PRÁCTICA 3

PROLOG

Gloria del Valle Cano & Leyre Canales Antón
gloria.valle@estudiante.uam.es & leyre.canales@estudiante.uam.es

Índice

1. Introducción

En esta práctica nos hemos introducido a la programación lógica dentro de la programación declarativa con `Prolog`. Se nos han propuesto una serie de ejercicios para los cuales hemos tenido que cambiar la manera de pensar respecto de `LISP`. Nos hemos adentrado en el sistema deductivo, hemos vuelto a hacer uso de la recursión y hemos desarrollado los árboles de deducción como sistema de aprendizaje y entendimiento de los problemas dados.

2. Ejercicio 1

Implemente un predicado `duplica(L,L1)`, que es cierto si la lista `L1` contiene los elementos de `L` duplicados.
Para comprobar que la los elementos de `L` están duplicados en `L1` deducimos que:

- *(Caso base)*: Si `L` está vacía, `L1` también lo estará.
- *(Caso general)*: Cada elemento `X` de `L` estará de manera repetida, `[X,X]` en `L1`.

Esta es la implementación resultante:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Ejercicio1
%%      duplica(L,L1)
%%
%%      Cierto si la lista L1 contiene
%%      los elementos duplicados de L
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

Hemos probado la implementación con estos casos de prueba, obteniendo con éxito lo esperado:

```
?- duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]).
true.

?- duplica([1, 2, 3], [1, 1, 2, 3, 3]).
false.

?- duplica([1, 2, 3], L1).
L1 = [1, 1, 2, 2, 3, 3].

?- duplica(L, [1, 2, 3]).
false.
```

Además, ayudándonos del manual de *SWI-Prolog* hemos probado el funcionamiento de `trace`, para observar la traza de funcionamiento del programa, lo cual nos ha ayudado a observar cómo se comporta `Prolog`. Hemos detenido el *tracer* con el predicado `notrace`.

```
?- trace.
true.

[trace]  ?- duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]).
   Call: (8) duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]) ? creep
   Call: (9) duplica([2, 3], [2, 2, 3, 3]) ? creep
   Call: (10) duplica([3], [3, 3]) ? creep
   Call: (11) duplica([], []) ? creep
   Exit: (11) duplica([], []) ? creep
   Exit: (10) duplica([3], [3, 3]) ? creep
```

```
   Exit: (9) duplica([2, 3], [2, 2, 3, 3]) ? creep
   Exit: (8) duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]) ? creep
true.

[trace]  ?- duplica([1, 2, 3], [1, 1, 2, 3, 3]).
   Call: (8) duplica([1, 2, 3], [1, 1, 2, 3, 3]) ? creep
   Call: (9) duplica([2, 3], [2, 3, 3]) ? creep
   Fail: (9) duplica([2, 3], [2, 3, 3]) ? creep
   Fail: (8) duplica([1, 2, 3], [1, 1, 2, 3, 3]) ? creep
false.

[trace]  ?- duplica([1, 2, 3], L1).
   Call: (8) duplica([1, 2, 3], _3304) ? creep
   Call: (9) duplica([2, 3], _3558) ? creep
   Call: (10) duplica([3], _3570) ? creep
   Call: (11) duplica([], _3582) ? creep
   Exit: (11) duplica([], []) ? creep
   Exit: (10) duplica([3], [3, 3]) ? creep
   Exit: (9) duplica([2, 3], [2, 2, 3, 3]) ? creep
   Exit: (8) duplica([1, 2, 3], [1, 1, 2, 2, 3, 3]) ? creep
L1 = [1, 1, 2, 2, 3, 3].

[trace]  ?- duplica(L, [1, 2, 3]).
   Call: (8) duplica(_3302, [1, 2, 3]) ? creep
   Fail: (8) duplica(_3302, [1, 2, 3]) ? creep
false.
```

3. Ejercicio 2

Implementa el predicado `invierte(L, R)` que se satisface cuando `R` contiene los elementos de `L` en orden inverso. Utiliza el predicado `concatena/3`, *(/n: indica n argumentos),* que se satisface cuando su tercer argumento es el resultado de concatenar las dos listas que se dan como primer y segundo argumento.

