PRÁCTICA 3 INTELIGENCIA ARTIFICIAL: PROLOG

Grupo 2301, Pareja 03

Alejandro Cabana Suárez y Aitor Arnaiz del Val

EJERCICIO 1

Implementa un predicado pertenece_m(X, L) que comprueba si el elemento está en la lista o en algunas de sus sublistas

CÓDIGO

```
pertenece_m(X, [X|_]) :- X \= [_|_].
pertenece_m(X, [L|_]) :- pertenece_m(X, L).
pertenece_m(X, [_|Rs]) :- pertenece_m(X, Rs).
```

PSEUDOCÓDIGO

```
FUNCTION: pertenece_m(X, L)

INPUT: X: Elemento que queremos ver si pertenece a la lista L o a alguna de
          sus sublistas
       L: Lista en la que queremos saber si esta o no el elemento X

OUTPUT: -Si introducimos como primer argumento un elemento susceptible de
         estar en la lista:
                -true si efectivamente pertenece a la lista
                -false en caso contrario
        -Si introducimos como primer elemento una variable (X):
                -Los elementos pertenecientes a la lista y a las sublistas
                 de esta, uno a uno.

PROCESSING:
Un elemento pertenece a una lista si:
            -o bien el primer el primer elemento de la lista no es una
                sublista y este coincide con nuestro elemento a comprobar

            -o bien el primer elemento de la lista es una sublista y nuestro
                 elemento a comprobar pertenece a esa sublista

            -o bien pertenece al resto de la lista
```

BATERÍA DE PRUEBAS

```
?- pertenece_m(X, [2,[1,3],[1,[4,5]]]).
X = 2 ;
X = 1 ;
X = 3 ;
X = 1 ;
X = 4 ;
X = 5 ;
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

CÓDIGO

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

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

PSEUDOCÓDIGO

```
FUNCTION: invierte(L, R)

INPUT: L: lista
       R: lista

OUTPUT: -Si uno de los argumentos es una lista y el otro una variable:
            -Nos devuelve la lista inversa a la pasada por parametro

        -Si ambos argumentos son listas:
            -true en caso de que una sean inversas la una de la otra
            -false en caso contrario

        -Si ambos argumentos son variables:
            -Devuelve que ambas listas han de ser iguales, y que una de
             ellas es vacia

PROCESSING: Una lista (L) es inversa de otra (R) si:
                -Ambas son vacias
                -R sin su ultimo elemento invierte a L sin su primer
                 elemento
```

BATERÍA DE PRUEBAS

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

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

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

EJERCICIO 3

Implementar el predicado insert(X-P, L, R) que inserte un par de elementos (X-P) en una lista de pares ordenados (L) en una posición (P), desplazando el resto de elementos, en la lista (R). Se considera que la primera posición de una lista es la 1.

CÓDIGO

```
insert([X–P], [], [X–P]).
insert([X–P], [X1–P1|Rs], [X–P|[X1–P1|Rs]]) :- P =< P1.
insert([X–P], [X1–P1|Rs], [X1–P1|L]) :- P > P1, insert([X–P], Rs, L).
```

PSEUDOCÓDIGO

```
FUNCTION: insert(X–P, L, R)

INPUT: X–P: Par de elementos a insertar (en un indice P)
       L: Lista donde se insertara el par de elementos mencionado
       R: Lista de retorno de la insercion resultante

OUTPUT: Lista con la insercion realizada (R)

PROCESSING: –Si uno de los argumentos es la lista vacia, la lista resultante
              es igual a la otra lista argumento.

            –Si el elemento a insertar tiene indice menor que el primer
              elemento de la lista donde se ha de insertar, se inserta en
              la primera posicion.

            –Si, por el contrario, tiene un indice mayor al primer elemento
              de la lista donde se ha de insertar, se inserta en el resto de
              esta.
```

BATERÍA DE PRUEBAS

```
?- insert([a-6],[], X).
X = [a-6].

?- insert([a-6],[p-0], X).
X = [p-0, a-6].

?- insert([a-6],[p-0, g-7], X).
X = [p-0, a-6, g-7],
false.

