Práctica 3: Programación declarativa, lógica y restricciones

AUTORES

Ignacio Amaya de la Peña – 11m021

Adrián Cámara Canuedo – 11m004

Borja Mas García – 11m001

Enunciado 1: Encuestas

Se dispone de dos encuestas sobre los equipos y jugadores de fútbol más populares. Los equipos están almacenados en hechos de un predicado equipo/1 cuyo argumento es una constante que identifica el equipo. Los jugadores lo están en un predicado jugador/1 cuyo argumento es una constante que identifica el jugador. Cada hecho representa la respuesta de un encuestado.

Definir el predicado equipos_preferidos/1 que verifica que su argumento es la lista de los equipos que aparecen el mayor número de veces en la encuesta.

Definir el predicado jugadores_no_populares/1 que verifica que su argumento es la lista de los jugadores que aparecen el menor número de veces en la encuesta.

Nota: la ordenación de la lista es indiferente.

Nota: Las encuestas (los hechos de los predicados jugador/1 y equipo/1) deben figurar en un fichero independiente ("encuesta.pl") que se debe incluir en el programa. Es decir, los hechos no deben aparecer explícitamente en el programa.

Solución

Lo primero que hemos hecho ha sido crear la encuesta para realizar después las pruebas sobre esta. Para ello en otro fichero llamado encuesta.pl hemos creado nuestra base de datos acerca de los resultados de las encuestas. Dichos resultados nos los hemos inventado.

Los datos de dicha encuesta eran los siguientes:

```
equipo(realmadrid).
equipo(realmadrid).
equipo(atletico).
equipo(zaragoza).
equipo(barsa).
equipo(atletico).
equipo(realmadrid).
equipo(barsa).
equipo(atletico).
```

```
jugador(messi).
jugador(alves).
jugador(alves).
jugador(ronaldo).
jugador(costa).
jugador(benzema).
jugador(courtois).
jugador(ronaldo).
jugador(courtois).
jugador(costa).
jugador(ronaldo).
jugador(courtois).
```

Una vez teníamos la encuesta en nuestro fichero principal la hemos importado la encuesta para poder trabajar con ella mediante el siguiente código.

```
:-include('encuesta').
```

Para realizarlos dos predicados que se nos piden hemos decidido desarrollar el predicado auxiliar numeroVeces que verifica si un elemento aparece en una lista un número determinado de veces. Para ello hemos puesto el caso base cuando no hay elementos y a partir de ahí hemos hecho dos casos que recorrerán la lista de forma recursiva. Uno de ellos incrementará el contador si el elemento coincide con el que se busca y el otro llamará recursivamente a numeroVeces sin incrementar el contador. El código de este predicado es el siguiente:

```
numeroVeces(_,[],0).
numeroVeces(X,[Z|Zs],N):-
      X==Z,
      numeroVeces(X,Zs,N1),
      N is N1+1.
numeroVeces(X,[Z|Zs],N):-
      X\==Z,
      numeroVeces(X,Zs,N).
```

A partir de este predicado auxiliar hemos realizado dos más. Uno de ellos comprueba que su segundo argumento es una lista con los elementos más repetidos de la lista pasada como primer argumento. El otro es análogo al anterior pero comprobando que esos elementos sean los menos repetidos. Para implementarlos hemos usado el meta predicado de agregación *setof* /3. El segundo argumento de masRepetido/2 será la lista en donde todos los elementos del primer parámetro de *setof/3* cumplen los objetivos siguientes: deben de ser miembros de la lista pasada como primer argumento en masRepetido/2 y no puede haber otro elemento de esta lista con un número de veces de aparición mayor. El otro predicado, menosRepetido es análogo al anterior pero comprobando que el número de veces de aparición nunca sea menor en otros elementos. Por tanto estos dos predicados son ambos meta predicados *setof/3* que hemos decidido encapsular en otros predicados para facilitar la legibilidad del código y su reutilización. El código de estos predicados es el siguiente:

```
masRepetido(L1,L2):-
      setof(X,((member(X,L1),numeroVeces(X,L1,N1),not((member(Y,L
      1),numeroVeces(Y,L1,N2),N1<N2)))),L2).


menosRepetido(L1,L2):-
      setof(X,((member(X,L1),numeroVeces(X,L1,N1),not((member(Y,L
      1),numeroVeces(Y,L1,N2),N1>N2)))),L2).
```

