Práctica 3 Inteligencia Artificial

Pareja 09 (Lucía Asencio y Juan Riera)

Abril 2018

EJERCICIO 1

PSEUDOCÓDIGO

Entrada: elemento X, lista L
Salida: T si X pertenece a L, o a una lista de L
Procesamiento:
Si X es el primer elemento de L y X no es una lista: True
Else
Si el primer elemento de L es una lista L2, y X pertenece a L2: T
Else
Si X no es el primer elemento de L pero X pertenece a los demás elementos de L: T
Else
F

CÓDIGO

```
/* EJERCICIO 1 */

/*
* Cierto elemento X pertenece a una lista cuyo primer elto es X
* si ese primer elto NO es una lista
*/
pertenece_m(X, [X|_]) :- X\=[_|_].
/*
* Cierto elto X pertenece a una lista cuyo primer elto es una lista L
* si X pertenece a L
*/
pertenece_m(X, [[Y|Zs]|_]) :- pertenece_m(X, [Y|Zs]).
/*
* Cierto elto X pertenece a una lista si pertenece a la lista formada
* por todos los elementos menos el primero
*/
pertenece_m(X, [_|Rs]) :- pertenece_m(X, Rs).

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

EJERCICIO 2

PSEUDOCÓDIGO:

Entrada: Lista L1, lista L2 de tamaño n
Salida:T si L1 es L2 invertida en orden
Procesamiento:
Si L1 está vacía y L2 también: T
Else
Si el inverso los n-1 últimos eltos de L1 concat al primer elto de L1 == L2: T
Else
F

CÓDIGO

```
/* EJERCICIO 2*/
concatena([], L, L).
concatena([X|L1], L2, [X|L3]) :-
    concatena(L1, L2, L3).

/*
* Si L es la lista vacia, entonces L es el inverso de la lista vacia
*/

invierte([], L) :-
    L=[].
/*
* Dada cierta lista A: si L es la concatenacion del inverso del resto
* de A con el primer elemento de A, entonces L es el inverso de A
*/
invierte([X|Y], L) :-
    invierte(Y, Inv),
    concatena(Inv , [X], L).

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

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

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

EJERCICIO 3

PSEUDOCÓDIGO

Entrada: Una lista cuyo único elemento será una tupla compuesta por un elemento cualquiera y un número, una lista de tuplas de la misma forma que la anterior ordenada ascendentemente por el segundo elemnto de las mismas, y un elemento cualquiera X.

Salida: True si el tercer argumento es la lista de entrada recibida como segundo argumento con el elemento recibido como primer argumento insertado, de tal forma que la lista final siga ordenada, y false en caso contrario

Procesamiento:

1. Si el elemento a insertar es mayor que el primer elemento de la lista recibida como tercer argumento, será verdadero si X es la concatenación de este primer elemento, y el resultado de insertar el elemento recibido como primer argumento en el resto de la lista recibida como segundo
2. Si el elemento insertar es menor que el primer elemento de al lista, X tendrá que ser el resultado de concatenar este primer elemento con la lista completa.
3. Si la lista recibida como segundo argumento está vacía, X tendrá que ser igual al primero.

CÓDIGO

```
/* Ejercicio 3 */
/* Si no hay lista en la que insertar, la lista contendra
*  unicamente el propio elemento a insertar
*/
insert(X, [], X).
/* Si el elemento a insertar es menor
* que el primer elemento de la lista, hemos encontrado su lugar
* y devolvemos true si el ultimo elemento de la lista es el
* resultado de concatenar el elemento y el resto de la lista */
insert([C-A], [D-B|R1], X) :-
    A=<B, concatena([C-A],
    [D-B|R1], X).
/* Si el elemento a insertar es mayor que el primer
* elemento de la lista seguimos buscando y devolvemos true
* si el ultimo argumento es el resultado de concatenar
* el primer elemento de la lista con el resultado de
* continuar la recursion con el resto de la lista, es decir,
* con el resto de la lista con el elemento insertado */
insert([C-A], [D-B|R1], [D-B|X]) :-
    A>B ,
    insert([C-A], R1, X).
```

EJERCICIO 4

EJERCICIO 4.1

PSEUDOCÓDIGO

Entrada: Elto X, lista L, num N
Salida: T si X está N veces en L
Procesamiento:
Si la lista está vacía y N = 0: T
Else
Si 1er elto de L es X y X está N-1 veces en el resto de L: T
Else
Si X no el el 1er elto de L y X está N veces en el resto de L: T
Else
False

