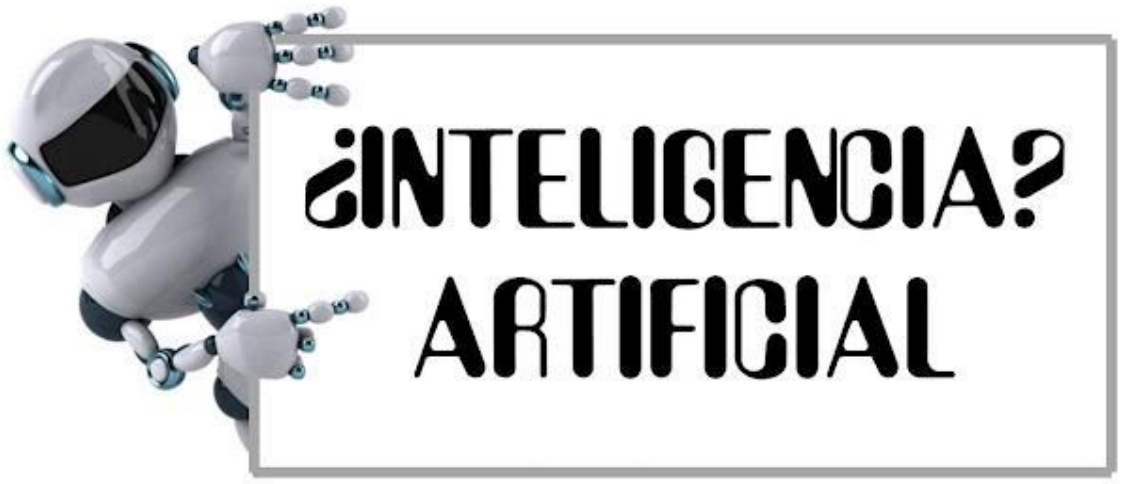
MEMORIA

Práctica 3: Prolog



Andrés Salas Peña
Ricardo Riol Fernández

Grupo 2301 - Pareja 08
Pr3 - Inteligencia Artificial

1. Introducción

En esta tercera práctica, nos iniciamos en un nuevo lenguaje como es SWI Prolog (**Pro**grammation en **Log**ique). Por tanto, como siempre que empezamos un nuevo lenguaje, los inicios fueron difíciles.

EL objetivo más funcional de esta tercera práctica es desarrollar un predicado que permita codificar unos ciertos caracteres. Para llegar a esto, la práctica se divide en 8 ejercicios en los que se nos piden determinados predicados que serán útiles para lograr el objetivo mencionado que se termina de implementar en el ejercicio 8. Además, dado nuestro nulo conocimiento previo del lenguaje, en el ejercicio 1 se nos facilita el código de una función para que sepamos cómo comenzar con el lenguaje.

2. Código

2.1 Ejercicio 1

Pseudocódigo

PERTENECE

Entrada:
X: elemento a buscar.
L: lista donde buscar el elemento.
Salida:
Tantos trues como veces pertenezca el elemento a la lista o cada uno de los valores que va tomando el elemento.
Procesamiento:
Se divide en: -Caso base en el que X y el primer elemento de la lista son el mismo
-Caso recursivo que indica si X pertenece al resto de la lista.

PERTENECE_M

Entrada:
X: elemento a buscar
L: lista con sublistas.
Salida:
Tantos trues como veces pertenezca el elemento a la lista o sublista o cada uno de los valores que va tomando el elemento.
Procesamiento:

Se divide en caso base en el que X pertenece al primer elemento (no sublista) de la lista, caso recursivo en el que se observa si X pertenece a la primera sublista y otro caso recursivo en el que se analiza si X pertenece al resto de la lista.