Con estos dos últimos predicados auxiliares realizar los predicados que se nos pedían resulta trivial. Usando la meta llamada de aridad 3 findall podemos recoger en un array todos los equipos y jugadores que tenemos en nuestra encuesta. Aplicando masRepetido o menosRepetido a esta lista tendremos en el segundo argumento de estos predicados la lista de los equipos preferidos o de los jugadores no populares. A continuación se detalla el código empleado.

```
equipos_preferidos(L):-
     findall(X,equipo(X),L1),
     masRepetido(L1,L).


jugadores_no_populares(L):-
     findall(X,jugador(X),L1),
     menosRepetido(L1,L).
```

Tras probar ambos predicados con nuestra encuesta de prueba obtenemos que los equipos preferidos son Atlético y Real Madrid y que los jugadores menos populares son Benzema y Messi, que es el resultado que obtenemos si hacemos nosotros el recuento.

Enunciado 2: La bandera de Italia

Representaremos figuras geométricas por estructuras de dos argumentos, cuyo nombre es el de la figura, el primer argumento sus dimensiones y el segundo su color. Para las dimensiones se utiliza otra estructura, de nombre desconocido (pero siempre el mismo para cada tipo de figura) y tantos argumentos como sea necesario para definir las dimensiones de la figura. Así, tenemos estructuras de la forma X(D,C) donde X puede ser cuadrado, rectángulo, triangulo, etc.; D será otra estructura con un argumento que contiene la longitud del lado (para el cuadrado), dos argumentos con la base y la altura (para el rectángulo), tres argumentos con las longitudes de dos lados y el valor del ángulo que forman (para el triángulo), etc.; y C contendrá una constante que identifica un color: rojo, blanco, verde, azul, etc.

Definir el predicado bandera_italia/2 que dada una lista de figuras geométricas L1 (que vendrá como entrada a la llamada en el primer argumento), las cuales pueden ser blancas, rojas o verdes, se verifica que el segundo argumento L2 contiene las figuras de L1 ordenadas como la bandera de Italia (es decir, primero las verdes, segundo las blancas y tercero las rojas).

Nota: la ordenación de las figuras del mismo color entre sí es indiferente. Puede ser la del ejemplo u otra cualquiera. No es necesario obtener todas las soluciones de ordenación diferente, solo una cualquiera.

Suponer que el usuario puede introducir (vía teclado) su figura geométrica favorita y la dimensión máxima de la misma, definir el predicado bandera_italia_adhoc/2 que dada una lista de figuras geométricas L1 (blancas, rojas, o verdes) en el primer argumento, lee la figura

geométrica favorita y la dimensión máxima (una después de la otra) y verifica que el segundo argumento L2 contiene las figuras de L1 que coinciden con la figura favorita y que tienen una dimensión menor o igual que la máxima ordenadas como la bandera de Italia.

Nota: Para comparar dimensiones deben ser estructuras idénticas (mismo nombre y aridad) y se utiliza orden alfabético-lexicográfico: se comparan alfabéticamente el nombre con el nombre y recursivamente los pares de argumentos en la misma posición, recorriendo las estructuras en profundidad. Este orden puede dar resultados como que bxh(3,10) es menor que bxh(4,5) (ya que 3 es menor que 4), aunque corresponde a un rectángulo de mayor área.

Solución

Para realizar el predicado bandera_italia/2 hemos usado el meta predicado findall tres veces consecutivas para tener tres listas. Estas listas contendrán cada una de ellas los elementos de la lista que tiene bandera_italiana como primer argumento que son de color verde, blanco y rojo respectivamente. Estos colores estarán almacenados en el segundo argumento de los functores que representan a las figuras.