Para la realización de este ejercicio utilizamos como función auxiliar `concatena(L1,L2,L)`, la cual verifica si `L` es la lista resultante escribiendo los elementos de `L2` a continuación de los de `L1`. Para ello:

- *(Caso base)*: Si la lista `L1` está vacía, entonces `L1` concatenado con `L2` es simplemente `L2`.
- *(Caso general)*: Si la lista `L1` tiene como primer elemento `X` y resto a continuación, entonces la concatenación de `L1` con `L2` será una lista `L` cuyo primer elemento sea `L` y cuyo resto es la concatenación de los restos de `L1` y `L2`.

La realización del predicado `concatena(L1,L2,L)` ha sido necesaria para realizar `invierte(L,R)`. Para ello:

- *(Caso base)*: La inversión de la lista vacía, será la lista vacía.
- *(Caso general)*: Si `L` contiene como primer elemento `X` y resto a continuación, entonces la inversión de `L` en `R` será, recursivamente, la inversión del resto sobre otra lista, es decir, tomando todos los elementos menos `X`, añadiendo a la salida esa última lista concatenada con los primeros elementos de `L`, dejando el resultado en `R`.

Es preciso resaltar que la notación que se utiliza por definición no es exactamente la de la implementación, ya que, necesitamos más nombres de variables para poder plantear el problema con mayor amplitud. Un ejemplo de equivalencia sería resaltar `L2` como el resto, o `L3` como resto invertido `R`.

En concreto, esta es la implementación resultante:

```
%%------------------------------------------
%%     Auxiliar (Ejercicio2)
%%     concatena(L1,L2,L)
%%
%%     Satisface cuando el tercer
%%     argumento es el resultado de
%%     concatenar L1 y L2
%%------------------------------------------

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%     Ejercicio2
%%     invierte(L,R)
%%
%%     Satisface cuando R contiene los
%%     elementos de L en orden inverso
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

Estos son los casos de prueba utilizados con las correspondientes salidas:

```
?- concatena([], [1, 2, 3], L).
L = [1, 2, 3].

?- concatena([1, 2, 3], [4, 5], L).
L = [1, 2, 3, 4, 5].

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

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

?- invierte([3, 4, 5, 9],L).
L = [9, 5, 4, 3].
```

4. Ejercicio 3

Implementar el predicado `palindromo(L)` que se satisface cuando `L` es una lista palíndroma, es decir, que se lee de la misma manera de izquierda a derecha y de derecha a izquierda. ¿Qué pasa si se llama `palindromo(L)`, donde `L` es una variable no instanciada? Explicar.

Se trata de una implementación sencilla sabiendo que tenemos `invierte(L,R)`, ya que en el caso de palíndromo podemos observar que `L` se debe leer igual hacia adelante que hacia detrás, por tanto esto por definición es sencillamente `invierte(L,L)`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%     Ejercicio3
%%     palindromo(L)
%%
%%     Satisface cuando L es palindroma
%%     se lee de la misma manera de
%%     derecha a izquierda que de
%%     izquierda a derecha
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

palindromo(L) :-
        invierte(L,L).
```

Así pues, obtenemos los siguientes resultados:

```
?- palindromo([1, 2, 1]).
true.

?- palindromo([1, 2, 1, 1]).
false.

?- palindromo([2,3,4,4,3,2]).
true.

?- palindromo([2,3,4,3,2]).
true.
```

El siguiente caso muestra lo que ocurre al llamar a `palindromo(L)`:

```
?- palindromo(L).
L = []
L = [_1554]
L = [_1554, _1554]
L = [_1554, _1560, _1554]
L = [_1554, _1560, _1560, _1554]
L = [_1554, _1560, _1566, _1560, _1554]
L = [_1554, _1560, _1566, _1566, _1560, _1554] .
```

`Prolog` no puede generar una lista con números aleatorios como tal, pero sí que puede satisfacer el problema en cuanto a direcciones de memoria. Nos da todos los casos que hemos ido declarando y le podemos ir preguntando si existen más soluciones. En nuestro caso hemos observado las salidas mostradas para asegurarnos de que la traza sigue siendo correcta.

