Inteligencia Artificial
Práctica 3

Martín Selgas, Blanca (blanca.martins@estudiante.uam.es)
Villar Gómez, Fernando (fernando.villarg@estudiante.uam.es)
Grupo 2302 - Pareja 3

19/04/2018

Índice

1. Introducción

En el presente documento se detalla la realización de los problemas pertenecientes a la tercera práctica de Inteligencia Artificial. En ellos se requiere la resolución de determinadas tareas en el lenguaje declarativo Prolog, aportando el pseudo-código, la formalización en lógica de primer orden, la codificación, la documentación y los comentarios de las funciónes utilizadas para tales fines.

El desarrollo de la práctica se ha realizado a través de **SWI-Prolog** por línea de comandos en un sistema Linux. Los ejercicios están orientados a la codificación de mensajes en secuencias de bits con el fin de comprimir la información, de manera que la transmisión sea más rápida y eficiente.

2. Ejercicio 1

pertenece_m

PSEUDOCÓDIGO

```
Entrada: X (elemento)
         L (lista)
Salida:  true si X está contenido en L
         false en caso contrario.

Procesamiento:
         Si el primer elemento de la lista L es X:
             evalúa a true.
         en caso contario
             buscamos X en el primer elemento de la lista L
             buscamos X en el resto de la lista L
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- x: elementos
- l: listas

Funciones:

- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]

Predicados:

- $Igual^2$ [$Igual(x_1, x_2)$: T si $x_1 = x_2$]
- $pertenece\_m^2$ [$pertenece\_m(x, l)$: T si x pertenece a l]

$\forall x, l\ [Igual(x, Primero(l)) \implies pertenece\_m(x, l)]$

$\forall x, l\ [pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, Resto(l)) \implies pertenece\_m(x, l)]$

CÓDIGO

```
/********************************************
*    Exercise 1:
*
*    pertenece_m(X, L)
*      Predicate that performs the same evaluation that 'pertenece', but in
*      this case, L may contain sublists.
*
*/
```

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

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

COMENTARIOS:

- El predicado $Igual^2$ siempre es falso cuando se compara un elemento con una lista; nunca se compararán dos listas dado que el primer argumento coincide con el de $pertenece\_m^2$, que es siempre un elemento.

- El predicado $pertenece\_m^2$ amplía la funcionalidad de $pertenece^2$ implementando la recursividad dentro de los elementos de la lista. De esta forma, se añade una nueva regla, que comprueba si el elemento x está contenido dentro de cada una de las sublistas que componen la lista l en caso de que el objeto contenido en la primera posición de l no sea un elemento.

- Desde el predicado de lógica de primer orden:

  $\forall x, l\ [pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, Resto(l)) \implies pertenece\_m(x, l)]$

  Se pretende ilustrar la estructura de Prolog, donde no existe el or lógico. Por tanto, de este predicado se deben deducir las dos últimas reglas de la base de conocimiento del programa proporcionado al pasarlo a FNC.

  1) $\forall x, l\ [pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, Resto(l)) \implies pertenece\_m(x, l)]$

  Eliminación de a implicación:
  2) $\forall x, l\ [\neg(pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, Resto(l))) \lor pertenece\_m(x, l)]$

  Aplicación de las leyes de De Morgan:
  3) $\forall x, l\ [(\neg pertenece\_m(x, Primero(l)) \land \neg pertenece\_m(x, Resto(l))) \lor pertenece\_m(x, l)]$

  Propiedad distributiva:
  4) $\forall x, l\ [(\neg pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, l)) \land (\neg pertenece\_m(x, Resto(l)) \lor pertenece\_m(x, l))]$

  Eliminación del cuantificador universal:
  5) $(\neg pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, l)) \land (\neg pertenece\_m(x, Resto(l)) \lor pertenece\_m(x, l))$

  Eliminación de la conjunción:
  6) $\neg pertenece\_m(x, Primero(l)) \lor pertenece\_m(x, l)$
  7) $\neg pertenece\_m(x, Resto(l)) \lor pertenece\_m(x, l)$

  Por último, una vez tenemos el predicado en FNC, si aplicamos la definición de la implicación a las dos cláusulas resultantes, obtenemos:

  P1) $pertenece\_m(x, Primero(l)) \implies pertenece\_m(x, l)$
  P2) $pertenece\_m(x, Resto(l)) \implies pertenece\_m(x, l)$

  que corresponden con las dos últimas reglas del programa *pertenece_m*.

En cuanto al predicado:

$\forall x, l \ [Igual(x, Primero(l)) \implies pertenece\_m(x, l)]$

Se ve directamente que coincide con la primera instrucción en Prolog.

NOTA: A partir de este momento, todas las funciones que se van a formalizar con lógica de primer orden van a seguir el esquema siguiente:

$\forall var_1, var_2 \ [[\exists var_{aux} \ (cond_1(var_1, var_{aux}) \wedge cond_2(var_{aux}, var_2))] \implies predicate(var_1, var_2)]$

Después de aplicar definición de implicación y la equivalencia entre negar un existencial y usar un universal con el contenido negado, llegaríamos a la expresión:

$\forall var_1, var_2, var_{aux} \ [\neg(cond_1(var_1, var_{aux}) \wedge cond_2(var_{aux}, var_2)) \vee predicate(var_1, var_2)]$

Que tras la eliminación de los cuantificadores universales y la aplicación, de nuevo, de la definición de implicación, terminaría siendo:

$(cond_1(var_1, var_{aux}) \wedge cond_2(var_{aux}, var_2)) \implies predicate(var_1, var_2)$

Cuyo equivalente en Prolog sería el siguiente comando:

```
predicate(var1, var2) :- cond1(var1, varaux), cond2(varaux, var2).
```

El motivo por el cual se va a usar este esquema es para dejar más claro al lector cuáles son las variables importantes de cada problema (es decir, los argumentos de los predicados y sus salidas), que serán las cuantificadas universalmente al principio de la proposición, y cuáles son simplemente variables auxiliares (utilizadas en el interior de las funciones para transportar información), que serán las cuantificadas existencialmente.

El resto de predicados en lógica de primer orden, es decir, los que no sigan el esquema anterior, tendrán una traducción trivial en Prolog puesto que por norma general serán casos base.

3. EJERCICIO 2

invierte

PSEUDOCÓDIGO

```
Entrada: L1 (lista)
         L2 (lista)
Salida:  true si L1 y L2 están invertidas
         false en caso contrario.

Procesamiento:
         Si L1 y L2 vacías:
             evalúa a true.
         en caso contrario
             Mientras L1 no vacía:
                 M := resto(L1) invertido
                 evalúa a la comparación de L2 con [M concatenado con primero(L1)]
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- x: elementos
- l: listas

Funciones:

- *primero*[1] [$primero(l)$: Primer elemento de la lista l]
- *resto*[1] [$resto(l)$: Sublista resto de la lista l]

Predicados:

- $Vacia$[1] [$Vacia(l)$: T si la lista l está vacía]
- $invierte$[2] [$invierte(l_1, l_2)$: T si l_2 es la inversa de l_1]
- $concatena$[3] [$concatena(l_1, l_2, l_3)$: T si l_3 es la concatenación de l_1 y l_2]

Constantes:

- Lista vacía: [ ]

$\forall l\ [Vacia(l) \implies invierte(l, [\,])]$

$\forall x, l_1, l_2\ [[\exists l_m(invierte(resto(l_1), l_m) \wedge concatena(l_m, primero(l_1), l_2))] \implies invierte(l_1, l_2)]$

CÓDIGO

```
/******************************************
*    Exercise 2:
*
*    invierte(L1, L2)
*      Predicate that evaluates if L2 is the reverse list of L1. When given
*      an uninitialized variable, it returns the reverse list of L1 through L2.
*
*/

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

/*
*    Examples:
*
*      ?- invierte([1, 2, 3], X).
*      X = [3, 2, 1]
******************************************/
```

COMENTARIOS:

- Este predicado se basa en la inversión de la primera lista pasada como argumento para su posterior comparación con la segunda. Así, se va recorriendo la lista l_1 eliminando el primer elemento en cada llamada y concatenándolo al final en cada regreso a la función. Una vez terminado el proceso recursivo de inversión de l_1, se comprueba si coincide con l_2.

