],T) :- % base case
    T = tree(_, _, R),
    R = tree(X1,_,_),    % element is found at the last right tree
    F is 1,!.
encode_elem(X1,[1|Rest],T) :-
    T = tree(_, _, R),
    encode_elem(X1,Rest,R). % the element is not at the left tree, nor the right
                            % keep recursively

% Encodes a whole list
encode_list([],[],_). % base case
encode_list([Fc|Rc],[Fk|Rk],T) :-
    encode_elem(Fc,Fk,T), % using last predicate as auxiliary
    encode_list(Rc,Rk,T). % carry on recursively



% Calculates the frequency of a given element in a given list
freq_elem(Elem, List, Freq) :-
    freq_elem(Elem, List, Freq, Freq),!.
freq_elem(_, [], _, 0).
freq_elem(Elem, [Elem|R], Freq, Actual) :-
    freq_elem(Elem, R, Freq, Next),
    Actual is Next+1.
freq_elem(Elem, [_|R], Freq, Actual) :-
    freq_elem(Elem, R, Freq, Actual).

% Calculates the frequency of the elements of a given list
frequencies([],_,[]).
frequencies([Elem|Rest], L, [LFf|LFr]) :-
    parse(LFf, Elem, Freq),
    freq_elem(Elem, L, Freq),
    frequencies(Rest,L,LFr).

% Checking a given element is allowed comparing it with the allowed symbols list
elem_belong(Elem, [Elem|_]).
elem_belong(Elem, [_|Ss]) :-
    elem_belong(Elem,Ss).

% Checking all the expression's symbols are allowed
belong([],_).
belong([Elem|Ls],Symbols) :-
        elem_belong(Elem,Symbols),
        belong(Ls,Symbols).

% Calculates the maximum element of a list
list_max([L|Ls],Max) :-
    list_max(Ls,L,Max),!.
list_max([],Max,Max).
list_max([L|Ls],Max0,Max) :-
    parse(Max0,_,MaxNum1),
    parse(L,_,MaxNum2),
    MaxNum1 is max(MaxNum1,MaxNum2),
    list_max(Ls,Max0,Max).
list_max([L|Ls],Max0,Max) :-
    parse(Max0,_,MaxNum1),
    parse(L,_,MaxNum2),
    MaxNum2 is max(MaxNum1,MaxNum2),
    list_max(Ls,L,Max).

% Deletes a given element of a list
delete_letter(_,[],[]).
delete_letter(Let,[Let|R],R):-!.
delete_letter(Let,[F|R],[F|Result]) :-
    delete_letter(Let,R,Result).

% Calculates and deletes the maximum of a list
maximum(Ls,A,Ys) :-
    list_max(Ls,A),
    delete_letter(A,Ls,Ys).

% Sorts the given list by frequency
freq_sort([],[]).
freq_sort(L,[M|Rs]) :-
    maximum(L,M,Rest),freq_sort(Rest,Rs).

```

```
% Dictionary for this assignment
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]).

encode(L,X) :-
    dictionary(Allowed_symbols), % Creating dictionary of allowed symbols
    invierte(Allowed_symbols, Symbols), % Inverting symbols
    belong(L,Symbols), % Checking all the expression list's symbols are allowed
    frequencies(Symbols,L,LF), % Calculating frequencies of each symbol
    freq_sort(LF,SL), % Sorting symbols by frequency
    build_tree(SL,T), % Building the tree
    encode_list(L,X,T). % Encoding the expression
```

Casos de prueba

```
?-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, 0], [1, 1, 0], [1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 0], [0], [1, 1, 1,
        1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 0], [0], [1,
         0], [1, 0], [1, 1, 1, 1, 1, 1, 1, 0], [1, 1, 0], [0], [1, 1, 1, 1, 1, 1, 1, 1, 1,
        0], [0], [1, 1, 1, 1, 1, 1, 0], [0], [1, 0], [1, 1, 1, 1, 0]]
    false
?-encode([i,a],X).
    X = [[0], [1, 0]]
    false
?-encode([i,2,a],X).
    false
```

Vamos a describir brevemente las funciones auxiliares utilizadas:

· `freq_elem(Elem, List, Freq)`: calcula la frecuencia del elemento Elem en la lista List.

· `frequencies(D, L, LF)`: calcula la frecuencia en la lista L de cada símbolo del diccionario D, devolviendo la lista de frecuencia en LF.

· `freq_belong(Elem, D)`: comprueba que un elemento dado está permitido, es decir, está en el diccionario D.

· `belong(L,D)`: comprueba que todos los elementos de la lista L están permitidos (aparecen en el diccionario D), devolviendo true o false.

· `list_max(L,Max)`: calcula el elemento máximo de la lista L.

· `delete_letter(Elem, L, L2)`: elimina un elemento dado de la lista L, devolviendo la lista modificada en L2.

· `maximum(L, Elem, L2)`: predicado que evalúa a verdadero si y solo si el elemento máximo de L es Elem y la lista sin este elemento es L2.

· `freq_sort(L,SL)`: ordena la lista L por frencuencia, devolviendo la lista ordenada en SL.