Programación Lógica y Funcional
Presentación del Curso

Ángel García Báez
ZS24019400@estudiantes.uv.mx

Maestría en Inteligencia Artificial

IIIA Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
Campus Sur, Calle Paseo Lote II, Sección 2a, No 112
Nuevo Xalapa, Xalapa, Ver., México 91097

17 de octubre de 2024

1. Algoritmo de unificación.

Programe en Prolog un algoritmo de unificación. A reportar: i) El algoritmo elegido comentado; ii) Su código, también comentado; iii) Los siguientes ejemplos de la ejecución **(50 puntos)**:

- $q(Y, g(a, b))$, $p(g(X, X), Y)$.
- $r(a, b, c)$, $r(X, Y, Z)$.
- mayor(padre(Y), Y), mayor(padre(Z), juan).
- conoce(padre(X), X), conoce(W, W).

A continuación se muestra el algoritmo implementado con ayuda de las diapositivas de Guerra-Hernández (2024) y el articulo original de Knight (1989) de donde se escogió el algoritmo de unificación de Robinson al ser el más accesible dentro de mis capacidades. Dicho algoritmo busca la discordancia entre términos y variables, los descompone a modo de comprobar que dichos términos puedan ser unificables, en caso de no serlo, sencillamente no lo hace y se detienen las búsquedas recursivas. En dicha versión del algoritmo se intento implementar una lista que guardara todas las unificaciones en caso de darse, pero lamentablemente no se pudo, en su lugar se dejo unificar/3, de forma que el tercer elemento sirva como una constante al termino de la unificación que indique si la cosa unifica satisfactoriamente o si por el contrario, en alguno de los casos falla la unificación y no devuelve más que un mensaje indicando dicho fallo.

```
1 %%%% 1.- UNIFICACION A MANO %%%%
2 % 1. Caso base: si los dos terminos son iguales, no
     hay mas sustituciones necesarias.
3 unificar(X, X, unifica) :- !.
4 % 2. Unificacion de variables: si el primer termino
     es una variable.
5 unificar(Var, Term, [(Var / Term)]) :-
6     var(Var), % Verificar que el primer termino es
        una variable
7     \+ Var == Term,  % La variable no debe ser
        identica al segundo termino
8     !. % Corta para que no haga backtracking si esta
         regla se cumple
```

```
% 3. Unificacion de variables: si el segundo termino es una variable.
unificar(Term, Var, [(Var / Term)]) :-
    var(Var), % Verifica que el segundo termino sea una variable
    \+ Var == Term, % Verifica que el segundo termino no es igual al primero
    !. % Evita que haga el backtracking si esta regla se cumple
% 4. Unificacion de terminos compuestos: descomponer y unificar recursivamente.
unificar(f(T1, T2), f(S1, S2), _) :-
    unificar(T1, S1, _), % Unificacion recursiva del primer componente.
    unificar(T2, S2, _). % Unificacion recursiva del segundo componente.
% 5. Unificacion falla si los terminos no pueden unificarse.
unificar(_, _, nounifica).
% unificar(q(Y, g(a, b)), p(g(X, X),Y ),L).
% unificar(r(a, b, c), r(X,Y, Z),L).
% unificar(mayor(padre(Y ),Y ), mayor(padre(Z), juan ),L).
% unificar(conoce(padre(X), X), conoce(W,W ),L).
```

A continuación se muestran los resultados de las consultas propuestas por el enunciado y añadí una al final para mostrar lo que ocurre cuando la unificación ocurre para un termino pero no para todos los presentes en la consulta:

```
?- unificar(q(Y, g(a, b)), p(g(X, X),Y ),L).
L = nounifica
?- unificar(r(a, b, c), r(X,Y, Z),L).
L = unifica,
X = a,
Y = b,
Z = c
?- unificar(mayor(padre(Y ),Y ), mayor(padre(Z), juan),L).
L = unifica,
Y = Z, Z = juan
```

```
?- unificar(conoce(padre(X), X), conoce(W,W ),L).
L = unifica,
W = X, X = padre(X)
?- unificar(f(X, b), f(a, e), R).
X = a
R = nounifica
```