5. Ejercicio 4

Implementar el predicado `divide(L,N,L1,L2)` que se satisface cuando la lista `L1` contiene los primeros `N` elementos de `L` y `L2` contiene el resto.
La base de conocimiento de este ejercicio la hemos planteado de la siguiente manera:

- *(Caso base)*: La división de una lista `L` en el elemento `#0` dejará en `L2` la lista origen `L`, mientras que `L1` tendrá la lista vacía.
- *(Caso general)*: Teniendo en la lista `L` como primer elemento `H` y `T` como resto de la misma, iremos dividiendo recursivamente la lista, dejando en `L1` la concatenación del primer elemento con `L2`, es por ello por lo que el numero de elemento, `X` lo iremos decrementando en cada iteración, `Xn`. Llamaremos recursivamente a la función `divide(L,N,L1,L2)` para continuar con el resto de la lista y dejar la otra parte correspondiente en `L2`.

Este es el código que hemos implementado:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Ejercicio4
%%      divide(L,N,L1,L2)
%%
%%      Satisface cuando L1 contiene los
%%      primeros N elementos de L
%%      L2 contiene el resto
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

divide(L,0,[],L).
divide([H|T],X,L1,Rest) :-
        Xn is X - 1,
        concatena([H],L2,L1),
        divide(T,Xn,L2,Rest).
```

A continuación mostramos tres casos de prueba para comprobar que la lógica planteada resuelve el problema con éxito:

```
?- divide(L, 3, [1, 2, 3], [4, 5, 6]).
L = [1, 2, 3, 4, 5, 6] .

?- divide([1, 2, 3, 4, 5], 3, L1, L2).
L1 = [1, 2, 3],
L2 = [4, 5] .

?- divide([6,7,3,4,2], 4, L1, L2).
L1 = [6, 7, 3, 4],
L2 = [2] .
```

6. Ejercicio 5

Implementar el predicado `aplasta(L, L1)` que se satisface cuando la lista `L1` es una versión *aplastada* de la lista `L`, es decir, si uno de los elementos de `L` es una lista, esta será remplazada por sus elementos, y así sucesivamente. ¿Qué pasa si se hace `?- aplasta(L,[1, 2, 3])`. Explicar.
Para encontrar la solución a este problema declaramos que:

- Si la entrada no es una lista (o como en la notación `\+`, no es el objetivo), entonces la salida es una lista cuyo único elemento es el de la entrada.
- La lista vacía aplastada será la lista vacía.
- Volvemos al punto inicial para cada uno de los elementos de la lista. De manera recursiva empezaremos por el primer elemento `X` y lo dejaremos en `L1`, y continuamos de la misma manera con el resto de la lista de entrada. Uniremos estos dos resultados con `concatena(L1,L2,L)` dejando el resultado en el último argumento.

La implementación siguiente detalla con más exactitud la lógica planteada:

```
%%-------------------------------------------
%%    Auxiliar (Ejercicio5)
%%    es_lista(L)
%%
%%    Satisface cuando el argumento
%%    de entrada es una lista
%%-------------------------------------------

es_lista([]).
es_lista([_|L]) :-
        es_lista(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%    Ejercicio5
%%    aplasta(L,L1)
%%
%%    Satisface si uno de los elementos
%%    de L es una lista, sera reemplazada
%%    por sus elementos
%%
%%    Si X no es lista, L1 es la lista
%%            que solo tiene como elemento X
%%    Verificamos recursivamente si Ln es
%%            la lista obtenida reemplazando
%%            cada Ln-1 por sus elementos
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

aplasta(X,[X]) :-
        \+ es_lista(X).
aplasta([],[]).
aplasta([X|Lini],Lsal) :-
        aplasta(X,L1),
        aplasta(Lini,L2),
        concatena(L1,L2,Lsal).
```

Finalmente, realizamos unas pruebas para observar que la implementación cumple con las expectativas:

```
?- aplasta([1, [2, [3, 4], 5], [6, 7]], L).
L = [1, 2, 3, 4, 5, 6, 7].

?- aplasta([[3, 2, [3, 4], 5], [6, 7]], L).
L = [3, 2, 3, 4, 5, 6, 7].