Una vez tenemos estas tres listas las encadenamos y ya tenemos la lista que queríamos como segundo argumento del predicado que nos pedían.

Para realizar este predicado no hemos precisado de ningún predicado auxliar.

El código del predicado es el siguiente:

```
bandera_italia(L1,L2):-
      findall(X,(member(X,L1),arg(2,X,verde)),V),
      findall(Y,(member(Y,L1),arg(2,Y,blanco)),B),
      findall(Z,(member(Z,L1),arg(2,Z,rojo)),R),
      append(V,B,L3),
      append(L3,R,L2).
```

También nos pedían que realizásemos otro predicado donde teníamos que pedir al usuario información sobre su figura favorita y sus dimensiones. Para ello hemos usado los predicados leer y escribir de entrada y salida. Y el predicado tab para imprimir un espacio en blanco después de realizar cada write.

Después buscamos por medio de un findall todos los equipos que pertenecen a la lista del primer argumento de bandera_italia_adhoc/2 y comprobamos que son de la figura introducida por el usuario. Esto se guarda en otra lista a la que aplicamos otro findall para encontrar los elementos cuyo primer argumento (el relativo al tamaño) es lexicográficamente inferior a las dimensiones introducidas por el usuario y obtener así otra lista.A la lista obtenida anteriormente le aplicamos el predicado bandera_italia/2 realizado anteriormente para ordenar la lista que se devolverá en el segundo argumento. El código es el siguiente:

```
bandera_italia_adhoc(L1,L2):-
      write('Escoja su figura favorita:'),
      tab(1),
      read(X),
      write('Elija sus dimensiones:'),
      tab(1),
      read(D),
      findall(Y,(member(Y,L1),functor(Y,X,_)),L3),
      findall(Z,(member(Z,L3),arg(1,Z,Zs),Zs@=<D),L4),
      bandera_italia(L4,L2).
```

Enunciado 3: Ordenación por criterio libre

Se trata de hacer un programa de orden superior para la ordenación de una lista, de manera que el criterio de ordenación está sin especificar y se obtiene como argumento. Por tanto, el criterio va a ser un objetivo ejecutable que se pasa en uno de los argumentos de la llamada al predicado principal del programa y puede ser tan complejo como, por ejemplo, el que corresponde al orden descendente por la longitud de cada elemento (dando por supuesto que son listas). Se debe:

1. Programar un predicado qsort(List,Order,Set) tal que Set es la lista List ordenada según el criterio Order. Utilizar el algoritmo de ordenación rápida (quicksort en las transparencias).

2. Escribir el segundo argumento de la llamada a qsort/3 que correspondería al criterio habitual: el de ordenación ascendente por el valor numérico de los elementos.

Pista: El criterio es una relación entre cada dos elementos sucesivos de la lista ordenada, por lo que es necesario especificar una forma en la que identificar, del objetivo ejecutable que corresponde al criterio, los argumentos que corresponden a dichos elementos. Por ejemplo, para el criterio habitual el objetivo sería X=<Y, siendo X e Y elementos sucesivos de la lista; el programa debe ser capaz de identificar cual es cual: en este caso, Y es el que sigue a X (de lo contrario sería orden descendente).

Otros criterios que podrían considerarse son:

- Orden ascendente alfabético-lexicográfico del tercer argumento de los elementos de la lista.

- Orden descendente del valor numérico resultante del polinomio en dos variables de cada elemento de la lista con los valores de las variables a 2 y 3, respectivamente.

- Orden ascendente del valor resultante de sumar todos los nodos hoja de los árboles binarios elementos de la lista.

- Cualquier orden ascendente o descendente que se puede expresar como un objetivo Prolog.