?- insert([a-6],[p-0, g-7, t-2], X).
X = [p-0, a-6, g-7, t-2],
false.
```

EJERCICIO 4

4.1 Implementar el predicado elem_count(X, L, Xn) que se satisface cuando el elemento (X) aparece (Xn) veces en la lista (L).

CÓDIGO

```
elem_count(_, [], 0).
elem_count(X, [X|Rs], Xn) :- elem_count(X, Rs, Xm), Xn is Xm + 1.
elem_count(X, [Y|Rs], Xn) :- X \= Y, elem_count(X, Rs, Xn).
```

PSEUDOCÓDIGO

```
FUNCTION: elem_count(X, L, Xn)

INPUT: X: Elemento
       L: Lista
       Xn: Numero de veces que X es susceptible de aparecer en L

OUTPUT: Si el Xn introducido es un numero:
            -true si Xn es, en efecto, el numero de veces que aparece
             X en L
             -false en caso contrario

        Si el Xn introducido es una variable:
            -El numero de veces que aparece X en L, en caso de que lo haga.
            -0 en caso de que X no aparezca ninguna vez en L.

PROCESSING: -Un elemento no aparece ninguna vez en una lista vacia

            -Si el primer elemento de L coincide con X, se ve si
             aparece en el resto de la lista y se incrementa Xn
             (numero de apariciones de X en L)

            -Si, por el contrario, no coincide con el primer elemento,
             se ve si esta (y cuantas veces) en el resto de la lista.
```

BATERÍA DE PRUEBAS

```
?- elem_count(b,[b,a,b,a,b],Xn).
Xn = 3,
false.

?- elem_count(a,[b,a,b,a,b],Xn).
Xn = 2,
false.
```

4.2 Implementar el predicado list_count(L1, L2, L3) que se satisface cuando la lista (L3) contiene las ocurrencias de los elementos de (L1) en (L2) en forma de par.

CÓDIGO

```
list_count([], _, []).
list_count([X|L1], L2, [X-Cx|L3]) :- elem_count(X, L2, Cx),
                                     list_count(L1, L2, L3).
```

PSEUDOCÓDIGO

```
FUNCTION: list_count(L1, L2, L3)

INPUT: L1: Lista de elementos (sin repetir) a contar en L2
       L2: Lista de elementos (posibles repeticiones)
       L3: Lista de pares elemento-apariciones_en_L2

OUTPUT: L3: lista de pares elemento_de_L1-recuento_en_L2


PROCESSING: -Si en L1 no hay ningun elemento, no hay apariciones
             (pues no hay elementos de los que hacer recuento)

            -Se hace el recuento de elementos cada elemento
             de L1 en L2 y se mete el par elemento-recuento
             en la lista L3 hasta que se nos acaben los
             elementos de L1
```

BATERÍA DE PRUEBAS

```
?- list_count([b],[b,a,b,a,b],Xn).
Xn = [b-3]
False

?- list_count([b,a],[b,a,b,a,b],Xn).
Xn = [b-3, a-2]
False

?- list_count([b,a,c],[b,a,b,a,b],Xn).
Xn = [b-3, a-2, c-0]
false
```

EJERCICIO 5

Implementar el predicado sort_list(L1, L2) que se satisface cuando la lista (L2) contiene los pares de elementos de la lista (L1) en orden.

CÓDIGO

```
sort_list([], []).
sort_list([X-P|Rs], L2) :- sort_list(Rs, L), insert([X-P], L, L2).
```

PSEUDOCÓDIGO

```
FUNCTION: sort_list(L1, L2)

INPUT: L1: Lista de elementos en desorden
       L2: Lista de elementos desordenados

OUTPUT: L2: Lista de elementos de L1 ordenados por el segundo
            elemento de cada par

PROCESSING: Una lista esta ordenada si:
                  -Es vacia
                  -O bien es el resultadi de insertar cada elemento
                   de la lista desordenada en orden (ascendente)
```

BATERÍA DE PRUEBAS

```
?-sort_list([p-0, a-6, g-7, t-2], X).
X = [p-0, t-2, a-6, g-7]
false

?-sort_list([p-0, a-6, g-7, p-9, t-2], X).
X = [p-0, t-2, a-6, g-7, p-9]
false

?-sort_list([p-0, a-6, g-7, p-9, t-2, 9-99], X).
X = [p-0, t-2, a-6, g-7, p-9, 9-99]
false
```

EJERCICIO 6

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.

CÓDIGO