?- aplasta(L,[1, 2, 3]).
- (Bucle infinito) -
Action (h for help) ? abort
% Execution Aborted
```

La razón por la cual no es viable preguntarle a `Prolog` la versión de la lista sin aplastar es porque no es capaz de seguir una lógica aleatoria, más concretamente, sí que puede intuir lo que puede ocurrir con la versión de `palindromo(L)` por la declaración de predicados planteada, pero en este caso no hemos planteado ninguna declaración que sea capaz de agrupar los elementos, ni tampoco puede volver hacia atrás.

7. Ejercicio 6

Implementar el predicado `primos(N, L)` que se satisface cuando la lista `L` contiene los factores primos del número `N`.
Para el desarrollo de este ejercicio hemos utilizado algunas funciones de `Prolog` que ya se encuentran definidas:

- `floor(Expresion)`: función que evalúa una expresión y devuelve el entero más pequeño o igual al resultado de la evaluación.
- `sqrt(N)`: función que devuelve la raíz cuadrada de un número `N` concreto.
- `between(Low, High, Value)`: función que dados dos enteros, `Low` y `High`, tal que `High>=Low`. El valor de `Value` se vincula a todos los enteros que existen entre los dos valores.
- *`Int1`* `mod` *`Int2`*: función que devuelve el módulo de dos enteros. Se comprueba si es igual a cero, tiene resto 0, con `=:= 0`.

Estas premisas han sido tenidas en cuenta para poder encontrar el menor divisor. Nuestra implementación está pensada de la siguiente manera:

- Función `divisor_N(X,N)`: encuentra el menor divisor de `N`, `X`, que sea mayor o igual a 2.
  1. Encontramos el suelo del resultado de la raíz del número.
  2. Miramos los enteros que se encuentran entre 2 y el resultado anterior.
  3. Nos quedamos con el entero que haga resto cero al utilizarlo para dividir el número.

  Además, siempre va a ser divisible por sí mismo.
- Función `primos(N,L)`: introduce en `L` la descomposición en factores primos de `N`. Para ello:
  1. *(Caso base)*: En el caso de que `N` sea 1, la lista resultante será la vacía.
  2. *(Caso general)*: Dado `N`, siendo `X` el primer elemento de la lista resultante:
     a) Para todo `N` mayor que 1.
     b) Encontramos el menor divisor exacto, con `divisor_N(X,N)`.
     c) Dividimos `N` por el divisor encontrado.

d) Llamamos recursivamente a la función para sequir encontrando los divisores restantes con el resultado de la división anterior.

Cabe destacar que hemos utilizado el corte, `!`, ya que no vamos a contemplar más resultados que los que salen matemáticamente. Podemos prescindir de él, pero se trata de una cuestión de preferencia para que no tengamos que preguntarle a `Prolog` por más resultados cuando tampoco los hay.
Esta es la implementación que resume nuestra lógica:

```
%%------------------------------------------------
%%      Auxiliar (Ejercicio6)
%%      divisor_N(X,N)
%%
%%      Satisface cuando X es el menor
%%      divisor de N, mayor o igual que 2
%%------------------------------------------------

divisor_N(X,N) :-
        M is floor(sqrt(N)),
        between(2,M,X),
        N mod X =:= 0,
        !.
divisor_N(N,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Ejercicio6
%%      primos(N,L)
%%
%%      Satisface cuando L contiene los
%%      factores primos del numero N
%%
%%      Verificamos si L es la
%%              descomposicion en factores
%%              primos que queremos
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

primos(1,[]).
primos(N,[X|L]) :-
        N > 1,
        divisor_N(X,N),
        Rest is N/X,
        primos(Rest,L),
        !.
```

A continuación mostramos la salida de nuestro programa. Podemos comprobar que nuestra implementación no funciona en ambos sentidos. Esto se podría resolver multiplicando los valores de la lista de entrada y dejarlos en `N`.

```
?- primos(100,L).
L = [2, 2, 5, 5] .

?- primos(N,[2, 2, 5, 5]).
ERROR: Arguments are not sufficiently instantiated

?- primos(250,L).
L = [2, 5, 5, 5] .