Aclaración: Por ejemplo, podemos pensar en la siguiente llamada qsort(L,"ascendente por el valor de la suma de las hojas de árboles binarios",S) que debería ordenar ascendentemente por los valores de la suma de las hojas de los árboles binarios (es decir, se deben sumar las hojas de los elementos de la lista, que se suponen árboles binarios, y comparar para realizar la ordenación). En esta llamada, el criterio tal como aparece no está expresado como objetivo Prolog. El criterio tendrá que estar expresado en Prolog. Por tanto, se trata de definir qué forma debe tener el argumento del criterio y modificar quicksort como sea necesario para realizar las comparaciones correspondientes. No hay que programar ni "reunir las hojas de un árbol binario", ni "sumar los elementos de una lista", ni "calcular la longitud de una lista", ni ninguna parte del criterio (ya que no es posible programar todo lo que podría usarse como criterio).

Solución

Para la realización de este apartado nos hemos apoyado en el siguiente código de las transparencias:

```
qsort([],[]).
qsort([X|L],SL) :-
      partition(L,X,Left,Right),
      qsort(Left,SLeft),
      qsort(Right,SRight),
      append(SLeft,[X|SRight],SL).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
      E < C,
      partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
      E >= C,
      partition(R,C,Left,Right1).
```

Hemos modificado el qsort para que sea de aridad tres. De esta forma el segundo elemento será el orden que queremos que se use para ordenar nuestra lista. Para realizar este qsort/3 hemos modificado también el predicado partition de forma que su tercer argumento sea el orden a utilizar (partition pasará a ser de aridad 5).

En qsort/3 no se han tenido que realizar cambios. Lo único que cambia respecto al de las transparencias es que se realiza una llamada al partition que nosotros hemos modificado incluyendo el orden.

Con el operador univ construimos un functor cuyo nombre es el orden que usaremos y los parámetos los elementos a comparar. Después lo llamamos con el meta predicado call o con

el meta predicado not dependiendo de los elementos de la lista en donde nos encontremos hasta que mediante llamadas recursivas el último parámetro de qsort contenga toda la lista ordenada con el orden que le hemos indicado.

```
qsort([],_,[]).
qsort([X|L],Order,SL) :-
      partition(L,X,Order,Left,Right),
      qsort(Left,Order,SLeft),
      qsort(Right,Order,SRight),
      append(SLeft,[X|SRight],SL).

partition([],_B,_,[],[]).
partition([E|R],C,Order,[E|Left1],Right):-
      P=..[Order,E,C],
      call(P),
      partition(R,C,Order,Left1,Right).
partition([E|R],C,Order,Left,[E|Right1]):-
      P=..[Order,E,C],
      not(P),
      partition(R,C,Order,Left,Right1).
```

Hemos programado los siguientes criterios de ordenación para poder usar nuestro qsort:

```
ordenCreciente(X,Y):-
      X<Y.
ordenDecreciente(X,Y):-
      X>Y.
orden3Lex(X,Y):-
      arg(3,X,Xs),
      arg(3,Y,Ys),
      Xs@=<Ys.
```

El primero corresponde al criterio habitual, es decir, la ordenación ascendente por el valor numérico de los elementos. El segundo los ordena por el valor decreciente. El tercero los ordena siguiendo un orden ascendente alfabético-lexicográfico del tercer argumento de los elementos de la lista (deberán ser functores).

A continuación especificamos unos ejemplos con las llamadas a qsort con cada uno de estos órdenes.

```
?- qsort([1,2,5,6,7,8,0,3,5],ordenCreciente,X).
X = [0,1,2,3,5,5,6,7,8]

?- qsort([23,1,3,5,3,0,1,3,64,2,25],ordenDecreciente,X).
X = [64,25,23,5,3,3,3,2,1,1,0]
```

```
?-
qsort([casa(2,4,12),perro(0,3,6),avion(24,3,3),perro(1,3,0),gato
(3,1,9)],orden3Lex,X).
X =
[perro(1,3,0),avion(24,3,3),perro(0,3,6),gato(3,1,9),casa(2,4,12
)]
```