Dentro de los resultados obtenidos, se muestra que la primer consulta no unifica, las siguientes 3 consultas si unifican completamente y que la consulta propuesta al final unifica solo 1 termino, pero al no poder unificar el otro termino, el algoritmo se detiene y devuelve que no es posible unificar toda la expresión. Podría omitirse el uso de la variable que reporta si la cosa unifica o no convirtiéndola en una variable anónima pero decidió dejarse por cuestiones de que sea didáctico, y me sirve para corroborar si la cosa logra su cometido satisfactoriamente.

2. Programa para pino/1.

Escriban un predicado pino/1 cuyo argumento es un entero positivo y su salida es como sigue **(10 puntos)**:

```
?- pino(5).
    *
   * *
  * * *
 * * * *
* * * * *
true.
```

A continuación se muestra el código resultante para poder crear el predicado pino/1 el cual al recibir un entero positivo da la salida de un arbolista bonito y centrado.

```
%%%% 2.- Dibujar el pino/1 %%%%
%%% Dibuja un pino de altura N %%%
pino(N) :-
    pino(N, 1).  % Llamada auxiliar con la fila
       inicial en 1.
% Caso base: Si la fila es mayor que N, termina.
pino(N, F) :-
    F > N, !.  % Corte para evitar backtracking.
% Imprime los espacios en blanco.
pino(N, F) :-
    F =< N,
    Espacios is N - F,  % Calcula el n mero de
       espacios en blanco.
    print_espacios(Espacios),  % Imprime los
       espacios.
    print_asteriscos(F),  % Imprime los asteriscos.
    nl,  % Nueva l nea.
    F1 is F + 1,  % Incrementa el contador de filas.
    pino(N, F1).  % Llama recursivamente para la
       siguiente fila.
% Predicado para imprimir los espacios.
print_espacios(0) :- !.  % Si no hay espacios, no
   imprime nada.

```

```
20 print_espacios(N) :-
21     write(' '),  % Imprime un espacio.
22     N1 is N - 1,
23     print_espacios(N1).  % Llama recursivamente.
24 % Predicado para imprimir los asteriscos.
25 print_asteriscos(0) :- !.  % Si no hay asteriscos,
       no imprime nada.
26 print_asteriscos(N) :-
27     write('*'),  % Imprime un asterisco.
28     write(' '),  % Imprime un espacio despu s del
          asterisco.
29     N1 is N - 1,
30     print_asteriscos(N1).  % Llama recursivamente.
```

Después de implementar el código, ocurría un error interesante y es que la lógica que había implementado era correcta dentro de lo que entendía factible para dibujar el árbol centrado, pero al estar programandolo en la versión en la nube de SWI-prolog, siempre me daba el pino pegado a la izquierda, es decir, omitía los espacios para que estuviera centrado. No fue hasta que probé el código en VSC cuando este mismo código, conseguía generar el pino satisfactoriamente como en el ejemplo del enunciado.

Por lo demás, el código funciona y consigue satisfacer adecuadamente lo solicitado en el enunciado.

3. Operaciones de conjuntos.

Sin usar sus definiciones predefinidas, implemente las siguientes operaciones sobre conjuntos representados como listas **(20 puntos)**:

- Subconjunto:

```
?- subset([1,3],[1,2,3,4]).
true.
?- subset([],[1,2]).
true.
```

- Intersección:

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

- Unión:

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

- Diferencia:

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

A continuación se muestran las definiciones realizadas de las implementaciones de las operaciones de conjuntos, con ayuda de las encontradas en el libro Clocksin (2003), las cuales quedan del siguiente modo:

```
%%%% 3.- CONJUNTOS %%%%
%%% Miembro %%%%
% Es necesario definir miembro para reutilizarlo en las siguientes operaciones
miembro(X,[X|_]). % Caso base, X es miembro de la lista de X pegada con la lista vacia
miembro(X,[_|Y]):- miembro(X,Y). % Verificar si X no esta en la cabeza, buscarlo en el resto de la lista
```

```
%%% Subconjunto %%%
% Definir que una lista es un subconjunto de otra
subconjunto([],_). % la lista vacia es un subconjunto de la lista Y o _.
subconjunto([A|X],Y):- %Verificar si en la lista [A|X] es subconjunto de Y
    miembro(A,Y), % Verifica si la cabeza de la lista es miembro de la lista Y
    subconjunto(X,Y). % Caso recursivo hasta llegar al caso base
% Esta funci n solo dice si algo es subconjunto de otro si o no.
%%% Intersecci n %%%
% Caso base
interseccion([],_,[] ). %La intersecci n entre la lista vacia y un conjunto X es la lista vacia
interseccion([X|R],Y,[X|Z]):- % Si el primer elemento de la forma parte de Y, entonces X forma parte de la intersecci n
    miembro(X,Y), % Verifica que X sea miembro en Y
    !, % Si no es miembro, para evitar el backtrackig
    interseccion(R,Y,Z). % Si es miembro aplica la llamada recursiva
interseccion([_|R],Y,Z):- interseccion(R,Y,Z). % Llamada recursiva al resto de la lista hasta agotarla
%%% Union %%%
% Caso base
union([],X,X). % La union de la lista vacia con X es X
% Condicion de recursividad
union([X|R],Y,Z):-  % Si la cabeza de la lista es miembro de Y, no se a ade a la lista para evitar que se dupliquen.
    miembro(X,Y), % Verifica que X sea miembro de la lista Y
    !, % Si es cierto, no hace backtracking,
    union(R,Y,Z). % Si lo es, dispara la recursividad sobre el resto de la lista
```

```
29 union([X|R],Y,[X|Z]):- union(R,Y,Z). % Si el primero
       no es miembro, continua buscando sobre el resto
   de la lista R.
30 %%% Diferencia %%%
31 % Caso base: La diferencia de una lista vac a con
   cualquier lista es una lista vac a.
32 diferencia([], _, []).
33 % Caso recursivo: Si el elemento X de la primera
   lista no est  en la segunda lista,
34 % se incluye en el resultado de la diferencia.
35 diferencia([X | R], Y, [X | Z]) :-
36     \+ miembro(X, Y),  % Verifica que X no sea
          miembro de Y.
37     diferencia(R, Y, Z).  % Llama recursivamente
          para el resto de la lista.
38 % Caso recursivo: Si el elemento X est  en la
   segunda lista, se ignora y se contin a.
39 diferencia([X | R], Y, Z) :-
40     miembro(X, Y),  % Verifica que X sea miembro de
          Y.
41     diferencia(R, Y, Z).  % Llama recursivamente
          para el resto de la lista.
42 % Consultas
43 %subconjunto([1,3],[1,2,3,4]).
44 %subconjunto([],[1,2]).
45 %interseccion([1,2,3],[2,3,4],L).
46 %union([1,2,3,4],[2,3,4,5],L).
47 %diferencia([1,2,3,4],[2,3,4,5],L).
48 %diferencia([1,2,3],[1,4,5],L).
```

Se probaron todas las metas que sugiere el enunciado sobre las definiciones propuestas anteriormente y al realizar las querys, el resultado fue que todas las querys (con los nombres un poco cambiados para los enunciados) llegan a los mismos resultados de forma satisfactoria.