?- primos(1040,L).
L = [2, 2, 2, 2, 5, 13].
```

8. Ejercicio 7

Codificación run-length de una lista: si la lista contiene `N` términos consecutivos iguales a X, estos son codificados como un par `[N, X]`.

Para la realización de este ejercicio vamos a necesitar declarar el predicado `length(L,N)`, para asignar a `N` la longitud de `L`. Para ello definimos:

- *(Caso base)*: La longitud de la lista vacía es 0.
- *(Caso general)*: Para cada entrada de la lista aumentamos en 1 el valor de `X`. Llamamos recursivamente a la función para contemplar todos los elementos de `L`.

Esta es la declaración de la función auxiliar:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%     Ejercicio7
%%     run_length(L,L1)
%%
%%     Satisface si la lista L contiene
%%     N terminos consecutivos iguales a
%%     X, codificados como par [N,X]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%-------------------------------------------
%%     Auxiliar (Ejercicio7)
%%     longitud(L,N)
%%
%%     Satisface cuando N es la longitud
%%     de la lista L
%%-------------------------------------------

longitud([],0).
longitud([_|L],N) :-
        longitud(L,X),
        N is X + 1.
```

Hemos probado la función para comprobar que funciona con éxito:

```
?- longitud([1,2],N).
N = 2.

?- longitud([1,2,6,7,7],N).
N = 5.

?- longitud(L,7).
L = [_2768, _2774, _2780, _2786, _2792, _2798, _2804] .

?- longitud([],N).
N = 0.
```

7.1. *Empezamos con el predicado `cod_primero(X, L, Lrem, Lfront)`. `Lfront` contiene todas las copias de `X` que se encuentran al comienzo de `L`, incluso `X`; `Lrem` es la lista de elementos que quedan.*

La lógica que hemos planteado para este problema es la siguiente:

- Dado `X` y una lista vacía de entrada, `Lfront` contendrá la lista formada sólo con `X`, mientras que en `Lrem` no habrá nada que dejar.
- Dado `X` y una lista `L` cuyo primer elemento sea `L` y otra lista `Lfront` dejaremos la ocurrencia de `X` en la misma.
- Dado `X` y la lista `L` cuyo primer elemento sea `T`, dejaremos en `Lrem` la ocurrencia de `T` y crearemos la lista `Lfront` con el elemento `X`. En el paso anterior introduciremos el corte para evitar que `X` no sea igual a `T`.

El código siguiente muestra la lógica que hemos utilizado para el problema:

```
1  %%-----------------------------------------
2  %%      Ejercicio 7.1
3  %%      cod_primero(X,L,Lrem,Lfront)
4  %%
5  %%      Satisface cuando Lrem contiene
6  %%      todas las copias de X que se
7  %%      encuentran al comienzo de L,
8  %%      incluso X; Lrem es la lista de
9  %%      elementos restantes
10 %%-----------------------------------------
11
12 cod_primero(X,[],[],[X]).
13 cod_primero(X,[X|Laux],Lrem,[X|Lfront]) :-
14         !,
15         cod_primero(X,Laux,Lrem,Lfront).
16 cod_primero(X,[T|Laux],[T|Laux],[X]).
```

Dados los siguientes casos de prueba obtenemos:

```
?- cod_primero(1, [1, 1, 2, 3], Lrem, Lfront).
Lrem = [2, 3],
Lfront = [1, 1, 1].

?- cod_primero(1, [2, 3, 4], Lrem, Lfront).
Lrem = [2, 3, 4],
Lfront = [1].

?- cod_primero(3, [3, 3, 3, 3, 4], Lrem, Lfront).
Lrem = [4],
Lfront = [3, 3, 3, 3, 3].
```

7.2. *El predicado* `cod_all(L,L1` *aplica el predicado* `cod_primero/4` *a toda la lista* `L`.
Para codificar toda la lista hemos seguido estos pasos:

- La codificación de la lista vacía será la lista vacía.
- Agrupamos cada primer elemento `X` de `L` con `cod_primero/4` y le pasamos el resto de la lista de manera recursiva para seguir haciendo lo mismo con el resto de elementos que dejamos en `Lrem`.

Este es el código resultante:

```
1  %%-----------------------------------------
2  %%      Ejercicio 7.2
3  %%      cod_all(L,L1)
4  %%
5  %%      Aplica cod_primero(X,L,Lrem,Lfront)
6  %%      a toda la lista L
7  %%-----------------------------------------
8
9  cod_all([],[]).
10 cod_all([X|L],[L1|Laux]) :-
11         cod_primero(X,L,Lrem,L1),
12         cod_all(Lrem,Laux).
```

