Práctica 2: Programación en Prolog

Programación Declarativa: Lógica y Restricciones

DIA

Vidal Peña, Arturo
W140307

4 de junio de 2019

Índice de contenidos

1. Código empleado y las explicaciones

1.1. Predicado menor/4

```
menor(A, B, Comp, M) :-
 functor(X, Comp, 2),
 arg(1,X,A),
 arg(2,X,B),
 call(X)->
 M = A;
 M = B.
```

En este predicado, preparo con *functor/3* una estructura en *X* de tres términos, y añado *Comp, A* y *B* con los predicados arg/3.

Luego, con *call/1* llamo a esa estructura y guardo el resultado en *M*.

1.2. Predicado menor_o_igual/2

```
menor_o_igual(X, Y) :-
 menor_o_igual_libre(X, Y);
 (nonvar(X), nonvar(Y), functor(X, Term1, Aridad1), functor(Y, Term2, Aridad2),
  (menor_nombre(Term1, Term2);
     menor_aridad(Term1,Term2,Aridad1,Aridad2);
     menor_o_igual_argumento(X, Y, Term1, Term2, Aridad1, Aridad2))).
```

Para empezar, compruebo el primer caso de que alguna (o ambas) variables sean variables libres (*menor_o_igual_libre*).

En caso contrario (si ninguna de las dos lo es), extraigo de ambas variables su término y su aridad para contrastarlos. Este predicado se validará si se cumple alguno de los siguientes.

1.2.1. Predicado menor_o_igual_libre/2

```
menor_o_igual_libre(X, Y) :-
 var(X), X = Y;
 var(Y), Y = X.
```

Será cierto si alguno de los términos es una variable libre, igualándolo al otro. Si ninguno de los dos es una variable libre, volveríamos a *menor_o_igual/2* para comprobar el resto de opciones.

1.2.2. Predicado menor_nombre/2

```
menor_nombre(Term1, Term2) :-
 Term1 @< Term2.
```

Este predicado, pasándole dos términos, se valida si el primero es menor que el segundo. Si no se validara, volvería a *menor_o_igual/2*, donde se comprobaría el predicado *menor_aridad/4*.

1.2.3. Predicado menor_aridad/4

```
menor_aridad(Term1, Term2, Aridad1, Aridad2) :-
 Term1 @= Term2,
 Aridad1 < Aridad2.
```

Este, recibe ambos términos y sus respectivas aridades y, comprobando que ambos términos son iguales, comprueba si la aridad del primero es menor a la del segundo. En el caso de que sea así, se valida. Si no, vuelve a *menor_o_igual/2*, y pasa al predicado *menor_igual_argumento/6*.

1.2.4. Predicado menor_o_igual_argumento/2

```
menor_o_igual_argumento(X, Y, Term1, Term2, Aridad1, Aridad2) :-
 Term1 @= Term2,
 Aridad1 == Aridad2,
 X =.. [Term1|ListaX],
 Y =.. [Term2|ListaY],
 menor_igual_argumento_rec(ListaX, ListaY).
```

Aquí, comprobamos primero que ambos términos son iguales, y que tienen la misma aridad. Luego, generamos en *X* e *Y* dos listas con sus respectivos términos y argumentos, para luego llamar a *menor_o_igual_argumento_rec/2*.

1.2.5. Predicado menor_o_igual_argumento_rec/2

```
menor_igual_argumento_rec([], []).

menor_igual_argumento_rec([Arg1|T1], [Brg1|T2]) :-
 (number(Arg1), number(Brg1), Arg1 =< Brg1 ;
    Arg1 @=< Brg1),
 menor_igual_argumento_rec(T1, T2).
```

Primero ponemos el caso base, con dos listas vacías de argumentos, que siempre será cierto.

En otro caso, comprobamos si los primeros argumentos de ambas listas pasadas son números, y comprobamos si el argumento de la primera lista es menor o igual al correspondiente en la segunda. Si no son números, hacemos la misma comprobación a nivel de término.

Hacemos esta llamada recursivamente, recorriendo ambas listas hasta que las acabamos (lista vacía), o encontramos un argumento en la primera lista que sea mayor al correspondiente en la segunda.

1.3. Predicado lista_hojas/2

```
lista_hojas([], []).

lista_hojas([H|T], [tree(H, void, void)|Hojas]) :-
 lista_hojas(T, Hojas).
```

Para este predicado, su caso base será con dos listas vacías.

En otro caso, recibiendo una lista de datos, genera una lista de hojas (o nodos) de la forma *tree('dato', void, void)*, donde *'dato'* es cada uno de los datos de la lista. Se llama recursivamente hasta acabar con todos los datos de la lista.

1.4. Predicado hojas_arbol/3

```
hojas_arbol([Hoja|Hojas],Comp,Arbol) :-
 hojasArbolRec(Hojas,Comp,Hoja,Arbol).
```

Este predicado llama a uno auxiliar recursivo *hojasArbolRec/4*, con el primer dato de la lista de hojas como argumento extra.

1.4.1. Predicado hojasArbolRec/3

```
hojasArbolRec([],_,Arbol,Arbol).

hojasArbolRec([H|T],Comp,Arbol,Tree) :-
 arg(1,H,RaizIzq),
 arg(1,Arbol,Raiz),
 menor(Raiz,RaizIzq,Comp,M),
 hojasArbolRec(T,Comp,tree(M,Arbol,H),Tree).
```

El caso base consiste en una lista de hojas vacía, sin criterio de comparación, con el árbol como tercer y cuarto argumento. Básicamente, unifica el árbol de salida.