Código

PERTENECE

```
pertenece(X, [X|_]).
pertenece(X, [_|Rs]) :- pertenece(X, Rs).
```

PERTENECE_M

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

Pruebas

PERTENECE

```
pertenece(1, [2, 1, 3, 1])
pertenece(X, [2, 1, 3, 1])
pertenece(1, L)
```

PERTENECE_M

```
pertenece_m(X, [2,[1,3],[1,[4,5]]])
pertenece_m([1,3], [2,[1,3],[1,[4,5]]])
```

Comentarios

El primero de los predicados nos lo dan implementado, el segundo se basa en la misma idea del primero y se tiene en cuenta que X no sea una sublista con la indicación dada para ello.

2.2 Ejercicio 2

Pseudocódigo

CONCATENA

Entrada:
L1: lista 1.
L2: lista 2.
Salida:
L3: lista que es la concatenación de L1 y L2.
Procesamiento:
Maneja el caso base de que la primera lista se dé vacía en cuyo caso L3 será L2. Y un caso recursivo en el que se va trabajando con cada uno de los primeros elementos de L1.

INVIERTE

Entrada:

L: lista a invertir.

Salida:

R: lista invertida

Procesamiento:

Caso base de si es vacía (entonces R también será vacía) y caso recursivo en el que se va invirtiendo llamando a los elementos restantes primero, y al primer elemento, el último.

Código

CONCATENA

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

INVIERTE

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

Pruebas

CONCATENA

```
concatena([], [1, 2, 3], L)
concatena([1, 2, 3], [4, 5], L)
```

INVIERTE

```
invierte([1, 2], L)
invierte([], L)
invierte([1, 2, 3, 4, 5], L)
```

Comentarios

En el enunciado se recomienda hacer un predicado concatena y por eso se realiza. Además es muy útil para nuestro predicado invierte que se encarga de reordenar los elementos de una lista.

2.3 Ejercicio 3

INSERT

Pseudocódigo

Entrada:

X-P: par de elementos a insertar.

L: lista de pares ordenados.

Salida:

R: lista con el par insertado en la posición p y desplazados los pares de L.

Procesamiento:

Se realiza un caso base en caso de ser la lista de pares vacía y un caso recursivo en el que se va cogiendo el segundo valor de cada par y se compara con el del par de elementos a insertar.

Código

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

Pruebas

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

Comentarios

Este ejercicio se encarga de insertar un par de elementos en el lugar correcto de una lista ordenada pero si esta no está ordenada, no la reordena.

2.4 Ejercicio 4

Ejercicio 4.1

ELEM_COUNT

Pseudocódigo

Entrada:
X: elemento a contar.
L: lista en la que buscar el elemento.
Salida:
Xn: número de veces que se encuentra X en la lista L.
Procesamiento:
Si el elemento o la lista son vacíos entonces aparecerá cero veces. Si no, se hace recurrencia aumentando en 1 el contador de Xn cada vez que el elemento coincide con el de la lista.

Código

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

Pruebas

```
elem_count(b, [b,a,b,a,b], Xn)
elem_count(a, [b,a,b,a,b], Xn)
```

Comentarios

Este predicado es una especie de auxiliar de la del 4.2 para realizar toda su tarea con cada uno de los elementos de una lista. Se encarga de contar el número de veces que aparece el elemento y de devolverlo en una lista.

Ejercicio 4.2

LIST_COUNT

Pseudocódigo

Entrada:
L1: lista de elementos a contar.
L2: lista en la que se contarán los elementos.
Salida:
L3: lista con los pares [elemento a contar-número de veces que aparece].
Procesamiento:
Caso base de que la lista de elementos a buscar sea vacía.
Caso recursivo en el que se llama al predicado anterior del 4.1 para cada elemento de la lista.

Código

```
list_count([], _, []).
list_count([X|RsX], Y, Xn):- elem_count(X, Y, Zn), concatena([X-Zn], L, Xn), list_count(RsX, Y, L).
```

Pruebas

```
list_count([b], [b,a,b,a,b], Xn)
list_count([b,a], [b,a,b,a,b], Xn)
list_count([b,a,c], [b,a,b,a,b], Xn)
```

Comentarios

Función principal del ejercicio que devuelve una lista con los pares [elemento-número de apariciones].