Comprobamos que empaquetamos los elementos que están repetidos en toda la lista:

```
?- cod_all([1, 1, 2, 3, 3, 3, 3], L).
L = [[1, 1], [2], [3, 3, 3, 3]]
false.

?- cod_all([1, 1, 2, 3, 3, 4, 4, 4, 5], L).
L = [[1, 1], [2], [3, 3], [4, 4, 4], [5]]
false.
```

7.3. *El predicado* `run_length(L,L1` *aplica el predicado* `cod_all` *y luego transforma cada una de las listas en la codificación run-length.*
Para terminar con la codificación `run_length/2`, dividimos el planteamiento en dos funciones:

1. Función `cod_length(L,L1)`: transforma la lista de entrada `L` codificada, al estilo *run-length* como par *[N,X]*.
   - La codificación de la lista vacía será nuevamente la lista vacía.
   - Cada sublista de elementos, `[X1|X2]`, se compondrá del elemento que la compone con el número de veces que aparece, `N`, (gracias a la llamada a `length/2`) y así lo haremos con el resto de sublistas de `L`.
2. Función `run_length(L,L1)`: aplica `cod_all(L,L1)` para empaquetar los elementos con la función anterior.

La siguiente implementación resume la lógica planteada:

```
1  %%-------------------------------------
2  %%      Ejercicio 7.3
3  %%      1) cod_length(L,L1)
4  %%      2) run_length(L,L1)
5  %%
6  %%      1) Transforma cada una de las listas
7  %%              a la codificacion como par [N,X]
8  %%      2) Aplica cod_all(L,L1) a toda la lista
9  %%-------------------------------------
10
11 cod_length([],[]).
12 cod_length([[X1|X2]|L],[[N,X1]|L1]) :-
13         longitud([X1|X2],N),
14         cod_length(L,L1).
15
16 run_length(L,L1) :-
17         cod_all(L,Laux),
18         cod_length(Laux,L1).
```

Comprobamos que la salida es la que esperamos:

```
?- run_length([1, 1, 1, 1, 2, 3, 3, 4, 4, 4, 4, 4, 5, 5], L).
L = [[4, 1], [1, 2], [2, 3], [5, 4], [2, 5]]
false.

?- run_length([1, 1, 3, 3, 3, 3], L).
L = [[2, 1], [4, 3]]
false.
```

9. Ejercicio 8

Implementa el predicado `build_tree(List, Tree)` que transforma una lista de pares de elementos ordenados en una versión simplificada de un *árbol de Huffman*. Para representar árboles usaremos las funciones `tree(Info, Left, Right)` y `nil`. También usaremos el predicado `concatena/3`. Los nodos hoja del árbol se corresponden con los elementos de la lista ordenada y almacenan el elemento en el campo *Info*, mientras que los nodos intermedios siempre almacenan un 1.

En este ejercicio se nos pide codificar un texto formado por símbolos utilizando códigos binarios mediante una estructura simplificada de un *árbol de Huffman*. En concreto, iremos construyendo el árbol insertando el elemento más frecuente a la izquierda, y así sucesivamente por la derecha hasta que sólo queden dos elementos, los cuales se insertan en el orden original. Para ello hemos implementado `build_tree(List,Tree)` planteando la siguiente lógica, desde abajo hasta arriba:

1. Si `List` no es una lista, introducimos el elemento con nombre `X` en el árbol, [`Tree = tree(X,nil,nil)`].

2. Extraemos el primer símbolo a codificar y lo colocamos a la izquierda, [`Left = tree(X,nil,nil)`].

3. Si hubiese dos elementos, insertamos por la izquierda, [`Left = tree(E1,nil,nil)`], y luego por la derecha, [`Right = tree(E2,nil,nil)`].

4. En el caso de que hubiese más de dos elementos, introducimos el elemento por la izquierda, [`Left = tree(X,nil,nil)`], y aplicamos recursivamente la evaluación del resto de los casos por la derecha, [`Tree = tree(1,Left,Right)`], ayudándonos de `Rest`.

A continuación representamos el código final:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Ejercicio8
%%      build_tree(List,Tree)
%%
%%      Transforma una lista de pares de
%%      elementos ordenados en una version del
%%      arbol de Huffman
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

build_tree(X-_,tree(X,nil,nil)).
build_tree([X|Rest],tree(1,Left,nil)) :-
        Rest=[],
        build_tree(X,Left).
build_tree([E1,E2|Rest],tree(1,Left,Right)) :-
        Rest=[],
        build_tree(E1,Left),
        build_tree(E2,Right).
build_tree([X|Rest],tree(1,Left,Right)) :-
        Rest=[_,_|_],
        build_tree(X,Left),
        build_tree(Rest,Right).
```

Para lo cual hemos obtenido las siguientes construcciones para cada caso facilitado:

```
?- 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-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.