4. Ejercicio 3

insert

PSEUDOCÓDIGO

```
Entrada: X-P (elemento)
         L (lista ordenada)
         R (lista)
Salida:  true si R es el resultado de insertar X en L
          false en caso contrario.

Procesamiento:
         Si L vacía:
             evalúa a la comparación de R y X.
         en caso contrario
             Sea Ls una lista vacía
             Mientras P mayor que posicion(primero(L)):
                 añadimos a Ls el primero(L)
```

```
                tomamos el resto(L)
            evalúa a la concatenación de Ls con X y L
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- x: listas de un único elemento, un par elemento, cantidad
- l: listas
- p: números reales

Funciones:

- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]

Predicados:

- $Vacia^1$ [$Vacia(l)$: T si la lista l está vacía]
- $Posicion^2$ [$Posicion(p, x)$: T si p es el valor "cantidad" del par x]
- $Menor^2$ [$Menor(p_1, p_2)$: T si $p_1 < p_2$
- $MayorOIgual^2$ [$MayorOIgual(p_1, p_2)$: T si $p_1 \geq p_2$]
- $concatena^3$ [$concatena(l_1, l_2, l_3)$: T si l_3 es la concatenación de l_1 y l_2]
- $insert^3$ [$insert(x, l_1, l_2)$: T si l_2 es el resultado de insertar en orden x en l_1]

$\forall x, l\ [Vacia(l) \implies insert(x, l, x)]$

$\forall x, l_1, l_2, p_1, p_2\ [(Posicion(p_1, x) \wedge Posicion(p_2, primero(l_1)) \wedge MenorOIgual(p_1, p_2) \wedge concatena(x, l_1, r)) \implies insert(x, l_1, r)]$

$\forall x, l_1, l_2, p_1, p_2\ [[\exists l_3\ (Posicion(p_1, x) \wedge Posicion(p_2, primero(l_1)) \wedge Mayor(p_1, p_2) \wedge insert(x, resto(l_1), l_3) \wedge concatena(primero(l_1), l_3, l_2))] \implies insert(x, l_1, l_2)]$

CÓDIGO

```
/*******************************************
*    Exercise 3:
*
*    insert(X-P, L, R)
*      Predicate that inserts a pair of elements (X-P) in an ordered pair list
*      (L) in the position P, returning the resulting list through R.
*
*/

insert([X-P], [], [X-P]).
insert([X-P], [Y-Q|Zs], R) :-   P =< Q,
                                concatena([X-P], [Y-Q|Zs], R).
insert([X-P], [Y-Q|Zs], R) :-   P > Q,
                                insert([X-P], Zs, R1),
                                concatena([Y-Q], R1, R).

/*
*    Examples:
*      ?- 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.
*******************************************/
```

COMENTARIOS:

- Dado que en Prolog las listas se recorren a través de ir accediendo al primer elemento y a la sublista resto, lo más sencillo es implementar un esquema de búsqueda lineal para realizar las inserciones.
- La idea principal del programa es que si el elemento que queremos insertar debe ir en la posición n, se realizará una concatenación de los $n-1$ primeros elementos, el elemento nuevo y posteriormente todos los que previamente estuvieran de la posición n en adelante. Para ello se itera sobre la lista comparando cada elemento con el nuevo.
- En caso de igualdad, se ha decidido que el nuevo elemento se coloque delante para ahorrarnos un paso más en la recursión y por tanto mejorar la eficiencia. Por tanto, si sobre una lista de pares se observa que varios tienen el mismo valor, podemos también saber cuáles son los más "antiguos".
- Esta función sólo funciona sobre listas que contenga pares separados por un guión en los que el segundo elemento sea un número.

5. Ejercicio 4

5.1. Ejercicio 4.1

elem_count

PSEUDOCÓDIGO

```
Entrada: X (elemento)
         L (lista)
         Xn (Número de apariciones)
Salida:  true si Xn se corresponde con el número de apariciones
         false en caso contrario.