En otro caso, recibiendo una lista, extrae en *RaizIzq* el dato de la hoja y en *Raiz* la raíz actual del árbol, usando el predicado *arg*. Luego, los compara extrayendo el menor según el criterio de comparación especificado en *Comp* y guardándolo en *M*; y por último, se llama de manera recursiva con el resto de la lista de hojas, el mismo criterio de comparación, un nuevo árbol con *M* como raíz, el resto del árbol y el dato extraído de la lista de hojas; y el árbol de salida en el último argumento.

El árbol devuelto siempre tendrá nodos del tipo *tree('dato',void,void)* en la parte derecha de cada nodo superior.

1.5. Predicado ordenacion/3

```
ordenacion(Arbol,Comp,Orden) :-
 ordenacion_aux(Arbol,Comp,[],Orden).
```

Este predicado, al igual que *hojas_arbol/3*, llama a uno auxiliar (*ordenacion_aux/4*)con una lista vacía como argumento extra.

1.5.1. Predicado ordenacion_aux/4

```
ordenacion_aux(tree(Element,void,void),_,Orden,OrdenF) :-
 append(Orden,[Element],OrdenF).

ordenacion_aux(tree(Element,Left,Right),Comp,Orden,OrdenF) :-
 append(Orden,[Element],OrdenS),
 makeList(tree(Element,Left,Right),[],Lista),
 borraElemento(Element,Lista,Salida),
 lista_hojas(Salida,Hojas),
 hojas_arbol(Hojas,Comp,Arbol),
 ordenacion_aux(Arbol,Comp,OrdenS,OrdenF).
```

Su caso base ocurre al tener sólo un nodo hoja en el árbol de entrada. El criterio de comparación por tanto no importa, ya que siempre será el último elemento a insertar (o el único). Por tanto, añade llamando a *append/3* el dato del nodo al final de la lista *Orden*, usando como parámetro de salida *OrdenF*.

En otro caso, si hay más datos en el árbol, añade el primer dato a una lista *OrdenS*, luego crea una lista auxiliar *Lista* con los elementos restantes llamando a *makeList/3* y borra el elemento que acaba de insertar en *OrdenS* de *Lista*. Seguidamente crea un nuevo árbol a partir de *Lista* llamando a *lista_hojas/2* y a *hojas_arbol/3*. Por último, se llama recursivamente con *OrdenS* como nueva lista a la que añadir los elementos.

1.5.2. Predicado makeList/3

```
makeList(tree(Element,void,void), Lista, [Element|Lista]).

makeList(tree(_,Left,Right), Lista, ListaSalida) :-
 Right = tree(ElementRight,_,_),
 append(Lista,[ElementRight],Lista2),
 makeList(Left,Lista2,ListaSalida).
```

Su caso base se basa en un árbol de un único elemento y una lista cualquiera, devolviendo el elemento en la cabeza de la lista.

En otro caso, si hay varios elementos en el árbol, ignora el primer dato y extrae el elemento del nodo de la derecha, lo añade a una lista auxiliar *Lista2* y se llama recursivamente con el resto del árbol (parte izquierda), *Lista2* y la lista de salida en la que unificar en el caso base.

1.5.3. Predicado borraElemento/3

```
borraElemento(_,[],[]).

borraElemento(Elemento,[Elemento|T],T).

borraElemento(Elemento,[H|T1],[H|T2]) :-
 borraElemento(Elemento,T1,T2).
```

Este predicado tiene dos casos base:

- Uno, en el que tiene que "borrarün elemento de una lista vacía, en cuyo caso devuelve la misma lista vacía.
- Otro, en el que recibe un elemento que se encuentra en la cabeza de la lista, y devuelve la cola de la lista.

En otro caso (el elemento a borrar no se encuentra en la cabeza), se llama recursivamente con la cola de la lista, hasta que encuentre el elemento y lo elimine o ya no quede nada a borrar.

1.6. Predicado ordenar/3

```
ordenar(Lista,Comp,Orden) :-
 lista_hojas(Lista,Hojas),
 hojas_arbol(Hojas,Comp,Arbol),
 ordenacion(Arbol,Comp,Orden).
```

Este predicado llama a *lista_hojas/2*, *hojas_arbol/3* y *ordenacion/3* con los argumentos que se le pasan.

2. Pruebas realizadas

2.1. menor/2

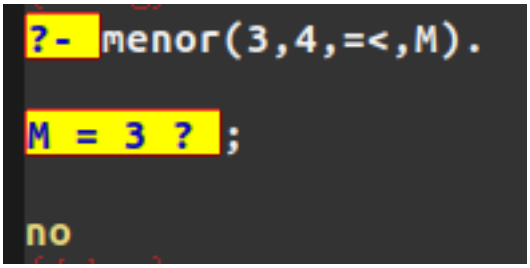

Figura 1: $3 \leq 4$

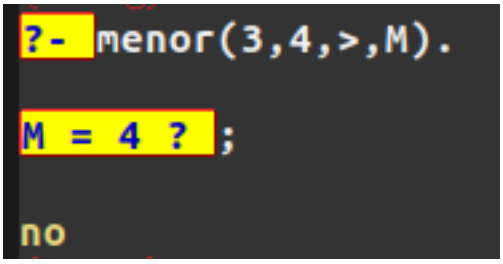

Figura 2: $3 > 4$

3. Comentarios adicionales

En mi opinión, considero que el enunciado de esta práctica no estaba correctamente redactado de manera que fuera de fácil comprensión. Considero que el haber necesitado dos ficheros de aclaraciones indica que hay algo a mejorar. Por otro lado, el enunciado de la primera práctica estaba mejor redactado y era más sencillo de comprender, con una explicación más clara de lo que se debía implementar y conseguir con cada predicado.

Ruego que se intente mejorar en la medida de lo posible para próximos años.