?- 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.

?- build_tree([b-7, g-4, 7-7], X).
X = tree(1, tree(b, nil, nil), tree(1, tree(g, nil, nil), tree(7, nil, nil)))
false.
```

8.1. *Implementar el predicado* `encode_elem(X1, X2, Tree)` *que codifica el elemento* `X1` *en* `X2` *basándose en la estructura del árbol* `Tree`.

Para ello hemos seguido la siguiente lógica:

1. Si `E1` es la raíz, no dejamos nada en `X2`.
2. Si `E1` tiene un hijo a la izquierda, codificamos el `0`.
3. Dejamos recursivamente el resto de la codificación concatenando `1` y la evaluación de `X1` con la parte derecha del árbol.

Este es el código que proponemos en relación a la lógica planteada:

```
1  %%-----------------------------------------
2  %%      Ejercicio 8.1
3  %%      encode_elem(X1,X2,Tree)
4  %%
5  %%      Codifica el elemento X1 en X2 basandose
6  %%      en la estructura del arbol de Huffman
7  %%-----------------------------------------
8
9  encode_elem(X1,[],tree(X1,nil,nil)).
10 encode_elem(X1,[0],tree(1,tree(X1,nil,nil),_)).
11 encode_elem(X1,X2,tree(1,_,Right)) :-
12         encode_elem(X1,Eval,Right),
13         concatena([1],Eval,X2).
```

Observamos que la salida es la esperada:

```
?- 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.
```

8.2. *Implementar el predicado* `encode_list(L1, L2, Tree)` *que codifica la lista* `L1` *en* `L2` *siguiendo la estructura del árbol* `Tree`.

Para ello hemos seguido la siguiente lógica de la misma manera que en los casos previos:

1. Si ambas listas de entrada están vacías, dejamos vacío el árbol.
2. Contatenamos el resultado de aplicar `encode_elem(X1,X2,Tree)` recursivamente, componiendo listas con la evaluación, `Eval`, de los elementos del árbol.

El código siguiente muestra lo explicado anteriormente:

```
%%-----------------------------------------
%%      Ejercicio 8.2
%%      encode_list(L1,L2,Tree)
%%
%%      Codifica la lista L1 en L2 basandose en
%%      la estructura del arbol de Huffman
%%-----------------------------------------

encode_list([],[],_).
encode_list([L1|R],L2,Tree) :-
        encode_list(R,Erest,Tree),
        encode_elem(L1,Eval,Tree),
        concatena([Eval],Erest,L2).
```

Tras la implementación, comprobamos los resultados con los siguientes casos de prueba:

```
?- 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.
```

8.3. *Implementar el predicado `encode(L1, L2)` que codifica la lista `L1` en `L2`. Para ello haced uso del predicado:* `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 implementar este ejercicio hemos seguido el siguiente razonamiento:

1. Ordenaremos la lista de entrada por `key=0`.
2. Comprobaremos la entrada de `dictionary(...)` y que los elementos de `L1` ordenados ya, pertenecen, gracias al predicado auxiliar declarado `pertenece_lista/2`, al diccionario.
3. Utilizamos la codificación `run_length/2` para hallar el número de veces que aparece un carácter en la lista.
4. Para ello, le damos la vuelta a la sintaxis de la codificación con la función auxiliar `forma_par/2` y lo dejamos de la forma *[número de apariciones - elemento]*.
5. Vamos introduciendo en la lista por el par nuevo originado con `ordenar_lista/2` la cual necesita de `insertar_lista/3`.
6. Invertirmos el orden de la lista para ordenar como debemos, con `invierte/2`.
7. Construimos el árbol con la lista `L1`, con la función `build_tree/2`.
8. Aplicamos `encode_list/2` a toda la lista `L1` basándonos en el árbol `Tree` creado.

A continuación mostramos la implementación que refleja nuestra idea:

```
%%-----------------------------------------
%%      Auxiliar (Ejercicio8.3)
%%      pertenece(X,L)
%%
%%      Satisface cuando X es elemento de L
%%-----------------------------------------
pertenece(X,[X|_]).
pertenece(X,[_|Res]) :-
        pertenece(X,Res).