Procesamiento:
         contador = 0
         Mientras L no vacía:
             Si primero(L) es igual a X:
                 aumentamos en uno contador
             tomamos resto(L)
         evalúa a la comparación de contador y Xn
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- x: elementos
- l: listas
- n: números naturales

Funciones:

- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]
- $suma^2$ [$suma(n_1, n_2)$: Suma de $n_1 + n_2$]

Predicados:

- $Vacia^1$ [$Vacia(l)$: T si la lista l está vacía]
- $Igual^2$ [$Igual(x_1, x_2)$: T si $x_1 = x_2$]

- *elem_count*[3] [$elem\_count(x, l, n)$: T si n contiene el número de apariciones de x en l]

<u>Constantes:</u>

- Números naturales: **1** y **0**

$\forall x, l[Vacia(l) \implies elem\_count(x, l, 0)]$

$\forall x, l, n[(Igual(x, primero(l)) \land elem\_count(x, resto(l), n)) \implies elem\_count(x, l, suma(n, 1)]$

$\forall x, l, n[(\neg Igual(x, primero(l)) \land elem\_count(x, resto(l), n)) \implies elem\_count(x, l, n)]$

CÓDIGO

```
/*****************************************
*    Exercise 4.1:
*
*    elem_count(X, L, Xn)
*      Predicate that satisfies when the element X appears Xn times in the list L.
*
*/

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

/*
*    Examples:
*      ?- elem_count(b,[b,a,b,a,b],Xn).
*      Xn = 3 ;
*      false.
*
*      ?- elem_count(a,[b,a,b,a,b],Xn).
*      Xn = 2 ;
*      false.
*****************************************/
```

COMENTARIOS:

- Otro caso sencillo de recursión sobre todos los elementos de una lista, recorridos de forma lineal. Cada vez que nos encontramos con una coincidencia, llamamos recursivamente a la función con la sublista resto e incrementamos el contador; si no hay coincidencia, sólo volvemos a llamar a la función. Cuando evaluemos la lista vacía, devolvemos el número de coincidencias en una lista vacía, 0, y todos los incrementos se van sumando hasta que da lugar al resultado deseado.

5.2. Ejercicio 4.2

<u>**list_count**</u>

PSEUDOCÓDIGO

```
 Entrada: L1 (lista)
          L2 (lista)
          L3 (Lista de pares)
 Salida:  true si los pares de L3 se corresponden con
          el número de apariciones de los elementos de L1 en L2,
          false en caso contrario.

 Procesamiento:
          M = lista vacía
          Mientras L1 no vacía:
              contador = elem_count de primero(L1) en L2
              insertamos en M la dupla resultado: [primero(L1) - contador]
```

```
              tomamos el resto(L1)
         evalúa a la comparación de M con L3
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- x: elementos
- l: listas
- n: números naturales

Funciones:

- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]

Predicados:

- $Vacia^1$ [$Vacia(l)$: T si la lista l está vacía]
- Par^3 [$Par(x_1, x_2, n)$: T si x_1 es el par [elemento-apariciones] formado por el elemento x_2 y el número natural n]
- $concatena^3$ [$concatena(l_1, l_2, l_3)$: T si l_3 es la concatenación de l_1 y l_2]
- $elem\_count^3$ [$elem\_count(x, l, n)$: T si n contiene el número de apariciones de x en l]
- $list\_count^3$ [$list\_count(l_1, l_2, l_3)$: T si l_3 contiene los pares [elemento-apariciones] asociados a las apariciones de los elementos de l_1 en l_2]

Constantes:

- Listas: [ ] (lista vacía)

$\forall l_1, l_2\ [Vacia(l_1) \implies list\_count(l_1, l_2, l_1)]$

$\forall l_1, l_2, l_3\ [[\exists l_{aux}, n, x\ (list\_count(resto(l_1), l_2, l_{aux}) \land elem\_count(primero(l_1), l_2, n) \land Par(x, primero(l_1), n) \land concatena(x, l_{aux}, l_3))] \implies list\_count(l_1, l_2, l_3)]$

CÓDIGO

```
/*****************************************
*    Exercise 4.2:
*
*    list_count(L1, L2, L3)
*      Predicate that satisfies when L3 contains the appearances of the elements
*      of L1 in L2, with format [element-appearances] (for example, [b-6]).
*
*/

list_count([], _, []).
list_count([X|Zs], L2, L3) :-   list_count(Zs, L2, L),
                                elem_count(X, L2, N),
                                concatena([X-N], L, L3).