CÓDIGO

```
  /* EJERCICIO 4.1 */

/*
* El numero de veces que aparece cualquier elemento en la lista vacia
* es 0
*/

elem_count(_, [], 0).
/*
* Un elto X aparece n veces en una lista L cuyo primer elto es X si
* aparece 1 vez mas que en el resto de L
*/
elem_count(X, [X|Rs], Xn) :-
    elem_count(X, Rs, Num),
    Xn is Num + 1.
/*
* Un elto X aparece n veces en una lista L cuyo primer elto no es X (si lo
* fuera, se hubiera usado la regla anterior) si aparece n veces en el
* resto de L
*/
elem_count(X, [Y|Rs], Xn) :-
    X\=Y,
    elem_count(X, Rs, Xn).

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

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

EJERCICIO 4.2

PSEUDOCÓDIGO

Entrada: L1 elementos a contar, L2 lista donde buscar elementos, L3 lista con la cuenta de las veces que aparece cada elemento de L1 en L2
Salida: T si L3 lleva la cuenta de cuántveces aparece en L2 cada elemento de L1
Procesamiento:
Si L1 está vacía y L3 está vacía: T
Else
Si el 1er elto de L3 es de la forma '1er elto X de L2 + un número N
y
Los demás eltos de L3 llevan la cuenta de las apariciones en L2 de los demás elementos de L1 : T
Else: False

CÓDIGO

```
/* EJERCICIO 4.2*/
/* Si no hay elementos que buscar, la lista con el numero de veces
que aparece cada elemento esta vacia*/
list_count([], [_|_], []).
/* Si N es el num de veces que aparece X en L, y Z lleva la cuenta de
las apariciones de los elementos de Y en L*/
list_count([X|Y], L, [X-N|Z]):-
    elem_count(X, L, N),
    list_count(Y, L, Z).
/*
?- 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

PSEUDOCÓDIGO

Entrada: Una lista L de tuplas cuyo segundo elemento debe ser un número, y un elemento cualquiera X

Salida: True si el segundo argumento es una lista que contenga los mismos elementos que la lista recibida como primer argumento, pero ordenados en orden ascendente según el segundo.

Procesamiento:

1. Si la lista L contiene dos o más elementos, devolvemos true si su segundo argumento X es el resultado de ordenar L sin el primer elemento (continuar con la recursion) y después insertar, utilizando el código del ejercicio 3, el primer elemento de L.
2. Si la lista L solo contiene un elemento, devolvemos true si el segundo argumento es igual que L
3. Si la ista L está vacía, ya está ordenada, así que devolveremos true si el segundo elemento es una lista vacía también.

CÓDIGO

```
/* Ejercicio 5 */
/* La version ordenada de una lista vacia es la propia lista */
sort_list([], []).
/* Si la lista solo tiene un elemento ya esta ordenada */
sort_list([C-A], [C-A]).
/* La version ordenada de una lista con dos elementos o mas
* es la resultante de insertar (mediante la funcion insert del
* ejercicio 3) el primer elemento de la misma
* en el lugar correspondiente del resto ordenado de la lista.
* (continuando con la recursion)
*/
sort_list([C-A|[D-B|R1]], X) :-
        sort_list([D-B|R1], L),
        insert([C-A], L, X).
```

COMENTARIOS

EJERCICIO 6

PSEUDOCÓDIGO

Entrada: Una lista L compuesta por tuplas compuestas por un elemento y un número, y un elemento cualquiera.
Salida: True si el segundo argumento es el resultado de construir un árbol a partir de la lista L como se indica en el enunciado (o como resulta del pseudocódigo)
Procesamiento:
1. Si L contiene un único elemento X el segundo argumento tendra que ser un árbol de un único nodo cuyo campo info sea el primer elemento de la tupla de X
2. Si L contiene dos o más elementos, el segundo argumento tendrá que ser un árbol cuyo campo info sea 1, cuyo brazo izquierdo sea el resultado de construir un árbol con el primer elemento de L, y cuyo brazo derecho sea el resultado de construir un árbol con el resto de L.

CÓDIGO

```
/* Ejercicio 6 */
/* El arbol asociado a una lista con dos o mas elementos
*  sera el que tenga como
* campo 'info' un 1, como hijo izquierdo el resultado
* de generar un arbol solo con el elemento actual
* y como hijo derecho el resultado de continuar
* la recursion con el resto de elementos de la lista */
build_tree([A-B|R1], tree(1, Y, X)) :-
    build_tree(R1, X),
    build_tree([A-B], Y).
/* El arbol asociado a una lista de un unico elemento
* es el que lleva como campo 'info' el propio elemento
* y como hijos izquierdo y derecho nil */
build_tree([A-_], tree(A, nil, nil)).
```

COMENTARIOS

Si la lista está vacía, no dará lugar a ningún árbol y se devolverá false.