2.5 Ejercicio 5

SORT_LIST

Pseudocódigo

Entrada:
L1: lista de elementos a ordenar.
Salida:

L2: lista de elementos ordenada.

Procesamiento:

Caso base de que la lista de elementos sea nula, en cuyo caso estarán ordenados.
Caso recursivo en el que se reordenan los elementos de la lista en una auxiliar.

Código

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

Pruebas

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

Comentarios

Para este ejercicio se llama al predicado insert del ejercicio 3 para poder insertar los elementos de una lista en otra auxiliar en su posición corespondiente.

2.6 Ejercicio 6

BUILD_TREE

Pseudocódigo

Entrada:

List: lista de pares de elementos ordenados.

Salida:

Tree: árbol simplificado de Huffman creado a partir de la lista dada.

Procesamiento:

Caso base de lista con un elemento.
Caso recursivo en el que se construye el árbol izquierdo y se llama al resto de elementos de la lista.

Código

```
build_tree([X-_], tree(X, nil, nil)).
build_tree([X-_|Rs], tree(1, T1, T2)):- T1 = tree(X,nil,nil), build_tree(Rs, T2).
```

Pruebas

```
build_tree([p-0, a-6, g-7, p-9, t-2, 9-99], X)
build_tree([p-55, a-6, g-7, p-9, t-2, 9-99], X)
build_tree([p-55, a-6, g-2, p-1], X)
build_tree([a-11, b-6, c-2, d-1], X)
```

Comentarios

El predicado usa fuertemente una idea simplona del árbol de Huffman pedido.

2.7 Ejercicio 7

Ejercicio 7.1

ENCODE_ELEM

Pseudocódigo

Entrada:
X1: elemento a codificar.
Tree: estructura de árbol según la cual se codificará.
Salida:
X2: versión codificada del elemento X1 basado en el árbol Tree.
Procesamiento:
Caso base de que el elemento a codificar esté en el nodo izquierdo del padre.
Caso base de que el elemento se encuentre en el nodo derecho del padre (un solo elemento, el último del árbol).
Caso recursivo llamando a la parte del árbol que falta por codificar.

Código

```
encode_elem(X1,X2,tree(1, tree(X1, nil,nil), _)):- concatena([], [0], X2).
encode_elem(X1,X2,tree(1,_, tree(X1, nil, nil))):- concatena([], [1], X2).
encode_elem(X1,X2,tree(1, _ , RS)) :- encode_elem(X1, AUX, RS), concatena([1], AUX, X2).
```

Pruebas

```
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)))))
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)))))
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)))))
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)))))
```

Comentarios

Al igual que pasó en el ejercicio 4, este primer apartado es un subpredicado del apartado 7.2. Este ejercicio 7.1 se encarga solo de codificar un elemento ayudándose de la estructura del árbol de Huffman.

Ejercicio 7.2

ENCODE_LIST

Pseudocódigo

Entrada:
L1: lista a codificar.
Tree: estructura del árbol según la cual se codificará.
Salida:
L2: versión codificada de la lista L1 basada en el árbol Tree
Procesamiento:
Caso base de que el árbol o la lista esté vacía
Caso Recursivo pasando elemento a elemento de la lista L1 de elementos al predicado encode_elem, luego es una llamada a encode_list con los restantes.

Código

```
encode_list([], [], _).
encode_list([X1|RsX1], X2, Tree):- encode_elem(X1, X2Aux, Tree), concatena([X2Aux], L, X2), encode_list(RsX1, L, Tree).
```

Pruebas

```
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)))))
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)))))
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)))))
encode_list([a,b], X, tree(1, tree(a, nil, nil), tree(1, tree(b, nil, nil), tree(1, tree(c, nil, nil), tree(d, nil, nil)))))
```

Comentarios

Predicado principal del ejercicio que devuelve una lista llena de sublistas que contienen la ruta hasta cada elemento.