/*
*    Examples:
*      ?- 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.
*******************************************/
```

COMENTARIOS:

- En este caso ya tenemos gran parte del trabajo hecho gracias a `elem_count`; simplemente tenemos que recorrer la lista de elementos que nos es proporcionada de forma recursiva, ejecutar `elem_count` para cada elemento y concatenar los pares generados.

6. Ejercicio 5

<u>**sort_list**</u>

PSEUDOCÓDIGO

```
 Entrada: L1 (lista)
          L2 (lista)
 Salida:  true si L2 se corresponde con una ordenación de L1
          false en caso contrario.

 Procesamiento:
          Si L1 y L2 vacías:
              evalúa a true
          Sea Ls una lista vacía.
          Mientras L1 no vacía:
              insertar primero(L1) en Ls usando insert
              tomamos resto(L1)
          evalúa a la comparación de L2 con Ls
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

<u>Variables:</u>

- x: elementos
- l: listas

<u>Funciones:</u>

- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]
- $lista^1$ [$lista(x)$: Lista de un elemento generada por x]

<u>Predicados:</u>

- $Vacia^1$ [$Vacia(l)$: T si la lista l está vacía]
- $insert^3$ [$insert(x, l_1, l_2)$: T si l_2 es el resultado de insertar en orden x en l_1]
- $sort\_list^2$ [$sort\_list(l_1, l_2)$: T si l_2 contiene los pares de l_1 ordenados de menor a mayor por su segundo valor]

$\forall l\ [Vacia(l) \implies sort\_list(l, l)]$

$\forall l_1, l_2[[\exists l_{aux}(sort\_list(resto(l_1), l_{aux}) \wedge insert(lista(primero(l_1)), l_{aux}, l_2))] \implies sort\_list(l_1, l_2)]$

CÓDIGO

```
/*****************************************
*    Exercise 5:
*
*    sort_list(L1, L2)
*      Predicate that satisfies when L2 contains the pairs of L1 sorted (as in
*      the previous exercises, with format [element-appearances]).
*
*/

sort_list([], []).
sort_list([X|Zs], L2) :- sort_list(Zs, L), insert([X], L, L2).

/*
*    Examples:
*      ?- 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.
*****************************************/
```

COMENTARIOS:

- Para la implementación de esta función no se ha utilizado ningún algoritmo de ordenación clásico, y de hecho el método que se usa (ir insertando en orden todos los elementos que se obtienen tras recorrer la lista proporcionada) no es excesivamente eficiente; sin embargo, su simpleza y sencillez en Prolog son fundamentales.
- Cabe mencionar que, ya que lo que recibe la función `insert` son listas, para insertar un elemento en una lista hay que utilizar como argumento `[X]` en lugar de simplemente `X`.

7. Ejercicio 6

build_tree

PSEUDOCÓDIGO

```
Entrada: L (lista)
         T (Árbol de Huffman simplificado)
Salida:  true si T se corresponde con el árbol de Huffman de L
         false en caso contrario.

Procesamiento:
         si L está vacía,
             evalúa a false
         en caso contrario, si L tiene un elemento,
             evalúa a T == tree(info(first(L)), nil, nil)
         en caso contrario,
             evalúa a T == tree(1, tree(info(first(L)), nil, nil), Right) AND
                      build_tree(rest(L), Right)
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- l: listas

- t: árboles

Funciones:

- $info^1$ [$info(x)$: Información del elemento x]
- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]

Predicados:

- $Vacia^1$ [$Vacia(l)$: T si la lista l está vacía]
- $Tree^4$ [$Tree(x, t_l, t_r, t)$: T si t es el árbol formado por el nodo con elemento x y por los subárboles t_l (izquierdo) y t_r (derecho)]
- $build\_tree^2$ [$build_tree(l, t)$: T si t es el árbol simplificado de Huffman generado por l]

Constantes:

- Árbol vacío: **nil**
- Número natural: **1**

$\forall l, t\ [(Tree(info(primero(l)), nil, nil, t) \wedge Vacia(resto(l))) \implies build\_tree(l, t)]$

$\forall l, t\ [[\exists t_l, t_r\ (Tree(info(primero(l)), nil, nil, t_l) \wedge Tree(1, t_l, t_r, t) \wedge build\_tree(resto(l), t_r))] \implies build\_tree(l, t)]$

CÓDIGO

```
/*******************************************
*    Exercise 6:
*
*    build_tree(L, T)
*      Predicate that transforms an ordered list of pairs into a simplified
*      Huffman tree.
*
*/

build_tree([X-_], tree(X, nil, nil)).
build_tree([X-_|Rs], tree(1, tree(X, nil, nil), Right)) :- build_tree(Rs, Right).