%%-----------------------------------------
%%      Auxiliar (Ejercicio8.3)
%%      pertenece_lista(L1,L2)
%%
%%      Satisface cuando los elementos de L1
%%      son elementos de L2
%%-----------------------------------------
pertenece_lista([],[_|_]).
pertenece_lista([X|Res],L) :-
        pertenece(X,L),
        pertenece_lista(Res,L).

%%-----------------------------------------
%%      Auxiliar (Ejercicio8.3)
%%      formar_par(L1,L2)
%%
%%      Satisface cuando los argumentos del
%%      arbol tienen la forma [tipo-num_veces]
%%-----------------------------------------
formar_par([],[]) :- !.
formar_par([[X1,X2|_]|Res],[[X2-X1]|Laux]) :-
        formar_par(Res,Laux).

%%-----------------------------------------
%%      Auxiliar (Ejercicio8.3)
%%      insertar_lista(L1,L2,L3)
%%
%%      Inserta en una lista el valor despues
%%      del guion del par [tipo-num_veces]
%%      L2 es nuestra cola auxiliar
%%-----------------------------------------
insertar_lista([X1-X2],[],[X1-X2]).
insertar_lista([X1-X2],[L1-L2|Res],[X1-X2,L1-L2|Res]) :-
        X2 =< L2.
insertar_lista([X1-X2],[L1-L2|Res],[L1-L2|Res2]) :-
        insertar_lista([X1-X2],Res,Res2),
        X2 > L2.

%%-----------------------------------------
%%      Auxiliar (Ejercicio8.3)
%%      ordenar_lista(L1,L2)
%%
%%      Satisface si encuentra la lista ordenada
%%      por el par [tipo-num_veces]
%%-----------------------------------------
ordenar_lista([],[]).
ordenar_lista([[X1-X2]|Res],Res2) :-
        ordenar_lista(Res,Cola),
        insertar_lista([X1-X2],Cola,Res2).

%%-----------------------------------------
%%      Ejercicio 8.3
%%      encode(L1,L2)
%%
%%      Codifica la lista L1 en L2 usando el
%%      predicado diccionario
%%-----------------------------------------
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([],nil).
encode(L1,Ldest) :-
        sort(0, @=<, L1, Lini),
        dictionary(D),
        pertenece_lista(Lini,D),
        run_length(Lini,Lrest),
        formar_par(Lrest,Lrest2),
        ordenar_lista(Lrest2,Lrest3),
        invierte(Lrest3,Lrest4),
        build_tree(Lrest4,Tree),
        encode_list(L1,Ldest,Tree).
```

La salida obtenida es la siguiente:

```
?- 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|...],
[1, 1, 1, 1|...], [0], [1, 1|...], [1|...], [...|...]|...]
false.

?- encode([i,2,a],X).
false.

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

?- encode([m,a,d,r,e,m,i,a,c,o,n,e,l,e,j,e,r,c,i,c,i,o],X).
X = [[1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 1, 1, 1|...],
[1, 1, 1, 0], [0], [1, 1, 1|...], [1, 0], [1|...], [...|...]|...]
false.

?- encode([v,a,y,a,s,e,m,a,n,a,s,a,n,t,a,l,l,e,v,o,´],X).
false.

?- encode([v,a,l,e,t,í,a],X).
false.

?- encode([v,a,l,e,t,i,a],X).
X = [[1, 0], [0], [1, 1, 1, 0], [1, 1, 1, 1, 1], [1, 1, 0],
[1, 1, 1|...], [0]]
false.

?- encode([v,o,y,y,a,p,a,2,4,a,n,y,o,s],X).
false.

?- encode([g,l,o,r,i,a,y,l,e,y,r,e],X).
X = [[1, 1, 1, 1, 1, 1, 0], [1, 1, 0], [1, 1, 1, 1, 0], [1, 0],
[1, 1, 1, 1|...], [1, 1, 1|...], [0], [1|...], [...|...]|...]
false.

?- encode([g,l],X).
X = [[1], [0]]
false.

?- encode([g,l],[[1],[0]]).
true
false.
```