Universidad Autónoma

Inteligencia Artificial

Práctica 3

Memoria

Autores:
Adrián Fernández
Santiago González-Carvajal

Pareja 7
Grupo 2302

16 de abril de 2018

Índice

1. Prolog básico

1.1. Comprobar si un elemento pertenece a una colección

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
```

Pseudocódigo

```
Input: X (elemento a buscar), L (coleccion de elementos)
Output: true si X pertenece a L, false en caso contrario

Process:
pertenece_m(X, L):
    if list(L):
        if empty(L):
            return false
        else:
            F <-- pop(L)
            return or(pertenece_m(X, F),
                      pertenece_m(X, L))
    else:
        return X = L
```

Código

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

Comentarios

Para el caso base, es condición necesaria que el elemento a evaluar no sea de tipo lista. Así se evita que se evalúen listas como elementos. El resto de la implementación es igual a la de *pertenece*, salvo que también se evalúa el primer elemento de la colección como si fuera una lista.

1.2. Invertir listas

Batería de pruebas

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

Pseudocódigo

```
Input: L (lista izquierda), R (lista derecha)
Output: true si L y R son inversas, false en caso contrario

Process:
invierte(L, R):
    if and(empty(L), empty(R)):
        return true
    else:
        F <— pop(L)
        return and(invierte(L, LI)
                   concatena(LI, F, R))
```

Código

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

invierte([], []).
invierte([X|L], R) :- invierte(L, LI), concatena(LI, [X], R).
```

Comentarios

El predicado *invierte* consiste en una llamada recursiva a sí misma (omitiendo el primer elemento), concatenando el resultado obtenido con el primer elemento mediante el predicado *concatena*.

1.3. Insertar en una lista de pares ordenados

Batería de pruebas

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

Pseudocódigo

```
Input:  X–P (par ordenado, X es etiqueta y P es el orden),
        L (lista de partes ordenados), R (lista resultado)
Output: true si R es el resultado de insertar el par X–P en L,
        false en caso contrario

Process:
insert(X–P, L, R):
    if empty(L):
        return X–P = R
    else:
        Y–Q <— pop(L)
        if P <= Q:
            return concatena([X–P], [Y–Q] + L, R)
        else:
            return and(insert([X–P], L, I), concatena([Y–Q], I, L))
```

*Nota: [X] significa lista con elemento único X.

Código

```
insert([X–P], [], [X–P]).
insert([X–P], [Y–Q|Rs], R) :- P=<Q, concatena([X–P], [Y–Q|Rs], R).
insert([X–P], [Y–Q|Rs], R) :- P>Q, insert([X–P], Rs, I),
                               concatena([Y–Q], I, R).
```

Comentarios

El predicado *insert* consiste en una llamada recursiva hasta que encuentra un elemento mayor o igual al evaluado o hasta que llega al final de la lista. Una vez en esa posición, concatena el elemento con el resto de la lista mediante el predicado *concatena*.

1.4. Contar elementos en una 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
```

Pseudocódigo

```
Input:  X (elemento), L (lista),
        Xn (numero de veces que se repite X en L)
Output: true si Xn es el numero de repeticiones de X en L,
        false en caso contrario

Process:
elem_count(X, L, Xn):
    if empty(L):
        return Xn = 0
    else:
        F <-- pop(L)
        if X = F:
            return and(elem_count(X, L, C), Xn = C+1)
        else:
            return elem_count(X, L, Xn)
```

Código

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

Comentarios

El predicado *elem_count* consiste en una recursión en la cual se incrementa el valor de Xn cada vez que se encuentra X en L.

1.5. Contar elementos de una lista en otra lista

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
```

Pseudocódigo

```
Input:  L1 (lista de elementos a evaluar),
        L2 (lista en la que se buscan los elementos),
        R (lista de pares elemento de L1 - numero de apriciones en L2)
Output: true si R es la lista resultado, false en caso contrario

Process:
list_count(L1, L2, R):
    if and(empty(L1), empty(R)):
        return true
    else:
        Y <-- pop(L1)
        X-Xn <-- pop(L3)
        if X = Y:
            return and(elem_count(X, L2, Xn),
                       list_count(L1, L2, L3))
```

Código

```
list_count([], _, []).
list_count([X|R1], L2, [X-Xn|R3]) :- elem_count(X, L2, Xn),
                                     list_count(R1, L2, R3).
```

Comentarios

El predicado *list_count* se sirve de la recursión para contar cada elemento de la lista **L1** mediante una llamada a el predicado *elem_count*.