/*
*    Examples:
*      ?- build_tree([], X).
*      false.
*
*      ?- build_tree([a-8], X).
*      X = tree(a, nil, nil) ;
*      false.
*
*      ?- 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-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.
*******************************************/
```

COMENTARIOS:

- El mecanismo es sencillo dado que siempre se realiza la misma acción (generar un nodo de la forma `tree(1, tree(X, nil, nil), R)`), salvo al final del todo, en el que hay que añadir el último elemento en ese nodo `R` con la expresión `tree(Y, nil, nil)`.
- Sin embargo, el punto anterior genera un pequeño inconveniente en el caso base de que tengamos que generar un árbol a partir de una lista con un solo elemento. Como ese elemento es el primero y el último a la vez, puede surgir la duda de si debería ser representado como `tree(1, tree(X, nil, nil), nil)` o `tree(X, nil, nil)`. El método elegido genera la segunda opción, y como la obtención de la primera requeriría cambios estructurales muy fuertes para el código, vamos a considerar este caso como una excepción a tratar de forma especial por futuras funciones; lo cual no supone problema alguno dado que realmente nunca se van a generar árboles con un elemento, porque carece totalmente de relevancia.

8. Ejercicio 7

8.1. Ejercicio 7.1

encode_elem

PSEUDOCÓDIGO

```
Entrada: X1 (elemento)
Entrada: X2 (lista)
         Tree (Árbol de Huffman simplificado)
Salida:  T si X2 es la lista con la codificación de X1 en el árbol Tree

Procesamiento:
         si Tree está vacío,
             evalúa a false
         si Tree tiene una sola hoja,
             evalúa a X2 == 0
         en caso contrario,
             si Tree tiene más de una hoja izquierda,
                 si Info de la primera hoja es igual a X1,
                     evalúa a X2 == 0
                 en caso contrario,
                     evalúa a first(X2) == 1 AND encode_elem(X1,rest(X2),Right(Tree))
             en caso contrario, si Tree tiene una hoja izquierda
                 si Info de la hoja izquierda es igual a X1,
                     evalúa a X2 == 0
                 en caso contrario, si Info de la hoja derecha es igual a X1,
                     evalúa a X2 == 1
                 en caso contrario,
                     evalúa a false
```

FORMALIZACIÓN EN LÓGICA DE PRIMER ORDEN

Variables:

- x: elementos
- l: listas
- t: árboles

Funciones:

- $lista^1$ [$lista(x)$: Lista compuesta por el elemento x]

Predicados:

- $Tree^4$ [$Tree(x, t_l, t_r, t)$: T si t es el árbol formado por el nodo con elemento x y por los subárboles t_l (izquierdo) y t_r (derecho)]
- $concatena^3$ [$concatena(l_1, l_2, l_3)$: T si l_3 es la concatenación de l_1 y l_2]
- $encode\_elem^3$ [$encode\_elem(x, l, t)$: T si l es la lista que contiene la codificación del elemento x en el árbol simplificado de Huffman t]

Constantes:

- Números naturales: **1** y **0**

$\forall x, t\ [Tree(x, nil, nil, t) \implies encode\_elem(x, lista(1), t)]$

$\forall x, t, t_r\ [[\exists t_l\ (Tree(x, nil, nil, t_l) \wedge Tree(1, t_l, t_r, t))] \implies encode\_elem(x, lista(0), t)]$

$\forall x_1, x_2, t\ [(x_1 \neq x_2) \implies [[\exists t_l, t_r\ (Tree(x_2, nil, nil, t_l) \wedge Tree(x_1, nil, nil, t_r) \wedge Tree(1, t_l, t_r, t))] \implies encode\_elem(x_1, lista(1), t)]]$

$\forall x_1, x_2, l, t, t_r\ [((x_1 \neq x_2) \wedge \neg Tree(x_1, nil, nil, t_r)) \implies [[\exists l_{aux}\ (encode\_elem(x_1, l_{aux}, t_r) \wedge concatena(lista(1), l_{aux}, l))] \implies encode\_elem(x_1, l, t)]]$

CÓDIGO

```
/*******************************************
*    Exercise 7.1:
*
*    encode_elem(X1, X2, Tree)
*      Predicate that encodes (returning the value through X2) the element X1,
*      based on the Huffman tree Tree.
*
*/

encode_elem(E, [0], tree(E, _, _)).
encode_elem(E, [0], tree(1, tree(E, _, _), _)).
encode_elem(E, [1], tree(1, tree(A, _, _), tree(E, _, _))) :- A \= E.
encode_elem(E, X, tree(1, tree(A, _, _), Right)) :- A \= E,
                                                    Right \= tree(E, _, _),
                                                    encode_elem(E, Y, Right),
                                                    concatena([1], Y, X).

