Inteligencia Artificial 2017-2018

Práctica 3: Prolog

Grupo: 2213

Celia San Gregorio Moreno

Álvaro Martínez Morales

Índice

[Ejercicio 1: predicado ‘pertenece’ 3](#_Toc511756377)

[Código 3](#_Toc511756378)

[Comentario 3](#_Toc511756379)

[Ejercicio 2: predicado ‘invierte’ 4](#_Toc511756380)

[Código 4](#_Toc511756381)

[Comentario 4](#_Toc511756382)

[Ejercicio 3: predicado ‘insert’ 5](#_Toc511756383)

[Código 5](#_Toc511756384)

[Comentario 5](#_Toc511756385)

[Ejercicio 4: conteo de elementos 6](#_Toc511756386)

[Ejercicio 4.1: predicado ‘elem\_count’ 6](#_Toc511756387)

[Código 6](#_Toc511756388)

[Comentario 6](#_Toc511756389)

[Ejercicio 4.2: predicado ‘list\_count’ 6](#_Toc511756390)

[Código 6](#_Toc511756391)

[Comentario 6](#_Toc511756392)

[Ejercicio 5: predicado ‘sort\_list’ 7](#_Toc511756393)

[Código 7](#_Toc511756394)

[Comentario 7](#_Toc511756395)

[Ejercicio 6: predicado ‘build\_tree’ 8](#_Toc511756396)

[Código 8](#_Toc511756397)

[Comentario 8](#_Toc511756398)

[Ejercicio 7: codificación de elementos mediante Árboles de Huffman 9](#_Toc511756399)

[Ejercicio 7.1: predicado ‘encode\_element’ 9](#_Toc511756400)

[Código 9](#_Toc511756401)

[Comentario 9](#_Toc511756402)

[Ejercicio 7.2: predicado ‘encode\_list’ 9](#_Toc511756403)

[Código 9](#_Toc511756404)

[Comentario 9](#_Toc511756405)

[Ejercicio 8: predicado ‘encode’ 10](#_Toc511756406)

[Código 10](#_Toc511756407)

[Comentario 10](#_Toc511756408)

# Ejercicio 1: predicado ‘pertenece’

### Código

%----------------%

% Ejercicio 1 %

%----------------%

pertenece\_m(X, [X|\_]) :- X \= [\_|\_].

pertenece\_m(X, [L|Rs]) :- pertenece\_m(X, L); pertenece\_m(X, Rs).

### Comentario

Este ejercicio solicitaba la implementación del predicado *pertenece\_m(X, L)*, el cual se satisface cuando el elemento X está incluido entre los elementos de L.

* En su consecución, en un primer lugar, se ha creado un caso base, el cual se satisface si encontramos equivalencia entre X y el primer elemento del L que está siendo evaluado, siempre y cuando el X evaluado no se esté considerando como una lista, para asegurarnos que este predicado no se intenta satisfacer cuando nos encontramos con una sub-lista en L.
* En segundo lugar, hay un segundo predicado el cual se satisface si el primer elemento de L es satisfactible mediante el primer predicado o este segundo predicado aplicado a X y el resto de la lista como segundo argumento.

De esta manera, fundamentalmente, se recorre la lista y posibles sub-listas mediante el primer predicado, y evalúa cada elemento del listado mediante el primer predicado.

# Ejercicio 2: predicado ‘invierte’

### Código

%----------------%

% Ejercicio 2 %

%----------------%

concatena([], L, L).

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

invierte([], []).

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

### Comentario

En este ejercicio se pedía implementar el predicado *invierte(L, R)*, que se satisface cuando la lista R contiene los elementos de L en orden inverso.

En primer lugar, diseñamos un caso base con la lista vacía: siempre que *invierte* se aplique sobre una lista vacía, el inverso de dicha lista será también una lista vacía.

La segunda regla establece que, para una lista con X como primer elemento y R como resto, L será su lista invertida si se cumplen dos condiciones:

* Que la variable L1 sea el inverso del resto R. Este predicado con caso recursivo recorrerá cada elemento de R hasta llegar a la lista vacía, momento en el que se validará la regla *invierte([], []).*
* Que la lista L sea el resultado de concatenar cada elemento X de la lista [X|R] al final de L1, de forma que L contenga los mismos elementos que [X|R] pero en orden inverso.

# Ejercicio 3: predicado ‘insert’

### Código

%----------------%

% Ejercicio 3 %

%----------------%

insert([X-P], [], [X-P]).

insert([X-P], [A-Q|Ls], R) :- P=<Q, concatena([X-P], [A-Q|Ls], R).

insert([X-P], [A-Q|Ls], [A-Q|Rs]) :- P>Q, insert([X-P], Ls, Rs).

### Comentario

Este ejercicio solicitaba la implementación del predicado *insert(X, L, R)*, el cual deberá satisfacerse con un R que suponga la inserción ordenada de X en L, una lista previamente ordenada, con el factor adicional de que debe estar enfocado a que el elemento X sea un doblete de elementos en el formato ‘*X-A’*, teniendo a A como factor de ordenación.

* Esto se ha llevado a cabo, en primer lugar, mediante el caso base de que, en la eventualidad de una L como lista vacía, el resultado de la inserción del elemento es una lista con el propio elemento.

En segundo lugar, hay dos casos relativamente similares; ambos casos cuentan con un L que es un listado de elementos.