4. Permutaciones.

Escriba un programa que regrese en su segundo argumento la lista de todas las permutaciones de la lista que es su primer argumento. Esto sin usar permutation/2 definida por Prolog. (10 puntos) Por ejemplo:

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

Para poder implementar dicho algoritmo, se tomo de base el algoritmo encontrado en el libro de Clocksin (2003) al cual se le hizo una pequeña modificación para poder obtener todas las permutaciones en una lista de listas a traves de usar findall. La implementación es la siguiente:

```
%%%% 4.- Permutaci n de listas %%%%
% Caso base: La permutaci n de una lista vac a es
   la lista vac a.
permutacion([], []).
% Caso recursivo: Selecciona un elemento y permuta
   el resto.
permutacion(L, [H | T]) :-
    append(V, [H | U], L),  % Descompone la lista L
       en cabeza H y cola U.
    append(V, U, Resto),    % Crea la lista con el
       resto sin la cabeza H
    permutacion(Resto, T).  % Llamada recursiva
       sobre el resto.
% Sacar todas las permutaciones en una sola lista
   usando la definici n de permutaci n
permutaciones(L, Permutaciones) :-
    findall(P, permutacion(L, P), Permutaciones).
% consulta
% permutaciones([1, 2, 3], L).
```

Al correr la consulta se logra llegar satisfactoriamente al mismo resultado planteado en el enunciado tal que:

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

5. Números de Peano.

Escriba un predicado que convierta números naturales de Peano (Recuerden: s(s(s(0))) = 3) a su equivalente decimal. Posteriormente implemente la suma y la resta entre dos números de Peano (10 puntos). Por ejemplo:

```
?- peanoToNat(s(s(s(0))),N).
N = 3.
?- peanoToNat(0,N).
N = 0.
?- sumaPeano(s(s(0)),s(0),R).
R = s(s(s(0))).
?- restaPeano(s(s(0)),s(0),R).
R = s(0).
```

A continuación se muestra el programa escrito en prolog para implementar números naturales de Peano a su equivalencia decimal, la suma y la resta entre estos elementos, para implementarlo fue necesario estudiar primero que son los numeros de Peano y para ello me base en el articulo de Antezana Iparraguirre (2019) donde da una explicación sobre los axiomas y el como se usan:

```
%%%%% 5.- NUMEROS DE PEANO %%%%
% 1. Conversi n de un n mero en notaci n de Peano
    a su equivalente decimal.
% Caso base: el equivalente decimal de 0 es 0.
peanoToNat(0, 0).
% Caso recursivo: para s(P), el valor es N + 1.
peanoToNat(s(P), N) :-
    peanoToNat(P, N1),
    N is N1 + 1.
% 2. Suma de dos n meros de Peano.
% Caso base: sumar 0 con cualquier n mero da ese
   mismo n mero.
sumaPeano(0, Y, Y).
% Caso recursivo: sumar s(X) con Y es equivalente a
   s(X + Y).
sumaPeano(s(X), Y, s(R)) :-
    sumaPeano(X, Y, R).
% 3. Resta de dos n meros de Peano.
% Caso base: restar 0 de cualquier n mero da ese
```

```
   mismo n mero.
restaPeano(X, 0, X).

% Caso recursivo: restar s(Y) de s(X) es equivalente
    a restar Y de X.
restaPeano(s(X), s(Y), R) :-
    restaPeano(X, Y, R).
% Metas que se computaron:
%peanoToNat(s(s(s(0))),N).
%N = 3.
%peanoToNat(0,N).
%N = 0.
%sumaPeano(s(s(0)),s(0),R).
%R = s(s(s(0))).
%restaPeano(s(s(0)),s(0),R).
%R = s(0).
```

Tras la implementación del código, se computaron las metas propuestas por el enunciado y se llegaron satisfactoriamente a los mismos resultados. Se incluye el código anexo con el documento.

Referencias

Antezana Iparraguirre, R. P. (2019). Demostración de teoremas de números naturales en el sistema axiomático de giuseppe peano. *Horizonte de la Ciencia*, 9(16).

Clocksin, William F. & Mellish, C. S. (2003). *Programming in PROLOG*. Springer Science & Business Media.

Guerra-Hernández, D. A. (2024). Programación Lógica y Funcional - Programación Lógica.

Knight, K. (1989). Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124.