/*
*    Examples:
*      ?-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.
*
*
*      ?- 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.
*
*       ?- encode_elem(a, X, tree(a, nil, nil)).
*       X = [1] ;
*       false.
*******************************************/
```

COMENTARIOS:

- Para esta función se tiene en cuenta la estructura cíclica que ya observamos en el ejercicio anterior: de cada `tree(1, tree(X, nil, nil), R)` leído, se compara `X` con el elemento deseado; si hay coincidencia se devuelve el 0 del final de la codificación, y si no hay coincidencia se sigue explorando el árbol `R`. Si llegamos al final del árbol, se evalúa de forma similar pero teniendo en cuenta que si la coincidencia se da en el último nodo, hay que concatenar un 1.
- Por último, para el caso especial donde existe sólo un elemento en el árbol, se devuelve 0 en caso de coincidencia; y para que esta instrucción no afecte al resto de casos (en concreto, al de que la coincidencia se dé en el último nodo), nos aseguramos cada vez que no haya una coincidencia de que el árbol derecho `R` no es de la forma `tree(E, nil, nil)`.

8.2. Ejercicio 7.2

encode_list

PSEUDOCÓDIGO

```
Entrada: L1 (lista)
         L2 (lista)
         Tree (Árbol de Huffman simplificado)
Salida:  T si L2 contiene las codificaciones de los elementos de L1 en el árbol Tree

Procesamiento:
         si L1 está vacía,
            evalúa a L2 == [] (L2 está vacía)
         en caso contrario,
            evalúa a encode_elem(first(L1), first(L2), Tree) AND
                     encode_list(rest(L1), rest(L2), Tree)
```

FORMALIZACIÓN EN LÓGICA DE SEGUNDO ORDEN

Variables:

- x: elementos
- l: listas
- t: árboles

Funciones:

- $lista^1$ [$lista(x)$: Lista compuesta por el elemento x]
- $primero^1$ [$primero(l)$: Primer elemento de la lista l]
- $resto^1$ [$resto(l)$: Sublista resto de la lista l]

Predicados:

- $Vacia^1$ [$Vacia(l)$: T si la lista l está vacía]

- *encode_elem*[3] [$encode\_elem(x, l, t)$: T si l es la lista que contiene la codificación del elemento x en el árbol simplificado de Huffman t]
- *encode_list*[3] [$encode\_list(l_1, l_2, t)$: T si l_2 es la lista que contiene las codificaciones de los elementos de l_1 en el árbol simplificado de Huffman t]

Constantes:

- Lista vacía: [ ]

$\forall l, t\ [Vacia(l) \implies encode\_list(l, [], t)]$

$\forall l_1, l_2, t\ [[\exists l_{aux_1}, l_{aux_2}\ (encode\_list(resto(l_1), l_{aux_1}, t) \land encode\_elem(primero(l_1), l_{aux_2}, t) \land concatena(lista(l_{aux_2}), l_{aux_1}, l_2))] \implies encode\_list(l_1, l_2, t)]$

CÓDIGO

```
/*****************************************
*    Exercise 7.2:
*
*    encode_list(L1, L2, Tree)
*      Performs the same task that encode_elem, but this time using lists of
*      elements and lists of codes.
*
*/

encode_list([], [], _).
encode_list([E|Rs], L, Tree) :- encode_list(Rs, L1, Tree),
                                encode_elem(E, X, Tree),
                                concatena([X], L1, L).