* Sin embargo, en el primero de ambos casos, el primer elemento de L es el elemento posterior en la ordenación respecto de X, siendo su A igual o menor que la del primer elemento de L. En ese caso, la inserción se satisface si el resultado de la inserción es la concatenación de X, el primer elemento de L y el resto de L (osease, a efectos prácticos, el L siendo actualmente evaluado al completo).
* En el segundo de los casos, el elemento X es mayor al primer elemento de L, y, por tanto, la inserción sólo se podrá satisfacer si X se puede insertar en el resto de L.

# Ejercicio 4: conteo de elementos

## Ejercicio 4.1: predicado ‘elem\_count’

### Código

%------------------%

% Ejercicio 4.1 %

%------------------%

elem\_count(\_, [], 0).

elem\_count(X, [X|Ls], C1) :- elem\_count(X, Ls, C), C1 is C + 1.

elem\_count(X, [Y|Ls], C1) :- X\=Y, elem\_count(X, Ls, C1).

### Comentario

## Ejercicio 4.2: predicado ‘list\_count’

### Código

%------------------%

% Ejercicio 4.2 %

%------------------%

list\_count([], [\_|\_], []).

list\_count([X|Ls], L2, [X-C|Rs]) :- elem\_count(X, L2, C), list\_count(Ls, L2, Rs).

### Comentario

# Ejercicio 5: predicado ‘sort\_list’

### Código

%----------------%

% Ejercicio 5 %

%----------------%

sort\_list([],[]).

sort\_list([X-P|L1], L3) :- sort\_list(L1, L2), insert([X-P], L2, L3).

### Comentario

En este ejercicio hemos implementado dos reglas para el predicado *sort\_list(L1, L2)*, que se satisface si la lista L2 contiene los pares de elementos de L1 ordenados por posición.

Diseñamos un caso base que se satisface cuando una lista vacía tiene como lista ordenada la misma lista vacía, ya que no hay elementos que comparar y colocar en sus respectivas posiciones.

La segunda regla indica que L3 es la lista de pares ordenados a partir de una lista [X-P|L1], cuyo primer elemento es el par X-P y su resto es L1, si se cumplen dos condiciones:

* Que la variable L1 sea el inverso del resto L2. Este predicado con caso recursivo recorrerá cada elemento de L1 hasta llegar a la lista vacía, momento en el que se validará la regla *sort\_list([], []).*
* Que la lista L3 sea el resultado de insertar el par X-P en la posición correspondiente de L2. El predicado *insert* se encargará de comprobar si L3 contiene el par X-P ordenado de acuerdo a su posición en la lista L2.

# Ejercicio 6: predicado ‘build\_tree’

### Código

%----------------%

% Ejercicio 6 %

%----------------%

build\_tree([X-\_], tree(X, nil, nil)).

build\_tree([X-P|RL], tree(1, L1, L2)) :- RL \= [], build\_tree([X-P], L1), build\_tree(RL, L2).

### Comentario

A partir de una lista de pares de elementos ordenados, se han implementado dos reglas con el predicado *build\_tree(List, Tree)* que permiten transformar dicha lista a un árbol de Huffman.

Para ello definimos un caso base que, a partir de una lista con un único par de elementos X-\_ (siendo \_ cualquier entero), construye un árbol con un nodo hoja. Este nodo hoja tiene como campo *Info* el elemento X. Los campos *Left* y *Right* serán nil, ya que no tiene sucesores.

La segunda regla establece que, a partir de una lista de pares [X-P|RL], con X-P como primer elemento y RL como resto, su árbol resultante es tree(1, L1, L2) si se cumplen estas condiciones:

* El resto RL no es una lista vacía.
* L1 se corresponde con el nodo hoja creado a partir del elemento X-P de la lista.
* L2 es el árbol resultante de iterar sobre los elementos de RL. Contendrá nodos intermedios marcados como 1 en su campo *Info*, tree(X-P, nil, nil) como campo *Left,* y el resto del árbol como campo *Right*.

# Ejercicio 7: codificación de elementos mediante Árboles de Huffman

## Ejercicio 7.1: predicado ‘encode\_element’

### Código

%-----------------%

% Ejercicio 7.1 %

%-----------------%

encode\_elem(\_, [], tree(\_, \_, nil)).

encode\_elem(X, R, tree(\_, tree(V, \_, \_), \_)) :- X = V, R = [0].

encode\_elem(X, R, tree(\_, \_, tree(V, \_, \_))) :- X = V, R = [1].

encode\_elem(X, [R1|R2], tree(\_, \_, tree(V, N1, N2))) :- X \= N2, N2 \= nil, R1 = 1, encode\_elem(X, R2, tree(V, N1, N2)).

### Comentario

[TODO]

## Ejercicio 7.2: predicado ‘encode\_list’

### Código

%-----------------%

% Ejercicio 7.2 %

%-----------------%

encode\_list([], [], \_).

encode\_list([X|RL], [R1|R2], T) :- encode\_elem(X, R1, T), encode\_list(RL, R2, T).

### Comentario

[TODO]

# Ejercicio 8: predicado ‘encode’

### Código

%----------------%

% Ejercicio 8 %

%----------------%

dictionary(X) :- X = [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, RLC), sort\_list(RLC, RSL), invierte(RSL, RI), build\_tree(RI, T), encode\_list(L1, L2, T).

### Comentario