1.6. Ordenar listas

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
```

Pseudocódigo

```
Input:  L1 (lista de pares a ordenar), L2 (lista de pares ordenados)
Output: true si L2 es L1 ordenada, false en caso contrario

Process:
sort_list(L1, L2):
    if and(empty(L1), empty(L2)):
        return true
    else:
        X–P <— pop(L1)
        return and (sort_list(L1, SL)
                    insert([X–P], SL, L2))
```

*Nota: [X] significa lista con elemento único X.

Código

```
sort_list([], []).
sort_list([X–P|Rs], SL) :- sort_list(Rs, SLr),
                           insert([X–P], SLr, SL).
```

Comentarios

El predicado *sort_list* se basa en *insert* para insertar cada elemento en orden partiendo de una lista vacía.

2. Codificar con árboles de Huffman

2.1. Crear árboles de Huffman a partir de listas

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))))
false
```

Pseudocódigo

```
Input:  L (lista de elementos), T (arbol construido a partir de L)
Output: true si T es el arbol construido a partir de L,
        false en caso contrario

Process:
build_tree(L, T):
   X–P <— pop(L)
   if empty(L):
      return T = tree(X, nil, nil)
   else:
      Y–Q <— pop(L)
      if empty(L):
         return and(build_tree([F], tree_left(T))
                    build_tree([S], tree_right(T)))
      else:
         return and(build_tree([F], tree_left(T))
                    build_tree([S] + L, tree_right(T)))
```

*Nota: [X] significa lista con elemento único X.

Código

```
build_tree([], nil).
build_tree([X–_], tree(X, nil, nil)).
build_tree([F, S|Rs], tree(1, Ti, Td)) :- Rs = [],
                                          build_tree([F], Ti),
                                          build_tree([S], Td).
build_tree([F, S|Rs], tree(1, Ti, Td)) :- Rs \= [],
                                          build_tree([F], Ti),
                                          build_tree([S|Rs], Td).
```

Comentarios

Para implementar *build_tree*, hace falta diferenciar entre el caso en el que la lista tiene 1, 2 o más elementos. En el caso en el que tiene 1, el árbol deberá ser de la forma **tree(X, nil, nil)**. Si tiene 2, el árbol será **tree(1, tree(X, nil, nil), tree(Y, nil, nil))**. Finalmente, en caso de que haya más de 2 elementos, se sealizará el paso recursivo, donde se construye el árbol izquierdo con el primer elemento y el derecho con el resto de elementos.

2.2. Codificar un elemento a partir de un árbol

Batería de pruebas

```
T = 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, T).
X = [0]
false
?- encode_elem(d, X, T).
X = [1, 1, 1]
false
```

Pseudocódigo

```
Input:  X (elemento a codificar), L (elemento codificado),
        T (arbol de codificacion)
Output: true si Y es X codificado mediante T, false en caso contrario

Process:
encode_elem(X, L, T):
    if T = tree(X, nil, nil):
        return empty(L)
    else:
        Ti <-- tree_left(T)
        if encode_elem(X, [], Ti):
            return L = [0]
        else:
            Td <-- tree_right(T)
            F <-- pop(L)
            if encode_elem(X, L, Td):
                return F = 1
            else:
              return false
```

*Nota: [X] significa lista con elemento único X.

Código

```
encode_elem(X, [], tree(X, nil, nil)).
encode_elem(X, [0], tree(1, Ti, _)) :- encode_elem(X, [], Ti).
encode_elem(X, [1|Rs], tree(1, _, Td)) :- encode_elem(X, Rs, Td).
```

Comentarios

Para implementar el predicado *encode_elem*, se asume que si el árbol **T** es una hoja con elemento **X**, entonces la lista **L** debe estar vacía. Partiendo de la lista vacía, se va añadiendo al principio un 0 si el elemento se ha encontrado a la izquierda, o un 1 si se ha encontrado ala derecha.

2.3. Codificar una lista a partir de un árbol

Batería de pruebas

```
T = tree(1, tree(a, nil, nil),
            tree(1, tree(b, nil, nil),
                    tree(1, tree(c, nil, nil),
                            tree(d, nil, nil))))

?- encode_list([a,d,a], X, T).
X = [[0], [1, 1, 1], [0]]
false
?- encode_list([q], X, T)).
false
```

Pseudocódigo

```
Input:  L1 (lista de elementos),
        L2 (lista de codificaciones de los elementos de L1),
        T (arbol de codificacion)
Output: true si L2 contiene las codificaciones de L1 por T,
        false en caso contrario

Process:
encode_list(L1, L2, T):
    if and(empty(L1), empty(L2)):
        return true
    else:
        X <-- pop(L1)
        Y <-- pop(L2)
        return and(encode_elem(X, Y, T)
                   encode_list(L1, L2, T))
```

Código

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

Comentarios

El predicado *encode_list* se sirve de la recursión para codificar cada elemento de la lista **L1** mediante una llamada a el predicado *encode_elem*.

2.4. Codificar una palabra a partir de la frecuencia de sus letras

Batería de pruebas

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

Pseudocódigo

```
Input:  L1 (lista de letras a codificar), L2 (lista de codificaciones)
Output: true si L2 contiene las codificaciones de L1,
        false en caso contrario

Process:
encode(L1, L2):
   return and(dictionary(D), list_count(D, L1, LC), sort_list(Lc, SL),
              invierte(SL, IL), build_tree(IL, T),
              encode_list(L1, L2, T))
```

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, LC),
                  sort_list(LC, SL), invierte(SL, IL),
                  build_tree(IL, T), encode_list(L1, L2, T).
```

Comentarios

Para implementar el predicado *encode*, hay que contar el número de cada letra del diccionario que hay en **L1**, ordenar la lista de letras según el número de repeticiones de mayor a menor y construir el árbol de codificación con esa lista. Para realizar esa tarea, debemos utilizar los predicados *list_count*, *sort_list*, *invierte*, *build_tree* y *encode_list*.