EJERCICIO 7

EJERCICIO 7.1

PSEUDOCÓDIGO

Entrada: Un elemento X, una lista L, y un árbol T
Salida: true si L es el resultado de codificar X a partir del árbol T como se indica en el enunciado (o como surge de seguir el pseudocódigo)
Procesamiento:
1. Si el X está en el campo info del nodo izquierdo del árbol, L tendrá que contener únicamente un 0.
2. Si no, L tendrá que valer la concatenacion de la lista que contiene únicamente un 1 y el resultado de volver al paso 1 sustituyendo T por el brazo derecho de T, es decir, continuando la recursión por el resto del árbol.
3. Si T tiene como campo 'info' el elemento X, el segundo argumento tendrá que ser una lista vacía.

CÓDIGO

```
/* Ejercio 7.1 */
/* El elemento codificado de X en el arbol
* tree(X, nil, nil) es una lista vacia, ya que no
* tenemos que tomar ni hijo izquierdo ni hijo derecho
* (caso base de la recursion) */
encode_elem(X, [], tree(X, nil, nil)).
/* El elemento X codificado en el arbol, si no esta en el
* hijo izquierdo, sera un 1 seguido del resultado de
* continuar la recursion por el hijo derecho */
encode_elem(X, [1|R1], tree(1, _, RTree)) :-
    encode_elem(X, R1, RTree).
/* El elemento X codificado en un arbol de la forma
* tree(1, tree(X, nil, nil)), es decir, que su hijo
* izquierdo es un arbol cuyo campo 'info' es X
* es una lista que solo contiene un [0] */
encode_elem(X, [0], tree(1, tree(X, nil, nil), _)).
```

COMENTARIOS

Si la el elemento no está en el árbol, los intentos de búsqueda fracasarán y se devolverá false.

EJERCICIO 7.2

PSEUDOCÓDIGO

Entrada: Una lista L1, un elemento L2 y un árbol T
Salida: true si el segundo argumento es la lista que contiene todos los elementos de L1 codificados como en el apartado 7.1 a partir del arbol T, o false en otro caso
Procesamiento:
1. Si L1 contiene elementos, el segundo elemento tendrá que ser una lista L2 que tenga como primer elemento el resultado de codificar el primer elemento de L1 y como resto de elementos los del resultado de codificar el resto de L1, ambas codificaciones hechas sobre el árbol T. 2. Si la lista L1 está vacía, el segundo argumento tendrá que ser otra lista vacía.

CÓDIGO

```
/* Ejercicio 7.2 */
/* El resultado de codificar una lista vacia
* es una lista vacia sin importar el arbol */
encode_list([], [], _).
/* El resultado de codificar una lista no vacia
* es una lista que contiene como primer elemento
* la codificacion del primer elemento de la lista
* recibida como argumento
* y como resto de elementos el resultado de continuar
* la recursion por el resto de la lista recibida
* como argumento */
encode_list([X|R], [EncX|Recursion], L) :-
       encode_elem(X, EncX, L),
       encode_list(R, Recursion, L).
```

COMENTARIOS

EJERCICIO 8

PSEUDOCÓDIGO

Entrada: L lista en claro y Lresult lista codificada
Salida: T si Lresult es L codificada
Procesamiento:
Si Lresult es L codificada con un árbol A
y
A proviene de una lista Linv
y
Linv es una lista Lsort invertida
y
Lsort contiene las letras del diccionario de mayor a menor frecuencia de aparición en L
: devuelve T

CÓDIGO

```
/* EJERCICIO 8*/

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]).


/*
* Para que Lresult sea [X|Y] (= L) codificada, necesitamos que
*   1. Lresult sea el resultado de codificar L con un arbol Ltree, y
*   2. Que Ltree este construido a partir de una lista Linv que ordene de
* mayor a menor frecuencia las letras del diccionario que se encuentren en L.
*
* Esta ultima condicion se traduce en que
*   1. Linv sea la inversion de una lista Lsort,
*   2. Donde Lsort contiene ordenadas de menor a mayor frecuencia las letras
* del diccionario Dict segun aparecen en L
*/
encode([X|Y], Lresult) :-
    dictionary(Dict),
    list_count(Dict, [X|Y], Lcount),
    sort_list(Lcount, Lsort),
    invierte(Lsort, Linv),
    build_tree(Linv, Ltree),
    encode_list([X|Y], Lresult, Ltree).
/*
encode([i,a],X).
X = [[0], [1, 0]] ;
false.
```

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

COMENTARIOS

Este predicado podría haberse simplificado de la siguiente manera: como el predicado *sort-list* nos da el orden de frecuencias de menor a mayor, luego nos vemos obligados a usar el predicado *invierte* para ordenarla de mayor a menor frecuencia. Hemos usado estos dos predicados por ser los que ya teníamos codificados, pero alterando la regla

```
insert([C-A], [D-B|R1], X) :-
    A=<B, concatena([C-A],
    [D-B|R1], X).
```

de manera que, en vez de A=<B, tuviéramos B<=A, podríamos ahorrarnos la inversión de la lista.