```
build_tree([], tree(1, nil, nil)).
build_tree([X-_], tree(X, nil, nil)).
build_tree([X-_|Rs], tree(1, tree(X, nil, nil), T)) :- Rs \= [],
                                                       build_tree(Rs, T).
```

PSEUDOCÓDIGO

```
FUNCTION: build-tree(List, Tree)

INPUT: List: Lista de elementos a meter en el arbol
       Tree: Arbol resultante de la insercion mencionada

OUTPUT: Tree: Arbol resultado

PROCESSING: -El arbol resultante de una lista vacia es vacio

            -El arbol resultante de una lista con un solo par
             Tiene como campo Info el primer elemento del par
             y como hijos, nil (ambos)

            -El arbol resultante de una lista con varios elementos
             se forma como el anterior, y sus siguientes elementos
             de forma recursiva de la misma forma, mientras queden
             elementos en List
```

BATERÍA DE PRUEBAS

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

?-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))))))
False

?-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))))
False

?-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 7

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

CÓDIGO

```
encode_elem(X, [0], tree(_, tree(X, _, _), _)).
encode_elem(X, [1], tree(_, _, tree(X, _, _))).
encode_elem(X, [1|R], tree(_, tree(Y, _, _), tree(Z, Tl, Tr))) :-
                                        Y \= X,
                                        Z\= X,
                                        encode_elem(X, R, tree(Z, Tl, Tr)).
```

PSEUDOCÓDIGO

```
FUNCTION: encode_elem(X1, X2, tree)

INPUT: X1: Elemento a codificar
       X2: Elemento codificado
       tree: Arbol usado para codificar X1 en X2

OUTPUT: X2: Codificacion de X1 basada en la estructura de tree

PROCESSING: Si el elemento que queremos codificar:
                -Esta en la rama de la izquierda: metemos un 0
                 en la siguiente posicion de la lista de codificacion

                -Esta en la rama de la derecha: metemos un 1
                 en la siguiente posicion de la lista de codificacion

                - No esta en ninguna de ambas: insertamos un 1 y seguimos
                  mientras queden elementos en el arbol.
```

BATERÍA DE PRUEBAS

```
?-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))))


?- 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]
false

?- 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]
false

?- 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]
false

?- 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]
false
```

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

CÓDIGO

```
encode_list([], [], _).
encode_list([X|R], [Ex|Er], T) :- encode_elem(X, Ex, T), encode_list(R, Er,
    T).
```

PSEUDOCÓDIGO

```
FUNCTION: encode_list(L1, L2, tree)

INPUT: L1: Lista a codificar
       L2: Lista codificado
       tree: Arbol usado para codificar L1 en L2

OUTPUT: L2: Codificacion de L1

PROCESSING: Codificamos, uno a uno, cada elemento de la lista y lo metemos
            en una lista de listas en la que cada sublista es la
            codificacion de cada elemento de L1, en orden
```

BATERÍA DE PRUEBAS

```
?- 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]]
false

?- 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]]
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

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

CÓDIGO

```
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(L1, L2) :- dictionary(D), list_count(D, L1, CD), sort_list(CD, OCD),
    invierte(OCD, IOCD), build_tree(IOCD , Tree), encode_list(L1, L2, Tree).
```

PSEUDOCÓDIGO

```
FUNCTION: encode(L1, L2)

INPUT: L1: Lista a codificar
       L2: Lista donde se guardara la codificacion de L1

OUTPUT: L2: Lista con la codificacion de L1

PROCESSING: Para codificar la lista L1 en L2:
            - Creamos un diccionario que
              tenga todos los posibles caracteres que pueden aparecer en la
              lista a codificar
            - Contamos cuantas veces aparece cada
              elemento del diccionario en la lista a codificar, generando
              una lista con los pares elemento-aparicion mencionados.
            - Ordenamos dicha lista por el numero de apariciones
            - Invertimos la lista ordenada, para ordenarla en orden
              decreciente de apariciones
            - Generamos un arbol a partir de dicha lista
            - Codificamos la lista L1 y guardamos la codificacion en L2
              basandonos en la estructura del arbol generado mencionado
```

BATERÍA DE PRUEBAS

```
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([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
```