2.8 Ejercicio 8

ENCODE

Pseudocódigo

Entrada:
L1: lista a codificar.
Salida:
L2: versión codificada de la lista L1 basada en el predicado dictionary
Procesamiento:
Rutina de llamadas los predicados anteriores en el orden adecuado.

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(X),
   list_count(X, L1, LContada),
   sort_list(LContada, LOrdenada),
        invierte(LOrdenada, LInvertida),
        build_tree(LInvertida, Tree),
        encode_list(L1, L2, Tree).
```

Pruebas

```
encode([i,n,t,e,l,i,g,e,n,c,i,a,a,r,t,i,f,i,c,i,a,l], X)
encode([i,a], X)
encode([i,2,a], X)
```

Comentarios

El planteamiento para la realización de este ejercicio se realizó con papel y boli, haciendo una pila de lllamadas de atrás hacia delante para saber qué necesitábamos para cada predicado.
Primero usamos la lista diccionario para tener todos los caracteres que se podrán codificar, luego contamos el número de apariciones de cada elemento de diccionario en nuestra lista a codificar. Tras esto, ordenamos la lista de menor a mayor número de apariciones. Después, invertimos el orden de la lista para poder construir adecuadamente su árbol. Y, finalmente, llamamos al predicado encode_list.
Reseñar que el árbol que se crea también contendrá aquellos elementos que no aparecen en nuestra lista a codificar pero no influyen en el resultado ya que se crean más tarde que los que sí aparecen.

3. Conclusiones

En este último apartado, recogeremos nuestra experiencia personal durante la elaboración de la práctica centrándonos en qué hemos aprendido y qué nos ha costado más implementar.

En primer lugar y antes de comenzar a picar código, nos leímos la información facilitada en Moodle sobre el lenguaje así como los hipervínculos a los que se llamaba.

Por otro lado, en cuanto a nuestra metodología de trabajo, queremos reseñar que en lugar de descargarnos el lenguaje y su entorno de programación, decidimos usar la versión online de Swi Prolog. La decisión de usar esta metodología es que con el uso de la versión online, podemos trabajar de manera más cómoda ambos compañeros simultáneamente e ir viendo el código del otro. El único problema que plantea esta decisión, es que el fichero lo pueden encontrar el resto de alumnos de la asignatura por lo que decidimos camuflar el nombre del fichero para que fuese difícil de encontrar y borrarlo una vez finalizado.

Adentrándonos ya en el código, las dificultades encontradas fueron, como cabía esperar, entender la lógica del lenguaje. Esto hizo que nos costase arrancar con los dos primeros ejercicios pero tras haberlos realizado, los tres siguientes los codificamos muy rápidamente.

Los ejercicios 6 y 7.1 ya eran más complicados y nos llevaron más tiempo. Además, gracias a la realización de los mismos, aprendimos sobre la estructura de los árboles de Huffman.

Finalmente, el ejercicio 8, el que consideramos el objetivo final de la práctica, tuvo como dificultad la escasa explicación que se proporcionaba del mismo. Una vez entendido el objetivo del ejercicio, la codificación se basó en realizar llamadas al resto de funciones que habíamos implementado a lo largo de la práctica.

Con esta práctica, nos hemos acercado por primera vez a un lenguaje basado en lógica como es SWI Prolog. Esto ha hecho que nuestra experiencia con la lógica no sea ya solo desde un marco más teórico si no también práctico. Por último, queríamos reseñar que desde un punto de vista personal, la realización del ejercicio 8 nos pareció muy profunda porque nos muestra a dónde podemos llegar con este lenguaje y con los distintos predicados que nosotros mismos habíamos ido implementando.