/*
*    Examples:
*      ?- 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.
*****************************************/
```

COMENTARIOS:

- Este predicado funciona exactamente igual que `list_count`; se itera recursivamente sobre la lista proporcionada aplicando `encode_elem` a cada elemento y concatenando los resultados.

9. EJERCICIO 8

encode

PSEUDOCÓDIGO

Entrada: L1 (lista)

```
          L2 (lista)
 Salida:  T si L2 contiene las codificaciones de los elementos de L1 según
          su frecuencia y si L1 sólo contiene letras del abecedario

 Procesamiento:
          dict := [a, b, c, ..., n, o, ..., z]
          si para algunas L3, L4, L5 y T se cumple que:
              list_count(dict, L1, L3) AND sort_list(L3, L4) AND
              invierte(L4, L5) AND build_tree(L5, T) AND encode_list(L1, L2, T)
          entonces,
              evalúa a true
          en caso contrario,
              evalúa a false
```

FORMALIZACIÓN EN LÓGICA DE SEGUNDO ORDEN

Variables:

- l: listas
- t: árboles

Predicados:

- *dictionary*[1] [$dictionary(l)$: T si l es la lista [a, b, ..., n, o, ..., z]]
- *list_count*[3] [$list\_count(l_1, l_2, l_3)$: T si l_3 contiene los pares [elemento-apariciones] asociados a las apariciones de los elementos de l_1 en l_2]
- *sort_list*[2] [$sort\_list(l_1, l_2)$: T si l_2 contiene los pares de l_1 ordenados de menor a mayor por su segundo valor]
- *invierte*[2] [$invierte(l_1, l_2)$: T si l_2 es la inversa de l_1]
- *build_tree*[2] [$build_tree(l, t)$: T si t es el árbol simplificado de Huffman generado por l]
- *encode_list*[3] [$encode\_list(l_1, l_2, t)$: T si l_2 es la lista que contiene las codificaciones de los elementos de l_1 en el árbol simplificado de Huffman t]
- *encode*[2] [$encode(l_1, l_2)$: T si l_2 es la lista que contiene las codificaciones de los elementos de l_1 (letras del abecedario) según su frecuencia de aparición]

$$\forall l_1, l_2\ [[\exists l_3, l_4, l_5, l_6, t\ (dictionary(l_3) \wedge list\_count(l_3, l_1, l_4) \wedge sort\_list(l_4, l_5) \wedge invierte(l_5, l_6) \wedge build\_tree(l_6, t) \wedge encode\_list(l_1, l_2, t)))] \implies encode(l_1, l_2)]$$

CÓDIGO

```
/****************************************
*    Exercise 8:
*
*    encode(L1, L2)
*      Predicate that encodes each element of L1 into L2 based on its frequence
*      inside L1.
*      Note: encode uses the predicate dictionary as a data base of the elements
*      we want to allow in L1.
*
*/

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(L3),
                    list_count(L3, L1, L4),
                    sort_list(L4, L5),
                    invierte(L5, L6),
                    build_tree(L6, T),
```

```
                    encode_list(L1, L2, T).

/*
*    Examples:
*      ?- 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.
*
*      ?- encode([i,a], X).
*      X = [[0], [1, 0]] ;
*      false.
*
*      ?- encode([i,2,a], X).
*      false.
*******************************************/
```

COMENTARIOS:

- Función final, que utiliza todas las que se han ido codificando a lo largo de la práctica. Seleccionamos los caracteres que queremos codificar (en este caso, el abecedario); contamos las apariciones de cada letra en la frase proporcionada; generamos los pares elemento-apariciones y los ordenamos de mayor a menor (usando `sort_list` e invirtiendo el resultado); construimos el árbol asociado a esos pares ordenados, y codificamos los caracteres de la frase proporcionada según ese árbol.
- Siempre que se intenten evaluar caracteres que no se obtengan mediante el predicado `dictionary`, la función `encode_elem` (a través de `encode_list`) dará error y por tanto se devolverá `false`, asegurándonos así de que no se evalúan caracteres incorrectos.
- En el ejemplo donde se usa la frase "inteligencia artificial" podemos ver que ninguna codificación se corresponde con una secuencia completa de unos, que debería ser el código de la letra menos usada. Esto ocurre porque estamos contando las apariciones de las letras de todo el abecedario, por lo que realmente el código completo de unos se corresponderá con alguna de las letras que no aparecen en "inteligencia artificial"; y como sólo se devuelven los códigos de las letras que sí aparecen, no llegamos a ver dicho código en ningún momento.